## Contents

*About This Book* ................................................................. ix

*What's New in SAS 9.4 DS2 Language Reference* ......................... xi

### PART 1  Introduction  1

**Chapter 1 • Introduction to the DS2 Language**  3
- Introduction to the DS2 Language ........................................ 3
- Running DS2 Programs .................................................. 4
- Supported Data Sources ................................................ 4
- Intended Audience ....................................................... 5
- When to Use DS2 ........................................................ 5

### PART 2  Getting Started  7

**Chapter 2 • DS2 and the DATA Step**  9
- What Is DS2? ............................................................ 9
- Similarities between DS2 and the DATA Step ......................... 9
- Differences between DS2 and the DATA Step ......................... 10

**Chapter 3 • Learning by Example: Using the Sample Programs**  15
- About the Getting Started Sample Programs ......................... 15
- Your First Sample Programs ........................................... 17
- See Also ................................................................. 25

**Chapter 4 • Building Blocks of DS2 Programs**  27
- Basic DS2 Language Concepts ........................................ 27
- What Is a DS2 Program? .............................................. 32
- Example: Block Scope .................................................. 32
- Example: A Simple Thread Program .................................. 34

**Chapter 5 • Understanding DS2 Methods and Packages**  37
- Modularity, Encapsulation, and Abstraction in DS2 ................ 37
- Getting Started with DS2 Methods ................................... 38
- Getting Started with DS2 Packages ................................... 40
- Example: Introduction to DS2 Methods .............................. 42
- Example: Introduction to DS2 Packages .............................. 44

### PART 3  DS2 Concepts  47

**Chapter 6 • DS2 Programming Semantics**  49
- Basic DS2 Program Syntax ............................................ 49
- Basic DS2 Program Semantics ........................................ 50
### Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to DS2 Variables</td>
<td>57</td>
</tr>
<tr>
<td>2</td>
<td>Variable Declaration</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>Variable Lists</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>Variable Scope</td>
<td>62</td>
</tr>
<tr>
<td>5</td>
<td>Predefined DS2 Variables</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>Definition of a Constant</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>Numeric Constants</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>Character Constants</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>Binary Constants</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>Date and Time Constants</td>
<td>70</td>
</tr>
<tr>
<td>11</td>
<td>Constant List</td>
<td>71</td>
</tr>
<tr>
<td>12</td>
<td>What Are the Data Types?</td>
<td>72</td>
</tr>
<tr>
<td>13</td>
<td>Define Data Types for a Column</td>
<td>74</td>
</tr>
<tr>
<td>14</td>
<td>Error Messages That Use the DOUBLE and REAL Data Types</td>
<td>75</td>
</tr>
<tr>
<td>15</td>
<td>Overview of Identifiers</td>
<td>77</td>
</tr>
<tr>
<td>16</td>
<td>Regular Identifiers</td>
<td>77</td>
</tr>
<tr>
<td>17</td>
<td>Delimited Identifiers</td>
<td>78</td>
</tr>
<tr>
<td>18</td>
<td>Referencing a Macro Variable in a Delimited Identifier</td>
<td>80</td>
</tr>
<tr>
<td>19</td>
<td>Support for Non-Latin Characters</td>
<td>80</td>
</tr>
<tr>
<td>20</td>
<td>DS2 Modes for Nonexistent Data</td>
<td>81</td>
</tr>
<tr>
<td>21</td>
<td>Reading and Writing Nonexistent Data in ANSI Mode</td>
<td>84</td>
</tr>
<tr>
<td>22</td>
<td>Reading and Writing Nonexistent Data in SAS Mode</td>
<td>85</td>
</tr>
<tr>
<td>23</td>
<td>Testing for Null Values</td>
<td>86</td>
</tr>
<tr>
<td>24</td>
<td>Testing for Missing Values</td>
<td>86</td>
</tr>
<tr>
<td>25</td>
<td>Type Conversion Definitions</td>
<td>87</td>
</tr>
<tr>
<td>26</td>
<td>Overview of Type Conversions</td>
<td>88</td>
</tr>
<tr>
<td>27</td>
<td>Type Conversion for Unary Expressions</td>
<td>88</td>
</tr>
<tr>
<td>28</td>
<td>Type Conversion for Logical Expressions</td>
<td>89</td>
</tr>
<tr>
<td>29</td>
<td>Type Conversion for Arithmetic Expressions</td>
<td>89</td>
</tr>
<tr>
<td>30</td>
<td>Type Conversion for Relational Expressions</td>
<td>90</td>
</tr>
<tr>
<td>31</td>
<td>Type Conversion for Concatenation Expressions</td>
<td>91</td>
</tr>
<tr>
<td>32</td>
<td>What Is an Expression?</td>
<td>93</td>
</tr>
<tr>
<td>33</td>
<td>Types of Expressions</td>
<td>94</td>
</tr>
<tr>
<td>34</td>
<td>Operators in Expressions</td>
<td>106</td>
</tr>
<tr>
<td>35</td>
<td>DS2 Dates, Times, and Timestamps</td>
<td>111</td>
</tr>
<tr>
<td>36</td>
<td>SAS Date, Time, and Datetime Values</td>
<td>113</td>
</tr>
</tbody>
</table>
Chapter 15 • DS2 Arrays ........................................................................................................... 121
Overview of DS2 Arrays ......................................................................................................... 121
Temporary Arrays .................................................................................................................... 122
Variable Arrays ....................................................................................................................... 123
Declaring Arrays with a HAVING Clause .................................................................................. 126
Array Assignment ..................................................................................................................... 128
Array Arguments ..................................................................................................................... 131
How to Query Array Dimensions ........................................................................................... 133
How to Output Array Content ................................................................................................. 134

Chapter 16 • DS2 Packages .................................................................................................... 135
Introduction to DS2 Packages .................................................................................................. 135
Packages and Scope ................................................................................................................ 137
Dot Operator in Packages ........................................................................................................ 142
Package Constructors and Destructors ..................................................................................... 142
User-Defined Packages ............................................................................................................ 143
Predefined DS2 Packages ........................................................................................................ 143

Chapter 17 • Threaded Processing .......................................................................................... 179
Overview of Threaded Processing ............................................................................................. 179
Threaded Processing and the SAS In-Database Code Accelerator ........................................... 180

Chapter 18 • Using DS2 and FedSQL ...................................................................................... 183
Dynamically Executing FedSQL Statements from DS2 ............................................................ 183

Chapter 19 • DS2 Input and Output ........................................................................................ 185
Overview of DS2 Input and Output .......................................................................................... 185
Reading Data Using the SET Statement .................................................................................. 186
Reading Data Using the Hash Package .................................................................................... 189
Reading and Writing Data Using the SQLSTMT Package and the
SQLEXEC Function .................................................................................................................... 189
Writing Data Using the OUTPUT Statement .......................................................................... 189
Column Order in Output Tables When Using Data Sources Outside SAS .............................. 190
NLS Transcoding Failures ........................................................................................................ 190

Chapter 20 • Combining Tables .............................................................................................. 191
Definitions for Combining Data ............................................................................................... 191
Combining Tables: Basic Concepts ......................................................................................... 191
Combining DS2 Tables: Methods ............................................................................................ 200

Chapter 21 • DS2 Reserved Words .......................................................................................... 219
Reserved Words in the DS2 Language ...................................................................................... 219

Chapter 22 • DS2 Formats ...................................................................................................... 225
Overview of Formats ............................................................................................................... 227
Chapter 31 • DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements ......................................................... 1003
  Dictionary ........................................................................ 1004
<table>
<thead>
<tr>
<th>Example</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL in a DS2 Program</td>
<td>1308</td>
</tr>
<tr>
<td>Make Two New Tables Based on a Condition</td>
<td>1310</td>
</tr>
<tr>
<td>Change Case of Text Output</td>
<td>1312</td>
</tr>
<tr>
<td>Scope</td>
<td>1313</td>
</tr>
<tr>
<td>Functions</td>
<td>1314</td>
</tr>
<tr>
<td>Arrays</td>
<td>1315</td>
</tr>
<tr>
<td>SELECT Statement</td>
<td>1319</td>
</tr>
<tr>
<td>GOTO and LEAVE Statements with Labels</td>
<td>1320</td>
</tr>
<tr>
<td>Overloaded Methods</td>
<td>1323</td>
</tr>
<tr>
<td>Analyze a Table Using Multiple Threads</td>
<td>1324</td>
</tr>
<tr>
<td>Using Four Threads to Compute Summary Statistics</td>
<td>1326</td>
</tr>
<tr>
<td>Data Cleaning</td>
<td>1328</td>
</tr>
<tr>
<td>SUBSTR Function</td>
<td>1329</td>
</tr>
<tr>
<td>PUT with SAS Formats</td>
<td>1330</td>
</tr>
<tr>
<td>Generate Statistics from Table Data</td>
<td>1331</td>
</tr>
<tr>
<td>Matrices and Non-linear Equations</td>
<td>1333</td>
</tr>
<tr>
<td>DS2 Matrices and Regression Analysis</td>
<td>1336</td>
</tr>
<tr>
<td>Use the SQLSTMT Package in Another Package</td>
<td>1339</td>
</tr>
<tr>
<td>Update the Values of a Table By Using Two Databases</td>
<td>1340</td>
</tr>
<tr>
<td>Store FedSQL Statements in a Hash Package</td>
<td>1341</td>
</tr>
<tr>
<td>Run a DS2 Program from z/OS Batch</td>
<td>1346</td>
</tr>
</tbody>
</table>

**Recommended Reading** | 1347

**Index** | 1349
About This Book

Syntax Conventions for the DS2 Language

**Typographical Conventions**

Type styles have special meanings when used in the documentation of the DS2 language syntax.

**UPPERCASE BOLD** identifies DS2 keywords such the names of statements and functions (for example, PUT).

**UPPERCASE ROMAN** identifies arguments and values that are literals (for example, FROM).

*italic* identifies arguments or values that you supply. Items in italic can represent user-supplied values that are either one of the following.

- nonliteral values assigned to an argument (for example, ALTER=alter-password).
- nonliteral arguments (for example, KEEP=(column-list)).

If more than one of an item in italics can be used, the items are expressed as item [, …item].

**monospace** identifies examples of SAS code.

**Syntax Conventions**

*SAS DS2 Language Reference* uses the Backus-Naur Form (BNF), specifically the same syntax notation used by Jim Melton in *SQL:1999 Understanding Relational Language Components*.

The main difference between traditional SAS syntax and the syntax that is used in the DS2 language reference documentation is in how optional syntax arguments are displayed. In traditional SAS syntax, angle brackets (< >) are used to denote optional syntax. In DS2 language syntax, square brackets ([ ]) are used to denote optional syntax and angle brackets are used to denote non-terminal components.

The following symbols are used in the DS2 language syntax.

::=  This symbol can be interpreted as “consists of” or “is defined as”.

Angle brackets identify a non-terminal component (that is, a syntax component that can be further resolved into lower level syntax grammar).

Square brackets identify optional arguments. Any argument that is not enclosed in square brackets is a required argument. Do not enter square brackets unless they are preceded by a backward slash (\), which denotes that they are literal.

Braces provide a method to distinguish required multi-word arguments. Do not enter braces unless they are preceded by a backward slash (\), which denotes that they are literal.

A vertical bar indicates that you can choose one value from a group. Values that are separated by bars are mutually exclusive.

An ellipsis indicates that the argument or group of arguments that follow the ellipsis can be repeated any number of times. If the ellipsis and the following arguments are enclosed in square brackets, they are optional.

A backward slash indicates that the next character is a literal.

The following examples illustrate the syntax conventions that are described in this section. These examples contain selected syntax elements, not the complete syntax.

```
SET <table-reference> [... [<table-reference>] [INDSNAME=variable];
[BY [DESCENDING] column [...[DESCENDING] column];
<table-reference>::= {table (table-options)} | sql-text
```

1. SET is in uppercase bold because it is the name of the statement.
2. <table-reference> is in angle brackets because it is a non-terminal argument that is further resolved into lower level syntax grammar. You must supply at least one <table-reference>.
3. BY and DESCENDING are in uppercase roman because they are literal arguments. DESCENDING is in square brackets because it is an optional argument.
4. column is in italics because it is an argument that you can supply.
5. The square brackets and ellipsis around the second instance of column indicate that you can repeat this argument any number of times as long as the arguments are separated by commas.
6. The <table-reference>::= non-terminal argument syntax is read as follows: A <table-reference> consists of a table name and table options or embedded SQL text.
7. The vertical bar (|) indicates you can supply either table [table-options] or sql-text, but not both.
8. The backslash (\) before the braces around sql-text indicate that those braces are literals and must be entered.
Overview

DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

The third maintenance release for SAS 9.4 has the following changes and enhancements:

• The DS2 language now supports the MERGE statement, which enables you to match-merge table data.

• The SAS In-Database Code Accelerator now supports a SET statement with embedded SQL, a SET statement that specifies multiple input tables, and the new MERGE statement.

The SAS In-Database Code Accelerator for Hadoop now supports the SPDE HDFS file format.

• Two new predefined DS2 packages, JSON and TZ, enable you to create and parse JSON text and to perform time zone processing, respectively.

Three new SQLSTMT package methods enable you to retrieve the number of columns, the name of a column by column number, and the type of column by column number.

New constructor syntax and two new methods enable you to create an SQLSTMT package instance and prepare it at a later time.

• Three new functions, CMP, CMPT, and FMTINFO, enable you to compare two character strings including and excluding trailing blanks and to return information about a SAS format, respectively. Two other new functions, SHA256HEX and SHA256MACHEX, convert a string to a 256-bit hash value based on the SHA256 algorithm and the Hash-based Message Authentication (HMAC) algorithm, respectively.

• The SELECT statement in embedded SQL text now supports the ORDER BY, INDSNUM, and WHERE clauses.

The February 2015 release of the SAS In-Database Code Accelerator for Hadoop uses HCatalog to process complex, non-delimited files. This enables the SAS In-Database Code Accelerator for Hadoop to support Avro, ORC, RCFile, and Parquet file types. In addition, you can now use the DBCREATE_TABLE_OPTS table option to specify the output SerDe, the output delimiter of the Hive table, the output escaped by, and any other CREATE TABLE syntax that is allowed by Hive.
The second maintenance release for SAS 9.4 has the following changes and enhancements:

- The SAS In-Database Code Accelerator for Hadoop is available.
- Nine new functions have been added.
- A new HTTP package is available for creating an HTTP client to access web services.
- A connection string parameter has been added to the SQLSTMT package.

The first maintenance release for SAS 9.4 has the following changes and enhancements:

- The default behavior for the SAS In-Database Code Accelerator has changed.
- Changes have been made to DS2 functions.
- The SAS In-Database Code Accelerator for Teradata now runs the DS2 data program as well as the thread program inside the database.
- New DS2 configuration and run-time loggers have been added to the SAS Logging Facility. You can now use formatted log messages.
- Several general enhancements have also been added.

---

**Overview of the DS2 Language**

DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

Several DS2 language elements accept embedded FedSQL syntax, and the run-time-generated queries can exchange data interactively between DS2 and any supported database. This action enables SQL preprocessing of input tables, which effectively combines the power of the two languages.

The DS2 procedure enables you to submit DS2 language statements from a Base SAS session. The procedure enables the requests to be processed by the DS2 data access technology that supports a scalable, threaded, high-performance, and standards-based way to access, manage, and share relational data. For more information about PROC DS2, see Base SAS Procedures Guide.

---

**SAS In-Database Code Accelerator**

The third maintenance release for SAS 9.4 has the following changes and enhancements:

- A SET statement with embedded SQL and a SET statement that specifies multiple input tables are now supported.

  *Note*: When using the SAS In-Database Code Accelerator for Hadoop, SET statements with multiple input tables requires Hive 1.3 or later.

- The new DS2 MERGE statement is supported.
• The SAS In-Database Code Accelerator for Hadoop supports reading and writing of HDFS-SPD Engine file formats.

• In addition, if a Hadoop data or thread program fails and MSGLEVEL=I is set, a message is written to the SAS log that contains a link to the MapReduce job log where you can find the error messages.

In the February 2015 release of the SAS In-Database Code Accelerator for Hadoop, the following changes and additions were made:

• The SAS In-Database Code Accelerator for Hadoop supports only Cloudera 5.2 and Hortonworks 2.1 or later.

• The SAS In-Database Code Accelerator for Hadoop uses HCatalog to process complex, non-delimited files.

• The SAS In-Database Code Accelerator for Hadoop now supports Avro, ORC, RCFile, and Parquet file types.

• For the SAS In-Database Code Accelerator for Hadoop, you can use the DBCREATE_TABLE_OPTS table option to specify the output SerDe, the output delimiter of the Hive table, the output escaped by, and any other CREATE TABLE syntax allowed by Hive.

In the second maintenance release for SAS 9.4, in-database processing for Hadoop has been enhanced by the addition of the SAS In-Database Code Accelerator for Hadoop. The SAS In-Database Code Accelerator for Hadoop runs the DS2 data program as well as the thread program inside the database.

In the first maintenance release for SAS 9.4, the following enhancements and changes have been made to the In-Database Code Accelerator:

• A new system option, DS2ACCEL controls whether the DS2 code is executed inside the database. The default value is NONE, which prevents DS2 code from executing inside the database.

• The PROC DS2 INDB option has changed its name to DS2ACCEL. The INDB option is still supported. However, the default value for this option has changed from YES to NO, which prevents DS2 code from executing in the database. This is a change in behavior from the 9.4 release.

• The SAS In-Database Code Accelerator for Teradata now runs the DS2 data program as well as the thread program inside the database.

---

**DS2Functions**

In the third maintenance release for SAS 9.4, the following enhancements have been made:

• Two new functions, CMP and CMPT, enable you to compare two character strings including and excluding trailing blanks, respectively.

• The FMTINFO function returns information about a SAS format.

• The SHA256HEX and SHA256MACHEX functions convert a string to a 256-bit hash value based on the SHA256 algorithm and the Hash-based Message Authentication (HMAC) algorithm, respectively.

In the second maintenance release for SAS 9.4, the following enhancements have been made:
Four new Perl regular expression (PRX) functions are available:

- The PRXCHANGE function performs a pattern-matching replacement.
- The PRXPAREN function returns the last bracket match for which there is a match in a pattern.
- The PRXPARSE function compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.
- The PRXPOSN function returns a character string that contains the value for a capture buffer.

Five new DBCS functions are available:

- The KCOUNT function returns the number of double-byte characters in an expression.
- The KSTRCAT function concatenates two or more character expressions.
- The KSTRIP function removes leading and trailing blanks from a character string.
- The KUDPATE function inserts, deletes, and replaces character value contents.
- The KUPDATES function inserts, deletes, and replaces the contents of the character value according to the byte position of the character value in the argument.

In the first maintenance release for SAS 9.4, the following enhancements have been made:

- The MISSING function also now supports all data types and package parameters.
- The new UUIDGEN function returns the short form of a Universally Unique Identifier (UUID).
- The MD5 function now supports Unicode character strings.

Predefined DS2 Packages

**DS2 TZ Package**

A new DS2 package, TZ, enables you to perform time zone processing on date and time data.

**DS2 HTTP Package**

In the second maintenance release of 9.4, a new predefined HTTP package is available. The HTTP package enables you to construct an HTTP client to access web services.

A new logger, App.TableServices.d2pkg.HTTP, is available that supports logging of HTTP traffic through the SAS logging facility.

**DS2 JSON Package**

A new DS2 package, JSON, enables you to create and parse JSON text.
DS2 Loggers and Logger Packages

In the first maintenance release for SAS 9.4, five new loggers are available:

App.TableServices.DS2.Config.Options
shows the options that are supplied to the DS2 compiler.

App.TableServices.DS2.Config.Source
shows the DS2 source code that is processed by the DS2 compiler.

App.TableServices.DS2.Config.Version
shows version information for all threaded kernel extensions that are loaded by the
DS2 compiler.

App.TableServices.DS2.Runtime.Calls
shows a trace of all method calls during execution.

App.TableServices.DS2.Runtime.SQL
shows all SQL statements that are either prepared by the DS2 compiler, executed by
the DS2 compiler, or both.

The following enhancements have been made to the LOG method:

• The maximum length of a message is now 65535 characters.

• Two new arguments have been added to specify formatted messages.

DS2 SQLSTMT Package

In the third maintenance release for SAS 9.4, three new methods are available:

• GETCOLUMNCOUNT returns the number of columns in the result set.

• GETCOLUMNNAME returns the column name of the result set column with the
designated index.

• GETCOLUMNMYPNAME returns the type name of the result set column with the
designated index.

In the second maintenance release for SAS 9.4, a connection string parameter is
available when declaring and instantiating an SQLSTMT package.

General Enhancements

The third maintenance release for SAS 9.4 has the following changes and enhancements:

• The DS2 language now supports the MERGE statement, which enables you to
match-merge table data.

• A new statement, DS2_OPTIONS, can specify or change the default behavior of a
DS2 program:
  • how DS2 processes a division by zero operation
  • write a note instead of an error message to the SAS log when an invalid function
    argument generates a missing value
  • non-existent values are processed as ANSI SQL null values
• create a trace of what statements are executed

• The SELECT statement in embedded SQL text now supports the PARTITION BY, ORDER BY, INDSNUM, and WHERE clauses.

• A new format, BESTDOTX, produces a US-locale-based value regardless of current locale.

• Partitioned tables using the DBCREATE_TABLE_OPTS table option are now supported.

In the second maintenance release for SAS 9.4, the Getting Started section has been rewritten and contains new examples.

In the first maintenance release for SAS 9.4, the following enhancements were made:

• If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.

• You can access any FCMP library as long as the connection string defines the catalog in which the FCMP library is located.

• The data type and character set encoding for an undeclared variable on the left side of an assignment statement is determined by the data type and character set encoding of the value on the right side of the assignment statement.

• The MDYAMPM format is now supported.
Part 1

Introduction

Chapter 1

Introduction to the DS2 Language ........................................ 3
Chapter 1
Introduction to the DS2 Language

Introduction to the DS2 Language

DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages. Several DS2 language elements accept embedded FedSQL syntax, and the runtime-generated queries can exchange data interactively between DS2 and any supported database. This allows SQL preprocessing of input tables, which effectively combines the power of the two languages.

In addition, DATA step logic can be transformed to run in environments where DS2 is supported and the DATA step is not. These environments include the following:

- SAS Federation Server
- SAS LASR Analytic Server
- SAS High-Performance Analytics
- SAS Embedded Process
- SAS Enterprise Miner
- SAS Decision Services

The DS2 procedure enables you to submit DS2 language statements from a Base SAS session. For more information about PROC DS2, see Base SAS Procedures Guide. A high-performance version of the DS2 procedure, PROC HPDS2, submits DS2 language statements for execution to either a single machine running multiple threads or to a distributed computing environment, including the SAS LASR Analytic Server. For more information about PROC HPDS2, see Base SAS Procedures Guide: High-Performance Procedures.
Because the DS2 language can be used with many data sources, the terms row, column, and table are used to describe the data elements. Comparing this to SAS DATA step terminology, a row corresponds to an observation, a column corresponds to a variable, and a table corresponds to a data set.

Running DS2 Programs

You can submit DS2 programs in one of the following ways.

• Through the Base SAS language interface using the DS2 procedure. A single PROC DS2 step can contain several DS2 programs.

  For more information, see “DS2” in Base SAS Procedures Guide.

• Directly to a data source using the SAS In-Database Code Accelerator.

  For more information about using the SAS In-Database Code Accelerator, see SAS In-Database Products: User’s Guide.

• Directly to the SAS Federation Server using the SAS LIBNAME engine for SAS Federation Server.

• Using the HPDS2 procedure from the SAS client to submit DS2 language statements for execution to either a single machine running multiple threads or to a distributed computing environment, including the SAS LASR Analytic Server.

  For more information, see Base SAS Procedures Guide: High-Performance Procedures.

Note: These execution methods require Base SAS. Some might require additional software licenses. For example, accessing any relational database management system (RDBMS) from Base SAS requires the appropriate SAS/ACCESS software license.

Supported Data Sources

DS2 can access the following data sources:

• Aster
• DB2 for UNIX and PC operating environments
• Greenplum
• Hadoop (Hive and HDMD)
• Memory Data Store (MDS)
• MySQL
• Netezza
• ODBC-compliant databases (such as Microsoft SQL Server)
• Oracle
• PostgreSQL
• SAP (Read-only)
Intended Audience

The information in this document is intended for the following users who perform in these roles:

- **Application developers** who write the client applications. They write applications that create tables, bulk load tables, manipulate tables, and query data.

- **Database administrators** who design and implement the client/server environment. They administer the data by designing the databases and setting up the data source metadata. That is, database administrators build the data model.

- **SAS programmers** who want or need to take advantage of the features of the DS2 language.

When to Use DS2

You do not necessarily have to convert your DATA step programs to DS2. Typically, DS2 programs are written for applications that carry out the following actions:

- require the precision that results from using the new supported data types

- benefit from using the new expressions or write methods or packages available in the DS2 syntax

- need to execute SAS FedSQL from within the DS2 program

- execute outside a SAS session, for example, on High-Performance Analytics Server or the SAS Federation Server

- take advantage of threaded processing in products such as the SAS In-Database Code Accelerator, SAS High-Performance Analytics Server, and SAS Enterprise Miner
Part 2

Getting Started

Chapter 2  
DS2 and the DATA Step .................................................. 9

Chapter 3  
Learning by Example: Using the Sample Programs ............... 15

Chapter 4  
Building Blocks of DS2 Programs ................................... 27

Chapter 5  
Understanding DS2 Methods and Packages ....................... 37
Chapter 2
DS2 and the DATA Step

What Is DS2?

DS2 is a SAS proprietary programming language that is used for data manipulation and data modeling applications. The DS2 language shares core features with the DATA step. However, DS2 capabilities extend far beyond those of the DATA step.

DS2 is a procedural language that has variables and scope, methods, packages, control flow statements, table I/O statements, and parallel programming statements. Methods and packages give DS2 modularity and data encapsulation. DS2 enables you to insert SQL directly into the SET statement, thus blending the power of two powerful data manipulation languages.

Similarities between DS2 and the DATA Step

DS2 and the DATA step share many language elements, and those elements behave in the same way:

- SAS formats.
- SAS functions.
- SAS statements such as DATA, SET, KEEP, DROP, RUN, BY, RETAIN, PUT, OUTPUT, DO, IF-THEN/ELSE, Sum, and others.
- DATA step keywords are included in the list of DS2 keywords.

You can perform most DATA step tasks using DS2:

- process variable arrays, multi-dimensional arrays, and hash tables
- convert between data types
- work with expressions
- calculate date and time values
• process missing values

*Note:* All supported DS2 syntax is covered in this document. Any syntax appearing in other SAS documentation is not part of the supported DS2 syntax unless it is also documented here.

## Differences between DS2 and the DATA Step

### Table 2.1  Comparison by Topic

<table>
<thead>
<tr>
<th>Topic</th>
<th>DATA Step</th>
<th>DS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming paradigm</td>
<td>Executable code resides in the DATA step and PROC step.</td>
<td>Executable code resides in methods.</td>
</tr>
<tr>
<td>Scope</td>
<td>No concept of scope. All variables are global.</td>
<td>Variables that are declared in a method have local scope. All other identifiers have global scope.</td>
</tr>
<tr>
<td></td>
<td>Variables are created by assignment. The data type of a variable is determined by the context of how it is first used. All variables have global scope.</td>
<td>There are three types of global scope:</td>
</tr>
<tr>
<td></td>
<td>Variables are also defined when the SET statement is used.</td>
<td>• data program</td>
</tr>
<tr>
<td></td>
<td>Variables are not explicitly declared. Variables are created by assignment. The data type of a variable is determined by the context of how it is first used. All variables have global scope. Variables are also defined when the SET statement is used.</td>
<td>• thread program</td>
</tr>
<tr>
<td></td>
<td>Variables are declared using the DECLARE statement, which also determines the data type and scope attributes of the variable.</td>
<td>• package</td>
</tr>
<tr>
<td></td>
<td>Variables can be declared by assignment, but a best practice is to enforce variable declaration strict mode by setting the system option DS2SCOND=ERROR or the PROC DS2 option SCOND=ERROR.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Global variables are also defined when the SET or SET FROM statement is used.</td>
<td></td>
</tr>
<tr>
<td>Keywords and reserved words</td>
<td>No reserved keywords.</td>
<td>Keywords are reserved words.</td>
</tr>
<tr>
<td>Quotation marks</td>
<td>Single or double quotation marks can delimit a character constant. Here are two examples that are equivalent:</td>
<td>ANSI SQL quoting standards are followed:</td>
</tr>
<tr>
<td></td>
<td>&quot;Tom&quot;</td>
<td>• Single quotation marks delimit a character constant.</td>
</tr>
<tr>
<td></td>
<td>'Tom'</td>
<td>• Double quotation marks delimit an identifier.</td>
</tr>
<tr>
<td></td>
<td>For example, &quot;Tom&quot; is a delimited identifier, and 'Tom' is a character constant.</td>
<td></td>
</tr>
<tr>
<td>Topic</td>
<td>DATA Step</td>
<td>DS2</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>PUT statement</td>
<td>Supports column and line parameters.</td>
<td>Column and line parameters are not supported. Dot notation parameters are not supported.</td>
</tr>
<tr>
<td>Variable attributes</td>
<td>Establishing the attributes of a variable requires the use of the LENGTH, FORMAT, INFORMAT, LABEL, and ATTRIB statements.</td>
<td>Establishing the attributes of a variable requires only the DECLARE statement and its HAVING clause.</td>
</tr>
<tr>
<td>Text that is resolved from macro variable references</td>
<td>Double quotation marks are required to reference the resolved value of a macro variable. Here is an example: my_host = &quot;&amp;syshostname&quot;;</td>
<td>Because ANSI SQL quoting standards are followed, double quotation marks denote delimited identifiers. To reference a macro variable in a literal string, use the %TSLIT macro function. Here is an example: my_host = %tslit(&amp;syshostname);</td>
</tr>
<tr>
<td>Data types</td>
<td>Two data types are supported: numeric and character. Numeric data is signed, fractional, limited to 8 bytes, and has approximate precision. Character data is fixed length.</td>
<td>Most ANSI SQL data types are supported. Numeric types of varying sizes and precision. Character data types can be fixed length and variable length. DS2 supports ANSI date, time, and timestamp data types, but can also process SAS date, time, and datetime values using conversion functions.</td>
</tr>
<tr>
<td>Missing and null values</td>
<td>Supports only missing values. No concept of a null value.</td>
<td>Supports both missing and null values. Nulls, from a database, can be processed in ANSI mode or in SAS mode.</td>
</tr>
<tr>
<td>Automatic data type conversion</td>
<td>SAS tries to convert between character and numeric data types when one data type is assigned to a variable of the other data type.</td>
<td>Has many more rules because of many more data types. Most data types are coercible. DATE, TIME, TIMESTAMP, BINARY, and VARBINARY data types are not coercible.</td>
</tr>
<tr>
<td>SQL language statements</td>
<td>Available in PROC SQL, not in the DATA step. Operates only on SAS data sets as tables.</td>
<td>SQL SELECT statements can be written directly in and used as input for a DS2 SET statement. In addition, the SQLSTMT predefined package provides a way to pass SQL statements to a DBMS for execution and to access the result set returned by the DBMS.</td>
</tr>
<tr>
<td>Topic</td>
<td>DATA Step</td>
<td>DS2</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SAS Macro</td>
<td>The DATA step can interact with a macro at run time (for example, CALL EXECUTE, SYMGET, and CALL SYMPUT).</td>
<td>When DS2 runs inside a Base SAS session (for example, in PROC DS2), SAS macros are available. SAS macro support is not available when DS2 runs in the SAS Federation Server and in grid computing environments such as the in-database SAS Embedded Process, the High-Performance Analytics grid, and the In-Database Code Accelerator.</td>
</tr>
<tr>
<td>Overwriting data sets</td>
<td>The DATA step, like SAS procedures, overwrites an existing data set.</td>
<td>In keeping with SQL and database standards, DS2 does not automatically overwrite existing data sets. You must use the <code>overwrite</code> option. Here is an example.</td>
</tr>
<tr>
<td></td>
<td>data one; set two; run; data one; set three; run;</td>
<td>proc ds2; data one;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>method run(); set two; end; enddata; run; data one / overwrite=yes; method run(); set three; end; enddata; run; quit;</td>
</tr>
</tbody>
</table>
## Reading from and writing to the same data set

<table>
<thead>
<tr>
<th>Topic</th>
<th>DATA Step</th>
<th>DS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permits reading from and writing to the same data set name in a DATA or PROC step.</td>
<td></td>
<td>DS2 does not allow reading from or writing to the same table in a single data program. In database fashion, the user must create a temporary table, drop the original table, and then rename the temporary table. This example is equivalent to what the SAS System does to implement the appearance of reading from and writing to the same data set at the successful conclusion of a DATA or PROC step.</td>
</tr>
<tr>
<td>data one;</td>
<td>proc ds2; data temp001; method run(); set one; s = s * 1; end; enddata; run; quit;</td>
<td>proc fedsql; drop table one; alter table temp001 rename to one; run; quit;</td>
</tr>
</tbody>
</table>
Chapter 3
Learning by Example: Using the Sample Programs

About the Getting Started Sample Programs
How to Run the Sample Programs .................................................. 15
Recommended Options .............................................................. 15
Using the DS2 Procedure ......................................................... 15
Verifying Access to DS2 ......................................................... 16

Your First Sample Programs ..................................................... 17
Overview of the Sample Programs .............................................. 17
Example 1: “Hello World!” Program – In a System Method .......... 17
Example 2: “Hello World!” Program – In a User-defined Method ... 18
Example 3: “Hello World!” Program – In a User-defined Package ... 18
Example 4: “Hello World!” Program – In the Implicit Loop .......... 20
Example 5: “Hello World!” Program – In Multiple Package Instances ... 21
Example 6: “Hello World!” Program – Using a Thread ................. 23

See Also .................................................................................... 25

About the Getting Started Sample Programs

How to Run the Sample Programs

You can run all of the getting started sample programs in this chapter from a SAS session using the DS2 procedure. Each is a complete program; simply copy a program into your SAS session and submit it.

The getting started sample programs do not rely on a pre-existing data source, nor do they require a connection to a database. When data is needed, the program creates it.

Note: To submit the getting started sample programs, you must have access to SAS 9.4 or later. Some features might not be available if you do not have the latest release.

Recommended Options

All of the getting started sample programs run correctly with the following SAS system option global statement:

options DS2SCOND=ERROR;
You can also override the default DS2SCOND option setting, WARNING, by specifying the following DS2 procedure option:

```
proc ds2 SCOND=ERROR;
```

The ERROR setting enforces variable declaration strict mode. In strict mode, a compilation error occurs if you do not explicitly declare a variable.

**Tip** Variable declaration strict mode is the recommended best practice when writing DS2 programs.

Unlike Base SAS, DS2 protects existing tables from being overwritten. However, if you are developing or changing a table, package, or thread, you need the ability to overwrite these tables.

The OVERWRITE=YES table option enables you to overwrite the table. Here are some examples:

```
package foo /overwrite=yes;
thread work.foo(int x) /overwrite=yes;
data foo (overwrite=yes);
data my_data /overwrite=yes;
```

**Tip** The OVERWRITE= option requires the forward slash (/) syntax with the PACKAGE and THREAD statements. You can use either the / syntax or the parentheses syntax with the DATA statement.

By default, the value of the OVERWRITE= table option is NO.

**Using the DS2 Procedure**

When you use the DS2 procedure to write a program, place your code within the following framework:

```
options ds2scond=error;
proc ds2;
... DS2 statements ...
run;
quit;
```

For more information, see “DS2” in *Base SAS Procedures Guide*.

**Verifying Access to DS2**

In a SAS session, run the following program:

```
proc ds2;
quit;
```

The following is written to the log:

```
3317  proc ds2;
3318  quit;

NOTE: PROCEDURE DS2 used (Total process time):
   real time           0.08 seconds
   cpu time            0.04 seconds
```

If you see an error message, such as ERROR: Procedure DS2 not found, see your system administrator.
Your First Sample Programs

Overview of the Sample Programs

Because many programmers prefer to learn by reading code, this chapter presents several sample programs before explaining the language constructs that the programs use. The sample programs will help you quickly learn DS2 syntax and concepts so that you avoid common pitfalls.

The first sample program is the “Hello World!” program. Subsequent sample programs are variations on this program. Some variations might seem needlessly complex. The point is to demonstrate common structural programming elements of the DS2 language, not the best way to write the “Hello World!” program.

Note: Although the sample programs introduce syntax and concepts in order of increasing complexity, they do not rely on a particular order to be fully understood.

Example 1: “Hello World!” Program – In a System Method

Here is one way to code the "Hello World!" program. This program writes "Hello World!" to the SAS log from the INIT( ) system method.

What to Notice

• The variable MESSAGE has local scope because it is declared in the INIT( ) method.
• The INIT( ) system method automatically runs first in a DS2 data program.
• Single quotation marks delimit the character constant.

In a SAS session, copy or enter the following program. Then, submit it.

```sas
proc ds2;
data _null_;  /* init() - system method */
   method init();
      declare varchar(16) message; /* method (local) scope */
      message = 'Hello World!';
      put message;
   end;
enddata;
run;
quit;
```

The following is written to the log:

```
Hello World:
```
Example 2: “Hello World!” Program – In a User-defined Method

This variation of the program writes "Hello World!" to the SAS log from a user-defined method.

What to Notice

- Because the variable MESSAGE has global scope in the data program, all methods can access it.
- The INIT( ) system method calls the user-defined GREET( ) method to write the message to the log.

Note: The GREET( ) method is defined before the method that references it. Otherwise, a compilation error would occur.

In a SAS session, copy or enter the following program. Then, submit it.

```sas
proc ds2;
  data _null_;  
    dcl varchar(16) message; /* data program (global) scope */
    /* greet() - user-defined method */
    method greet();
      put message;
    end;
    /* init() - automatically runs first in the data program. */
    method init();
      message = 'Hello World';
      message = cat(message, '!');
      greet();
    end;
  enddata;
run;
quit;
```

The following is written to the log:

```
Hello World!
```

Example 3: “Hello World!” Program – In a User-defined Package

This variation of the program writes “Hello World!” and other messages to the SAS log through a user-defined package.
What to Notice

• The PACKAGE statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.

• The variable MESSAGE is declared inside the package, not in the data program.

• The package contains a constructor and two package methods to manipulate the greeting string.
  
The FORWARD statement enables the SETMESSAGE( ) method to be defined after methods that reference it. Otherwise, a compilation error would occur.
  
The SETMESSAGE( ) method uses the THIS operator to distinguish the global variable MESSAGE from the parameter that is named MESSAGE.

• In the data program, the DECLARE PACKAGE statement simultaneously declares a package variable and constructs an instance of the package using the package constructor.

• Dot notation provides access to package methods from the data program.

In a SAS session, copy or enter the following program. Then, submit it.

```sas
proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message; /* package (global) scope */
  FORWARD setMessage;

  /* greeting(MESSAGE) - constructor */
  method greeting(varchar(100) message);
    setMessage(message);
  end;

  method greet();
    put message;
  end;

  method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
  end;
endpackage;
run;

/* data program */
data _null_; /* declares and instantiates an instance of the GREETING package */
dcl package greeting g('Hello World!'); /* data program (global) scope */

  /* init() - automatically runs first in the data program.*/
  method init();
    g.greet();
    g.setMessage('What''s new?'); /* change greeting */
    g.greet();
  end;
enddata;
run;
```
The following is written to the log:

```
Hello World!
What's new?
```

---

**Example 4: “Hello World!” Program – In the Implicit Loop**

This variation contains two data programs: one to create a table of greetings and one to process the table.

### What to Notice

- In the first data program, the DATA statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.
- The GREETING package has two constructors: a default constructor and one that accepts an argument.
- The second data program uses the two-step method for instantiating a package:
  1. The DECLARE PACKAGE statement declares a global package variable.
  2. The INIT( ) method uses the _NEW_ operator to create the package instance. By using the [THIS] operator, the instance is also global.
- In the second data program, the RUN( ) method uses the implicit loop of the SET statement to read and process the MESSAGE variable from each row in the table.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2;
/* data program # 1 - Creates a table of greetings */
data work.greetings /overwrite=yes;
dcl char(100) message; /* data program (global) scope */
  method init();
    message = 'Hello World!'; output;
    message = 'What''s new?'; output;
    message = 'Good-bye World!'; output;
  end;
enddata;
run;
quit;
proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
dcl varchar(100) message; /* package (global) scope */
  forward setMessage;

  /* greeting() - default constructor */
  method greeting();
    setMessage('This is the default greeting.');
  end;
end;
```
/* greeting(MESSAGE) - constructor */
method greeting(varchar(100) message);
    setMessage(message);
end;

method greet();
    put message;
end;

method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
end;
endpackage;
run;

/* data program #2 */
data _null_;
dcl package greeting g; /* package (global) scope */

/* init() - automatically runs first in the data program. */
method init();
    /* package instance has global scope in the data program */
    g = _NEW_ [this] greeting();
    g.greet();
end;

/* run() - automatically runs after INIT() completes. */
method run();
    /* Implicit loop reads each row from the table */
    set work.greetings;
    g.setMessage(message); /* MESSAGE is read from row by SET statement */
    g.greet();
end;
enddata;
run;
quit;

The following is written to the log:

This is the default greeting.
Hello World!
What's new?
Good-bye World!

Example 5: “Hello World!” Program – In Multiple Package Instances

This version of the program uses multiple instantiation of packages to obtain the same results as Example 4, without creating a table.
What to Notice

- The GREETING package is identical to the GREETING package in the previous example. Because the package already exists, the package block could have been omitted from this program.
- You can use different constructors in the same DECLARE PACKAGE statement.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
dcl varchar(100) message; /* package (global) scope */
FORWARD setMessage;

/* greeting() - default constructor */
method greeting();
setMessage('This is the default greeting.');
end;

/* greeting(MESSAGE) - constructor */
method greeting(varchar(100) message);
setMessage(message);
end;

method greet();
put message;
end;

method setMessage(varchar(100) message);
/* Must use THIS. to distinguish global */
/* variable MESSAGE from parameter named MESSAGE. */
this.message = message;
end;
endpackage;
run;

/* data program */
data _null_
/* All package instances have global scope in the data program. */
dcl package greeting
g0() g1('Hello World!') g2('What''s new?') g3('Good-bye World!');

/* init() - automatically runs first in the data program. */
method init();
g0.greet();
g1.greet();
g2.greet();
g3.greet();
end;
enddata;
run;
quit;
```
The following is written to the log:

```
This is the default greeting.
Hello World!
What’s new?
Good-bye World!
```

**Example 6: “Hello World!” Program – Using a Thread**

This version of the program uses a thread to read a table and pass variables to the data program.

**What to Notice**

- This single DS2 program includes two data programs, a package, and a thread program.
- The data program specifies two threads in the SET FROM statement.
- In the thread program, the RUN( ) method uses the implicit loop of the SET statement to read and process each MESSAGE variable from the table.
- In the data program, the RUN( ) method uses the implicit loop of the SET FROM statement to read and process each MESSAGE variable from the thread program.
- In the log, the thread program’s TERM( ) output shows that the thread program ran twice, once per thread. In addition, the value of _N_ indicates the number of times that the RUN( ) method executed on behalf of each thread.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2;
/* data program #1 - Creates a table of greetings */
data work.greetings /overwrite=yes;
dcl char(100) message; /* data program (global) scope */
method init();
  message = 'Hello World!'; output;
  message = 'What''s new?'; output;
  message = 'Good-bye World!'; output;
end;
enddata;
run;

/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
dcl varchar(100) message; /* package (global) scope */
forward setMessage;

/* greeting() - default constructor */
method greeting();
  setMessage('This is the default greeting.');
end;

/* greeting(MESSAGE) - constructor */
method greeting(varchar(100) message);
  setMessage(message);
```
method greet();
    put message;
end;

method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
end;
endpackage;
run;

/* thread program - Read the table */
thread work.t /overwrite=yes;

/* run() - system method */
method run();
    set work.greetings;
    output; /* output variables to calling program */
end;

/* term() - system method */
method term();
    put _all_;
end;
endthread;
run;

/* data program #2 */
data _null_;
dcl package greeting g; /* data program (global) scope */
dcl thread work.t t; /* data program (global) scope */

/* init() - automatically runs first in the data program. */
method init();
    /* package instance has global scope in the data program */
    g = _NEW_ [this] greeting();
    g.greet();
end;

/* run() - automatically runs after INIT() completes. */
method run();
    /* Implicit loop reads each row from the table */
    set from t threads=2;
    g.setMessage(message); /* MESSAGE read from row by SET FROM statement */
    g.greet();
end;
enddata;
run;
quit;
The following is written to the log:

```
This is the default greeting.
_N_=4 message=Good-bye World!

_N_=1 message=
Hello World!
What's new?
Good-bye World!
```

See Also

- For a high-level overview of DS2 concepts, see Chapter 4, “Building Blocks of DS2 Programs,” on page 27.
- For more information about DS2 methods and packages, see Chapter 5, “Understanding DS2 Methods and Packages,” on page 37.
- To look at advanced, real-world examples of DS2 programs, see Appendix 4, “DS2 Example Programs,” on page 1305.
Chapter 4
Building Blocks of DS2 Programs

Basic DS2 Language Concepts

Introducing DS2 Data Types

Automatic Conversions of Data Types

DS2 Programming Blocks and Scope

Variable Declaration in DS2

DS2 Methods and Packages

Parallel Processing in DS2

What Is a DS2 Program?

Example: Block Scope

Example: A Simple Thread Program

---

Basic DS2 Language Concepts

Introducing DS2 Data Types

Unlike Base SAS, DS2 supports many of the ANSI SQL data types that are native to the data sources that SAS supports. Thus, you can declare DS2 variables that do not require data type conversions to access data that is stored in a data source. The ability to avoid data type conversions enables you to move data efficiently to and from a database or other data source.

Note: The types of data that you can store depend on the native types that your data source supports.

For more information, see Chapter 9, “DS2 Data Types,” on page 71.

The following table summarizes factors to consider when choosing DS2 data types.
### Table 4.1 DS2 Data Types: Quick Reference

<table>
<thead>
<tr>
<th>DS2 Data Type Category</th>
<th>Considerations</th>
</tr>
</thead>
</table>
| Character              | CHAR(n) and VARCHAR(n) use one byte per character. NCHAR(n) and NVARCHAR(n), which handle Unicode national language character sets, use two or four bytes per multi-byte character. When you specify the length of a variable, \( n \), the length determines the size of a fixed-length variable or the maximum size of a variable-length variable. \[
\text{Note: Fixed-length CHAR(n) is the equivalent of a DATA step character variable, where } n \text{ is the number of characters. It is also the default type for an undeclared DS2 character variable.}
\]
| Fractional numeric     | DECIMAL(\( p, s \)) (alias: NUMERIC) has exact precision. Other fractional numeric types include DOUBLE, FLOAT(\( p \)), and REAL (single-precision floating point) and are considered approximate. \[
\text{Note: DOUBLE is the equivalent of a DATA step numeric variable. It is also the default type for an undeclared DS2 numeric variable.}
\]
| Integer numeric        | Signed, exact whole numbers with varying storage sizes: TINYINT (-128 to 127) – 1 byte SMALLINT (-32,768 to 32,767) – 2 bytes INTEGER (-2,147,483,648 to 2,147,483,647) – 4 bytes BIGINT (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) – 8 bytes |
| Binary                 | BINARY(\( n \)) is fixed length. VARBINARY(\( n \)) is variable length. |
| Date and time          | DS2 supports ANSI date, time, and timestamp data types, but can also process SAS date, time, and datetime values using these conversion functions: \[
\begin{itemize}
  \item TO_DATE
    \begin{itemize}
      \item casts a SAS numeric date to a DS2 DATE
    \end{itemize}
  \item TO_TIME
    \begin{itemize}
      \item casts a SAS numeric time to a DS2 TIME
    \end{itemize}
  \item TO_TIMESTAMP
    \begin{itemize}
      \item casts a SAS numeric datetime to a DS2 TIMESTAMP
    \end{itemize}
  \item TO_DOUBLE
    \begin{itemize}
      \item casts a DS2 DATE, TIME, or TIMESTAMP to a SAS numeric date, time, or datetime
    \end{itemize}
\end{itemize}
\]
Automatic Conversions of Data Types

About Automatic Conversions
To avoid unintended results, you must understand the DS2 rules for automatic data type conversion.

CAUTION:
A type conversion can lead to the loss of data or precision, or both. Data type conversions are especially critical if you save DS2 data types in SAS data sets, because SAS data sets support only two data types. That is, DS2 variables might be automatically converted to either fixed-length character or numeric double.

An automatic type conversion occurs under the following circumstances:

• A character type is used in a numeric expression.
• A numeric type is used in a character expression.
• A call to a method supplies an argument value that does not exactly match the signature of the method.
• The types of the operands differ in a logical, arithmetic, relational, or concatenation expression.
• A DS2 data type is saved to a data source that does not support the type.

Coercion and Precedence
DS2 uses type coercion and precedence rules to determine the resulting data type for a conversion:

• Coercible data types can automatically convert to multiple data types.
• Non-coercible data types automatically convert to only character data types.
• Precedence determines the conversion type when an expression contains more than one data type.

For more information, see Chapter 12, “DS2 Type Conversions,” on page 87.

Conversion of Nulls and Missing Values
The mode that you use to process null and missing values can affect your data. The default mode for processing nulls and missing values, either SAS mode or ANSI mode, depends on the environment in which you submit your DS2 program and the options that you choose. For example, by default, the DS2 procedure processes data in SAS mode. The SAS Federation Server processes data in ANSI mode.

CAUTION:
During multiple conversions, it is possible to lose the original meaning of data.
This is particularly true in the values of SAS special missing values (_., .A-.Z).

For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

DS2 Programming Blocks and Scope
A programming block defines a section of a DS2 program that encapsulates variables and code. Programming blocks encourage the creation of modular, reusable code. In addition, a programming block defines the scope of identifiers within that block. In DS2,
it is possible for variables to have the same name and data type, as long as they have different scope.

The following table summarizes the characteristics of DS2 programming blocks and scope.

<table>
<thead>
<tr>
<th>Block</th>
<th>Delimiters</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data program</td>
<td>DATA...ENDDATA</td>
<td>Variables that are declared at the top of this programming block have global scope within the data program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in a data program are included in the program data vector (PDV). <strong>Note</strong>: Global variables exist for the duration of the data program.</td>
</tr>
<tr>
<td>Package</td>
<td>PACKAGE...ENDPACKAGE</td>
<td>Variables that are declared at the top of this programming block have global scope within the package. Package-scope variables are not included in the PDV of a data program that is using an instance of the package. <strong>Note</strong>: Package-scope global variables exist for the duration of the package instance.</td>
</tr>
<tr>
<td>Thread program</td>
<td>THREAD...ENDTHREAD</td>
<td>Variables that are declared at the top of this programming block have global scope within the thread program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in a thread program are included in the thread output set. <strong>Note</strong>: Thread-scope global variables exist for the duration of the thread program instance, but they can be passed to the SET FROM statement in the data program.</td>
</tr>
<tr>
<td>Method</td>
<td>METHOD...END</td>
<td>A method is a subblock of a data program, package, or thread program. Method names have global scope within the enclosing block. Variables that are declared at the top of this programming block have local scope. Local variables are not included in the PDV. <strong>Note</strong>: Local variables exist for the duration of the method call.</td>
</tr>
<tr>
<td>DO loop</td>
<td>DO...END</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>

**TIP** Although method names have global scope, to forward reference a method, use the FORWARD statement at the top of the outer programming block. For more information, see “FORWARD Statement” on page 898.

As the preceding table shows, there are three types of global scope:

- Data program
- Package
- Thread program
Variable Declaration in DS2

Implicit Declaration of Variables
As in Base SAS, DS2 enables you to implicitly create global variables by assignment. However, this is not recommended for these reasons:

- Subtle errors can occur (for example, if a variable name is misspelled).
- The data types of such variables are limited to DOUBLE and CHAR(n).
- Undeclared variables might make your program difficult for others to read and understand.

Variable Declaration with the DECLARE Statement
Unlike Base SAS, DS2 enables you to explicitly declare variables using the DECLARE statement. You can control all attributes of a variable in a single DECLARE statement.

The DECLARE statement takes this form:

```
DECLARE data-type variable-list [HAVING having-clause];
```

Note: A DECLARE statement is allowed only at the top of the programming block in which it is used. Otherwise, a compilation error occurs.

For more information, see “DECLARE Statement” on page 877.

Variable Declaration Strict Mode
A best practice is to always run your DS2 programs using variable declaration strict mode. This enforces the explicit declaration of all program variables.

For more information about controlling variable declaration strict mode, see “DS2SCOND= System Option” on page 966.

DECLARE Statement and Scope
When you use a DECLARE statement to define a variable, the variable assumes the scope of the programming block in which the variable is declared. A method is a subblock of another programming block. Therefore, a variable that is declared in a method has local scope and exists only when the method executes. A variable that is declared outside a method has global scope within that programming block.

For a sample program that demonstrates scope, see “Example: Block Scope” on page 32.

Declaring a Package Instance
Although a package instance is simply a type of variable, it is a special case that is worth mentioning because it has two parts:

package variable
  a variable whose data type is a reference to a type of package.

package instance
  an instance of a type of package. Ideally, a package instance is always referenced by at least one package variable.

TIP The scope of each part is determined by the programming block in which the part is declared. Therefore, a package variable can have a different scope than the package instance that it refers to.
For more information, see “Packages and Scope” on page 137.

**DS2 Methods and Packages**

Methods and packages are explained in greater detail in Chapter 5, “Understanding DS2 Methods and Packages,” on page 37.

**Parallel Processing in DS2**

DS2 supports parallel execution of a single program that can operate on different parts of a table. This type of parallelism is classified as Single Program, Multiple Data (SPMD) parallelism. In DS2, it is the responsibility of the programmer to identify the program statements that can operate in parallel.

**TIP**
For programs that are CPU bound, using a thread program on Symmetric Multiprocessing (SMP) hardware can improve performance. For programs that are either CPU or I/O bound, Massively Parallel Processing (MPP) hardware can improve performance.

For more information, see Chapter 17, “Threaded Processing,” on page 179.

For a sample program that demonstrates a simple use of threads, see “Example: A Simple Thread Program” on page 34.

---

**What Is a DS2 Program?**

For the purposes of getting started with DS2, a DS2 program is a set of DS2 statements that runs in the DS2 procedure. The getting started sample programs demonstrate that a DS2 program can serve many purposes, including but not limited to the following:

- to define and store one or more packages or threads, in permanent or temporary locations.
- to create one or more data sets or tables, in permanent or temporary locations.
- to run one or more data programs, using any, all, or none of the above components.
- any combination of the above. That is, you can create data, packages, and threads, plus run one or more data programs within a single DS2 program.

The order and number of programming blocks in a DS2 program does not matter, as long as the program compiles and contains enough RUN statements to execute the program.

**TIP** Always check the log for compilation errors.

In addition, in a SAS session, you can alternate between the DS2 procedure and Base SAS to test your code and to achieve your programming goals.

---

**Example: Block Scope**

The following program demonstrates how scope determines the visibility of program identifiers.
What to Notice

- This program has six INTEGER variables that have the name `i`.
- This program has three user-defined methods named SHOWME().

```plaintext
options ds2scond=error;
proc ds2;
   /* INNERPKG */
package innerPkg /overwrite=yes;
   dcl int i; /* i is global in this package */
   dcl varchar(100) str;

   /* init() - initializes package variables */
   method init();
      i = 5; /* global i */
      str = 'I am INNERPKG!';
   end;

   /* showMe() - displays values that INNERPKG can "see" */
   method showMe() returns int;
      dcl int i; /* local i */
      i = 10; /* local i */
      put str;
      put 'Local i=' i;
      put 'Global i=' this.i;
      return 1;
   end;
endpackage;
run;

/* OUTERPKG */
package outerPkg /overwrite=yes;
   dcl int i; /* i is global in this package */
   dcl package innerPkg ip();
   dcl varchar(100) str;

   /* init() - initializes package variables */
   method init();
      i = 15; /* global i */
      str = 'I am OUTERPKG!';
      ip.init(); /* tell INNERPKG to initialize itself */
   end;

   /* showMe() - displays values that OUTERPKG can "see" */
   method showMe();
      dcl int i; /* local i */
      i = 20; /* local i */
      put str;
      put 'Local i=' i; /* local i */
      put 'Global i=' this.i; /* global i */
      ip.showMe(); /* tell INNERPKG to show what it can "see" */
   end;
endpackage;
run;
```
Example: A Simple Thread Program

The following program demonstrates how a thread creates data and passes variables to the data program.
What to Notice

• The parameterized thread accepts a value that must be initialized by the data program using the SETPARMS() system method.

• The OVERWRITE=YES table option enables the thread program to be overwritten.

  Note: The THREAD statement syntax requires the ‘/’ (slash character) syntax.

• Because the data program specifies two threads, the thread program runs in two separate threads in a single process.

  Note: This thread program produces one set of output variables per thread.

• Because threads run asynchronously, the order of processing is unpredictable, as the log shows.

• In the data program, the global accumulator variable TOTAL is implicitly retained because of the total + answer; Sum statement syntax.

---

options DS2SCOND=ERROR;
proc ds2;
  /* thread program - Creates data in a loop */
  thread work.t (double d) /overwrite=yes;
    dcl int x;
    dcl double y;

    method init();
      dcl int i; /* local - not included in the output table */
      do i = 1 to 9;
        x = i;
        y = i * 2.5 + d;
        put 'THREAD: i=' i ' x= ' x ' y= ' y;
        output; /* output variables include X and Y */
      end;
    end;

    method term();
      put 'THREAD TERM (_ALL_):';
      put _all_;  
    end;
  endthread;
run;

  /* data program - Reads data from a thread program */
data;
  dcl thread work.t t;
  dcl double answer total;

  method init();
    t.setparms(1.25); /* initialize parameter of thread */
    put 'INIT (_ALL_):';
    put _all_;  
  end;

  method run();
    set from t threads=2; /* input variables include X and Y */
    answer = x + y;
The following is written to the log:

```plaintext
INIT (_ALL_):
total=0 answer= . x= y= ._N_=1
THREAD: i= 1 x= 1 y= 3.75
THREAD: i= 1 x= 1 y= 3.75
THREAD: i= 2 x= 2 y= 6.25
THREAD: i= 3 x= 3 y= 8.75
THREAD: i= 4 x= 4 y= 11.25
THREAD: i= 5 x= 5 y= 13.75
THREAD: i= 6 x= 6 y= 16.25
THREAD: i= 7 x= 7 y= 18.75
THREAD: i= 8 x= 8 y= 21.25
THREAD: i= 9 x= 9 y= 23.75
THREAD TERM (_ALL_):
    THREAD: i= 4 x= 4 y= 11.25
d=1.25 x= y= ._N_=1
THREAD: i= 5 x= 5 y= 13.75
THREAD: i= 6 x= 6 y= 16.25
    x= 1 y= 3.75 answer= 4.75 total= 4.75
THREAD: i= 7 x= 7 y= 18.75
THREAD: i= 8 x= 8 y= 21.25
THREAD: i= 9 x= 9 y= 23.75
THREAD TERM (_ALL_):
    d=1.25 x= y= ._N_=1
    x= 2 y= 6.25 answer= 8.25 total= 13
    x= 3 y= 8.75 answer= 11.75 total= 24.75
    x= 4 y= 11.25 answer= 15.25 total= 40
    x= 5 y= 13.75 answer= 18.75 total= 58.75
    x= 6 y= 16.25 answer= 22.25 total= 81
    x= 7 y= 18.75 answer= 25.75 total= 106.75
    x= 8 y= 21.25 answer= 29.25 total= 136
    x= 9 y= 23.75 answer= 32.75 total= 168.75
    x= 1 y= 3.75 answer= 4.75 total= 173.5
    x= 2 y= 6.25 answer= 8.25 total= 181.75
    x= 3 y= 8.75 answer= 11.75 total= 193.5
    x= 4 y= 11.25 answer= 15.25 total= 208.75
    x= 5 y= 13.75 answer= 18.75 total= 227.5
    x= 6 y= 16.25 answer= 22.25 total= 249.75
    x= 7 y= 18.75 answer= 25.75 total= 275.5
    x= 8 y= 21.25 answer= 29.25 total= 304.75
    x= 9 y= 23.75 answer= 32.75 total= 337.5
TERM: (_ALL_)
total=337.5 answer= . x=9 y=23.75 _N_=19
```
Chapter 5
Understanding DS2 Methods and Packages

Modularity, Encapsulation, and Abstraction in DS2 ........................................... 37
Getting Started with DS2 Methods ................................................................. 38
  What Are DS2 Methods? ........................................................................... 38
  What Are DS2 System Methods? ............................................................... 38
  What Are DS2 User-defined Methods? ...................................................... 39
Getting Started with DS2 Packages ............................................................. 40
  What Are DS2 Packages? ......................................................................... 40
  What Are DS2 Predefined Packages? ......................................................... 42
Example: Introduction to DS2 Methods ....................................................... 42
Example: Introduction to DS2 Packages ....................................................... 44

Modularity, Encapsulation, and Abstraction in DS2

When you use DS2 methods and packages, you approach writing SAS programs differently than you do when programming in Base SAS. These DS2 language constructs follow a more structured-programming and object-oriented approach.

In general, DS2 methods are like functions, procedures, subroutines, and the methods of object-oriented languages such as Java. A method can be thought of as a module that contains a sequence of instructions to perform a specific task. DS2 methods can exist only within a data program, thread program, or package.

Thus, methods enable you to break up a complex problem into smaller modules. Such modules are easier to design, implement, and test. Code reuse can shorten development time and help standardize often-repeated or business-specific programming tasks. Also, modular programming enhances readability and understandability by testers and other programmers.

DS2 packages enable encapsulation and abstraction of behavior. DS2 packages are similar to classes in object-oriented languages. However, a DS2 package can also be used as a bucket of useful but unrelated methods and variables, if that meets your needs.

A DS2 package bundles data and methods into a named object that can be stored and reused by other DS2 programs. Although DS2 packages do not hide their data or methods (there is no concept of public and private), packages can be designed to abstract
behavioral details. In such a package, the methods define the object and enable controlled manipulation of package data.

Getting Started with DS2 Methods

What Are DS2 Methods?

Because executable code can reside only in methods, methods are the structural building blocks of DS2 programs.

There are two types of methods:

System Methods
  Also known as predefined, these methods provide the structural and functional framework for your program to execute.

User-defined Methods
  Similar to functions, procedures, and subroutines in other languages, these are the methods that you create or that someone else created for reuse.

All methods are subblocks of other programming blocks. Each method creates a method scope in which local variables can be defined.

A method programming block begins with the METHOD keyword and ends with the END keyword, as the following example shows.

method foo();
...DS2 variables and statements...
end;

For more information, see “Methods” on page 51.

What Are DS2 System Methods?

System methods provide the structural framework that runs your code. That is, to create a program, simply add DS2 statements to one or more of these system methods: INIT( ), RUN( ), TERM( ).

System methods also provide special functionality that SAS programmers need and expect for their data and thread programs, such as implicit looping. In addition, system methods enable the semantic grouping of code, which helps make DS2 programs easier to organize, understand, and maintain.

Here are basic facts about system methods:

- Every DS2 program contains—and executes—all three of the main system methods. If you do not explicitly code a system method, the system executes a default version.
- You can explicitly code any, all, or none of the system methods.
  
  Note: A program that contains no system methods is syntactically correct, but performs no useful function.
- You cannot change the signature of a system method, and you cannot overload a system method.
- You cannot directly call a system method, with the exception of SETPARMS( ), which applies only to thread programs.

For more information, see “Overview of System Methods” on page 957.
The following table summarizes the execution details of each DS2 system method.

### Table 5.1  DS2 System Methods: Execution Details

<table>
<thead>
<tr>
<th>System Method</th>
<th>Execution Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT( )</td>
<td>Automatically executes one time, as the first method of a program. For more information, see “INIT Method” on page 958.</td>
</tr>
<tr>
<td>RUN( )</td>
<td>Automatically executes after INIT( ) completes. The RUN( ) method is the functional equivalent of the DATA step, running as an implicit loop if the method contains a SET or SET FROM statement. For more information, see “RUN Method” on page 959.</td>
</tr>
<tr>
<td>TERM( )</td>
<td>Automatically executes one time, as the last method of a program. For more information, see “TERM Method” on page 963.</td>
</tr>
<tr>
<td>SETPARMS( )</td>
<td>Executes one time, when called from a data program, to initialize the values of a parameterized thread. For more information, see “SETPARMS Method” on page 961.</td>
</tr>
</tbody>
</table>

### What Are DS2 User-defined Methods?

Similar to functions, procedures, and subroutines in other languages, these are the methods that you create or that someone else created for reuse. Because methods are subblocks of other programming blocks, user-defined methods can exist in data programs, packages, and thread programs.

The following table summarizes basic concepts of user-defined methods.

### Table 5.2  DS2 User-defined Methods: Basic Concepts

<table>
<thead>
<tr>
<th>User-defined Method Topic</th>
<th>Description</th>
</tr>
</thead>
</table>
| Scope                     | The name of a user-defined method has global scope within the programming block in which it is defined. In addition, each method creates a method scope in which local variables can be defined.  
  Note: You can call a user-defined method from wherever the method is in scope, and you can do it as many times as needed. |
| Method parameters         | A user-defined method can accept arguments in the following ways:  
  • by value. The argument value is copied to the method.  
  • by reference (IN_OUT parameter). The method modifies the value of the argument variable. |
| Return values             | Like a function, a user-defined method can return a value, but only if the signature contains no IN_OUT parameters. |
Method overloading

User-defined methods that have the same name can exist in the same scope if their argument signatures are unique. That is, if only the return type differs, then the overloading is ambiguous.

The following examples show an overloaded method with three unique argument signatures:

```plaintext
method squareIt(int value) returns int;  
   return value**2;  
end;

method squareIt(decimal(6,2) value) returns decimal(8,4);  
   return value**2;  
end;

method squareIt(int value, IN_OUT int square);  
   square = value**2;  
end;
```

Calling

Unlike system methods, which automatically run, user-defined methods must be called. You can call a user-defined method as many times as needed.

For more information, see “METHOD Statement” on page 910.

---

### Getting Started with DS2 Packages

#### What Are DS2 Packages?

DS2 packages are language constructs that bundle variables and methods into named objects that can be stored and reused by other DS2 programs. The main benefit of a package derives from the reusability of a set of useful methods. However, DS2 packages are capable of much more.

A DS2 package is similar to a class in object-oriented languages. Thus, you can write packages that approximate the encapsulation and abstraction of object-oriented classes.

*Note:* A DS2 package is not a program. A DS2 package is a template for instantiating an object that can be used in a program.

There are two types of packages:

- **Predefined Packages**
  These packages encapsulate common functionality that is useful to many customer solutions (for example, manipulating hash and matrix data structures). Predefined packages are part of DS2.

- **User-defined Packages**
  These are the packages that you create or that someone else created for reuse.

A package is defined by a package programming block. A package begins with the `PACKAGE` keyword and ends with the `ENDPACKAGE` keyword, as the following example shows.

```plaintext
package foo;
```
The following table summarizes basic concepts of both user-defined and predefined packages.

**Table 5.3 DS2 Packages: Basic Concepts**

<table>
<thead>
<tr>
<th>Package Topic</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Scope**                     | The names of package methods and variables have global scope within the package.  
   *Note:* You can access a package’s methods and variables from wherever the package instance is in scope. |
| **Package methods**           | Because packages are not programs, they do not have system methods that run automatically. Thus, to execute a package method, your program must call it.  
   After you instantiate a package, use dot notation to access a method of the package instance, as the following example shows. |
|                               | method run();  
   dcl package matrix m(2, 3);  
   dcl double mr mc;  
   mr=m.rows();  
   mc=m.cols();  
   put mr=;  
   put mc=;  
   end; |
| **Preparing packages for use**| A package must be compiled and stored before it can be used in a program. You can use the DS2 procedure to define and store a user-defined package.  
   For more information, see “PACKAGE Statement” on page 921. |
| **Overwriting packages**      | Unlike Base SAS, DS2 protects existing tables from being overwritten. However, if you are developing or changing a package, you need the ability to overwrite the package. To overwrite an existing package, use the OVERWRITE=YES table option, as the following example shows. |
|                               | package greeting /overwrite=yes;  
   ...DS2 variables, statements, and methods...  
   endpackage;  
   run; |
# Instantiating packages

To use a package, you create an instance of the package (a package instance) and a variable that references the instance (a package variable). You can instantiate a package in two ways:

- in a single DECLARE PACKAGE statement that simultaneously declares a package variable and constructs one or more package instances:
  
  ```
  dcl package matrix m0() m1(3, 3) m2(5, 4);
  ```

- by first declaring the package variable and then assigning a package instance, using the \_NEW\_ operator:

  ```
  data _null_
  
  dcl package matrix m1 m2; /* package (global) scope */
  
  method init();
  
  dcl package matrix m3; /* method (local) scope */
  
  m1 = \_NEW\_ [THIS] matrix(3, 2); /* package (global) scope */
  
  m3 = \_NEW\_ matrix(5, 4); /* method (local) scope */
  
  m2 = \_NEW\_ [m1] matrix(); /* same scope as m1 instance */
  
  end;
  
  ...DS2 methods...
  
  enddata;
  ```

  `Note:` If you declare a package variable without simultaneously declaring an instance of the package, the value of the package variable is NULL.

For more information, see Chapter 16, “DS2 Packages,” on page 135.

## What Are DS2 Predefined Packages?

Predefined packages encapsulate common functionality that is useful to many customer solutions.

For a list of the predefined packages that are available with DS2, see “Predefined DS2 Packages” on page 143.

You can find many example programs that use predefined packages in Appendix 4, “DS2 Example Programs,” on page 1305.

### Example: Introduction to DS2 Methods

The following program demonstrates many characteristics of both system and user-defined DS2 methods.

#### What to Notice

- This program has three overloaded user-defined methods named SQUAREIT().
- This program contains three system methods.

```options DS2SCOND=ERROR;```
proc ds2;
/* data program */
data _null_
   dcl int root1 result1;
   dcl decimal(6,2) root2;
   dcl decimal(8,4) result2;

   /* overloaded user-defined method */
   method squareIt(int value) returns int;
      return value**2;
   end;

   /* overloaded user-defined method */
   method squareIt(decimal(6,2) value) returns decimal(8,4);
      return value**2;
   end;

   /* overloaded user-defined method with IN_OUT parameter */
   method squareIt(int value, IN_OUT int square);
      square = value**2;
   end;

   /* system method */
   method init();
      root1 = 3.01;
      root2 = 3.01;
   end;

   /* system method */
   method run();
      result1 = squareIt(root1);
      put 'The square of INTEGER ' root1 ' is ' result1;

      result2 = squareIt(root2);
      put 'The square of DECIMAL ' root2 ' is ' result2;

      root1 = 4.99;
      squareIt(root1, result1);
      put 'The square of INTEGER ' root1 ' is ' result1;
   end;

   /* system method */
   method term();
      put _all_; /* final values of global variables */
   end;
enddata;
run; quit;

The following is written to the log:

The square of INTEGER  3  is  9
The square of DECIMAL  3.01  is  9.0601
The square of INTEGER  4  is  16
root1= root2= result1= result2= _N_=1
Example: Introduction to DS2 Packages

The following program demonstrates some basic concepts of packages. This is a version of the “Hello World!” program that uses a package to write messages to the SAS log.

What to Notice

- The PACKAGE statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.
- In the data program, the DECLARE PACKAGE statement simultaneously declares package variables and constructs two separate instances of the package using the package constructor.
- Dot notation provides access to package methods and package-global variables from the data program.

Note: Although this sample program shows that you can directly access package-global variables, the recommended best practice is to avoid this practice unless you have a valid reason to do so.

```sas
options DS2SCOND=ERROR;
proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
dcl varchar(100) message;    /* package (global) scope */

/* greeting(MESSAGE) - constructor that accepts an argument */
method greeting(varchar(100) message);
   THIS.message = message;
end;

/* greeting() - default constructor */
method greeting();
end;

/* package method */
method greet();
   if not missing(message) then
      put message;
   else put 'Message is null or missing.';
end;
run;

/* data program */
data _null_;
/* declares and instantiates two instances of the GREETING package */
dcl package greeting g1('Hello World!') g2();    /* data program (global) scope */

/* init() - system method */
method init();
g1.greet();
```

Chapter 5 • Understanding DS2 Methods and Packages
g2.greet();
g2.message = 'Good-bye World!';
g2.greet();
end;
enddata;
run;
quit;

The following is written to the log:

Hello World!
Message is null or missing.
Good-bye World!
Part 3

DS2 Concepts

Chapter 6
DS2 Programming Semantics ................................................. 49

Chapter 7
DS2 Variables .......................................................... 57

Chapter 8
DS2 Constants .......................................................... 67

Chapter 9
DS2 Data Types .......................................................... 71

Chapter 10
DS2 Identifiers .......................................................... 77

Chapter 11
How DS2 Processes Nulls and SAS Missing Values ................. 81

Chapter 12
DS2 Type Conversions .................................................. 87

Chapter 13
DS2 Expressions .......................................................... 93

Chapter 14
Dates and Times in DS2 ................................................ 111

Chapter 15
DS2 Arrays ............................................................... 121

Chapter 16
DS2 Packages ........................................................... 135

Chapter 17
Threaded Processing .................................................... 179

Chapter 18
Using DS2 and FedSQL .............................................. 183
Chapter 19

DS2 Input and Output .................................................. 185

Chapter 20

Combining Tables ....................................................... 191

Chapter 21

DS2 Reserved Words .................................................... 219
## Basic DS2 Program Syntax

A DS2 program consists of a list of declarations followed by a list of method statements. Here is an example of a simple declare list:

```
declare int x;
declare double d;
```

Here is an example of a simple method statement list:

```
method init();
end;

method run();
end;

method term();
end;
```

Combining the two lists creates a simple DS2 program.

```
declare int x;
declare double d;

method init();
end;

method run();
end;
```
method term();
end;

Although a DS2 program is typically more complex, this simple program contains several syntactic elements:

Keywords:
• DECLARE
• DOUBLE
• METHOD
• END

Identifiers:
• x
• d
• INIT
• RUN
• TERM

Lexical Separators:
• (  
• )  
• ;

The program illustrates how to declare an identifier, either in a DECLARE statement or in a METHOD statement. It also illustrates how the high-level structure of a DS2 program consists of a sequence of variable declarations followed by a sequence of METHOD statements. The next section explains what these terms mean in DS2.

---

**Basic DS2 Program Semantics**

**Variable Declaration Statements**

A variable declaration allocates memory space and identifies that memory with an identifier, called the variable name. The declaration, either explicitly or implicitly, allocates memory for the variable and designates the type of data that can be saved at that memory location. In DS2, you declare variables by using the DECLARE statement. A DECLARE statement performs the following actions:

• assigns an identifier to a memory location. That identifier becomes the variable name.

• designates the type of data that the variable can hold.

• allocates a specified amount of memory to the variable.

More than one variable can be declared in one DECLARE statement. For more information, see the “DECLARE Statement” on page 877.
Methods

Methods are basic program execution units. In DS2, the method structure is used to group related program statements in one syntactically identifiable location. The group of statements in the method can then be easily invoked, or executed, multiple times.

DS2 methods are similar to functions, in languages such as C, and methods in Java. In Base SAS, a user-written function that is created by the FCMP procedure is equivalent to a method. In addition, the label target of a LINK statement is functionally similar to a rudimentary method.

A method defines a scoping block. Therefore, any parameters and any variable declarations in the method body are local to the method.

A type signature, or simply signature, is defined to be the ordered list of the method's parameter types. If any two method definitions have the same name, but different type signatures, the method is overloaded. An error occurs if two method definitions have the same name and same type signature.

Note: You cannot overload a method based on CHAR and NCHAR data types alone if session encoding requires multiple bytes. If a session encoding requires multiple bytes per character, for example, UTF-8, then CHAR and NCHAR are identical types and both use NCHAR. Consequently, the two method definitions would be seen as the same.

There are two types of methods in DS2: system methods, and user-defined methods.

A DS2 program can contain the following three system methods:

```sas
method init();
end;
method run();
end;
method term();
end;
```

Every DS2 program will contain, either implicitly or explicitly, these three methods. If you do not define any one of these methods in a DS2 program, the DS2 compiler will create an empty version of it (like those above). These methods are meant to provide a more structured framework than the SAS DATA Step implicit loop concept. In Base SAS, the entire DATA Step program was included in the implicit loop. In DS2, the implicit loop is represented by the RUN method, with the INIT and TERM methods providing initialization and finalization code, respectively.

When a DS2 program executes, here are the results:

1. The INIT method runs. Any initializations take place.
2. Variables in the program data vector which have not been retained will be set to the appropriate missing values. For more information, see the “RETAIN Statement” on page 930.
3. The RUN method executes.
4. Execution control then depends on the status of any input statement in the RUN method. Currently, the only input statement in DS2 is the SET statement. If the RUN method meets one of these conditions, then processing proceeds to Step 5. Otherwise, processing proceeds to Step 2 so that the RUN method can execute again:
   - No input statements
   - An input statement that has completed execution
5. The TERM method executes, and any final statements execute.

The INIT, RUN, and TERM methods must be defined without any parameters and without a return value. If you specify a parameter for the INIT, RUN, or TERM methods, an error will occur.

If you do not specify an OUTPUT statement in the DS2 program, the DS2 compiler will provide one with no parameters that executes at the end of the RUN method.

If you attempt to call the INIT, RUN, or TERM method directly from a DS2 program, an error will occur.

User-defined methods can be created by enclosing statements that you would like executed one or more times within METHOD and END statements. For more information about user-defined methods, see the “METHOD Statement” on page 910.

Note: When using PROC DS2, DS2 programs are delimited by RUN statements. If additional DS2 code is found after a RUN statement, this code composes a new, distinct DS2 program from the DS2 program before the previous RUN statement.

---

Scope of DS2 Identifiers

**Programming Blocks**

A programming block is a section of code that begins and ends with an ordered pair of keywords. The following keywords create programming blocks:

- `DATA...ENDDATA`
- `PACKAGE...ENDPACKAGE`
- `THREAD...ENDTHREAD`
- `DO...END`
- `METHOD...END`

In this documentation, these terms are used for programming blocks.

- A data programming block or **data program** refers to code bounded by DATA...ENDDATA statements.
- A package programming block or **package** refers to the stored library of variables and methods bounded by PACKAGE...ENDPACKAGE statements. The variables and methods of a package can be used by DS2 programs, threads, or other packages.
- A thread programming block, or **thread program**, refers to a stored program that is bounded by the THREAD...ENDTHREAD statements. The thread program can be called by the SET FROM statement in a DS2 program or package.
- A DO programming block, or **DO loop**, refers to a subblock of programming statements that are bounded by the DO and END statements.
- A method programming block or **method block** refers to a subblock of programming statements that are bounded by the METHOD and END statements.

Some blocks can be nested. In this example, there is one data program, defined by the DATA and ENDDATA statements, and three nested method blocks, defined by the three method statements.

```plaintext
data _null_;```
declare int x;
method init();
    declare double d;
end;
method run();
end;
method term();
end;
enddata;

A variable declared in the outermost programming block is called a **global** variable, or a variable having **global scope**. A variable declared in any nested block is called a **local** variable, or a variable having scope that is local to that block. DS2 also assigns global scope to undeclared variables. In the preceding example, X is a global variable, and D is a variable that is local to the nested INIT method.

**Note:** A DS2 program can have multiple subprograms followed by an optional data program. The following restrictions apply:

- There can be only one data program and the data program must be the last subprogram.
- The ENDPACKAGE, ENDMETHOD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

**Variable Lookup**

When a variable is referenced, DS2 will always search for the variable's declaration beginning in the block of the reference. Then, if it is not found there, it will search successively in any outer containing blocks or program. In this example, any reference to X in the INIT method will refer to the global declaration of X.

declare int x;
method init();
end;

Because methods are blocks, they can contain declarations themselves. In this example, any reference to X in the INIT method will refer to the local declaration of X, but any reference to X in the RUN method will refer to the global declaration of X.

declare int x;
method init();
    declare int x;
end;
method run();
end;
Definition of Scope

Scope can be considered an attribute of identifiers. Identifiers can refer to a number of program entities: method names, functions, data names, labels, or program variables. This section uses program variables as examples, but any identifier is subject to scoping rules.

Scope describes where in a program a variable can be accessed. Global variables have global scope and are accessible from anywhere in the program. Local variables have local scope and are accessible only from within the program or block in which the variable was declared.

In DS2, a variable is accessible only as long as program execution is taking place within the scope of the variable. That is, the values of variables are accessible only when a statement in the scope of the variable is actively executing.

In the following example, the variable X is in scope only while the INIT method is executing. Neither the RUN or the TERM methods can refer to it.

```
data;
  declare int x;    /* global x in global scope */
method init();
  x = 5;          /* global x assigned 5 */
end;
method run();
end;
method term();
end;
enddata;
```

Each variable in any given scope must have a unique name, but variables in different scopes can have the same name. When scopes are nested, if a variable in an outer scope has the same name as a variable in an inner scope, the variable within the outer scope is hidden by the variable within the inner scope. For example, in the following program two different variables share the same name, X. Global variable X has global scope, and local variable X has local scope. Within the local scope of method INIT, local variable X hides global variable X. Therefore, the assignment statement assigns 5 to local variable X.

```
data;
  declare int x;    /* global x in global scope */
method init();
  declare int x;  /* local x in local scope */
  x = 5;          /* local x assigned 5 */
end;
method run();
end;
method term();
end;
enddata;
```
Variable Lifetime

The lifetime of a variable is the time during which the variable exists. Global variables exist for the duration of the program. Local variables exist for the duration of the block in which the variable was declared. The value of a global variable will be set to a missing or null value before entry into the RUN method unless that global variable appears within a RETAIN statement in the current program block.

In the following example, the variable X exists only while the INIT method is executing and the variable Y exists for the duration of the data program.

data;
  declare double y;

  method init();
    declare double x;
  end;
enddata;

During a variable's lifetime, it can be overshadowed by a locally declared variable of the same name, as in this example:

declare int x;

method init();
  declare int x;
end;

method run();
end;

Although the global variable X has lifetime for the entire program, it is not directly accessible from the INIT method because of the local declaration of X in the INIT method.
Chapter 7
DS2 Variables

Introduction to DS2 Variables

The properties of DS2 program variables are that they have a name, a scope, and a data type.
A name, or identifier, is one or more tokens, or symbols, that is given to a variable. Names are discussed in Chapter 10, “DS2 Identifiers,” on page 77.
Variables can have either global or local scope depending on where the variable is declared. For more information, see “Variable Scope” on page 63.
Variable data types are assigned either implicitly or explicitly depending on how they are declared. For more information, see “Variable Declaration” on page 58.

Note: The term “data type” includes any data type attributes such as precision, character set encoding, and length. For complete information about data types, see Chapter 9, “DS2 Data Types,” on page 71.
Variable Declaration

There are three ways to declare a variable and its data type:

- Explicit declaration by using the DECLARE statement

  The DECLARE statement associates a data type with each variable in a variable list or an array. If the DECLARE statement is used outside a method, a global variable is created. If the DECLARE statement is used within a method, a local variable is created. Within a method, DECLARE statements must precede method statements. Otherwise, an error occurs.

  For more information, see “Variable Scope” on page 63 and “DECLARE Statement” on page 877.

- Implicit declaration by using a SET statement

  The SET statement reads the column information for each specified table. For each column in each table, the SET statement creates a global variable in the DS2 program with the same data types as those of the column.

  For more information, see “Reading Data Using the SET Statement” on page 186 and “SET Statement” on page 937.

- Implicit declaration by using an undeclared variable in a programming block

  If you use a variable without declaring it, DS2 assigns the variable a data type. By default, a warning is sent to the SAS log. The data type for an undeclared variable on the left side of an assignment statement is determined by the data type of the value on the right side of the assignment statement. If the data type of the value on the right side of the assignment statement is numeric or NULL, then type DOUBLE is assigned to the left side variable. Otherwise, the data type of the value on the right side is assigned to the left side variable.

  The data type for an undeclared variable on the right side of an assignment statement is DOUBLE.

  Note: A best practice is to declare every variable. By doing so, you can avoid data type mismatches among data sources.

  To control how DS2 handles an undeclared variable, you can use the DS2SCOND system option or the SCOND option on the DS2 procedure:

  Table 7.1  Settings to Control Variable Declaration

<table>
<thead>
<tr>
<th>DS2SCOND/SCOND Setting</th>
<th>Effect on Variable Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>WARNING</td>
<td>Declaration by assignment occurs. Warning messages are written to the SAS log. This is the default behavior.</td>
</tr>
<tr>
<td>NONE</td>
<td>Declaration by assignment occurs. No messages are written to the SAS log.</td>
</tr>
<tr>
<td>NOTE</td>
<td>Declaration by assignment occurs. A note is written to the SAS log.</td>
</tr>
</tbody>
</table>
**DS2SCOND/SCOND Setting**

<table>
<thead>
<tr>
<th>DS2SCOND Setting</th>
<th>Effect on Variable Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Declaration by assignment does not occur. An error message is written to the SAS log. This is also known as variable declaration strict mode.</td>
</tr>
</tbody>
</table>

For more information, see the “DS2SCOND= System Option” on page 966 and “DS2” in *Base SAS Procedures Guide*.

The following example shows how to define variables explicitly and implicitly, or by assignment. FIRSTNAME and LASTNAME are declared as variables with the data type of CHAR(20). PNUM is declared by assignment. DS2 assigns PNUM a data type based on the value on the right side of the assignment, which is CHAR(11):

```sas
data phonenums;
    dcl char(20) firstName lastname;
    method init();
    firstName='Sam';
    lastName='Alesski';
    pnum = '19192223454';
    ... more DS2 statements ...
end;
enddata;
```

### Variable Lists

**Overview of DS2 Variable Lists**

A DS2 variable list is a collection of DS2 variables. Many DS2 statements and table options use variable lists for specification of sets of variables. For example, the KEEP, DROP, and VARARRAY statements can use variable lists for the specification of the set of variables to keep, drop, or reference. Here are some examples:

```sas
keep name address city state zip phone;
drop repl-rep5;
vararray int grades[*] assignment: quiz: exam:;
```

A variable list can also provide a convenient mechanism for specifying variables of interest to a DS2 method or a package.

DS2 supports the following forms of variable lists.

- name variable list
- numbered range variable list
- name range variable list
- name prefix variable list
- type variable list
- special name variable list

For more information, see “Types of Variable Lists” on page 60.
The different forms of variable lists can be mixed within a single variable list. For example, the following is a valid variable list.

\texttt{u x1-x3 u:}

Assuming the program data vector illustrated below, the above variable list would expand to \texttt{u x1 x2 x3 u u1 u2}. Note that a single variable can be referenced multiple times in a variable list expansion.

\begin{table}[h]
\centering
\begin{tabular}{cccccccccc}
  PDV & u & v & w & x1 & x2 & x3 & x4 & u1 & u2 \\
\end{tabular}
\end{table}

\textit{Types of Variable Lists}

\textbf{Name Variable Lists}
A name variable list is simply a list of variable names.

\texttt{location date pressure temperature}

\textbf{Numbered Range Variable Lists}
A numbered range variable list expands to reference global variables with a specified prefix and a numeric suffix ranging between two specified numbers. The numbered range variable list has the following syntax.

\texttt{prefixn1-prefixn2}

\(n1\) and \(n2\) represent the beginning and ending of the range, inclusive. Variable names are constructed during the numbered range list expansion by concatenating the prefix with each number in the numbered range.

For example, the numbered range variable list \texttt{x1-x5} expands to \texttt{x1 x2 x3 x4 x5}. The numbered range variable list \texttt{score10-score5} expands to \texttt{score10 score9 score8 score7 score6 score5}.

\textbf{Name Range Variable Lists}
A name range variable list expands to reference all global variables whose variable definition occurred between the definition of two specified variables, inclusive. The name range variable list has the following syntax.

\texttt{var1--var2}

\texttt{sales_jan--sales_mar} is an example.

DS2 maintains a seen list of defined variables. As variables are defined, the variables are added to the seen list in their order of definition. For example, consider the following statements.

\begin{verbatim}
declare double reg1_id reg2_id reg1_rev reg2_rev;
declare double reg1_exp reg2_exp total_rev total_exp;
keep reg1_rev--reg2_exp;
\end{verbatim}

The seen list after executing the DECLARE statements would be \texttt{reg1_id reg2_id reg1_rev reg2_rev reg1_exp reg2_exp total_rev total_exp}. Therefore, the variable list \texttt{reg1_rev--reg2_exp} in the KEEP statement expands to \texttt{reg1_rev reg2_rev reg1_exp reg2_exp}.\n
\textit{Chapter 7 • DS2 Variables}
Name range variable lists rely on the order of variable definition, that is, \(x--a\) includes all variables in order of variable definition from \(x\) to \(a\) inclusive.

**Name Prefix Variable Lists**

A name prefix variable list expands to reference all global variables that begin with a specified prefix. The name prefix variable list has the following syntax.

\[prefix:\]

An example of a name prefix variable list is \(sales:\), which expands to all variables whose names begin with “sales”, such as \(sales\_jan, sales\_feb\), and \(sales\_mar\).

**Type Variable Lists**

A type variable list expands to reference all global variables of a specified type. The type variable list has the following syntax.

\[data-type:\]

An example of a type variable list is \(smallint\ int\), which expands to all variables of type SMALLINT or INT. The types that are supported by type variable lists are these.

- TINYINT
- SMALLINT
- INTEGER
- BIGINT
- REAL
- FLOAT (matches DOUBLE and FLOAT)
- DOUBLE (matches DOUBLE and FLOAT)
- BINARY (matches BINARY and VARBINARY)
- CHAR (matches CHAR, VARCHAR, and CHARACTER)
- NCHAR (matches NCHAR and NVARCHAR)
- CHARACTER (matches CHAR, VARCHAR, and CHARACTER)
- DATE
- TIME
- TIMESTAMP

**Special Name Variable Lists**

A special name variable list expands to reference a specific group of Read-only, global variables. The _ALL_ variable is supported. _ALL_ references all global variables in the DS2 program.

For more information, see “Predefined DS2 Variables” on page 64.

**Expansion of Variable Lists**

Variable lists expand to reference global scalar variables that match the variable list type. Local variables are never included in a variable list expansion.

In the following example, variable \(x\) is a global variable and variable \(y\) is local to method INIT. The \(x:\) name prefix variable list in the KEEP statement expands to
reference only variable $x_1$. Local variable $x_2$ is not included in the variable list expansion. Therefore, variable $x_2$ is not written to the table \texttt{example}.

```plaintext
data example;
    declare double x1;
    keep x::;

    method init();
        declare double x2;
        end;
enddata;
run;
```

Variable list expansion considers all global variables regardless of where the variable is defined in the program. In the following example, \texttt{_ALL_} in the \texttt{KEEP} statement expands to reference global variables $x_1$, $x_2$, and $x_3$. Therefore, variables $x_1$, $x_2$, and $x_3$ are written to table \texttt{example}.

```plaintext
data example;
    declare double x1;
    keep \_\_ALL\_;
    declare double x2;

    method init();
        x3=17;
        end;
enddata;
run;
```

**Creating Named Variable Lists**

The \texttt{VARLIST} statement creates a named variable list that can be used in multiple DS2 statements. Here is an example.

```plaintext
varlist vars [x1-x5 u v w];

method run();
    compute(vars);
    pkg.doStuff(vars);
end;
```

Note that the \texttt{VARLIST} statement is limited to the global scope of the DS2 package or program. The \texttt{VARLIST} statement cannot be used to create a local variable list.

For more information, see the “VARLIST Statement” on page 955.

**Unnamed Variable Lists**

DS2 provides a mechanism for specification of an anonymous, or unnamed, variable list as part of an expression of another statement. The unnamed variable list has the following syntax.

```
[variable-list]
```

An example of an unnamed list is \texttt{compute([x y z])};.

In this method expression, an unnamed variable list referencing variables $x$, $y$, and $z$ is created and passed as the single argument to the \texttt{compute} method.
If unnamed variable list syntax is used in an expression, then an unnamed variable list is created in the global scope referencing the variables in *variable-list*. An unnamed variable list is inaccessible from statements other than the statement in which the unnamed variable list was specified. Therefore, it cannot be reused in other expressions.

**Passing Variable List Arguments**

A DS2 variable list can be passed as an argument to a DS2 method, to a DS2 function, or to a method of a user-defined package. DS2 variable lists are always passed by reference and cannot be passed by value. A DS2 variable list argument can either be a named variable list that is created with a VARLIST statement, or it can be an anonymous variable list. Here is an example.

```
varlist scores [assignment: quiz: exam:];
method init();
    myMeans.by([gender]);
    myMeans.var(scores);
end;
```

Use the following syntax to specify a variable list parameter for a DS2 method.

```
VARLIST parameter-name
```

Here is an example.

```
method m(varlist list1, varlist list2);
    ...
end;
```

### Variable Scope

#### Global Variables

A variable with global scope, a global variable, is declared in one of three ways: in the outermost programming block of a DS2 program, using a DECLARE statement, implicitly declared inside a programming block using a SET statement, or implicitly declared inside a programming block by using an undeclared variable. Variables with global scope can be accessed from anywhere in the program and exist for the duration of the program. Global variables can be used in a THIS expression in any program block. For information about declaring DS2 variables, see “Variable Declaration” on page 58. For information about using the THIS expression, see “THIS Expression” on page 101.

#### Local Variables

A variable with local scope, a local variable, is declared within an inner programming block, such as a method or package, by using a DECLARE statement. Variables with local scope are known only within the inner programming block where they are declared. For more information, see “Programming Blocks” on page 52.
Global and Local Variables in DS2 Output

Only global variables, by default, are included in the output. Local variables that are used for program loops and indexes do not need to be explicitly dropped from the output. Local variables are always created at the start of a method invocation, such as an iteration of the implicit loop of the RUN method, and are destroyed at the end of each invocation. Therefore, it is not recommended to use local variables as accumulator variables in the RUN method.

All global variables are named in the program data vector (PDV). The PDV is the set of values that are written to an output table when DS2 outputs a row. For more information about the PDV, see “Processing a DATA Step: A Walk-through” in SAS Language Reference: Concepts.

Example of Global and Local Variables

The following program shows both global (A, B, and TOTAL) and local variables (C):

```plaintext
data;
  dcl int a;  /* A is a global variable */
  method init();
    dcl int c;  /* C is a local variable */
    a = 1;      /* A is a global variable */
    b = 2;      /* B is undeclared so it is global */
    c = a + b;
    this.total = a + b + c;  /* THIS.TOTAL simultaneously declares the variable TOTAL as a global variable with the data type of DOUBLE and assigns a value to it based on the values of A, B, and C. */
  end;
enddata;
run;
```

1. A is a global variable because it is declared in the outermost DS2 program.
2. C is a local variable because it is declared inside the method block, METHOD INIT().
3. Because A is a global variable, it can be referenced within the method block, METHOD INIT().
4. Because B is not declared in METHOD INIT(), it defaults to being a global variable. DS2 assigns B a data type of DOUBLE. B appears in the PDV and the output table.
5. THIS.TOTAL simultaneously declares the variable TOTAL as a global variable with the data type of DOUBLE and assigns a value to it based on the values of A, B, and C.

Predefined DS2 Variables

Predefined variables are variables that are automatically declared within a method block and discarded when the method is complete. The values of predefined variables are retained from one iteration of a RUN method to the next. These variables are added to the program data vector but are not output to the table being created. Predefined variables are temporary and are not saved with your data. The predefined variable is available in the method block, and you can use them just like any variable that you declare yourself.
Two predefined variables are created:

_N_

is initially set to 0 by the INIT method. Each time the RUN method executes, the value increments by 1. The data type for _N_ is BIGINT. This is a Read-only variable; you cannot assign a value to _N_.

_N_

is initially set to 1 by the INIT method. Each time the RUN method executes, the value increments by 1. That is, the first time the RUN method executes, the value of _N_ is 1. The value indicates the number of times that the program has looped through the method. The variable can be used to count rows, but it is a good row counter only when one record is read per iteration. The data type for _N_ is BIGINT. You can assign a numeric value to _N_, but in the next iteration of the RUN method, the value reverts to the value assigned by the program.

Although the predefined variables are not output variables, you can use the PUT statement with the _ALL_ argument to print the values of predefined variables to the SAS log.

The following simple DS2 program illustrates using the PUT _ALL_ statement to print the values of the _N_ and _N_ variables to the SAS log.

data inp /overwrite=yes;
    dcl double a;
    method init();
        a = 1; output;
    end;
enddata; run;

data;
    method init();
        put 'init' _n=; put _ALL_;
    end;
    method run();
        set inp;
        put 'run' _n=; put _ALL_;
    end;
    method term();
        put 'term' _n=; put _ALL_; 
    end;
enddata; run;

Log 7.1 SAS Log with _N and _N_ Variable Values

NOTE: Execution succeeded. One row affected.
init _n=0
a=1 _n=1
run _n=1
a=1 _n=1
term _n=2
a=1 _n=2
Chapter 8
DS2 Constants

Definition of a Constant

A constant is a number, character string, binary number, date, time, or timestamp that indicates a fixed value. The following are examples of DS2 constants:

107
'Trends in Business'
date '2008-01-01'
b'10011001'
x'FFE3546F'

Numeric Constants

A numeric constant is a negative or positive numeric value that is either an integer constant or a fractional constant.

integer constant

is a numeric value without a fractional component, that is, a whole number. An integer constant value is stored as an exact numeric. An integer constant without the N or n suffix is a BIGINT, INTEGER, SMALLINT, or TINYINT data type value. An integer constant with the N or n suffix is a DECIMAL or NUMERIC data type value. An integer constant has the following forms where integer is a sequence of one or more digits, 0 through 9:

integer

integerN|n
The following values are valid integer constants:

0
124
124N
+124n
-124n

fractional constant

is a numeric value that has a fractional or decimal component. A fractional constant value is stored as either an exact numeric or an approximate numeric. A fractional constant without the N or n suffix is a REAL or DOUBLE data type value that is stored as an approximate numeric. A fractional constant with the N or n suffix is a DECIMAL or NUMERIC data type value that is stored as an exact numeric up to the maximum precision of the data type. Approximate fractional constants support standard notation or scientific (E) notation. Exact fractional constants support only standard notation. A fractional constant has the following forms where integer, fraction, and exponent is a sequence of one or more digits, 0 through 9:

integer.
integer.N|n
integer.fraction
.fraction
integer.fractionE[e][+-]exponent
integerE[e][+-]exponent
.fractionE[e][+-]exponent
integer.fractionN|n
.fractionN|n

The following values are valid fractional constants:

0.
124.0
+124.0
-57.33
.5
1E100
99.99e4
- .33e-2
- .33e-2
123456789012345678901234567890.12N

Character Constants

A character constant is a sequence of characters enclosed in single quotation marks and can be written using the following formats:

'character-string'
n'character-string'

For character strings that contain national characters, use the NCHAR constant, n'character-string'. National characters can take multiple bytes of storage.
Character and NCHAR constants can include a newline for character constants that span multiple lines.

*Note:* A sequence of characters enclosed in double quotation marks is not a character constant, it is an identifier. For more information, see “Delimited Identifiers” on page 78.

The following are valid character constants:

```sql
'PHONE'
'n'STÄDTE'
'Phone
Number'  /* constant that spans more than one line */
```

*Note:* To have a single quotation mark (') or apostrophe (') in a string, use an additional quotation mark. Here is an example.

```sql
'Isn''t life beautiful'
```

### Binary Constants

Binary constants can be written as either a bit string or as a hexadecimal string using the following formats:

- `b'bit-string'`
- `x'hexadecimal-string'`

*bit-string* is a sequence of binary digits, 0 and 1.

*hexadecimal-string* is a string of hexadecimal characters, 0 - 9 and A - F.

Both *bit-string* and *hexadecimal-string* must be enclosed in single quotation marks.

DS2 converts both the bit constant and the hexadecimal constant to their binary equivalents for use in the DS2 program. Binary and hexadecimal constants are padded with zeros to a byte boundary.

The following are valid binary constants:

```sql
b'11100011'
x'FF143E99'
```

### Date and Time Constants

Date and time constants can be written as date, time, or timestamp strings using the following formats:

- `date 'yyyy-mm-dd'`
- `time 'hh:mm:ss'`
- `time 'hh:mm:ss.fraction'`
- `timestamp 'yyyy-mm-dd hh:mm:ss'`
- `timestamp 'yyyy-mm-dd hh:mm:ss.fraction'`

*yyyy* is a four-digit year.

*mm* is a two-digit month.
**Constant List**

A constant list is a grouping of numeric and character constants that are enclosed in parentheses, separated by commas or blanks, that can be used in simple array assignments and in IN expressions. An example of a constant list is `(2, '33', 2*–5, 2*(1, '3'))`. 

A constant list can be recursive and they can contain other constant lists. Individual list items can be repeated by using an iterator. An iterator is a number followed by the asterisk (`*`). The iterator indicates the number of times to repeat the constant or constant list that follows the asterisk. In the example above, the constant list item `2*–5` results in `–5, –5`, and the list item `2*(1, '3')` results in `1, '3', 1, '3'`. 

The above constant list `(2, '33', 2*–5, 2*(1, '3'))` is equivalent to `(2, '33', –5, –5, 1, '3', 1, '3')`. 

A constant list has no data type nor does it have scalar values.
Chapter 9

DS2 Data Types

What Are the Data Types?

A data type is an attribute of every column that specifies what type of data the column stores. The data type is the characteristic that identifies a piece of data as a character string, an integer, a floating-point number, or a date or time. The data type also determines how much memory to allocate for the column’s value.

The following table lists the set of data types that are supported by DS2. Note that not all data types are available for table storage on each data source.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded.</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>stores fixed-length binary data, where n is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CHAR($n$)</td>
<td>stores a fixed-length character string, where $n$ is the maximum number of characters to store. The maximum number of characters is required to store each value regardless of the actual size of the value. If <code>char(10)</code> is specified and the character string is only five characters long, the value is right padded with spaces.</td>
</tr>
<tr>
<td>DATE</td>
<td>stores a calendar date. A date literal is specified in the format <code>yyyy-mm-dd</code>: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as <code>1975-09-24</code>. DS2 complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of dates. For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS data set or an SPD data set.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC($p,s$)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>stores a signed, approximate, double-precision, floating-point number. Data defined as FLOAT is treated the same as DOUBLE.</td>
</tr>
</tbody>
</table>
| INTEGER | stores a regular size signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded.  
*Note:* Integer division by zero does not produce the same result on all operating systems. It is recommended that you avoid integer division by zero. |
<p>| NCHAR($n$) | stores a fixed-length character string like CHAR but uses a Unicode national character set, where $n$ is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters. |</p>
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVARCHAR(n)</td>
<td>stores a varying-length character string like VARCHAR but uses a Unicode national character set, where ( n ) is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.</td>
</tr>
<tr>
<td>REAL</td>
<td>stores a signed, approximate, single-precision, floating-point number.</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer data types do not store decimal values; fractional portions are discarded.</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>stores a time value. A time literal is specified in the format ( hh:mm:ss[.nnnnnnnnn] ); a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as 06:30:00. When supported by a data source, the ( p ) parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>stores both date and time values. A timestamp literal is specified in the format ( yyyy-mm-dd hh:mm:ss[.nnnnnnnnn] ); a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as 1975-09-24 06:30:00. When supported by a data source, the ( p ) parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.</td>
</tr>
<tr>
<td>TINYINT</td>
<td>stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>stores varying-length binary data, where ( n ) is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If \texttt{varbinary}(10) is specified and the binary string uses only five bytes, only five bytes are stored in the column.</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>stores a varying-length character string, where ( n ) is the maximum number of characters to store. The maximum number of characters is not required to store each value. If \texttt{varchar}(10) is specified and the character string is only five characters long, only five characters are stored in the column.</td>
</tr>
</tbody>
</table>
Data Type Characteristics

**Numeric Data Types**

Numeric data types store numbers (for example, quantities and currency values). The choice of a numeric data type depends on the type of number that is being stored and on how the number will be used.

Characteristics to consider when choosing a numeric data type include the following:

- whether you want to use exact numeric data types or approximate numeric data types. Exact numeric data types such as BIGINT, DECIMAL, INTEGER, NUMERIC, and TINYINT, represent a value exactly. Approximate numeric data types, such as REAL, DOUBLE, and FLOAT, do not store the exact values that are specified for many numbers. For many applications, the tiny difference between the specified value and the stored approximation is not noticeable unless exact numeric behavior is required.

- whether you want to store whole numbers or decimal numbers. Note that for integer data types, if a decimal value is inserted, the fractional portion is discarded. For example, if 2.7 is inserted, the stored value is 2.

- how to store numbers with decimal points. Floating-point format is a number in which the decimal point is not fixed. A floating-point data type is used for longer decimal values and for quickly calculating a large range of numbers. For the floating-point data types, if you enter an integer that is too large, an error occurs. Enter the value using floating-point notation. Fixed point is a method for storing a number in which the decimal points line up. The fixed-point data type DECIMAL is used for exact precision of fractional components, such as money.

- a data type's precision. The precision for a data type specifies the total number of digits that a number can contain. The more digits, the higher the precision for the value. If a value is out of range, an error occurs. For example, for the BIGINT data type, an error occurs for a value that is larger than 9,223,372,036,854,775,807.

To avoid errors or incorrect results, you must consider the results when performing operations on numeric values, particularly for the integer data types. The INTEGER data type has a precision of 10 digits and a range from –2,147,483,648 to 2,147,483,647. If you multiply two large integers (for example, 33432*79879) and the result is larger than 2,147,483,647, an overflow error occurs. If you expect a result that is larger than the data type's precision or range, you can assign the result as a larger data type such as DOUBLE or BIGINT, or you can enter the expression as a double constant (for example, 33432.0*79879.0), not an integer, in order to force the expression to be evaluated as a double precision, floating-point number.

**Character Data Types**

DS2 provides several character data types that store character string (text) data. Character data types can contain alphabetic characters, numeric digits 0 through 9, and other special characters.

*Note:* If a character string includes a number, DS2 automatically converts it to a numeric type and uses that number in any calculation.
Each character data type provides a parameter for specifying the maximum number of characters.

The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character, for example, UTF-8, then CHAR and NCHAR are identical types and both use NCHAR.

You can specify the character set encoding information for CHAR and VARCHAR data types in a DECLARE, METHOD, or VARRAY statement. The default encoding depends on your operating system and locale. For a complete list of character set encoding values, see “Character Sets for Encoding in NLS” in the SAS National Language Support (NLS): Reference Guide.

Note: Trailing blanks are always ignored even for columns that have the VARCHAR data type.

Note: The ENCODING= system option is not supported when you are reading or updating SAS data sets. DS2 converts a given ENCODING= value to the least common denominator encoding when multiple encoding values are specified with the CHARACTER SET syntax. The ENCODING= value that is used can differ from what really is created in the data set. This conflict can cause potential transcoding errors.

**Date and Time Data Types**

DS2 supports several data types for the specific purpose of storing dates and times. The date and time ranges are data source specific.

---

**Define Data Types for a Column**

When defining a data type, use the data type keywords for the data types supported by DS2.

Keep in mind that for data to be stored, the data type must be available for data storage in that data source. Although DS2 supports several data types, the data types that can be defined for a particular table depend on the data source, because each data source does not necessarily support all of the DS2 data types. In addition, data sources support variations of the standard SQL data types. That is, a specific data type that you specify might map to a different data type and might also have different attributes in the underlying data source. This occurs when a data source does not natively support a specific data type, but data values of a similar data type can be converted without data loss. For example, to support the INTEGER data type, a SAS data set maps the data type definition to SAS numeric, which is a DOUBLE.

For details about data source implementation for each data type, see Appendix 1, “Data Type Reference,” on page 1267.

In addition, the CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source or the specified precision or scale is not within the data source range. For
information about the CT_PRESERVE= connection argument, see *SAS Federation Server: Administrator's Guide*.

---

**Error Messages That Use the DOUBLE and REAL Data Types**

When error messages contain numeric values for DOUBLE and REAL data types, these values are written to the log using the BESTw. format. The BESTw. format is the default format for writing numeric values. When you use BESTw., SAS chooses the format that provides the most information about the value according to the available field width.

Because SAS uses the BESTw. format, the DOUBLE and REAL data type values might not be exactly the same as the values that you use in your programs. BESTw. rounds the value, and if SAS can display at least one significant digit in the decimal portion, within the width specified, BESTw. produces the result as a decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.
Overview of Identifiers

An identifier is one or more tokens, or symbols, that name programming language entities, such as variables, method names, package names, and arrays, as well as data source objects, such as table names and column names.

The DS2 language supports ANSI SQL:1999 standards for both regular and delimited identifiers.

Regular identifiers are the type of identifiers that you see in most programming languages. They are not case-sensitive so that the identifier `Name` is the same as `NAME` and `name`. Only certain characters are allowed in regular identifiers.

Delimited identifiers are case-sensitive only for identifiers that require double quotation marks, that is, table and schema names. Other delimited identifiers are not case-sensitive. Delimited identifiers allow any character and must be enclosed in double quotation marks. Variable names `"Name"`, `"NAME"`, `Name`, `name`, and `NaMe` all represent the same variable.

By supporting ANSI SQL:1999 identifiers, the DS2 language is compatible with data sources that also support the ANSI SQL:1999 identifiers.

*Note:* Identifiers for SAS and SPD data sets are limited to 32 characters.

Regular Identifiers

When you name regular identifiers, use these rules:

- The length of a regular identifier can be 1 to 256 characters.
• The first character of a regular identifier must be a letter or underscore. Subsequent characters can be letters, digits, or underscores. Note that you cannot begin a regular identifier with two underscores.

• Regular identifiers are case-insensitive.

The following are valid regular identifiers:

```plaintext
firstName
lastName
_phone_num
phone_num1
```

Letters in regular identifiers are stored internally as uppercase letters, which allows letters to be written in any case. For example, an unquoted, input column name of `phone_num1`, `Phone_Num1`, or `PHONE_NUM1` is displayed in the output as uppercase, `PHONE_NUM1`. Identifiers that do not contain special characters or spaces and are enclosed in double quotation marks are treated as regular identifiers as if the quotation marks were not present.

**Note:** Each data source has its own naming conventions, all of which are supported by the DS2 language. When your program contains identifiers specific for a particular data source, you must follow the naming conventions for that data source. For more information, see the topic on naming conventions for your data source in *SAS/ACCESS for Relational Databases: Reference*.

---

### Delimited Identifiers

When you name delimited identifiers, follow these rules:

• The length of a delimited identifier can be 1 to 256 characters.

• Begin and end delimited identifiers with double quotation marks.

• Delimited identifiers consist of any sequence of characters, including spaces and special characters, between the beginning and ending double quotation marks.

• The only delimited identifiers that are case-sensitive are table and schema names. Other delimited identifiers are not case-sensitive.

A string of characters enclosed in double quotation marks is interpreted as an identifier and not as a character constant. Character constants can be enclosed only in single quotation marks.

The following delimited identifiers are valid:

```plaintext
"x\&yz"  
"01"  
"(area)phone_num"  
"a**B"
```

You can use delimited identifiers for terms that might otherwise be a reserved word. For example, to use the term “date” other than for a date declaration, you would use it as the delimited identifier “date”. Here is an example.

```plaintext
/* In DATA step, there are no reserved words for variables. */
/* So, this doesn't cause an error. However, how do you */
/* use such a variable in DS2? */
data a;
  date = mdy(8,14,2012);
```
run;

/* This program gives an error because the reserved word DATE */
/* is used when a variable is expected. */
proc ds2;
data b(overwrite=yes);
method run();
set a;
if month(date) = 8 then
   put 'August';
else
   put 'Not August';
end;
enddata;
run; quit;

/* One solution is to quote the reserved word */
proc ds2;
data b(overwrite=yes);
method run();
set a;
if month("date") = 8 then
   put 'August';
else
   put 'Not August';
end;
enddata;
run; quit;

/* Another solution is to rename the variable on input and output */
/* to avoid the reserved word in DS2 code. */
proc ds2;
data b(overwrite=yes rename=(sas_date="date");
method run();
set a(rename="date"=sas_date));
if month(sas_date) = 8 then
   put 'August';
else
   put 'Not August';
end;
enddata;
run; quit;

A warning is issued for tables that are created with delimited column names that are then referenced in DS2 programs that are submitted to data sources that are not case-sensitive. Data sources that are not case-sensitive remove the quotation marks and treat the column name as not delimited.

A delimited identifier is similar to name literals in the SAS DATA step language.

Note: Each data source has its own naming conventions, all of which are supported by the DS2 language. When your program contains identifiers specific for a particular data source, you must follow the naming conventions for that data source. For more information, see the topic on naming conventions for your data source in SAS/ACCESS for Relational Databases: Reference.
Referencing a Macro Variable in a Delimited Identifier

To reference a macro variable in a delimited identifier, use the SAS macro function %TSLIT, which overrides the need for double quotation marks around the literal string and puts single quotation marks around the input value. For example, the following statement includes the %TSLIT function to specify the macro variable, %profit:

```
put %tslit(PROFIT: &profit);
```

The %TSLIT macro function is stored in the default autocall macro library. The syntax is as follows:

```
%TSLIT([literal-text ]macro-call);
```

Note: If you do not specify literal-text and a null value is passed to the macro variable reference, a warning message Argument 1 to function TRANSLATE referenced by the %SYSFUNC or %QSYSFUNC macro function is out of range. is written to the SAS log.

Support for Non-Latin Characters

The DS2 language supports only Latin characters for regular identifiers. To use non-Latin characters, the identifier must be delimited using double quotation marks.
Chapter 11
How DS2 Processes Nulls and SAS Missing Values

DS2 Modes for Nonexistent Data

Many relational databases such as Oracle and DB2 implement ANSI SQL null values. Therefore, the concept of null values using DS2 is the same as using the SQL language for databases that support ANSI SQL. It is important to understand how DS2 processes SAS missing values because data can be lost.

*Note:* Only DOUBLE or CHAR data types can have missing values. Data types other than DOUBLE or CHAR can have only null values.

Because there are significant differences in processing null values and SAS missing values, DS2 has two modes for processing nonexistent data: ANSI SQL null mode (ANSI mode) and SAS missing value mode (SAS mode).

The behavior of nonexistent data depends on how you connect to your data source:

- DS2 code that is submitted to PROC DS2 or PROC HPDS2 processes the data using SAS mode. PROC DS2 provides the ANSIMODE option that enables you to process data using ANSI mode.
- DS2 code that is submitted to the SAS In-Database Code Accelerator or the SAS Scoring Accelerator processes the data using SAS mode.
- DS2 code that is submitted to the SAS Federation Server processes data in ANSI mode.

In most instances, no mode change is necessary to process nonexistent data. You can be in any mode and still operate on null and missing values together.

The following are instances of when you might want to change the mode:

- when a client application processes SAS data sets and the mode for nonexistent data is in ANSI mode
when the processing of SAS data sets is complete and the client application is ready to return to ANSI mode

CAUTION:

If the mode is not set for the desired results, data is lost. In ANSI mode, when DS2 reads a SAS numeric missing value from a SAS data set, it converts the SAS missing value to an ANSI null value. If the ANSI null value is then written to a SAS data set, DS2 converts the ANSI null value to the SAS numeric missing value (.). If the SAS numeric missing value from the input data set is a special numeric missing value, such as .A, the .A is lost during the conversion to and from ANSI null, and the SAS numeric missing value (.) is written to the output data set. In the following example, column x is of data type DOUBLE. The example illustrates how a SAS special numeric missing value (.A of the third observation) is transformed to the SAS numeric missing value (.) by a DS2 program run in ANSIMODE.

/* assume input data set indata contains */
  x
  100
  .
  .A

/* Run the following program using ANSI mode */
proc ds2 ansimode;
  data outdata;
    method run();
    set indata;
    end;
  enddata;
  run;
quit;

/* the output dataset outdata contains */
  x
  100
  .
  .

In SAS mode, when DS2 reads an ANSI null value of data type CHAR from an ANSI data source, it converts the ANSI null value to a SAS character missing value (blank-filled character string). If the SAS character missing value is then written to an ANSI data set, the output CHAR column value is a blank-filled character string rather than the ANSI null value. In the following example, column x is of data type CHAR(5). The example illustrates that the ANSI null value (in the second observation) is transformed to a blank-filled string (" ") by a DS2 program that is run in SAS mode.

/* assume input data set indata contains */
  x
  'abc '
  null

/* run the following program using SAS mode */
proc ds2;
  data db.outdata;
    method run();
    set db.indata;
Differences between Processing Null Values and SAS Missing Values

Processing SAS missing values is different from processing null values and has significant implications in these situations:

- when filtering data (for example, in a WHERE clause, a HAVING clause, a subsetting IF statement, or an outer join ON clause). SAS mode interprets null values as SAS missing values, which are known values, whereas ANSI mode interprets null values as unknown values.

- when submitting outer joins in ANSI mode, internal processing might generate nulls for intermediate result tables. DS2 might generate SAS missing values in SAS mode for intermediate result tables. Therefore, for intermediate result tables, nulls are interpreted as unknown values in ANSI mode and in SAS mode, missing values are interpreted as known values.

- when comparing a blank space, SAS mode interprets the blank space as a missing value. In ANSI mode, a blank space is a blank space, it has no special meaning.

- DS2 interprets a null as a SAS missing value in these cases:
  - in SAS mode, a null is used in a computation or assignment involving a floating-point or fixed-width character value
  - a null is passed to a SAS format or function

Note: If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.

The following are attribute and behavior differences between null values and SAS missing values:

<table>
<thead>
<tr>
<th>Attribute or Behavior</th>
<th>ANSI SQL Null value</th>
<th>SAS Missing Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal representation</td>
<td>metadata</td>
<td>floating-point or character</td>
</tr>
</tbody>
</table>
Attribute or Behavior | ANSI SQL Null value | SAS Missing Values  
---|---|---
evaluation by logical operators | is an unknown value that is compared by using three-valued logic, whose resolved values are True, False, and Unknown. For example, \texttt{WHERE \textit{col1} = null} returns \textit{UNKNOWN}. | is a known value that when compared, resolves to a Boolean result  
collating sequence order | appears as the smallest value | appears as the smallest value

Reading and Writing Nonexistent Data in ANSI Mode

Many relational databases such as Oracle and DB2 implement ANSI SQL null values. Therefore, the concept of null values using DS2 is the same as using the SQL language for databases that support ANSI SQL. It is important to understand how DS2 processes SAS missing values because data can be lost.

SAS missing value data types can be only DOUBLE or CHAR. Therefore, only the conversion for these data types is shown. The following table shows the value returned to the client application when DS2 reads a null value or a SAS missing value from a data source in ANSI mode:

Table 11.2 Reading Nonexistent Data Values in ANSI Mode

<table>
<thead>
<tr>
<th>Column Data Type</th>
<th>Nonexistent Data Value</th>
<th>Value Returned to the Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>\texttt{._} or \texttt{.A–Z} *</td>
<td>null</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>CHAR</td>
<td>\texttt{'} \texttt{␣} \texttt{'}</td>
<td>\texttt{'} \texttt{␣} \texttt{'}</td>
</tr>
</tbody>
</table>

\* The value \texttt{.} \texttt{␣} \texttt{.} (period followed by an underscore) or \texttt{A–Z} represent a special numeric missing value. When SAS prints a special missing value, it prints only the letter or underscore.

\* \* The value \texttt{'} \texttt{␣} \texttt{'} is a blank space between single quotation marks, which in ANSI mode, is a blank space, not nonexistent data.

This next table shows the value stored when nonexistent data values are written to data sources in ANSI mode:
### Table 11.3  Storing Nonexistent Data in ANSI Mode

<table>
<thead>
<tr>
<th>Column Data Type</th>
<th>Nonexistent Data Value</th>
<th>Value Stored in the SAS Data Set</th>
<th>Value Stored in the ANSI SQL Null Supported Data Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>null</td>
<td>.</td>
<td>null</td>
</tr>
<tr>
<td>CHAR</td>
<td>null</td>
<td>‘ ’</td>
<td>null</td>
</tr>
<tr>
<td>CHAR</td>
<td>‘ ’</td>
<td>‘ ’</td>
<td>‘ ’</td>
</tr>
</tbody>
</table>

* The value ‘ ’ is a blank space between single quotation marks, which in ANSI mode, is a blank space, not nonexistent data.

In ANSI mode, a blank space is always interpreted as a blank space and not as nonexistent data. Also, in ANSI mode, SAS missing values are converted to ANSI null values at input or assignment.

### Reading and Writing Nonexistent Data in SAS Mode

When the client application uses SAS mode, nonexistent data values are treated like SAS missing values in the Base SAS environment.

The following table shows how nonexistent data values are read in SAS mode:

#### Table 11.4  Reading Nonexistent Data Values in SAS Mode

<table>
<thead>
<tr>
<th>Column Data Type</th>
<th>Nonexistent Data Value</th>
<th>Value Returned to the Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>‘.’, ‘_’, or ‘A–Z’</td>
<td>‘.’, ‘_’, or ‘A–Z’</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>null</td>
<td>.</td>
</tr>
<tr>
<td>CHAR</td>
<td>‘ ’</td>
<td>‘ ’</td>
</tr>
<tr>
<td>CHAR</td>
<td>null</td>
<td>‘ ’</td>
</tr>
</tbody>
</table>

* The value ‘.’ (period followed by an underscore) or ‘A–Z’ represent a special numeric missing value. When SAS prints a special missing value, it prints only the letter or underscore.

** The value ‘ ’ is a blank space between single quotation marks, which in SAS mode, is nonexistent data.

*** When the SET statement encounters a null for a fixed-width character string, the string contains blank characters for the length of the string.

The next table shows how missing data values are written to a data source in SAS mode:
Table 11.5  Writing Nonexistent Data Values to a SAS Data Source in SAS Mode

<table>
<thead>
<tr>
<th>Column Data Type</th>
<th>Nonexistent Data Value</th>
<th>Value Stored in the SAS Data Set</th>
<th>Value Stored in ANSI SQL Null Supported Data Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>., _, or .A</td>
<td>., _, or .A-Z</td>
<td>null</td>
</tr>
<tr>
<td>CHAR</td>
<td>' '</td>
<td>' '</td>
<td>' '*</td>
</tr>
</tbody>
</table>

* The value ' ' is a blank space between single quotation marks, which in SAS mode, is nonexistent data.

In SAS mode, ANSI null values for type DOUBLE or CHAR are converted to SAS missing values at input or assignment.

Testing for Null Values

DS2 provides the NULL function to test for a null value. The NULL function has one argument, which can be an expression. If the expression is null, the function returns 1. If the expression is not null, the function returns 0.

This example shows a test for a null value:

```plaintext
if null(numCopies) then put 'Number of copies is unknown.';
else put 'Number of copies is' numCopies;
```

The NMISS function returns the number of null and SAS missing numeric values. The NMISS function requires numeric values and works with multiple numeric values, whereas NULL works with only one value that can be either numeric or character.

For more information, see the “NULL Function” on page 670 and the “NMISS Function” on page 641.

Testing for Missing Values

DS2 provides the MISSING function to test for a null value. The MISSING function has one argument, which can be a numeric or character expression. If the expression is null, the function returns 1. If the expression is not null, the function returns 0.

The NMISS function returns the number of null and SAS missing numeric values. The NMISS function requires numeric values and works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

For more information, see the “MISSING Function” on page 629 and the “NMISS Function” on page 641.
Type Conversion Definitions

- **binary data type**
  - refers to the BINARY and VARBINARY data type

- **character data type**
  - refers to the CHAR, VARCHAR, NCHAR, and NVARCHAR data types

- **coercible data type**
  - a data type that can be converted to multiple data types, not just a character data type

- **date/time data type**
  - refers to the DATE, TIME, and TIMESTAMP data types

- **non-coercible data type**
  - a data type that only can be converted to a character data type

- **numeric data type**
  - refers to the DECIMAL, DOUBLE (or FLOAT), REAL, BIGINT, INT, NUMERIC, SMALLINT, and TINYINT data types

- **standard character conversion**
  - if an expression is not one of the character data types, it is converted to a CHAR data type.

- **standard numeric conversion**
  - if an expression has a coercible, non-numeric data type, it is converted to a DOUBLE data type.
Overview of Type Conversions

Operands in an expression must be of the same general data type, numeric, character, binary, or date/time, in order for DS2 to resolve the expression. When it is necessary, DS2 converts an operand's data type to another data type, depending on the operands and operators in the expression. This process is called type conversion. For example, the concatenation operator ( || ) operates on character data types. In a concatenation of the character string 'First' and the numeric integer 1, the INTEGER data type for the operand 1 is converted to a CHAR data type before the concatenation takes place.

When an operand data type is converted within the same general data type, the operand data type is promoted. Operands with a data type of SMALLINT and TINYINT are promoted to INTEGER, and operands of type REAL are promoted to DOUBLE. Type promotion is performed for all operations on SMALLINT, TINYINT, and REAL, including arguments for method and function expressions.

Numeric and character data types are coercible. The BINARY, VARBINARY and the date/time data types DATE, TIME, and TIMESTAMP, are non-coercible and only can be converted to one of the character data types.

When DS2 evaluates an expression, if the data types of the operands match exactly, no type conversion or promotion is necessary and the expression is resolved. Otherwise, each operand must go through a standard numeric conversion or a standard character conversion, depending on the operator.

The results of a numeric or character expression are based on a data type precedence. If both operands have different types within the same general data type, the data type of the expression is that of the operand with the higher precedence, where 1 is the highest precedence. For example, for numeric data types, a data type of DOUBLE has the highest precedence. If an expression has an operand of type INTEGER and an operand of type DOUBLE, the data type of the expression is DOUBLE. A list of precedences can be found in the topics that follow, if applicable, for the different types of expressions.

For a table showing all type conversions, see Appendix 2, “DS2 Type Conversions for Expression Operands,” on page 1299.

Type Conversion for Unary Expressions

In unary expressions, such as +1 or -444, the standard numeric conversion is applied to the operand. The following table shows the data type for unary expressions:

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary plus</td>
<td>same as the operand or DOUBLE for converted operands</td>
</tr>
<tr>
<td>Unary minus</td>
<td>same as the operand or DOUBLE for converted operands</td>
</tr>
</tbody>
</table>

For a table showing all type conversions, see Appendix 2, “DS2 Type Conversions for Expression Operands,” on page 1299.
Type Conversion for Logical Expressions

In logical expressions, such as $a \land b$ or $(a \neq \text{start}) \lor (f = \text{finish})$, the standard numeric conversion is applied to each operand. The following table shows the precedence used to determine the data type of the expression, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Data Type of Either Operand</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>2</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>3</td>
<td>all other numeric data types</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Type Conversion for Arithmetic Expressions

In arithmetic expressions, such as $a<>b$ or $a + (b \times c)$, the standard numeric conversion is applied to each operand.

The following table shows the precedence used to determine the data type of arithmetic expressions for the addition, subtraction, multiplication, and division operators, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Data Type of Either Operand</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>2</td>
<td>DECIMAL, NUMERIC</td>
<td>DECIMAL, NUMERIC</td>
</tr>
<tr>
<td>3</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>4</td>
<td>all other numeric data types</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
The following table shows the data type for arithmetic expressions that use the min, max, and power operators:

**Table 12.4  Data Type Conversion for the Min, Max, and Power Operator Expressions**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Data Type</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>min or max</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>min or max</td>
<td>DECIMAL, NUMERIC</td>
<td>DECIMAL, NUMERIC</td>
</tr>
<tr>
<td>min or max</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>**</td>
<td>all numeric data types</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

---

**Type Conversion for Relational Expressions**

In relational expressions, such as \( x \leq y \) or \( i > 4 \), the standard conversion that is applied depends on the operand data types. The data type of the expression is always INTEGER, as shown in the following tables.

**Table 12.5  Data Type Conversion for Relational Expressions except IN Expressions**

<table>
<thead>
<tr>
<th>Order of Data Type Resolution</th>
<th>Data Type of Either Operand</th>
<th>Standard Conversion</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any numeric data type</td>
<td>numeric</td>
<td>INTEGER</td>
</tr>
<tr>
<td>2</td>
<td>CHAR, NCHAR</td>
<td>character</td>
<td>INTEGER</td>
</tr>
<tr>
<td>3</td>
<td>DATE, TIME, TIMESTAMP</td>
<td>none, data types must match</td>
<td>INTEGER</td>
</tr>
<tr>
<td>4</td>
<td>VARCHAR, VARCHAR</td>
<td>none, error returned</td>
<td>not applicable</td>
</tr>
</tbody>
</table>

**Table 12.6  Data Type Conversion for IN Expressions**

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Data Type of Either Operand</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>non-numeric</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>2</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>3</td>
<td>DECIMAL, NUMERIC</td>
<td>DECIMAL</td>
</tr>
</tbody>
</table>
Type Conversion for Concatenation Expressions

In concatenation expressions, such as `a || b` or `x ! y`, the standard character conversion is applied to each operand. The following table shows the precedence used to determine the data type of the expression, where 1 is the highest precedence and 2 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Data Type of Either Operand</th>
<th>Expression Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if either is type NCHAR</td>
<td>NCHAR</td>
</tr>
<tr>
<td>2</td>
<td>CHAR</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table 12.7  Data Type Conversion for Concatenation Expressions
What Is an Expression?

An expression is made of up of operands, and optional operators, that form a set of instructions and that resolves to a value.

An operand can be a single constant or variable, or it can be an expression. Operators are the symbols that represent either a calculation, comparison, or concatenation of operators.

The following are examples of DS2 expressions:

```
a = b * c
"col1"
e || 1 || z
a >= b**{c - 8}
```

```
system.put(a*5,hex.)
```
Overview of Expressions

The basic type of expression is a primary expression. Complex expressions combine expressions and operators. Other expressions invoke a DS2 construct, such as a method expression or a function expression. The system expression and the THIS expression refer to expressions that are global in scope. The IN expression returns a Boolean result based on whether the result of an expression is contained in a list.

Expression kinds commonly refer to a segment of code. For example, an AND expression refers to the AND operator and the operands that it processes. A binary expression refers to a binary constant or a hexadecimal constant such as $x'ff00effc'$.

Whether an expression is simple or complex, invokes a construct, or is global in scope, expressions of all kinds have a value and a data type.

Primary Expression

In their simplest form, primary expressions are numbers, character strings, binary and hexadecimal constants, literal values, date and time values, identifiers, and null values, as in these primary expressions:

```
a
"var_a"
5.33
'Company'
date '2007-08-24'
```

The following table shows basic primary expressions, their data type(s), and an example:

<table>
<thead>
<tr>
<th>Type of Expression</th>
<th>Short Description</th>
<th>Data Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Groups same type data</td>
<td>Same type as individual items in the array</td>
<td>a[5, b+c]</td>
</tr>
<tr>
<td>Binary</td>
<td>Binary and hexadecimal constants</td>
<td>VARBINARY</td>
<td>$x'FE'$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b'01000011'</td>
</tr>
<tr>
<td>Character</td>
<td>Character string</td>
<td>CHAR, VARCHAR</td>
<td>'New Report'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a='Stock';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b='Report';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c=a</td>
</tr>
<tr>
<td>National Character</td>
<td>National Character string</td>
<td>NCHAR, NVARCHAR</td>
<td>n'New Report'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a=n'Stock';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>b=n'Report';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Table 13.1 Data Types and Examples of Primary Expressions

94 Chapter 13 • DS2 Expressions
<table>
<thead>
<tr>
<th>Type of Expression</th>
<th>Short Description</th>
<th>Data Type</th>
<th>Example</th>
</tr>
</thead>
</table>
| Dot               | System, THIS, and package method expressions | The resolved type of the expression | `system.put(x,5.)`  
`this.s`  
`p.calc(2,6,9)`  |
| Date / time       | Date and time values | DATE, TIME, TIMESTAMP, or DOUBLE | `date '2007-01-01'` |
| Identifier        | Provides a name for various language elements or is a keyword | The declared or default type. The default is DOUBLE. | `a`  
`"part1"`  
`IN` |
| Integer           | Integer numbers   | INTEGER, BIGINT | 123 |
| New               | Instantiates a package method | The resolved type of the expression | `a=new_package_name();`  
`*` |
| Numeric           | Integer, real, and floating point numbers, or a missing or null value | DOUBLE, FLOAT, REAL | 5  
`4.3` |
| Null              | Null expression   | none | NULL |
| Parentheses       | Operator and operands enclosed in parentheses for higher evaluation precedence | The resolved type of the expression enclosed in parentheses. | `(a + b) - c`  
`*` |

* For the purpose of the example, the primary expression is contained in an expression or an assignment statement. The example expression is highlighted.

**Complex Expression**

A complex expression combines expressions and operators to create a more expansive expression, as in these expressions:

```
    a + b * -c - 5
    a = b = 5
    x | y & a < c * d
    x**y**z - 9 >= f
    z >< c + e <= u**y + 10
    x || c + 7
    a in (1,2,3)
```

Evaluation of complex expressions is based on the operator order of precedence, as shown in Table 13.5 on page 107, and the data types of the primary expressions. Before any calculations can be done, operand data types must be the same general data type: numeric, character, binary, or date/time. If the data types are the same, processing can proceed. If they are not the same, the operand data types are converted based on the operator and the data type of the operands.
In the expression \( a + \frac{b}{-c} - 5 \), assume that \( a \) is 1.35 with a type of DOUBLE, \( b \) is 2 with a data type of INTEGER, and \( c \) is 3 with a data type of INTEGER. \( \frac{b}{-c} \) or \( \frac{2}{-3} \) evaluates to INTEGER 0. The INTEGER 0 is converted to a DOUBLE 0 before being added to \( a \). Then \( a + 0.0 \) evaluates to DOUBLE 1.35. The INTEGER 5 is converted to a DOUBLE 5 before the addition of the DOUBLE 1.35. The final result of the expression is DOUBLE –3.65.

For information about data type conversion, see Chapter 12, “DS2 Type Conversions,” on page 87.

In the expression \( a = b = 5 \), if \( b \) is a value other than 5, then \( b = 5 \) is evaluated to 0. Therefore, \( a \) is assigned a value of 0. The first equal sign (=) is an assignment operator and the second equal sign is a logical equality operator. For more information, see “Example 2: Using an Expression with Multiple Equals Signs” on page 868.

Note: DS2 supports using \( eq \) as well as the equal sign.

Array Expression

An array expression is a primary expression that represents a grouping of data items of the same data type. Although an array can have multiple dimensions, individual data item values are scalar values. Data items are accessed by specifying an index into the array.

The array expression consists of an array identifier followed by an array index expression for each dimension in the array, as in this syntax:

\[
\text{array-identifier} \{ \text{index-expression} \ [ \ , \ \cdots \ , \ 	ext{index-expression} \ ] \} \\
\]

Note: Brackets in the syntax convention indicate optional syntax. The escape character (\) before a bracket indicates that the bracket is required for the syntax. Indexes in an array expression must be contained by brackets ([ ]).

The array identifier can be either a declared array variable or a variable used in a THIS expression. The index expression is a primary expression that resolves to an integer.

The following are examples of array expressions:

\[
a[i] \\
\mathtt{a[}j + 2, k-3\mathtt{]} \\
\mathtt{this.c[}2, \mathtt{vwind, a[i]}\mathtt{]} \\
\]

When an array is declared, the index values specify the boundaries for the array. If an index expression is beyond the boundaries of the array, DS2 issues an error.

The value of an array expression is the value of the indexed value in the array. For example, if the array values are \( a[1] = 12, a[2] = 15, \) and \( a[3] = 20 \), the value of the array expression \( a[2] \) is 15.

Note: Arrays are 1-based. The array index starts at 1.

Function Expression

A function expression invokes a function within a DS2 program. To invoke a function, use this syntax:

\[
\text{function-name} \ ( \ [ \ 	ext{argument} \ [ \ , \ \cdots \ , \ \text{argument}] \ ] \ ) \\
\]

Functions might require arguments. If the function expression contains arguments, the argument data types are converted, if necessary, to the data types of the function signature, which is the argument order and data type for a function. Parentheses in the
function call are required, whether the function takes arguments or no arguments are required. For example, the TIME function does not require arguments:

```
t = time();
```

A function expression resolves to the value returned by the function. In the function expression above, `time()` resolves to the current time of day.

Methods and functions are similar. Functions have global scope. Methods are programming blocks and have local scope.

If the name of a function is identical to a method name, DS2 invokes the method. Functions with the same name as a method can be invoked only by using a SYSTEM expression. For more information, see “SYSTEM Expression” on page 101.

For a list of DS2 functions, see Chapter 23, “DS2 Functions,” on page 335.

**IN Expression**

An IN expression determines whether an expression is contained in a constant list. See “Constant List” on page 70.

The following is the syntax of an IN expression:

```
expression [ not-operator ] IN constant-list
```

*Note:* Any valid data type for your data source can be used in `constant-list`. If any argument is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already). If any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already). Otherwise, all arguments are converted to a BIGINT. The result is always INTEGER, either 0 or 1.

Any of the NOT operators (∼, ^, or NOT) are valid before the IN operator, which results in the logical negation of the expression value.

The following table shows the results of some IN expressions:

<table>
<thead>
<tr>
<th>Input Values</th>
<th>IN Expression</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = 2</code></td>
<td><code>a in (5,34,2,67)</code></td>
<td>1</td>
</tr>
<tr>
<td><code>b = 3</code></td>
<td><code>b not in (3,22,43,65)</code></td>
<td>0</td>
</tr>
</tbody>
</table>

**LIKE Expression**

**Overview of the LIKE Expression**

A LIKE expression determines whether a character string matches a pattern-matching specification.

Here is the syntax of a LIKE expression:

```
expression [ NOT] LIKE pattern-matching-expression [ESCAPE character-expression]
```

The expressions can be any character string or binary string data type.
If \textit{expression} matches the pattern specified by \textit{pattern-matching-expression}, a value of 1 (true) is returned. Otherwise, a value of 0 (false) is returned.

NOT LIKE returns the inverse value of LIKE. For example, if \texttt{x like y} is true, then \texttt{x not like y} is false.

The ESCAPE argument is used to search for literal instances of the percent (\%) and underscore (\_) characters, which are usually used for pattern matching.

\textbf{Patterns for Searching}

Patterns consist of three classes of characters.

\textbf{Table 13.3 Pattern-matching Characters}

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>underscore (_)</td>
<td>matches any single character</td>
</tr>
<tr>
<td>percent sign (%)</td>
<td>matches any sequence of zero or more characters</td>
</tr>
<tr>
<td>Note: Be aware of the effect of trailing blanks. To match values, you might have to use the TRIM function to remove trailing blanks.</td>
<td></td>
</tr>
<tr>
<td>any other character</td>
<td>matches that character</td>
</tr>
</tbody>
</table>

\textbf{Searching for Literal \% and \_}

Because the \% and \_ characters have special meaning in the context of the LIKE expression, you must use the ESCAPE argument to search for these character literals in the input character string.

These examples use the values \texttt{app, a\_, a\_\_, bbaal, and ba\_l}.

- The condition \texttt{like 'a\_%'} matches \texttt{app, a\_, and a\_\_}, because the underscore (\_) in the search pattern matches any single character (including the underscore), and the percent (\%) in the search pattern matches zero or more characters, including '%\_' and '_\_'.
- The condition \texttt{like 'a\_^%'} escape '^' matches only \texttt{a\_\_}, because the escape character (\^) specifies that the pattern search for a literal '%\_'.
- The condition \texttt{like 'a\_\%'} escape '_' matches none of the values, because the escape character (_) specifies that the pattern search for an 'a' followed by a literal '%\_', which does not apply to any of these values.

\textbf{Searching for Mixed-case Strings}

The DS2 LIKE expression is case sensitive. To search for mixed-case strings, use the UPCASE function as the following example shows:

\texttt{upcase(str) like 'SM\%'}

\textbf{LIKE Expression Examples}

The following table shows examples of the matches that would result when searching these strings: Smith, Smooth, Smothers, Smart, Smuggle.
### Table 13.4  Examples of LIKE Expressions

<table>
<thead>
<tr>
<th>LIKE Expression Example</th>
<th>Matching Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str like 'Sm%'</code></td>
<td>Smith, Smooth, Smothers, Smart, Smuggle</td>
</tr>
<tr>
<td><code>str like '%th'</code></td>
<td>Smith, Smooth</td>
</tr>
<tr>
<td><code>str LIKE 'S__gg%'</code></td>
<td>Smuggle</td>
</tr>
<tr>
<td><code>str like 'S_o'</code></td>
<td>(no matches)</td>
</tr>
<tr>
<td><code>str like 'S_o%'</code></td>
<td>Smooth, Smothers</td>
</tr>
<tr>
<td><code>str like 'S%th'</code></td>
<td>Smith, Smooth</td>
</tr>
<tr>
<td><code>str not like 'Z'</code></td>
<td>Smith, Smooth, Smothers, Smart, Smuggle</td>
</tr>
</tbody>
</table>

### Method Expression

A method expression invokes a method that has been defined by the METHOD statement.

To invoke a method, use this syntax:

```
method-name ( [ expression [ , … expression ] ]
```

Methods are invoked based on the method name and signature. DS2 first identifies the method name. If a method name is identical to a function name, DS2 invokes the method. A function with the same name as a method can be invoked by using a SYSTEM expression. For more information, see “SYSTEM Expression” on page 101.

Because DS2 allows overloaded methods, DS2 invokes the method whose arguments best match the number of arguments and the argument data types in the method signature, which is the argument order and data type for the method. The best match will be the one for which the number of method parameters is equal to the number of arguments, and such that no other method signature has as many exact parameter type matches for the given argument list. If a best match is not found, an error occurs.

Once the method to execute is identified, DS2 converts argument data types to the data type of the corresponding method parameter, if necessary. The method then executes.

A method expression resolves to the value returned by the method.

In the following example, methods CONCAT and ADD are defined and then invoked in the INIT() method. The highlighted expressions in the INIT() method are method expressions:

```
method concat(char(100) x, char(100) y) returns char(200);
    return trim(x)|| y;
end;

method add(double x, double y) returns double;
    return x + y;
end;
```
method init();
    dcl char(200) r;
    r = concat('abc', 'def');
    d = add(100, 101);
end;

In this next example, the D method is an overloaded method. DS2 must compare the
method expression arguments to the method signatures to find the best match:

method d(double x, double y) returns double;
    return x + y;
end;

method d(int x, int y) returns int;
    return x + y;
end;

method init();
    dcl double r;
    dcl int i;
    r = d(1.2345, 5.6789);
    i = d(99, 100);
end;

The first method calls the D method whose signature requires values with DOUBLE data
types. The second method calls the D method whose signature requires values with
INTEGER data types.

This final example shows that DS2 cannot determine whether the values in the method
expression have a data type of INTEGER or DOUBLE. Because it is ambiguous, DS2
issues an error:

method d(int x, double y) returns double;
    return x + y;
end;

method d(double x, int y) returns double;
    return x + y;
end;

method run();
    d = d(100, 102);
end;

For more information, see the “METHOD Statement” on page 910.

**Package Method Expression**

A package method expression instantiates a method that is defined in a package. To
invoke a package method expression, use this syntax:

```
```

Package method expressions execute in a manner similar to method expressions. That is,
once DS2 has determined that the package and the method exist, the best match of
method signatures is determined, argument data types are converted if necessary, and the
method executes.

In the following example, the highlighted expressions are package method expressions:
declare package myadd a1();
a1.sale(3,4);
a1.add(1,2);
a2.bonus(5,12);
a2.add(10,20);

The first two package method expressions invoke the SALE and ADD methods in the A1 package, which was instantiated from the MYADD package. The last two package method expressions invoke the BONUS and ADD methods in the A2 package.

Note: You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement. For more information, see “Using DS2 Packages in Expressions” in SAS FedSQL Language Reference.

For information about packages, see the “PACKAGE Statement” on page 921 and Chapter 16, “DS2 Packages,” on page 135.

**SYSTEM Expression**

When a method and a function have identical names, the method call takes precedence over the function call. The function can then only be invoked by using a SYSTEM expression.

To invoke a SYSTEM expression, use this syntax:

```
SYSTEM.function-expression
```

A SYSTEM expression prepends a function expression with the dot notation, `system.`. For example, if SUM is the name of a method as well as the name of a function, the SUM function only can be invoked by using the SYSTEM expression:

```
system.sum(a,b,c)
```

**THIS Expression**

A THIS expression provides an alternate method to simultaneously declare and use a global scalar variable from anywhere within a DS2 program. A THIS expression is used to circumvent the standard variable lookup. In a THIS expression, DS2 searches for a scalar variable declaration of the identifier in global scope. If there is no such declaration, DS2 declares the identifier in global scope with DOUBLE type. Global variables can be referenced by all programming blocks in a DS2 program.

To invoke a THIS expression, use this syntax:

```
THIS.variable-name
```

A THIS expression prepends a variable with the dot notation, `THIS`.

**Note:** DS2 stores `THIS.variable-name` only as `variable-name`. If you have a local variable with the same name as the global scalar variable and DS2 issues a diagnostic message about the variable, you will not be able to distinguish which variable is a problem. For example, if DS2 issues a warning message that `x` is not declared, you would not know whether the message refers to the global variable, `THIS.x`, or the local variable, `x`.

In the following example, the variable `s` becomes a global variable by using a THIS expression:

```
method init();
    this.s = sum(a,b,c,d,e);
end;
```

In the following example, the variable `s` becomes a global variable by using a THIS expression:

```sql
method run();
    this.s = sum(a,b,c,d,e);
end;
```
The THIS expression provides a method to access a global variable that is hidden by a local variable with the same name. Here is an example.

data;
declare double x; /* declare global x */

method run();
declare double x; /* declare local x */
/* Two variables exist with same name, "x". */
/* Identifier "x" refers to local x in scope of run method. Global x is hidden */
/* by local x. */
this.x = 1.0; /* assign 1.0 to global x */
x = 0.0; /* assign 0.0 to local x */
output; /* global x with 1.0 output */
end;
enddata;
run;

**IF Expression**

**Overview of the IF Expression**
The conditional IF expression is used to select between two values based on whether a conditional expression evaluates to true (a nonzero value) or false (zero).

To invoke an IF expression, use this syntax:

```
IF expression-1 THEN expression-2 ELSE expression-3
```

If `expression-1` is a nonzero value, the result of the IF expression is the value of `expression-2`. Otherwise, the result of the IF expression is the value of `expression-3`. Here is an example.

```
m=(if missing(u) then 0 else u);
```

The IF expression can be used wherever any other expression can be used. The precedence of an IF expression is lower than the arithmetic and logic operators, Therefore, parentheses are necessary in mixed expressions like this one.

```
r = 25.5 + (if sum < 15 then -a else b*2);
```

Without the parentheses, the plus (+) operator would be evaluated first resulting in a parse error from the subexpression `25.5 + if`.

**Nested IF Expressions**
IF expressions can be nested to select between many values for a multi-way decision.

```
IF condition-expression-1 THEN result-expression-1
ELSE IF condition-expression-2 THEN result-expression-2
...
ELSE IF condition-expression-n THEN result-expression-n
ELSE result-expression-default
```

102 Chapter 13 • DS2 Expressions
The condition expressions are evaluated in order. The result of the nested IF expression chain is the associated result-expression of the first condition-expression that evaluates to true (a nonzero value). If all the condition-expressions evaluate to false (zero), the result of the IF expression is the result-expression-default. Here is an example.

```sql
grade = if score >= 90 then 'A'
    else if score >= 80 then 'B'
    else if score >= 70 then 'C'
    else if score >= 60 then 'D'
    else if score >= 0  then 'F'
    else NULL;
```

**IF Expression Data Type**

The data type of an IF expression is determined by examining the type of the first result expression, `expression-2`.

**IF** `expression-1` **THEN** `expression-2` **ELSE** `expression-3`

If `expression-2` is not a numeric data type, then the IF expression is assigned the type of `expression-2`.

If `expression-2` is a numeric data type, then the IF expression is assigned the wider numeric data type of `expression-2` and `expression-3`. For example, if `expression-2` is an SMALLINT and `expression-3` is a DOUBLE, then the IF expression is assigned type DOUBLE. If `expression-2` is a numeric data type and `expression-3` is not a numeric data type, then the IF expression is assigned the type of `expression-2`.

If the first result expression in a nested IF expression chain is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the nested IF expression chain. In the following example, `t` is a TINYINT, `b` is a BIGNINT, and `d` is a DECIMAL(10,5). The 0 in the ELSE is assigned type BIGINT. Therefore, the nested IF expression chain is assigned the type decimal(10,5), the widest numeric type of TINYINT, BIGNINT, and DECIMAL(10,5).

```sql
r = if n < 0 then t
    else if n = 0 then b
    else if n > 0 then d
    else 0;
```

*Note:* If 0.0 had been used for the ELSE value instead of 0, then the ELSE result would have been assigned type DOUBLE instead of type BIGINT. With the type DOUBLE ELSE expression, the widest numeric type of the result expressions would be type DOUBLE. Therefore, the nested IF expression chain would be assigned type DOUBLE instead of type decimal(10,5).

**Lazy Evaluation of IF Expressions**

The IF expression uses lazy evaluation for the result expressions.

**IF** `expression-1` **THEN** `expression-2` **ELSE** `expression-3`

For example, you could use this code to check for division by zero.

```sql
a = if c ne 0 then b/c else null;
```

Expression `expression-1` is always evaluated, but only one of `expression-2` or `expression-3` is evaluated. The expression that is not selected as the result of the IF expression is not evaluated. Thus, if `expression-1` is a nonzero value (true), then only `expression-2` is evaluated. If `expression-1` is zero (false), then only `expression-3` is evaluated.

Lazy evaluation also applies to the result expressions of nested if expression chains.
IF condition-expression-1 THEN result-expression-1
ELSE IF condition-expression-2 THEN result-expression-2
... 
ELSE IF condition-expression-n THEN result-expression-n
ELSE result-expression-default

The selected result expression is the only result expression evaluated. Lazy evaluation also applies to the condition expressions. Condition expressions are evaluated in order until a condition evaluates to true (nonzero) or all conditions are evaluated. If the IF expression has \( n \) condition expressions and the \( i \)th condition is the first nonzero condition, then only the first 1 to \( i \) conditions are evaluated. The \( i+1 \) to \( n \) conditions are not evaluated.

**SELECT Expression**

**Overview of the SELECT Expression**

A SELECT expression is used to select between multiple expressions based on the values of other expressions.

To invoke a SELECT expression, use this syntax:

```
SELECT [(select-expression)]
    WHEN (when-expression) […]WHEN (when-expression) result-expression
    […]WHEN (when-expression) […]WHEN (when-expression) result-expression
    OTHERWISE [default-result-expression]
END
```

The SELECT expression evaluates each WHEN expression in order until a matching expression is found. Then the associated result-expression is evaluated as the result of the SELECT expression.

The SELECT expression can be used wherever any other expression can be used. Here is an example.

\[
r = 25.5 + \text{select} (t) \text{ when (1) } -a \text{ when (3) } b*2 \text{ end;}
\]

**SELECT Expression with a Selection Expression**

If a selection expression is present, then it is evaluated. Then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to the same value as the selection expression. If all the WHEN expressions evaluate to different values than the selection expression, the result of the SELECT expression is the default result expression if present. Otherwise, it is a missing or null value. Here is an example.

\[
s = \text{select} (t)
    \text{ when (1) } x*10
    \text{ when (3) } x
    \text{ when (5) } x*100
    \text{ when (0) } 0
    \text{ otherwise .}
end;
\]

If \( t \) is 5, then the SELECT expression evaluates to 'x*100'.

If \( t \) is 5, then the SELECT expression evaluates to 'x*100'.
**SELECT Expression without a Selection Expression**

If a selection expression is not present, then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to true (a nonzero value). If all the WHEN expressions evaluate to false (zero), the result of the select expression is the default result expression if present. Otherwise, it is a missing or null value. Here is an example.

```plaintext
grade = select
  when (score >= 90) 'A'
  when (score >= 80) 'B'
  when (score >= 70) 'C'
  when (score >= 60) 'D'
  when (score >= 0 ) 'F'
end;
```

If `score` is 76, then the first `when-expression` to evaluate to true is `score>=70`. The `select-expression` evaluates to 'C'.

**Optional Otherwise Expression**

If an otherwise default result expression is not supplied, then DS2 provides a default result value to select when none of the WHEN expressions are selected. If the SELECT expression has type DOUBLE or CHAR in SAS mode, the default result value is a missing value (.). For all other data types in either mode, the default result value is NULL.

**Result Expression with Multiple When Expressions**

Multiple WHEN expressions can be associated with a single result expression. The WHEN expressions are listed consecutively followed by the single result expression. If any of the WHEN expressions associated with a result expression is the first matching WHEN expression, then the result of the SELECT expression is the result expression.

For example, the following SELECT expression evaluates to 'airplane' if the value of variable `t` is either 'A', 'a', 'P', or 'p'.

```plaintext
s = select (t)
  when ('A')
  when ('a')
  when ('P')
  when ('p') 'airplane'
  when ('C')
  when ('c') 'car'
  when ('T')
  when ('t') 'train'
  otherwise 'walk'
end;
```

**SELECT Expression Data Type**

The type of a SELECT expression is determined by examining the type of the first result expression. If the first result expression is not a numeric data type, then the SELECT expression is assigned the type of the first result expression.

If the first result expression is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the SELECT expression.
In the following example, \( t \) is a TINYINT, \( b \) is a BIGINT, \( d \) is a DECIMAL\((10,5)\), and \( s \) is a CHAR\( (10) \). The select expression is assigned the type DECIMAL\((10,5)\), the widest numeric type of TINYINT, BIGINT, DECIMAL\((10,5)\), and CHAR\( (10) \).

\[
r = \text{select (t)} \\
  \text{when (1) t} \\
  \text{when (2) b} \\
  \text{when (3) d} \\
  \text{otherwise s} \\
\text{end;}
\]

*Note:* If \( s \) had been assigned to the first result expression, then the type of the SELECT expression would have been CHAR\( (10) \). If the first result expression has a non-numeric type, then the non-numeric type is assigned as the type of the SELECT expression.

**Lazy Evaluation of the SELECT Expression**

The SELECT expression uses lazy evaluation for the WHEN expressions. The WHEN expressions are evaluated in order until a matching WHEN expression is found or all when expressions are evaluated. If the SELECT expression has \( n \) WHEN expressions and the \( i \)th WHEN expression is selected, then only the first 1 to \( i \) WHEN expressions are evaluated. The \( i+1 \) to \( n \) when expressions are not evaluated.

Lazy evaluation also applies to the result expressions of the SELECT expression. The selected result expression is the only result expression evaluated.

In the following example, if \( n[i] \) equals 0, then only the first two WHEN expressions (\( n[i] < 0 \) and \( n[i] = 0 \)) are evaluated and only the second result expression (\( y\times10-r2 \)) is evaluated.

\[
m(\text{select} \\
  \text{when (n[i] < 0) y\times100-r1} \\
  \text{when (n[i] = 0) y\times10-r2} \\
  \text{when (n[i] > 0) y\times10} \\
  \text{end});
\]

---

**Operators in Expressions**

**Operator Precedence**

An operator symbolizes a type of operation that is to be performed on an operand, such as addition, comparison, and logical negation. When an expression contains multiple operators and operands, DS2 resolves the expression by using operator precedence. Operations are performed from the highest order of precedence to the lowest order of precedence.

The highest order of precedence is 1 and the lowest order of precedence is 9. Within a precedence level, with the exception of exponentiation, minimum, and maximum operators, operators associate from left to right. The exponentiation, minimum, and maximum operators associate from right to left.

By using the precedence order in Table 13.5 on page 107, in the expression \( 5+a^{*}b*3 \), \( a^{*}b \) is calculated first and then multiplied by 3, and that result is added to 5.

*Tip:* In DS2, \( x < y < z \) is evaluated like \( x < y \) and \( y < z \).
The following table lists the operators and their order of precedence:

Table 13.5  Operators and Their Order of Precedence in DS2 Expressions

<table>
<thead>
<tr>
<th>Order of Precedence</th>
<th>Symbol</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( )</td>
<td>left to right</td>
</tr>
<tr>
<td>1</td>
<td>SELECT expression</td>
<td>left to right</td>
</tr>
<tr>
<td>2</td>
<td>+, –</td>
<td>right to left</td>
</tr>
<tr>
<td>2</td>
<td>^ or ~</td>
<td>left to right</td>
</tr>
<tr>
<td>2</td>
<td>**, &lt;&gt;, &gt;=&lt;</td>
<td>left to right</td>
</tr>
<tr>
<td>3</td>
<td>*, /</td>
<td>left to right</td>
</tr>
<tr>
<td>4</td>
<td>+, –</td>
<td>left to right</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>..</td>
<td>left to right</td>
</tr>
<tr>
<td>6</td>
<td>IN, LIKE</td>
<td>left to right</td>
</tr>
<tr>
<td>7</td>
<td>=, ^= or ~=</td>
<td>right to left</td>
</tr>
<tr>
<td>7</td>
<td>&gt;=, &lt;=, &gt;, &lt;</td>
<td>left to right</td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>or !</td>
</tr>
<tr>
<td>10</td>
<td>IF expression</td>
<td>right to left</td>
</tr>
<tr>
<td>none</td>
<td>:=</td>
<td>none</td>
</tr>
<tr>
<td>none</td>
<td><em>NEW</em></td>
<td>none</td>
</tr>
</tbody>
</table>

* In DS2, $x < y < z$ is evaluated like $x < y$ and $y < z$.

**Expression Values by Operator**

The following table shows the resolved value for expressions that are based on an operator:
### Table 13.6 Expression Values by Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary expressions</strong></td>
<td></td>
</tr>
<tr>
<td>Unary plus</td>
<td>Is the same as the expression operand</td>
</tr>
<tr>
<td>Unary minus</td>
<td>Is the arithmetic negation of the operand</td>
</tr>
<tr>
<td>NOT or ^ or ~</td>
<td>If the operand is nonzero, result is 0. If the operand is zero or missing, result is 1. If the operand is null, result is null.</td>
</tr>
<tr>
<td><strong>Logical expressions</strong></td>
<td></td>
</tr>
<tr>
<td>OR or</td>
<td>or ! • the logical OR of the two operands • null when one operand is zero or missing and the other operand is null • null when both operands are null • 1 when either operand is nonzero (even if the other operand is null or missing) • 0 when both operands are zero or missing</td>
</tr>
<tr>
<td>AND or &amp; • the logical AND of the two operands • null when one operand is nonzero and the other operand is null • null when both operands are null • 1 when both operands are nonzero • 0 when either operand is zero or missing (even if the other operand is null)</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic expressions</strong></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>the arithmetic sum of the operands</td>
</tr>
<tr>
<td>−</td>
<td>the arithmetic difference of the operands</td>
</tr>
<tr>
<td>*</td>
<td>the arithmetic product of the operands</td>
</tr>
<tr>
<td>/</td>
<td>the arithmetic quotient of the operands</td>
</tr>
<tr>
<td>**</td>
<td>the left operand raised to the power of the right operand</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>the minimum of the left and right operands</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>the maximum of the left and right operands</td>
</tr>
<tr>
<td>any arithmetic operator</td>
<td>null when either or both operators are null</td>
</tr>
<tr>
<td><strong>Relational expressions</strong></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>1 when the left operand is less than the right operand; otherwise, 0</td>
</tr>
<tr>
<td>Operator</td>
<td>Value</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>&gt;</td>
<td>1 when the left operand is greater than the right operand; otherwise, 0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1 when the left operand is less than or equal to the right operand; otherwise, 0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>1 when the left operand is greater than or equal to the right operand; otherwise, 0</td>
</tr>
<tr>
<td>=</td>
<td>1 when the left operand is equal to the right operand; otherwise, 0</td>
</tr>
<tr>
<td>^=</td>
<td>1 when the left operand is not equal to the right operand; otherwise, 0</td>
</tr>
<tr>
<td>*</td>
<td>null when either or both operators are null</td>
</tr>
</tbody>
</table>

**Concatenation expression**

|| or !! | the left operand merged with the right operand to form a new value |
| ..      | strips each argument before concatenating |

**IF expression**

if | 1 when the comparison expression is contained in the constant list |

**IN expression**

in | 1 when the comparison expression is contained in the constant list |

**LIKE expression**

LIKE | 1 when the comparison expression is contained in the constant list |

**SELECT expression**

SELECT | 1 when the comparison expression is contained in the constant list |

* The || concatenation operator does not remove spaces. You can use the TRIM function to remove trailing spaces. However, the .. operator performs the same function as TRIM followed by concatenation. It is faster to use the .. operator (a .. b) than to use the TRIM function (TRIM(a) || TRIM(b)).
Chapter 14
Daten and Times in DS2

DS2 Dates, Times, and Timestamps

Overview of DS2 Dates, Times, and Timestamps
Declaring Date, Time, and Timestamp Variables
DS2 Date, Time, and Timestamp Values
Operations on DS2 Dates and Times
SAS Date, Time, and Datetime Values
Converting SAS Date, Time, and Datetime Values to a DS2 Date, Time, or Timestamp Value
Converting DS2 Date, Time, and Timestamp Values to SAS Date, Time, or Datetime Values
Date, Time, and Datetime Functions
Date, Time, and Datetime Formats

DS2 Dates, Times, and Timestamps

Overview of DS2 Dates, Times, and Timestamps
DS2 supports the SQL style date and time conventions that are used in other data sources. When your data source is a not a SAS data set, DS2 can process dates and times that have a data type of DATE, TIME, and TIMESTAMP.

Date and time values with a data type of DATE, TIME, and TIMESTAMP can be converted to a SAS date, time, or datetime value, but DS2 cannot convert a SAS date, time, or datetime value to a value having a DATE, TIME, or TIMESTAMP data type.

DS2 provides date and time functions that convert any date or time value to SAS date, time, and datetime values, and back again to a recognizable date or time value. For more information, see “Date, Time, and Datetime Functions” on page 115.

The date and time intervals that are supported in ANSI SQL are not supported in DS2.

Declaring Date, Time, and Timestamp Variables

You declare a date, time, or timestamp variable by using the DATE, TIME, or TIMESTAMP data types in the DECLARE statement, as in this example:
dcl date dt;
dcl time tm;
dcl timestamp tmstmp;

Note: If you use a precision when you declare a time or timestamp variable, the time or timestamp values are not rounded to the specified precision until they are output by the DATA statement. Internally, the time or timestamp constant values are simply copied to the time or timestamp variable.

For additional information about the DS2 date and time data types, see Chapter 9, “DS2 Data Types,” on page 71.

DS2 Date, Time, and Timestamp Values

Once you declare a date, time, or timestamp variable, the value of the variable can be only a DS2 date, time, or timestamp constant that has the following syntax:

DATE 'yyyy-mm-dd'
TIME 'hh:mm:ss[.fraction]'
TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.fraction]'

where

- *yyyy* is a four-digit year
- *mm* is a two-digit month, 01–12
- *dd* is a two-digit day, 01–31
- *hh* is a two-digit military hour, 00–23
- *nn* is a two-digit minute, 00–59
- *ss* is a two-digit second, 00–60
- *fraction* can be one to nine digits, 0–9, is optional, and represent a fraction of a second

The string portion of the value after the DATE, TIME, or TIMESTAMP keyword must be enclosed in single quotation marks.

In the date constant, the hyphens are required and the length of the date string must be 10.

In the time constant, the colons are required. If the fraction of a second is not present, the time string must be 8 characters long and exclude the period. DS2 issues an error if the period is present without a fraction. If the fraction of second is present, the fraction can be up to 9 digits long and the time string can be up to 18 characters long (including the period).

In the timestamp constant, the hyphens in the date are required as well as the colons in the time. If the fraction of a second is not present, the timestamp string must be 19 characters long and exclude the period. If the fraction of a second is present, the fraction can be up to 9 digits long and the timestamp string can be up to 29 characters long.

The following are examples of DS2 date, time, and timestamp constants:

date '2012-01-31'
time '20:44:59'
timestamp '2012-02-07 07:00:00.7569'
Operations on DS2 Dates and Times

The only operations that can be performed on DATE, TIME, and TIMESTAMP values are operations that use the relational operators <, >, <=, >=, =, ^=, and IN, such as in the following statement:

```plaintext
if tm in(time'10:22:31', time'12:55:01') then
  if tm < time'13:30:00' then put 'Early afternoon';
  else put 'Time not available';
```

DS2 does not calculate date and time intervals on values that have the data types of DATE, TIME, and TIMESTAMP.

SAS Date, Time, and Datetime Values

A SAS date value is the number of days between January 1, 1960 and a specified date. Dates before January 1, 1960 are negative numbers; dates after are positive numbers. For example, the SAS date value for January 1, 1960 is 0, -365 for January 1, 1959, and 17532 for January 1, 2008.

A SAS time value is the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

A SAS datetime value is the number of seconds between January 1, 1960 and a specific hour, minute, and second of a specific date.

DS2 supports SAS date, time, and datetime values as DOUBLE types. When a numeric column is input from a SAS data set and the numeric column has a SAS date, time, or datetime format, the column is processed as a type DATE, TIME, or TIMESTAMP. If the numeric column in a SAS data set does not have a format or has a format that is not a SAS date, time, or datetime format, the column is processed as type DOUBLE.

All calculations on dates and times are done as a SAS date value, a SAS time value, or a SAS datetime value. For more information, see “Date, Time, and Datetime Functions” on page 115.

After calculations are complete, there are other functions that can then format the SAS date, time, and datetime values to recognizable date and time formats.

Converting SAS Date, Time, and Datetime Values to a DS2 Date, Time, or Timestamp Value

SAS date, time, and datetime values can be converted to DS2 dates, time, and timestamp values by using the TO_DATE, TO_TIME, and TO_TIMESTAMP functions. The argument of these functions is any value or expression that represents a SAS date, time, or datetime value and has a type DOUBLE. You can then use either the PUT statement or a format in the DECLARE statement to format the date, time, or timestamp value.

Here is an example.

```plaintext
data _null_;  
dcl date ds2d having format YYMMDD10.;  
dcl time ds2t having format TIME18.9;
```
Converting DS2 Date, Time, and Timestamp Values to SAS Date, Time, or Datetime Values

DS2 date, time, and timestamp values can be converted to a SAS datetime value by using the TO_DOUBLE function. This function converts the date, time, or timestamp CHAR or NCHAR string to a SAS datetime value with a data type of DOUBLE. You can then use any DS2 format to display the value in a date, time, or datetime format.

The following DS2 program illustrates how you can convert a DS2 timestamp to a SAS date, time, and datetime values:

data _null_
method run()

dcl timestamp DS2ts;
dcl double sasdtval sasd sastm;
dcl char(28) fmtdate fmttime fmtdt;
DS2ts = timestamp '2012-06-04 10:54:34.012';
put DS2ts;
sasdtval = to_double(DS2ts);
sasd = datepart(sasdtval);
sastm = timepart(sasdtval);
put sasdtval:best16.7;
put sasd:best.;
put sastm:best.;
fmtdate = put(sasd, yymmdd10.);
fmttime = put(sastm, time.);
fmtdt = put(sasdtval, datetime21.7);
put fmtdate=;
put fmttime=;
put fmtdt=;
end;
enddata;
run;
The following output is written to the SAS log:

<table>
<thead>
<tr>
<th align="right">2012-06-04 10:54:34.012000000</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">1654426474.012</td>
</tr>
<tr>
<td align="right">19148</td>
</tr>
<tr>
<td align="right">39274.012</td>
</tr>
<tr>
<td align="right">fmtdate=2012-06-04</td>
</tr>
<tr>
<td align="right">fmttime=10:54:34</td>
</tr>
<tr>
<td align="right">fmtdt=04JUN12:10:54:34.0120</td>
</tr>
</tbody>
</table>

In this example, a SAS date value is formatted to look like DS2 date value, but it has a data type of DOUBLE and not DATE. SAS date and time values cannot be assigned to DS2 date or time variables. Their data types are different. If you attempt to assign a SAS date or time value to a DS2 date or time variable, DS2 issues a data type invalid conversion error.

For more information, see the “PUT Function” on page 725. For a complete list of formats, see “Date, Time, and Datetime Formats” on page 116.

---

**Date, Time, and Datetime Functions**

In order to perform date and time calculations, DS2 date and time functions do the following:

- convert a date or time into a SAS date, time, or datetime value
- convert a SAS date, time, or datetime value into a recognizable date or time
- extract a date or a time from a SAS datetime value

The following tables list the date and time functions and what they do. For specific information about any of these functions, see Chapter 23, “DS2 Functions,” on page 335.

**Table 14.1  Functions That Convert Dates and Times into a SAS Date, Time, or Datetime Value**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE or TODAY</td>
<td>Returns the current date as a SAS date value</td>
</tr>
<tr>
<td>DATEJUL</td>
<td>Converts a Julian date to a SAS date value</td>
</tr>
<tr>
<td>DHMS</td>
<td>Returns a SAS datetime value from date, hour, minute, and second values</td>
</tr>
<tr>
<td>HMS</td>
<td>Returns a SAS time value from hour, minute, and second values</td>
</tr>
<tr>
<td>MDY</td>
<td>Returns a SAS date value from month, day, and year values</td>
</tr>
<tr>
<td>TIME</td>
<td>Returns the current time of day as a SAS time value.</td>
</tr>
<tr>
<td>YYQ</td>
<td>Returns a SAS date value from a year and quarter year values</td>
</tr>
</tbody>
</table>
Table 14.2 Functions That Format a SAS Date, Time, or Datetime Value as a Recognizable Date or Time

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY</td>
<td>Returns the day of the month from a SAS date value</td>
</tr>
<tr>
<td>HOUR</td>
<td>Returns the hour from a SAS time or datetime value</td>
</tr>
<tr>
<td>JULDATE</td>
<td>Returns the Julian date from a SAS date value</td>
</tr>
<tr>
<td>JULDATE7</td>
<td>Returns a seven-digit Julian date from a SAS date value</td>
</tr>
<tr>
<td>MINUTE</td>
<td>Returns the minute from a SAS time or datetime value</td>
</tr>
<tr>
<td>MONTH</td>
<td>Returns a number that represents the month from a SAS date value</td>
</tr>
<tr>
<td>QTR</td>
<td>Returns the quarter of the year from a SAS date value</td>
</tr>
<tr>
<td>SECOND</td>
<td>Returns the second from a SAS time or datetime value</td>
</tr>
<tr>
<td>WEEKDAY</td>
<td>Returns an integer that corresponds to the day of the week, from a SAS date value</td>
</tr>
<tr>
<td>YEAR</td>
<td>Returns the year from a SAS date value</td>
</tr>
</tbody>
</table>

Table 14.3 Functions That Extract Date and Times from SAS Date, Time, and Datetime Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEPART</td>
<td>Extracts the date from a SAS datetime value and returns the date as a SAS date value</td>
</tr>
<tr>
<td>TIMEPART</td>
<td>Extracts the time from a SAS datetime value and returns the time as a SAS datetime value</td>
</tr>
</tbody>
</table>

Date, Time, and Datetime Formats

DS2 formats write SAS date, time, and datetime values as recognizable dates and times. You use the PUT function to format a SAS date, time, or datetime value:

PUT(sasDateOrTime, format.);

The first argument to the PUT function is the SAS date, time, or datetime. The second argument is the format.

See “Converting DS2 Date, Time, and Timestamp Values to SAS Date, Time, or Datetime Values” on page 114 for an example of formatting dates and times in a DS2
program. The following table displays the results of formatting the date March 17, 2012 for each of the DS2 formats.

**Table 14.4 Examples of DS2 Date and Time Formats**

<table>
<thead>
<tr>
<th>Type of Language Element</th>
<th>Language Element</th>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date formats</td>
<td>DATE.</td>
<td>19069</td>
<td>17MAR12</td>
</tr>
<tr>
<td></td>
<td>DATE9.</td>
<td>19069</td>
<td>17MAR2012</td>
</tr>
<tr>
<td></td>
<td>DAY.</td>
<td>19069</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>DDMMYY.</td>
<td>19069</td>
<td>17/03/12</td>
</tr>
<tr>
<td></td>
<td>DDMMYY10.</td>
<td>19069</td>
<td>17/03/2012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYB.</td>
<td>19069</td>
<td>17 03 12</td>
</tr>
<tr>
<td></td>
<td>DDMMYYB10.</td>
<td>19069</td>
<td>17 03 2012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYC.</td>
<td>19069</td>
<td>17:03:12</td>
</tr>
<tr>
<td></td>
<td>DDMMYYC10.</td>
<td>19069</td>
<td>17:03:2012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYD.</td>
<td>19069</td>
<td>17-03-12</td>
</tr>
<tr>
<td></td>
<td>DDMMYYD10.</td>
<td>19069</td>
<td>17-03-2012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYN6.</td>
<td>19069</td>
<td>170312</td>
</tr>
<tr>
<td></td>
<td>DDMMYYN8.</td>
<td>19069</td>
<td>17032012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYP.</td>
<td>19069</td>
<td>17.03.12</td>
</tr>
<tr>
<td></td>
<td>DDMMYYP10.</td>
<td>19069</td>
<td>17.03.2012</td>
</tr>
<tr>
<td></td>
<td>DDMMYYS.</td>
<td>19069</td>
<td>17/03/12</td>
</tr>
<tr>
<td></td>
<td>DDMMYYS10.</td>
<td>19069</td>
<td>17/03/2012</td>
</tr>
<tr>
<td></td>
<td>DOWNAME.</td>
<td>19069</td>
<td>Monday</td>
</tr>
<tr>
<td></td>
<td>JULIAN.</td>
<td>19069</td>
<td>12077</td>
</tr>
<tr>
<td></td>
<td>MMDDYY.</td>
<td>19069</td>
<td>03/17/12</td>
</tr>
<tr>
<td></td>
<td>MMDDYY10.</td>
<td>19069</td>
<td>03/17/2012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYB.</td>
<td>19069</td>
<td>03 17 12</td>
</tr>
<tr>
<td>Type of Language Element</td>
<td>Language Element</td>
<td>Input</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------</td>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>MMDDYYB10.</td>
<td>19069</td>
<td>03 17 2012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYC.</td>
<td>19069</td>
<td>03:17:12</td>
</tr>
<tr>
<td></td>
<td>MMDDYYC10.</td>
<td>19069</td>
<td>03:17:2012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYD.</td>
<td>19069</td>
<td>03-17-12</td>
</tr>
<tr>
<td></td>
<td>MMDDYYD10.</td>
<td>19069</td>
<td>03-17-2012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYN6.</td>
<td>19069</td>
<td>031712</td>
</tr>
<tr>
<td></td>
<td>MMDDYYN8.</td>
<td>19069</td>
<td>03172012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYP.</td>
<td>19069</td>
<td>03.17.12</td>
</tr>
<tr>
<td></td>
<td>MMDDYYP10.</td>
<td>19069</td>
<td>03.17.2012</td>
</tr>
<tr>
<td></td>
<td>MMDDYYS.</td>
<td>19069</td>
<td>03/17/12</td>
</tr>
<tr>
<td></td>
<td>MMDDYYS10.</td>
<td>19069</td>
<td>03/17/2012</td>
</tr>
<tr>
<td></td>
<td>MMYY.</td>
<td>19069</td>
<td>03M2012</td>
</tr>
<tr>
<td></td>
<td>MMYYC.</td>
<td>19069</td>
<td>03:2012</td>
</tr>
<tr>
<td></td>
<td>MMYYD.</td>
<td>19069</td>
<td>03-2012</td>
</tr>
<tr>
<td></td>
<td>MMYYN.</td>
<td>19069</td>
<td>032012</td>
</tr>
<tr>
<td></td>
<td>MMYYP.</td>
<td>19069</td>
<td>03.2012</td>
</tr>
<tr>
<td></td>
<td>MMYYS.</td>
<td>19069</td>
<td>03/2012</td>
</tr>
<tr>
<td></td>
<td>MONNAME.</td>
<td>19069</td>
<td>March</td>
</tr>
<tr>
<td></td>
<td>MONTH.</td>
<td>19069</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>MONNYY.</td>
<td>19069</td>
<td>MAR2012</td>
</tr>
<tr>
<td></td>
<td>WEEKDATE.</td>
<td>19069</td>
<td>Monday, March 17, 2012</td>
</tr>
<tr>
<td></td>
<td>WEEKDATX</td>
<td>19069</td>
<td>Monday, 17 March 2012</td>
</tr>
<tr>
<td></td>
<td>WEEKDAY.</td>
<td>19069</td>
<td>2</td>
</tr>
<tr>
<td>Quarter formats</td>
<td>QTR.</td>
<td>19069</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>QTRR.</td>
<td>19069</td>
<td>1</td>
</tr>
<tr>
<td>Type of Language Element</td>
<td>Language Element</td>
<td>Input</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------</td>
<td>--------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Datetime formats</td>
<td>DATEAMPM.</td>
<td>19069</td>
<td>01JAN60:04:53:28 AM</td>
</tr>
<tr>
<td></td>
<td>DATETIME.</td>
<td>19069</td>
<td>01JAN60:04:53:28</td>
</tr>
<tr>
<td></td>
<td>DTDATE.</td>
<td>19069</td>
<td>01JAN60</td>
</tr>
<tr>
<td></td>
<td>DTMONYY.</td>
<td>19069</td>
<td>JAN60</td>
</tr>
<tr>
<td></td>
<td>DTWKDATX.</td>
<td>19069</td>
<td>Friday, 1 January 1960</td>
</tr>
<tr>
<td></td>
<td>DTYEAR.</td>
<td>19069</td>
<td>1960</td>
</tr>
<tr>
<td></td>
<td>DTYYQC.</td>
<td>19069</td>
<td>60:1</td>
</tr>
<tr>
<td>Time formats</td>
<td>HOUR.</td>
<td>19069</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>TIME.</td>
<td>19069</td>
<td>4:53:28</td>
</tr>
<tr>
<td></td>
<td>TIMEAMPM.</td>
<td>19069</td>
<td>4:53:28 AM</td>
</tr>
<tr>
<td></td>
<td>TOD.</td>
<td>19069</td>
<td>04:53:28</td>
</tr>
<tr>
<td>Year formats</td>
<td>YEAR.</td>
<td>19069</td>
<td>2012</td>
</tr>
<tr>
<td></td>
<td>YYMM.</td>
<td>19069</td>
<td>2012M03</td>
</tr>
<tr>
<td></td>
<td>YYMMC.</td>
<td>19069</td>
<td>2012:03</td>
</tr>
<tr>
<td></td>
<td>YYMMD.</td>
<td>19069</td>
<td>2012-03</td>
</tr>
<tr>
<td></td>
<td>YYMMN.</td>
<td>19069</td>
<td>201203</td>
</tr>
<tr>
<td></td>
<td>YYMMP.</td>
<td>19069</td>
<td>2012.03</td>
</tr>
<tr>
<td></td>
<td>YYMMS.</td>
<td>19069</td>
<td>2012/03</td>
</tr>
<tr>
<td></td>
<td>YYMMDD.</td>
<td>19069</td>
<td>12-03-17</td>
</tr>
<tr>
<td></td>
<td>YYMMDDB.</td>
<td>19069</td>
<td>12 03 17</td>
</tr>
<tr>
<td></td>
<td>YYMMDDC.</td>
<td>19069</td>
<td>12:03:17</td>
</tr>
<tr>
<td></td>
<td>YYMMDDD.</td>
<td>19069</td>
<td>12–03–17</td>
</tr>
<tr>
<td></td>
<td>YYMMDDN.</td>
<td>19069</td>
<td>20120317</td>
</tr>
<tr>
<td></td>
<td>YYMMDDP.</td>
<td>19069</td>
<td>12.03.17</td>
</tr>
<tr>
<td></td>
<td>YYMMDDS.</td>
<td>19069</td>
<td>12/03/17</td>
</tr>
<tr>
<td>Type of Language Element</td>
<td>Language Element</td>
<td>Input</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>YYMON.</td>
<td>19069</td>
<td>2012MAR</td>
</tr>
<tr>
<td>Year/Quarter formats</td>
<td>YYQ.</td>
<td>19069</td>
<td>2012Q1</td>
</tr>
<tr>
<td></td>
<td>YYQC.</td>
<td>19069</td>
<td>2012:1</td>
</tr>
<tr>
<td></td>
<td>YYQD.</td>
<td>19069</td>
<td>2012-1</td>
</tr>
<tr>
<td></td>
<td>YYQP.</td>
<td>19069</td>
<td>2012.1</td>
</tr>
<tr>
<td></td>
<td>YYQS.</td>
<td>19069</td>
<td>2012/1</td>
</tr>
<tr>
<td></td>
<td>YYQN.</td>
<td>19069</td>
<td>20121</td>
</tr>
<tr>
<td></td>
<td>YYQR.</td>
<td>19069</td>
<td>2012Q1</td>
</tr>
<tr>
<td></td>
<td>YYQRD.</td>
<td>19069</td>
<td>2012-I</td>
</tr>
<tr>
<td></td>
<td>YYQRP.</td>
<td>19069</td>
<td>2012I</td>
</tr>
<tr>
<td></td>
<td>YYQRS.</td>
<td>19069</td>
<td>2012/1</td>
</tr>
<tr>
<td></td>
<td>YYQRN.</td>
<td>19069</td>
<td>2012I</td>
</tr>
<tr>
<td></td>
<td>YYQZ.</td>
<td>19069</td>
<td>1201</td>
</tr>
</tbody>
</table>
Overview of DS2 Arrays

In DS2, an array is a named aggregate collection of homogeneous data. DS2 has two types of arrays: temporary and variable. These arrays have the following characteristics.

- homogeneous by type
- multidimensional (the number of bounds can be $\geq 1$)
- indexed by signed integer values
- exists only for the duration of the DS2 program or DS2 procedure
Temporary Arrays

Overview of Temporary Arrays

The elements of a temporary array are temporary in that they are not located in the PDV and therefore do not appear in any result table. Temporary data element values are automatically retained across iterations rather than being reset to missing at the beginning of the next iteration. Temporary arrays exist only for the duration of the DS2 program.

You use the DECLARE statement to specify the name, data type, and number and size of the array bounds. You can also use a HAVING clause in the DECLARE statement to associate label, format, and informat attributes with a temporary array. For example, the following DECLARE statement specifies a three element temporary array that stores three temporary double values outside the PDV.

declare double a[3]

For more information about the PDV, see “Processing a DATA Step: A Walk-through” in SAS Language Reference: Concepts.

Temporary Array Declaration

Temporary array declarations are similar to scalar declarations. In addition to the data type and name, you can also specify the number and size of the array bounds. Multiple bounds (or dimensions) are specified using comma separators.

The form of signed integer pairs specifies the lower and upper bounds for each dimension of the array \([l:h]\), where \(l\) represents the lowest index for the given bound and \(h\) represents the highest index for the given bound. The lower bound specification, \(l\), is
optional. If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

An error is returned if the upper bound, \( h \), is less than the lower bound. If you specify an array bound with only one integer, then that integer is interpreted as the upper bound. The default lowest bound is 1.

The upper bound of an array can also be sized based on the number of elements in a dimension of a previously declared array. You use a DIM function call for the upper bound. The DIM function is the only function that can be used to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

The part of the DECLARE statement for temporary array declaration is as follows.

```
DECLARE data-type <variable-list> [having-clause];
```

For more information, see “DECLARE Statement” on page 877.

---

**Variable Arrays**

**Overview of Variable Arrays**

Variable arrays are a way to simplify processing of a series of variables that have a similar name or purpose in the input data. The elements of a variable array refer to variables in the PDV. Variable arrays exist only for the duration of the DS2 program. However, the content of the referenced variables might be preserved in one or more result sets.

You use the VARARRAY statement to specify the name, data type, and number and size of the array bounds. For example, the following VARARRAY statement specifies a three element variable array that refers to three double variables (a1, a2, a3) in the PDV.

```
vararray double a[3];
```

The VARARRAY statement in the previous example creates any of the double variables (a1, a2, a3) that have not previously been created. Variable array element a[1] references variable a1, a[2] references variable a2, and a[3] references variable a3.

After a variable array is created, the variable array elements act as a second set of identifiers that can be used to read or modify the data that is stored in the referenced variables in the PDV.

You can also use a HAVING clause in the VARARRAY statement to associate label, format, and informat attributes with a variable array.

For more information, see Chapter 15, “DS2 Arrays,” on page 121 and the “VARARRAY Statement” on page 951.

**Variable Array Declaration**

Variable array declarations are similar to scalar declarations. In addition to the data type and name, you can also specify the number and size of the array bounds. Multiple bounds (or dimensions) are specified using comma separators.
Array bounds have two forms.

**signed integer pairs**

The form of signed integer pairs specifies the lower and upper bounds for each dimension of the array. \([l:h]\), where \(l\) represents the lowest index for the given bound and \(h\) represents the highest index for the given bound. The lower bound specification, \(l\), is optional. If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

An error is returned if the upper bound, \(h\), is less than 1. If you specify an array bound with only one integer, then that integer is interpreted as the upper bound. The default lowest bound is 1.

```c
vararray double a[5];
```

declares an array \(a\) of type double, with five elements indexed from 1 to 5. `vararray char b[5,10];` declares a two dimensional character array \(b\) with 5 elements in the first dimension and 10 elements in the second dimension for a total of 50 elements in the array. `vararray int c[3] x y z;` declares an array \(c\) with three elements. The array is indexed with a lower bound of 1 and an upper bound of 3.

The upper bound of an array can also be sized based on the number of elements in a dimension of a previously declared array. You use a DIM function call for the upper bound. The DIM function is the only function that can be used to specify an upper array bound. The DIM function cannot be used to specify the lower bound of a dimension.

```c
* (asterisk)
```

The * form specifies a one-dimensional array in which the lower bound is 1 and the upper bound is the number of variables in the variable list.

For more information, see “Variable Lists” on page 59. For more information about how to declare variable arrays and how to specify multiple bounds, see the “VARARRAY Statement” on page 951.

### Definition of Variables in a VARARRAY Statement

A VARARRAY statement defines any variable in its variable list that has not previously been defined. Some variable list types reference only existing variables and therefore do not result in the definition of new variables. The following table shows which variable list types can define new variables and which variable list types reference only existing variables.

<table>
<thead>
<tr>
<th>Variable List</th>
<th>Example</th>
<th>Variable Expansion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(x) (y) (z)</td>
<td>immediate</td>
<td>can define new variables</td>
</tr>
<tr>
<td>numbered range</td>
<td>(x1-x5)</td>
<td>immediate</td>
<td>can define new variables</td>
</tr>
<tr>
<td>name range</td>
<td>(sales_jan)--(sales_mar)</td>
<td>delayed</td>
<td>reference only existing variables</td>
</tr>
<tr>
<td>name prefix</td>
<td>(sales:)</td>
<td>delayed</td>
<td>reference only existing variables</td>
</tr>
<tr>
<td>type</td>
<td>(smallint)</td>
<td>delayed</td>
<td>reference only existing variables</td>
</tr>
<tr>
<td>Variable List</td>
<td>Example</td>
<td>Variable Expansion</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>special name</td>
<td><em>all</em></td>
<td>delayed</td>
<td>references only existing variables</td>
</tr>
</tbody>
</table>

The name and numbered range variable lists can be expanded without examining all the variables in the DS2 program. Therefore, these types of variable lists are expanded immediately when the VARARRAY statement is encountered in the program.

The other types of variable lists must examine all variables that are defined in the DS2 program. Expansion of these variable lists is delayed until after all statements in the DS2 program have been examined and all variables have been defined.

This delay can lead to some unexpected error conditions. For example, consider the following program.

```plaintext
data;
  vararray int x[1] x:;
method run();
  x1 = 5.0;
end;
enddata;
run;
```

1. The VARARRAY statement does not create any variables because the prefix variable list x: references only existing variables. The expansion of prefix variable list x: is delayed until all statements in the program have been examined.

2. In the assignment statement variable x1 is undefined. Therefore, the assignment statement assigns the type of the right hand side value (DOUBLE) to variable x1.

After all the program statements are examined, the prefix variable list x: is expanded to x1, the only existing variable with prefix x. The DS2 compiler then issues a compilation error because variable x1 of type DOUBLE is incompatible with variable array x of type INTEGER.

One way to remove the error condition is to change the VARARRAY statement, `vararray int x[1] x:;`, to `vararray int x[1] x1;` or `vararray int x[1];`. The revised statement defines the variable x1 as type INTEGER.

**Delayed Variable Definition with DIM Variable Array Bounds**

If a variable array has an upper dimension bound based on the dimension of another array, then the definition of variables for the array can be delayed until all statements in the program have been examined. Here is an example.

```plaintext
data;
  vararray double x[*] x:;
  vararray int out[dim(x)];
method init();
  out1 = 0.0;
end;
method run();
  set in;
end;
```
The expansion of prefix variable list x: is delayed until all statements in the program have been examined. Therefore, the size of variable array x is not known until all statements have been examined.

The out variable array has the default variable list out1-outn, where n is the number of elements specified for the variable array. The determination of the number of elements in out is delayed until the size of array x is known which was delayed until all statements have been examined.

In the assignment statement, variable out1 is undefined. Therefore, the assignment statement assigns the type of the right hand side value, DOUBLE, to variable out1.

After all statements in the above DS2 program are processed, the following occurs. Assume the table in has 3 double variables, x1 x2 x3.

- The prefix variable list x: is expanded to x1 x2 x3.
- The size of variable array x is determined to be 3.
- The size of variable array out is determined to be 3 (dim(x)).
- The default variable list out1-out3 is expanded to out1 out2 out3.
- The variables out2 and out3 are defined as type INTEGER because they were not previously defined. Note that out1 was defined as type DOUBLE by the assignment statement out1 = 0.0;

The DS2 compiler issues a compilation error because variable out1 of type DOUBLE is incompatible with variable array out of type INTEGER.

One way to remove the error condition is to change the array assignment, out1 = 0.0;, to out[1]=0.0;. This change updates the out1 data value by means of an out array reference to prevent the assignment statement from assigning type DOUBLE to the variable out1.

---

**Declaring Arrays with a HAVING Clause**

The declaration statement for a temporary or variable array can contain a HAVING clause. The HAVING clause associates a label, format, and informat attribute with the array. If the array is a variable array, then the HAVING clause is also associated with the variables referenced by the variable array.

The decision about when to apply the HAVING clause to a variable that is referenced by the variable array depends on when the variable list that contains the variable reference is expanded. Name and numbered range variable lists are normally expanded when the VARARRAY statement is processed. Therefore, the HAVING clause is applied to all variables referenced by these lists at that time. Name range, name prefix, type, and special name variable lists are expanded after all statements in the program have been examined and all variables in the program have been defined. Consequently, the HAVING clause for all variables referenced by these variable lists is applied after all statements in the DS2 program have been examined.

Consider the following program.

data;
1 declare double x1 x2 having format 5.0;
2 vararray double x[3] having format 5.2;
Declaring Arrays with a HAVING Clause

3 declare double x3 having format ROMAN5.;
enddata;

1. The DECLARE statement is processed. Variables x1 and x2 are defined. The HAVING clause **format 5.0** is associated with variables x1 and x2.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>5.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

2. The VARARRAY statement is processed. Default variable list x1-x3 is expanded to x1 x2 x3. Variable x3 is defined. The HAVING clause **format 5.2** is associated with variables x1 x2 x3.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
</tr>
</tbody>
</table>

3. The DECLARE statement is processed. The HAVING clause **format ROMAN5.** is associated with variable x3.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
</table>
   | Format   | 5.2| 5.2| ROMAN5.

Now consider what happens if the variable list type is modified to a type that results in delayed processing of the variable list.

```
data;
1 declare double x1 x2 having format 5.0;
2 vararray double x[*] x: having format 5.2;
3 declare double x3 having format ROMAN5.;
enddata;
```

1. The DECLARE statement is processed. Variables x1 and x2 are defined. The HAVING clause **format 5.0** is associated with variables x1 and x2.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>5.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

2. The VARARRAY statement processing is delayed or begins processing. Prefix variable list x: cannot be expanded until all statements in the program have been examined and all variables are defined.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>5.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

3. The DECLARE statement is processed. Variable x3 is defined. The HAVING clause **format ROMAN5.** is associated with variable x3.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
</table>
   | Format   | 5.0| 5.0| ROMAN5.

4. The VARARRAY statement completes processing. Prefix variable list x: is expanded to x1 x2 x3. The HAVING clause **format 5.2** is associated with variables x1 x2 x3.

<table>
<thead>
<tr>
<th>Variable</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
</tr>
</tbody>
</table>
Array Assignment

Overview of Array Assignment

DS2 supports array assignment with the := operator. The syntax for assigning an array or constant list is as follows:

```
adarray:=array;
adarray:=(constant list);
```

In an array assignment, `array` can be either a temporary or variable array.

Array Assignment from Another Array

When you assign one array to another array, the data types of the two arrays must be compatible (either the same or convertible). The number of dimensions and the total number of elements in each dimension do not have to be the same.

Consider the assignment from array `y` to array `x` as shown in this statement.

```
x:=y;
```

During the assignment, each element of array `y` is assigned to each element of array `x`, for example, `x[1]=y[1]; ... x[n]=y[n];`

The basic algorithm for evaluating `x[i] = y[i]` follows. First `y[i]` is examined to see whether it is missing (SAS mode) or null (ANSI mode).

- If `y[i]` is missing or null, then missing or null is assigned to `x[i].`
- If `y[i]` is not missing or null, then the types of `x[i]` and `y[i]` are examined.
  - If the type of `y[i]` is different from the type of `x[i]`, then `y[i]` is converted to the type of `x[i].`
    - If the conversion of `y[i]` succeeds, then the result of the conversion is assigned to `x[i].`
    - If the conversion of `y[i]` fails, then missing or null is assigned to `x[i].`
  - If the type of `y[i]` is the same as the type of `x[i]`, then `y[i]` is assigned to `x[i].`

If array `x` and array `y` do not have the same number of elements, then as many elements as possible are assigned from array `y` to array `x` and null or missing is assigned to any remaining elements in array `x`. The length of array `x` is not modified by the assignment.

In the following example, array `x` has ten elements, array `y` has seven elements, and array `y` is assigned to array `x`. Therefore, the seven elements from array `y` are assigned to the first seven elements of array `x`, and missing is assigned to the last three elements of array `x`.

```
data _null_;    
  method init();
  declare double x[10];
  declare double y[7];
```

1 The decision to assign missing or null to an array element depends on data type of the array and whether the program is running in SAS or ANSI mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.
x := (0 0 0 0 0 0 0 0 0 0);
y := (1 2 3 4 5 6 7);
x := y;
put x[*]=;
end;
enddata;
run;

The following lines are written to the SAS log.

\[

**Special Case for Double Missing Values**

If \(y[i]\) is a DOUBLE with a SAS missing value, for example, \(Z\), then DS2 tries to preserve the SAS missing value during the assignment according to these rules:

- If \(x[i]\) is a DOUBLE, then the SAS missing value from \(y[i]\) is assigned to \(x[i]\).
- If \(x[i]\) is a character string, then the missing character representation of \(y[i]\) (for example, \(Z\) for \(Z\), is assigned to \(x[i]\)).

This special case processing only occurs when DS2 is in SAS mode and the type of \(y[i]\) is a DOUBLE.

For more information about null and missing values, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81. For an example of an array assignment, see “Example: Arrays” on page 1315.

**Array Assignment with Variable Arrays**

The elements of a variable array reference variables in the PDV. In the array assignment statement \(x := y\), the value of the elements of \(y\) are assigned elementwise to the elements of \(x\). If either \(x\) or \(y\) is a variable array (vararray), then the assignment is always \(x[i]=y[[i]]\).

Array assignment to a variable array does not modify the elements (this is, the references) in the variable array. Instead, the data in the variables referenced by the elements of the array are modified. Similarly, in an array assignment from a variable array, the data in the variables referenced by the elements of the variable array are used for the assignment.

**Array Assignment from a Constant List**

To assign from a constant list to an array, the constants in the constant list must be compatible (either the same or convertible) with the data type of the array. The constant list and the array do not have to have the same number of dimensions or the same number of elements in each dimension.

Assume this array assignment statement.

\[
x := (c1 \ c2 \ c3 \ldots \ cn);\]

During array assignment from a constant list to array \(x\), each element of the constant is assigned to each element of array \(x\) as shown in the following expanded form.

\[
x[1] = c1; \ 
x[2] = c2;\]
The basic algorithm for evaluating $x[i] = ci$ follows. First $ci$ is examined to see whether it is missing or null.\(^1\)

- If $ci$ is missing or null, then missing (SAS mode) or null (ANSI mode) is assigned to $x[i]$.
- If $ci$ is not missing or null, then the types of $x[i]$ and $ci$ are examined.
  - If the type of $ci$ is different from the type of $x[i]$, then $ci$ is converted to the type of $x[i]$.
    - If the conversion of $ci$ succeeds, then the result of the conversion is assigned to $x[i]$.
    - If the conversion of $ci$ fails, then missing or null is assigned to $x[i]$.
  - If the type of $ci$ is the same as the type of $x[i]$, then $ci$ is assigned to $x[i]$.

If the constant list and the array $x$ do not have the same number of elements, then as many constants as possible are assigned from the constant list to array $x$ and a null or missing value is assigned to any remaining elements in array $x$. The length of array $x$ is not modified by the assignment.

In the following example, a constant list having five constants is assigned to array $x$ having seven double elements. The five constants in the constant list are assigned to the first five elements of array $x$, and missing is assigned to the last two elements of array $x$.

```sas
data _null_
method init();
declare double x[7];
x := (1 '2' 3.3 '' .Z);
put x[*]=;
end;
enddata;
run;
```

The following lines are written to the SAS log.

```
```

Note: The types of the elements in the constant list can be heterogeneous as long as all the types of the elements are convertible to the type of the assigned to array.

Here is another example of an array assignment from a constant list.

```sas
declare char(2) a[2, 3];
... 
a := ('aa' 'bb' 'cc')('dd' 'ee' '');
```

The elements in array $a$ after the above assignment statement would look like this.

```
'aa'  'bb'  'cc'
'dd'  'ee'  '
```

---

\(^1\) The decision to assign missing or null to an array element depends on data type of the array and whether the program is running in SAS or ANSI mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.
Array Arguments

Overview of Array Arguments

DS2 arrays can be passed as arguments to DS2 methods. DS2 arrays are always passed by reference to methods. DS2 arrays cannot be passed by value, that is, a copy of the array cannot be supplied as an argument to a method. DS2 array arguments must have the same type as specified by the array parameter in order to match the array parameter. Array arguments do not support implicit type conversion to a different type. DS2 arrays are passed as either a bounded array parameter, for example, \( a[8] \) or an unbounded array parameter, for example, \( a[*] \).

Defining Array Parameters

A DS2 method can be defined with array parameters. The kind of array (temporary or variable) is specified in the parameter definition. The following table illustrates the syntax for defining different kinds of parameters.

<table>
<thead>
<tr>
<th>Type of Parameter</th>
<th>Parameter Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar parameter</td>
<td><code>data-type parameter-name</code></td>
<td>double x</td>
</tr>
<tr>
<td>temporary array parameter</td>
<td><code>data-type parameter-name [bounds]</code></td>
<td>double x[5]</td>
</tr>
<tr>
<td>variable array parameter</td>
<td><code>VARARRAY data-type parameter-name [bounds]</code></td>
<td>vararray double x[2,4]</td>
</tr>
</tbody>
</table>

The data type and kind of an array argument must exactly match the type and kind specified in the array parameter definition. DS2 does not convert array arguments to a different kind or data type. For example, if a parameter is defined as a temporary array of doubles, then the argument must be a temporary array of doubles. If a variable array of doubles or a temporary array of integers is passed as an argument for the temporary array of doubles parameter, then an error occurs.

The following DS2 program illustrates the definition of a method that has array parameters and illustrates calls to the method using array arguments.

```plaintext
data _null_;  
declare double x;  
declare double y[4];  
vararray double z[4];  
  
method m(double u, double v[4], vararray double w[4]);  
do i = 1 to 4;  
v[i] = u;  
w[i] = u;  
end;  
end;  
  
method init();  
m(x, y, z);     /* call method m */
```
Bounded Array Parameters

A bounded array parameter supplies explicit bounds information for accessing elements of the array argument. These are examples.

```plaintext
method m(double a[4]);
method m(vararray double a[5:10,3:6]);
```

An example of using a bounded array parameter is showing how it matches any DS2 array argument with the same number of elements regardless of the dimensionality of the array argument. For example, bounded array parameter `a[2,4]` would match array arguments `a1[8]`, `a2[11:12,11:14]`, and `a3[2,2,2]` because arrays `a1`, `a2`, and `a3` each have 8 elements. If the dimensionality of the array argument differs from the array parameter, then an element of the array parameter is mapped to the corresponding element in the array argument. This mapping is based on the position of the element in the array, using row-major order. For example, array parameter `a1[2,1]` accesses the 5th element of an 8 element array and thus would map to `a1[5]`, `a2[12,11]`, and `a3[2,2,1]`.

Unbounded Array Parameters

An unbounded array parameter does not supply any explicit bounds information for the corresponding array argument. Here is an example.

```plaintext
method m(double a[*]);
```

The asterisk (*) for the array bounds specifies that parameter `a` is an unbounded array parameter.

An unbounded array parameter matches any DS2 array argument regardless of the number of elements or dimensionality of the array argument. In a DS2 method, the array parameter is treated as a one-dimensional array even if the corresponding array argument is a multi-dimensional array. The unbound array parameter is mapped to the multi-dimensional array using row-major order. Consider the 2x3 array `a[2, 3].`

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

If array `a` is passed to a method as an unbounded array parameter `b`, then `b` will be accessed as a one-dimensional array of six elements.

|   | 1 | 2 | 3 | 4 | 5 | 6 |

Note that accessing an element of array parameter `b` results in the access of an element of array `a`, because array `a` is passed by reference to the DS2 method. Here is an example.

```plaintext
method m(double b[*]);
b[1] = 10; /* assigns 10 to a[1, 1] */
b[2] = 20; /* assigns 20 to a[1, 2] */
b[3] = 30; /* assigns 30 to a[1, 3] */
b[4] = 40; /* assigns 40 to a[2, 1] */
b[5] = 50; /* assigns 50 to a[2, 2] */
```
method init();
    declare double a[2, 3];
    m(a);
end;

In an array expression of the form a[i], where a is an unbound array parameter, bounds checking of the index, i, is performed at run time. If an index of an array parameter is beyond the boundaries of the array argument, DS2 issues an error and the array expression will evaluate to NULL or missing.

Note: An unbounded array parameter cannot be used as an argument for a bounded array parameter.

How to Query Array Dimensions

The following functions can be used to obtain dimension information about an array. For more information about each function, see Chapter 23, “DS2 Functions,” on page 335.

**DIM(a)**
Returns the number of elements in the first dimension of array a

**DIM(a, n)**
Returns the number of elements in dimension n of array a

**LBOUND(a)**
Returns the lower bound of the first dimension of array a

**LBOUND(a, n)**
Returns the lower bound of dimension n of array a

**HBOUND(a)**
Returns the upper bound of the first dimension of array a

**HBOUND(a, n)**
Returns the upper bound of dimension n of array a

**NDIMS(a)**
Returns the number of dimensions of array a

For any of the query functions, the array argument a can be a temporary array or a variable array, and the dimension argument n should be an expression that evaluates to an integral value. The following example illustrates these query functions.

do i = 1 to dim(a1);
    put a1[i];
end;

numelems = 0;
do i = 1 to ndims(a2);
    numelems = numelems + dim(a2, i);
end;

do i = lbound(a2, 1) to hbound(a2, 1);
    do j = lbound(a2, 2) to hbound(a2, 2);
        do k = lbound(a2, 3) to hbound(a2, 3);
sum = sum + a2[i,j,k];
end;
end;
end;

If an array function is called with a dimension value outside the dimensions of the array, then a run-time error will occur and the function will return a NULL integer value.

---

### How to Output Array Content

The DS2 PUT statement can be used to output individual elements of an array or all elements of an array.

The syntax to output an individual array element is as follows.

**PUT** array-name[element]<>;

The syntax to output all elements of an array is as follows.

**PUT** array-name[*]<>;

The PUT statement can output elements of temporary arrays and variable arrays. When all elements of an array are output with the **array-name[*]** syntax, all of the elements of the array are output with the same format. In other words, different formats cannot be specified for different array elements with **array-name[*]**.

The following example illustrates the put statement output the contents of an array:

```sas
data _null_
   vararray varchar(10) x[10];
   declare double      y[2,2,2];
   method init();
   x[1] = 'a';
   do i = 2 to dim(x);
      x[i] = x[i-1] || x[1];
   end;
   put 'X:' x[*];
   y := (10 20 30 40 50 60 70 80);
   put 'Y:' y[*]=;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
X: a a aa aaa aaaaa aaaaaaa aaaaaaaaa aaaaaaaaaa aaaaaaaaaaa
Y: y[1,1,1]=10 y[1,1,2]=20 y[1,2,1]=30 y[1,2,2]=40 y[2,1,1]=50 y[2,1,2]=60
   y[2,2,1]=70 y[2,2,2]=80
```

For more information, see the “PUT Statement” on page 925.
Chapter 16
DS2 Packages

Introduction to DS2 Packages

A DS2 package is a collection of methods and variables that can be used in DS2 programs. A DS2 package supports a set of related tasks and is designed for reuse.

There are two types of packages:

User-defined packages
These are packages that you can use to store methods for any purpose.

For more information, see “User-Defined Packages” on page 143.

Predefined packages
These packages are predefined in DS2.
For more information, see “Predefined DS2 Packages” on page 143.

FCMP
- Supports calls to FCMP functions and subroutines from within the DS2 language.

Hash and hash iterator
- Enables you to quickly and efficiently store, search, and retrieve data based on unique lookup keys.

HTTP
- Constructs an HTTP client to access HTTP web services.

JSON
- Enables you to create and parse JSON text.

Logger
- Provides a basic interface (open, write, and level query) to the SAS logging facility.

Matrix
- Provides a powerful and flexible matrix programming capability.

SQLSTMT
- Provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

TZ
- Provides a way to process local and international time and date values.

To use a package, a DS2 program, another package, or a thread instantiates the package and accesses its methods. For a comparison between packages, DS2 programs, and threads, see “Block Statements” on page 862.

**Note:** You can invoke a DS2 package method as a function in a FedSQL SELECT statement. For more information, see “Using DS2 Packages in Expressions” in *SAS FedSQL Language Reference* and “Package Method Expression” on page 100.

A package is used as a template to construct an instance of the package. A package variable is used to reference a particular instance of the package. Here is an example.

```sas
/* Create package animal */
package animal;
   declare varchar(100) s;
   method animal(varchar(100) s);
      this.s = s;
   end;

   method speak();
      put s;
   end;
endpackage;

data _null_;  
   method init();
      /* Create variable a1 of type animal. */
      declare package animal a1;
      /* Create variable a2 of type animal. 
      Construct an instance of type animal. 
      Set variable a2 to reference the newly constructed animal instance. */
```
Packages and Scope

Overview of Packages and Scope

The lifetime of a package instance is dependent on the scope in which the instance is created. A package instance is deleted automatically when execution exits the scope in which the instance was created.

By default, a package instance that is created in a method is created in the local scope of the method. As a result, these package instances are local to a method and are deleted when on return from the method.

If a method needs to return a package instance, then the package instance needs to be created in a scope outside the scope of the method. Otherwise, the package instance would be deleted prematurely when the method returns. With the _NEW_ operator, you can specify that the instance be created in a different scope than the default scope.

Specifying Scope

The scope in which to construct a package instance can be specified with the _NEW_ operator. The specified scope can be the scope of another package instance or the global scope of the package, thread, or data program containing the package instance.

Here is the syntax for the _NEW_ operator.

```
package-variable = _NEW_ [[THIS] | [package-instance]]
package-name (constructor-arguments);
```

For more information, see the “_NEW_ Operator” on page 857.

This example uses the _NEW_ operator to construct an instance of the package `mypkg` in the global scope of the data program. The package instance that is returned by the _NEW_ operator is assigned to the variable `p`.

```
package mypkg/overwrite=yes;
    method m(double x) returns double;
```
return x+99;
end;
endpackage;

data _null_;  
dcl package mypkg p;
method m();
    p = _new_ [this] mypkg();
end;
enddata;

This causes the instance assigned to the variable p to have global scope.

It is important to note the difference between the scope of the package variable and the scope of the package instance.

In the following example, the variable p is declared local to the method m and the instance is global to the entire program.

data _null_;  
method m();
    dcl package mypkg p;
    p = _new_ [this] mypkg();
end;

method init();
    m();
    /* instance of mypkg created in global */
    /* scope and therefore still exists. It is */
    /* a dead reference because the instance */
    /* is not referenced by any package variable */
    /* and therefore is inaccessible. */
end;
enddata;
run;

If the instance is not explicitly deleted or reassigned to a variable with global scope, then the instance becomes a dead reference after the method m completes.

For more information, see the "_NEW_ Operator" on page 857.

Global Scope

The THIS keyword is used as an argument in the _NEW_ operator to specify that a package instance be created in a global scope. DS2 supports three types of global scope:

- program
- package
- thread

A package instance created in a data program with the _NEW_ operator and the THIS scope keyword is created in program global scope. The lifetime of the package instance is the entire program. Package instance p in the previous topic’s example is in program global scope. Package instance p is not deleted until the program exits.

A package instance created in a package block with the _NEW_ operator and the THIS scope keyword is created in package global scope. Package global scope is limited to the lifetime of a specific instance of the package. When the package instance is deleted, all package instances in the instance’s global scope are also deleted.
The following example illustrates a package instance with package global scope.

```plaintext
package pkgA;
    declare double i;
    method pkgA(double i);
        this.i = i;
    end;
endpackage;

package pkgB;
    declare package pkgA a;
    method pkgB(double i);
        a = _new_ [THIS] pkgA(i);
    end;
endpackage;

data _null_; 
    method init();
        declare package pkgB b(5);
    end;
enddata;
run;
```

1. The package instance referenced by package variable `b` is created in the local scope of the data program's INIT method. Package instance `b` is deleted when the INIT method returns.

2. The package instance referenced by package variable `a` is created in the package global scope of package instance `b`. Package instance `a` is deleted when package instance `b` is deleted.

Package instance `a` is in the package global scope of package instance `b`. When package instance `b` is destroyed at the return of the INIT method, package instance `a` is also destroyed.

The third type of global scope is thread global scope. A package instance created with the THIS scope keyword in a thread program is in thread global scope. Thread global scope is limited to the lifetime of a specific thread. When the thread exits, all package instances in the thread's global scope are automatically deleted.

**Package-Specific Scope**

The _NEW_ operator can also be used to specify that a new package instance has the same lifetime as an existing package instance by specifying the existing package instance as the scope argument. The syntax is as follows.

```plaintext
package-variable = _NEW_ [existing-package-instance] package-name
                  {{constructor-arguments}};
```

The new package instance is then created in the same scope in which the existing package instance was created. The existing package instance must be created prior to the time that the _NEW_ operator executes. The existing package instance and the new package instance can be different package types.

In this example, two `pkgA` package instances are created with the same lifetime as a hash package instance.

```plaintext
package pkgA;
    declare double i;
```
method pkgA(double i);
   this.i = i;
end;
endpackage;

data _null_
   declare package hash h();
   declare int          i;
   declare package pkgA a;
   method init();
       h.definekey('i');
       h.definedata('a');
       h.definedone();

       i = 55;
       a = _new_ [h] pkgA(i);
       h.add();

       i = 66;
       a = _new_ [h] pkgA(i);
       h.add();

end;
enddata;

1 A hash package instance referenced by package variable h is created in program
   global scope.

2 A pkgA package instance is created in the same scope in which package instance h
   was created. The pkgA package instance is assigned to package variable a.

3 The pkgA package instance referenced by variable a is added to hash h with key i
   (55).

4 A second pkgA package instance is created in the same scope in which package
   instance h was created. The second pkgA package instance is assigned to package
   variable a.

5 The second pkgA instance referenced by variable a is added to hash h with key i
   (66).

Both instances of pkgA effectively have the same lifetime as the hash package instance
h because the pkgA instances are created in the same scope (program global scope) as
the hash package instance h. Thus, when the hash package instance h is destroyed at the
end of the program, both instances of pkgA are also destroyed.

Returning Package Instances from Methods

You can use the RETURN statement to return package instances from methods. Here is
an example.

data_null_
   dcl package mypkg p;
   method r() returns package mypkg;
       return _new_ [this] mypkg();
   end;

method init();
  p = r();
  x = p.m(100);
  put x=;
end;

method term();
  x = p.m(200);
  put x=;
end;
enddata;

In this case, the method r returns a global instance of the package, mypkg, which is then used in the INIT method. Because the lifetime of p is global, it can be used again in other methods as shown here in the TERM method.

The variable p must be declared in global scope in order to do this. If it had been declared local to the INIT method, it would, of course, have not been available in the TERM method. Also, in that case, the instance of the package would have become a dead reference after the INIT method had finished, unless it had been explicitly deleted as in the following modified INIT method.

method init();
  p = r();
  x = p.m(100);
  put x=;
  p.delete();
end;

This effect could be more easily achieved by declaring p to be local to the INIT method.

**Passing Package Arguments**

In addition to returning packages from methods, DS2 allows package instances to be passed to methods. This example uses the packages mypkg and mypkg2 that were created in “Package-Specific Scope” on page 139. The instance p2 is passed to the method tp, which then instantiates the subpackage mypkg in p2.

data _null_
  dcl package mypkg2 p2();

  method tp(package mypkg2 p2);
    p2.p = _new_ [p2] mypkg();
  end;

  method init();
    dcl package mypkg p;
    tp(p2);
    p = p2.p;
    x = p.m(100);
    put x=;
  end;
enda;

p2 is declared in a global scope and p and p2 are both automatically deleted at the end of the program. The following tp method is nearly equivalent to the tp method in the...
above program, but not tying the scope of \( p \) to \( p2 \) is slightly different in that you can delete \( p2 \) separately from \( p \) if necessary.

```ds2
method tp(package mypkg2 p2);
    p2.p = _new_ [this] mypkg();
end;
```

*Note:* If either THIS or \( p2 \)'s package scope were not used, the instance of \( p \) would be deleted at the end of the method \( tp \). This is the default behavior of packages.

---

**Dot Operator in Packages**

In the DS2 language, standard dot notation is restricted to three-level names. If you have nested packages, the standard dot notation requires a series of operations to make the method call. Here is an example.

```ds2
dcl package TOP top();
...
  t1=top.middle;
  t2=t1.bottom;
  result=t2.calledmethod();
```

However, when referencing DS2 packages, you can use the dot(.) as a standard binary operator. This enables you to access methods of nested packages by using a four-level name.

The previous example can be simplified as follows.

```ds2
dcl package TOP top();
...
  result=top.middle.bottom.calledmethod();
```

---

**Package Constructors and Destructors**

Constructors and destructors are special package methods that are used during construction and destruction of package instances. The constructor method initializes a newly constructed package instance. The destructor method performs cleanup, releases resources held by the package instance before the package instance is destroyed, or both. A package’s constructor method has the same name as the class, and a package’s destructor is the DELETE method. Constructors and destructors do not have return types and do not return values.

DS2 automatically calls a package’s constructor when an instance of the package is constructed. DS2 automatically calls a package’s destructor when a package instance goes out of scope and is destroyed. Note that creating a package variable with a DECLARE PACKAGE statement does not result in DS2 calling the package’s constructor. A package’s destructor is called only when a package instance is constructed with either a DECLARE PACKAGE statement with constructor arguments or with a _NEW_ operator.

For more information, see the applicable DECLARE PACKAGE statements and _NEW_ operators in the language reference section of this document.
User-Defined Packages

You can store methods that you create in user-defined packages. These packages can be thought of as libraries of your methods. Any type of method can be saved in a package. Once you have stored methods in a package (using the PACKAGE statement), you can access them by creating an instance of the package with only a DECLARE statement or with the _NEW_ operator.

```
declare package package-name instance-name;
instance-name = _new_ package-name();
```

Alternatively, you can use the condensed constructor syntax:

```
declare package package-name instance-name();
```

For more information, see the “PACKAGE Statement” on page 921, “DECLARE PACKAGE Statement” on page 884, and the “_NEW_ Operator” on page 857.

Here is an example of a very simple user-defined package called MATH. It contains a method that adds two numbers.

```
package math;
  method add(double x, double y) returns double;
    return x+y;
  end;
endpackage;
```

In the next example, two numbers are added by using the ADD method in the MATH package that was created in the previous example. First, the MATH package is declared and instantiated. Then the ADD method is called and the result is assigned to SUM.

```
data _null_;
  dcl double sum;
  method init();
    dcl package math f();
    sum = f.add(2,3);
    put 'sum= ' sum;
  end;
enddata;
```

Predefined DS2 Packages

Overview of Predefined DS2 Packages

SAS provides the following predefined packages for use in the DS2 language:

FCMP

Supports calls to FCMP functions and subroutines from within the DS2 language.

For more information, see “Using the FCMP Package” on page 144.

Hash and hash iterator

Enables you to quickly and efficiently store, search, and retrieve data based on unique lookup keys. The hash package keys and data are variables. Key and data
values can be directly assigned constant values, values from a table, or values can be computed in an expression.

For more information, see “Using the Hash Package” on page 145 and “Using the Hash Iterator Package” on page 157.

HTTP
Constructs an HTTP client to access HTTP web services.

For more information, see “Using the HTTP Package” on page 158.

JSON
Enables you to create and parse JSON text.

Logger
Provides a basic interface (open, write, and level query) to the SAS logging facility.

For more information, see “Using the Logger Package” on page 164.

Matrix
Provides a powerful and flexible matrix programming capability. It provides a DS2-level implementation of SAS/IML functionality.

For more information, see “Using the MATRIX Package” on page 167.

SQLSTMT
Provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

For more information, see “Using the SQLSTMT Package” on page 172.

TZ
Provides a way to process local and international time and date values.

For more information, see “Using the TZ Package” on page 176.

Using the FCMP Package

Overview of FCMP Packages
The DS2 language supports calls to functions and subroutines that are available or are created with the FCMP procedure through an FCMP package.

You create an FCMP package by using the LANGUAGE=FCMP and TABLE= options in a PACKAGE statement. After the package is created, you declare an instance of the FCMP package. There are two ways to construct an instance of an FCMP package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:

```
declare package fcmp banking;
banking = _new_ fcmp();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package fcmp banking();
```

For more information, see “DECLARE PACKAGE Statement, FCMP Package” on page 997 and “PACKAGE Statement” on page 921.

FCMP Package Capabilities
These are the capabilities of using the FCMP package in the DS2 language:

- Call an FCMP function or subroutine with scalar DOUBLE, CHAR, and NCHAR parameters, return parameters of both.
The DS2 language does automatic type conversion so that almost any type is supported. An example is a conversion from TINYINT to DOUBLE. For more information, see “Overview of Type Conversions” on page 88.

- Call an FCMP function or subroutine with scalar DOUBLE OUTARGS parameters. The FCMP procedure’s DOUBLE OUTARGS parameter is treated as an IN_OUT parameter in the METHOD statement. For more information about the IN_OUT parameter, see the “METHOD Statement” on page 910.

  Note: DS2 must pass a DOUBLE variable, not an expression, through an OUTARGS parameter.

**Considerations and Limitations When Using the FCMP Package**

- The FCMP package does not support VARARGS functions calls and therefore cannot use the FCMP procedure’s VARARGS interface.

- Errors caused when information is passed between the FCMP procedure and the FCMP package are not always reported correctly. For example, if you supply an incorrect table name in the PACKAGE statement, an error is written in the log file. However, there is no indication given that the operation fails.

- The FCMP package assumes the session encoding and currently has no mechanism that allows different encodings for different parameters within the same function call or for the same parameter across multiple function calls.

- You can access any FCMP library as long as the connection string defines the catalog in which the FCMP library is located.

**Using the Hash Package**

**Overview of Hash Packages**

The hash package provides an efficient, convenient mechanism for quick data storage and retrieval. The hash package stores and retrieves data based on unique lookup keys. Depending on the number of unique lookup keys and the size of the table, the hash package lookup can be significantly faster than a standard format lookup or an array.

Before you use a DS2 hash package, you must define and construct an instance of (instantiate) the hash package.

After you define and create a hash package instance, you can perform many tasks, including the following:

- Store and retrieve data.
- Replace and remove data.
- Output a table that contains the data in the hash package.

For example, suppose that you have a large table that contains numeric lab results that correspond to a unique patient number and weight and a small table that contains patient numbers (a subset of those in the large table). You can load the large table into a hash package using the unique patient number as the key and the weight values as the data. You can then iterate over the small table using the patient number to look up the current patient in the hash package whose weight is over a certain value and output that data to a different table.
Defining and Creating a Hash Package Instance

To create an instance of a hash package, you provide keys, data, and optional initialization data about the hash instance to construct. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls.

In the following example, the hash instances, h1 and h2, have the same instance definition. The hash instance h1 is fully defined at construction while h2 is defined at construction and through a series of method calls.

```
declare package hash h1([key], [data1 data2 data3], 0, 'testdata', '', '', '', 'multidata');
declare package hash h2();
method init();
  h2.keys([key]);
  h2.data([data1 data2 data3]);
  h2.dataset('testdata');
  h2.multidata();
  h2.defineDone();
end;
```

For more information, see “Defining a Hash Instance By Using Constructors” on page 146 and “Defining a Hash Instance By Using Method Calls” on page 147.

Defining a Hash Instance By Using Constructors

A constructor is a method that you can use to instantiate a hash package and initialize the hash package data.

There are three different methods for creating a hash package instance with constructors.

- Create a partially defined hash instance.

  ```
  DECLARE PACKAGE HASH instance(hashexp, {'datasource' | '{sql-text}'}, 'ordered', 'duplicate', 'suminc', 'multidata');
  ```

  The key and data variables are defined by method calls. The optional parameters that provide the initialization data can be specified either in the DECLARE PACKAGE statement as shown above, in the _NEW_ operator, by method calls, or a combination of any of these. A single DEFINEDONE method call completes the definition.

- Create a completely defined hash instance with the specified key and data variables.

  ```
  DECLARE PACKAGE HASH instance([keys], [data])
  [, hashexp, {'datasource' | '{sql-text}'}, 'ordered', 'duplicate', 'suminc', 'multidata']);
  ```

  The key and data variables are defined in the DECLARE PACKAGE statement, which indicates that the instance should be created as completely defined. No additional initialization data can be specified with subsequent method calls.

- Create a completely defined hash instance with only the specified key variables (a keys-only hash instance). There are no data variables.

  ```
  DECLARE PACKAGE HASH instance([keys])[ hashexp, {'datasource' | '{sql-text}'},
  'ordered', 'duplicate', 'suminc', 'multidata']);
  ```

  The key and data variables are defined in the DECLARE PACKAGE statement, which indicates that the instance should be created as completely defined. No additional initialization data can be specified with subsequent method calls.
For more information about the optional parameters, see “Providing Initialization Data for a Hash Package” on page 148. For more information about defining the optional parameters using method calls, see “Defining a Hash Instance By Using Method Calls” on page 147.

Note: All variables that are passed to a hash instance must be global variables.

**Defining a Hash Instance By Using Method Calls**

If a hash instance is partially defined during construction of the instance, then the instance can be further defined through calls to the following methods.

- KEYS
- DEFINEKEY
- DATA
- DEFINEDATA
- DATASET
- DUPLICATE
- HASHEXP
- ORDERED
- MULTIDATA
- SUMINC
- DEFINEDONE

For more information about these methods, see Chapter 31, “DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements,” on page 1003.

Note: After a hash instance specification is completed by a call to the DEFINEDONE method, a subsequent call to any of the above methods results in an error.

The following is an example of a hash instance, h, defined by using the method calls.

```plaintext
data _null_;  
declare package hash h(0, 'testdata');  
method init();  
  h.keys([key]);  
  h.data([data1 data2 data3]);  
  h.ordered('descending');  
  h.duplicate('error');  
  h.defineDone();  
end;  
enddata;
```

**Defining Key and Data Variables**

The hash package uses unique lookup keys to store and retrieve data. The keys and the data are variables that you use to initialize the hash package by using dot notation method calls.

You can define the key and data variables in one of three ways.

- Use the variable methods, DEFINEDATA and DEFINEKEY.
- Use the variable list methods, DATA and KEYS.
- Use key and data variable lists specified in the DECLARE PACKAGE statement.

If an instance of the hash package is not completely defined at construction, that is keys and data variables are not specified at construction, you must call the DEFINEDONE method to complete initialization of the hash instance.
Here are examples.

```/* Keys and data defined using the implicit variable method */
declare package hash h();
h.definekey('k');
h.definedata('d');
h.definedone();
```

```/* Keys and data defined using the variable list methods */
declare package hash h();
h.keys([k]);
h.data([d]);
h.definedone();
```

```/* Keys and data defined using the variable list constructors */
declare package hash h([k],[d]);
```

Key variables must be a DS2 built-in type (character, numeric, or date-time). Data variables can be either a DS2 built-in type or a built-in or user-defined package type.

For more information, see “Implicit Variable and Variable List Methods” on page 149.

### Providing Initialization Data for a Hash Package

In addition to the keys and data, you can provide the following optional parameters when you initialize a hash package:

- the internal table size (`hashexp`) where the size of the hash table is $2^n$
- the name of the table to load (`datasource`) or a FedSQL query to select the data to load
- whether or how the data is returned in key-value order (`ordered`)
- whether to ignore duplicate keys when loading a table (`duplicate`)
- the name of a variable that maintains a summary count of hash package keys (`suminc`)
- whether multiple data items are allowed for each key (`multidata`)

You can specify the initialization data in the `DECLARE PACKAGE` statement, the `_NEW_` operator, by method calls, or a combination of these ways.

**Note:** When you initialize hash package data using a constructor in the `DECLARE PACKAGE` statement or the `_NEW_` operator, you must provide the optional parameters in this order: `hashexp`, `datasource`, `ordered`, `duplicate`, `suminc`, and `multidata`. These positional constructor parameters must all be enclosed in a single set of parentheses, separated by commas, and, except for the `hashexp` parameter, wrapped by single quotation marks. Because the optional parameters are positional, you must provide a placeholder for each parameter to the last parameter that you specify. The placeholder that must be used depends on the parameter. For more information about the placeholders, see the “`DECLARE PACKAGE` Statement, Hash Package” on page 1012 or the “`_NEW_` Operator, Hash Package” on page 1044.

In the following example, to specify an ascending order and to replace duplicates, you must use \(-1\) as a place holder for the `hashexp` parameter, empty single quotations marks (""") as the place holder for the `datasource` parameter, \('a'\) for `ordered` parameter, \('replace'\) for `duplicate`. Because the `duplicate` parameter is the last one specified, no place holder is required for the `suminc` and `multidata` parameters. Here is an example.

```declare package hash variable-name(8, '', 'a', 'replace');```
For more information, see “Defining a Hash Instance By Using Constructors” on page 146, “Defining a Hash Instance By Using Method Calls” on page 147, and “Using the _NEW_ Operator to Create a Hash Instance” on page 149.

**Using the _NEW_ Operator to Create a Hash Instance**

As an alternative to using the DECLARE PACKAGE statement to create a hash variable and a hash instance, you can use the DECLARE PACKAGE statement to create the hash variable and the _NEW_ operator to create the hash instance. You declare a hash package variable using the DECLARE PACKAGE statement. Then you use the _NEW_ operator to instantiate an instance of the hash package and set the hash variable to reference the newly instantiated hash instance. With this scenario, initialization data cannot be provided using the DECLARE PACKAGE statement. You can provide initialization data for the hash instance with the _NEW_ operator and subsequent method calls if the hash instance was not constructed fully defined.

In the following example the DECLARE PACKAGE statement tells the compiler that the variable MYHASH is of type hash package. At this point, you have declared only the variable MYHASH. It has the potential to reference a hash instance, but it currently references nothing and therefore is a null package reference. You should declare the hash variable package only once. The _NEW_ operator creates an instance of the hash package and assigns it to the variable MYHASH.

```
declare package hash myhash();
myhash = _new_ hash(8, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

The above statement is equivalent to the following code:

```
declare package hash myhash(8, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

For more information, see the “_NEW_ Operator, Hash Package” on page 1044.

**Implicit Variable and Variable List Methods**

When you define a hash instance, you specify a series of key and data variables. After the hash instance is completely defined, the key and data variables can be implicitly or explicitly read and written during subsequent operations.

**Note:** All variables that are passed to a hash instance must be global variables.

There are two ways to pass keys and data variables to the hash instance: implicit variables methods and variable list methods.

The implicit variable method is similar to the DATA step hash object interface. Using the implicit variable methods, the key and data variables are defined through a series of DEFINEKEY and DEFINEDATA method calls and a single DEFINEDONE method call. Then the set of key and data variable definitions is used during execution of the other hash package methods as implicit arguments.

The following example defines a hash table with two key variables, k1 and k2, and two data variables, d1 and d2. The FIND method reads the values of the implicit key variables, looks up the key values in the hash table. If the key values are found, DS2 writes the corresponding data values to the implicit data variables defined for the hash instance.

```
declare package hash h();
h.definekey('k1');
h.definekey('k2');
h.definedata('d1');
h.definedata('d2');
h.definedone();
```
The variable list method involves specifying variables of interest in variable lists as explicit arguments when the hash methods are called.

This example uses the variable list methods and is the same as the one above that uses implicit variable methods.

```sas
declare package hash h();
rc=h.keys([k1 k2]);
rc=h.data([d1 d2]);
h.find([k1 k2], [d1 d2]);
```

The variable list method provides flexibility to use variables other than the implicit key and data variables. The following FIND method looks for the values that are stored in \( x \) and \( y \). If the values are found, they are written to \( u \) and \( v \).

```sas
h.find([x y], [u v]);
```

For more information about variable lists, see “Variable Lists” on page 59.

All of the hash implicit variable methods work with hash instances that have both keys and data and with hash instances that are keys-only.

Some variable list methods only work with keys-only hash instances while others work only with hash instances that have both keys and data. A run-time error occurs if a keys-only hash instance invokes a variable list method that only works for a hash instance that has keys and data, and vice versa. For more information about which methods work with keys-only hash instances, see each method in Chapter 31, “DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements,” on page 1003.

### Non-Unique Key and Data Pairs

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

For example, assume that the key is a patient ID and the data is a visit date. If the patient were to visit multiple times, multiple visit dates would be associated with the patient ID. When you create a hash package with the MULTIDATA parameter or method set to YES, multiple sets of the data variables are associated with the key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE parameter or method. The DUPLICATE parameter or method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

To traverse a multiple data item list, you must know the current list item. Start by calling the FIND method for a given key. The FIND method sets the current list item. Then to determine whether the key has multiple data values, call the HAS_NEXT method. After you have determined that the key has another data value, you can retrieve that value with
the FIND_NEXT method. The FIND_NEXT method sets the current list item to the next
item in the list and sets the corresponding data variable or variables for that item.

In addition to moving forward through the list for a given key, you can loop backward
through the list by using the HAS_PREV and FIND_PREV methods in a similar manner.

Note: The items in a multiple data item list are maintained in the order in which you
insert them.

For more information about the MULTIDATA and DUPLICATE parameters, see the
“DECLARE PACKAGE Statement, Hash Package” on page 1012 or the “NEW
Operator, Hash Package” on page 1044. For more information about the MULTIDATA
and DUPLICATE methods, see the “MULTIDATA Method” on page 1043 and the
“DUPLICATE Method” on page 1026.

**Maintaining Key Summaries**

You can maintain a summary count for a hash package key by using the SUMINC
parameter or method. SUMINC instructs the hash package to allocate internal storage in
each record to store a summary value in the record each time that the record is used by a
FIND, CHECK, or REF method. The SUMINC value is also used to maintain a
summary count of hash parameter keys after a FIND, CHECK, or REF method.

SUMINC is given a variable, which holds the sum increment, that is, how much to add
to the key summary for each reference to the key. The SUMINC value can be greater
than, less than, or equal to 0. The SUMINC value is also used to initialize the summary
on an ADD method. Each time the ADD method occurs, the key to the SUMINC value
is initialized.

The SUM method retrieves the summary value for a given key when only one data item
exists per key.

If multiple items exist, the SUMDUP method retrieves the current value of the key
summary.

You can use key summaries in conjunction with the DATASOURCE parameter or DATA
method. As a table is read into the hash package using the DEFINEDONE method or a
DECLARE PACKAGE statement, all key summaries are set to the SUMINC value and
all subsequent FIND, CHECK, or ADD methods change the corresponding key
summaries.

For more information about the SUMINC parameter, see the “DECLARE PACKAGE
Statement, Hash Package” on page 1012. For more information about the SUMINC and
SUMDUP methods, see the “SUMINC Method” on page 1072 and the “SUMDUP
Method” on page 1069.

**Storing and Retrieving Data**

After you initialize the hash package's key and data variables, you can store data in the
hash package using the ADD method, or you can use the DATASOURCE parameter or
DATASET method to load a table into the hash package. If you use the DATASOURCE
parameter or DATASET method, and if the table contains more than one row with the
same value of the key, by default, SAS keeps the first row in the hash table and ignores
subsequent rows. To store the last instance in the hash package or to send an error to the
log if there is a duplicate key, use the DUPLICATE parameter or method. To allow
duplicate values for each key, use the MULTIDATA parameter or method.

You can then use the FIND method to search and retrieve data from the hash package.
Use the FIND_NEXT and FIND_PREV methods to search and retrieve data if multiple
data items exist for each key.
For more information, see the “ADD Method, Hash Package” on page 1004, “FIND Method” on page 1027, the “FIND_NEXT Method” on page 1028, and the “FIND_PREV Method” on page 1031.

You can consolidate a FIND method and ADD method using the REF method. In the following example, you can reduce the amount of code from this:

```plaintext
rc = h.find();
if (rc != 0) then
    rc = h.add();
```

to a single method call:

```plaintext
rc = h.ref();
```

For more information, see the “REF Method” on page 1056.

Note: You can also use the hash iterator package to retrieve the hash package data, one data item at a time, in forward and reverse order. For more information about the hash iterator package, see “Using the Hash Iterator Package” on page 157.

**Replacing and Removing Data**

You can remove or replace data in the hash package using one of the following methods:

- Use the REMOVE method to remove the data items from the specified key.
- Use the REMOVEALL method to remove all the data items.
- Use the REMOVEDUP method to remove data items for keys that have multiple data items.
- Use the REPLACE method to replace all data items.
- Use the REPLACEDUP method to replace only the current data item.

*Note:* If an associated hash iterator is pointing to the key, the REMOVE method does not remove the key or data from the hash package. An error message is issued to the log.

For more information, see the “REMOVE Method” on page 1057, the “REMOVEALL Method” on page 1060, the “REMOVEDUP Method” on page 1061, the “REPLACE Method” on page 1062, and the “REPLACEDUP Method” on page 1064.

**Saving Hash Package Data in a Table**

You can create a table that contains the data in a specified hash package by using the OUTPUT method.

In the following example, the first table program generates the `data1` table. The second table program creates a hash package `h` with one key, `k`, and two data, `d1` and `d2`. Then each row from the `data1` table is read and the keys and data are added to hash package `h`. Finally, the data values stored in hash package `h` are output to the `out1` table. The third table program outputs the contents of the `out1` table.

```plaintext
/* Generate and output 5 rows for table data1. */
data data1(overwrite=yes);
declare double k d1 d2;
method init();
declare int i;
do i = 1 to 5;
    k = i; d1 = i*10; d2 = i*2; output;
end;
end;
```
enddata;
run;

data _null_
   declare double k d1 d2;
   declare package hash h(0, '', 'descending');
/* Define key and data variables for hash h. */
method init();
   h.defineKey('k');
   h.defineData('d1');
   h.defineData('d2');
   h.defineDone();
end;
/* Read rows from table data1.
* Add key and data values from rows to hash h. */
method run();
   set data1;
   h.add();
end;
/* Add additional key and data values to hash h.
* Output hash h to table out1. */
method term();
   k = 11; d1 = 110; d2 = 22; h.add();
   k = 12; d1 = 120; d2 = 24; h.add();
   k = 13; d1 = 130; d2 = 26; h.add();
   k = 14; d1 = 140; d2 = 28; h.add();
   h.output('out1');
end;
enddata;
run;
/* Outputs rows from table out1. */
data;
   method run();
      set out1;
   end;
enddata;
run;

Predefined DS2 Packages  153
Note that the hash package keys are not stored as part of the output table. If you want to include the key in the output table, you must define the key as data in the DEFINEDATA method. In the previous example, the DEFINEDATA method would be written this way:

```javascript
h.defineKey('k');
h.defineData('k');
h.defineData('d1');
h.defineData('d2');
```
With the above modification, the following lines output.

**Output 16.2** Output Rows from Table OUT1 with Keys and Data

<table>
<thead>
<tr>
<th>k</th>
<th>d1</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>140</td>
<td>28</td>
</tr>
<tr>
<td>13</td>
<td>130</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>120</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

**Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time**

You can delay the decision of which rows to get from a table until run time by using a hash instance. At run time, the hash instance is created and loaded with the selected rows. The rows are selected based on a FedSQL query specified for the data source. You can use a hash iterator to loop over the rows and access the row data.

In the following example, the rows are selected by the `SELECT * FROM test WHERE a=1` FedSQL query. This query is passed to the `execute` method that loads them into the hash package. A hash iterator loops over the rows and outputs the selected rows to the SAS log and the `result` table.

```sas
data test;
  a=1;  b=2; output;
  a=11; b=22; output;
  a=1;  b=3; output;
  a=22; b=44; output;
run;

proc ds2;
package pkg /overwrite=yes;
  dcl double a b;
  
  method execute(char(200) sql);
    dcl package hash  h();
    dcl package hiter hi(h);
    h.keys([a]);
    h.data([a b]);
    h.multidata('yes');
```
h.dataset(sql);
h.definedone();

put 'SELECTED ROWS:';
rc = hi.first();
do while(rc = 0);
   put a= b=;
   rc = hi.next();
end;

h.output('result');
end;
endpackage;
run; quit;

proc ds2;
data _null_;  
   method init();
      declare package pkg p1();
      p1.execute('{SELECT * FROM test WHERE a=1}');
   end;
enddata;
run; quit;

proc print data=test;
   title2 'TEST TABLE';
run; quit;

proc print data=result;
   title2 'RESULT TABLE';
run; quit;

The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>SELECTED ROWS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1 b=2</td>
</tr>
<tr>
<td>a=1 b=3</td>
</tr>
</tbody>
</table>

The input and output tables are as follows.
**Using Hash Package Attributes**

There are two attributes available to use with hash packages. NUM_ITEMS returns the number of items in a hash package and ITEM_SIZE returns the size (in bytes) of an item.

The following example retrieves the number of items in a hash package:

```plaintext
num_items = myhash.num_items;
```

The following example retrieves the size of an item in a hash package:

```plaintext
item_size = myhash.item_size;
```

You can obtain an idea of how much memory the hash package is using with the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash package requires, nor does it take into account any necessary internal alignments. Therefore, the use of ITEM_SIZE does not provide exact memory usage, but it gives a good approximation. For more information, see the “ITEM_SIZE Attribute” on page 1040 and “NUM_ITEMS Attribute” on page 1050.

**Using the Hash Iterator Package**

You use a hash iterator package to store and search data based on unique lookup keys. The hash iterator package enables you to retrieve the hash package data in forward or reverse key order.
You declare a hash iterator package by using the DECLARE PACKAGE statement. After you declare the new hash iterator package, use the _NEW_ operator to instantiate the package, using the hash package name as a parameter. For example:

```
declare package hiter myiter;
myiter = _new_ hiter('h');
```

The DECLARE PACKAGE statement tells the compiler that the variable MYITER is of type hash iterator. At this point, you have declared only the variable MYITER. It has the potential to reference a hash iterator instance, but it currently references nothing and thus is a null package reference. You should declare the hash iterator package variable only once. The _NEW_ operator constructs an instance of the hash iterator package and assigns it to the variable MYITER. The hash package, H, is passed as a constructor parameter.

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operators to declare and instantiate a hash iterator package, you can declare and instantiate a package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package hiter myiter('h');
```

For more information, see the “DECLARE PACKAGE Statement, Hash Iterator Package” on page 1020, and the “_NEW_ Operator, Hash Iterator Package” on page 1048.

Note: You must declare and instantiate a hash package before you create a hash iterator package.

---

**Using the HTTP Package**

**Overview of the HTTP Package**

Use the HTTP package to construct an HTTP client in order to access HTTP web servers.

Here are the general tasks:

1. Declare and instantiate an HTTP package.
2. Create an HTTP GET, HEAD, or POST method.
   
   You can use additional HTTP package methods to add header information and to send request data.
3. Execute the HTTP GET, HEAD, or POST method.
4. Retrieve the response information from the web server:
   
   - The response body as a complete entity or by streaming
   - The response content type
   - The response header

The HTTP package also enables you to perform these tasks:

- Retrieve status codes from HTTP responses.
- Set a socket time-out value.
- Log the HTTP traffic between the HTTP client and server using the SAS logging facility.
Declaring and Instantiating an HTTP Package

You must first declare and instantiate an HTTP package. There are two ways to construct an instance of an HTTP package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  
  ```
  declare package http httpclt;
  httpclt = _new_ http();
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:
  
  ```
  declare package http httpclt();
  ```

For more information, see “DECLARE PACKAGE Statement, HTTP Package” on page 1083 and “_NEW_ Operator, HTTP Package” on page 1092.

TIP Web service applications might require only a single HTTP client to synchronously handle HTTP traffic. Or, if your application requires it, you can instantiate multiple HTTP clients to asynchronously request and process data.

Create an HTTP GET, HEAD, or POST Method

1. Use the CREATEGETMETHOD, CREATEHEADMETHOD, or CREATEPOSTMETHOD method to create the GET, HEAD, and POST method.
   
   For more information, see “CREATEGETMETHOD Method” on page 1077, “CREATEHEADMETHOD Method” on page 1081, and “CREATEPOSTMETHOD Method” on page 1082.

2. (Optional) To add a header to the HTTP GET method, use the ADDREQUESTHEADER method.
   
   For more information, see “ADDREQUESTHEADER Method” on page 1076.

3. (Optional) To add a request body to the HTTP method, use the SETREQUESTBODYASBINARY or SETREQUESTBODYASSTRING method.
   
   For more information, see “SETREQUESTBODYASBINARY Method” on page 1093 and “SETREQUESTBODYASSTRING Method” on page 1094.

4. (Optional) To specify the request content type in the HTTP method, use the SETREQUESTCONTENTTYPE method.
   
   For more information, see “SETREQUESTCONTENTTYPE Method” on page 1095.

Executing an HTTP GET, HEAD, and POST Method

When you execute the HTTP GET, HEAD, and POST methods, you send the request to an HTTP web server.

Note: Before you execute the HTTP GET, HEAD, and POST methods, you must create the HTTP GET, HEAD, or POST method. For more information, see “Create an HTTP GET, HEAD, or POST Method” on page 159.

For most HTTP methods, use the EXECUTEMETHOD method to send the request to the web server. For more information, see “EXECUTEMETHOD Method” on page 1084.

The EXECUTEMETHOD method does not support streaming of the response body. If you want to stream the response body, a different execute method, EXECUTEMETHODSTREAM, is required. For more information, see “Retrieving an HTTP Resource” on page 160.

To send another request, repeat the process starting with creating the GET, HEAD, or POST request and ending with the EXECUTEMETHOD or EXECUTEMETHODSTREAM method.
This example program instantiates an HTTP package (the client), creates an HTTP GET method, executes the GET method to send a request to an HTTP web service, and retrieves the body information from the response from the HTTP web service as a string.

data _null_;  
method run();  
   /* instantiate the package */  
   declare package http h();  
   declare varchar(1024) character set utf8 body;  
   declare int rc;  

   /* create a GET */  
   h.createGetMethod('http://api.worldbank.org/countries/fr/');  
   /* execute the GET */  
   h.executeMethod();  
   /* retrieve the response body as a string */  
   h.getResponseBodyAsString(body, rc);  
   put body;  
end;

The following lines are written to the SAS log.

```xml
<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1" pages="1" per_page="50" total="1" xmlns:wb="http://www.worldbank.org">
   <wb:country id="FRA">
      <wb:name>France</wb:name>
      <wb:region id="ECS">Europe &amp; Central Asia (all income levels)</wb:region>
      <wb:adminregion id="" />
      <wb:incomeLevel id="OEC">High income: OECD</wb:incomeLevel>
      <wb:lendingType id="LNX">Not classified</wb:lendingType>
      <wb:capitalCity>Paris</wb:capitalCity>
      <wb:longitude>2.35097</wb:longitude>
      <wb:latitude>48.8566</wb:latitude>
   </wb:country>
</wb:countries>
```

**Retrieving an HTTP Resource**

You can use the following HTTP package methods to retrieve headers, the response body, and the response body type from an HTTP resource. If you are retrieving the response body, you can retrieve it as one entity or stream the response body.

<table>
<thead>
<tr>
<th>Task</th>
<th>HTTP package method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve header</td>
<td>“GETRESPONSEHEADERSASSTRING Method” on page 1090</td>
</tr>
<tr>
<td>Get response body as one entity*</td>
<td>“GETRESPONSEBODYASBINARY Method” on page 1086</td>
</tr>
<tr>
<td></td>
<td>“GETRESPONSEBODYASSTRING Method” on page 1087</td>
</tr>
</tbody>
</table>
Considerations When Using the HTTP Package

• The HTTP package supports only GET, HEAD, and POST HTTP methods.

• Each client sends requests and processes responses synchronously. The application can create multiple clients to asynchronously perform actions on HTTP resources.

• The HTTP package stores data for requests and from responses in memory as DS2 string values or binary values. If you want to store the data on disk as a file, consider using the HTTP procedure. For more information, see “HTTP” in Base SAS Procedures Guide.

• The HTTP package can send requests to a secure HTTP endpoint that requires authentication. When an HTTP end-point requires client authentication, it responds to the client with its list of supported authentication mechanisms. The HTTP package currently supports two of the three most common authentication mechanisms: Basic and Negotiate. Since Basic authentication in itself does not provide any credential confidentiality, it should be used only when the data is being encrypted via Transport Layer Security (TLS). For complete information about how SAS validates TLS, see “Installing and Configuring TLS and Certificates” in Encryption in SAS. Negotiate authentication supports Kerberos, and, when on Windows, supports NT LAN Manager (NTLM).

Logging HTTP Traffic

The HTTP package supports logging through the SAS logging facility.

The App.TableServices.d2pkg.HTTP logger logs errors, headers, and data that are sent back and forth from the HTTP client and the web server.

For more information, see “HTTP Package Logger” on page 1302.

Using the JSON Package

Overview of the JSON Package

Java Script Object Notation (JSON) is a text-based, open standard data format that is designed for human-readable data interchange. JSON is based on a subset of the JavaScript programming language and uses JavaScript syntax for describing data objects.

The JSON package provides an interface to create and parse JSON text. The JSON package Write methods accumulate the Write requests in memory, and the text can be retrieved. The JSON package parser enables you to read and parse text.
Declaring and Instantiating the JSON Package

There are two ways to construct an instance of a JSON package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  ```
  declare package json myjsonpkg;
  myjsonpkg = _new_ json();
  ```
- Use the DECLARE PACKAGE statement along with its constructor syntax:
  ```
  declare package json myjsonpkg();
  ```

For more information, see the “DECLARE PACKAGE Statement, JSON Package” on page 1101, and the “_NEW_ Operator, JSON Package” on page 1113.

Writing JSON Text

To create JSON text, you create a JSON writer instance by using the CREATEWRITER method.

JSON output consists of two types of data structure containers:

- **JSON object container ({}):**
  
  Begins with a left brace ({}) and ends with a right brace (}). An object container collects name-value pairs that are written as pairs of names and values. A value can be any of the supported JSON data types, an object, or an array. Each name is followed a colon and then the value. The name-value pairs are separated by a comma.

  Use the WRITEOBJOPEN and WRITECLOSE methods to create the object container.

- **JSON array container ([ ]):**
  
  Begins with a left bracket ([) and ends with a right bracket (]). An array container collects a list of values that are written as a list of values without names. A value can be any of the supported JSON data types, an object, or an array. Values are separated by a comma.

  Use the WRITEARRAYOPEN and WRITECLOSE methods to create the array container.

The top-level container can include any number of containers. Containers, likewise, can nest containers to an arbitrary depth. When nesting containers, be careful to observe the data structure requirements of the current container.

- Objects require a list of name-value pairs, where the value can itself be an object or array.
- Arrays have no such structural requirement of name-value pairs and are merely a list of values, objects, or arrays.

With JSON package methods, you can write character and numeric values, null values, and Boolean true and false values.

Here is an example of JSON text output:

```json
{"SASJSONExport":"1.0","SASTableData+CLASS": [{"Name":"Joyce","Sex": "F","Age":11,"Height":51.3,"Weight":50.5},{"Name":"Thomas","Sex":"M","Age":11,"Height":57.5,"Weight":85}]}
```

You can get the JSON text that is produced by the writer by using the WRITERGETTEXT method.

When you finish writing the text, call the DESTROYWRITER method to remove the writer instance.
The following example creates some JSON text and writes it to the SAS log.

```sas
data _null_;   
  method init(); 
    dcl package json j(); 
    dcl double dblVal; 
    dcl int rc; 
    dcl nvarchar(30) jsontxt; 
    
    rc = j.createWriter(); 
      if rc=0 then rc = j.writeObjectOpen(); 
      if rc=0 then rc = j.writeString(' Hello World! '); 
      if rc=0 then rc = j.writeClose(); 
    j.writerGetText(rc, jsontxt); 
    put rc= jsontxt=; 
  end; 
enddata; 
run; 
```

The following line is written to the SAS log.

```
rc=0 jsontxt=" Hello World! "]
```

For more information about the methods that enable you to write JSON text, see on page 1099.

**Parsing JSON Text**

To parse, or read, JSON text, you create a JSON writer instance by using the CREATEPARSER method. You can provide the input JSON text to the parser with either the CREATEPARSER method, the SETPARSERINPUT method, or both.

The GETNEXTTOKEN method is used to return the next validate JSON language element form the JSON text. The GETNEXTTOKEN method can also return the token type, parse flags, the line number, and the column number. The JSON package IS* methods (for example, ISLEFTBRACE) enable you to query the following token types:

- Boolean true
- Boolean false
- float
- integer
- label
- left brace ( { )
- left bracket ( [ )
- null
- numeric
- partial
- right brace ( } )
- right bracket ( ] )
- string

When you finish parsing the text, call the DESTROYPARSER method to remove the parser instance.

The following example creates a parser and uses the CREATEPARSER method to provide the input JSON text.
data _null_;  
    method init();  
    dcl package json j();  
    dcl int rc tokenType parseFlags;  
    dcl bigint lineNum colNum;  
    dcl nvarchar(128) token abc t1;  
    abc = 'xyz';  
    t1 = '{"abc" : 1 }';  
    rc = j.createParser( t1 );  
    if (rc ne 0) then goto TestError;  

    * obj open;  
    j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );  
    if ( rc ne 0 ) then goto TestError;  

    * obj label;  
    j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );  
    if ( rc ne 0 ) then goto TestError;  

    * obj value;  
    j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );  
    if ( rc ne 0 ) then goto TestError;  

    * obj close;  
    j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );  
    if ( rc ne 0 ) then goto TestError;  

    Exit:  
    rc = j.destroyParser();  
    return;  

    TestError:  
    put 'Test ended abnormally.';  
    goto Exit;  

end;  
enddata;  
run;  

For more information about the methods that enable you to parse JSON text, see on page 1099.

**Using the Logger Package**

**Overview of the Logger Package**

In the SAS logging facility, a logger is a named entity that identifies a message category. A logger's attributes consist of a level and one or more appenders that process the log events for the message category. The level indicates the threshold, or lowest event level, that will be processed for this message category.

You use a logger package to interface with the SAS logging facility. After you declare the new logger package, you can send messages to the logger at a specified logging level.
Declaring and Instantiating a Logger Package

There are two ways to construct an instance of a logger package.

- Use the DECLARE PACKAGE statement along with the \_NEW\_ operator:

  ```
  declare package logger logpkg;
  logpkg = _new_ logger();
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

  ```
  declare package logger logpkg();
  ```

For more information, see the “DECLARE PACKAGE Statement, Logger Package” on page 1125, and the “\_NEW\_ Operator, Logger Package” on page 1130.

Unformatted and Formatted Messages

You can specify a raw, or unformatted, message that is output directly as written to the SAS logging facility.

You can also use $s format markers to create a formatted message. Each $s format marker in the message format is replaced by the content of the corresponding argument that you specify.

For more information, see “LOG Method, Logger Package” on page 1128.

Log Messages to a Table

1. Create an empty table to store the messages.

   Here is an example:

   ```
   libname logs 'c:\temp';
   data logs.edmlog;
     length seqno  8;
     length date   8; format date DATETIME19.;
     length msg    $ 256;
     stop;
   run;
   ```

2. Create a logger and appender with an XML logging configuration file.

   Here is an example that creates an App.Program logger named **App.Program.EDM**. The appender that the logger writes to is **EDMAppender**. The appender is configured to write to a SAS data set name **edmlog** in **c:\temp**. Save this XML logging configuration file as **edm-l4s.xml**.

   ```xml
   <?xml version="1.0"?>
   <logging:configuration xmlns:logging="http://www.sas.com/xml/logging/1.0/">
     <appender name="EDMAppender" class="DBAppender">
       <param name="ConnectionString" value="DRIVER=base;CATALOG=base;
       schema=(name=base;primarypath='C:\temp')/"/>
       <param name="MaxBufferedEvents" value="1000"/>
       <param name="TableName" value="edmlog"/>
       <param name="Column" value="sn"/>
       <param name="Column" value="d"/>
   ```
3. Start SAS with the LOGCONFIGLOC system option set to the name of the XML configuration file that you created in Step 2.

Here is an example.

```sas
options logconfigloc edm-l4s.xml
```

4. Create a DS2 logger package instance that is associated with the logger that you created in the XML configuration file from Step 2.

Here is an example.

```sas
proc ds2;
data _null_;  
dcl package logger l('App.Program.EDM');  
method init();  
dcl double i;  
i = l.islevelactive(4);  put i=;  
l.log('T', 'Hello World! Trace');  
l.log('D', 'Hello World! Debug');  
l.log('I', 'Hello World! Info');  
l.log('W', 'Hello World! Warning');  
l.log('E', 'Hello World! Error');  
end;  
enddata;  
run;  
quit;
```

You can print the contents of the `edmlog` data set.

```sas
libname logs 'c:\temp';  
proc print data=logs.edmlog; run;
```

The data set contents looks like this.

<table>
<thead>
<tr>
<th>Obs</th>
<th>seqno</th>
<th>date</th>
<th>msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>285</td>
<td>25APR2013:17:11:42</td>
<td>Hello World! Trace</td>
</tr>
<tr>
<td>2</td>
<td>286</td>
<td>25APR2013:17:11:42</td>
<td>Hello World! Debug</td>
</tr>
<tr>
<td>3</td>
<td>287</td>
<td>25APR2013:17:11:42</td>
<td>Hello World! Info</td>
</tr>
<tr>
<td>4</td>
<td>288</td>
<td>25APR2013:17:11:42</td>
<td>Hello World! Warning</td>
</tr>
<tr>
<td>5</td>
<td>289</td>
<td>25APR2013:17:11:42</td>
<td>Hello World! Error</td>
</tr>
</tbody>
</table>

For more information about the SAS logging facility, see *SAS Logging: Configuration and Programming Reference*. 
Using the MATRIX Package

Overview of the MATRIX Package
A matrix is a two-dimensional array of numeric or character values. The dimensions of a matrix are defined by the number of rows and columns. The elements of an \( n \times p \) matrix are arranged in \( n \) rows and \( p \) columns.

The matrix package provides a DS2-level implementation of SAS/IML functionality. You can use matrix package methods to perform complex tasks such as matrix inversion. You can perform arithmetic, relational, and logical operations. You can perform some operations on an elementwise basis and other operations on the entire data matrix.

You can load data in a matrix package using an array or external data. You can output data by writing the entire matrix to an array at one time or row-by-row. The array can then be written to a result table.

For more information about these methods, see Chapter 35, “DS2 Matrix Package Methods, Operators, and Statements,” on page 1133.

Declaring and Instantiating a MATRIX Package
A matrix package is created by declaring and instantiating the matrix package using the DS2 DECLARE PACKAGE statement. Here is an example:

```sas
declare package matrix m(2, 2);
```

This statement creates a 2 x 2 matrix and stores its instance in the variable `m`.

Note: The matrix is filled with zeros, not null or missing values.

For more information, see “DECLARE PACKAGE Statement, Matrix Package” on page 1155.

Matrix Data Input
There are two ways to initialize or load data into a matrix package.

- initialize with zero values
  - declare package matrix instance-name(rows,columns);
  - instance-name=_new_matrix(rows,columns);
- initialize with array values
  - /* loads an array using the array name */
    ```sas
    instance-name=_new_matrix(array-name, rows, columns);
    ```
  - /* loads an array using the IN method */
    ```sas
    instance-name=_new_matrix(rows,columns);
    instance-name.in(array-name);
    ```
  - /* loads row-by-row using an input table*/
    /* a variable array, and the SET statement */
    ```sas
    instance-name=_new_matrix(rows,columns);
    ...
    set table-name;
    instance-name.in(variable-array-name, i);
    ```
These examples create a 2x2 matrix that is initialized with zero values.

```plaintext
declare package matrix m(2,2);
m=_new_ matrix(2, 2);
```

These examples load array $a$ into the matrix $m$.

/* simple array */
method init();
dcl double a[3, 3];
dcl package matrix m;

```
a :=(1, 2, -1, 2, 1, 0, -1, 1, 2);
m=_new_ matrix(a, 3, 3);
end;
```

/* variable array */
vararray double a[3,3];
dcl package matrix m;

```plaintext
method init();
a := (1, 2, -1, 2, 1, 0, -1, 1, 2);
m = _new_ matrix(a, 3, 3);
end;
```

These examples load array $va$ into the matrix $m$ using the IN method.

/* simple array */
dcl double va[3,3];
dcl package matrix m;
m = _new matrix(3,3);
m.in(va);

/* variable array */
vararray double va[3,3];
dcl package matrix m;
m.in(va); */

This example reads table $x$ using the SET statement into variable array $a$. That variable array is then used to load the matrix $m$.

vararray double a[4];
dcl package matrix m;

/* Create an empty matrix to hold the input values */
method init();
```
m = _new_ [this] matrix(4, 4);
i = 1;
end;
```

/* Read and initialize each row of matrix from vararray a */
method run();
```
set x;
m.in(a, i);
i + 1;
end;
```
For more information, see the “_NEW_ Operator, Matrix Package” on page 1181 and the “IN Method” on page 1172.

Note: The _NEW_ operator is part of the code stream. It is not used in the declarations.

**Matrix Data Output**

You can output matrix data by writing the entire matrix to an array at one time or row-by-row.

Use the TOARRAY or TOVARARRAY methods to write the matrix to an array at one time. In this example, array a is loading into matrix m. The transpose of matrix m is then calculated and is output to array c.

dcl double a[3,3];
dcl double b[3,3];

method run();
  dcl package matrix m r;
  dcl double i j;

  a := {1,2,3,4,5,6,7,8,9};

  m = _new_ matrix(a, 3, 3);
  r = m.trans();
  r.toarray(b);

  do i = 1 to 3;
    do j = 1 to 3;
      put b[i,j];
    end;
  end;
end;

For more information, see the “TOARRAY Method” on page 1192 and the “TOVARARRAY Method” on page 1194.

Variable arrays can be used to output data by using the OUT method and the OUTPUT statement. The OUT method writes the matrix row data to a variable array. The OUTPUT statement writes the data to a result table. The matrices are output one row at a time to the result table a row at a time by means of the variable array. For more information, see the “OUT Method” on page 1188 and the “Example 1: Loading and Writing Data” on page 1173.

**Matrix Operations**

The following table summarizes the operations that can be performed on matrices using typical DS2 dot syntax method calls.
<table>
<thead>
<tr>
<th>Type of Operation</th>
<th>Method Name</th>
<th>Operation performed</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Binary Arithmetic | ADD         | Addition                                 | • The array dimensions for the matrices used in addition or subtraction operations do not have to be compatible. But the number of columns in the first matrix has to equal the number of rows in the second matrix or the second matrix has to be a 1x1 matrix.  
• The array dimensions for the matrices used in a multiplication operation have to be compatible. The number of columns in the first matrix must equal the number of rows in the second matrix. |
|                   | MULT        | Multiplication                            |                                                                                                                |
|                   | SUB         | Subtraction                               |                                                                                                                |
| Unary             | ABS         | absolute value                            | • Unary operations are performed on a single matrix.  
• Matrices must be square or singular. For inverse and determinant operations, the matrix must be square.                                                   |
|                   | COPY        | copy matrix                               |                                                                                                                |
|                   | DET         | determinant of a square matrix            |                                                                                                                |
|                   | EXP         | exponential value                         |                                                                                                                |
|                   | FLOOR       | integer part of each matrix value         |                                                                                                                |
|                   | INVERSE     | inverse                                   |                                                                                                                |
|                   | LOG         | natural logarithm                         |                                                                                                                |
|                   | SQRT        | square root                               |                                                                                                                |
|                   | TRANS       | transposition                             |                                                                                                                |
| Binary relational | EQ          | equal to                                  | • The result of any binary relational operation is a matrix whose entries tell how the $[i,j]^{th}$ element of the first matrix compares to the $[i,j]^{th}$ element of the second matrix. The result values are either 0 if the comparison is false or 1 if the comparison is true.  
• The matrix sizes must match or you can use a scalar comparison. |
|                   | GT          | greater than or equal to                  |                                                                                                                |
|                   | GE          | greater than                              |                                                                                                                |
|                   | LE          | less than or equal to                     |                                                                                                                |
|                   | LT          | less than                                 |                                                                                                                |
|                   | NE          | not equal to                              |                                                                                                                |
| Binary logical    | AND         | and comparison                            | • The result of any binary logical operation is a matrix whose entries tell how the $[i,j]^{th}$ element of the first matrix compares to the $[i,j]^{th}$ element of the second matrix. The result values are either 0 if the comparison is false or 1 if the comparison is true. |
|                   | OR          | or comparison                             |                                                                                                                |
| ALL relational    | ALL_EQ      | ALL equal                                 | • The ALL relational operations produce a scalar result that indicates whether the $[i,j]^{th}$ element of the first matrix satisfies the comparison with the $[i,j]^{th}$ element of the second matrix. The scalar result is 0 or 1. All of the $[i,j]$ element comparisons must be true in order for the result to be 1. Otherwise, the result is 0.  
• The matrix sizes must match or you can use a scalar comparison. |
<p>|                   | ALL_GE      | ALL greater than or equal to              |                                                                                                                |
|                   | ALL_GT      | ALL greater than                          |                                                                                                                |
|                   | ALL_LE      | ALL less than or equal to                 |                                                                                                                |
|                   | ALL_LT      | ALL less than                             |                                                                                                                |
|                   | ALL_NE      | ALL not equal to                          |                                                                                                                |</p>
<table>
<thead>
<tr>
<th>Type of Operation</th>
<th>Method Name</th>
<th>Operation performed</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY relational</td>
<td>ANY_EQ</td>
<td>ANY equal</td>
<td>• The ANY relational operations produce a scalar result that indicates whether the ([i, j])th element of the first matrix satisfies the comparison with the ([i, j])th element of the second matrix. The scalar result is 0 or 1. If any of the ([i, j]) element comparisons is true, the result is 1. Otherwise, the result is 0.</td>
</tr>
<tr>
<td></td>
<td>ANY_GE</td>
<td>ANY greater than or equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY_GT</td>
<td>ANY greater than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY_LE</td>
<td>ANY less than or equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY_LT</td>
<td>ANY less than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANY_NE</td>
<td>ANY not equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The matrix sizes must match or you can use a scalar comparison.</td>
</tr>
<tr>
<td>ALL logical</td>
<td>ALL_AND</td>
<td>ALL AND</td>
<td>• The ALL logical operations produce a scalar result that indicates whether the ([i, j])th element of the first matrix satisfies the comparison with the ([i, j])th element of the second matrix. The scalar result is 0 or 1. All of the ([i, j]) element comparisons must be true in order for the result to be 1. Otherwise, the result is 0.</td>
</tr>
<tr>
<td></td>
<td>ALL_OR</td>
<td>ALL OR</td>
<td>• The matrix sizes must match or you can use a scalar comparison.</td>
</tr>
<tr>
<td>ANY logical</td>
<td>ANY_AND</td>
<td>ANY AND</td>
<td>• The ANY relational operations produce a scalar result that indicates whether the ([i, j])th element of the first matrix satisfies the comparison with the ([i, j])th element of the second matrix. The scalar result is 0 or 1. If any of the ([i, j]) element comparisons is true, the result is 1. Otherwise, the result is 0.</td>
</tr>
<tr>
<td></td>
<td>ANY_OR</td>
<td>ANY OR</td>
<td>• The matrix sizes must match or you can use a scalar comparison.</td>
</tr>
<tr>
<td>Elementwise</td>
<td>EDIV</td>
<td>elementwise division</td>
<td>• Elementwise operations enable you to apply an operation to a general matrix using a matrix with the same dimensions, a vector whose row dimension matches the row dimension of the general matrix, a vector whose column dimension matches the column dimension of the general matrix, or a 1x1 matrix effectively allowing a scalar operation on each ([i, j]) element.</td>
</tr>
<tr>
<td></td>
<td>EMAX</td>
<td>elementwise maximum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMIN</td>
<td>elementwise minimum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMOD</td>
<td>elementwise remainder of the division of elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMULT</td>
<td>elementwise multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EPOW</td>
<td>elementwise raise to a power</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Elementwise operations produce a result matrix from the element-by-element operations on two argument matrices.</td>
</tr>
</tbody>
</table>

**Considerations When Using a Matrix Package**

- When a matrix is created using a constructor, for example, `m= new matrix(2, 2);` or `declare package matrix m(2, 2);`, the matrix is filled with zeros, not missing values.

- A matrix that contains null or missing values is not very useful. If a matrix does contain null or missing values, some operations on it might be considered anomalous. For example, if you multiply matrices with null or missing values, you receive a run-time error. But if you add or subtract matrices with null or missing
values, you do not receive an error. If you divide a matrix element by zero, you do not get a floating-point exception. The result will be a missing value.

- If you use a loop to input or output matrix data, you want to avoid operations on the matrix while the loop is being processed. The reason is that the matrix is only partially filled until the loop is complete.

- It is not always easy to keep track of what size matrix you have. Make sure that the dimensions for your matrix operations are consistent.

- When a matrix is declared in a thread program, each thread program has its own, individual instance of a matrix. The DS2 matrix package does not support data partitioning between nodes or threads to perform parallel matrix operations. Instead, each thread performs the matrix operations on its own instance of the matrix.

Moving Values from a Matrix into a DS2 Array

The following example shows how to use the TOARRAY method. Using this method is the only way that you can move values from a matrix into a DS2 array for use in a DS2 program.

```plaintext
dcl double a[3, 3];
dcl double c[3, 3];
declare package matrix m;

a := (1, 2, 3, 4, 5, 6, 7, 8, 9);
m=_new_ matrix(a, 3, 3);
m.toarray(c);

do i=1 to 3;
    do j=1 to 3;
        put c[i, j];
    end;
end;
```

Using the SQLSTMT Package

Overview of the SQLSTMT Package

The SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS. The FedSQL statements can create, modify, or delete tables. If the FedSQL statements selects rows from a table, the SQLSTMT package provides methods for interrogating the rows returned in a result set.

When an SQLSTMT instance is created, the FedSQL statement is sent to the FedSQL language processor which, in turn, sends the statement to the DBMS to be prepared and stored in the instance. The instance can then be used to efficiently execute the FedSQL statement multiple times. With the delay of the statement prepare until run time, the FedSQL statement can be built and customized dynamically during execution of the DS2 program.

Hadoop Distribution

If you are using a Hadoop distribution, the use of the SQLSTMT package requires Hive 0.13 or later.

Here is a simple example of using the SQLSTMT to insert values into a Teradata table.

```plaintext
dcl package sqlstmt s('insert into td.testdata (x, y, z) values (?, ?, ?)', [x y z]);
```
do i=1 to 5;
  x=i;
  y=i*1.1;
  z=i*10.01;
  s.execute();
end;
end;

The following rows are inserted into the table:
1 1.1 10.01
2 2.2 20.02
3 3.3 30.03
4 4.4 40.04
5 5.5 50.05

For more examples, see Appendix 4, “DS2 Example Programs,” on page 1305.

**Declaring and Instantiating an SQLSTM T Package**

You use the DECLARE PACKAGE statement to declare the SQLSTM T package. When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance.

There are three ways to construct an instance of an SQLSTM T package.

- Use the DECLARE PACKAGE statement along with its constructor syntax. There are two syntax forms:

  ```sql
  DECLARE PACKAGE SQLSTM T
  variable
  [('sql-txt', [parameter-variable-list])];
  DECLARE PACKAGE SQLSTM T
  variable
  [('sql-txt', [connection-string])];
  ```

- Use the DECLARE PACKAGE statement along with the _NEW_ operator. There are two syntax forms:

  ```sql
  DECLARE PACKAGE SQLSTM T variable;
  variable
  = _NEW_ SQLSTM T('sql-txt', [parameter-variable-list]);
  DECLARE PACKAGE SQLSTM T variable;
  variable
  = _NEW_ SQLSTM T('sql-txt', [connection-string]);
  ```

  **Note:** The DECLARE PACKAGE statement does not construct the SQLSTM T package instance until the _NEW_ operator is executed. The SQL statement prepare does not occur until the _NEW_ operator is executed.

- Use the DECLARE PACKAGE statement without SQL text.

  ```sql
  DECLARE PACKAGE SQLSTM T variable( );
  variable
  = _NEW_ SQLSTM T ( );
  ```

  With the _NEW_ operator, the sql-text can be a string value that is generated from an expression or a string value that is stored in a variable.

If the DECLARE statement includes arguments for construction within its parentheses (and omitting arguments is valid for the SQLSTM T package), then the package instance is allocated. If no parentheses are included, then a variable is created but the package instance is not allocated.
Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

**Specifying FedSQL Statement Parameter Values**
If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values are one of the following:

- the current values of the variables specified in the constructor DECLARE PACKAGE statement or the _NEW_ operator

  For more information, see “DECLARE PACKAGE Statement, SQLSTMT Package” on page 1200 and the “_NEW_ Operator, SQLSTMT Package” on page 1228.

- the current values of the variables specified in the BINDPARAMETERS method

  If you use the BINDPARAMETERS method and execute a FedSQL statement, the values of bound variables are read when the statement is executed and used as the values of the statement’s parameters. If the type of a bound variable differs from the corresponding parameter’s type, the bound variable’s value is converted to the parameter’s type. For more information, see the “BINDPARAMETERS Method” on page 1198.

- the values specified in the SET type methods

  For more information about these methods, see Chapter 36, “DS2 SQLSTMT Package Methods, Operators, and Statements,” on page 1197.

Parameter values must be specified exclusively with bound variables or exclusively with the SET type methods.

*Note:* The rules for identifiers for the FedSQL language apply to variables used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

**Specifying a Connection String**
A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

You can specify a connection string when you declare and instantiate an SQLSTMT package. The connection string parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see the SAS Federation Server: Administrator’s Guide

If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 or DS2 procedure by using the attributes of the currently assigned libref.

**Executing the FedSQL Statement**
The EXECUTE method executes the FedSQL statement and returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists when an SQL UPDATE or DELETE statement does not affect any rows.

When the FedSQL statement is executed, the values of bound variables are read and used as the values of the statement’s parameters.
An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed.

The FedSQL statement executes dynamically at run time. Because the statement is prepared at run time, it can be built and customized dynamically during the execution of the DS2 program.

**Accessing Result Set Data**

The FETCH method returns the next row of data from the result set. A status indicator is returned. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists if the next row to be fetched is located after the end of the result set.

If variables are bound to the result set columns with the BINDRESULTS method or by the FETCH method, then the fetched data for each result set column is placed in the variable bound to that column. If the variables are not bound to the result set columns, the fetched data can be returned by the GET type methods.

*Note:* For character data, you can call the GET type method repeatedly until all of the result set column data is retrieved. For numeric data, you can call the GET type method only once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator. For more information, see the GET type methods in Chapter 36, “DS2 SQLSTMT Package Methods, Operators, and Statements,” on page 1197.

An SQLSTMT instance maintains only one result set. The CLOSERESULTS method automatically releases the result set when the FedSQL statement is executed or deleted.

A run-time error occurs if the FETCH method is called before the FedSQL statement is executed.

**Comparing the SQLSTMT Package and the SQLEXEC Function**

The following table compares the SQLSTMT package and the SQLEXEC function.

<table>
<thead>
<tr>
<th>SQLSTMT Package</th>
<th>SQLEXEC Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicable when FedSQL statements are executed multiple times</td>
<td>applicable when a FedSQL statement is executed only once</td>
</tr>
<tr>
<td>allocates, prepares, executes, and frees a FedSQL statement dynamically at run time</td>
<td>allocates, prepares, executes, and frees a FedSQL statement dynamically at run time</td>
</tr>
<tr>
<td>supports the passing of parameters</td>
<td>does not support the passing of parameters</td>
</tr>
<tr>
<td>produces a result set</td>
<td>does not produce a result set</td>
</tr>
<tr>
<td>supports run-time SELECT query generation</td>
<td>cannot be used with a SELECT statement</td>
</tr>
<tr>
<td>similar to the Java Database Connectivity (JDBC) PreparedStatement class</td>
<td>similar to the SQL EXECUTE IMMEDIATE statement or the JDBC Statement.executeUpdate(String) method</td>
</tr>
</tbody>
</table>
**Using the TZ Package**

**Overview of the TZ Package**
DS2 supports the SQL style date and time conventions that are used in other data sources. When your data source is not a SAS data set, DS2 can process dates and times that have a data type of DATE, TIME, and TIMESTAMP. SAS date, time, and datetime values can be converted to DS2 dates, time, and timestamp values by using the TO_DATE, TO_TIME, and TO_TIMESTAMP functions. However, these functions do not incorporate a time zone.

The TZ package enables you to process local and international time and date values.

**Declaring and Instantiating a TZ Package**
There are two ways to construct an instance of a TZ package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:

  ```
  declare package tz tzpkg;
  tzpkg = _new_ tz();
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

  ```
  declare package tz tzpkg();
  ```

For more information, see the “[DECLARE PACKAGE Statement, TZ Package” on page 1251, and the “_NEW_ Operator, TZ Package” on page 1259.

**Returning Time and Time Zone Information**
You can use the TZ package to return the following values:

- current local time
- current Coordinated Universal Time (UTC) time
- current time zone ID
- current time zone name
- the time zone offset of the time zone from UTC at the specified local time
- the time zone offset of the time zone from UTC at the specified UTC time

Here is an example of how to use the TZ package to get the world clock.

```sas
data _null_
  method init()
  declare package tz tzone();
  dcl double tokyo_time london_time new_york_time utc_time;
  tokyo_time = tzone.getLocalTime('Asia/Tokyo');
  london_time = tzone.getLocalTime('Europe/London');
  new_york_time = tzone.getLocalTime('America/New_York');
  utc_time = tzone.getUTCTime();
  put utc_time = datetime. ;
  put tokyo_time = datetime. ;
  put london_time = datetime. ;
```
The following lines are written to the SAS log:

```
utc_time=18NOV14:13:53:11
tokyo_time=18NOV14:22:53:11
london_time=18NOV14:13:53:11
new_york_time=18NOV14:08:53:11
```

For more information about the methods to perform these actions, see Chapter 37, “DS2 TZ Package Methods, Operators, and Statements,” on page 1251. For more information about time zone ID and names, see “Time Zone IDs and Time Zone Names” in SAS National Language Support (NLS): Reference Guide.

### Returning Time Zone Offset
You can use the TZ package to return the time zone offset from UTC at either the specified local time or at the specified UTC time.

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form +|-hhmm or +|-hh:mm

Here is an example that returns the time zone offset from 'asia/tokyo' and from 'America/New_York'.

```sas
data _null_;  
method init();

declare package tz tzzone('asia/tokyo') ;
dcl double new_york local_time;
dcl char(40) cstr ;

local_time = tzzone.getoffset() ;
put local_time time.;

new_york = tzzone.getoffset('America/New_York') ;
put new_york time.;

end;
enddata ;
run;
```

The following lines are written to the SAS log:

```
9:00:00
-4:00:00
```

### Converting Local or UTC Time
You can use the TZ package to convert local to one of the following time formats:

- ISO8601 with or without a time zone offset
- a TIMESTAMP string with a time zone ID
- UTC time
In addition, you can convert UTC time to local time.

In this example,
```
data _null_;  
  method init();

    declare package tz tzone('asia/tokyo') ;
    dcl double local_time ;
    dcl char(35) local_time_iso local_time_utc local_time_tz;

    local_time = tzone.tolocaltime(15550) ;
    put local_time time.;

    local_time_iso = tzone.toiso8601(15500) ;
    put local_time_iso ;

    local_time_utc = tzone.toutctime(15500) ;
    put local_time_utc time.;

    local_time_tz =tzone.totimestampz(15500);
    put local_time_tz;

end;
enddata ;
run;
```

The following lines are written to the SAS log.

```
13:19:10
1960-01-01T04:18:20.00+09:00
-4:41:40
1960-01-01 04:18:20.00 asia/tokyo
```
Overview of Threaded Processing

Typically, DS2 code runs sequentially. That is, one process runs to completion before the next process begins. It is possible to run more than one process concurrently, using threaded processing. In threaded processing, each concurrently executing section of code is said to be running in a thread. DS2 threading works well both on a machine with multiple cores and within a massively parallel processing (MPP) database.

A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

To enable DS2 code to run in threads:

1. Create the thread by enclosing your DS2 code between THREAD...ENDTHREAD statements.
2. Create one or more instances of the thread in a DS2 program by using a DECLARE THREAD statement.
3. Execute the thread or threads by using a SET FROM statement.

In this example, a very simple thread, T, is created by using the THREAD statement.

```plaintext
thread t;
    dcl int x;
    method init();
```
In this DS2 program, an instance of T is declared, and two threads are executed, using the SET FROM statement in the RUN method. Each of the two threads generates three observations for x for a total of six observations in the output table.

```
data;
dcl thread t t_instance;
method run();
    set from t_instance threads=2;
    put 'x= ' x ;
end;
enddata;
```

When you run the DS2 program, the SAS log might display the following output. Because of how threads are processed, the order of the output could be different.

```
x= 1
x= 2
x= 3
x= 1
x= 2
x= 3
```

**Note:** If one computation thread can keep up with the I/O thread, then that single thread is used for all computation.

**Note:** A single reader feeds all threads. A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

For more information, see the “THREAD Statement” on page 947.

---

**Threaded Processing and the SAS In-Database Code Accelerator**

A DS2 threaded program enables concurrent transformation of data. A DS2 threaded program consists of a thread program of transformations that support concurrency and a data program of transformations that require serialization.

When executed, a DS2 threaded program uses multiple threads. Each thread executes the thread program to concurrently input and transform a portion of the data. If the data program specifies serial transformations, an additional thread is used to combine the data from the multiple threads and perform serialized transformations on the data. The transformed data is then output.

The SAS In-Database Code Accelerator publishes the DS2 thread program to the database and executes the thread program in parallel inside the database. If the data program does not contain any transformations requiring serialization, the SAS In-Database Code Accelerator also publishes and executes the data program in parallel inside the database.
Examples of thread programs include large transpositions, computationally complex programs, scoring models, and BY-group processing.

The SAS In-Database Code Accelerator is available for Greenplum, Hadoop, and Teradata and must be licensed at your site.

For more information about using the SAS In-Database Code Accelerator, see *SAS In-Database Products: User's Guide*. 
Dynamically Executing FedSQL Statements from DS2

You can embed and execute FedSQL statements from within your DS2 programs. You can use FedSQL with DS2 in the following instances:

- You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement.
  
  For more information, see “Using DS2 Packages in Expressions” in SAS FedSQL Language Reference.

- You can use the SQLSTMT package to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.
  
  The SQLSTMT package is intended for use with FedSQL statements that are executed multiple times, statements with parameters, or statements that generate a result set. For more information, see “Using the SQLSTMT Package” on page 172.

- You can also use the SQLEXEC function to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.
  
  The SQLEXEC function is intended for use with FedSQL statements that are executed only one time, do not have parameters, and do not produce a result set. For more information, see the “SQLEXEC Function” on page 786.

- You can load data into a hash instance at run time by using a FedSQL SELECT statement in the DECLARE PACKAGE statement or the DATASET method.
  
  For more information, see “Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time” on page 155 the “DATASET Method” on page 1011 and the “DECLARE PACKAGE Statement, Hash Package” on page 1012.

- You can use the SET statement to input data by using a FedSQL SELECT statement.
  
  For more information, see “SET Statement with Embedded FedSQL” on page 188 and the “SET Statement” on page 937.
You can use these methods to input DS2 data:

- You can generate data using variable or array assignment within packages and methods in your DS2 program.

- You can read an existing table by using the SET statement.
  For more information, see “Reading Data Using the SET Statement” on page 186 and the “SET Statement” on page 937.

- You can read data using a hash package.
  For more information, see “Reading Data Using the Hash Package” on page 189.

- You can retrieve data using an SQLSTMT package by executing a SELECT statement and accessing the result set.
  For more information, see “Reading and Writing Data Using the SQLSTMT Package and the SQLEXEC Function” on page 189.

You can use these methods to output DS2 data:

- You output DS2 data by using the OUTPUT statement. The OUTPUT statement writes rows to a result table.
For more information, see “Writing Data Using the OUTPUT Statement” on page 189.

• You can use the hash package OUTPUT method.
  For more information, see “OUTPUT Method” on page 1053 and “Saving Hash Package Data in a Table” on page 152.

• You can use FedSQL INSERT and UPDATE statements in the SQLEXEC function or the SQLSTMT package.
  “Reading and Writing Data Using the SQLSTMT Package and the SQLEXEC Function” on page 189, “SQLEXEC Function” on page 786, and “Accessing Result Set Data” on page 175.

Reading Data Using the SET Statement

Overview of the SET Statement
The SET statement is flexible and has a variety of uses in DS2 programming. These uses are determined by the options and statements that you use with the SET statement:

• reading rows and columns from existing tables for further processing in a DS2 program
• concatenating and interleaving tables, and performing one-to-one reading of tables
  For more information, see Chapter 20, “Combining Tables,” on page 191.

Each time the SET statement executes, one row is read into the program data vector. SET reads all columns and all rows from the input tables unless you specify otherwise. A SET statement can contain multiple tables; a DS2 program can contain multiple SET statements.

For more information, see “SET Statement” on page 937.

Note: A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

Note: The SET statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

SET Statement Compilation
When the DS2 compiler evaluates a SET statement, it reads the column information for each table-reference. For each column in each table, it creates a global variable in the DS2 program with the same type attributes as those of the column. In this way, the SET statement creates a set of associated column variables. An error occurs if a global variable already exists and the type does not match the type of the table column. This means that if two tables have a column with the same name, those columns' types must be compatible.

SET Statement Execution
In a data program, when a SET statement executes, it reads the first row from the first table. Successive executions continue to read rows until the current table has been
completely read. Then the first row from the next table is read. Each time a row is read, 
the row values are assigned to the corresponding column variables. Column variables 
that are created by the SET statement are retained. SET statement execution ends when 
all rows from all tables have been completely read.

For a thread program, the order of rows entering any given thread is undefined unless 
you use a BY statement. Even then, there is no way in a thread program for a thread to 
assume that all the rows are received before any rows from the other table or tables.

Any column variable that does not appear in the table being read is set to a SAS missing 
value or a null value depending on whether you are in SAS or ANSI mode. Variables 
that are declared by a SET statement are initialized as follows:

- In SAS mode:
  - DOUBLE data types are initialized to the SAS numeric missing value (.)
  - Fixed-width CHAR and NCHAR data types are initialized to the SAS character 
    missing value (a blank filled string).
  - All other data types, including VARCHAR and NVARCHAR are initialized to 
    ANSI null.

- In ANSI mode all types are initialized to ANSI null.

*Note:* For more information, see Chapter 11, “How DS2 Processes Nulls and SAS 
Missing Values,” on page 81.

Here is an example. In this program, the column variable modifiers is not specified in the 
second and fourth rows. The OUTPUT method uses the values of all defined variables 
whether they have been assigned a value after the last OUTPUT statement.

```sas
proc ds2;
data test (overwrite=yes);
dcl char string1 string2 modifiers having informat $char8. format $char8.;
method init();
string1='aBc'; string2='AbC'; modifiers='i'; output;
string1='   abc'; string2='abc'; output;
string1='   abc'; string2='abc'; modifiers='l'; output;
string1=' abc'; string2='   abx'; output;
string1=' abc'; string2='   abx'; modifiers='l'; output;
end;
enddata;
run;

data test_out (overwrite=yes);
dcl double result;
method run();
set test;
result=compare(string1, string2, modifiers);
put 'String 1= ' string1 'String 2= ' string2 'Modifier= ' modifiers
     'Result= ' result;
end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.
To ensure that the second and fourth rows do not have a value specified for the `modifiers` column variable, you must set the variable to a missing or null value.

```
proc ds2;
  data test (overwrite=yes);
    dcl char string1 string2 modifiers having informat $char8. format $char8.;
    method init();
      string1='aBc'; string2='AbC'; modifiers='i'; output;
      string1='   abc'; string2='abc'; modifiers=' '; output;
      string1='   abc'; string2='abc'; modifiers='l'; output;
      string1=' abc'; string2='   abx'; modifiers=' '; output;
      string1=' abc'; string2='   abx'; modifiers='l'; output;
    end;
  enddata;
end;

data test_out (overwrite=yes);
  method run();
    set test;
    result=compare(string1, string2, modifiers);
    put 'String 1= ' string1 'String 2= ' string2 'Modifier= ' modifiers 'Result= ' result;
end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
String 1=  aBc      String 2=  AbC      Modifier=  i        Result=  0
String 1=     abc   String 2=     abc   Modifier=           Result=  -1
String 1=     abc   String 2=     abc   Modifier=  l        Result=  0
String 1=   abc     String 2=     abx   Modifier=           Result=  2
String 1=   abc     String 2=     abx   Modifier=  l        Result=  -3
```

### SET Statement with Embedded FedSQL

A SET statement can use FedSQL code to read a table, as in this example:

```
set {select * from catalog_base.investment};
```

It is possible to interleave table names and embedded FedSQL in a SET statement. In this example, the SET statement reads the same table twice:

```
set {select * from catalog_base.investment} catalog_base.investment;
```

Embedded FedSQL used in a SET statement must be valid FedSQL code, and it must resolve to a set of table rows. Otherwise, an error occurs.

**Note:** There is no guarantee on the order or rows that is surfaced from embedded SQL, regardless of what that embedded SQL contains:

```
set {select * from sql13a order by "X", "Y"};
```
Some environments might preserve the order imposed by the ORDER BY clause, but others do not. DS2 wraps the embedded SQL with additional code. Also, the ORDER BY clause is not honored because DS2 cannot detect and produce FIRST or LAST information. A better way is to write the program with the BY outside of the embedded SQL text:

```sql
set {select * from sql13a}; by "X" "Y";
```

---

**Reading Data Using the Hash Package**

You can use a hash package to read data from a table.

The DECLARE PACKAGE statement, the _NEW_ operator, and the DATASET method accept either of these methods to identify a data source:

- a name that identifies a table
- a valid FedSQL SELECT statement that resolves to a set of table rows

For more information, see “Using the Hash Package” on page 145.

---

**Reading and Writing Data Using the SQLSTM Package and the SQLEXEC Function**

The SQLSTM package and the SQLEXEC function enable DS2 programs to dynamically generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table. With an instance of the SQLSTM package or the SQLEXEC function, the FedSQL statement allocate, prepare, execute, and free occurs at run time.

In addition, the SQLSTM package can interrogate the result set that is produced by the executed FedSQL statement.

For more information, see “Using the SQLSTM Package” on page 172 and the “SQLEXEC Function” on page 786.

**TIP** By using a parameterized FedSQL query in the SQLSTM package, you can achieve what the DATA step SET statement with KEY= does.

---

**Writing Data Using the OUTPUT Statement**

The OUTPUT statement creates an output row, using values for the row that are contained in the global variables when the output statement executes. The OUTPUT statement writes the current row to a table immediately, not at the end of the DS2 program. If no table name is specified in the OUTPUT statement, the row is written to the table or tables that are listed in the DATA statement.

DS2 keeps track of the values in the order in which the compiler encounters them within a DS2 program, whether they are read from existing tables or created in the program.

If you do not supply an OUTPUT statement, DS2 adds one implicitly at the end of the RUN method that writes rows to the table or tables that are being created.
Placing an explicit OUTPUT statement in a DS2 program overrides the automatic output, and adds a row to a table only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write a row to any one table, however, there is no implicit OUTPUT statement at the end of the RUN method. In this situation, a DS2 program writes a row to a table only when an explicit OUTPUT executes. You can use the OUTPUT statement alone or as part of an IF-THEN/ELSE or SELECT statement or in DO loop processing.

*Note:* OUTPUT statements in thread programs cannot contain any table names. Each output row is returned to the data program that started the thread.

---

**Column Order in Output Tables When Using Data Sources Outside SAS**

Some data sources choose their one preferred order for columns in the output table from DS2. For example, on Hive, the BYPARTITION columns are always moved to the end of the table. This is common as various data sources try to optimize their performance.

The order of declaration in a DS2 program might not be used as the order of columns in the data source. For example, if you use `keep K1 - K4;`, you might not get the columns as you expect or you might get an error because `K1` appears after `K4` in the CREATE TABLE statement.

---

**NLS Transcoding Failures**

Transcoding is the process of converting character data from one encoding to another encoding. An NLS transcoding failure can occur during row input or output operations, or during string assignment. By default, this run-time error causes row processing to halt. You can change the default behavior by using one of the following options:

- **SAS Federation Server:** specify the DEFAULTATTR= connection option with the XCODE_WARN=n statement handle option.
- **PROC FEDSQL and PROC DS2:** set the XCODE= option.

Using the options, you can choose to ignore the errors and continue processing of the row.

For more information, see the SAS Federation Server and SAS procedure documentation.
Chapter 20
Combining Tables

Definitions for Combining Data
In the context of D2 processing, combining data has these meanings:

• concatenating
• interleaving
• one-to-one reading
• match-merging

The two statements that are used for combining tables are MERGE and SET.

Combining Tables: Basic Concepts

What You Need to Know Before Combining Information Stored in Multiple Tables

Many applications require input data to be in a specific format before the data can be processed to produce meaningful results. The data typically comes from multiple sources and might be in different formats. Therefore, you often, if not always, have to take
intermediate steps to logically relate and process data before you can analyze it or create reports from it.

Application requirements vary, but there are common factors for all applications that access, combine, and process data. Once you have determined what you want the output to look like, you must perform the following tasks:

- Determine how the input data is related.
- Ensure that the data is properly sorted, if necessary.
- Select the appropriate access method to process the input data.
- Select the appropriate tools to complete the task.

### The Four Ways That Data Can Be Related

#### Data Relationship Categories

Relationships among multiple sources of input data exist when each of the sources contains common data, either at the physical or logical level. For example, employee data and department data could be related through an employee ID column that shares common values. Another table could contain numeric sequence numbers whose partial values logically relate it to a separate table by row number.

You must be able to identify the existing relationships in your data. This knowledge is crucial for understanding how to process input data in order to produce desired results. All related data falls into one of these four categories, characterized by how rows relate among the tables:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

To obtain the results that you want, you should understand how each of these methods combines rows, how each method treats duplicate values of common columns, and how each method treats missing values or nonmatched values of common columns. Some of the methods also require that you preprocess your tables by sorting them. See the description of each method in “Methods for Combining Tables” on page 194.

#### One-to-One Relationship

In a one-to-one relationship, typically a single row in one table is related to a single row from another based on the values of one or more selected columns. A one-to-one relationship implies that each value of the selected column occurs no more than once in each table. When you work with multiple selected columns, this relationship implies that each combination of values occurs no more than once in each table.

In the following example, rows in tables Salary and Taxes are related by common values for EmployeeNumber.
One-to-One Relationship

<table>
<thead>
<tr>
<th>EmployeeNumber</th>
<th>Salary</th>
<th>EmployeeNumber</th>
<th>TaxBracket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>55000</td>
<td>1111</td>
<td>0.18</td>
</tr>
<tr>
<td>3333</td>
<td>72000</td>
<td>1234</td>
<td>0.28</td>
</tr>
<tr>
<td>4876</td>
<td>32000</td>
<td>3333</td>
<td>0.32</td>
</tr>
<tr>
<td>5489</td>
<td>17000</td>
<td>4222</td>
<td>0.18</td>
</tr>
</tbody>
</table>

One-to-Many and Many-to-One Relationships

A one-to-many or many-to-one relationship between input tables implies that one table has at most one row with a specific value of the selected column, but the other input table can have more than one occurrence of each value. When you work with multiple selected columns, this relationship implies that each combination of values occurs no more than once in one table. However, the combination can occur more than once in the other table. The order in which the input tables are processed determines whether the relationship is one-to-many or many-to-one.

In the following example, rows in tables One and Two are related by common values for column A. Values of A are unique in table One but not in table Two.

One-to-Many Relationship

<table>
<thead>
<tr>
<th>ONE</th>
<th>TWO</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
</tr>
</tbody>
</table>

In the following example, rows in tables One, Two, and Three are related by common values for column ID. Values of ID are unique in tables One and Three but not in Two. For values 2 and 3 of ID, a one-to-many relationship exists between rows in tables One and Two, and a many-to-one relationship exists between rows in tables Two and Three.
Many-to-Many Relationships

The many-to-many category implies that multiple rows from each input table can be related based on values of one or more common columns.

In the following example, rows in tables BreakDown and Maintenance are related by common values for column Vehicle. Values of Vehicle are not unique in either table. A many-to-many relationship exists between rows in these tables for values AAA and CCC of Vehicle.

Overview of Methods for Combining Tables

Methods for Combining Tables

You can use these methods to combine tables:

- concatenating
- interleaving
- one-to-one reading
- match-merging
**Concatenating**

The following figure shows the results of concatenating two tables. Concatenating the tables appends the rows from one table to another table. The data program reads Data1 sequentially until all rows have been processed, and then reads Data2. Table Combined contains the results of the concatenation. Note that the tables are processed in the order in which they are listed in the SET statement.

```plaintext
data concatenate;
  method run();
    set data1 data2;
  end;
enddata;
run;
```

*Note:* For a thread program, the order of rows entering any given thread is undefined unless you use a BY statement. Even then, there is no way in a thread program for a thread to assume that all the rows are received before any rows from the other table or tables.

**Figure 20.5 Concatenating Two Tables**

![Concatenating Two Tables](image)

**Interleaving**

The following data program interleaves two tables. Interleaving intersperses rows from two or more tables, based on one or more common columns. Table Combined shows the results.

```plaintext
data interleave;
  method run();
    set data1 data2;
    by year;
  end;
enddata;
run;
```
One-to-One Reading

The following figure shows the results of one-to-one reading. One-to-one reading combines rows from two or more tables by creating rows that contain all of the columns from each contributing table. Rows are combined based on their relative position in each table, that is, the first row in one table with the first in the other, and so on. The data program stops after it has read the last row from the smallest table. Table Combined shows the results.

```plaintext
data one2one;
  method run();
  set data1;
  set data2;
  end;
enddata;
run;
```

The following data program results in one-to-one reading of two tables.

Match-Merging

The following figure shows the results of match-merging. Match-merging combines rows from two or more tables into a single row in a new table based on the values of one or more common columns. The following data program results in one-to-one reading of two tables. Table Combined shows the results.

```plaintext
data one2one;
  method run();
  merge data1 data2;
```
by year;
end;
enddata;
run;

Figure 20.8  Match-Merging Two Tables

How to Prepare Your Tables

Guidelines to Prepare Your Tables
Before combining tables, follow these guidelines to produce the results that you want:

• Know the structure and the contents of the tables.

• Look at sources of common problems.

• Ensure that rows are in the correct order, or that they can be retrieved in the correct order (for example, by presorting them).

• Test your program.

Knowing the Structure and Contents of the Tables
To help determine how your data is related, look at the structure of the tables. To see the table structure, execute the DATASETS procedure, the CONTENTS procedure, or access the SAS Explorer window in your windowing environment to display the descriptor information. Descriptor information includes the number of rows in each table, the name and attributes of each column, and an alphabetic list of extended attributes (including table and column extended attributes). To print a sample of the rows, use the PRINT procedure or the REPORT procedure.

You can also use functions such as VTYPE and VLENGTH to show specific descriptor information. For more information, see Chapter 23, “DS2 Functions,” on page 335.

Looking at Sources of Common Problems
If your program does not execute correctly, review your input data for the following errors:

• columns that have the same name but that represent different data

  DS2 includes only one column of a given name in the new table. If you are merging two tables that have columns with the same names but different data, the values from the last table that was read are written over the values from other tables.

  To correct the error, you can rename columns before you combine the tables by using the RENAME= table option in the SET or MERGE statement. Or you can use the DATASETS procedure.
• common columns with the same data but different attributes

The way DS2 handles these differences depends on which attributes are different:

• type attribute

If the type attributes are incompatible, DS2 stops processing the data program and issues an error message stating that the columns are incompatible.

To correct this error, you must use a separate DS2 data program to change the types as necessary. The DS2 statements that you use depend on the nature of the column. The following table contains the result attribute when combining columns.

Table 20.1 Type Promotion

<table>
<thead>
<tr>
<th>Left-hand side</th>
<th>Right-hand side</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>INTEGER</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>SMALLINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>TINYINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEGER</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TINYINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>DECIMAL</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>BIGINT</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DECIMAL (X,Y)</td>
<td>DECIMAL (X,Z)</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>TIME*</td>
<td>TIME*</td>
<td>TIME*</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>ERROR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
</tbody>
</table>
### Combining Tables: Basic Concepts

<table>
<thead>
<tr>
<th>Left-hand side</th>
<th>Right-hand side</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARBINARY*</td>
<td>VARBINARY</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>BINARY</td>
<td>BINARY</td>
<td>BINARY</td>
</tr>
<tr>
<td>BINARY</td>
<td>VARBINARY</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>CHAR(N)</td>
<td>CHAR(N)</td>
<td>CHAR(N)</td>
</tr>
<tr>
<td>CHAR(N)**</td>
<td>CHAR(N)**</td>
<td>VARCHAR(N)**</td>
</tr>
<tr>
<td>VARCHAR(N)</td>
<td>VARCHAR(N)</td>
<td>VARCHAR(N)</td>
</tr>
<tr>
<td>VARCHAR(N)</td>
<td>VARCHAR(M)</td>
<td>VARCHAR(MAX(N, M))</td>
</tr>
<tr>
<td>CHAR(N)</td>
<td>CHAR(M)</td>
<td>VARCHAR(MAX(N, M))</td>
</tr>
<tr>
<td>VARCHAR(N)</td>
<td>CHAR(M)</td>
<td>VARCHAR(MAX(N, M))</td>
</tr>
<tr>
<td>NCHAR(M)</td>
<td>VARCHAR(M)</td>
<td>VARCHAR(M)</td>
</tr>
<tr>
<td>NCHAR(M)</td>
<td>NCHAR(M)</td>
<td>NCHAR(M)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>CHARACTER</td>
<td>ERROR</td>
</tr>
<tr>
<td>DATE</td>
<td>DOUBLE</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

* No Hive support
** For DB2, Impala, ODBC, and Oracle

**Note:** A best practice is to declare every variable in a thread program. By doing so, you can avoid type mismatches among data sources.

**• length attribute**

If the length attribute is different, DS2 takes the length from the table that contains the column with the maximum length. In the following example, all tables that are listed in the MERGE statement contain the column Mileage. In Quarter1, the length of the column Mileage is four bytes; in Quarter2, it is eight bytes and in Quarter3 and Quarter4, it is six bytes. In the output table Yearly, the length of the column Mileage is eight bytes, which is the length derived from Quarter2.

```plaintext
data yearly;
  dcl char(4) quarter1 char(8) quarter2 char(6) quarter3 quarter4;
  method run();
    merge quarter1 quarter2 quarter3 quarter4;
    by Account;
  end;
enddata;
run;
```

**• label, format, and informat attributes**

If any of these attributes are different, DS2 takes the attribute from the first table that contains the column with that attribute. However, any label, format, or informat that you explicitly specify overrides a default. If all tables contain explicitly specified attributes, the one specified in the first table overrides the others.
You can also use functions such as VLABEL to show specific descriptor information. For more information, see Chapter 23, “DS2 Functions,” on page 335.

Testing Your Program
As a final step in preparing your tables, you should test your program. Create small temporary tables that contain a sample of rows that test all of your program's logic. If your logic is faulty and you get unexpected output, you can debug your program.

Combining DS2 Tables: Methods

Concatenating

Definition
Concatenating tables is the combining of two or more tables, one after the other, into a single table. The number of rows in the new table is the sum of the number of rows in the original tables. The order of row is sequential. All rows from the first table are followed by all rows from the second table, and so on.

In the simplest case, all input tables contain the same columns. If the input tables contain different columns, rows from one table have missing values for columns defined only in other tables. In either case, the columns in the new table are the same as the columns in the old tables.

Syntax
Use this form of the SET statement to concatenate tables:

SET table(s);

Arguments

| table(s) | specifies any valid table name. |

For more information, see “SET Statement” on page 937.

DS2 Processing during Concatenation

Compilation phase
DS2 reads the descriptor information of each table that is named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.

Execution — Step 1
DS2 reads the first row from the first table into the program data vector. It processes the first row and executes other statements in the data program. It then writes the contents of the program data vector to the new table.

The SET statement does not reset the values in the program data vector to missing, except for columns whose value is calculated or assigned during the data program. Columns that are created by the data program are set to missing at the beginning of each iteration of the data program unless they are retained. Variables that are read from a table are not.
Execution — Step 2

In the data program, DS2 continues to read one row at a time from the first table until it finds an end-of-file indicator. The values of the columns in the program data vector are then set to missing, and the data program begins reading rows from the second table, and so on, until it reads all rows from all tables. For a thread program, the order of rows entering any given thread is undefined unless you use a BY statement. Even then, there is no way in a thread program for a thread to assume that all the rows are received before any rows from the other table or tables.

Example 1: Concatenation Using the Data Program

In this example, each table contains the columns Common and Number, and the rows are arranged in the order of the values of Common. Generally, you concatenate tables that have the same columns. In this case, each table also contains a unique column to show the effects of combining tables more clearly. The following program uses a SET statement to concatenate the tables and then prints the results:

```sas
/*  set concatenates */
data concatenate (overwrite=yes);
method run();
  set animal plant;
end;
enddata;
run;
proc print data=concatenate;
run;
quit;
```

```sas
data animal(overwrite=yes);
dcl varchar(10) common animal number;
method init();
  common='a'; animal='Ant'; number='5'; output;
  common='b'; animal='Bird'; number=''; output;
  common='c'; animal='Cat'; number='17'; output;
  common='d'; animal='Dog'; number='9'; output;
  common='e'; animal='Eagle'; number=''; output;
  common='f'; animal='Frog'; number='76'; output;
end;
enddata;
run;
```

```sas
data plant(overwrite=yes);
dcl varchar(10) common plant number;
method init();
  common='g'; plant='Grape'; number='69'; output;
  common='h'; plant='Hazelnut'; number='55'; output;
  common='i'; plant='Indigo'; number=''; output;
  common='j'; plant='Jicama'; number='14'; output;
  common='k'; plant='Kale'; number='5'; output;
  common='l'; plant='Lentil'; number='77'; output;
end;
enddata;
run;
```
Output 20.1 Concatenated Table (SET Statement)

![Animal and Plant tables](image)

The resulting table, Concatenate, has 12 rows, which is the sum of the rows from the combined tables. The program data vector contains all columns from all tables. The values of columns found in one table but not in another are set to missing.

**Example 2: Concatenation Using SQL**

You can also use the SQL language to concatenate tables. In this example, SQL reads each row in both tables and creates a new table named Combined. The following shows the YEAR1 and YEAR2 input tables:

```sql
proc sql;
create table combined as
  select * from animal
  union all
  select * from plant;
select * from combined;
quit;
```

```sql
proc print data=combined;
run;
quit;
```

The output is exactly the same as the output when using the SET statement.

---

**Interleaving**

**Definition**

Interleaving uses a SET statement and a BY statement to combine multiple tables into one new table. This is also known as BY-group processing. BY-group processing is a method of combining rows from one or more tables that are grouped or ordered by
values of one or more common columns. When a BY statement is specified immediately after a SET statement, the SET statement interleaves the rows of the input tables in sorted order. The sort order or sort key is specified by the column names in the BY statement. The number of rows in the new table is the sum of the number of rows from the original tables.

The keyword DESCENDING can be used before the name of the column in the BY statement in order to sort that column in descending instead of ascending order.

**Syntax**

Use this form of the SET statement to interleave tables when you use a BY variable:

```
SET table(s);
BY [DESCENDING]column [...[DESCENDING] column];
```

**Arguments**

- **table**
  - specifies a table name.

- **DESCENDING**
  - specifies that the tables are sorted in descending order by the column that is specified. DESCENDING means largest to smallest for numeric columns, or reverse alphabetical for character columns.

- **column**
  - specifies each column by which the table is sorted. These columns are referred to as BY variables for the current data program.

**Sort Requirements**

When a BY statement is used, internally DS2 requests the rows in sorted order. If the rows are already sorted, "re-sorting" of the data might be necessary.

*Note:* When the SAS In-Database Code Accelerator executes the DS2 programs, BY groups might not necessarily be in sorted order depending on whether the host environment supports sorting as opposed to simply hashing. The SAS In-Database Code Accelerator chooses the most efficient technique for clustering like values together without causing local re-sorting of potentially large data volumes.

**DS2 Processing during Interleaving**

Compilation phase

- DS2 reads the descriptor information of each table that is named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.

- In the DS2 program, SAS identifies the beginning and end of each BY group by creating two temporary variables for each BY column: FIRST.variable and LAST.variable. Their values indicate whether an observation has the following characteristics:
  - the first one in a BY group
  - the last one in a BY group
  - neither the first nor the last one in a BY group
  - both first and last, as is the case when there is only one observation in a BY group.

When a row is the first in a BY group, SAS sets the value of FIRST.variable to 1 for the column whose value changed, as well as for all of the columns that follow
in the BY statement. For all other rows in the BY group, the value of FIRST.variable is 0. Likewise, if the row is the last in a BY group, SAS sets the value of LAST.variable to 1 for the column whose value changes on the next row, as well as for all of the columns that follow in the BY statement. For all other rows in the BY group, the value of LAST.variable is 0. For the last row in a table, the values of all LAST.variable variables are set to 1. These temporary variables are available for DS2 programming but are not added to the output table.

You can take actions conditionally, based on whether you are processing the first or the last row in a BY group.

Note: See “Example 3: Interleaving Tables” on page 940 for an example that illustrates BY-group processing.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The BY statement requires a utility file and the SAS UTILLOC= system option allocates space for that utility file. For more information, see the SAS UTILLOC= system option in SAS System Options: Reference.

Execution — Step 1
DS2 compares the first row from each table that is named in the SET statement to determine which BY group should appear first in the new table. It reads all rows from the first BY group from the selected table. If this BY group appears in more than one table, it reads from the tables in the order in which they appear in the SET statement. The values of the columns in the program data vector are set to missing each time DS2 starts to read a new table and when the BY group changes.

Execution — Step 2
DS2 compares the next rows from each table to determine the next BY group and then starts reading rows from the selected table in the SET statement that contains rows for this BY group. DS2 continues until it has read all rows from all tables.

Example 1: Interleaving in the Simplest Case
In this example, each table contains the BY variable Common, and the rows are arranged in order of the values of the BY variable. The following example creates the Animal and the Plant input tables.

```sas
data animal(overwrite=yes);
  dcl varchar(10) common animal number;
  method init();
    common='a'; animal='Ant'; number='5'; output;
    common='b'; animal='Bird'; number=''; output;
    common='c'; animal='Cat'; number='17'; output;
    common='d'; animal='Dog'; number='9'; output;
    common='e'; animal='Eagle'; number=''; output;
    common='f'; animal='Frog'; number='76'; output;
  end;
enddata;
run;

data plant(overwrite=yes);
  dcl varchar(10) common plant number;
  method init();
    common='a'; plant=''; number='69'; output;
    common='b'; plant='Bamboo'; number='55'; output;
    common='c'; plant='Cabbage'; number=''; output;
end;
enddata;
run;
```
The following program uses SET and BY statements to interleave the tables, and prints the results:

```plaintext
/*  set with by interleaves */
data interleave (overwrite=yes);
  method run();
    set animal plant; by common;
  end;
enddata;
run;

proc print data=interleave;
run;
quit;
```

**Output 20.2  Interleaved Table (SET Statement)**

The resulting table Interleave has 12 rows, which is the sum of the rows from the combined tables. The new table contains all columns from both tables. The value of columns found in one table but not in the other are set to missing, and the rows are arranged by the values of the BY variable.

**Example 2: Interleaving with Duplicate Values of the BY Variable**

If the tables contain duplicate values of the BY variables, the rows are written to the new table in the order in which they occur in the original tables. This example contains duplicate values of the BY variable Common. The following program creates the Animal and Plant input tables:

```plaintext
data animal(overwrite=yes);
```
The following program uses SET and BY statements to interleave the tables, and prints the results:

```plaintext
/* set with by interleaves */
data interleave (overwrite=yes);
   method run();
      set animal plant; by common;
   end;
enddata;
run;

proc print data=interleave;
run;
quit;
```
The number of rows in the new table is the sum of the rows in all the tables. The rows are written to the new table in the order in which they occur in the original tables.

**Example 3: Interleaving with Different BY Values in Each Table**

The tables Animal and Plant both contain values that are present in one table but not in the other. The following program creates the Animal and the Plant input tables:

```sas
data animal(overwrite=yes);
  dcl varchar(10) common animal number;
  method init();
    common='a'; animal='Ant'; number='5'; output;
    common='c'; animal='Bird'; number=''; output;
    common='d'; animal='Cat'; number='17'; output;
    common='e'; animal='Dog'; number='9'; output;
  end;
enddata;
run;

data plant(overwrite=yes);
  dcl varchar(10) common plant number;
  method init();
    common='a'; plant='Grape'; number='69'; output;
    common='b'; plant='Hazelnut'; number='55'; output;
    common='c'; plant='Indigo'; number=''; output;
    common='d'; plant='Jicama'; number='14'; output;
    common='e'; plant='Kale'; number='5'; output;
    common='f'; plant='Lentil'; number='77'; output;
  end;
enddata;
run;
```
This program uses SET and BY statements to interleave these tables, and prints the results:

```sas
data interleave (overwrite=yes);
  method run();
    set animal plant; by common;
  end;
enddata;
run;

proc print data=interleave;
run;
quit;
```

**Output 20.4  Interleaved table with Different BY Variables (SET Statement)**

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>number</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td></td>
<td>69</td>
<td>Grape</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td></td>
<td>55</td>
<td>Hazelnut</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Bird</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td></td>
<td></td>
<td>Indigo</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>Cat</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td></td>
<td>14</td>
<td>Jicama</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>Dog</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td></td>
<td>5</td>
<td>Kale</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td></td>
<td>77</td>
<td>Lentil</td>
</tr>
</tbody>
</table>

The resulting table has ten rows arranged by the values of the BY variable.

**Comments and Comparisons**

- In other languages, the term merge is often used to mean interleave. DS2 reserves the term merge for the operation in which rows from two or more tables are combined into one row. The rows in interleaved tables are not combined; they are copied from the original tables in the order of the values of the BY variable.
- If one table has multiple rows with the same BY value, the DATA step preserves the order of those rows in the result.
- To use the data program, the input tables must be appropriately sorted. SQL does not require the input tables to be in order.

**One-to-One Reading**

**Definition**

One-to-one reading combines rows from two or more tables into one row by using two or more SET statements to read rows independently from each table. This process is also
called one-to-one matching. The new table contains all the columns from all the input tables. The number of rows in the new table is the number of rows in the smallest original table. If the tables contain common columns, the values that are read in from the last table replace the values that were read in from earlier tables.

**Syntax**

Use this form of the SET statement for one-to-one reading:

```plaintext
SET table-1;
SET table-2;
```

**Arguments**

- `table-1` specifies a table name. `table-1` is the first table that the data program reads.
- `table-2` specifies a table name. `table-2` is the second table that the data program reads.

**CAUTION:**

Use care when you combine tables with multiple SET statements. Using multiple SET statements to combine rows can produce undesirable results. Test your program on representative samples of the tables before using this method to combine them.

For more information, see “SET Statement” on page 937.

**DS2 Processing during a One-to-One Reading**

**Compilation phase**

DS2 reads the descriptor information of each table named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.

**Execution — Step 1**

When DS2 executes the first SET statement, DS2 reads the first row from the first table into the program data vector. The second SET statement reads the first row from the second table into the program data vector. If both tables contain the same columns, the values from the second table replace the values from the first table, even if the value is missing. After reading the first row from the last table and executing any other statements in the data program, DS2 writes the contents of the program data vector to the new table. The SET statement does not reset the values in the program data vector to missing, except for those columns that were created or assigned values during the data program.

**Execution — Step 2**

DS2 continues reading from one table and then the other until it detects an end-of-file indicator in one of the tables. DS2 stops processing with the last row of the shortest table and does not read the remaining rows from the longer table.

**Example 1: One-to-One Reading: Processing an Equal Number of Rows**

The tables Animal and Plant both contain the column Common, and are arranged by the values of that column. The following program creates the Animal and the Plant input tables:

```plaintext
data animal(overwrite=yes);
dcl varchar(10) common animal number;
method init();
    common='a'; animal='Ant'; output;
```
The following program uses two SET statements to combine rows from Animal and Plant, and prints the results:

```plaintext
data one2one (overwrite=yes);
  method run();
    set animal;
    set plant;
  end;
enddata;
run;
```

```
proc print data=one2one;
run;
quit;
```

**Output 20.5**  One-to-One Reading with an Equal Number of Rows (SET Statement)

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Grape</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Hazelnut</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Indigo</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Jicama</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Kale</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>Frog</td>
<td>Lentil</td>
</tr>
</tbody>
</table>
Each row in the new table contains all the columns from all the tables. Note, however, that the Common column value in row 6 contains a “g.” The value of Common in row 6 of the Animal table was overwritten by the value in Plant, which was the table that DS2 read last.

**One-to-One Reading with an Uneven Number of Rows**

The tables Animal and Plant both contain the column Common, and are arranged by the values of that column. The tables have different number of rows. The following program creates the Animal and the Plant input tables:

```plaintext
data animal(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='a'; animal='Bird'; output;
      common='b'; animal='Cat'; output;
      common='c'; animal='Dog'; output;
      common='d'; animal='Eagle'; output;
      common='e'; animal='Frog'; output;
   end;
enddata;
run;

data plant(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Grape'; output;
      common='b'; plant='Hazelnut'; output;
      common='c'; plant='Indigo'; output;
      common='d'; plant='Jicama'; output;
   end;
enddata;
run;

The following program uses two SET statements to combine rows from Animal and Plant, and prints the results:

data one2onerowsnotequal (overwrite=yes);
   method run();
      set animal;
      set plant;
   end;
enddata;
run;
quit;
proc print data=one2onerowsnotequal;
run;
```
One-to-One Reading with an Uneven Number of Rows (SET Statement)

The result table contains only four rows because DS2 stops processing with the last row from the shortest table.

Comments and Comparisons

• Using multiple SET statements with other DS2 statements makes the following applications possible:
  • merging one row with many
  • conditionally merging rows
  • reading from the same table twice

Match-Merging

Definition

Match-merging combines rows from two or more tables into a single row in a new table according to the values of a common column. The number of rows in the new table is the sum of the largest number of rows in each BY group in all tables. To perform a match-merge, use the MERGE statement with the required BY statement. When you perform a match-merge, all tables are sorted by the columns that you specify in the BY statement.

Syntax

Use this form of the MERGE statement to match-merge tables:

MERGE table(s);
BY column(s);

Arguments

table
  names at least two existing tables from which rows are read.

column
  names each column by which the table is sorted. These columns are referred to as BY variables.

For more information, see the “MERGE Statement” on page 906 and the “BY Statement” on page 868.
**DS2 Processing during Match-Merging**

Compilation phase  
DS2 reads the descriptor information of each table that is named in the MERGE statement and then creates a program data vector that contains all the rows from all tables as well as rows created by the data program.

Execution – Step 1  
DS2 looks at the first BY group in each table that is named in the MERGE statement to determine which BY group should appear first in the new table. The data program reads into the program data vector the first row in that BY group from each table, reading the tables in the order in which they appear in the MERGE statement. If a table does not have rows in that BY group, the program data vector contains missing values for the rows that are unique to that table.

Execution – Step 2  
Each row in any of the input tables is used exactly once in the output table. Columns that are unique to a table are filled with missing or null values if that table is exhausted while producing a BY group.

Execution – Step 3  
DS2 repeats these steps until it reads all rows from all BY groups in all tables.

**CAUTION:**  
BY variables in a DS2 merge that have a DECIMAL or NUMERIC data type are converted to a DOUBLE data type. If matching DECIMAL columns are not BY variables, the DECIMAL columns remain as a DECIMAL data type.

**CAUTION:**  
If there is a type, scale, or precision mismatch between columns with a DECIMAL or NUMERIC data type between tables, the column is converted to a DOUBLE data type.

**Example 1: Merging Rows**  
The tables Animal and Plant each contain the BY variable Common, and the rows are arranged in order of the values of the BY variable. The following program creates the Animal and the Plant input tables:

```plaintext
data animal(overwrite=yes);  
dcl varchar(10) common animal;  
method init();  
  common='a'; animal='Ant'; output;  
  common='b'; animal='Bird'; output;  
  common='c'; animal='Cat'; output;  
  common='d'; animal='Dog'; output;  
  common='e'; animal='Eagle'; output;  
  common='f'; animal='Frog'; output;  
end;  
enddata;  
run;
```

```plaintext
data plant(overwrite=yes);  
dcl varchar(10) common plant;  
method init();  
  common='a'; plant='Grape'; output;  
  common='b'; plant='Hazelnut'; output;  
  common='c'; plant='Indigo'; output;  
  common='d'; plant='Jicama'; output;  
```

Combining DS2 Tables: Methods  
213
The following program merges the tables according to the values of the \textit{BY} variable \textit{Common}, and prints the results:

```plaintext
data mmerge (overwrite=yes);
  method run();
   merge animal plant;
     by common;
  end;
enddata;
run;
```

```
proc print data=mmerge;
run;
quit;
```

\textbf{Output 20.7} \textit{Simple Match Merge (MERGE Statement)}

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Grape</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Hazelnut</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Indigo</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Jicama</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Kale</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Lentil</td>
</tr>
</tbody>
</table>

Each row in the new table contains all the columns from all the tables.

\textbf{Example 2: Match-Merge with Duplicate Values of the \textit{BY} Variable}

In the following example, the tables Animal and Plant contain duplicate values of the \textit{BY} variable \textit{Common}. The following program creates the Animal and the Plant input tables:

```plaintext
data animal(overwrite=yes);
  dcl varchar(10) common animal;
  method init();
    common='a'; animal='Ant'; output;
    common='a'; animal='Ape';  output;
    common='b'; animal='Bird';  output;
    common='c'; animal='Cat';  output;
    common='d'; animal='Dog';  output;
    common='e'; animal='Eagle';  output;
  end;
enddata;
run;
```
data plant(overwrite=yes);
  dcl varchar(10) common plant;
  method init();
    common='a'; plant='Apple'; output;
    common='b'; plant='Banana'; output;
    common='c'; plant='Coconut'; output;
    common='c'; plant='Celery'; output;
    common='d'; plant='Dewberry'; output;
    common='e'; plant='Eggplant'; output;
  end;
enddata;
run;

The following program produces the merged table MATCH1, and prints the results:

data mmdiffby (overwrite=yes);
  method run();
    merge animal plant;
    by common;
  end;
enddata;
run;
quit;

proc print data=mmdiffby;
run;
quit;

Output 20.8  Match Merge with Duplicate BY Variables (MERGE Statement)

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ape</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ant</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>Celery</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
</tbody>
</table>

In row 2 of the output, the value of the column Plant is not retained. Match-merging also
did not duplicate values in Animal for row 5.

Note: The MERGE statement does not produce a Cartesian product on a many-to-many
match-merge. Instead, it performs a one-to-one merge while there are rows in the BY
group in at least one table. When all rows in the BY group have been read from one
table and there are still more rows in another table, DS2 fills the columns with
missing or null values.
Example 3: Match-Merge with Non-matched Rows

When DS2 performs a match-merge with nonmatched rows in the input tables, DS2 retains the values of all columns in the program data vector even if the value is missing. The tables Animal and Plant do not contain all values of the BY variable Common. The following program creates the Animal and the Plant input tables:

```plaintext
data animal(overwrite=yes);
dcl varchar(10) common animal;
method init();
   common='a'; animal='Ant'; output;
   common='c'; animal='Cat'; output;
   common='d'; animal='Dog'; output;
   common='e'; animal='Eagle'; output;
end;
enddata;
run;

data plant(overwrite=yes);
dcl varchar(10) common plant;
method init();
   common='a'; plant='Apple'; output;
   common='b'; plant='Banana'; output;
   common='c'; plant='Coconut'; output;
   common='e'; plant='Eggplant'; output;
   common='f'; plant='Fig'; output;
end;
enddata;
run;
```

The following program produces the merged table Mmnomrow, and prints the results:

```plaintext
data mmmomrow (overwrite=yes);
method run();
   merge animal plant;
   by common;
end;
enddata;
run;
quit;
```

```plaintext
proc print data=mmnomrow;
run;
quit;
```
Output 20.9  Match Merge with Non-Matched Rows (MERGE Statement)

As the output shows, all values of the column Common are represented in the new table, including missing values for the columns that are in one table but not in the other.
Reserved Words in the DS2 Language

The following words are reserved as DS2 language keywords and cannot be used as variable names or in any other way that differs from their intended use.

*Note:* You can use a reserved word as a variable name if the word is enclosed in double quotation marks. For more information and an example, see “Delimited Identifiers” on page 78.

**Table 21.1 DS2 Reserved Words**

<table>
<thead>
<tr>
<th>Special Characters</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>___KPLIST <em><strong>ALL___NULL___RC___ROWSET___TEMPORARY___THREADID</strong></em></td>
<td>ABORT</td>
<td>BIGINT</td>
<td>CALL</td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>BINARY</td>
<td>CATALOG</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>AS</td>
<td>BY</td>
<td>CHAR</td>
<td>DCL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CHARACTER</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>COMMIT</td>
<td>DECLARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONTINUE</td>
<td>DELETE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DESCENDING</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DS2_OPTIONS</td>
</tr>
<tr>
<td>E</td>
<td>ELIF</td>
<td>ELSE</td>
<td>ENCRYPT</td>
<td>END</td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>-----</td>
</tr>
<tr>
<td>F</td>
<td>FILE</td>
<td>FILENAME</td>
<td>FLOAT</td>
<td>FORMAT</td>
</tr>
<tr>
<td>G</td>
<td>HAVING</td>
<td>I</td>
<td>IDENTITY</td>
<td>IF</td>
</tr>
<tr>
<td>H</td>
<td>KEEP</td>
<td>K</td>
<td>LABEL</td>
<td>LEAVE</td>
</tr>
<tr>
<td>L</td>
<td>MERGE</td>
<td>M</td>
<td>NATIONAL</td>
<td>NCHAR</td>
</tr>
<tr>
<td>M</td>
<td>O</td>
<td>ODSD</td>
<td>OR</td>
<td>OTHER</td>
</tr>
<tr>
<td>N</td>
<td>PACKAGE</td>
<td>P</td>
<td>REAL</td>
<td>REMOVE</td>
</tr>
<tr>
<td>O</td>
<td>REAL</td>
<td>P</td>
<td>SELECT</td>
<td>SET</td>
</tr>
<tr>
<td>R</td>
<td>SELECT</td>
<td>S</td>
<td>TABLE</td>
<td>THEN</td>
</tr>
<tr>
<td>S</td>
<td>TABLE</td>
<td>T</td>
<td>UNTIL</td>
<td>UPDATE</td>
</tr>
<tr>
<td>T</td>
<td>UNTIL</td>
<td>U</td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARARRAY</td>
<td>WHEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARBINARY</td>
<td>WHERE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td>WHILE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARLIST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARYING</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part 4

DS2 Language Reference

Chapter 22
DS2 Formats ................................................................. 225

Chapter 23
DS2 Functions .............................................................. 335

Chapter 24
DS2 Informats ............................................................... 853

Chapter 25
DS2 Operators .............................................................. 857

Chapter 26
DS2 Statements ............................................................ 861

Chapter 27
DS2 System Methods ...................................................... 957

Chapter 28
DS2 System Options ...................................................... 965

Chapter 29
DS2 Table Options ........................................................ 969

Chapter 30
DS2 FCMP Package Methods, Operators, and Statements ...... 997

Chapter 31
DS2 Hash and Hash Iterator Package Attributes,
Methods, Operators, and Statements .............................. 1003

Chapter 32
DS2 HTTP Package Methods, Operators, and Statements ...... 1075

Chapter 33
DS2 JSON Package Methods, Operators, and Statements ...... 1099

Chapter 34
DS2 Logger Package Methods, Operators, and Statements ...... 1125
Chapter 22
DS2 Formats

Overview of Formats .......................................................... 227
General Format Syntax ....................................................... 227
Using Formats in DS2 ......................................................... 228
How to Specify Formats ....................................................... 228
Output Formatted Data in DS2 .............................................. 228
Validation of DS2 Formats .................................................... 229
DS2 Format Examples .......................................................... 229
Format Categories ............................................................... 229
Dictionary ......................................................................... 234
$BASE64xw. Format .............................................................. 234
$BINARYw. Format ............................................................... 235
$CHARw. Format ................................................................. 236
$HEXw. Format ................................................................. 237
$OCTALw. Format ............................................................... 238
$QUOTEw. Format .............................................................. 239
$REVERJw. Format ............................................................. 240
$REVERSw. Format ............................................................ 241
$UPCASEw. Format ............................................................. 242
$w. Format .................................................................. 243
BESTw. Format ................................................................. 244
BESTDOTXw. Format .......................................................... 245
BESTDw.p Format .............................................................. 246
BINARYw. Format .............................................................. 248
COMMAw.d Format ............................................................ 249
COMMAXw.d Format .......................................................... 250
Dw.p Format ................................................................ 251
DATEw. Format ................................................................. 253
DATEAMPMw.d Format ....................................................... 254
DATETIMEw.d Format ........................................................ 256
DAYw. Format ................................................................. 258
DDMMYYw. Format ............................................................ 258
DDMMYYxw. Format .......................................................... 258
DOLLARw.d Format ............................................................ 260
DOLLARXw.d Format .......................................................... 263
DOWNAMEw. Format ........................................................ 264
DTDATEw. Format ............................................................. 265
DTMONYYw. Format .......................................................... 266
DTWKDATXw. Format ......................................................... 268
Chapter 22 • DS2 Formats

- DTYEARw. Format ................................................................. 269
- DTYQYCW. Format ............................................................... 270
- EW. Format ........................................................................... 271
- EUROw.d Format ................................................................... 272
- EUROXw.d Format ................................................................. 273
- FLOATw.d Format ................................................................. 275
- FRACTw. Format ................................................................. 275
- HEXw. Format ....................................................................... 276
- HHMMw.d Format .................................................................. 277
- HOURw.d Format ................................................................. 279
- IEEEw.d Format ..................................................................... 280
- JULIANw. Format .................................................................... 282
- MDYAMPWw.d Format ........................................................ 283
- MMDDYYw. Format .............................................................. 284
- MMDDYYxw. Format ............................................................ 286
- MMSSw.d Format .................................................................... 288
- MMYyw. Format ..................................................................... 289
- MMYxw. Format ..................................................................... 290
- MONNAMEw. Format ............................................................ 292
- MONTHw. Format ................................................................. 293
- MOYyw. Format ..................................................................... 294
- NEGPARENw.d Format .......................................................... 295
- NENGOw. Format ................................................................... 296
- OCTALw. Format .................................................................... 297
- PERCENTw.d Format ............................................................ 298
- PERCENTNw.d Format ......................................................... 299
- QTRw. Format ....................................................................... 301
- QTRRw. Format ..................................................................... 301
- ROMANw. Format ............................................................... 302
- SIZEKw.d Format ................................................................. 303
- SIZEKMGw.d Format .......................................................... 304
- TIMEw.d Format .................................................................... 305
- TIMEw.d Format ................................................................. 305
- TODw.d Format ........................................................................ 307
- VAXRBw.d Format ............................................................... 310
- w.d Format ............................................................................ 311
- WEEKDATEw. Format .......................................................... 312
- WEEKDATEXw. Format ......................................................... 314
- WEEKDAYw. Format ............................................................ 315
- YEARw. Format ...................................................................... 316
- YENw.d Format ..................................................................... 317
- YYMMw. Format ..................................................................... 318
- YYMMxw. Format ................................................................. 319
- YYMMDw. Format ................................................................. 321
- YYMMDXw. Format ............................................................. 323
- YYMOWNw. Format ............................................................. 325
- YYOw. Format ..................................................................... 326
- YYOxw. Format ..................................................................... 327
- YYQRw. Format ..................................................................... 329
- YYQxw. Format ..................................................................... 330
- YYQZw. Format ..................................................................... 332
- Zw.d Format ........................................................................... 333
Overview of Formats

A format is an instruction that DS2 uses to write data values. You use formats to control the written appearance of data values, or, in some cases, to group data values together for analysis. For example, the ROMANw. format, which converts numeric values to roman numerals, writes the numeric value 2006 as MMVI.

General Format Syntax

DS2 formats have the following form:

\[ \text{[ $ ] format [ w ] . [ d ]} \]

Here is an explanation of the syntax:

$  
indicates a character format; its absence indicates a numeric format.

format  
names the format. The format is a DS2 format or a user-defined format that was previously defined with the INVALUE statement in PROC FORMAT. For more information about user-defined formats, see the Base SAS Procedures Guide.

Restriction  
To create and access user-defined formats, a Base SAS session must be available in order to access the SAS catalog file that stores the SAS format definitions.

w  
specifies the format width, which for most formats is the number of columns in the input data.

d  
specifies an optional decimal scaling factor in the numeric formats. DS2 divides the input data by 10 to the power of d.

Tip  
When the value of d is greater than 15, the precision of the decimal value after the fifteenth decimal place might not be accurate.

Formats always contain a period (.) as a part of the name. If you omit the w and the d values from the format, DS2 uses default values. The d value that you specify with a format tells DS2 to display that many decimal places, regardless of how many decimal places are in the data. Formats never change or truncate the internally stored data values.

For example, in DOLLAR10.2, the w value of 10 specifies a maximum of 10 columns for the value. The d value of 2 specifies that two of these columns are for the decimal part of the value, which leaves eight columns for all the remaining characters in the value. This includes the decimal point, the remaining numeric value, a minus sign if the value is negative, the dollar sign, and commas, if any.

If the format width is too narrow to represent a value, DS2 tries to squeeze the value into the space available. Character formats truncate values on the right. Numeric formats sometimes revert to the BESTw.d format. DS2 prints asterisks if you do not specify an adequate width. In the following example, the result is x=**.
x=123;
put x= 2.;

If you use an incompatible format, such as using a numeric format to write character values, DS2 first attempts to use an analogous format of the other type. If this is not feasible, an error message that describes the problem appears in the SAS log.

Using Formats in DS2

How to Specify Formats

In DS2, specify formats as an attribute of the HAVING clause of the DECLARE statement. The HAVING clause provides equivalent functionality to the Base SAS FORMAT and LABEL statements, except that now the attribute must be specified in the declaration statement of the variable.

For example, in the following statement, the columns profit and loss are declared with the EURO13.2 format.

dcl double profit loss having format euro13.2;

Note: In DS2, a format for a column cannot be changed or removed. However, you can use formats as arguments in the PUT function to output data in a different format. For more information, see “Output Formatted Data in DS2” on page 228.

Output Formatted Data in DS2

You can use formats as arguments in the PUT function to output formatted data. If no format is specified in the PUT function, the variable's default format is used.

For example, in this case, the PUT function returns the formatted value of 99 using the BEST12. format.

x=99;
y=put(x, best12.);

If the PUT function is used without a format, all data is output without formatting.

Any type conversions of the value that is formatted are done based on the format name. Any value that is passed to the PUT function with a numeric format is converted, if necessary, to DOUBLE. Any value that is passed to the PUT function with a character format is converted to NCHAR. For more information, see the “PUT Function” on page 725.

DS2 supports SAS formats as follows.

- Both SAS-supplied and user-defined formats can be used. For information about how to create your own format in SAS, see PROC FORMAT in the Base SAS Procedures Guide.

  Note: To create and access user-defined formats, a Base SAS session must be available in order to access the SAS catalog file that stores the SAS format definitions.

- Only the SAS data set and SPD data sets support storing and retrieving a format with a column.
• Formats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.

• You associate SAS formats with a column by using the HAVING clause of the DS2 DECLARE statement. For more information, see “How to Specify Formats” on page 228.

Validation of DS2 Formats

Formats are not validated by a data source or applied to a column until execution time. When metadata for a column is requested, the format name is returned without validation.

DS2 Format Examples

```plaintext
declare datetime shipdate having format dateampm22.2;
  x = put (name, $char8.);
  x = put (price, myformat. -c);
```

Format Categories

Formats can be categorized by the types of values that they operate on. Each DS2 format belongs to one of the following categories:

Character
  writes character data values from character variables.

Date and time
  writes character data values from character variables.

Numeric
  writes numeric data values from numeric variables.

The following table provides brief descriptions of DS2 formats. For more detailed information, see the individual formats.

<table>
<thead>
<tr>
<th>Category</th>
<th>Language Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>$BASE64Xw. Format (p. 234)</td>
<td>Converts character data into ASCII text by using Base 64 encoding.</td>
</tr>
<tr>
<td></td>
<td>$BINARYw. Format (p. 235)</td>
<td>Converts character data to binary representation.</td>
</tr>
<tr>
<td></td>
<td>$CHARw. Format (p. 236)</td>
<td>Writes standard character data.</td>
</tr>
<tr>
<td></td>
<td>$HEXw. Format (p. 237)</td>
<td>Converts character data to hexadecimal representation.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>OCTAL</strong></td>
<td>Format (p. 238)</td>
<td>Converts character data to octal representation.</td>
</tr>
<tr>
<td><strong>QUOTE</strong></td>
<td>Format (p. 239)</td>
<td>Writes data values that are enclosed in double quotation marks.</td>
</tr>
<tr>
<td><strong>REVERJ</strong></td>
<td>Format (p. 240)</td>
<td>Writes character data in reverse order and preserves blanks.</td>
</tr>
<tr>
<td><strong>REVERS</strong></td>
<td>Format (p. 241)</td>
<td>Writes character data in reverse order and left aligns.</td>
</tr>
<tr>
<td><strong>SUPCASE</strong></td>
<td>Format (p. 242)</td>
<td>Converts character data to uppercase.</td>
</tr>
<tr>
<td><strong>w. Format (p. 243)</strong></td>
<td></td>
<td>Writes standard character data.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>Format (p. 253)</td>
<td>Writes SAS date values in the form ddmmyy, ddmmyyyy, or dd-mmm-yyyy.</td>
</tr>
<tr>
<td><strong>DATEAMPM</strong></td>
<td>Format (p. 254)</td>
<td>Writes SAS datetime values in the form ddmmyy:hh:mm:ss.ss with AM or PM.</td>
</tr>
<tr>
<td><strong>DATETIME</strong></td>
<td>Format (p. 256)</td>
<td>Writes SAS datetime values in the form ddmmyy:hh:mm:ss.ss.</td>
</tr>
<tr>
<td><strong>DAY</strong></td>
<td>Format (p. 258)</td>
<td>Writes SAS date values as the day of the month.</td>
</tr>
<tr>
<td><strong>DDMMYY</strong></td>
<td>Format (p. 258)</td>
<td>Writes SAS date values in the form ddm&lt;yy&gt;yy or dd/mm/&lt;yy&gt;yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.</td>
</tr>
<tr>
<td><strong>DDMMYYxw</strong></td>
<td>Format (p. 260)</td>
<td>Writes SAS date values in the form ddm&lt;yy&gt;yy or dd-mm-yy&lt;yy&gt;, where x in the format name is a character that represents the special character that separates the day, month, and year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
</tr>
<tr>
<td><strong>DOWNAME</strong></td>
<td>Format (p. 264)</td>
<td>Writes SAS date values as the name of the day of the week.</td>
</tr>
<tr>
<td><strong>DTDATE</strong></td>
<td>Format (p. 265)</td>
<td>Expects a SAS datetime value as input and writes the SAS date values in the form ddmmyy or ddmmyyyy.</td>
</tr>
<tr>
<td><strong>DTMONYY</strong></td>
<td>Format (p. 266)</td>
<td>Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmmyyyy.</td>
</tr>
<tr>
<td><strong>DTWKDATX</strong></td>
<td>Format (p. 268)</td>
<td>Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).</td>
</tr>
<tr>
<td><strong>DTYEAR</strong></td>
<td>Format (p. 269)</td>
<td>Writes the date part of a SAS datetime value as the year in the form yy or yyyy.</td>
</tr>
<tr>
<td><strong>DTYYQC</strong></td>
<td>Format (p. 270)</td>
<td>Writes the date part of a SAS datetime value as the year and the quarter and separates them with a colon (:).</td>
</tr>
<tr>
<td><strong>HHMM</strong></td>
<td>Format (p. 277)</td>
<td>Writes SAS time values as hours and minutes in the form hh:mm.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HOURw.d Format (p. 279)</td>
<td>Writes SAS time values as hours and decimal fractions of hours.</td>
<td></td>
</tr>
<tr>
<td>JULIANw. Format (p. 282)</td>
<td>Writes SAS date values as Julian dates in the form yyddd or yyyyydd.</td>
<td></td>
</tr>
<tr>
<td>MDYAMPMw.d Format (p. 283)</td>
<td>Writes datetime values in the form mm/dd/yy&lt;yy&gt; hh:mm AM</td>
<td>PM. The year can be either two or four digits.</td>
</tr>
<tr>
<td>MMDDYYw. Format (p. 284)</td>
<td>Writes SAS date values in the form mmdd&lt;yy&gt;yy or mm/dd/&lt;yy&gt;yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>MMDDYYw. Format (p. 286)</td>
<td>Writes SAS date values in the form mmdd&lt;yy&gt;yy or mm/dd/&lt;yy&gt;yy, where the x in the format name is a character that represents the special character, which separates the month, day, and year. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>MMSSw.d Format (p. 288)</td>
<td>Writes SAS time values as the number of minutes and seconds since midnight.</td>
<td></td>
</tr>
<tr>
<td>MMYYw. Format (p. 289)</td>
<td>Writes SAS date values in the form mmM&lt;yy&gt;yy, where M is the separator and the year appears as either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>MMYYxw. Format (p. 290)</td>
<td>Writes SAS date values in the form mm&lt;yy&gt;yy or mm-&lt;yy&gt;yy, where the x in the format name is a character that represents the special character that separates the month and the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>MONNAMEw. Format (p. 292)</td>
<td>Writes SAS date values as the name of the month.</td>
<td></td>
</tr>
<tr>
<td>MONTHw. Format (p. 293)</td>
<td>Writes SAS date values as the month of the year.</td>
<td></td>
</tr>
<tr>
<td>MONYYw. Format (p. 294)</td>
<td>Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.</td>
<td></td>
</tr>
<tr>
<td>NENGOw. Format (p. 296)</td>
<td>Writes SAS date values as Japanese dates in the form e.yymmddd.</td>
<td></td>
</tr>
<tr>
<td>QTRw. Format (p. 301)</td>
<td>Writes SAS date values as the quarter of the year.</td>
<td></td>
</tr>
<tr>
<td>QTRRw. Format (p. 301)</td>
<td>Writes SAS date values as the quarter of the year in Roman numerals.</td>
<td></td>
</tr>
<tr>
<td>TIMEw.d Format (p. 305)</td>
<td>Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss.</td>
<td></td>
</tr>
<tr>
<td>TIMEAMPw.d Format (p. 307)</td>
<td>Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.</td>
<td></td>
</tr>
<tr>
<td>TODw.d Format (p. 308)</td>
<td>Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>WEEKDATEw. Format (p. 312)</td>
<td>Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).</td>
<td></td>
</tr>
<tr>
<td>WEEKDATXw. Format (p. 314)</td>
<td>Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).</td>
<td></td>
</tr>
<tr>
<td>WEEKDAYw. Format (p. 315)</td>
<td>Writes SAS date values as the day of the week.</td>
<td></td>
</tr>
<tr>
<td>YEARw. Format (p. 316)</td>
<td>Writes SAS date values as the year.</td>
<td></td>
</tr>
<tr>
<td>YYMMw. Format (p. 318)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yyMmm, where M is a character separator to indicate that the month number follows the M and the year appears as either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>YYMMxw. Format (p. 319)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yymm or &lt;yy&gt;yy-mm, where the x in the format name is a character that represents the special character that separates the year and the month, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>YYMMDw. Format (p. 321)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yymmdd or &lt;yy&gt;yy–mm–dd, where a hyphen is the separator and the year appears as either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>YYMMDxw. Format (p. 323)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yymmdd or &lt;yy&gt;yy-mm-dd, where the x in the format name is a character that represents the special character that separates the year, month, and day. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>YYYMonw. Format (p. 325)</td>
<td>Writes SAS date values in the form yymmm or yyyyymm.</td>
<td></td>
</tr>
<tr>
<td>YYQw. Format (p. 326)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.</td>
<td></td>
</tr>
<tr>
<td>YYQcw. Format (p. 327)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yyq or &lt;yy&gt;yy-q, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.</td>
<td></td>
</tr>
<tr>
<td>YYQRw. Format (p. 329)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.</td>
<td></td>
</tr>
<tr>
<td>YYQRcw. Format (p. 330)</td>
<td>Writes SAS date values in the form &lt;yy&gt;yyqr or &lt;yy&gt;yy-qr, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits and qr is the quarter of the year expressed in roman numerals.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>YYQZw. Format (p. 332)</td>
<td>Writes SAS date values in the form (&lt;yy&gt;&lt;qq&gt;), the year appears as 2 or 4 digits, and qq is the quarter of the year.</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>BESTw. Format (p. 244)</td>
<td>SAS chooses the best notation.</td>
</tr>
<tr>
<td></td>
<td>BESTDOTXw. Format (p. 245)</td>
<td>Specifies that SAS choose the best notation and use a dot as a decimal separator.</td>
</tr>
<tr>
<td></td>
<td>BESTDw.p Format (p. 246)</td>
<td>Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.</td>
</tr>
<tr>
<td></td>
<td>BINARYw. Format (p. 248)</td>
<td>Converts numeric values to binary representation.</td>
</tr>
<tr>
<td></td>
<td>COMMAw.d Format (p. 249)</td>
<td>Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>COMMAXw.d Format (p. 250)</td>
<td>Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>Dw.p Format (p. 251)</td>
<td>Prints numeric values, possibly with a great range of values, lining up decimal places for values of similar magnitude.</td>
</tr>
<tr>
<td></td>
<td>DOLLARw.d Format (p. 262)</td>
<td>Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>DOLLARXw.d Format (p. 263)</td>
<td>Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>Ew. Format (p. 271)</td>
<td>Writes numeric values in scientific notation.</td>
</tr>
<tr>
<td></td>
<td>EUROw.d Format (p. 272)</td>
<td>Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>EUROXw.d Format (p. 273)</td>
<td>Writes numeric values with a leading euro symbol (E), a period that separates every three digits, and a comma that separates the decimal fraction.</td>
</tr>
<tr>
<td></td>
<td>FLOATw.d Format (p. 275)</td>
<td>Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.</td>
</tr>
<tr>
<td></td>
<td>FRACTw. Format (p. 275)</td>
<td>Converts numeric values to fractions.</td>
</tr>
<tr>
<td></td>
<td>HEXw. Format (p. 276)</td>
<td>Converts real binary (floating-point) values to hexadecimal representation.</td>
</tr>
<tr>
<td></td>
<td>IEEEw.d Format (p. 280)</td>
<td>Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.</td>
</tr>
<tr>
<td></td>
<td>NEGPARENw.d Format (p. 295)</td>
<td>Writes negative numeric values in parentheses.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>OCTAL</td>
<td>$w$. Format (p. 297)</td>
<td>Converts numeric values to octal representation.</td>
</tr>
<tr>
<td>PERCENT</td>
<td>$w.d$ Format (p. 298)</td>
<td>Writes numeric values as percentages.</td>
</tr>
<tr>
<td>PERCENTN</td>
<td>$w.d$ Format (p. 299)</td>
<td>Produces percentages, using a minus sign for negative values.</td>
</tr>
<tr>
<td>ROMAN</td>
<td>$w$. Format (p. 302)</td>
<td>Writes numeric values as roman numerals.</td>
</tr>
<tr>
<td>SIZEK</td>
<td>$w.d$ Format (p. 303)</td>
<td>Writes a numeric value in the form $nK$ for kilobytes.</td>
</tr>
<tr>
<td>SIZEKMG</td>
<td>$w.d$ Format (p. 304)</td>
<td>Writes a numeric value in the form $nKB$ for kilobytes, $nMB$ for megabytes, or $nGB$ for gigabytes.</td>
</tr>
<tr>
<td>VAXRB</td>
<td>$w.d$ Format (p. 310)</td>
<td>Writes real binary (floating-point) data in VMS format.</td>
</tr>
<tr>
<td>$w.d$ Format (p. 311)</td>
<td>Writes standard numeric data one digit per byte.</td>
<td></td>
</tr>
<tr>
<td>YEN</td>
<td>$w.d$ Format (p. 317)</td>
<td>Writes numeric values with yen signs, commas, and decimal points.</td>
</tr>
<tr>
<td>Zw.$d$ Format (p. 333)</td>
<td>Writes standard numeric data with leading 0s.</td>
<td></td>
</tr>
</tbody>
</table>

**Dictionary**

**$\$BASE64Xw$. Format**

Converts character data into ASCII text by using Base 64 encoding.

- **Category:** Character
- **Alignment:** Left

**Syntax**

$\$BASE64Xw.$$

**Arguments**

$w$

specifies the width of the output field.

- **Default** 1
- **Range** 1–32767
Details

Base 64 is an industry encoding method whose encoded characters are determined by using a positional scheme that uses only ASCII characters. Several Base 64 encoding schemes have been defined by the industry for specific uses, such as email or content masking. SAS maps positions 0 – 61 to the characters A – Z, a – z, and 0 – 9. Position 62 maps to the character +, and position 63 maps to the character /.

The following are some uses of Base 64 encoding:

• embed binary data in an XML file
• encode passwords
• encode URLs

The '=' character in the encoded results indicates that the results have been padded with zero bits. In order for the encoded characters to be decoded, the '=' must be included in the value to be decoded.

Example

a=put {'x', $base64x64.};

<table>
<thead>
<tr>
<th>Value of x</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;FCA01A7993BC&quot;</td>
<td>RkNBMDFBMzK5M0JD</td>
</tr>
<tr>
<td>&quot;MyPassword&quot;</td>
<td>TXlQYXNzd29yZA==</td>
</tr>
<tr>
<td>&quot;www.mydomain.com/myhiddenURL&quot;</td>
<td>d3d3Lm15ZG9tYWluLmNvb19teWhpZGRlblVSTA==</td>
</tr>
</tbody>
</table>

$BINARYw. Format

Converts character data to binary representation.

**Category:** Character

**Alignment:** Left

**Syntax**

$BINARYw.

**Arguments**

w

specifies the width of the output field.

**Default** The default width is calculated based on the length of the variable to be printed.

**Range** 1–32767
Comparisons

The $BINARYw.$ format converts character values to binary representation. The BINARYw. format converts numeric values to binary representation.

Example

```plaintext
a = put ('name', $binary16);
```

<table>
<thead>
<tr>
<th>Value of name</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASCII</td>
</tr>
<tr>
<td></td>
<td>---+--------1---+----2</td>
</tr>
<tr>
<td>AB</td>
<td>010000101000010</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “BINARYw. Format” on page 248

$CHARw. Format

Writes standard character data.

Category: Character  
Alignment: Left  
Alias: $w. and $Fw.

Syntax

$CHARw.

Arguments

$w

Specifies the width of the output field. You can specify a number or a column range.

Default: 8 if the length of variable is undefined; otherwise, the length of the variable.

Range: 1–32767

Comparisons

- The $CHARw. format is identical to the $w. and $Fw. formats.
- The $CHARw., $Fw., and $w. formats do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data before output.
Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>a=put ('XYZ', $char.);</td>
<td>XYZ</td>
</tr>
</tbody>
</table>

$\text{HEX}w$. Format

Converts character data to hexadecimal representation.

- **Category**: Character
- **Alignment**: Left

Syntax

$\text{HEX}w$.  

**Arguments**

- $w$ specifies the width of the output field.

- **Default**: The default width is calculated based on the length of the variable to be printed.

- **Range**: 1–32767

- **Tips**: To ensure that SAS writes the full hexadecimal equivalent of your data, make $w$ twice the length of the variable or field that you want to represent. If $w$ is greater than twice the length of the variable that you want to represent, $\text{HEX}w$. pads it with blanks.

Details

The $\text{HEX}w$. format converts each character into two hexadecimal characters. Each blank counts as one character, including trailing blanks.

Comparisons

The HEX$w$. format converts real binary numbers to their hexadecimal equivalent.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>a=put ('bank', $\text{hex}.');</td>
<td>62616E6B</td>
</tr>
</tbody>
</table>
$\textit{OCTAL}w$. Format

Converts character data to octal representation.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Left</td>
</tr>
</tbody>
</table>

**Syntax**

$\textit{OCTAL}w$.  

**Arguments**

\textit{w} specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>The default width is calculated based on the length of the variable to be printed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>1–32767</td>
</tr>
<tr>
<td>Tip</td>
<td>Because each character value generates three octal characters, increase the value of \textit{w} by three times the length of the character value.</td>
</tr>
</tbody>
</table>

**Comparisons**

The $\textit{OCTAL}w$. format converts character values to the octal representation of their character codes. The OCTAL\textit{w}. format converts numeric values to octal representation.

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a=put('art', $octal15.);}</td>
<td>141162164</td>
</tr>
<tr>
<td>\texttt{a=put('rice', $octal15.);}</td>
<td>162151143145</td>
</tr>
<tr>
<td>\texttt{ba=put('bank', $octal15.);}</td>
<td>142141156153</td>
</tr>
</tbody>
</table>

**See Also**

Formats:
$QUOTEw. Format

Writes data values that are enclosed in double quotation marks.

**Category:** Character  
**Alignment:** Left

### Syntax

$QUOTEw.

### Arguments

\(w\)

specifies the width of the output field.

**Default**

2 if the length of the variable is undefined; otherwise, the length of the variable + 2

**Range**

2–32767

**Tip**

Make \(w\) wide enough to include the left and right quotation marks.

### Details

The following list describes the output that SAS produces when you use the $QUOTEw. format.

- When your data value is not enclosed in quotation marks, SAS encloses the output in double quotation marks.
- When your data value is not enclosed in quotation marks, but the value contains a single quotation mark, SAS takes the following action:
  - encloses the data value in double quotation marks
  - does not change the single quotation mark.
- When your data value begins and ends with single quotation marks, and the value contains double quotation marks, SAS takes the following action:
  - encloses the data value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
- When your data value begins and ends with single quotation marks, and contains two single contiguous quotation marks, SAS takes the following action:
  - encloses the value in double quotation marks
  - does not change the single quotation marks.
- When your data value begins and ends with single quotation marks, and contains both double quotation marks and single, contiguous quotation marks, SAS takes the following action:
• encloses the value in double quotation marks
• duplicates the double quotation marks that are found in the data value
• does not change the single quotation marks.

• When the length of the target field is not large enough to contain the string and its quotation marks, SAS returns all blanks.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put('SAS',QUOTE.);</td>
<td>&quot;SAS&quot;</td>
</tr>
<tr>
<td>a=put('SAS''s',QUOTE.);</td>
<td>&quot;SAS's&quot;</td>
</tr>
<tr>
<td>a=put('ad''verb',QUOTE16.);</td>
<td>&quot;ad'verb&quot;</td>
</tr>
<tr>
<td>a=put(&quot;ad&quot; &quot;verb&quot;,QUOTE16.);</td>
<td>&quot;&quot;ad&quot;&quot;&quot;&quot;verb&quot;&quot;</td>
</tr>
<tr>
<td>a=put(&quot;ad&quot;''&quot;verb&quot;&quot;,QUOTE20);</td>
<td>&quot;&quot;&quot;&quot;ad&quot;&quot;&quot;&quot;&quot;&quot;verb&quot;&quot;&quot;&quot;</td>
</tr>
</tbody>
</table>

$REVERJw. Format

Writes character data in reverse order and preserves blanks.

Category: Character
Alignment: Right

Syntax

$REVERJw.

Arguments

w
specifies the width of the output field.

Default 1 if w is not specified
Range 1–32767

Comparisons

The $REVERJw. format is similar to the $REVERSw. format except that $REVERSw. left aligns the result by trimming all leading blanks.
### Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put('ABCD###',$reverj7.);</td>
<td>###DCBA</td>
</tr>
<tr>
<td>a=put('###ABCD',$reverj7.);</td>
<td>DCBA###</td>
</tr>
</tbody>
</table>

*Note:* The character # represents a blank space.

### See Also

**Formats:**
- “$REVERSw. Format” on page 241

---

### $REVERSw. Format

Writes character data in reverse order and left aligns.

**Category:** Character  
**Alignment:** Left

#### Syntax

$REVERSw. $w$

**Arguments**

$w$

specifies the width of the output field.

**Default**  
1 if $w$ is not specified

**Range**  
1–32767

#### Comparisons

The $REVERSw.$ format is similar to the $REVERJw.$ format except that $REVERJw.$ does not left align the result.

#### Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--------</td>
</tr>
</tbody>
</table>

---
Statements | Results
---|---
a=put('ABCD###',$revers7); | DCBA

Note: The character # represents a blank space.

See Also

Formats:
- “$REVERJw. Format” on page 240

$UPCASEw. Format

Converts character data to uppercase.

Category: Character
Alignment: Left

Syntax

$UPCASEw.

Arguments

w

specifies the width of the output field.

Default 8 if the length of the variable is undefined; otherwise, the length of the variable.

Range 1–32767

Details

Special characters, such as hyphens and other symbols, are not altered.

Example

Statements | Results
---|---

---+---1

a=put('coxe-ryan',upcase9.); | COXE-RYAN
$w$. Format

Writestandard character data.

**Category:** Character  
**Alignment:** Left  
**Alias:** $Fw.$ and $CHARw.$

### Syntax

\[
\text{}\text{}\text{}\text{}\text{}\text{}\text{}$w.$
\]

### Arguments

\[w\] specifies the width of the output field. You can specify a number or a column range.

**Default**  
1 if the length of the identifier is undefined; otherwise, the length of the identifier

**Range**  
1–32767

### Comparisons

The $w.$, $Fw.$, and the $CHARw.$ formats are identical, and they do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data before output.

### Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(' Cary',5.);</td>
<td>Cary</td>
</tr>
<tr>
<td>a=put(' Tokyo',5.);</td>
<td>Tokyo</td>
</tr>
</tbody>
</table>

* The character # represents a blank space.

### See Also

**Functions:**

- “LEFT Function” on page 602
BESTw. Format

SAS chooses the best notation.

**Category:** Numeric  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

**Syntax**

BESTw.

**Arguments**

**w**  
specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1–32</td>
</tr>
</tbody>
</table>

**Tip**  
If you print numbers between 0 and .01 exclusively, then use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and -.01 exclusively, use a field width of at least 8.

**Details**

The BESTw. format is the default format for writing numeric values. When there is no format specification, SAS chooses the format that provides the most information about the value according to the available field width. BESTw. rounds the value, and if SAS can display at least one significant digit in the decimal portion, within the width specified, BESTw. produces the result in decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.

**Comparisons**

- The BESTw. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The BESTDOTXw format writes as many significant digits as possible in the output field. Integers are printed with a decimal.
- The Dw.p format writes numbers with the desired precision and more alignment than the BESTw. format.
- The w.d format aligns decimal points, if possible, but does not necessarily show the same precision for all numbers.
Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----</td>
<td>----+----</td>
</tr>
<tr>
<td>a=put(1257000,best6.);</td>
<td>1.26E6</td>
</tr>
<tr>
<td>a=put(1257000,best3.);</td>
<td>1E6</td>
</tr>
</tbody>
</table>

See Also

Formats:

- “BESTDw.p Format” on page 246
- “BESTDOTXw. Format” on page 245
- “Dw.p Format” on page 251
- “w.d Format” on page 311

BESTDOTXw. Format

Specifies that SAS choose the best notation and use a dot as a decimal separator.

**Category:** Numeric

**Alignment:** Right

**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

BESTDOTX.w.

**Arguments**

w

specifies the width of the output field.

(Default 12)

**Range** 1–32

**Tip** If you print numbers between 0 and .01 exclusively, then use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and -.01 exclusively, use a field width of at least 8.
Details
If the NLDECSEPARATOR system option is disabled, the BEST\(w\) and BESTDOTX\(w\) formats process data the same way. If the NLDECSEPARATOR system option is enabled, then the results from the BEST\(w\) and BESTDOTX\(w\) formats are different. See the following table to understand the differences:

<table>
<thead>
<tr>
<th>LOCALE option</th>
<th>Default decimal separator character for the locale</th>
<th>NLDECSEPARATOR option</th>
<th>Separator character used by BEST(w)</th>
<th>Separator character used by BESTDOTX(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>en_US</td>
<td>Dot</td>
<td>Disabled (default)</td>
<td>Dot</td>
<td>Dot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enabled</td>
<td>Dot</td>
<td>Dot</td>
</tr>
<tr>
<td>fr_FR</td>
<td>Comma</td>
<td>Disabled (default)</td>
<td>Dot</td>
<td>Dot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enabled</td>
<td>Comma</td>
<td>Dot</td>
</tr>
</tbody>
</table>

For more information about the NLDECSEPARATOR system option, see SAS National Language Support (NLS): Reference Guide.

Comparisons
The BESTDOTX\(w\) format writes as many significant digits as possible in the output field. Integers are printed with a decimal. The BEST\(w\) format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.

See Also

Formats:

- “BEST\(w\). Format” on page 244

BESTD\(w,p\) Format
Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Right</td>
</tr>
<tr>
<td>Interaction:</td>
<td>When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.</td>
</tr>
</tbody>
</table>

Syntax

BESTD\(w,p\)
**Arguments**

\[ w \]

specifies the width of the output field.

**Default**

12

**Range**

1–32

**p**

specifies the precision.

**Default**

3

**Range**

0 to \( w - 1 \)

**Requirement**

must be less than \( w \)

**Tip**

If \( p \) is omitted or is specified as 0, then \( p \) is set to 3.

**Details**

The BESTD\( w.p \) format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Integers are printed without a decimal point. Larger values of \( p \) print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of \( p \) print the data values with less precision and a greater chance of decimal point alignment.

The format chooses the number of decimal places to be printed for ranges of values, even when the underlying values can be represented with fewer decimal places.

**Comparisons**

- The BEST\( w. \) format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The D\( w.p \) format writes numbers with the desired precision and more alignment than the BEST\( w. \) format.
- The BESTD\( w.p \) format is a combination of the BEST\( w. \) format and the D\( w.p \) format in that it formats all numeric data, and it does a better job of aligning decimals than the BEST\( w. \) format.
- The \( w.d \) format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>----+----1----+</td>
<td></td>
</tr>
<tr>
<td>a=put(12345, bestd14.);</td>
<td>12345</td>
</tr>
<tr>
<td>a=put(123.45, bestd14.);</td>
<td>123.45000000</td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
a=put(1.2345, bestd14.); | 1.2345000
a=put(.12345, bestd14.); | 0.1234500
a=put(1.23456789, bestd14.); | 1.23456789

See Also

Formats:
- “BESTw. Format” on page 244
- “Dw.p Format” on page 251
- “w.d Format” on page 311

**BINARYw. Format**

Converts numeric values to binary representation.

- **Category:** Numeric
- **Alignment:** Left

**Syntax**

`BINARYw.`

**Arguments**

`w`

specifies the width of the output field.

- **Default:** 8
- **Range:** 1–64

**Comparisons**

BINARYw. converts numeric values to binary representation. The $BINARYw.$ format converts character values to binary representation.

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = put (123.45, binary8.);</td>
<td>011110111</td>
</tr>
<tr>
<td>a = put (1.2345, binary8.);</td>
<td>011110111</td>
</tr>
</tbody>
</table>
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = put (123, binary8.);</code></td>
<td>01111011</td>
</tr>
<tr>
<td><code>a = put (-123, binary8.);</code></td>
<td>10000101</td>
</tr>
</tbody>
</table>

### See Also

**Formats:**
- “$BINARYw. Format” on page 235

### COMMA\(w.d\) Format

Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.

**Category:** Numeric  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDEEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

### Syntax

`COMMA\(w.[d]\)`

**Arguments**

\(w\)

- specifies the width of the output field.  
  - **Default**: 6  
  - **Range**: 1–32  
  - **Tip**: Make \(w\) wide enough to write the numeric values, the commas, and the optional decimal point.

\(d\)

- specifies the number of digits to the right of the decimal point in the numeric value.  
  - **Range**: 0–31  
  - **Requirement**: must be less than \(w\)
Comparisons

- The COMMA\(w.d\) format is similar to the COMMAX\(w.d\) format, but the COMMAX\(w.d\) format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The COMMA\(w.d\) format is similar to the DOLLAR\(w.d\) format except that the COMMA\(w.d\) format does not print a leading dollar sign.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (23451.23,comma10.2);</td>
<td>23,451.23</td>
</tr>
<tr>
<td>a=put (123451.234,comma10.2);</td>
<td>123,451.23</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “COMMAX\(w.d\) Format” on page 250
- “DOLLAR\(w.d\) Format” on page 262

COMMAX\(w.d\) Format

Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.

Category: Numeric
Alignment: Right
Interaction: When the DECIMALCONV= system option is set to STDIEEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

COMMAX\([w.d]\)

Arguments

\(w\)
- specifies the width of the output field.
  - Default: 6
  - Range: 1–32
Tip: Make \( w \) wide enough to write the numeric values, the commas, and the optional decimal point.

\( d \)

specifies the number of digits to the right of the decimal point in the numeric value.

Range: 0–31

Requirement: must be less than \( w \)

Comparisons

The COMMA\( w.d \) format is similar to the COMMAX\( w.d \) format, but the COMMAX\( w.d \) format reverses the roles of the decimal point and the comma. This convention is common in European countries.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=\text{put} (23451.23,\text{commax}10.2); )</td>
<td>23.451,23</td>
</tr>
<tr>
<td>( a=\text{put} (123451.234,\text{commax}10.2); )</td>
<td>123.451,23</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “COMMA\( w.d \) Format” on page 249

Dw.p Format

Prints numeric values, possibly with a great range of values, lining up decimal places for values of similar magnitude.

- **Category:** Numeric
- **Alignment:** Right
- **Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

\[ \text{D}[w.p] \]
Arguments

\[ w \]

specifies the width of the output field.

Default 12
Range 1–32

\[ p \]

specifies the significant digits.

Default 3
Range 0–16
Requirement must be less than \( w \)

Details

The \( D_w.p \) format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Larger values of \( p \) print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of \( p \) print the data values with less precision and a greater chance of decimal point alignment.

Comparisons

- The BEST\( w \) format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up.
- \( D_w.p \) writes numbers with the desired precision and more alignment than BEST\( w \) format

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (12345,d10.4);</td>
<td>12345.0</td>
</tr>
<tr>
<td>a=put (1234.5,d10.4);</td>
<td>1234.5</td>
</tr>
<tr>
<td>a=put (123.45,d10.4);</td>
<td>123.45000</td>
</tr>
<tr>
<td>a=put (12.345,d10.4);</td>
<td>12.34500</td>
</tr>
<tr>
<td>a=put (1.2345,d10.4);</td>
<td>1.23450</td>
</tr>
<tr>
<td>a=put (.12345,d10.4);</td>
<td>0.12345</td>
</tr>
</tbody>
</table>

See Also

Formats:
DATEw. Format

Writes SAS date values in the form ddmmyy, ddmmyyyy, or dd-mmm-yyyy.

**Category:** Date and Time

**Alignment:** Right

---

**Syntax**

`DATEw.`

**Arguments**

`w`

specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5–11</td>
</tr>
</tbody>
</table>

**Tip**

Use a width of 9 to print a 4-digit year without a separator between the day, month, and year. Use a width of 11 to print a 4-digit year using a hyphen as a separator between the day, month, and year.

---

**Details**

The DATEw. format writes SAS date values in the form ddmmyy, ddmmyyyy, or dd-mmm-yyyy. Here is an explanation of the syntax:

- `dd` is an integer that represents the day of the month.
- `mmm` is the first three letters of the month name.
- `yy` or `yyyy` is a two-digit or four-digit integer that represents the year.

---

**Example**

The example table uses the input value of 17241, which is the SAS date value that corresponds to March 16, 2007.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (17241,date5.);</td>
<td>16MAR</td>
</tr>
<tr>
<td>a=put (17241,date6.);</td>
<td>16MAR</td>
</tr>
<tr>
<td>Statements</td>
<td>Results</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>a=put (17241,date7.);</code></td>
<td>16MAR07</td>
</tr>
<tr>
<td><code>a=put (17241,date8.);</code></td>
<td>16MAR07</td>
</tr>
<tr>
<td><code>a=put (17241,date9.);</code></td>
<td>16MAR2007</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Functions:**

- “DATE Function” on page 466

---

### DATEAMPMw.d Format

Writes SAS datetime values in the form `ddmmmyy:hh:mm:ss.ss` with AM or PM.

**Category:** Date and Time  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

**Syntax**

`DATEAMPMw.[d]`

**Arguments**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| w | specifies the width of the output field.  
Default 19  
Range 7–40  
Tip SAS requires a minimum `w` value of 13 to write AM or PM. For widths between 10 and 12, SAS writes a 24-hour clock time.  

| d | specifies the number of digits to the right of the decimal point in the seconds value.  
Range 0–39  
Requirement must be less than `w`  
Note If `w–d< 17`, SAS truncates the decimal values.
Details

The DATEAMPMw.d format writes SAS datetime values in the form ddmmyy:hh:mm:ss.ss. Here is an explanation of the syntax:

- **dd**: is an integer that represents the day of the month.
- **mmm**: is the first three letters of the month name.
- **yy**: is a two-digit integer that represents the year.
- **hh**: is an integer that represents the hour.
- **mm**: is an integer that represents the minutes.
- **ss.ss**: is the number of seconds to two decimal places.

Comparisons

The DATEAMPMw.d format is similar to the DATETIMEw.d format except that DATEAMPMw.d prints AM or PM at the end of the time.

Example

The example table uses the input value of 1489955694, which is the SAS datetime value that corresponds to 01:14:54 PM on March 19, 2007.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (1489955694, dateampm.);</td>
<td>19MAR07:01:14:54 PM</td>
</tr>
<tr>
<td>a=put (1489955694, dateampm7.);</td>
<td>19MAR07</td>
</tr>
<tr>
<td>a=put (1489955694, dateampm10.);</td>
<td>19MAR07:13</td>
</tr>
<tr>
<td>a=put (1489955694, dateampm13.);</td>
<td>19MAR07:01 PM</td>
</tr>
<tr>
<td>a=put (1489955694, dateampm22.2.);</td>
<td>19MAR2007:01:14:54.00 PM</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “DATETIMEw.d Format” on page 256
**DATETIMEw.d Format**

Writes SAS datetime values in the form `ddmmmyy:hh:mm:ss.ss`.

**Category:** Date and Time  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

**Syntax**

`DATETIMEw.[d]`

**Arguments**

`w`

specifies the width of the output field.

- **Default:** 16  
- **Range:** 7–40  
- **Tip:** SAS requires a minimum `w` value of 16 to write a SAS datetime value with the date, hour, and seconds. Add an additional two places to `w` and a value to `d` to return values with optional decimal fractions of seconds.

`d`

specifies the number of digits to the right of the decimal point in the seconds value.

- **Range:** 0–39  
- **Requirement:** must be less than `w`

**Details**

The DATETIMEw.d format writes SAS datetime values in the form `ddmmmyy:hh:mm:ss.ss`. Here is an explanation of the syntax:

- `dd` is an integer that represents the day of the month.
- `mmm` is the first three letters of the month name.
- `yy` is a two-digit integer that represents the year.
- `hh` is an integer that represents the hour in 24–hour clock time.
- `mm` is an integer that represents the minutes.
**ss.ss**

is the number of seconds to two decimal places.

*Note:* If \( w=d,17 \), SAS truncates the decimal values.

## Comparisons

The DATEAMPM\( w,d \) format is similar to the DATETIME\( w,d \) format except that DATEAMPM\( w,d \) prints AM or PM at the end of the time.

## Example

The example table uses the input value of 1510285759, which is the SAS datetime value that corresponds to 3:49:19 AM on November 10, 2007.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>---+----1----+----2</td>
</tr>
<tr>
<td>a=put (1510285759,datetime.);</td>
<td>10NOV07:03:49:19</td>
</tr>
<tr>
<td>a=put (1510285759,datetime7.);</td>
<td>10NOV07</td>
</tr>
<tr>
<td>a=put (1510285759,datetime12.);</td>
<td>10NOV07:03</td>
</tr>
<tr>
<td>a=put (1510285759,datetime18.);</td>
<td>10NOV07:03:49:19</td>
</tr>
<tr>
<td>a=put (1510285759,datetime18.1);</td>
<td>10NOV07:03:49:19.0</td>
</tr>
<tr>
<td>a=put (1510285759,datetime19.);</td>
<td>10NOV2007:03:49:19</td>
</tr>
<tr>
<td>a=put (1510285759,datetime20.1);</td>
<td>10NOV2007:03:49:19.0</td>
</tr>
<tr>
<td>a=put (1510285759,datetime21.2);</td>
<td>10NOV2007:03:49:19.00</td>
</tr>
</tbody>
</table>

## See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

## Formats:

- “DATEw. Format” on page 253
- “DATEAMPMw.d Format” on page 254
- “TIMEw.d Format” on page 305

## Functions:

- “DATETIME Function” on page 468
**DAYw. Format**

Writes SAS date values as the day of the month.

**Category:** Date and Time  
**Alignment:** Right

---

**Syntax**

`DAYw:`

**Arguments**

`w`  
specifies the width of the output field.

**Default** 2  
**Range** 2–32

---

**Example**

The example table uses the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.


<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a=put(18792,day2.);</code></td>
<td>14</td>
</tr>
</tbody>
</table>

---

**See Also**

Chapter 14, “Dates and Times in DS2,” on page 111

---

**DDMMYYw. Format**

Writes SAS date values in the form `ddmm<yy>` or `dd/mml<yy>`, where a forward slash is the separator and the year appears as either 2 or 4 digits.

**Category:** Date and Time  
**Alignment:** Right

---

**Syntax**

`DDMMYYw.`
**Arguments**

*w*

specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>2–10</td>
</tr>
<tr>
<td>Interaction</td>
<td>When <em>w</em> has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When <em>w</em> is 7, the date appears as a two-digit year without slashes.</td>
</tr>
</tbody>
</table>

**Details**

The DDMMYY*w.* format writes SAS date values in the form `ddmm<yy>yy` or `dd/mm/<yy>yy`. Here is an explanation of the syntax:

- **dd** is an integer that represents the day of the month.
- `/` is the separator.
- **mm** is an integer that represents the month.
- `<yy>yy` is a two-digit or four-digit integer that represents the year.

**Example**

The following examples use the input value of 18985, which is the SAS date value that corresponds to December 24, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18985,ddmmyy.);</td>
<td>24/12/11</td>
</tr>
<tr>
<td>a=put(18985,ddmmyy5.);</td>
<td>24/12</td>
</tr>
<tr>
<td>a=put(18985,ddmmyy6.);</td>
<td>241211</td>
</tr>
<tr>
<td>a=put(18985,ddmmyy7.);</td>
<td>241211</td>
</tr>
<tr>
<td>a=put(18985,ddmmyy8.);</td>
<td>24/12/11</td>
</tr>
<tr>
<td>a=put(18985,ddmmyy10.);</td>
<td>24/12/2011</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**
DDMMYYxw. Format

Writes SAS date values in the form *ddmm<yy>yy* or *dd-mm-yyyy*, where *x* in the format name is a character that represents the special character that separates the day, month, and year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

**Category:** Date and Time  
**Alignment:** Right

**Syntax**

```
DDMMYYxw.
```

**Arguments**

*\( x \)*  
identifies a separator or specifies that no separator appear between the day, the month, and the year. Here are the valid values:

- **B** separates with a blank
- **C** separates with a colon
- **D** separates with a hyphen
- **N** indicates no separator
- **P** separates with a period
- **S** separates with a slash.

*\( w \)*  
specifies the width of the output field.

**Default** 8  
**Range** 2–10
Interactions  When \( w \) has a value of from 2 to 5, the date appears with as much of
the day and the month as possible. When \( w \) is 7, the date appears as a
two-digit year without separators.

When \( x \) has a value of N, the width range changes to 2–8.

Details

The DDMMYYxw. format writes SAS date values in the form \( ddmm<yy>yy \) or
\( ddXmmX<yy>yy \). Here is an explanation of the syntax:

\( dd \)

is an integer that represents the day of the month.

\( X \)

is a specified separator.

\( mm \)

is an integer that represents the month.

\(<yy>yy \)

is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 18702, which is the SAS date value that
corresponds to March 16, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18702,ddmmyy&lt;5.);</td>
<td>16:03</td>
</tr>
<tr>
<td>a=put(18702,ddmmyyd8.);</td>
<td>16-03-11</td>
</tr>
<tr>
<td>a=put(18702,ddmmyyyn8.);</td>
<td>16032011</td>
</tr>
<tr>
<td>a=put(18702,ddmmyyp10.);</td>
<td>16.03.2011</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “DATEw. Format” on page 253
- “DDMMYYw. Format” on page 258
- “MMDDYYxw. Format” on page 286
- “YYMMDxDDw. Format” on page 323

Functions:

- “DAY Function” on page 469
DOLLARw.d Format

Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

Category: Numeric
Alignment: Right
Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

DOLLARw.[d]

Arguments

w
specifies the width of the output field.

Default 6
Range 2–32

d
specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31
Requirement must be less than w

Details

The DOLLARw.d format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLARw.d always produces one of these codes.

Comparisons

- The DOLLARw.d format is similar to the DOLLARXw.d format, but the DOLLARXw.d format reverses the roles of the decimal point and the comma. This convention is common in European countries.
• The DOLLARw.d format is the same as the COMMAw.d format except that the COMMAw.d format does not write a leading dollar sign.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1254.71,dollar10.2);</td>
<td>$1,254.71</td>
</tr>
</tbody>
</table>

See Also

Formats:
• “COMMAw.d Format” on page 249
• “DOLLARXw.d Format” on page 263

DOLLARXw.d Format

Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.

**Category:** Numeric  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

**Syntax**

DOLLARXw.\[d\]

**Arguments**

\(w\)

specifies the width of the output field.

Default 6  
Range 2–32

\(d\)

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0  
Range 2–31
Requirement must be less than $w$

Details
The DOLLARXw.d format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLARXw.d always produces one of these codes.

Comparisons
- The DOLLARXw.d format is similar to the DOLLARw.d format, but the DOLLARXw.d format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLARXw.d format is the same as the COMMAXw.d format except that the COMMAw.d format does not write a leading dollar sign.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1254.71,dollarx10.2);</td>
<td>$1.254,71</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “COMMAXw.d Format” on page 250
- “DOLLARw.d Format” on page 262

DOWNAMEw. Format

Writes SAS date values as the name of the day of the week.

Category: Date and Time
Alignment: Right

Syntax

DOWNAMEw.
Arguments

\(w\)

specifies the width of the output field.

Default 9

Range 1–32

Tip If you omit \(w\), SAS prints the entire name of the day.

Details

If necessary, SAS truncates the name of the day to fit the format width. For example, the format DOWNAME2. prints the first two letters of the day name.

Example

The example table uses the input value of 18702, which is the SAS date value that corresponds to March 16, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18702,downame.);</td>
<td>Wednesday</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “WEEKDAYw. Format” on page 315

DTDATEw. Format

Expects a SAS datetime value as input and writes the SAS date values in the form \(ddmmmyy\) or \(ddmmmyyyy\).

Category: Date and Time

Alignment: Right

Syntax

\texttt{DTDATEw.}

Arguments

\(w\)

specifies the width of the output field.
Default  7
Range    5–9
Tip      Use a width of 9 to print a 4-digit year.

Details
The DTDATEw. format writes SAS date values in the form dddmmyy or dddmmyyyy. Here is an explanation of the syntax:

\[ \begin{align*}
    \text{dd} & \quad \text{is an integer that represents the day of the month.} \\
    \text{mmm} & \quad \text{are the first three letters of the month name.} \\
    \text{yy or yyyy} & \quad \text{is a two-digit or four-digit integer that represents the year.}
\end{align*} \]

Comparisons
The DTDATEw. format produces the same type of output that the DATEw. format produces. The difference is that the DTDATEw. format requires a SAS datetime value.

Example
The example table uses the input value of 1636516159, which is the SAS datetime value that corresponds to 3:49:19 a.m. on November 10, 2011, and prints both a two-digit and a four-digit year for the DTDATEw. format.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1636516159,dtdate.);</td>
<td>10NOV11</td>
</tr>
<tr>
<td>a=put(1636516159,dtdate9.);</td>
<td>10NOV2011</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “DATEw. Format” on page 253

DTMONYYw. Format
Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.

Category: Date and Time
Alignment: Right
Syntax

DTMONYYw.

Arguments

w

specifies the width of the output field.

Default 5

Range 5–7

Details

The DTMONYYw. format writes SAS datetime values in the form mmmyy or mmmyyyy. Here is an explanation of the syntax:

mmm

is the first three letters of the month name.

yy or yyyy

is a two-digit or four-digit integer that represents the year.

Comparisons

The DTMONYYw. format and the MONYYw. format are similar in that they both write date values. The difference is that DTMONYYw. expects a SAS datetime value as input, and MONYYw. expects a SAS date value.

Example

The example table uses as input the value 1634364532, which is the SAS datetime value that corresponds to October 16, 2011, at 06:08:52 a.m.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1634364532,dtmonyy.);</td>
<td>OCT11</td>
</tr>
<tr>
<td>a=put(1634364532,dtmonyy5.);</td>
<td>OCT11</td>
</tr>
<tr>
<td>a=put(1634364532,dtmonyy6.);</td>
<td>OCT11</td>
</tr>
<tr>
<td>a=put(1634364532,dtmonyy7.);</td>
<td>OCT2011</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
DTWKDATXw. Format

Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).

Category: Date and Time
Alignment: Right

Syntax

DTWKDATXw.

Arguments

w
specifies the width of the output field.

Default 29
Range 3–37

Details

The DTWKDATXw. format writes SAS date values in the form day-of-week, dd month-name, yy, or yyyy. Here is an explanation of the syntax:

day-of-week
is either the first three letters of the day name or the entire day name.

dd
is an integer that represents the day of the month.

month-name
is either the first three letters of the month name or the entire month name.

yy or yyyy
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTWKDATXw. format is similar to the WEEKDATXw. format in that they both write date values. The difference is that DTWKDATXw. expects a SAS datetime value as input, and WEEKDATXw. expects a SAS date value.

Example

The example table uses as input the value 1634364532, which is the SAS datetime value that corresponds to October 16, 2011, at 06:08:52 a.m.
<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1634364532,dtwkdatx.);</td>
<td>Sunday, 16 October 2011</td>
</tr>
<tr>
<td>a=put(1634364532,dtwkdatx3.);</td>
<td>Sun</td>
</tr>
<tr>
<td>a=put(1634364532,dtwkdatx8.);</td>
<td>Sun</td>
</tr>
<tr>
<td>a=put(1634364532,dtwkdatx25.);</td>
<td>Sunday, 16 Oct 2011</td>
</tr>
</tbody>
</table>

**See Also**
- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**
- “DATETIMEw.d Format” on page 256
- “WEEKDATXw. Format” on page 314

---

**DTYEARw. Format**

Writes the date part of a SAS datetime value as the year in the form yy or yyyy.

**Category:** Date and Time  
**Alignment:** Right

**Syntax**

`DTYEARw.`

**Arguments**

`w`

specifies the width of the output field.

- **Default:** 4  
- **Range:** 2–4

**Comparisons**

The DTYEARw. format is similar to the YEARw. format in that they both write date values. The difference is that DTYEARw. expects a SAS datetime value as input, and YEARw. expects a SAS date value.
Example

The example table uses as input the value 1634364532, which is the SAS datetime value that corresponds to October 16, 2011, at 06:08:52 a.m.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1634364532,dtyear.);</td>
<td>2011</td>
</tr>
<tr>
<td>a=put(1634364532,dtyear2.);</td>
<td>11</td>
</tr>
<tr>
<td>a=put(1634364532,dtyear3.);</td>
<td>11</td>
</tr>
<tr>
<td>a=put(1634364532,dtyear4.);</td>
<td>2011</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “DATETIMEw.d Format” on page 256
- “YEARw. Format” on page 316

DTYYQCw. Format

Writes the date part of a SAS datetime value as the year and the quarter and separates them with a colon (::).

**Category:** Date and Time

**Alignment:** Right

**Syntax**

`DTYYQCw.`

**Arguments**

`w`

specifies the width of the output field.

**Default** 4

**Range** 4–6

**Details**

The DTYYQCw. format writes SAS datetime values in the form yy or yyyy, followed by a colon (:) and the numeric value for the quarter of the year.
Example

The example table uses as input the value 1634364532, which is the SAS datetime value that corresponds to October 16, 2011, at 06:08:52 p.m.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (1634364532, dtyyqc.);</td>
<td>11:4</td>
</tr>
<tr>
<td>a=put (1634364532, dtyyqc4.);</td>
<td>11:4</td>
</tr>
<tr>
<td>a=put (1634364532, dtyyqc5.);</td>
<td>11:4</td>
</tr>
<tr>
<td>a=put (1634364532, dtyyqc6.);</td>
<td>2011:4</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “DATETIMEw.d Format” on page 256

Ew. Format

Writes numeric values in scientific notation.

- **Category:** Numeric
- **Alignment:** Right
- **Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

Ew.

Arguments

w

specifies the width of the output field.

- **Default:** 12
- **Range:** 7–32
Details
SAS reserves the first column of the result for a minus sign.

Example
The example table uses the input value of 1257.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1257,e10.);</td>
<td>1.257E+03</td>
</tr>
<tr>
<td>a=put(-1257,e10.);</td>
<td>-1.257E+03</td>
</tr>
</tbody>
</table>

EUROw.d Format
Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction.

Category: Numeric
Alignment: Right

Syntax
EUROw.d

Arguments
w
specifies the width of the output field.

Default 6
Range 1-32
Tip If you want the euro symbol to be part of the output, be sure to choose an adequate width.

d
specifies the number of digits to the right of the decimal point in the numeric value.

Default 0
Range 0-31
Requirement must be less than w
Comparisons

• The EUROw.d format is similar to the EUROXw.d format, but EUROXw.d format reverses the roles of the decimal point and the comma. This convention is common in European countries.

• The EUROw.d format is similar to the DOLLARw.d format, except that DOLLARw.d format writes a leading dollar sign instead of the euro symbol.

Note: The EUROXw.d format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an error occurs. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>1,254.71</td>
</tr>
<tr>
<td></td>
<td>1,255</td>
</tr>
<tr>
<td></td>
<td>1,254.71</td>
</tr>
<tr>
<td></td>
<td>1,254.710</td>
</tr>
</tbody>
</table>

See Also

Formats:

• “EUROXw.d Format” on page 273
Tip
If you want the euro symbol to be part of the output, be sure to choose an adequate width.

$d$

specifies the number of digits to the right of the decimal point in the numeric value.

<table>
<thead>
<tr>
<th>Default</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>0 – 31</td>
</tr>
<tr>
<td>Requirement</td>
<td>must be less than $w$</td>
</tr>
</tbody>
</table>

Comparisons

- The EUROX$_w$.d format is similar to the EURO$_w$.d format, but EURO$_w$.d format reverses the roles of the comma and the decimal point. This convention is common in English-speaking countries.
- The EUROX$_w$.d format is similar to the DOLLARX$_w$.d format, except that DOLLARX$_w$.d format writes a leading dollar sign instead of the euro symbol.

Note: The EUROX$_w$.d format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an error occurs. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1254.71, eurox10.2);</td>
<td>€1.254,71</td>
</tr>
<tr>
<td>a=put(1254.71, eurox5.);</td>
<td>1.255</td>
</tr>
<tr>
<td>a=put(1254.71, eurox9.2);</td>
<td>€1.254,71</td>
</tr>
<tr>
<td>a=put(1254.71, eurox15.3);</td>
<td>€1.254,710</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “DOLLARX$_w$.d Format” on page 263
- “EURO$_w$.d Format” on page 272
**FLOATw.d Format**

Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the $d$th power.

- **Category:** Numeric
- **Alignment:** Left

**Syntax**

`FLOATw.[d]`

**Arguments**

- **$w$**
  - specifies the width of the output field.
  - **Requirement** width must be 4

- **$d$**
  - specifies the power of 10 by which to multiply the value.
  - **Default** 0
  - **Range** 0 – 31

**Details**

Values that are written by FLOAT4. typically are those meant to be read by some other external program that runs in your operating environment and that expects these single-precision values. If the value that is to be formatted is a missing value, or if it is out-of-range for a native single-precision, floating-point value, a single-precision value of zero is generated.

**Example**

In the example below, you use the VARBINARY data type in the DECLARE statement to get a hexadecimal representation of a binary number.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1,float4.);</td>
<td>0000803F</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number that is stored in IEEE form.

---

**FRACTw. Format**

Converts numeric values to fractions.

- **Category:** Numeric
Syntax

FRACTw.

Arguments

w

specifies the width of the output field.

Default 10

Range 4–32

Details

Dividing the number 1 by 3 produces the value 0.33333333. To write this value as 1/3, use the FRACTw. format. FRACTw. writes fractions in reduced form, that is, 1/2 instead of 50/100.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(0.6666666667,fract8.);</td>
<td>2/3</td>
</tr>
<tr>
<td>a=put(0.2784,fract8.);</td>
<td>174/625</td>
</tr>
</tbody>
</table>

HEXw. Format

Converts real binary (floating-point) values to hexadecimal representation.

Category: Numeric

Alignment: Left
Range | 1–16
--- | ---
Tip If \( w < 16 \), the HEX \( w \). format converts real binary numbers to fixed-point integers before writing them as hexadecimal characters. It also writes negative numbers in two's complement notation, and right aligns digits. If \( w \) is 16, HEX \( w \). displays floating-point values in their hexadecimal form.

**Details**

In any operating environment, the least significant byte written by HEX \( w \). is the rightmost byte. Some operating environments store integers with the least significant digit as the first byte. The HEX \( w \). format produces consistent results in any operating environment regardless of the order of significance by byte.

*Note:* Different operating environments store floating-point values in different ways. However, the HEX16. format writes hexadecimal representations of floating-point values with consistent results in the same way that your operating environment stores them.

**Comparisons**

The HEX \( w \). numeric format and the $HEXw$. character format both generate the hexadecimal equivalent of values.

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>a=put(35.4, hex8.);</td>
<td>00000023</td>
</tr>
<tr>
<td>a=put(88, hex8.);</td>
<td>00000058</td>
</tr>
<tr>
<td>a=put(2.33, hex8.);</td>
<td>00000002</td>
</tr>
<tr>
<td>a=put(-150, hex8.);</td>
<td>FFFFFFF6A</td>
</tr>
</tbody>
</table>

**See Also**

**Formats:**

- “$HEXw$. Format” on page 237

---

**HHMMw.d Format**

*Writes SAS time values as hours and minutes in the form \( hh:mm \).*

**Category:** Date and Time

**Alignment:** Right
Syntax

HHMMw.[d]

Arguments

w
specifies the width of the output field.

Default 5
Range 2–20

d
specifies the number of digits to the right of the decimal point in the minutes value. The digits to the right of the decimal point specify a fraction of a minute.

Default 0
Range 0–19

Details

The HHMMw.d format writes SAS datetime values in the form hh:mm. Here is an explanation of the syntax:

hh
is an integer.

mm
is the number of minutes that range from 00 through 59.

SAS rounds hours and minutes that are based on the value of seconds in a SAS time value.

Comparisons

The HHMMw.d format is similar to the TIMEw.d format except that the HHMMw.d format does not print seconds.

The HHMMw.d format and the TIMEw.d format write a leading blank for the single-hour digit. The TODw.d format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 46796, which is the SAS time value that corresponds to 12:59:56 p.m..

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(46796,hhmm.);</td>
<td>13:00</td>
</tr>
<tr>
<td>a=put(46796,hhmm8.2);</td>
<td>12:59.93</td>
</tr>
</tbody>
</table>
In the first example, SAS rounds up the time value four seconds based on the value of seconds in the SAS time value. In the second example, by adding a decimal specification of 2 to the format shows that fifty-six seconds is 93% of a minute.

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “HOURw.d Format” on page 279
- “MMSSw.d Format” on page 288
- “TIMEw.d Format” on page 305
- “TODw.d Format” on page 308

Functions:

- “HMS Function” on page 528
- “HOUR Function” on page 532
- “MINUTE Function” on page 628
- “SECOND Function” on page 775
- “TIME Function” on page 802

HOURw.d Format

Writes SAS time values as hours and decimal fractions of hours.

Syntax

\[
\text{HOURw.d} \\
\]

Arguments

\(w\)

specifies the width of the output field.

Default 2

Range 2–20

\(d\)

specifies the number of digits to the right of the decimal point in the hour value. Therefore, SAS prints decimal fractions of the hour.
Range  
0-19

Requirement  
must be less than \( w \)

Details
SAS rounds hours based on the value of minutes in the SAS time value.

Example
The example table uses the input value of 41400, which is the SAS time value that corresponds to 11:30 AM.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = \text{put}(41400, \text{hour}4.1) );</td>
<td>11.5</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “HHMMw.d Format” on page 277
- “MMSSw.d Format” on page 288
- “TIMEw.d Format” on page 305
- “TODw.d Format” on page 308

Functions:
- “HMS Function” on page 528
- “HOUR Function” on page 532
- “MINUTE Function” on page 628
- “SECOND Function” on page 775
- “TIME Function” on page 802

IEEEw.d Format
Generates an IEEE floating-point value by multiplying a number by 10 raised to the \( d \)th power.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Left</td>
</tr>
<tr>
<td>CAUTION:</td>
<td>Large floating-point values and floating-point values that require precision might not be identical to the original SAS value when they are written to an IBM mainframe by using the IEEE format and read back into SAS using the IEEE informat.</td>
</tr>
</tbody>
</table>
Syntax

`IEEEw.[d]`

Arguments

\( w \)

specifies the width of the output field.

- Default: 8
- Range: 3–8
- Tip: If \( w \) is 8, an IEEE double-precision, floating-point number is written. If \( w \) is 5, 6, or 7, an IEEE double-precision, floating-point number is written, which assumes truncation of the appropriate number of bytes. If \( w \) is 4, an IEEE single-precision floating-point number is written. If \( w \) is 3, an IEEE single-precision, floating-point number is written, which assumes truncation of one byte.

\( d \)

specifies to multiply the number by \( 10^d \).

- Default: 0
- Range: 0–10

Details

This format is useful in operating environments where IEEE\( w.d \) is the floating-point representation that is used. In addition, you can use the IEEE\( w.d \) format to create files that are used by programs in operating environments that use the IEEE floating-point representation.

Typically, programs generate IEEE values in single-precision (4 bytes) or double-precision (8 bytes). Programs perform truncation solely to save space on output files. Machine instructions require that the floating-point number be one of the two lengths. The IEEE\( w.d \) format allows other lengths, which enables you to write data to files that contain space-saving truncated data.

Example

In the following example, you use the `VARBINARY` data type in the `DECLARE` statement to produce results that are hexadecimal representations of binary numbers stored in IEEE form.

```plaintext
data _null_;  
method run();  
   dcl varbinary(8) a;  
a=put(1,ieee4.);  
put a=;  
a=put(1,ieee5.);  
put a=;  
end;  
enddata;
```
run;

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| `a=put(1,ieee4.);`  | 3FF00000
| `a=put(1,ieee5.);`  | 3FF0000000

* The results contain a binary value.

**JULIANw. Format**

Writes SAS date values as Julian dates in the form `yyddd` or `yyyyddd`.

**Category:** Date and Time  
**Alignment:** Left

**Syntax**

`JULIANw.`

**Arguments**

`w`  
specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>Range</th>
<th>Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5–7</td>
<td>If <code>w</code> is 5, the JULIANw. format writes the date with a two-digit year. If <code>w</code> is 7, the JULIANw. format writes the date with a four-digit year.</td>
</tr>
</tbody>
</table>

**Details**

The JULIANw. format writes SAS date values in the form `yyddd` or `yyyyddd`. Here is an explanation of the syntax:

- `yy` or `yyyy`  
is a two-digit or four-digit integer that represents the year.

- `ddd`  
is the number of the day, 1–365 (or 1–366 for leap years), in that year.

**Example**

The example table uses the input value of 17524, which is the SAS date value that corresponds to December 24, 2007 (the 358th day of the year).
Statements | Results
--- | ---
a=put(17524, julian5.); | 07358
a=put(17524, julian7.); | 2007358

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:
- “DATEJUL Function” on page 466

**MDYAMP\(w.d\) Format**

Writes datetime values in the form `mm/dd/yyyy hh:mm AM|PM`. The year can be either two or four digits.

**Category:** Date and Time

**Alignment:** Right

**Interaction:**
When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in *SAS System Options: Reference*.

**Note:** The default time period is AM.

**Syntax**

MDYAMP\(w.d\).

**Arguments**

\(w\)

specifies the width of the output field.

Default: 19

Range: 8–40

**Details**

The MDYAMP\(w.d\) format writes SAS datetime values in the following form:

`mm/dd/yyyy[yy] hh:mm[AM | PM]`:

`mm`

is an integer between 1 and 12 that represents the month.

`dd`

is an integer between 1 and 31 that represents the day of the month.

`yy` or `yyyy`

specifies a two-digit or four-digit integer that represents the year.
hh

is an integer between 00 and 23 that represents hours.

mm

is an integer between 00 and 59 that represents minutes.

AM | PM

specifies either the time period 00:01−12:00 noon (PM) or the time period 12:01−12:00 midnight (AM). The default is AM.

date and time separator characters

is one of several special characters, such as the slash (/), colon (:), or a blank character that SAS uses to separate date and time components.

Comparisons

The MDYAMPMy. format writes datetime values with separators in the form mm/dd/yyyy hh:mm AM | PM, and requires a space between the date and the time.

The DATETIMEw.d format writes datetime values with separators in the form ddmmyy<yy>: hh:mm:ss.ss.

Example

This example uses the input value of 1694836380, or the SAS datetime value that corresponds to 3:53:00 PM on September 15, 2013.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>put dt mdyampm18.</td>
<td>9/15/2013 3:53 PM</td>
</tr>
</tbody>
</table>

See Also

Formats:

• “DATETIMEw.d Format” on page 256

MMDDYYw. Format

Writes SAS date values in the form mmdd<yy>yy or mm/dd<yy>yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.

Category: Date and Time
Alignment: Right

Syntax

MMDDYYw.

Arguments

w

specifies the width of the output field.
Default 8

Range 2–10

Interaction When \( w \) has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When \( w \) is 7, the date appears as a two-digit year without slashes.

Details

The MMDDYY\( w \) format writes SAS date values in the form \( mm<yy>yy \) or \( mm/dd/<yy>yy \). Here is an explanation of the syntax:

\( mm \)

is an integer that represents the month.

\( / \)

is the separator.

\( dd \)

is an integer that represents the day of the month.

\( <yy>yy \)

is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 18925, which is the SAS date value that corresponds to October 25, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18925,mmddyy2.);</td>
<td>10</td>
</tr>
<tr>
<td>a=put(18925,mmddyy3.);</td>
<td>10</td>
</tr>
<tr>
<td>a=put(18925,mmddyy4.);</td>
<td>1025</td>
</tr>
<tr>
<td>a=put(18925,mmddyy5.);</td>
<td>10/25</td>
</tr>
<tr>
<td>a=put(18925,mmddyy6.);</td>
<td>102511</td>
</tr>
<tr>
<td>a=put(18925,mmddyy7.);</td>
<td>102511</td>
</tr>
<tr>
<td>a=put(18925,mmddyy8.);</td>
<td>10/25/11</td>
</tr>
<tr>
<td>a=put(18925,mmddyy10.);</td>
<td>10/25/2011</td>
</tr>
</tbody>
</table>

See Also

• Chapter 14, “Dates and Times in DS2,” on page 111
**Formats:**
- “DATEw. Format” on page 253
- “DDMMYYw. Format” on page 258
- “MMDDYYxw. Format” on page 286
- “YYMMDDw. Format” on page 321

**Functions:**
- “DAY Function” on page 469
- “MDY Function” on page 624
- “MONTH Function” on page 635
- “YEAR Function” on page 845

---

**MMDDYYxw. Format**

Writes SAS date values in the form `mmdd<yy>yy` or `mm-dd<yy>yy`, where the `x` in the format name is a character that represents the special character, which separates the month, day, and year. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

**Category:** Date and Time

**Alignment:** Right

---

**Syntax**

`MMDDYYxw.`

**Arguments**

`x`

identifies a separator or specifies that no separator appear between the month, the day, and the year. Here are the valid values:

- B: separates with a blank
- C: separates with a colon
- D: separates with a hyphen
- N: indicates no separator
- P: separates with a period
- S: separates with a slash.

`w`

specifies the width of the output field.
Default  8
Range    2–10

Interactions
When $w$ has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When $w$ is 7, the date appears as a two-digit year without separators.

When $x$ has a value of N, the width range changes to 2–8.

Details
The MMDDYYxw. format writes SAS date values in the form $mmdd<yy>$ or $mmXddXXy$. Here is an explanation of the syntax:

$mm$
  is an integer that represents the month.

$X$
  is a specified separator.

$dd$
  is an integer that represents the day of the month.

$<yy>$
  is a two-digit or four-digit integer that represents the year.

Example
The following examples use the input value of 18922, which is the SAS date value that corresponds to October 22, 2011.

| Statements                  | Results  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18922,mmddyyyc5.);</td>
<td>10:22</td>
</tr>
<tr>
<td>a=put(18922,mmddyyd8.);</td>
<td>10-22-11</td>
</tr>
<tr>
<td>a=put(18922,mmddyyyy8.);</td>
<td>10222011</td>
</tr>
<tr>
<td>a=put(18922,mmddyyyyp10.);</td>
<td>10.22.2011</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “DATEw. Format” on page 253
- “DDMMYYxw. Format” on page 260
- “MMDDYYxw. Format” on page 286
- “YYMMDxDDw. Format” on page 323
MMSSw.d Format

Writes SAS time values as the number of minutes and seconds since midnight.

**Category:** Date and Time

**Alignment:** Right

**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in *SAS System Options: Reference*.

**Syntax**

\[
\text{MMSSw}.d
\]

**Arguments**

\(w\)

specifies the width of the output field.

- **Default:** 5
- **Range:** 2–20
- **Tip:** Set \(w\) to a minimum of 5 to write a value that represents minutes and seconds.

\(d\)

specifies the number of digits to the right of the decimal point in the seconds value. Therefore, the SAS time value includes fractional seconds.

- **Range:** 0–19
- **Restriction:** must be less than \(w\)

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(4530,mmss.);</td>
<td>75:30</td>
</tr>
</tbody>
</table>
See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “HHMMw.d Format” on page 277
- “TIMEw.d Format” on page 305

Functions:

- “HMS Function” on page 528
- “MINUTE Function” on page 628
- “SECOND Function” on page 775

**MMYYw. Format**

Writers SAS date values in the form $mmM<yy>yy$, where M is the separator and the year appears as either 2 or 4 digits.

**Category:** Date and Time

**Alignment:** Right

**Syntax**

`MMYYw.`

**Arguments**

$w$

specifies the width of the output field.

- **Default**: 7
- **Range**: 5–32
- **Interaction**: When $w$ has a value of 5 or 6, the date appears with only the last two digits of the year. When $w$ is 7 or more, the date appears with a four-digit year.

**Details**

The MMYYw. format writes SAS date values in the form $mmM<yy>yy$. Here is an explanation of the syntax:

$mm$

is an integer that represents the month.

$M$

is the character separator.

$<yy>yy$

is a two-digit or four-digit integer that represents the year.
Example

The following examples use the input value of 18925, which is the SAS date value that corresponds to October 25, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18925,mmyy5.);</td>
<td>10M11</td>
</tr>
<tr>
<td>a=put(18925,mmyy6.);</td>
<td>10M11</td>
</tr>
<tr>
<td>a=put(18925,mmyy.);</td>
<td>10M2011</td>
</tr>
<tr>
<td>a=put(18925,mmyy7.);</td>
<td>10M2011</td>
</tr>
<tr>
<td>a=put(18925,mmyy10.);</td>
<td>10M2011</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “MMYYXw. Format” on page 290
- “YYMMw. Format” on page 318

**MMYYXw. Format**

Writes SAS date values in the form \( mm<yy>yy \) or \( mm-<yy>yy \), where the \( x \) in the format name is a character that represents the special character that separates the month and the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

**Category:** Date and Time  
**Alignment:** Right

**Syntax**

**MMYYXw.**

**Arguments**

\( x \)

identifies a separator or specifies that no separator appear between the month and the year. Here are the valid values:

\( C \)

separates with a colon
D
separates with a hyphen
N
indicates no separator
P
separates with a period
S
separates with a forward slash.

\textit{w}

specifies the width of the output field.

| Default | 7 |
| Range   | 5–32 |

\textbf{Interactions}

When $x$ is set to N, no separator is specified. The width range is then 4–32, and the default changes to 6.

When $x$ has a value of C, D, P, or S and \textit{w} has a value of 5 or 6, the date appears with only the last two digits of the year. When \textit{w} is 7 or more, the date appears with a four-digit year.

When $x$ has a value of N and \textit{w} has a value of 4 or 5, the date appears with only the last two digits of the year. When \textit{x} has a value of \text{N} and \textit{w} is 6 or more, the date appears with a four-digit year.

\section*{Details}

The MMYY\textit{wx}. format writes SAS date values in the form $mm<\text{yy}yy$ or $mmX<\text{yy}yy$.

\begin{itemize}
  \item \textit{mm}
    \begin{itemize}
      \item is an integer that represents the month.
    \end{itemize}
  \item \textit{X}
    \begin{itemize}
      \item is a specified separator.
    \end{itemize}
  \item $<\text{yy}yy$
    \begin{itemize}
      \item is a two-digit or four-digit integer that represents the year.
    \end{itemize}
\end{itemize}

\section*{Example}

The following examples use the input value of 18822, which is the SAS date value that corresponds to July 14, 2011.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{Statements} & \textbf{Results} \\
\hline
\texttt{a=put(18822,mmyyc5.);} & 07:11 \\
\texttt{a=put(18822,mmyyd7.);} & 07-2011 \\
\texttt{a=put(18822,mmyyn4.);} & 0711 \\
\hline
\end{tabular}
\end{center}
Statements | Results
--- | ---
a=put(18822,mmyyp8.); | 07.2011
a=put(18822,mmyys10.); | 07/2011

See Also
• Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
• “MMYYw. Format” on page 289
• “YYMMw. Format” on page 318

**MONNAMEw. Format**

Writes SAS date values as the name of the month.

**Category:** Date and Time

**Alignment:** Right

**Syntax**

MONNAMEw.

**Arguments**

`w`

specifies the width of the output field.

**Default** 9

**Range** 1–32

**Tip** Use MONNAME3. to print the first three letters of the month name.

**Details**

If necessary, SAS truncates the name of the month to fit the format width.

**Example**

The example table uses the input value of 18691, which is the SAS date value that corresponds to March 5, 2011.
MONTHw. Format

Writes SAS date values as the month of the year.

Category: Date and Time
Alignment: Right

Syntax

MONTHw.

Arguments

w

specifies the width of the output field.

Default 2
Range 1–32

Details

The MONTHw. format writes the month (1 through 12) of the year from a SAS date value.

Example

The example table uses the input value of 18871, which is the SAS date value that corresponds to September 01, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18691,monname1.);</td>
<td>M</td>
</tr>
<tr>
<td>a=put(18691,monname3.);</td>
<td>Mar</td>
</tr>
<tr>
<td>a=put(18691,monname5.);</td>
<td>March</td>
</tr>
</tbody>
</table>

See Also

• Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

• “MONTHw. Format” on page 293
\[ a = \text{put}(18871, \text{month}.); \]

9

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “MONNAMEw. Format” on page 292

MONYYw. Format

Writes SAS date values as the month and the year in the form \textit{mmyy} or \textit{mmyyyy}.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Right</td>
</tr>
</tbody>
</table>

Syntax

\texttt{MONYYw}.

Arguments

\texttt{w}

specifies the width of the output field.

Default: 5

Range: 5–7

Details

The MONYYw. format writes SAS date values in the form \textit{mmyy} or \textit{mmyyyy}. Here is an explanation of the syntax:

\texttt{mmm}

is the first three letters of the month name.

\texttt{yy} or \texttt{yyyy}

is a two-digit or four-digit integer that represents the year.

Comparisons

The MONYYw. format and the DTMONYYw. format are similar in that they both write date values. The difference is that MONYYw. expects a SAS date value as input, and DTMONYYw. expects a datetime value.

Example

The example table uses the input value of 18985, which is the SAS date value that corresponds to December 24, 2011.
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----1</td>
<td>DEC11</td>
</tr>
<tr>
<td>a=put(18985,monyy5.);</td>
<td>DEC11</td>
</tr>
<tr>
<td>a=put(18985,monyy7.);</td>
<td>DEC2011</td>
</tr>
</tbody>
</table>

### See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

### Formats:

- “DTMONYYw. Format” on page 266
- “DDMMYYw. Format” on page 258
- “MMDDYYw. Format” on page 284
- “YYMMDDw. Format” on page 321

### Functions:

- “MONTH Function” on page 635
- “YEAR Function” on page 845

---

### NEGPARENw.d Format

Writes negative numeric values in parentheses.

**Category:** Numeric  
**Alignment:** Right  
**Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

#### Syntax

`NEGPARENw.[d]`

**Arguments**

- `w`
  - specifies the width of the output field.
  - Default: 6
  - Range: 1–32

- `d`
  - specifies the number of digits to the right of the decimal point in the numeric value.
Details

The NEGPARENw.d format attempts to right align output values. If the input value is negative, NEGPARENw.d displays the output by enclosing the value in parentheses, if the field that you specify is wide enough. Otherwise, it uses a minus sign to represent the negative value. If the input value is nonnegative, NEGPARENw.d displays the value with a leading and trailing blank to ensure proper column alignment. It reserves the last column for a close parenthesis even when the value is positive.

Comparisons

The NEGPARENw.d format is similar to the COMMAw.d format in that it separates every three digits of the value with a comma.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(100,negparen6.);</td>
<td>100</td>
</tr>
<tr>
<td>a=put(1000,negparen6.);</td>
<td>1,000</td>
</tr>
<tr>
<td>a=put(-200,negparen6.);</td>
<td>(200)</td>
</tr>
<tr>
<td>a=put(-2000,negparen6.);</td>
<td>(2,000)</td>
</tr>
</tbody>
</table>

NENGOw. Format

Writes SAS date values as Japanese dates in the form e.yymmdd.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Left</td>
</tr>
</tbody>
</table>

Syntax

NENGOw.

Arguments

w

specifies the width of the output field.

Default 10
Details

The NENGOw. format writes SAS date values in the form e.yymmdd. Here is an explanation of the syntax:

- **e**
  - is the first letter of the name of the emperor (Meiji, Taisho, Showa, or Heisei).

- **yy**
  - is an integer that represents the year.

- **mm**
  - is an integer that represents the month.

- **dd**
  - is an integer that represents the day of the month.

If the width is too small, SAS omits the period.

Example

The example table uses the input value of 18702, which is the SAS date value that corresponds to March 16, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18702,nengo3.);</td>
<td>H23</td>
</tr>
<tr>
<td>a=put(18702,nengo6.);</td>
<td>H23/03</td>
</tr>
<tr>
<td>a=put(18702,nengo8.);</td>
<td>H.230316</td>
</tr>
<tr>
<td>a=put(18702,nengo9.);</td>
<td>H23/03/16</td>
</tr>
<tr>
<td>a=put(18702,nengo10.);</td>
<td>H.23/03/16</td>
</tr>
</tbody>
</table>

OCTALw. Format

Converts numeric values to octal representation.

- **Category:** Numeric
- **Alignment:** Left

Syntax

OCTALw.
Arguments

w
 specifies the width of the output field.
Default 3
Range 1–24

Details
If necessary, the OCTALw. format converts numeric values to integers before displaying them in octal representation.

Comparisons
OCTALw. converts numeric values to octal representation. The $OCTALw.$ format converts character values to octal representation.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(3592, octal6.);</td>
<td>007010</td>
</tr>
</tbody>
</table>

See Also

Formats:
• “$OCTALw.$ Format” on page 238

PERCENTw.d Format

Writes numeric values as percentages.

Category: Numeric
Alignment: Right
Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

PERCENTw.[d]
**Arguments**

\(w\)  
specifies the width of the output field.  

Default 6  
Range 4–32  

\(d\)  
specifies the number of digits to the right of the decimal point in the numeric value.  

Range 0–31  
Requirement must be less than \(w\)

**Details**

The PERCENT\(w.d\) format multiplies values by 100, formats them the same as the BEST\(w.d\) format, adds a percent sign (%) to the end of the formatted value, and encloses negative values in parentheses. The PERCENT\(w.d\) format allows room for a percent sign and parentheses, even if the value is not negative.

**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(0.1,percent10.);</td>
<td>10%</td>
</tr>
<tr>
<td>a=put(1.2,percent10.);</td>
<td>120%</td>
</tr>
<tr>
<td>a=put(-.05,percent10.);</td>
<td>( 5%)</td>
</tr>
</tbody>
</table>

**See Also**

Formats:  
- “PERCENT\(w.d\) Format” on page 299

---

**PERCENT\(w.d\) Format**

Produces percentages, using a minus sign for negative values.  

- **Category:** Numeric  
- **Alignment:** Right  
- **Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.
Syntax
PERCENTN$w$.[$d$]

Arguments

$w$
specifies the width of the output field.

Default 6
Range 4–32

d
specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31
Requirement must be less than $w$

Details
The PERCENTN$w$.[$d$] format multiplies negative values by 100, formats them the same as the BEST$w$.[$d$] format, adds a minus sign to the beginning of the value, and adds a percent sign (%) to the end of the formatted value. The PERCENTN$w$.[$d$] format allows room for a percent sign and a minus sign, even if the value is not negative.

Comparisons
The PERCENTN$w$.[$d$] format produces percents by using a minus sign instead of parentheses for negative values. The PERCENT$w$.[$d$] format produces percents by using parentheses for negative values.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(-0.1,percentn.);</td>
<td>-10%</td>
</tr>
<tr>
<td>a=put(.2,percentn.);</td>
<td>20%</td>
</tr>
<tr>
<td>a=put(.8,percentn.);</td>
<td>80%</td>
</tr>
<tr>
<td>a=put(-0.05,percentn.);</td>
<td>-5%</td>
</tr>
<tr>
<td>a=put(-6.3,percentn.);</td>
<td>-630%</td>
</tr>
</tbody>
</table>
See Also

Formats:

- “PERCENTw.d Format” on page 298

QTRw. Format

Writes SAS date values as the quarter of the year.

Category: Date and Time

Alignment: Right

Syntax

QTRw.

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32

Example

The example table uses the input value of 18691, which is the SAS date value that corresponds to March 5, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18691,qtr.);</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “QTRRw. Format” on page 301

QTRRw. Format

Writes SAS date values as the quarter of the year in Roman numerals.

Category: Date and Time
Syntax
QTRR<sup>w</sup>.

Arguments
<sup>w</sup>

specifies the width of the output field.
Default 3
Range 3–32

Example
The example table uses the input value of 18885, which is the SAS date value that corresponds to September 15, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18885,qtrr.);</td>
<td>III</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:
- “QTR<sup>w</sup>. Format” on page 301
Details

The ROMANw. format truncates a floating-point value to its integer component before the value is written.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(2006,roman.);</td>
<td>MMVI</td>
</tr>
</tbody>
</table>

SIZEKw.d Format

Writes a numeric value in the form nK for kilobytes.

- **Category:** Numeric
- **Alignment:** Right

Syntax

`SIZEKw.[d]`

Arguments

- **w**
  - Specifies the width of the output field.
  - Default: 9
  - Range: 2–33

- **d**
  - Specifies the number of digits to the right of the decimal point in the numeric value.
  - Default: 0
  - Range: 0–31

Details

To write a numeric value in the form nK by using the SIZEKw.d format, the value of n is calculated by dividing the numeric value by 1,024. The symbol K indicates that the value is a multiple of 1,024.
Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1024, sizek.);</td>
<td>1K</td>
</tr>
<tr>
<td>a=put(200943, sizek.);</td>
<td>197K</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “SIZEKMGw.d Format” on page 304

SIZEKMGw.d Format

Writes a numeric value in the form nKB for kilobytes, nMB for megabytes, or nGB for gigabytes.

Category: Numeric
Alignment: Right

Syntax

SIZEKMGw.[d]

Arguments

w
specifies the width of the output field.

Default 9
Range 2–33

d
specifies the number of digits to the right of the decimal point in the numeric value.

Default 0
Range 0–31

Details

When you specify the SIZEKMGw.d format, SAS determines the best suffix: KB for kilobytes; MB for megabytes; or GB for gigabytes; and divides the SAS numeric value by one of the following values:

KB
1024
Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----1</td>
<td></td>
</tr>
<tr>
<td>a=put(3688, sizekmg.);</td>
<td>4KB</td>
</tr>
<tr>
<td>a=put(1048576, sizekmg.);</td>
<td>1MB</td>
</tr>
<tr>
<td>a=put(83409922345, sizekmg.);</td>
<td>80MB</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “SIZEKw.d Format” on page 303

TIMEw.d Format

Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss.

<table>
<thead>
<tr>
<th>Category: Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment: Right</td>
</tr>
<tr>
<td>Interaction:</td>
</tr>
<tr>
<td>When the DECIMALCONV= system option is set to STDIEEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference</td>
</tr>
</tbody>
</table>

Syntax

TIMEw.[d]

Arguments

w

specifies the width of the output field.

Default 8

Range 2–20

Tip Make w large enough to produce the desired results. To obtain a complete time value with three decimal places, you must allow at least 12 spaces:
Eight spaces to the left of the decimal point, one space for the decimal point itself, and three spaces for the decimal fraction of seconds.

\[ d \]

specifies the number of digits to the right of the decimal point in the seconds value.

- **Default**: 0
- **Range**: 0–19
- **Requirement**: must be less than \( w \)

**Details**

The TIME\( w.d \) format writes SAS time values in the form \( hh:mm:ss.ss \). Here is an explanation of the syntax:

- \( hh \)
  - is an integer.
  - *Note*: If \( hh \) is a single digit, TIME\( w.d \) places a leading blank before the digit. For example, the TIME\( w.d \) format writes 9:00 instead of 09:00.

- \( mm \)
  - is the number of minutes, ranging from 00 through 59.

- \( ss.ss \)
  - is the number of seconds, ranging from 00 through 59, with the fraction of a second following the decimal point.

**Comparisons**

The TIME\( w.d \) format is similar to the HHMM\( w.d \) format except that TIME\( w.d \) includes seconds.

The TIME\( w.d \) format and the HHMM\( w \)write a leading blank for a single-hour digit. The TOD\( w.d \) format writes a leading zero for a single-hour digit.

**Example**

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=(59083,time.);</td>
<td>16:24:43</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**

- “HHMM\( w.d \) Format” on page 277
- “HOUR\( w.d \) Format” on page 279
TIMEAMPMw.d Format

 Writes SAS time values as hours, minutes, and seconds in the form \( hh:mm:ss.ss \) with AM or PM.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Right</td>
</tr>
<tr>
<td>Interaction:</td>
<td></td>
</tr>
</tbody>
</table>

When the DECIMALCONV= system option is set to STDIEEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

Syntax

\[
\text{TIMEAMPMw.}[d]
\]

Arguments

\( w \)

specifies the width of the output field.

- Default: 11
- Range: 2–20

\( d \)

specifies the number of digits to the right of the decimal point in the seconds value.

- Default: 0
- Range: 0–19
- Requirement: must be less than \( w \)

Details

The TIMEAMPMw.d format writes SAS time values in the form \( hh:mm:ss.ss \) with AM or PM. Here is an explanation of the syntax:

\( hh \)

is an integer that represents the hour.
$mm$

is an integer that represents the minutes.

$ss.ss$

is the number of seconds to two decimal places.

Times greater than 23:59:59 PM appear as the next day.

Make $w$ large enough to produce the desired results. To obtain a complete time value with three decimal places and AM or PM, you must allow at least 11 spaces ($hh:mm:ss$ PM). If $w$ is less than 5, SAS writes AM or PM only.

**Comparisons**

- The TIMEAMPMM$w$.d format is similar to the TIMEM$w$.d format except, that TIMEAMPMM$w$.d prints AM or PM at the end of the time.
- TIME$w$.d writes hours greater than 23:59:59 PM, and TIMEAMP$w$.d does not.

**Example**

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----+----1----+</td>
</tr>
<tr>
<td></td>
<td>a=put(59083,timeampm3.);</td>
</tr>
<tr>
<td></td>
<td>a=put(59083,timeampm5.);</td>
</tr>
<tr>
<td></td>
<td>a=put(59083,timeampm7.);</td>
</tr>
<tr>
<td></td>
<td>a=put(59083,timeampm11.);</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**

- “TIMEf.d Format” on page 305

---

**TODw.d Format**

Writes SAS time values and the time portion of SAS datetime values in the form $hh:mm:ss.ss$.

**Category:** Date and Time

**Alignment:** Right

**Interaction:** When the DECIMALCONV= system option is set to STDOUTEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.
Syntax

TODw.d

Arguments

w
specifies the width of the output field.

Default 8
Range 2–20
Tip SAS writes a zero for a zero hour if the specified width is sufficient (for example, 02:30 or 00:30).

d
specifies the number of digits to the right of the decimal point in the seconds value.

Default 0
Range 0–19
Requirement must be less than w

Details

The TODw.d format writes SAS datetime values in the form hh:mm:ss.ss. Here is an explanation of the syntax:

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The TODw.d format writes a leading zero for a single-hour digit. The TIMEw.d format and the HHMMw.d format write a leading blank for a single-hour digit.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1472049623,tod9.);</td>
<td>14:40:23</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111
Formats:
- "TIMEw.d Format" on page 305
- "TIMEAMPMw.d Format" on page 307

Functions:
- "TIMEPART Function" on page 803

VAXRBw.d Format
Writes real binary (floating-point) data in VMS format.

Category: Numeric
Alignment: Right

Syntax
VAXRBw.[d]

Arguments

\( w \)
specifies the width of the output field.

Default 8
Range 2–8

\( d \)
specifies the power of 10 by which to divide the value.

Default 0
Range 0–31

Details
Use the VAXRBw.d format to write data in native VAX/VMS floating-point notation.

Example
In the following example, you use the VARBINARY data type so that the result is the hexadecimal representation for the integer.

data _null_;  
  method run();  
    dcl varbinary(20) a;  
    a=put(1,vaxrb8.);  
    put a=;  
  end;  
enddata;  
run;
<table>
<thead>
<tr>
<th>Statements</th>
<th>Results *</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1,vaxrb8.);</td>
<td>8040000000000000</td>
</tr>
</tbody>
</table>

* The result is the hexadecimal representation for the integer.

**w.d Format**

Writes standard numeric data one digit per byte.

- **Category:** Numeric
- **Alignment:** Right
- **Alias:** Fw.d
- **Interaction:** When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.

**Syntax**

\[ w.[d] \]

**Arguments**

- **w**
  - Specifies the width of the output field.
  - **Range:** 1–32
  - **Tip:** Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

- **d**
  - Specifies the number of digits to the right of the decimal point in the numeric value.
  - **Range:** 0–31
  - **Requirement:** must be less than \( w \)
  - **Tip:** If \( d \) is 0 or you omit \( d \), \( w.d \) writes the value without a decimal point.

**Details**

The \( w.d \) format rounds to the nearest number that fits in the output field. If \( w.d \) is too small, SAS might shift the decimal to the BEST\( w \) format. The \( w.d \) format writes negative numbers with leading minus signs. In addition, \( w.d \) right aligns before writing and pads the output with leading blanks.
Comparisons

The Zw.d format is similar to the w.d format except that Zw.d pads right-aligned output with 0s instead of blanks.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>put (23.45, 6.3);</td>
<td>23.450</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “Zw.d Format” on page 333

WEEKDATEw. Format

Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).

Category: Date and Time
Alignment: Right

Syntax

WEEKDATEw.

Arguments

w

specifies the width of the output field.

Default 29
Range 3–37

Details

The WEEKDATEw. format writes SAS date values in the form day-of-week, month-name dd, yy (or yyyy). Here is an explanation of the syntax:

dd

is an integer that represents the day of the month.

yy or yyyy

is a two-digit or four-digit integer that represents the year.
If \( w \) is too small to write the complete day of the week and month, SAS abbreviates as needed.

**Comparisons**

The `WEEKDATEw.` format is the same as the `WEEKDATXw.` format except that `WEEKDATXw.` prints \( dd \) before the month's name.

**Example**

The example table uses the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18792,weekdate3.);</td>
<td>Tue</td>
</tr>
<tr>
<td>a=put(18792,weekdate9.);</td>
<td>Tuesday</td>
</tr>
<tr>
<td>a=put(18792,weekdate15.);</td>
<td>Tue, Jun 14, 11</td>
</tr>
<tr>
<td>a=put(18792,weekdate17.);</td>
<td>Tue, Jun 14, 2011</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**

- “DTWKDATXw. Format” on page 268
- “DATEw. Format” on page 253
- “DDMMYYw. Format” on page 258
- “MMDDYYw. Format” on page 284
- “TODw.d Format” on page 308
- “WEEKDATXw. Format” on page 314
- “YYMMDDw. Format” on page 321

**Functions:**

- “JULDATE Function” on page 586
- “MDY Function” on page 624
- “WEEKDAY Function” on page 842
WEEKDATXw. Format

Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).

**Category:** Date and Time  
**Alignment:** Right

### Syntax

WEEKDATXw.

### Arguments

w

Specifies the width of the output field.

**Default:** 29  
**Range:** 3–37

### Details

The WEEKDATXw. format writes SAS date values in the form day-of-week, dd month-name, yy (or yyyy). Here is an explanation of the syntax:

- **dd** is an integer that represents the day of the month.
- **yy** or **yyyy** is a two-digit or a four-digit integer that represents the year.

If w is too small to write the complete day of the week and month, then SAS abbreviates as needed.

### Comparisons

The WEEKDATEw. format is the same as the WEEKDATXw. format, except that WEEKDATEw. prints dd after the month’s name.

The DTWKDATXw. format is the same as the WEEKDATXw. format, except that DTWKDATXw. expects a datetime value as input.

### Example

The example table uses the input value of 18681, which is the SAS date value that corresponds to February 23, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18681,weekdatx.);</td>
<td>Wednesday, 23 February 2011</td>
</tr>
</tbody>
</table>
See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “DTWKDATXw. Format” on page 268
- “DATEw. Format” on page 253
- “DDMMYYw. Format” on page 258
- “MMDDYYw. Format” on page 284
- “TODw.d Format” on page 308
- “WEEKDATEw. Format” on page 312
- “YYMMDDw. Format” on page 321

Functions:

- “JULDATE Function” on page 586
- “MDY Function” on page 624
- “WEEKDAY Function” on page 842

WEEKDAYw. Format

Writes SAS date values as the day of the week.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment:</td>
<td>Right</td>
</tr>
</tbody>
</table>

Syntax

WEEKDAYw.

Arguments

\( w \)

specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>1–32</td>
</tr>
</tbody>
</table>

Details

The WEEKDAYw. format writes a SAS date value as the day of the week (where 1=Sunday, 2=Monday, and so on).

Example

The example table uses the input value of 18681, which is the SAS date value that corresponds to February 23, 2011.
## YEARw. Format

Wrintes SAS date values as the year.

**Category:** Date and Time  
**Alignment:** Right

### Syntax

```plaintext
YEARw.
```

### Arguments

- **w**  
  Specifies the width of the output field.

  **Default:** 4  
  **Range:** 2–32

### Tip

If `w` is less than 4, the last two digits of the year print. Otherwise, the year value prints as four digits.

### Comparisons

The `YEARw.` format is similar to the `DTYEARw.` format in that they both write date values. The difference is that `YEARw.` expects a SAS date value as input, and `DTYEARw.` expects a SAS datetime value.

### Example

The example table uses the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.
<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=\text{put}(18792,\text{year}2.); )</td>
<td>11</td>
</tr>
<tr>
<td>( a=\text{put}(18792,\text{year}4.); )</td>
<td>2011</td>
</tr>
</tbody>
</table>

### See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

### Formats:
- “DTYEARw. Format” on page 269

---

**YENw.d Format**

Writes numeric values with yen signs, commas, and decimal points.

- **Category:** Numeric
- **Alignment:** Right

---

**Syntax**

\[ \text{YENw.d} \]

**Arguments**

- **w**
  - Specifies the width of the output field.
  - **Default:** 1
  - **Range:** 1–32

- **d**
  - Specifies the number of digits to the right of the decimal point in the numeric value.
  - **Restriction:** must be either 0 or 2
  - **Tip:** If \( d \) is 2, then YENw.d writes a decimal point and two decimal digits. If \( d \) is 0, then YENw.d does not write a decimal point or decimal digits.

---

**Details**

The YENw.d format writes numeric values with a leading yen sign and with a comma that separates every three digits of each value.

The hexadecimal representation of the code for the yen sign character is 5B on EBCDIC systems and 5C on ASCII systems. The monetary character these codes represent might be different in other countries.
**Example**

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1254.71, yen10.2);</td>
<td>¥1,254.71</td>
</tr>
</tbody>
</table>

**YYMMw. Format**

Writes SAS date values in the form `<yy>yyMmm`, where M is a character separator to indicate that the month number follows the M and the year appears as either 2 or 4 digits.

- **Category:** Date and Time
- **Alignment:** Right

**Syntax**

YYMMw.

**Arguments**

w

specifies the width of the output field.

- **Default** 7
- **Range** 5–32
- **Interaction** When w has a value of 5 or 6, the date appears with only the last two digits of the year. When w is 7 or more, the date appears with a four-digit year.

**Details**

The YYMMw. format writes SAS date values in the form `<yy>yyMmm`. Here is an explanation of the syntax:

- `<yy>yy` is a two-digit or four-digit integer that represents the year.
- M is the character separator.
- `mm` is an integer that represents the month.

**Comparisons**

- The YYMMw.d format is similar to the YYMMxw.d format, except the YYMMxw.d format contains a separator such as a hyphen, colon, slash, or period between the year and month.
Example

The following examples use the input value of 18925, which is the SAS date value that corresponds to October 25, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18925,yymm5.);</td>
<td>11M10</td>
</tr>
<tr>
<td>a=put(18925,yymm6.);</td>
<td>11M10</td>
</tr>
<tr>
<td>a=put(18925,yymm.);</td>
<td>2011M10</td>
</tr>
<tr>
<td>a=put(18925,yymm7.);</td>
<td>2011M10</td>
</tr>
<tr>
<td>a=put(18925,yymm10.);</td>
<td>2011M10</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “MMYYw. Format” on page 289
- “YYMMxw. Format” on page 319

YYMMxw. Format

Writes SAS date values in the form <yy>yy-mm or <yy>yy-mm, where the x in the format name is a character that represents the special character that separates the year and the month, which can be a hyphen (−), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Category: Date and Time
Alignment: Right

Syntax

YYMMxw.

Arguments

x

identifies a separator or specifies that no separator appear between the year and the month. Here are the valid values:

C

separates with a colon
D separates with a hyphen

N indicates no separator

P separates with a period

S separates with a forward slash

\( w \)
specifies the width of the output field.

| Default | 7 |
| Range   | 5–32 |

**Interactions**

When \( x \) is set to \( N \), no separator is specified. The width range is then 4–32, and the default changes to 6.

When \( x \) has a value of \( C, D, P, \) or \( S \) and \( w \) has a value of 5 or 6, the date appears with only the last two digits of the year. When \( w \) is 7 or more, the date appears with a four-digit year.

When \( x \) has a value of \( N \) and \( w \) has a value of 4 or 5, the date appears with only the last two digits of the year. When \( x \) has a value of \( N \) and \( w \) is 6 or more, the date appears with a four-digit year.

**Details**

The \( YYMMxw. \) format writes SAS date values in the form \(<yy>yy\)mm or \(<yy>yyXmm.\) Here is an explanation of the syntax:

\(<yy>yy\)
- is a two-digit or four-digit integer that represents the year.

\(X\)
- is a specified separator.

\(mm\)
- is an integer that represents the month.

**Comparisons**

- The YYMM\(xw.d \) format is similar to the YYMM\(xw.d \) format, except the YYMM\(xw.d \) format contains a separator such as a hyphen, colon, slash, or period, between the year and month.

**Example**

The following examples use the input value of 18822, which is the SAS date value that corresponds to July 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| ----+----1----+
### Statements

<table>
<thead>
<tr>
<th>a=put(18822,,yymmc5.);</th>
<th>11:07</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18822,,yymmd.);</td>
<td>2011-07</td>
</tr>
<tr>
<td>a=put(18822,,yymmn4.);</td>
<td>1107</td>
</tr>
<tr>
<td>a=put(18822,,yymmp8.);</td>
<td>2011.07</td>
</tr>
<tr>
<td>a=put(18822,,yymms10.);</td>
<td>2011/07</td>
</tr>
</tbody>
</table>

### See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

### Formats:

- “MMYYxw. Format” on page 290
- “YYMMw. Format” on page 318

### YYMMDw. Format

Writes SAS date values in the form `<yy>ymmd` or `<yy>yy–mm–dd`, where a hyphen is the separator and the year appears as either 2 or 4 digits.

**Category:** Date and Time  
**Alignment:** Right

### Syntax

`YYMMDw.`

### Arguments

- `w` specifies the width of the output field.
  - **Default:** 8  
  - **Range:** 2–10  
  - **Interaction:** When `w` has a value of from 2 to 5, the date appears with as much of the year and the month as possible. When `w` is 7, the date appears as a two-digit year without hyphens.

### Details

The `YYMMDw.` format writes SAS date values in the form `<yy>ymmd` or `<yy>yy–mm–dd`. Here is an explanation of the syntax:
<yy>yy
is a two-digit or four-digit integer that represents the year.

is the separator.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

Comparisons

• The YYMMDDw.d format is similar to the YYMMDDxw.d format, except the YYMMDDxw.d format contains separators, such as a colon, slash, or period between the year, month, and day.

Example

The following examples use the input value of 18720, which is the SAS date value that corresponds to April 3, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18720,yymmdd2.);</td>
<td>11</td>
</tr>
<tr>
<td>a=put(18720,yymmdd3.);</td>
<td>11</td>
</tr>
<tr>
<td>a=put(18720,yymmdd4.);</td>
<td>1104</td>
</tr>
<tr>
<td>a=put(18720,yymmdd5.);</td>
<td>11-04</td>
</tr>
<tr>
<td>a=put(18720,yymmdd6.);</td>
<td>110403</td>
</tr>
<tr>
<td>a=put(18720,yymmdd7.);</td>
<td>110403</td>
</tr>
<tr>
<td>a=put(18720,yymmdd8.);</td>
<td>11-04-03</td>
</tr>
<tr>
<td>a=put(18720,yymmdd10.);</td>
<td>2011-04-03</td>
</tr>
</tbody>
</table>

See Also

• Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

• “DATEw. Format” on page 253
• “DDMMYYw. Format” on page 258
• “MMDDYYw. Format” on page 284
• “YYMMDDxw. Format” on page 323
Functions:
- “DAY Function” on page 469
- “MDY Function” on page 624
- “MONTH Function” on page 635
- “YEAR Function” on page 845

YYMMDxw. Format

Writes SAS date values in the form <yy>yyymmdd or <yy>yy-mm-dd, where the x in the format name is a character that represents the special character that separates the year, month, and day. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Category: Date and Time
Alignment: Right

Syntax

YYMMDxw.

Arguments

x
identifies a separator or specifies that no separator appear between the year, the month, and the day. Here are the valid values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Separates with a blank</td>
</tr>
<tr>
<td>C</td>
<td>Separates with a colon</td>
</tr>
<tr>
<td>D</td>
<td>Separates with a hyphen</td>
</tr>
<tr>
<td>N</td>
<td>Indicates no separator</td>
</tr>
<tr>
<td>P</td>
<td>Separates with a period</td>
</tr>
<tr>
<td>S</td>
<td>Separates with a slash</td>
</tr>
</tbody>
</table>

w
specifies the width of the output field.

Default 8
Range 2–10

Interactions When w has a value of from 2 to 5, the date appears with as much of the year and the month. When w is 7, the date appears as a two-digit year without separators.
When \( x \) has a value of N, the width range is 2–8.

**Details**

The YYMMDD\( x \). format writes SAS date values in the form \(<y\>yymmdd\) or \(<y\>y\>XmmXdd\). Here is an explanation of the syntax:

\(<y\>y\> is a two-digit or four-digit integer that represents the year.

\(X\) is a specified separator.

\(mm\) is an integer that represents the month.

\(dd\) is an integer that represents the day of the month.

**Comparisons**

- The YYMMDD\( w,d \) format is similar to the YYMMDD\( xw,d \) format, but YYMMDD\( xw,d \) format contains a separator between the year and month, such as a colon, slash, or period.

**Example**

The following examples use the input value of 18922, which is the SAS date value that corresponds to October 22, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>+---------------------------------+-----------</td>
<td></td>
</tr>
<tr>
<td>a=put(18922,yymmddc5.);</td>
<td>11:10</td>
</tr>
<tr>
<td>a=put(18922,yymmddd8.);</td>
<td>11-10-22</td>
</tr>
<tr>
<td>a=put(18922,yymmddn8.);</td>
<td>20111022</td>
</tr>
<tr>
<td>a=put(18922,yymmddp10.);</td>
<td>2011.10.22</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**

- “DATE\( w \). Format” on page 253
- “DDMMYY\( x \). Format” on page 260
- “MMDDYY\( x \). Format” on page 286
- “YYMMDD\( w \). Format” on page 321

**Functions:**
YYMONw. Format

Writes SAS date values in the form yymm or yyyyymm.

- **Category:** Date and Time
- **Alignment:** Right

**Syntax**

YYMONw.

**Arguments**

w

Specifies the width of the output field. If the format width is too small to print a four-digit year, only the last two digits of the year are printed.

- **Default:** 7
- **Range:** 5–32

**Details**

The YYMONw. format abbreviates the month's name to three characters.

**Example**

The example table uses the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18792, yymon6.);</td>
<td>11JUN</td>
</tr>
<tr>
<td>a=put(18792, yyymon7.);</td>
<td>2011JUN</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**

- “MMYYw. Format” on page 289
YYQw. Format

Writes SAS date values in the form <yy>yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.

**Category:** Date and Time  
**Alignment:** Right

### Syntax

YYQw.

### Arguments

w

specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>Range</th>
<th>Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4–32</td>
<td>When w has a value of 4 or 5, the date appears with only the last two digits of the year. When w is 6 or more, the date appears with a four-digit year.</td>
</tr>
</tbody>
</table>

### Details

The YYQw. format writes SAS date values in the form <yy>yyQq. Here is an explanation of the syntax:

<yy>yy

is a two-digit or four-digit integer that represents the year.

Q

is the character separator.

q

is an integer (1, 2, 3, or 4) that represents the quarter of the year.

### Comparisons

The YYQw. format is similar to the YYQxw. format, but the YYQxw. format has separators between the YY and Q, such as a hyphen, slash, period, or colon.

### Example

The following examples use the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1979Q1</td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
a=put(18792,yyq4.); | 11Q2
a=put(18792,yyq5.); | 11Q2
a=put(18792,yyq6.); | 2011Q2
a=put(18792,yyq10.); | 2011Q2

**See Also**
- Chapter 14, “Dates and Times in DS2,” on page 111

**Formats:**
- “YYQxw. Format” on page 327
- “YYQRw. Format” on page 329
- “YYQRxw. Format” on page 330
- “YYQZw. Format” on page 332

---

**YYQxw. Format**

Writes SAS date values in the form `<yy>yyq` or `<yy>yy-q`, where the `x` in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

**Category:** Date and Time

**Alignment:** Right

**Syntax**

`YYQxw`. 

**Arguments**

`x`

identifies a separator or specifies that no separator appear between the year and the quarter. Here are the valid values:

- `C` separates with a colon
- `D` separates with a hyphen
- `N` indicates no separator
P
  separates with a period

S
  separates with a forward slash.

w
  specifies the width of the output field.

Default  6
Range    4–32

Interactions
When $x$ is set to $N$, no separator is specified. The width range is then 3–32, and the default changes to 5.

When $w$ has a value of 4 or 5, the date appears with only the last two digits of the year. When $w$ is 6 or more, the date appears with a four-digit year.

When $x$ has a value of $N$ and $w$ has a value of 3 or 4, the date appears with only the last two digits of the year. When $x$ has a value of $N$ and $w$ is 5 or more, the date appears with a four-digit year.

Details
The YYQxw. format writes SAS date values in the form $<yy>yyq$ or $<yy>yyXq$. Here is an explanation of the syntax:

$<yy>yy$
  is a two-digit or four-digit integer that represents the year.

X
  is a specified separator.

$q$
  is an integer (1,2,3, or 4) that represents the quarter of the year.

Comparisons
The YYQw. format is similar to the YYQxw. format, but the YYQxw. format has separators between the YY and Q, such as a hyphen, slash, period, or colon.

Example
The following examples use the input value of 18822, which is the SAS date value that corresponds to July 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18822,yyqc4.);</td>
<td>11:3</td>
</tr>
<tr>
<td>a=put(18822,yyqd.);</td>
<td>2011-3</td>
</tr>
<tr>
<td>a=put(18822,yyqn3.);</td>
<td>113</td>
</tr>
</tbody>
</table>
YYQRw. Format

Wants SAS date values in the form `<yy>yyQqr`, where `Q` is the separator, the year appears as either 2 or 4 digits, and `qr` is the quarter of the year expressed in roman numerals.

**Category:** Date and Time  
**Alignment:** Right

## Syntax

YYQRw.

## Arguments

`w`  
specifies the width of the output field.

<table>
<thead>
<tr>
<th>Default</th>
<th>Range</th>
<th>Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6–32</td>
<td>When the value of <code>w</code> is too small to write a four-digit year, the date appears with only the last two digits of the year.</td>
</tr>
</tbody>
</table>

## Details

The YYQRw. format writes SAS date values in the form `<yy>yyQqr`. Here is an explanation of the syntax:

- `<yy>yy`  
  is a two-digit or four-digit integer that represents the year.
- `Q`  
  is the character separator.
is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Comparisons

The YYQRw. format is similar to the YYQRxw. format, but the YYQRxw. format has separators between the YY and QR, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 18792, which is the SAS date value that corresponds to June 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(18792,yyqr6.);</td>
<td>11QII</td>
</tr>
<tr>
<td>a=put(18792,yyqr7.);</td>
<td>2011QII</td>
</tr>
<tr>
<td>a=put(18792,yyqr8.);</td>
<td>2011QII</td>
</tr>
<tr>
<td>a=put(18792,yyqr9.);</td>
<td>2011QII</td>
</tr>
<tr>
<td>a=put(18792,yyqr10.);</td>
<td>2011QII</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “YYQw. Format” on page 326
- “YYQRxw. Format” on page 330
- “YYQxw. Format” on page 327
- “YYQZw. Format” on page 332

YYQRxw. Format

Writes SAS date values in the form <yy>yyqr or <yy>yy-qr, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits and qr is the quarter of the year expressed in roman numerals.

Category: Date and Time
Alignment: Right
Syntax

YYQRxw.

Arguments

\(x\)

identifies a separator or specifies that no separator appear between the year and the quarter. Here are the valid values:

- C separates with a colon
- D separates with a hyphen
- N indicates no separator
- P separates with a period
- S separates with a forward slash.

\(w\)

specifies the width of the output field.

Default 8

Range 6–32

Interactions

When \(x\) is set to \(N\), no separator is specified. The width range is then 5–32, and the default changes to 7.

When the value of \(w\) is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQRxw. format writes SAS date values in the form \(<yy>yyqr\) or \(<yy>yyXqr\). Here is an explanation of the syntax:

- \(<yy>yy\)
  
  is a two-digit or four-digit integer that represents the year.

- X
  
  is a specified separator.

- qr
  
  is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Comparisons

The YYQRw. format is similar to the YYQRxw. format, but the YYQRxw. format has separators between the YY and QR, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 18985, which is the SAS date value that corresponds to December 24, 2011.
### YYQZw. Format

Writes SAS date values in the form `<yy><qq>`, the year appears as 2 or 4 digits, and `qq` is the quarter of the year.

**Category:** Date and Time

**Alignment:** Right

### Syntax

`YYQZw`.  

### Arguments

- **Z**  
  specifies that no separator appear between the year and the quarter.

- **w**  
  specifies the width of the output field.

  **Default** 4

  **Note** 6
Details

The YYQZw. format writes SAS date values in the form <yy> <qq>. Here is an explanation of the syntax:

<yy>
  is a two-digit or four-digit integer that represents the year.
Z
  specifies that there is no separator.
<qq>
  is an integer (01, 02, 03, or 04) that represents the quarter of the year.

Example

The following examples use the input value of 18822, which is the SAS date value that corresponds to July 14, 2011.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put (18822,yyqz6.);</td>
<td>201103</td>
</tr>
<tr>
<td>a=put (18822,yyqz4.);</td>
<td>1103</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Formats:

- “YYQw. Format” on page 326
- “YYQxw. Format” on page 327
- “YYQRw. Format” on page 329
- “YYQRxw. Format” on page 330

Zw.d Format

Writes standard numeric data with leading 0s.

- Category: Numeric
- Alignment: Right
- Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in SAS System Options: Reference.
Syntax

\[ Z_{\text{w}.[d]} \]

Arguments

\( w \)

specifies the width of the output field.

Default 1

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

\( d \)

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Tip If \( d \) is 0 or you omit \( d \), \( Z_{\text{w}.d} \) writes the value without a decimal point.

Details

The \( Z_{\text{w}.d} \) format writes standard numeric values one digit per byte and fills in 0s to the left of the data value.

The \( Z_{\text{w}.d} \) format rounds to the nearest number that will fit in the output field. If \( w.d \) is too large to fit, SAS might shift the decimal to the BESTw. format. The \( Z_{\text{w}.d} \) format writes negative numbers with leading minus signs. In addition, it right aligns before writing and pads the output with leading zeros.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=put(1350, z8.);</td>
<td>00001350</td>
</tr>
<tr>
<td>a=put(1350, z8.);</td>
<td>00001350</td>
</tr>
</tbody>
</table>
Chapter 23
DS2 Functions

Overview of DS2 Functions .................................................. 340
General Function Syntax ..................................................... 341
Using Functions ................................................................. 341
  Restrictions on Function Arguments .................................. 341
  Data Type Conversion in Functions .................................... 342
  Missing and Null Values in DS2 Numeric Functions ............... 342
  Missing and Null Values in DS2 Character Functions .......... 342
  Using a System Expression to Execute a Function ............... 342
  Notes on Descriptive Statistic Functions ......................... 343
DS2 Function Examples ....................................................... 343
Function Categories ......................................................... 343
Dictionary ........................................................................ 358
  ABS Function ................................................................. 358
  ANYALNUM Function ...................................................... 359
  ANYALPHA Function ...................................................... 362
  ANYCNTRL Function ...................................................... 364
  ANYDIGIT Function ....................................................... 366
  ANYFIRST Function ....................................................... 368
  ANYGRAPH Function ..................................................... 370
  ANYLOWER Function ....................................................... 372
  ANYNAME Function ....................................................... 374
  ANYPRINT Function ...................................................... 376
  ANYPUNCT Function ...................................................... 378
  ANYSpace Function ......................................................... 381
  ANYUPPER Function ....................................................... 383
  ANYXDIGIT Function ...................................................... 385
  ARCOS Function ............................................................. 386
  ARCSH Function ............................................................. 387
  ARSIN Function ............................................................. 388
  ARSINH Function ........................................................... 389
  ARTANH Function .......................................................... 390
  ATAN Function .............................................................. 391
  ATAN2 Function ............................................................ 392
  BAND Function .............................................................. 393
  BETA Function ............................................................... 394
  BETAINV Function ........................................................ 395
  BLACKCLPRC Function .................................................... 396
  BLACKPTPRC Function .................................................... 399
  BLKSHCLPRC Function ................................................... 401
BLKSHPTPRC Function .............................................. 403
BLSHIFT Function .................................................. 405
BNOT Function ...................................................... 406
BOR Function ....................................................... 406
BRSHIFT Function .................................................. 407
BXOR Function ....................................................... 408
BYTE Function ....................................................... 408
CAT Function ........................................................ 409
CATS Function ....................................................... 411
CATT Function ....................................................... 413
CATX Function ....................................................... 415
CEIL Function ......................................................... 419
CEILZ Function ....................................................... 420
CHOOSEC Function .................................................. 421
CHOSENE Function .................................................. 423
CMP Function ......................................................... 424
CMPT Function ....................................................... 425
COALESCE Function ................................................ 426
COALESCEC Function .............................................. 427
COMB Function ....................................................... 428
COMPARE Function .................................................. 429
COMPBL Function .................................................... 433
COMPFUZZ Function ................................................. 434
COMPOUND Function .............................................. 436
COMPRESS Function ............................................... 437
CONSTANT Function ............................................... 439
CONVX Function ..................................................... 444
CONVXP Function ................................................... 445
COS Function ........................................................ 447
COSH Function ....................................................... 448
COUNT Function ..................................................... 449
COUNTC Function .................................................... 451
COUNTW Function ................................................... 454
CSS Function ........................................................ 458
CUMIPMT Function ................................................ 459
CUMPRINC Function .............................................. 460
CV Function .......................................................... 462
DATDIF Function ..................................................... 463
DATE Function ......................................................... 466
DATEJUL Function .................................................... 466
DATEPART Function .................................................. 467
DATETIME Function ................................................ 468
DAY Function ........................................................ 469
DEQUOTE Function ................................................ 470
DEVIANCE Function ............................................... 472
DHMS Function ....................................................... 476
DIGAMMA Function ................................................. 477
DIM Function ........................................................ 478
DIVIDE Function ..................................................... 480
DUR Function ......................................................... 481
DURP Function ....................................................... 483
EFFRANTE Function ............................................... 485
ERF Function ........................................................ 486
ERFC Function ....................................................... 487
EXP Function ........................................................ 488
FACT Function ........................................................ 488
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND Function</td>
<td>489</td>
</tr>
<tr>
<td>FINDC Function</td>
<td>492</td>
</tr>
<tr>
<td>FINDW Function</td>
<td>499</td>
</tr>
<tr>
<td>FLOOR Function</td>
<td>505</td>
</tr>
<tr>
<td>FLOOZ Function</td>
<td>506</td>
</tr>
<tr>
<td>FMTINFO Function</td>
<td>507</td>
</tr>
<tr>
<td>FUZZ Function</td>
<td>509</td>
</tr>
<tr>
<td>GAMINV Function</td>
<td>510</td>
</tr>
<tr>
<td>GAMMA Function</td>
<td>511</td>
</tr>
<tr>
<td>GARKHCLPRC Function</td>
<td>512</td>
</tr>
<tr>
<td>GARKHPTPRC Function</td>
<td>514</td>
</tr>
<tr>
<td>GCD Function</td>
<td>517</td>
</tr>
<tr>
<td>GEODIST Function</td>
<td>517</td>
</tr>
<tr>
<td>GEOMEAN Function</td>
<td>520</td>
</tr>
<tr>
<td>GEOMEANZ Function</td>
<td>522</td>
</tr>
<tr>
<td>HARMean Function</td>
<td>523</td>
</tr>
<tr>
<td>HARMeanZ Function</td>
<td>525</td>
</tr>
<tr>
<td>HBOUND Function</td>
<td>526</td>
</tr>
<tr>
<td>HMS Function</td>
<td>528</td>
</tr>
<tr>
<td>HOLIDAY Function</td>
<td>529</td>
</tr>
<tr>
<td>HOUR Function</td>
<td>532</td>
</tr>
<tr>
<td>INDEX Function</td>
<td>533</td>
</tr>
<tr>
<td>INDEXC Function</td>
<td>535</td>
</tr>
<tr>
<td>INDEXW Function</td>
<td>536</td>
</tr>
<tr>
<td>INPUTC Function</td>
<td>538</td>
</tr>
<tr>
<td>INPUTN Function</td>
<td>539</td>
</tr>
<tr>
<td>INT Function</td>
<td>541</td>
</tr>
<tr>
<td>INTINDEX Function</td>
<td>542</td>
</tr>
<tr>
<td>INTCK Function</td>
<td>545</td>
</tr>
<tr>
<td>INTCYCLE Function</td>
<td>552</td>
</tr>
<tr>
<td>INTDT Function</td>
<td>555</td>
</tr>
<tr>
<td>INTFIT Function</td>
<td>556</td>
</tr>
<tr>
<td>INTGET Function</td>
<td>557</td>
</tr>
<tr>
<td>INTINDEX Function</td>
<td>560</td>
</tr>
<tr>
<td>INTNX Function</td>
<td>564</td>
</tr>
<tr>
<td>INTRR Function</td>
<td>571</td>
</tr>
<tr>
<td>INTSEAS Function</td>
<td>572</td>
</tr>
<tr>
<td>INTSHIFT Function</td>
<td>576</td>
</tr>
<tr>
<td>INTTEST Function</td>
<td>578</td>
</tr>
<tr>
<td>INTTS Function</td>
<td>580</td>
</tr>
<tr>
<td>INTZ Function</td>
<td>581</td>
</tr>
<tr>
<td>IPMT Function</td>
<td>582</td>
</tr>
<tr>
<td>IQR Function</td>
<td>584</td>
</tr>
<tr>
<td>IRR Function</td>
<td>585</td>
</tr>
<tr>
<td>JULDATE Function</td>
<td>586</td>
</tr>
<tr>
<td>JULDATE7 Function</td>
<td>587</td>
</tr>
<tr>
<td>KCOUNT Function</td>
<td>588</td>
</tr>
<tr>
<td>KSTRCAT Function</td>
<td>589</td>
</tr>
<tr>
<td>KSTRIP Function</td>
<td>590</td>
</tr>
<tr>
<td>KUPDATE Function</td>
<td>592</td>
</tr>
<tr>
<td>KUPDATES Function</td>
<td>595</td>
</tr>
<tr>
<td>KURTOSIS Function</td>
<td>597</td>
</tr>
<tr>
<td>LARGEST Function</td>
<td>598</td>
</tr>
<tr>
<td>LBOUND Function</td>
<td>600</td>
</tr>
<tr>
<td>LCM Function</td>
<td>601</td>
</tr>
<tr>
<td>LEFT Function</td>
<td>602</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page Number</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>LENGTH Function</td>
<td>603</td>
</tr>
<tr>
<td>LENGTHH Function</td>
<td>605</td>
</tr>
<tr>
<td>LENGTHM Function</td>
<td>606</td>
</tr>
<tr>
<td>LENGTHN Function</td>
<td>607</td>
</tr>
<tr>
<td>LAGAMMA Function</td>
<td>608</td>
</tr>
<tr>
<td>LOG Function</td>
<td>609</td>
</tr>
<tr>
<td>LOGBETA Function</td>
<td>610</td>
</tr>
<tr>
<td>LOG10 Function</td>
<td>611</td>
</tr>
<tr>
<td>LOG1PX Function</td>
<td>612</td>
</tr>
<tr>
<td>LOG2 Function</td>
<td>613</td>
</tr>
<tr>
<td>LOWCASE Function</td>
<td>614</td>
</tr>
<tr>
<td>MAD Function</td>
<td>615</td>
</tr>
<tr>
<td>MARGRCLPRC Function</td>
<td>616</td>
</tr>
<tr>
<td>MARGRPTPRC Function</td>
<td>618</td>
</tr>
<tr>
<td>MAX Function</td>
<td>621</td>
</tr>
<tr>
<td>MD5 Function</td>
<td>622</td>
</tr>
<tr>
<td>MDY Function</td>
<td>624</td>
</tr>
<tr>
<td>MEAN Function</td>
<td>625</td>
</tr>
<tr>
<td>MEDIAN Function</td>
<td>626</td>
</tr>
<tr>
<td>MIN Function</td>
<td>627</td>
</tr>
<tr>
<td>MINUTE Function</td>
<td>628</td>
</tr>
<tr>
<td>MISSING Function</td>
<td>629</td>
</tr>
<tr>
<td>MOD Function</td>
<td>631</td>
</tr>
<tr>
<td>MODZ Function</td>
<td>633</td>
</tr>
<tr>
<td>MONTH Function</td>
<td>635</td>
</tr>
<tr>
<td>MORT Function</td>
<td>635</td>
</tr>
<tr>
<td>N Function</td>
<td>637</td>
</tr>
<tr>
<td>NDMIPS Function</td>
<td>638</td>
</tr>
<tr>
<td>NETPV Function</td>
<td>640</td>
</tr>
<tr>
<td>NMISI Function</td>
<td>641</td>
</tr>
<tr>
<td>NOMRATE Function</td>
<td>642</td>
</tr>
<tr>
<td>NOTALNUM Function</td>
<td>644</td>
</tr>
<tr>
<td>NOTALPHA Function</td>
<td>645</td>
</tr>
<tr>
<td>NOTCNTRL Function</td>
<td>648</td>
</tr>
<tr>
<td>NOTDIGIT Function</td>
<td>649</td>
</tr>
<tr>
<td>NOTFIRST Function</td>
<td>651</td>
</tr>
<tr>
<td>NOTGRAPH Function</td>
<td>653</td>
</tr>
<tr>
<td>NOTLOWER Function</td>
<td>655</td>
</tr>
<tr>
<td>NOTNAME Function</td>
<td>657</td>
</tr>
<tr>
<td>NOTPRINT Function</td>
<td>659</td>
</tr>
<tr>
<td>NOTPUNCT Function</td>
<td>661</td>
</tr>
<tr>
<td>NOTSPACE Function</td>
<td>663</td>
</tr>
<tr>
<td>NOTUPPER Function</td>
<td>666</td>
</tr>
<tr>
<td>NOTXDIGIT Function</td>
<td>668</td>
</tr>
<tr>
<td>NPV Function</td>
<td>669</td>
</tr>
<tr>
<td>NULL Function</td>
<td>670</td>
</tr>
<tr>
<td>NWKDOM Function</td>
<td>673</td>
</tr>
<tr>
<td>ORDINAL Function</td>
<td>675</td>
</tr>
<tr>
<td>PCTL Function</td>
<td>676</td>
</tr>
<tr>
<td>PERM Function</td>
<td>678</td>
</tr>
<tr>
<td>PMT Function</td>
<td>679</td>
</tr>
<tr>
<td>POISSON Function</td>
<td>680</td>
</tr>
<tr>
<td>POWER Function</td>
<td>681</td>
</tr>
<tr>
<td>PPMT Function</td>
<td>682</td>
</tr>
<tr>
<td>PROBBETA Function</td>
<td>683</td>
</tr>
<tr>
<td>PROBBNML Function</td>
<td>684</td>
</tr>
</tbody>
</table>
Overview of DS2 Functions

A **DS2 function** performs a computation or system manipulation on arguments and returns a value. A function expression invokes a function from anywhere in a DS2 program, method, or thread. Most functions use arguments supplied by the user, but a few obtain their arguments from the operating environment.

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAMINIT Function</td>
<td>789</td>
</tr>
<tr>
<td>STRIP Function</td>
<td>790</td>
</tr>
<tr>
<td>SUBSTR Function</td>
<td>792</td>
</tr>
<tr>
<td>SUBSTRN Function</td>
<td>794</td>
</tr>
<tr>
<td>SUM Function</td>
<td>798</td>
</tr>
<tr>
<td>SUMABS Function</td>
<td>799</td>
</tr>
<tr>
<td>TAN Function</td>
<td>800</td>
</tr>
<tr>
<td>TANH Function</td>
<td>801</td>
</tr>
<tr>
<td>TIME Function</td>
<td>802</td>
</tr>
<tr>
<td>TIMEPART Function</td>
<td>803</td>
</tr>
<tr>
<td>TIMEVALUE Function</td>
<td>803</td>
</tr>
<tr>
<td>TINV Function</td>
<td>805</td>
</tr>
<tr>
<td>TO_DATE Function</td>
<td>807</td>
</tr>
<tr>
<td>TO_DOUBLE Function</td>
<td>808</td>
</tr>
<tr>
<td>TO_TIME Function</td>
<td>811</td>
</tr>
<tr>
<td>TO_TIMESTAMP Function</td>
<td>812</td>
</tr>
<tr>
<td>TODAY Function</td>
<td>813</td>
</tr>
<tr>
<td>TRANSLATE Function</td>
<td>814</td>
</tr>
<tr>
<td>TRANSTRN Function</td>
<td>815</td>
</tr>
<tr>
<td>TRANWRD Function</td>
<td>818</td>
</tr>
<tr>
<td>TRIGAMMA Function</td>
<td>820</td>
</tr>
<tr>
<td>TRIM Function</td>
<td>821</td>
</tr>
<tr>
<td>TRIMN Function</td>
<td>822</td>
</tr>
<tr>
<td>TRUNC Function</td>
<td>823</td>
</tr>
<tr>
<td>UNIFORM Function</td>
<td>824</td>
</tr>
<tr>
<td>UPCASE Function</td>
<td>825</td>
</tr>
<tr>
<td>USS Function</td>
<td>826</td>
</tr>
<tr>
<td>UUIDGEN Function</td>
<td>826</td>
</tr>
<tr>
<td>VAR Function</td>
<td>827</td>
</tr>
<tr>
<td>VERIFY Function</td>
<td>828</td>
</tr>
<tr>
<td>VFORMAT Function</td>
<td>829</td>
</tr>
<tr>
<td>VINARRAY Function</td>
<td>831</td>
</tr>
<tr>
<td>VINFORMATT Function</td>
<td>832</td>
</tr>
<tr>
<td>VLABEL Function</td>
<td>833</td>
</tr>
<tr>
<td>VLENGTH Function</td>
<td>835</td>
</tr>
<tr>
<td>VNAME Function</td>
<td>836</td>
</tr>
<tr>
<td>VTYPE Function</td>
<td>837</td>
</tr>
<tr>
<td>WEEK Function</td>
<td>839</td>
</tr>
<tr>
<td>WEEKDAY Function</td>
<td>842</td>
</tr>
<tr>
<td>WHICHC Function</td>
<td>842</td>
</tr>
<tr>
<td>WHICHN Function</td>
<td>844</td>
</tr>
<tr>
<td>YEAR Function</td>
<td>845</td>
</tr>
<tr>
<td>YIELDP Function</td>
<td>846</td>
</tr>
<tr>
<td>YRDIF Function</td>
<td>847</td>
</tr>
<tr>
<td>YYQ Function</td>
<td>850</td>
</tr>
</tbody>
</table>
If the data types of the arguments in the function expression are not what is expected by the DS2 function, DS2 performs a type conversion on the arguments so that they have the appropriate data type. If the type conversion is successful, the function executes. Otherwise, an error is issued. For information, see Chapter 12, “DS2 Type Conversions,” on page 87.

Note: DS2 does not support CALL routines. DS2 functions and methods can alter variable argument values if the return type of the function or method is VOID.

Note: The date and time functions work only with SAS date, time, and datetime values. They do not work with values having the DATE, TIME, and TIMESTAMP data types. For information about working with dates, see Chapter 14, “Dates and Times in DS2,” on page 111.

General Function Syntax

The syntax of a function is

\[ \text{function-name}(\text{argument}[, \ldots \text{argument}]) \]

Here is an explanation of the syntax:

- \textit{function-name} names the function.

- \textit{argument} can be a variable name, constant, or a DS2 expression, including another function. The number and type of arguments that DS2 allows are described with individual functions. Multiple arguments are separated by a comma.

Tip If the value of an argument is invalid (for example, missing or outside the prescribed range), an error occurs and the function's return expression is set to a missing or null value.

Using Functions

Restrictions on Function Arguments

If the value of an argument is invalid, an error occurs and the return expression is set to a missing or null value. Here are some common restrictions on function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.

- Most functions do not permit missing or null values as arguments. Exceptions include some of the descriptive statistics functions and financial functions.

By default when a function argument contains a missing or null value, an error occurs and a message is printed to the SAS log. You can use the MISSING_NOTE option in the DS2_OPTIONS statement to not produce an error and write a note to the SAS log when a function argument contains a missing or null value. For more information, see “DS2_OPTIONS Statement” on page 894.
In general, the allowed range of the arguments is platform-dependent, such as with the EXP function.

**Data Type Conversion in Functions**

The number of arguments and the argument data types that DS2 expects for a function is called the function signature. When a function executes, the arguments in the function expression are compared to the function signature. If the arguments and the signature match, the function executes. If the number of arguments do not match, an error occurs.

If the number of arguments match but one or more argument data types do not match, DS2 attempts to convert the argument data types to those of the function signature. If the type conversion is successful, the function executes. Otherwise, an error occurs.

The documentation for each of the functions include a valid data type for all function arguments. The valid data type is the argument data type that DS2 uses in executing the function. Because DS2 does type conversion, some argument data types in the function expression do not need to be the same as the valid data types for the function argument. For example, if the function expression contains an argument with a data type of INTEGER and the valid data type for that argument is DOUBLE, DS2 converts the data type to DOUBLE when the function executes. Only four data types, VARBINARY and the date/time data types, DATE, TIME, TIMESTAMP, are non-coercible, meaning that the function expression must contain the valid data type.

**Missing and Null Values in DS2 Numeric Functions**

For functions that have an input data type of DOUBLE, if a null is passed to the function, the null value is converted to a missing value.

After the function is processed, if the function returns a DOUBLE and if the function returns a missing value, a missing value is returned in SAS mode and a null value is returned in ANSI mode.

For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

**Missing and Null Values in DS2 Character Functions**

If you pass a character function a null value and the function should return a null value, a null value is returned regardless of whether you are in ANSI mode or SAS mode.

**Using a System Expression to Execute a Function**

The syntax for a function is similar to the syntax for a method, and when DS2 encounters a method or function expression, it must determine what type of expression it is. If a method expression and a function expression have the same name, DS2 executes the method. The only way to execute a function that has the same name as a method is to use a system expression. A system expression has the following syntax:

```
system.function-expression
```

When DS2 encounters a system expression, the call to the method of the same name is bypassed and DS2 calls the function.
Notes on Descriptive Statistic Functions

DS2 provides functions that return descriptive statistics. Except for the MISSING function, the functions correspond to the statistics produced by the Base SAS MEANS procedure. The computing method for each statistic is discussed in the statistical procedures section of the *Base SAS Procedures Guide*. SAS calculates descriptive statistics for the non-null or nonmissing values of the arguments.

DS2 Function Examples

```plaintext
x=max(cash,credit);
x=sqrt(1500);
NewCity=left(upcase(City));
x=min(YearTemperature-July,YearTemperature-Dec);
s=repeat('----+',16);
x=min((enroll-drop),(enroll-fail));
if sum(cash,credit)>1000 then
  put 'Goal reached';
```

Function Categories

Functions can be categorized by the types of values that they operate on. Each DS2 function belongs to one of the following categories:

Array
- operates on a named aggregate collection of homogenous data

Bitwise Logical Operations
- operates on one or more bit patterns or binary numbers at the level of their individual bits

Character
- operates on character data and SQL expressions

Date and Time
- operates on date and time values

Descriptive Statistics
- operates on values that measure central tendency, variation among values, and the shape of distribution values

Distance
- returns the geodetic distance

Financial
- calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks.

Hyperbolic
- performs hyperbolic calculations such as sine, cosine, and tangent
Mathematical
  operates on values to perform general mathematical calculations

Numeric
  operates on numeric values

Probability
  returns probability calculations.

Quantile
  returns a quantile from specific distributions

Random Number
  returns random variates from specific distributions

Special
  operates on null values and SAS missing values, suspends execution of a program,
  specifies numeric informats at run time, and executes a FedSQL statement.

Trigonometric
  operates on values to perform trigonometric calculations

Truncation
  operates on values to limit the number of digits

Variable Information
  operates on variables and returns names, types, lengths, informats, labels, and other
  variable information

The following table provides brief descriptions of DS2 functions. For more detailed
information, see the individual functions.

<table>
<thead>
<tr>
<th>Category</th>
<th>Language Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>STD Function (p. 787)</td>
<td>Returns the standard deviation.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>DIVIDE Function (p. 480)</td>
<td>Returns the result of a division that handles special missing values for ODS output.</td>
</tr>
<tr>
<td></td>
<td>ERF Function (p. 486)</td>
<td>Returns the value of the (normal) error function.</td>
</tr>
<tr>
<td></td>
<td>ERFC Function (p. 487)</td>
<td>Returns the value of the complementary (normal) error function.</td>
</tr>
<tr>
<td>Array</td>
<td>DIM Function (p. 478)</td>
<td>Returns the number of elements in an array.</td>
</tr>
<tr>
<td></td>
<td>HBOUND Function (p. 526)</td>
<td>Returns the upper bound of an array.</td>
</tr>
<tr>
<td></td>
<td>LBOUND Function (p. 600)</td>
<td>Returns the lower bound of an array.</td>
</tr>
<tr>
<td></td>
<td>NDIMS Function (p. 638)</td>
<td>Returns the number of dimensions in an array.</td>
</tr>
<tr>
<td>Bitwise Logical</td>
<td>BAND Function (p. 393)</td>
<td>Returns the bitwise logical AND of two arguments.</td>
</tr>
<tr>
<td>Operations</td>
<td>BLSHIFT Function (p. 405)</td>
<td>Returns the bitwise logical left shift of two arguments.</td>
</tr>
<tr>
<td></td>
<td>BNOT Function (p. 406)</td>
<td>Returns the bitwise logical NOT of an argument.</td>
</tr>
<tr>
<td></td>
<td>BOR Function (p. 406)</td>
<td>Returns the bitwise logical OR of two arguments.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>BRSHIFT Function (p. 407)</td>
<td>Returns the bitwise logical right shift of two arguments.</td>
</tr>
<tr>
<td></td>
<td>BXOR Function (p. 408)</td>
<td>Returns the bitwise logical EXCLUSIVE OR of two arguments.</td>
</tr>
<tr>
<td>Character</td>
<td>ANYALNUM Function (p. 359)</td>
<td>Searches a character string for an alphanumeric character, and returns the first position at which the character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYALPHA Function (p. 362)</td>
<td>Searches a character string for an alphabetic character, and returns the first position at which the character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYCNTRL Function (p. 364)</td>
<td>Searches a character string for a control character, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYDIGIT Function (p. 366)</td>
<td>Searches a character string for a digit, and returns the first position at which the digit is found.</td>
</tr>
<tr>
<td></td>
<td>ANYFIRST Function (p. 368)</td>
<td>Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYGRAPH Function (p. 370)</td>
<td>Searches a character string for a graphical character, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYLOWER Function (p. 372)</td>
<td>Searches a character string for a lowercase letter, and returns the first position at which the letter is found.</td>
</tr>
<tr>
<td></td>
<td>ANYNAME Function (p. 374)</td>
<td>Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYPRINT Function (p. 376)</td>
<td>Searches a character string for a printable character, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYPUNCT Function (p. 378)</td>
<td>Searches a character string for a punctuation character, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYSIZE Function (p. 381)</td>
<td>Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>ANYUPPER Function (p. 383)</td>
<td>Searches a character string for an uppercase letter, and returns the first position at which the letter is found.</td>
</tr>
<tr>
<td></td>
<td>ANYXDIGIT Function (p. 385)</td>
<td>Searches a character string for a hexadecimal character that represents a digit, and returns the first position at which that character is found.</td>
</tr>
<tr>
<td></td>
<td>BYTE Function (p. 408)</td>
<td>Returns one character in the ASCII or the EBCDIC collating sequence.</td>
</tr>
<tr>
<td></td>
<td>CAT Function (p. 409)</td>
<td>Does not remove leading or trailing blanks, and returns a concatenated character string.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CATS Function (p. 411)</td>
<td>Removes leading and trailing blanks, and returns a concatenated character string.</td>
<td></td>
</tr>
<tr>
<td>CATT Function (p. 413)</td>
<td>Removes trailing blanks, and returns a concatenated character string.</td>
<td></td>
</tr>
<tr>
<td>CATX Function (p. 415)</td>
<td>Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.</td>
<td></td>
</tr>
<tr>
<td>CHOOSEC Function (p. 421)</td>
<td>Returns a character value that represents the results of choosing from a list of arguments.</td>
<td></td>
</tr>
<tr>
<td>CHOOSEN Function (p. 423)</td>
<td>Returns a numeric value that represents the results of choosing from a list of arguments.</td>
<td></td>
</tr>
<tr>
<td>CMP Function (p. 424)</td>
<td>Compares two character strings including trailing blanks.</td>
<td></td>
</tr>
<tr>
<td>CMPT Function (p. 425)</td>
<td>Compares two character strings excluding trailing blanks.</td>
<td></td>
</tr>
<tr>
<td>COALESCEC Function (p. 427)</td>
<td>Returns the first non-null or nonmissing value from a list of character arguments.</td>
<td></td>
</tr>
<tr>
<td>COMPARE Function (p. 429)</td>
<td>Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.</td>
<td></td>
</tr>
<tr>
<td>COMPBL Function (p. 433)</td>
<td>Removes multiple blanks from a character string.</td>
<td></td>
</tr>
<tr>
<td>COMPRESS Function (p. 437)</td>
<td>Returns a character string with specified characters removed from the original string.</td>
<td></td>
</tr>
<tr>
<td>COUNT Function (p. 449)</td>
<td>Counts the number of times that a specified substring appears within a character string.</td>
<td></td>
</tr>
<tr>
<td>COUNTC Function (p. 451)</td>
<td>Counts the number of characters in a string that appear or do not appear in a list of characters.</td>
<td></td>
</tr>
<tr>
<td>COUNTW Function (p. 454)</td>
<td>Counts the number of words in a character string.</td>
<td></td>
</tr>
<tr>
<td>DEQUOTE Function (p. 470)</td>
<td>Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.</td>
<td></td>
</tr>
<tr>
<td>FIND Function (p. 489)</td>
<td>Searches for a specific substring of characters within a character string.</td>
<td></td>
</tr>
<tr>
<td>FINDC Function (p. 492)</td>
<td>Searches a string for any character in a list of characters.</td>
<td></td>
</tr>
<tr>
<td>FINDW Function (p. 499)</td>
<td>Returns the character position of a word in a string, or returns the number of the word in a string.</td>
<td></td>
</tr>
<tr>
<td>INDEX Function (p. 533)</td>
<td>Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>INDEXC Function (p. 535)</td>
<td>Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.</td>
<td></td>
</tr>
<tr>
<td>INDEXW Function (p. 536)</td>
<td>Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.</td>
<td></td>
</tr>
<tr>
<td>LEFT Function (p. 602)</td>
<td>Left aligns a character expression.</td>
<td></td>
</tr>
<tr>
<td>LENGTH Function (p. 603)</td>
<td>Returns the length of a character string, excluding trailing blanks, and returns a 0 for a blank character string.</td>
<td></td>
</tr>
<tr>
<td>LENGTHC Function (p. 605)</td>
<td>Returns the length of a character string, including trailing blanks.</td>
<td></td>
</tr>
<tr>
<td>LENGTHM Function (p. 606)</td>
<td>Returns the amount of memory, in characters, that is allocated for a character string.</td>
<td></td>
</tr>
<tr>
<td>LENGTHN Function (p. 607)</td>
<td>Returns the length of a character string, excluding trailing blanks.</td>
<td></td>
</tr>
<tr>
<td>LOWCASE Function (p. 614)</td>
<td>Converts all letters in a character expression to lowercase.</td>
<td></td>
</tr>
<tr>
<td>MD5 Function (p. 622)</td>
<td>Returns the result of the message digest of a specified string in binary format.</td>
<td></td>
</tr>
<tr>
<td>NOTALNUM Function (p. 644)</td>
<td>Searches a character string for a non-alphanumeric character, and returns the first position at which the character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTALPHA Function (p. 645)</td>
<td>Searches a character string for a nonalphabetic character, and returns the first position at which the character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTCNTRL Function (p. 648)</td>
<td>Searches a character string for a character that is not a control character, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTDIGIT Function (p. 649)</td>
<td>Searches a character string for any character that is not a digit, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTFIRST Function (p. 651)</td>
<td>Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAMES=V7, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTGRAPH Function (p. 653)</td>
<td>Searches a character string for a non-graphical character, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTLOWER Function (p. 655)</td>
<td>Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTNAME Function (p. 657)</td>
<td>Searches a character string for an invalid character in a SAS variable name under VALIDVARNAMES=V7, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTPRINT Function (p. 659)</td>
<td>Searches a character string for a nonprintable character, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NOTPUNCT Function (p. 661)</td>
<td>Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTSPACE Function (p. 663)</td>
<td>Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTUPPER Function (p. 666)</td>
<td>Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>NOTXDIGIT Function (p. 668)</td>
<td>Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.</td>
<td></td>
</tr>
<tr>
<td>QUOTE Function (p. 730)</td>
<td>Adds double quotation marks to a character value.</td>
<td></td>
</tr>
<tr>
<td>RANK Function (p. 747)</td>
<td>Returns the position of a character in the ASCII or EBCDIC collating sequence.</td>
<td></td>
</tr>
<tr>
<td>REPEAT Function (p. 753)</td>
<td>Repeats a character expression.</td>
<td></td>
</tr>
<tr>
<td>REVERSE Function (p. 754)</td>
<td>Reverses a character expression.</td>
<td></td>
</tr>
<tr>
<td>RIGHT Function (p. 754)</td>
<td>Right aligns a character expression.</td>
<td></td>
</tr>
<tr>
<td>SCAN Function (p. 772)</td>
<td>Returns the nth word from a character expression.</td>
<td></td>
</tr>
<tr>
<td>SHA256HEX Function (p. 776)</td>
<td>Returns the result of the message digest of a specified string and converts the string to hexadecimal representation.</td>
<td></td>
</tr>
<tr>
<td>SHA256HMACHEX Function (p. 778)</td>
<td>Returns the result of the message digest of a specified string by using the Hash-based Message Authentication (HMAC) algorithm.</td>
<td></td>
</tr>
<tr>
<td>STRIP Function (p. 790)</td>
<td>Returns a character string with all leading and trailing blanks removed.</td>
<td></td>
</tr>
<tr>
<td>SUBSTR Function (p. 792)</td>
<td>Returns a substring, allowing a result with a length of zero.</td>
<td></td>
</tr>
<tr>
<td>SUBSTRN Function (p. 794)</td>
<td>Returns a substring, allowing a result with a length of zero.</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE Function (p. 814)</td>
<td>Replaces specific characters in a character expression.</td>
<td></td>
</tr>
<tr>
<td>TRANSTRN Function (p. 815)</td>
<td>Replaces or removes all occurrences of a substring in a character string.</td>
<td></td>
</tr>
<tr>
<td>TRANWRD Function (p. 818)</td>
<td>Replaces or removes all occurrences of a word in a character string.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TRIM Function (p. 821)</td>
<td>Removes trailing blanks from a character</td>
<td>expression, and returns one blank if the string is missing.</td>
</tr>
<tr>
<td>TRIMN Function (p. 822)</td>
<td>Removes trailing blanks from character</td>
<td>expressions, and returns a string with a length of zero if the expression</td>
</tr>
<tr>
<td></td>
<td>expression is missing.</td>
<td></td>
</tr>
<tr>
<td>UPCASE Function (p. 825)</td>
<td>Converts all letters in an argument to</td>
<td>uppercase.</td>
</tr>
<tr>
<td>VERIFY Function (p. 828)</td>
<td>Returns the position of the first character</td>
<td>that is unique to an expression.</td>
</tr>
<tr>
<td>WHICHC Function (p. 842)</td>
<td>Returns the first position of a character</td>
<td>string from a list of character strings.</td>
</tr>
<tr>
<td>Character String</td>
<td>PRXCHANGE Function (p. 712)</td>
<td>Performs a pattern-matching replacement.</td>
</tr>
<tr>
<td>PRXMATCH Function (p. 715)</td>
<td>Searches for a pattern match and returns</td>
<td>the position at which the pattern is found.</td>
</tr>
<tr>
<td>PRXPAREN Function (p. 718)</td>
<td>Returns the last bracket match for which</td>
<td>there is a match in a pattern.</td>
</tr>
<tr>
<td>PRXPARSE Function (p. 719)</td>
<td>Compiles a Perl regular expression (PRX)</td>
<td>that can be used for pattern matching of a character value.</td>
</tr>
<tr>
<td>PRXPOSN Function (p. 721)</td>
<td>Returns a character string that contains</td>
<td>the value for a capture buffer.</td>
</tr>
<tr>
<td>Combinatorial</td>
<td>COMB Function (p. 428)</td>
<td>Computes the number of combinations of n elements taken r at a time.</td>
</tr>
<tr>
<td>PERM Function (p. 678)</td>
<td>Computes the number of permutations of n</td>
<td>items that are taken r at a time.</td>
</tr>
<tr>
<td>Date and Time</td>
<td>DATDIF Function (p. 463)</td>
<td>Returns the number of days between two dates after computing the difference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>between the dates according to specified day count conventions.</td>
</tr>
<tr>
<td></td>
<td>DATE Function (p. 466)</td>
<td>Returns the current date as a SAS date value.</td>
</tr>
<tr>
<td></td>
<td>DATEJUL Function (p. 466)</td>
<td>Converts a Julian date to a SAS date value.</td>
</tr>
<tr>
<td></td>
<td>DATEPART Function (p. 467)</td>
<td>Extracts the date from a SAS datetime value.</td>
</tr>
<tr>
<td></td>
<td>DATETIME Function (p. 468)</td>
<td>Returns the current date and time of day as a SAS datetime value.</td>
</tr>
<tr>
<td></td>
<td>DAY Function (p. 469)</td>
<td>Returns the day of the month from a SAS date value.</td>
</tr>
<tr>
<td></td>
<td>DHMS Function (p. 476)</td>
<td>Returns a SAS datetime value from date, hour, minute, and second values.</td>
</tr>
<tr>
<td></td>
<td>HMS Function (p. 528)</td>
<td>Returns a SAS time value from hour, minute, and second values.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HOLIDAY Function (p. 529)</td>
<td>Returns a SAS date value of a specified holiday for a specified year.</td>
<td></td>
</tr>
<tr>
<td>HOUR Function (p. 532)</td>
<td>Returns the hour from a SAS time or datetime value.</td>
<td></td>
</tr>
<tr>
<td>INTCINDEX Function (p. 542)</td>
<td>Returns the cycle index when a date, time, or timestamp interval and value are specified.</td>
<td></td>
</tr>
<tr>
<td>INTCK Function (p. 545)</td>
<td>Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.</td>
<td></td>
</tr>
<tr>
<td>INTCYCLE Function (p. 552)</td>
<td>Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.</td>
<td></td>
</tr>
<tr>
<td>INTDT Function (p. 555)</td>
<td>Specifies the number of days to add to a DATE value.</td>
<td></td>
</tr>
<tr>
<td>INTFIT Function (p. 556)</td>
<td>Returns a time interval that is aligned between two dates.</td>
<td></td>
</tr>
<tr>
<td>INTGET Function (p. 557)</td>
<td>Returns a time interval based on three date or datetime values.</td>
<td></td>
</tr>
<tr>
<td>INTINDEX Function (p. 560)</td>
<td>Returns the seasonal index when a date, time, or timestamp interval and value are specified.</td>
<td></td>
</tr>
<tr>
<td>INTNX Function (p. 564)</td>
<td>Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.</td>
<td></td>
</tr>
<tr>
<td>INTSEAS Function (p. 572)</td>
<td>Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.</td>
<td></td>
</tr>
<tr>
<td>INTSHIFT Function (p. 576)</td>
<td>Returns the shift interval that corresponds to the base interval.</td>
<td></td>
</tr>
<tr>
<td>INTTEST Function (p. 578)</td>
<td>Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.</td>
<td></td>
</tr>
<tr>
<td>INTTS Function (p. 580)</td>
<td>Specifies the number of seconds to add to a TIMESTAMP value.</td>
<td></td>
</tr>
<tr>
<td>JULDATE Function (p. 586)</td>
<td>Returns the Julian date from a SAS date value.</td>
<td></td>
</tr>
<tr>
<td>JULDATE7 Function (p. 587)</td>
<td>Returns a seven-digit Julian date from a SAS date value.</td>
<td></td>
</tr>
<tr>
<td>MDY Function (p. 624)</td>
<td>Returns a SAS date value from month, day, and year values.</td>
<td></td>
</tr>
<tr>
<td>MINUTE Function (p. 628)</td>
<td>Returns the minute from a SAS time or datetime value.</td>
<td></td>
</tr>
<tr>
<td>MONTH Function (p. 635)</td>
<td>Returns a number that represents the month from a SAS date value.</td>
<td></td>
</tr>
<tr>
<td>NWKDOM Function (p. 673)</td>
<td>Returns the date for the nth occurrence of a weekday for the specified month and year.</td>
<td></td>
</tr>
<tr>
<td>QTR Function (p. 729)</td>
<td>Returns the quarter of the year from a SAS date value.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SECOND Function</td>
<td>(p. 775)</td>
<td>Returns the second from a SAS time or datetime value.</td>
</tr>
<tr>
<td>TIME Function</td>
<td>(p. 802)</td>
<td>Returns the current time of day as a numeric SAS time value.</td>
</tr>
<tr>
<td>TIMEPART Function</td>
<td>(p. 803)</td>
<td>Extracts a time value from a SAS datetime value.</td>
</tr>
<tr>
<td>TO_DATE Function</td>
<td>(p. 807)</td>
<td>Returns a DATE value from a DOUBLE value that specifies a SAS date value.</td>
</tr>
<tr>
<td>TO_DOUBLE Function</td>
<td>(p. 808)</td>
<td>Returns a DOUBLE value that specifies a SAS date, time, or datetime value.</td>
</tr>
<tr>
<td>TO_TIME Function</td>
<td>(p. 811)</td>
<td>Returns a TIME value from a DOUBLE value that specifies a SAS time value.</td>
</tr>
<tr>
<td>TO_TIMESTAMP Function</td>
<td>(p. 812)</td>
<td>Returns a TIMESTAMP value from a DOUBLE value that specifies a SAS time value.</td>
</tr>
<tr>
<td>TODAY Function</td>
<td>(p. 813)</td>
<td>Returns the current date as a numeric SAS date value.</td>
</tr>
<tr>
<td>WEEK Function</td>
<td>(p. 839)</td>
<td>Returns the week-number value.</td>
</tr>
<tr>
<td>WEEKDAY Function</td>
<td>(p. 842)</td>
<td>From a SAS date value, returns an integer that corresponds to the day of the week.</td>
</tr>
<tr>
<td>YEAR Function</td>
<td>(p. 845)</td>
<td>Returns the year from a SAS date value.</td>
</tr>
<tr>
<td>YRDIF Function</td>
<td>(p. 847)</td>
<td>Returns the difference in years between two dates according to specified day count conventions; returns a person’s age.</td>
</tr>
<tr>
<td>YYQ Function</td>
<td>(p. 850)</td>
<td>Returns a SAS date value from year and quarter year values.</td>
</tr>
<tr>
<td>DBCS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KCOUNT Function</td>
<td>(p. 588)</td>
<td>Returns the number of double-byte characters in an expression.</td>
</tr>
<tr>
<td>KSTRCAT Function</td>
<td>(p. 589)</td>
<td>Concatenates two or more character expressions.</td>
</tr>
<tr>
<td>KSTRIP Function</td>
<td>(p. 590)</td>
<td>Returns a character string with all leading and trailing blanks removed.</td>
</tr>
<tr>
<td>KUPDATE Function</td>
<td>(p. 592)</td>
<td>Inserts, deletes, and replaces character value contents.</td>
</tr>
<tr>
<td>KUPDATES Function</td>
<td>(p. 595)</td>
<td>Inserts, deletes, and replaces character value contents.</td>
</tr>
<tr>
<td>Descriptive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSS Function</td>
<td>(p. 458)</td>
<td>Returns the corrected sum of squares.</td>
</tr>
<tr>
<td>CV Function</td>
<td>(p. 462)</td>
<td>Returns the coefficient of variation.</td>
</tr>
<tr>
<td>GEOMEAN Function</td>
<td>(p. 520)</td>
<td>Returns the geometric mean.</td>
</tr>
<tr>
<td>GEOMEANZ Function</td>
<td>(p. 522)</td>
<td>Returns the geometric mean, using zero fuzzing.</td>
</tr>
<tr>
<td>HARMean Function</td>
<td>(p. 523)</td>
<td>Returns the harmonic mean.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HARMEANZ</td>
<td>Function (p. 525)</td>
<td>Returns the harmonic mean, using zero fuzzing.</td>
</tr>
<tr>
<td>IQR</td>
<td>Function (p. 584)</td>
<td>Returns the interquartile range.</td>
</tr>
<tr>
<td>KURTOSIS</td>
<td>Function (p. 597)</td>
<td>Returns the kurtosis.</td>
</tr>
<tr>
<td>LARGEST</td>
<td>Function (p. 598)</td>
<td>Returns the kth largest non-null or nonmissing value.</td>
</tr>
<tr>
<td>MAD</td>
<td>Function (p. 615)</td>
<td>Returns the median absolute deviation from the median.</td>
</tr>
<tr>
<td>MAX</td>
<td>Function (p. 621)</td>
<td>Returns the largest value from a list of arguments.</td>
</tr>
<tr>
<td>MEAN</td>
<td>Function (p. 625)</td>
<td>Returns the arithmetic mean (average) of the non-null or nonmissing arguments.</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>Function (p. 626)</td>
<td>Returns the median value.</td>
</tr>
<tr>
<td>MIN</td>
<td>Function (p. 627)</td>
<td>Returns the smallest value.</td>
</tr>
<tr>
<td>N</td>
<td>Function (p. 637)</td>
<td>Returns the number of non-null or nonmissing numeric values.</td>
</tr>
<tr>
<td>NMISS</td>
<td>Function (p. 641)</td>
<td>Returns the number of null and SAS missing numeric values.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>Function (p. 675)</td>
<td>Orders a list of values, and returns a value that is based on a position in the list.</td>
</tr>
<tr>
<td>PCTL</td>
<td>Function (p. 676)</td>
<td>Returns the percentile that corresponds to the percentage.</td>
</tr>
<tr>
<td>RANGE</td>
<td>Function (p. 746)</td>
<td>Returns the difference between the largest and the smallest values.</td>
</tr>
<tr>
<td>RMS</td>
<td>Function (p. 755)</td>
<td>Returns the root mean square.</td>
</tr>
<tr>
<td>SKEWNESS</td>
<td>Function (p. 782)</td>
<td>Returns the skewness.</td>
</tr>
<tr>
<td>SMALLEST</td>
<td>Function (p. 784)</td>
<td>Returns the kth smallest non-null or nonmissing value.</td>
</tr>
<tr>
<td>STD</td>
<td>Function (p. 787)</td>
<td>Returns the standard deviation.</td>
</tr>
<tr>
<td>STDERR</td>
<td>Function (p. 788)</td>
<td>Returns the standard error of the mean.</td>
</tr>
<tr>
<td>SUM</td>
<td>Function (p. 798)</td>
<td>Returns the sum of the non-null or nonmissing arguments.</td>
</tr>
<tr>
<td>SUMABS</td>
<td>Function (p. 799)</td>
<td>Returns the sum of the absolute values of the nonmissing arguments.</td>
</tr>
<tr>
<td>USS</td>
<td>Function (p. 826)</td>
<td>Returns the uncorrected sum of squares.</td>
</tr>
<tr>
<td>VAR</td>
<td>Function (p. 827)</td>
<td>Returns the variance.</td>
</tr>
<tr>
<td>Distance</td>
<td>GEODIST Function (p. 517)</td>
<td>Returns the geodetic distance between two latitude and longitude coordinates.</td>
</tr>
</tbody>
</table>

Chapter 23 • DS2 Functions
<table>
<thead>
<tr>
<th>Category</th>
<th>Language Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financial</td>
<td>BLACKCLPRC Function (p. 396)</td>
<td>Calculates call prices for European options on futures, based on the Black model.</td>
</tr>
<tr>
<td></td>
<td>BLACKPTPRC Function (p. 399)</td>
<td>Calculates put prices for European options on futures, based on the Black model.</td>
</tr>
<tr>
<td></td>
<td>BLKSHCLPRC Function (p. 401)</td>
<td>Calculates call prices for European options on stocks, based on the Black-Scholes model.</td>
</tr>
<tr>
<td></td>
<td>BLKSHPTPRC Function (p. 403)</td>
<td>Calculates put prices for European options on stocks, based on the Black-Scholes model.</td>
</tr>
<tr>
<td></td>
<td>COMPOUND Function (p. 436)</td>
<td>Returns compound interest parameters.</td>
</tr>
<tr>
<td></td>
<td>CONVX Function (p. 444)</td>
<td>Returns the convexity for an enumerated cash flow.</td>
</tr>
<tr>
<td></td>
<td>CONVXP Function (p. 445)</td>
<td>Returns the convexity for a periodic cash flow stream, such as a bond.</td>
</tr>
<tr>
<td></td>
<td>CUMIPMT Function (p. 459)</td>
<td>Returns the cumulative interest paid on a loan between the start and end period.</td>
</tr>
<tr>
<td></td>
<td>CUMPRINC Function (p. 460)</td>
<td>Returns the cumulative principal paid on a loan between the start and end period.</td>
</tr>
<tr>
<td></td>
<td>DUR Function (p. 481)</td>
<td>Returns the modified duration for an enumerated cash flow.</td>
</tr>
<tr>
<td></td>
<td>DURP Function (p. 483)</td>
<td>Returns the modified duration for a periodic cash flow stream, such as a bond.</td>
</tr>
<tr>
<td></td>
<td>EFFRATE Function (p. 485)</td>
<td>Returns the effective annual interest rate.</td>
</tr>
<tr>
<td></td>
<td>GARKHCLPRC Function (p. 512)</td>
<td>Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.</td>
</tr>
<tr>
<td></td>
<td>GARKHPTPRC Function (p. 514)</td>
<td>Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.</td>
</tr>
<tr>
<td></td>
<td>INTRR Function (p. 571)</td>
<td>Returns the internal rate of return as a decimal value.</td>
</tr>
<tr>
<td></td>
<td>IPMT Function (p. 582)</td>
<td>Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.</td>
</tr>
<tr>
<td></td>
<td>IRR Function (p. 585)</td>
<td>Returns the internal rate of return as a percentage.</td>
</tr>
<tr>
<td></td>
<td>MARGRCLPRC Function (p. 616)</td>
<td>Calculates call prices for European options on stocks, based on the Margrabe model.</td>
</tr>
<tr>
<td></td>
<td>MARGRPTPRC Function (p. 618)</td>
<td>Calculates put prices for European options on stocks, based on the Margrabe model.</td>
</tr>
<tr>
<td></td>
<td>MORT Function (p. 635)</td>
<td>Returns amortization parameters.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>NETPV Function (p. 640)</strong></td>
<td>Returns the net present value as a percent.</td>
<td></td>
</tr>
<tr>
<td><strong>NOMRATE Function (p. 642)</strong></td>
<td>Returns the nominal annual interest rate.</td>
<td></td>
</tr>
<tr>
<td><strong>NPV Function (p. 669)</strong></td>
<td>Returns the net present value with the rate expressed as a percentage.</td>
<td></td>
</tr>
<tr>
<td><strong>PMT Function (p. 679)</strong></td>
<td>Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.</td>
<td></td>
</tr>
<tr>
<td><strong>PPMT Function (p. 682)</strong></td>
<td>Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.</td>
<td></td>
</tr>
<tr>
<td><strong>PVP Function (p. 727)</strong></td>
<td>Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.</td>
<td></td>
</tr>
<tr>
<td><strong>SAVINGS Function (p. 770)</strong></td>
<td>Returns the balance of a periodic savings by using variable interest rates.</td>
<td></td>
</tr>
<tr>
<td><strong>TIMEVALUE Function (p. 803)</strong></td>
<td>Returns the equivalent of a reference amount at a base date by using variable interest rates.</td>
<td></td>
</tr>
<tr>
<td><strong>YIELDP Function (p. 846)</strong></td>
<td>Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.</td>
<td></td>
</tr>
<tr>
<td><strong>Hyperbolic</strong></td>
<td><strong>ARCOSH Function (p. 387)</strong></td>
<td>Returns the inverse hyperbolic cosine.</td>
</tr>
<tr>
<td><strong>ARSINH Function (p. 389)</strong></td>
<td>Returns the inverse hyperbolic sine.</td>
<td></td>
</tr>
<tr>
<td><strong>ARTANH Function (p. 390)</strong></td>
<td>Returns the inverse hyperbolic tangent.</td>
<td></td>
</tr>
<tr>
<td><strong>Mathematical</strong></td>
<td><strong>ABS Function (p. 358)</strong></td>
<td>Returns the absolute value of a numeric value expression.</td>
</tr>
<tr>
<td><strong>BETA Function (p. 394)</strong></td>
<td>Returns the value of the beta function.</td>
<td></td>
</tr>
<tr>
<td><strong>COALESCE Function (p. 426)</strong></td>
<td>Returns the first non-null or nonmissing value from a list of numeric arguments.</td>
<td></td>
</tr>
<tr>
<td><strong>COMPFUZZ Function (p. 434)</strong></td>
<td>Performs a fuzzy comparison of two numeric values.</td>
<td></td>
</tr>
<tr>
<td><strong>CONSTANT Function (p. 439)</strong></td>
<td>Computes machine and mathematical constants.</td>
<td></td>
</tr>
<tr>
<td><strong>DEVIANCE Function (p. 472)</strong></td>
<td>Returns the deviance based on a probability distribution.</td>
<td></td>
</tr>
<tr>
<td><strong>DIGAMMA Function (p. 477)</strong></td>
<td>Returns the value of the digamma function.</td>
<td></td>
</tr>
<tr>
<td><strong>ERF Function (p. 486)</strong></td>
<td>Returns the value of the (normal) error function.</td>
<td></td>
</tr>
<tr>
<td><strong>ERFC Function (p. 487)</strong></td>
<td>Returns the value of the complementary (normal) error function.</td>
<td></td>
</tr>
<tr>
<td><strong>EXP Function (p. 488)</strong></td>
<td>Returns the value of the e constant raised to a specified power.</td>
<td></td>
</tr>
<tr>
<td><strong>FACT Function (p. 488)</strong></td>
<td>Computes a factorial.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>GAMMA Function (p. 511)</td>
<td>Returns the value of the gamma function.</td>
</tr>
<tr>
<td></td>
<td>GCD Function (p. 517)</td>
<td>Returns the greatest common divisor for a set of integers.</td>
</tr>
<tr>
<td></td>
<td>LCM Function (p. 601)</td>
<td>Returns the least common multiple for a set of integers.</td>
</tr>
<tr>
<td></td>
<td>LGAMMA Function (p. 608)</td>
<td>Returns the natural logarithm of the Gamma function.</td>
</tr>
<tr>
<td></td>
<td>LOG Function (p. 609)</td>
<td>Returns the natural logarithm (base e) of a numeric value expression.</td>
</tr>
<tr>
<td></td>
<td>LOGBETA Function (p. 610)</td>
<td>Returns the logarithm of the beta function.</td>
</tr>
<tr>
<td></td>
<td>LOG10 Function (p. 611)</td>
<td>Returns the base-10 logarithm of a numeric value expression.</td>
</tr>
<tr>
<td></td>
<td>LOG1PX Function (p. 612)</td>
<td>Returns the log of 1 plus the argument.</td>
</tr>
<tr>
<td></td>
<td>LOG2 Function (p. 613)</td>
<td>Returns the base 2 logarithm of a numeric value expression.</td>
</tr>
<tr>
<td></td>
<td>MOD Function (p. 631)</td>
<td>Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.</td>
</tr>
<tr>
<td></td>
<td>MODZ Function (p. 633)</td>
<td>Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.</td>
</tr>
<tr>
<td></td>
<td>POWER Function (p. 681)</td>
<td>Returns the value of a numeric value expression raised to a specified power.</td>
</tr>
<tr>
<td></td>
<td>SIGN Function (p. 779)</td>
<td>Returns a number that indicates the sign of a numeric value expression.</td>
</tr>
<tr>
<td></td>
<td>SQRT Function (p. 787)</td>
<td>Returns the square root of a value.</td>
</tr>
<tr>
<td></td>
<td>TRIGAMMA Function (p. 820)</td>
<td>Returns the value of the trigamma function.</td>
</tr>
<tr>
<td></td>
<td>WHICHN Function (p. 844)</td>
<td>Returns the first position of a number from a list of numbers.</td>
</tr>
<tr>
<td>Probability</td>
<td>POISSON Function (p. 680)</td>
<td>Returns the probability from a Poisson distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBBETA Function (p. 683)</td>
<td>Returns the probability from a beta distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBBNML Function (p. 684)</td>
<td>Returns the probability from a binomial distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBBNRM Function (p. 685)</td>
<td>Returns a probability from a bivariate normal distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBCHI Function (p. 686)</td>
<td>Returns the probability from a chi-square distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBDF Function (p. 687)</td>
<td>Calculates significance probabilities for Dickey-Fuller tests for unit roots in time series.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PROBF Function (p. 693)</td>
<td>Returns the probability from an F distribution.</td>
<td></td>
</tr>
<tr>
<td>PROBGAM Function (p. 694)</td>
<td>Returns the probability from a gamma distribution.</td>
<td></td>
</tr>
<tr>
<td>PROBHYPR Function (p. 695)</td>
<td>Returns the probability from a hypergeometric distribution.</td>
<td></td>
</tr>
<tr>
<td>PROBMC Function (p. 697)</td>
<td>Returns a probability or a quantile from various distributions for multiple comparisons of means.</td>
<td></td>
</tr>
<tr>
<td>PROBMED Function (p. 708)</td>
<td>Computes cumulative probabilities for the sample median.</td>
<td></td>
</tr>
<tr>
<td>PROBNEGB Function (p. 709)</td>
<td>Returns the probability from a negative binomial distribution.</td>
<td></td>
</tr>
<tr>
<td>PROBNORM Function (p. 710)</td>
<td>Returns the probability from the standard normal distribution.</td>
<td></td>
</tr>
<tr>
<td>PROBT Function (p. 711)</td>
<td>Returns the probability from a t distribution.</td>
<td></td>
</tr>
<tr>
<td>Quantile</td>
<td>BETAINV Function (p. 395)</td>
<td>Returns a quantile from the beta distribution.</td>
</tr>
<tr>
<td></td>
<td>GAMINV Function (p. 510)</td>
<td>Returns a quantile from the gamma distribution.</td>
</tr>
<tr>
<td></td>
<td>PROBIT Function (p. 696)</td>
<td>Returns a quantile from the standard normal distribution.</td>
</tr>
<tr>
<td></td>
<td>TINV Function (p. 805)</td>
<td>Returns a quantile from the t distribution.</td>
</tr>
<tr>
<td>Random Number</td>
<td>RANBIN Function (p. 731)</td>
<td>Returns a random variate from a binomial distribution.</td>
</tr>
<tr>
<td></td>
<td>RANCAU Function (p. 732)</td>
<td>Returns a random variate from a Cauchy distribution.</td>
</tr>
<tr>
<td></td>
<td>RAND Function (p. 733)</td>
<td>Generates pseudo-random numbers from a distribution that you specify.</td>
</tr>
<tr>
<td></td>
<td>RANEXP Function (p. 744)</td>
<td>Returns a random variate from an exponential distribution.</td>
</tr>
<tr>
<td></td>
<td>RANGAM Function (p. 745)</td>
<td>Returns a random variate from a gamma distribution.</td>
</tr>
<tr>
<td></td>
<td>RANNOR Function (p. 748)</td>
<td>Returns a random variate from a normal distribution.</td>
</tr>
<tr>
<td></td>
<td>RANPOI Function (p. 749)</td>
<td>Returns a random variate from a Poisson distribution.</td>
</tr>
<tr>
<td></td>
<td>RANTBL Function (p. 750)</td>
<td>Returns a random variate from a tabled probability distribution.</td>
</tr>
<tr>
<td></td>
<td>RANTRI Function (p. 751)</td>
<td>Returns a random variate from a triangular distribution.</td>
</tr>
<tr>
<td></td>
<td>RANUNI Function (p. 752)</td>
<td>Returns a random variate from a uniform distribution.</td>
</tr>
<tr>
<td></td>
<td>STREAMINIT Function (p. 789)</td>
<td>Specifies a seed value to use for subsequent pseudo-random number generation by the RAND function.</td>
</tr>
<tr>
<td></td>
<td>UNIFORM Function (p. 824)</td>
<td>Returns a random variate from a uniform distribution.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Special</td>
<td>INPUTC Function (p. 538)</td>
<td>Enables you to specify a character informat at run time.</td>
</tr>
<tr>
<td></td>
<td>INPUTN Function (p. 539)</td>
<td>Enables you to specify a numeric informat at run time.</td>
</tr>
<tr>
<td></td>
<td>MISSING Function (p. 629)</td>
<td>Returns a number that indicates whether the argument contains a missing value.</td>
</tr>
<tr>
<td></td>
<td>NULL Function (p. 670)</td>
<td>Returns a 1 if the argument is null and a 0 if the argument is not null.</td>
</tr>
<tr>
<td></td>
<td>PUT Function (p. 725)</td>
<td>Returns a value using a specified format.</td>
</tr>
<tr>
<td></td>
<td>SLEEP Function (p. 783)</td>
<td>For a specified period of time, suspends the execution of a program that invokes this function.</td>
</tr>
<tr>
<td></td>
<td>SQLEXEC Function (p. 786)</td>
<td>Executes a FedSQL statement to create, delete, or update a table or to insert rows into a table.</td>
</tr>
<tr>
<td></td>
<td>UUIDGEN Function (p. 826)</td>
<td>Returns the short form of a Universally unique identifier (UUID).</td>
</tr>
<tr>
<td>Trigonometric</td>
<td>ARCOS Function (p. 386)</td>
<td>Returns the arccosine in radians.</td>
</tr>
<tr>
<td></td>
<td>ARSIN Function (p. 388)</td>
<td>Returns the arcsine in radians.</td>
</tr>
<tr>
<td></td>
<td>ATAN Function (p. 391)</td>
<td>Returns the arctangent in radians.</td>
</tr>
<tr>
<td></td>
<td>ATAN2 Function (p. 392)</td>
<td>Returns the arctangent of the x and y coordinates of a right triangle, in radians.</td>
</tr>
<tr>
<td></td>
<td>COS Function (p. 447)</td>
<td>Returns the cosine in radians.</td>
</tr>
<tr>
<td></td>
<td>COSH Function (p. 448)</td>
<td>Returns the hyperbolic cosine in radians.</td>
</tr>
<tr>
<td></td>
<td>SEC Function (p. 774)</td>
<td>Returns the secant.</td>
</tr>
<tr>
<td></td>
<td>SIN Function (p. 780)</td>
<td>Returns the trigonometric sine.</td>
</tr>
<tr>
<td></td>
<td>SINH Function (p. 781)</td>
<td>Returns the hyperbolic sine.</td>
</tr>
<tr>
<td></td>
<td>TAN Function (p. 800)</td>
<td>Returns the tangent.</td>
</tr>
<tr>
<td></td>
<td>TANH Function (p. 801)</td>
<td>Returns the hyperbolic tangent.</td>
</tr>
<tr>
<td>Truncation</td>
<td>CEIL Function (p. 419)</td>
<td>Returns the smallest integer greater than or equal to a numeric value expression.</td>
</tr>
<tr>
<td></td>
<td>CEILZ Function (p. 420)</td>
<td>Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.</td>
</tr>
<tr>
<td></td>
<td>FLOOR Function (p. 505)</td>
<td>Returns the largest integer less than or equal to a numeric value expression.</td>
</tr>
</tbody>
</table>
### Category | Language Elements | Description
--- | --- | ---
FLOORZ Function (p. 506) | Returns the largest integer that is less than or equal to the argument, using zero fuzzing. |
FUZZ Function (p. 509) | Returns the nearest integer if the argument is within 1E-12 of that integer. |
INT Function (p. 541) | Returns the integer value, fuzzed to avoid unexpected floating-point results. |
INTZ Function (p. 581) | Returns the integer portion of the argument, using zero fuzzing. |
ROUND Function (p. 756) | Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted. |
ROUNDE Function (p. 765) | Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples. |
ROUNDZ Function (p. 767) | Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing. |
TRUNC Function (p. 823) | Truncates a numeric value to a specified length. |
VFORMAT Function (p. 829) | Returns the format that is associated with the specified variable. |
VINARRAY Function (p. 831) | Returns a value that indicates whether the specified variable is a member of an array. |
VINFORMAT Function (p. 832) | Returns the informat that is associated with the specified variable. |
VLABEL Function (p. 833) | Returns the label that is associated with the specified variable. |
VLENGTH Function (p. 835) | Returns the size of the specified variable. |
VNAME Function (p. 836) | Returns the name of the specified variable. |
VTYPE Function (p. 837) | Returns the full name of the data type that is associated with a variable. |

### Dictionary

**ABS Function**

Returns the absolute value of a numeric value expression.

**Category:** Mathematical
**Returned data type:** BIGINT, DECIMAL, DOUBLE, NUMERIC

**Syntax**

\[
\text{ABS}(\text{expression})
\]

**Arguments**

\[\text{expression}\]

specifies any valid expression that evaluates to a numeric value.

**Data type** BIGINT, DECIMAL, DOUBLE, NUMERIC

**See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

If the result is a number that does not fit into the range of the argument's data type, the ABS function fails.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

**Example**

The following statements illustrate the ABS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcl int x; x=abs(-345);</td>
<td>345</td>
</tr>
<tr>
<td>dcl double z; z=abs((3 x 50.5) / 5);</td>
<td>30.3</td>
</tr>
</tbody>
</table>

**ANYALNUM Function**

Searches a character string for an alphanumerical character, and returns the first position at which the character is found.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

\[
\text{ANYALNUM}(\text{expression}[\text{, start}])
\]
### Arguments

**expression**

- Specifies any valid expression that evaluates to a character string.

  **Data type** CHAR, NCHAR

**start**

- Specifies the position at which the search should start and the direction in which to search.

  **Data type** INTEGER

### Details

The results of the ANYALNUM function depend directly on the translation table that is in effect (see “**TRANTAB= System Option**” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the **ENCODING** and the **LOCALE** system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found, ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, **start**, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of **start** is positive, the search proceeds to the right.
- If the value of **start** is negative, the search proceeds to the left.
- If the value of **start** is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of **start** is greater than the length of the string.
- The value of **start** = 0.

### Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

### Examples

**Example 1: Scanning a String from Left to Right**

The following example uses the ANYALNUM function to search a string from left to right for alphanumeric characters.

```sas
data ltrtest;
  dcl char(15) string c;
  dcl double j;
```
method run();
    string='Next = Last + 1';
    j=0;
    do until (j=0);
        j=anyalnum(string, j+1);
        if j=0 then put 'The end';
        else do;
            c=substr(string, j, 1);
            put j= c=;
        end;
    end;
end;
enddata;
run;

SAS writes the following output to the log:

j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=L
j=9 c=a
j=10 c=s
j=11 c=t
j=15 c=1
The end

Example 2: Scanning a String from Right to Left
The following example uses the ANYALNUM function to search a string from right to left for alphanumeric characters.

data rtltest;
    dcl char(15) string c;
    dcl double j;
    method run();
        string='Next = Last + 1';
        j=9999999;
        do until(j=0);
            j=anyalnum(string, 1-j);
            if j=0 then put 'The end';
            else do;
                c=substr(string, j, 1);
                put j= c=;
            end;
        end;
    end;
enddata;
run;
SAS writes the following output to the log:

```
j=15 c=l
j=11 c=t
j=10 c=s
j=9 c=a
j=8 c=L
j=4 c=t
j=3 c=x
j=2 c=e
j=1 c=N
The end
```

See Also

Functions:

• “NOTALNUM Function” on page 644

ANYALPHA Function

Searches a character string for an alphabetic character, and returns the first position at which the character is found.

- **Category:** Character
- **Returned data type:** DOUBLE

**Syntax**

```
ANYALPHA('expression'[, start])
```

**Arguments**

- **expression** specifies any valid expression that evaluates to a character string.
  - **Data type:** CHAR, NCHAR
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **start** specifies the position at which the search should start and the direction in which to search.
  - **Data type:** INTEGER

**Details**

The results of the ANYALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the
position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \textit{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

ANYALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of \textit{start} is greater than the length of the string.
- The value of \textit{start} = 0.

\textbf{Comparisons}

The ANYALPHA function searches a character string for an alphabetic character. The NOTALPHA function searches a character string for a non-alphabetic character.

\textbf{Examples}

\textit{Example 1: Searching a String for Alphabetic Characters}

The following example uses the ANYALPHA function to search a string from left to right for alphabetic characters.

```plaintext
data _null_
   dcl char(18) string c;
   dcl double j i;
   method run();
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyalpha(string, j+1);
      if j=0 then put 'The end';
      else do;
         c=substr(string, j, 1);
         put j= c=;
      end;
   end;
end;
enddata;
run;
```
SAS writes the following output to the log:

```
  j=1 c=N
  j=2 c=e
  j=3 c=x
  j=4 c=t
  j=9 c=n
  j=16 c=E
  The end
```

**Example 2: Identifying Control Characters By Using the ANYALPHA Function**

You can execute the following program to show the control characters that are identified by the ANYALPHA function.

```sql
data testany;
  dcl nchar(3) byte1 hex1;
  dcl double dec anyalpha1;

  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec,hex2.);
      anyalpha1=anyalpha(byte1);
      output;
    end;
  end;
enddata;
run;

proc print data=testany;
run;
```

**See Also**

**Functions:**
- “NOTALPHA Function” on page 645

---

**ANYCNTRL Function**

Searches a character string for a control character, and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

```
ANYCNTRL('expression' [, start])
```
Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type  CHAR, NCHAR

See  Chapter 13, “DS2 Expressions,” on page 93

start

specifies the position at which the search should start and the direction in which to search.

Data type  INTEGER

Details

The results of the ANYCNTRL function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The ANYCNTRL function searches a string for the first occurrence of a control character. If such a character is found, ANYCNTRL returns the position in the string of that character. If no such character is found, ANYCNTRL returns a value of 0.

If you use only one argument, ANYCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

• If the value of start is positive, the search proceeds to the right.
• If the value of start is negative, the search proceeds to the left.
• If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYCNTRL returns a value of zero when one of the following is true:

• The character that you are searching for is not found.
• The value of start is greater than the length of the string.
• The value of start = 0.

Comparisons

The ANYCNTRL function searches a character string for a control character. The NOTCNTRL function searches a character string for a character that is not a control character.

Example

You can execute the following program to show the control characters that are identified by the ANYCNTRL function.

```sas
data testany);
  dcl nchar(3) byte1 hex1;
  dcl double dec anycntrl1;

  method run();
```
do dec=0 to 255;
    byte1=byte(dec);
    hex1=put(dec,hex2.);
    anycntrl1=anycntrl(byte1);
    if anycntrl1 then output;
    end;
end;
enddata;
run;

proc print data=testany;
run;
quit;

See Also

Functions:
- “NOTCNTRL Function” on page 648

ANYDIGIT Function

Searches a character string for a digit, and returns the first position at which the digit is found.

<table>
<thead>
<tr>
<th>Category</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

ANYDIGIT('expression'[, start])

Arguments

expression
  specifies any valid expression that evaluates to a character string.
  
  Data type  CHAR, NCHAR
  See  Chapter 13, “DS2 Expressions,” on page 93

start
  specifies the position at which the search should start and the direction in which to search.
  
  Data type  INTEGER

Details

The ANYDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.
The ANYDIGIT function searches a string for the first occurrence of any character that is a digit. If such a character is found, ANYDIGIT returns the position in the string of that character. If no such character is found, ANYDIGIT returns a value of 0.

If you use only one argument, ANYDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
- The value of start = 0.

**Comparisons**

The ANYDIGIT function searches a character string for a digit. The NOTDIGIT function searches a character string for any character that is not a digit.

**Example**

The following example uses the ANYDIGIT function to search for a character that is a digit.

```
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anydigit(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
j=14 c=1  
j=15 c=2  
j=17 c=3  
The end
```
ANYFIRST Function

Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character
Returned data type: DOUBLE

Syntax

ANYFIRST('expression[, start]

Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

start
specifies the position at which the search should start and the direction in which to search.

Data type INTEGER

Details

The ANYFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_) and uppercase or lowercase English letters. If such a character is found, ANYFIRST returns the position in the string of that character. If no such character is found, ANYFIRST returns a value of 0.

If you use only one argument, ANYFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.
ANYFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of \texttt{start} is greater than the length of the string.
- The value of \texttt{start} = 0.

**Comparisons**

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

**Example**

The following example uses the ANYFIRST function to search a string for any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

```sas
data _null_; 
  dcl char(18) string c; 
  dcl double j i; 
  method run(); 
  string='Next = _n_ + 12E3;'; 
  j=0; 
  do until(j=0); 
    j=anyfirst(string, j+1); 
    if j=0 then put 'The end'; 
    else do; 
      c=substr(string, j, 1); 
      put j= c=; 
    end; 
  end; 
enddata; 
run;
```

SAS writes the following output to the log:

```
j=1 c=N 
j=2 c=e 
j=3 c=x 
j=4 c=t 
j=8 c=_ 
j=9 c=n 
j=10 c=_ 
j=16 c=E 
The end
```

**See Also**

**Functions:**

- “NOTFIRST Function” on page 651
ANYGRAPH Function

Searches a character string for a graphical character, and returns the first position at which that character is found.

**Category:** Character

**Returned data type:** DOUBLE

### Syntax

\[
\text{ANYGRAPH}(\text{string}[, \text{start}])
\]

### Arguments

**expression**

specifies any valid expression that evaluates to a character string.

**Data type** CHAR, NCHAR

**See** Chapter 13, “DS2 Expressions,” on page 93

**start**

specifies the position at which the search should start and the direction in which to search.

**Data type** INTEGER

### Details

The results of the ANYGRAPH function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, ANYGRAPH returns the position in the string of that character. If no such character is found, ANYGRAPH returns a value of 0.

If you use only one argument, ANYGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \( \text{start} \), specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \( \text{start} \) is positive, the search proceeds to the right.
- If the value of \( \text{start} \) is negative, the search proceeds to the left.
- If the value of \( \text{start} \) is less than the negative length of the string, the search begins at the end of the string.

ANYGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of \( \text{start} \) is greater than the length of the string.
• The value of \( start = 0 \).

**Comparisons**

The ANYGRAPH function searches a character string for a graphical character. The NOTGRAPH function searches a character string for a non-graphical character.

**Examples**

**Example 1: Searching a String for Graphical Characters**
The following example uses the ANYGRAPH function to search a string for graphical characters.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anygraph(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
  enddata;  
run;  
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c==
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

**Example 2: Identifying Control Characters By Using the ANYGRAPH Function**
You can execute the following program to show the control characters that are identified by the ANYGRAPH function.

```sas
data testany (overwrite=yes);  
dcl nchar(3) byte1 hex1;  
dcl double dec anygraph1;  
```
method run();
   do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec,hex2.);
      anygraph=anygraph(byte1);
      output;
   end;
end;
enddata;
run;

proc print data=testany;
run;
quit;

See Also

Functions:
- “NOTGRAPH Function” on page 653

ANYLOWER Function

Searches a character string for a lowercase letter, and returns the first position at which the letter is found.

Category: Character

Returned data type: DOUBLE

Syntax

ANYLOWER(‘expression’, [start])

Arguments

expression
   specifies any valid expression that evaluates to a character string.

Data type      CHAR, NCHAR

See
   Chapter 13, “DS2 Expressions,” on page 93

start
   specifies the position at which the search should start and the direction in which to search.

Data type      INTEGER
Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The ANYLOWER function searches a string for the first occurrence of a lowercase letter. If such a character is found, ANYLOWER returns the position in the string of that character. If no such character is found, ANYLOWER returns a value of 0.

If you use only one argument, ANYLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
- The value of start = 0.

Comparisons

The ANYLOWER function searches a character string for a lowercase letter. The NOTLOWER function searches a character string for a character that is not a lowercase letter.

Example

The following example uses the ANYLOWER function to search a string for any character that is a lowercase letter.

```sas
data _null_;
  dcl char(18) string c;
  dcl double j i;
  method run();
    string='Next = _n_ + 12E3;';
    j=0;
    do until(j=0);
      j=anylower(string, j+1);
      if j=0 then put 'The end';
      else do;
        c=substr(string, j, 1);
        put j= c=;
      end;
    end;
  end;
enddata;
run;
```
SAS writes the following output to the log:

```
j=2 c=e
j=3 c=x
j=4 c=t
j=9 c=n
The end
```

See Also

Functions:
- “NOTLOWER Function” on page 655

### ANYNAME Function

Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

**Category:** Character  
**Returned data type:** DOUBLE

#### Syntax

\[ \text{ANYNAME('expression'[,$start])] \]

#### Arguments

**expression**  
specifies any valid expression that evaluates to a character string.  

**Data type**  
CHAR, NCHAR  

**start**  
specifies the position at which the search should start and the direction in which to search.  

**Data type**  
INTEGER

#### Details

The ANYNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, ANYNAME returns the position in the string of that character. If no such character is found, ANYNAME returns a value of 0.
If you use only one argument, ANYNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

ANYNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of `start` is greater than the length of the string.
- The value of `start` = 0.

**Comparisons**

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under `VALIDVARNAME=V7`. The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under `VALIDVARNAME=V7`.

**Example**

The following example uses the ANYNAME function to search a string for any character that is valid in a SAS variable name under `VALIDVARNAME=V7`.

```sas
data _null_;
  dcl char(18) string c;
  dcl double j i;
  method run();
    string='Next = _n_ + 12E3;';
    j=0;
    do until(j=0);
      j=anyname(string, j+1);
      if j=0 then put 'The end';
      else do;
        c=substr(string, j, 1);
        put j= c=;
      end;
    end;
  end;
enddata;
run;
```
SAS writes the following output to the log:

```
j=1  c=N
j=2  c=e
j=3  c=x
j=4  c=t
j=8  c=_
j=9  c=n
j=10 c=_
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
```

The end

See Also

Functions:

- “NOTNAME Function” on page 657

---

**ANYPRINT Function**

Searches a character string for a printable character, and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

```
ANYPRINT('expression'[, start])
```

**Arguments**

- **expression**
  - specifies any valid expression that evaluates to a character string.
  - Data type: CHAR, NCHAR
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **start**
  - specifies the position at which the search should start and the direction in which to search.
  - Data type: INTEGER

**Details**

The results of the ANYPRINT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.
The ANYPRINT function searches a string for the first occurrence of a printable character. If such a character is found, ANYPRINT returns the position in the string of that character. If no such character is found, ANYPRINT returns a value of 0.

If you use only one argument, ANYPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
- The value of start = 0.

**Comparisons**

The ANYPRINT function searches a character string for a printable character. The NOTPRINT function searches a character string for a non-printable character.

**Examples**

**Example 1: Searching a String for a Printable Character**

The following example uses the ANYPRINT function to search a string for printable characters.

```sql
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anyprint(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
end;  
enddata;  
run;
```
SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

**Example 2: Identifying Control Characters By Using the ANYPRINT Function**

You can execute the following program to show the control characters that are identified by the ANYPRINT function.

```sas
data testany;
  dcl nchar(3) byte1 hex1;
  dcl double dec anyprint1;
  
  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec,hex2.);
      anyprint1=anyprint(byte1);
      output;
    end;
  end;
enddata;
run;

proc print data=testany;
run;
quit;
```

**See Also**

**Functions:**

- "NOTPRINT Function" on page 659

**ANYPUNCT Function**

Searches a character string for a punctuation character, and returns the first position at which that character is found.
Syntax

\texttt{ANYPUNCT('expression[, start])}

Arguments

\textit{expression}

specifies any valid expression that evaluates to a character string.

\begin{itemize}
  \item \textbf{Data type}: CHAR, NCHAR
  \item \textbf{See}: Chapter 13, “DS2 Expressions,” on page 93
\end{itemize}

\textit{start}

specifies the position at which the search should start and the direction in which to search.

\begin{itemize}
  \item \textbf{Data type}: INTEGER
\end{itemize}

Details

The results of the \texttt{ANYPUNCT} function depend directly on the translation table that is in effect (see “\texttt{TRANTAB=} System Option” in \textit{SAS National Language Support (NLS): Reference Guide}) and indirectly on the \texttt{ENCODING} and the \texttt{LOCALE} system options.

The \texttt{ANYPUNCT} function searches a string for the first occurrence of a punctuation character. If such a character is found, \texttt{ANYPUNCT} returns the position in the string of that character. If no such character is found, \texttt{ANYPUNCT} returns a value of 0.

If you use only one argument, \texttt{ANYPUNCT} begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \texttt{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

\begin{itemize}
  \item If the value of \texttt{start} is positive, the search proceeds to the right.
  \item If the value of \texttt{start} is negative, the search proceeds to the left.
  \item If the value of \texttt{start} is less than the negative length of the string, the search begins at the end of the string.
\end{itemize}

\texttt{ANYPUNCT} returns a value of zero when one of the following is true:

\begin{itemize}
  \item The character that you are searching for is not found.
  \item The value of \texttt{start} is greater than the length of the string.
  \item The value of \texttt{start} = 0.
\end{itemize}

Comparisons

The \texttt{ANYPUNCT} function searches a character string for a punctuation character. The \texttt{NOTPUNCT} function searches a character string for a character that is not a punctuation character.
Examples

**Example 1: Searching a String for Punctuation Characters**
The following example uses the ANYPUNCT function to search a string for punctuation characters.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anypunct(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
end;  
enddata;
run;
```

SAS writes the following output to the log:

```
j=8 c=_
j=10 c=_
j=18 c=;
The end
```

**Example 2: Identifying Control Characters By Using the ANYPUNCT Function**
You can execute the following program to show the control characters that are identified by the ANYPUNCT function.

```sas
data testany (overwrite=yes);  
dcl nchar(3) byte1 hex1;  
dcl double dec anypunct1;  
method run();  
  do dec=0 to 255;  
    byte1=byte(dec);  
    hex1=put(dec,hex2.);  
    anypunct1=anypunct(byte1);  
    output;  
  end;  
end;  
enddata;  
run;
```

See Also

Functions:
- “NOTPUNCT Function” on page 661
ANYSPACE Function

Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

\[ \text{ANYSPACE('expression'[, start])} \]

Arguments

\textit{expression}

specifies any valid expression that evaluates to a character string.

Data type \[ \text{CHAR, NCHAR} \]

See \[ \text{Chapter 13, “DS2 Expressions,” on page 93} \]

\textit{start}

specifies the position at which the search should start and the direction in which to search.

Data type \[ \text{INTEGER} \]

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see “\texttt{TRANTAB= System Option}” in \textit{SAS National Language Support (NLS): Reference Guide}) and indirectly on the \texttt{ENCODING} and the \texttt{LOCALE} system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \textit{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of \textit{start} is greater than the length of the string.
Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

Example 1: Searching a String for a Whitespace Character

The following example uses the ANYSPACE function to search a string for a character that is a whitespace character.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anyspace(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
j=5 c=  
j=7 c=  
j=11 c=  
j=13 c=  
The end
```

Example 2: Identifying Control Characters By Using the ANYSPACE Function

You can execute the following program to show the control characters that are identified by the ANYSPACE function.

```sas
data testany (overwrite=yes);  
dcl nchar(3) byte1 hex1;  
dcl double dec anyspace1;  
method run();  
  do dec=0 to 255;  
    byte1=byte(dec);  
    hex1=put(dec,hex2.);  
    anyspace1=anyspace(byte1);  
  end;  
end;  
run;  
```
ANYUPPER Function

Searches a character string for an uppercase letter, and returns the first position at which the letter is found.

**Category:** Character  
**Returned data type:** DOUBLE

**Syntax**

```
ANYUPPER('expression', start)
```

**Arguments**

*expression*

specifies any valid expression that evaluates to a character string.

**Data type**  CHAR, NCHAR

**Start**

specifies the position at which the search should start and the direction in which to search.

**Data type**  INTEGER

**Details**

The results of the ANYUPPER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:
• If the value of \textit{start} is positive, the search proceeds to the right.
• If the value of \textit{start} is negative, the search proceeds to the left.
• If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when one of the following is true:

• The character that you are searching for is not found.
• The value of \textit{start} is greater than the length of the string.
• The value of \textit{start} = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Example

The following example uses the ANYUPPER function to search a string for an uppercase letter.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anyupper(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
j=1  c=N  
j=16  c=E  
The end
```

See Also

Functions:

• “NOTUPPER Function” on page 666
ANYXDIGIT Function

Searches a character string for a hexadecimal character that represents a digit, and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

ANYXDIGIT('expression'[start])

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

start

specifies the position at which the search should start and the direction in which to search.

Data type INTEGER

Details

The ANYXDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYXDIGIT function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, ANYXDIGIT returns the position in the string of that character. If no such character is found, ANYXDIGIT returns a value of 0.

If you use only one argument, ANYXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
- The value of start = 0.
Comparisons

The ANYXDIGIT function searches a character string for a character that is a hexadecimal character. The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character.

Example

The following example uses the ANYXDIGIT function to search a string for a hexadecimal character that represents a digit.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anyxdigit(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
j=2 c=e  
j=14 c=1  
j=15 c=2  
j=16 c=E  
j=17 c=3  
The end
```

See Also

Functions:

- “NOTXDIGIT Function” on page 668

ARCOS Function

Returns the arccosine in radians.

- **Category:** Trigonometric
- **Returned data type:** DOUBLE
Syntax

ARCOS(expression)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range  

-1 to 1

Data type  

DOUBLE

See  

Chapter 13, “DS2 Expressions,” on page 93

Details

The ARCOS function returns the arccosine (inverse cosine) of the argument. The value that is returned is specified in radians.

Example

The following statements illustrate the ARCOS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=arcos(1);</td>
<td>0</td>
</tr>
<tr>
<td>x=arcos(0);</td>
<td>1.57079632679489</td>
</tr>
<tr>
<td>x=arcos(-0.5);</td>
<td>2.09439510239319</td>
</tr>
</tbody>
</table>

ARCOSH Function

Returns the inverse hyperbolic cosine.

Category: Hyperbolic

Returned data type: DOUBLE

Syntax

ARCOSH(expression)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range  

expression >= 1
Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The ARCOSH function computes the inverse hyperbolic cosine. The ARCOSH function is mathematically defined by the following equation, where expression $\geq 1$. In the equation, expression is represented by $x$.

$$ARCOSH(x) = \log\left(x + \sqrt{x^2 - 1}\right)$$

Example

The following statements illustrate the ARCOSH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=arcosh(5);</td>
<td>2.29243166956117</td>
</tr>
<tr>
<td>x=arcosh(13);</td>
<td>3.25661395480005</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “ARSINH Function” on page 389
- “ARTANH Function” on page 390
- “COSH Function” on page 448
- “TANH Function” on page 801
- “SINH Function” on page 781

ARSIN Function

Returns the arcsine in radians.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

ARSIN(expression)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.
Details
The ARSIN function returns the arcsine (inverse sine) of the argument. The value that is returned is specified in radians.

Example
The following statements illustrate the ARSIN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=arsin(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=arsin(1);</td>
<td>1.57079632679489</td>
</tr>
<tr>
<td>x= arsin(-0.5);</td>
<td>-0.52359877559829</td>
</tr>
</tbody>
</table>

ARSINH Function
Returns the inverse hyperbolic sine.

Category: Hyperbolic
Returned data type: DOUBLE

Syntax
ARSINH(expression)

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>$-\infty &lt; x &lt; \infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

Details
The ARSINH function computes the inverse hyperbolic sine. The ARSINH function is mathematically defined by the following equation, where $-\infty < x < \infty$. 
\[ \text{ARSINH}(x) = \log\left(x + \sqrt{x^2 + 1}\right) \]

Replace the infinity symbol with the largest double precision number that is available on your machine. In the equation, \( \text{expression} \) is represented by \( x \).

**Example**

The following statements illustrate the ARSINH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{arsinh}(5) );</td>
<td>2.31243834127275</td>
</tr>
<tr>
<td>( x = \text{arsinh}(-5) );</td>
<td>-2.31243834127275</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**
- “ARCOSH Function” on page 387
- “ARTANH Function” on page 390
- “COSH Function” on page 448
- “TANH Function” on page 801
- “SINH Function” on page 781

---

**ARTANH Function**

Returns the inverse hyperbolic tangent.

**Category:** Hyperbolic  
**Returned data type:** DOUBLE

**Syntax**

\[ \text{ARTANH}(\text{expression}) \]

**Arguments**

\( \text{expression} \)  
 specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>(-1 &lt; \text{expression} &lt; 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>
Details

The ARTANH function computes the inverse hyperbolic tangent. The ARTANH function is mathematically defined by the following equation, where \(-1 < expression < 1\).

In the equation, \(expression\) is represented by \(x\).

\[
\text{ARTANH}(x) = \frac{1}{2} \log \left( \frac{1 + x}{1 - x} \right)
\]

Example

The following statements illustrate the ARTANH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcl double x; x=artanh(.5);</td>
<td>0.54930614433405</td>
</tr>
<tr>
<td>dcl double x1; x=artanh(-.5);</td>
<td>-0.54930614433405</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “ARCOSH Function” on page 387
- “ARSINH Function” on page 389
- “COSH Function” on page 448
- “TANH Function” on page 801
- “SINH Function” on page 781

ATAN Function

Returns the arctangent in radians.

**Category:** Trigonometric

**Alias:** ARTAN

**Returned data type:** DOUBLE

**Syntax**

\[
\text{ATAN}(expression)
\]

**Arguments**

\(expression\)

specifies any valid expression that evaluates to a numeric value.
### Details

The ATAN function returns the 2-quadrant arctangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is \(x\) and whose value ranges from \(-\pi/2\) to \(\pi/2\).

### Comparisons

The ATAN function is similar to the ATAN2 function except that ATAN2 calculates the arctangent of the angle from the ratio of two arguments rather than from one argument.

### Example

The following statements illustrate the ATAN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x=\text{atan}(0);)</td>
<td>0</td>
</tr>
<tr>
<td>(x=\text{atan}(1);)</td>
<td>0.78539816339744</td>
</tr>
<tr>
<td>(x=\text{atan}(-9.0);)</td>
<td>-1.460139105621</td>
</tr>
</tbody>
</table>

### See Also

Functions:
- “ATAN2 Function” on page 392

---

**ATAN2 Function**

Returns the arctangent of the \(x\) and \(y\) coordinates of a right triangle, in radians.

- **Category:** Trigonometric
- **Returned data type:** DOUBLE

### Syntax

\[\text{ATAN2}(\text{expression-1}, \text{expression-2})\]

### Arguments

- **expression-1**
  - specifies any valid expression that evaluates to a numeric value.
- **expression-2**
  - specifies the \(x\) coordinate of the end of the hypotenuse of a right triangle.
**expression-2**

specifies any valid expression that evaluates to a numeric value. `expression-2` specifies the y coordinate of the end of the hypotenuse of a right triangle.

**Details**

The ATAN2 function returns the arctangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arc tangent of `expression-1 / expression-2`, except that the signs of both arguments are used to determine the quadrant of the result.

**Comparisons**

The ATAN2 function is similar to the ATAN function except that ATAN calculates the arctangent of the angle from the value of one argument rather than from two arguments.

**Example**

The following statements illustrate the ATAN2 function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=atan2(-1, 0.5);</code></td>
<td>-1.10714871779409</td>
</tr>
<tr>
<td><code>x=atan2(6,8);</code></td>
<td>0.643501108793828</td>
</tr>
<tr>
<td><code>x=atan2(5,-3);</code></td>
<td>2.11121582786548</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81

**Functions:**

- “ATAN Function” on page 391

---

**BAND Function**

Returns the bitwise logical AND of two arguments.

**Category:** Bitwise Logical Operations

**Returned data type:** DOUBLE
Syntax

\texttt{BAND(expression-1, expression-2)}

\textbf{Arguments}

\texttt{expression-1, expression-2}

specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>between 0 and ((2^{32})-1) inclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>\texttt{DOUBLE}</td>
</tr>
</tbody>
</table>

See Chapter 13, “DS2 Expressions,” on page 93

\textbf{Example}

The following statements illustrate the \texttt{BAND} function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x=band(9,11);}</td>
<td>9</td>
</tr>
<tr>
<td>\texttt{x=band(15,5);}</td>
<td>5</td>
</tr>
</tbody>
</table>

\textbf{BETA Function}

Returns the value of the beta function.

\textbf{Category:} Mathematical

\textbf{Returned data type:} \texttt{DOUBLE}

\textbf{Syntax}

\texttt{BETA(a, b)}

\textbf{Arguments}

\texttt{a}

is the first shape parameter.

<table>
<thead>
<tr>
<th>Range</th>
<th>(a &gt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>\texttt{DOUBLE}</td>
</tr>
</tbody>
</table>

\texttt{b}

is the second shape parameter.

| Range | \(b > 0\) |
Data type: DOUBLE

Details

The BETA function is mathematically given by this equation:

\[ \beta(a, b) = \int_{0}^{1} x^{a-1}(1-x)^{b-1} dx \]

Note the following:

\[ \beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \]

In the previous equation, \( \Gamma(.) \) is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following statements illustrate the BETA function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=beta(5,3);</td>
<td>0.0095238095238</td>
</tr>
<tr>
<td>x=beta(15,45);</td>
<td>1.6710294365008E-15</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “LOGBETA Function” on page 610

BETAINV Function

Returns a quantile from the beta distribution.

Category: Quantile

Returned data type: DOUBLE

Syntax

\[ \text{BETAINV}(p, a, b) \]

Arguments

\( p \)

is a numeric probability.
Range \( 0 \leq p \leq 1 \)

Data type DOUBLE

\( a \)

is a numeric shape parameter.

Range \( a > 0 \)

Data type DOUBLE

\( b \)

is a numeric shape parameter.

Range \( b > 0 \)

Data type DOUBLE

Details

The BETAINV function returns the \( p \)th quantile from the beta distribution with shape parameters \( a \) and \( b \). The probability that an observation from a beta distribution is less than or equal to the returned quantile is \( p \).

Note: BETAINV is the inverse of the PROBBETA function.

Example

The following example illustrates the BETAINV function.

data test (overwrite=yes);
  dcl double y z;
  method run();
    y=betainv(0.001, 2, 4);
    put 'y=' y;
  end;
enddata;
run;

The following line is written to the SAS log.

\[ y = 0.01010178788373 \]

See Also

Functions:

- “PROBBETA Function” on page 683

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Category: Financial
**Returned data type:** DOUBLE

**Syntax**

\[ \text{BLACKCLPRC}(E, t, F, r, \sigma) \]

**Arguments**

- \( E \)
  
  is a nonmissing, positive value that specifies exercise price.
  
  Requirement Specify \( E \) and \( F \) in the same units.
  
  Data type DOUBLE

- \( t \)
  
  is a nonmissing value that specifies time to maturity, in years.
  
  Data type DOUBLE

- \( F \)
  
  is a nonmissing, positive value that specifies future price.
  
  Requirement Specify \( F \) and \( E \) in the same units.
  
  Data type DOUBLE

- \( r \)
  
  is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.
  
  Data type DOUBLE

- \( \sigma \)
  
  is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of \( r \)).
  
  Data type DOUBLE

**Details**

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The function is based on the following relationship:

\[ \text{CALL} = e^{-rt}(FN(d_1) - EN(d_2)) \]

**Arguments**

- \( F \) specifies future price.

- \( N \) specifies the cumulative normal density function.

- \( E \) specifies the exercise price of the option.
$r$

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

t

specifies the time to expiration, in years.

d_1 = \frac{\ln(F/E) + (\sigma^2 t)}{\sigma \sqrt{t}}

d_2 = d_1 - \sigma \sqrt{t}

The following arguments apply to the preceding equation:

$\sigma$

specifies the volatility of the underlying asset.

$\sigma^2$

specifies the variance of the rate of return.

For the special case of $t=0$, the following equation is true:

\[ \text{CALL} = \max((F - E), 0) \]

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

Comparisons

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKCLPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=blackclprc(50, .25, 48, .05, .25);</td>
<td>1.55130142723117</td>
</tr>
<tr>
<td>b=blackclprc(9, 1/12, 10, .05, .2);</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

Functions:

• “BLACKPTPRC Function” on page 399
BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Category: Financial
Returned data type: DOUBLE

Syntax
BLACKPTPRC(E, t, F, r, sigma)

Arguments

\( E \)

is a nonmissing, positive value that specifies exercise price.

Requirement Specify \( E \) and \( F \) in the same units.

Data type DOUBLE

\( t \)

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

\( F \)

is a nonmissing, positive value that specifies future price.

Requirement Specify \( F \) and \( E \) in the same units.

Data type DOUBLE

\( r \)

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

\( \sigma \)

is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of \( r \)).

Data type DOUBLE

Details

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The function is based on the following relationship:

\[ \text{PUT} = \text{CALL} + e^{-rt}(E - F) \]

Arguments
\( E \)

specifies the exercise price of the option.

\( r \)

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

\( t \)

specifies the time to expiration, in years.

\( F \)

specifies future price.

\[
d_1 = \frac{\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right) t}{\sigma \sqrt{t}}
\]

\[
d_2 = d_1 - \sigma \sqrt{t}
\]

The following arguments apply to the preceding equation:

\( \sigma \)

specifies the volatility of the underlying asset.

\( \sigma^2 \)

specifies the variance of the rate of return.

For the special case of \( t=0 \), the following equation is true:

\[
PUT = \max(E - F, 0)
\]

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

**Comparisons**

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. These functions return a scalar value.

**Example**

The following statements illustrate the BLACKPTPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=blackptprc(298, .25, 350, .06, .25);</td>
<td>1.85980563934969</td>
</tr>
<tr>
<td>b=blackptprc(145, .5, 170, .05, .2);</td>
<td>1.41234979911583</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “BLACKCLPRC Function” on page 396
BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

- **Category:** Financial
- **Returned data type:** DOUBLE

**Syntax**

`BLKSHCLPRC(E, t, S, r, sigma)`

**Arguments**

- **E**
  - is a nonmissing, positive value that specifies the exercise price.
  - **Requirement**: Specify $E$ and $S$ in the same units.
  - **Data type**: DOUBLE

- **t**
  - is a nonmissing value that specifies the time to maturity, in years.
  - **Data type**: INTEGER

- **S**
  - is a nonmissing, positive value that specifies the share price.
  - **Requirement**: Specify $S$ and $E$ in the same units.
  - **Data type**: DOUBLE

- **r**
  - is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.
  - **Data type**: DOUBLE

- **sigma**
  - is a nonmissing, positive fraction that specifies the volatility of the underlying asset.
  - **Data type**: DOUBLE

**Details**

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

```
CALL = SN(d_1) - E N(d_2) e^{-rt}
```

**Arguments**
$S$

is a nonmissing, positive value that specifies the share price.

$N$

specifies the cumulative normal density function.

$E$

is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\ln \left( \frac{S}{E} \right) + \left( r + \frac{\sigma^2}{2} \right) t}{\sigma \sqrt{t}}$$

$$d_2 = d_1 - \sigma \sqrt{t}$$

The following arguments apply to the preceding equation:

$t$

specifies the time to expiration, in years.

$r$

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

$\sigma$

specifies the volatility (the square root of the variance).

$\sigma^2$

specifies the variance of the rate of return.

For the special case of $t=0$, the following equation is true:

$$\text{CALL} = \max(S - E, 0)$$

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

**Comparisons**

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

**Example**

The following statements illustrate the BLKSHCLPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = blkshclprc(50, .25, 48, .05, .25);</td>
<td>1.79894201954462</td>
</tr>
<tr>
<td>b = blkshclprc(9, 1/12, 10, .05, .2);</td>
<td>1</td>
</tr>
</tbody>
</table>
BLKSHPTPRC Function
Calculates put prices for European options on stocks, based on the Black-Scholes model.

Syntax
BLKSHPTPRC\((E, \ t, \ S, \ r, \ sigma)\)

Arguments
\(E\)
is a nonmissing, positive value that specifies the exercise price.

Requirement: Specify \(E\) and \(S\) in the same units.

Data type: DOUBLE

\(t\)
is a nonmissing value that specifies the time to maturity, in years.

Data type: INTEGER

\(S\)
is a nonmissing, positive value that specifies the share price.

Requirement: Specify \(S\) and \(E\) in the same units.

Data type: DOUBLE

\(r\)
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type: DOUBLE

\(sigma\)
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type: DOUBLE

Details
The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:
\[ \text{PUT} = \text{CALL} - S + Ee^{-rt} \]

**Arguments**

\( S \)
- is a nonmissing, positive value that specifies the share price.

\( E \)
- is a nonmissing, positive value that specifies the exercise price of the option.

\[ d_1 = \frac{\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t}{\sigma\sqrt{t}} \]
\[ d_2 = d_1 - \sigma\sqrt{t} \]

The following arguments apply to the preceding equation:

\( t \)
- specifies the time to expiration, in years.

\( r \)
- specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

\( \sigma \)
- specifies the volatility (the square root of the variance).

\( \sigma^2 \)
- specifies the variance of the rate of return.

For the special case of \( t = 0 \), the following equation is true:

\[ \text{PUT} = \max((E - S), 0) \]

For information about the basics of pricing, see “Using Pricing Functions” in *SAS Functions and CALL Routines: Reference.*

**Comparisons**

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

**Example**

The following statements illustrate the BLKSHPTPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.56597442946066</td>
</tr>
<tr>
<td>a=blkshptprc(230,.5,290,.04,.25);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.64091943067592</td>
</tr>
<tr>
<td>b=blkshptprc(350,3,400,.05,.2);</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Functions:
- “BLKSHCLPRC Function” on page 401

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Category: Bitwise Logical Operations
Returned data type: DOUBLE

Syntax

\texttt{BLSHIFT(}\textit{expression-1, expression-2})\texttt{)}

Arguments

\textit{expression-1}

specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>between 0 and (2^{32})–1 inclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

\textit{expression-2}

specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>0 to 31, inclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

Example

The following statement illustrates the BLSHIFT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x=blshift(7,2);}</td>
<td>28</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “BRSHIFT Function” on page 407
BNOT Function

Returns the bitwise logical NOT of an argument.

Category: Bitwise Logical Operations
Returned data type: DOUBLE

Syntax

BNOT(expression)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range between 0 and \(2^{32}-1\) inclusive

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Example

The following statement illustrates the BNOT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=bnot(16);</td>
<td>4294967279</td>
</tr>
</tbody>
</table>

BOR Function

Returns the bitwise logical OR of two arguments.

Category: Bitwise Logical Operations
Returned data type: DOUBLE

Syntax

BOR(expression-1, expression-2)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.
BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

**Category:** Bitwise Logical Operations

**Returned data type:** DOUBLE

## Syntax

\[
\text{BRSHIFT}(\text{expression-1}, \text{expression-2})
\]

## Arguments

**expression-1**

specifies any valid expression that evaluates to a numeric value.

- **Range:** between 0 and \((2^{32})-1\) inclusive
- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**expression-2**

specifies any valid expression that evaluates to a numeric value.

- **Range:** 0 to 31, inclusive
- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

## Example

The following statement illustrates the BRSHIFT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a=brshift(4,8);</code></td>
<td>12</td>
</tr>
</tbody>
</table>
BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

<table>
<thead>
<tr>
<th>Category</th>
<th>Bitwise Logical Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

$$\text{BXOR}(\text{expression-1, expression-2})$$

Arguments

expression-1, expression-2

specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Range</th>
<th>between 0 and ((2^{32})-1) inclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

See

Chapter 13, “DS2 Expressions,” on page 93

Example

The following statement illustrates the BXOR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=bxor(128,64);</td>
<td>192</td>
</tr>
</tbody>
</table>

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

<table>
<thead>
<tr>
<th>Category</th>
<th>Character</th>
</tr>
</thead>
</table>
Returned data type: VARCHAR

Syntax

BYTE(n)

Arguments

n
specifies an integer that represents a specific ASCII or EBCDIC character.

Range 0–255

Data type NCHAR

Details

For EBCDIC collating sequences, n is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

The following statement illustrates the BYTE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=byte(80);</td>
<td>♣</td>
</tr>
</tbody>
</table>

See Also

Functions:

• “RANK Function” on page 747
Syntax

\texttt{CAT(item-1[, \ldots item-n])}

Arguments

\textit{item}

specifies a constant, variable, or expression, either character or numeric. If \textit{item} is numeric, then its value is converted to a character string by using the \texttt{BESTw} format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

\textbf{Length of Returned Variable}

If the \texttt{CAT} function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

\textbf{Length of Returned Variable: Special Cases}

The \texttt{CAT} function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the \texttt{CAT} function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when \texttt{CAT} is called from the macro processor

If \texttt{CAT} returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after \texttt{CAT} finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank line in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets \texttt{_ERROR_} to 1

The \texttt{CAT} function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the \texttt{BESTw} format.

Comparisons

The results of the \texttt{CAT}, \texttt{CATS}, \texttt{CATT}, and \texttt{CATX} functions are usually equivalent to results that are produced by certain combinations of the concatenation operators | | and .., and the TRIM and LEFT functions. However, the default length for the \texttt{CAT}, \texttt{CATS}, \texttt{CATT}, and \texttt{CATX} functions is different from the length that is obtained when you use the concatenation operators. For more information, see “Length of Returned Variable” on page 410.
Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

Example

The following example shows how the CAT function concatenates strings.

```sas
data _null_;  
dcl varchar(25) x y z a;  
dcl varchar(70) result;  
method init();  
x='  The 2012 Olym';  
y='pic Arts Festi';  
z='  val included works by D  ';  
a='ale Chihuly.';  
result=cat(x,y,z,a);  
put result=;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
result=  The 2012 Olympic Arts Festi  val included works by D  ale Chihuly.
```

See Also

Functions:

- “CATS Function” on page 411
- “CATT Function” on page 413
- “KSTRCAT Function” on page 589
- “LEFT Function” on page 602
- “STRIP Function” on page 790

---

**CATS Function**

Removes leading and trailing blanks, and returns a concatenated character string.

**Category:** Character  
**Returned data type:** CHAR, NCHAR

**Syntax**

```
CATS(item-1[, ...item-n])
```
**Arguments**

*item*

specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BESTw. format. In this case, leading blanks are removed and SAS does not write a note to the log.

**Details**

**Length of Returned Variable**

If the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

**Length of Returned Variable: Special Cases**

The CATS function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATS function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATS is called from the macro processor

If CATS returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATS finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATS function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BESTw. format.

**Comparisons**

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .., and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators. For more information, see “Length of Returned Variable” on page 412.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.
Example
The following example shows how the CATS function concatenates strings.

```sas
data _null_;  
dcl char(25) x y z a;  
dcl char(70) result;  
method init();  
x='  The Olym';  
y='pic Arts Festi';  
z='  val includes works by D  ';  
a='ale Chihuly.';  
result=cats(x,y,z,a);  
put result=;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
result=The Olympic Arts Festival includes works by Dale Chihuly.
```

See Also

Functions:
- “CAT Function” on page 409
- “CATT Function” on page 413
- “CATX Function” on page 415
- “STRIP Function” on page 790

### CATT Function

Removes trailing blanks, and returns a concatenated character string.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>CHAR, NCHAR</td>
</tr>
</tbody>
</table>

#### Syntax

\[ \text{CATT} \left( \text{item-1[, \ldots item-n]} \right) \]

#### Arguments

- **item**
  
  specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BESTw. format. In this case, leading blanks are removed and SAS does not write a note to the log.
Details

Length of Returned Variable
If the CATT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases
The CATT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATT function has the following length:
• up to 200 characters in WHERE clauses and in PROC SQL
• up to 32767 characters in PROC DS2, except in WHERE clauses
• up to 65534 characters when CATT is called from the macro processor

If CATT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:
• changes the result to a blank value in PROC DS2 and in PROC SQL
• writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
• writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
• sets _ERROR_ to 1

The CATT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BESTw. format.

Comparisons
The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to results that are produced by certain combinations of the concatenation operators | | and .., and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators. For more information, see “Length of Returned Variable” on page 414.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

Example
The following example shows how the CATT function concatenates strings.
```sas
data _null_
  method init();
  dcl char(25) x y z a;
  dcl char(70) result;
```

414 Chapter 23 • DS2 Functions
CATX Function

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

## Syntax

```sas
CATX(delimiter, item-1[, ... item-n])
```

## Arguments

- **delimiter**
  - Specifies a character string that is used as a delimiter between concatenated items.

- **item**
  - Specifies a constant, variable, or expression, either character or numeric. If `item` is numeric, then its value is converted to a character string by using the `BESTw` format. In this case, SAS does not write a note to the log. For more information, see “The Basics” on page 415.

## Details

### The Basics

The CATX function first copies `item-1` to the result, omitting leading and trailing blanks. Then for each subsequent argument `item-i`, `i=2, ..., n`, if `item-i` contains at least one non-

blank character, then CATX appends delimiter and item-i to the result, omitting leading and trailing blanks from item-i. CATX does not insert the delimiter at the beginning or end of the result. Blank items do not produce delimiters at the beginning or end of the result, nor do blank items produce multiple consecutive delimiters.

**Length of Returned Variable**
If the CATX function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the || or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

**Length of Returned Variable: Special Cases**
The CATX function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATX function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATX is called from the macro processor

If CATX returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATX finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

**Comparisons**
The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to results that are produced by certain combinations of the concatenation operators || and .., and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see “Length of Returned Variable” on page 416.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

**Note:** In the case of variables that have missing values, the concatenation produces different results.

**Example**
The following example shows how the CATX function concatenates strings. The first data program creates the Values table. The second and third data programs use the Values table as input.
/* This data program creates the Values table. */
data values;
dcl char(4) x1 x2 x3 x4;
method init();

/* simple values */
x1='A'; x2='B'; x3='C'; x4='D';
output;
x1='XX'; x2='YY'; x3='ZZ'; x4='WW';
output;

/* values with leading, trailing, and embedded white space */
x1='XX'; x2='Y Y '; x3=' Z Z'; x4=' WW ';
output;

/* CHAR set to missing */
x1='E'; x2= . ; x3='F'; x4='G';
output;
x1='H'; x2= . ; x3= . ; x4='J';
output;

/* CHAR set to zero-length strings */
x1='X'; x2=''; x3=''; x4='W';
output;

/* CHAR set to the null value */
x1='X'; x2=null; x3='Z' ; x4=null;
output;
end;
run;

/* This data program creates the Concat1 table. */
data concat1;
dcl char(1) sp;
dcl char(4) x1 x2 x3 x4;
dcl char(20) test1 test2 spacey;
method run();
set values;
SP='^';
test1 = catx(sp, x1, x2, x3, x4);
test2 = strip(x1)
  || sp || strip(x2)
  || sp || strip(x3)
  || sp || strip(x4);
spacey = x1 || sp || x2 || sp || x3 || sp || x4;
end;
run;

/* This data program creates the Concat2 table. */
/* The example shows what happens when the delimiter contains */
/* space characters. */
data concat2;
dcl char(3) sp;
dcl char(4) x1 x2 x3 x4;
dcl char(20) test1 test2 spacey;
method run();
set values;
SP = ' ^ ';  

``` SAS 

`test1 = catx(SP, x1, x2, x3, x4);`  

`test2 = strip(x1)`  

`| | strip(SP) | | strip(x2)`  

`| | strip(SP) | | strip(x3)`  

`| | strip(SP) | | strip(x4);`  

`spacey = strip(x1)`  

`| | sp | | strip(x2)`  

`| | sp | | strip(x3)`  

`| | sp | | strip(x4);`  

end;`  

`run;`  

`quit;`  

`proc print data=concat1;`  

`title 'The Concat1 table';`  

`run;`  

`proc print data=concat2;`  

`title 'The Concat2 table';`  

`run;`  

```

Output 23.1 Table Showing Concatenated Characters

<table>
<thead>
<tr>
<th>Obs</th>
<th>sp</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>test1</th>
<th>test2</th>
<th>spacey</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td>A^B^C^D</td>
<td>A^B^C^D</td>
<td>A^B^C^D</td>
</tr>
<tr>
<td>2</td>
<td>XX</td>
<td>YY</td>
<td>ZZ</td>
<td>WW</td>
<td></td>
<td>XX^YY^ZZ^WW</td>
<td>XX^YY^ZZ^WW</td>
<td>XX^YY^ZZ^WW</td>
</tr>
<tr>
<td>3</td>
<td>XX</td>
<td>YY</td>
<td>ZZ</td>
<td>WW</td>
<td></td>
<td>XX^YY^ZZ^WW</td>
<td>XX^YY^ZZ^WW</td>
<td>XX^YY^ZZ^WW</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>.</td>
<td>F</td>
<td>G</td>
<td></td>
<td>E^A^F^G</td>
<td>E^A^F^G</td>
<td>E^A^F^G</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td>X^A^W</td>
<td>X^A^W</td>
<td>X^A^W</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td>X^A^Z</td>
<td>X^A^Z</td>
<td>X^A^Z</td>
</tr>
</tbody>
</table>

The Concat1 Data Set

Chapter 23 • DS2 Functions
CEIL Function

Returns the smallest integer greater than or equal to a numeric value expression.

**Category:** Truncation

**Returned data type:** DECIMAL, DOUBLE, NUMERIC

### Syntax

`CEIL(expression)`

### Arguments

**expression**

specifies any valid expression that evaluates to a numeric value.

**Data type**

DECIMAL, DOUBLE, NUMERIC

**See**

Chapter 13, “DS2 Expressions,” on page 93
Details
If the result is a number that does not fit into the range of the argument's data type, the CEIL function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons
Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example
The following statements illustrate the CEIL function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=ceil(-2.4);</td>
<td>-2</td>
</tr>
<tr>
<td>b=ceil(1+1.e-11);</td>
<td>2</td>
</tr>
<tr>
<td>c=ceil(-1+1.e-11);</td>
<td>0</td>
</tr>
<tr>
<td>d=ceil(1+1.e-13);</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also
Functions:
- “CEILZ Function” on page 420
- “FLOOR Function” on page 505
- “FLOORZ Function” on page 506

CEILZ Function
Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Category: Truncation
Returned data type: DOUBLE

Syntax
CEILZ(expression)
Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type: DOUBLE

See: Chapter 13, “DS2 Expressions,” on page 93

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEILZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = ceilz(2.1);</td>
<td>3</td>
</tr>
<tr>
<td>b = ceilz(3);</td>
<td>3</td>
</tr>
<tr>
<td>c = ceilz(1+1.e-11);</td>
<td>2</td>
</tr>
<tr>
<td>d = ceilz(223.456);</td>
<td>224</td>
</tr>
<tr>
<td>e = ceilz(-223.456);</td>
<td>-223</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “CEIL Function” on page 419
- “FLOOR Function” on page 505
- “FLOORZ Function” on page 506

CHOOSEC Function

Returns a character value that represents the results of choosing from a list of arguments.

Category: Character

Returned data type: VARCHAR, NVARCHAR
Syntax

\[
\text{CHOOSEC}(\text{index-expression}, \text{selection-1}, \ldots \text{selection-n})
\]

Arguments

\text{index-expression}

specifies any valid expression that evaluates to an integer value.

Data type INTEGER

\text{selection}

specifies a character constant, variable, or expression. The value of this argument is returned by the CHOOSEC function.

Data type DOUBLE

Details

The CHOOSEC function uses the value of \text{index-expression} to select from the arguments that follow. For example, if \text{index-expression} is 3, CHOOSEC returns the value of \text{selection-3}. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

Comparisons

The CHOOSEC function is similar to the CHOOSEN function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

Example

The following example shows how CHOOSEC chooses from a series of values:

```sas
data test (overwrite=yes);
  dcl char fruit color planet sport;
  method init();
    Fruit=choosec(1, 'apple', 'orange', 'pear', 'fig');
    Color=choosec(3, 'red', 'blue', 'green', 'yellow');
    Planet=choosec(2, 'Mars', 'Mercury', 'Uranus');
    Sport=choosec(-3, 'soccer', 'baseball', 'gymnastics', 'skiing');
    put Fruit= Color= Planet= Sport=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
fruit=apple  color=green  planet=Mercury  sport=baseball
```

See Also

Functions:

- “CHOOSEN Function” on page 423
CHOOSEN Function

CHOOSEN returns a numeric value that represents the results of choosing from a list of arguments.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

`CHOOSEN(index-expression, selection-1[, ...selection-n])`

**Arguments**

- **index-expression**
  - Specifies any valid expression that evaluates to an integer value.
  - Data type: INTEGER
  - See: [Chapter 13, “DS2 Expressions,” on page 93](#)

- **selection**
  - Specifies a numeric constant, variable, or expression. The value of this argument is returned by the CHOOSEN function.
  - Data type: DOUBLE

**Details**

The CHOOSEN function uses the value of `index-expression` to select from the arguments that follow. For example, if `index-expression` is 3, CHOOSEN returns the value of `selection-3`. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

**Comparisons**

The CHOOSEN function is similar to the CHOOSEC function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

**Example**

The following example shows how CHOOSEN chooses from a series of values:

```plaintext
data test;
  dcl double itemnumber rank score value;
  method run();
    ItemNumber=choosen(5,100,50,3784,498,679);
    Rank=choosen(-2,1,2,3,4,5);
    Score=choosen(3,193,627,33,290,5);
    Value=choosen(-5,-37,82985,-991,3,1014,-325,3,54,-618);
    put 'ItemNumber=' ItemNumber;
    put 'Rank=' Rank;
    put 'Score=' Score;
```

```sas
put 'Value= ' Value;
end;
enddata;
run;

SAS writes the following output to the log:

<table>
<thead>
<tr>
<th>ItemNumber</th>
<th>Rank</th>
<th>Score</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>679</td>
<td>4</td>
<td>33</td>
<td>1014</td>
</tr>
</tbody>
</table>
```

**CMP Function**

Compares two character strings including trailing blanks.

- **Category:** Character
- **Returned data type:** BIGINT

**Syntax**

```sas
CMP (string-1, string-2)
```

**Arguments**

- **string-1**
  - Specifies a character constant, variable, or expression.
  - **Data type:** CHAR, VARCHAR

- **string-2**
  - Specifies a character constant, variable, or expression.
  - **Data type:** CHAR, VARCHAR

**Details**

In the CMP function, if string-1 and string-2 do not differ, CMP returns a value of zero. If the arguments differ, the sign of the result is negative if string-1 precedes string-2 in a sort sequence, and positive if string-1 follows string-2 in a sort sequence.

The CMP function does not remove trailing blanks.

**Comparisons**

The CMP function compares two strings but does not remove trailing blanks. The CMPT function compares two strings and does remove trailing blanks.

**Example**

The following example uses the CMP function to compare two different strings.

```sas
proc ds2;
```
data test (overwrite=yes);
dcl double nopad pad greaterthan lessthan;
method run();
    nopad=cmp('abc', 'def');
    pad=cmp('abc', 'abc ');
    greaterthan=cmp('abc', 'abcdef');
    lessthan=cmp('abcdef', 'abc');
    put nopad= pad= greaterthan= lessthan=;
end;
enddata;
run;

nopad=-1 pad=-1 greaterthan=-1 lessthan=1

See Also

Functions:
• “CMPT Function” on page 425

CMPT Function

Compares two character strings excluding trailing blanks.

Category:  Character
Returned data type:  BIGINT

Syntax

CMPT (string-1, string-2)

Arguments

string-1
- specifies a character constant, variable, or expression.

Data type  CHAR, VARCHAR

string-2
- specifies a character constant, variable, or expression.

Data type  CHAR, VARCHAR

Details

In the CMPT function, if string-1 and string-2 do not differ, CMPT returns a value of zero. If the arguments differ, the sign of the result is negative if string-1 precedes string-2 in a sort sequence, and positive if string-1 follows string-2 in a sort sequence. The CMPT function removes trailing blanks.
Comparisons

The CMPT function compares two strings and removes trailing blanks. The CMP function compares two strings and does not remove trailing blanks.

Example

The following example uses the CMPT function to compare two different strings.

```plaintext
proc ds2;
data test (overwrite=yes);
dcl double nopad pad greaterthan lessthan;
method run();
  nopad=cmpt('abc', 'def');
  pad=cmpt('abc', 'abc ');
  greaterthan=cmpt('abc', 'abcdef');
  lessthan=cmpt('abcdef', 'abc');
  put nopad= pad= greaterthan= lessthan=;
end;
enddata;
run;
```

nopad=0 pad=0 greaterthan=-1 lessthan=1

See Also

Functions:

- “CMP Function” on page 424

COALESCE Function

Returns the first non-null or nonmissing value from a list of numeric arguments.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Mathematical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

```
COALESCE(expression[, …expression])
```

Arguments

- **expression**
  - specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Data type</th>
<th>DOUBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>
Details
COALESCE accepts one or more numeric expressions. The COALESCE function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all expressions are null or missing, then the COALESCE function returns a null or a missing value depending on whether you are in ANSI mode or SAS mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

Comparisons
The COALESCE function searches numeric expressions, whereas the COALESCEC function searches character expressions.

Example
The following statement illustrates the COALESCE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x = COALESCE(., .A, 33, 22, 44, .);}</td>
<td>33</td>
</tr>
</tbody>
</table>

See Also
Functions:
- “COALESCEC Function” on page 427

COALESCEC Function
Returns the first non-null or nonmissing value from a list of character arguments.

Category: Character
Returned data type: NCHAR

Syntax
COALESCEC(expression[, …expression])

Arguments
expression
specifies any valid expression that evaluates to a character value.

Data type NCHAR

See Chapter 13, “DS2 Expressions,” on page 93
Details

COALESCEC accepts one or more character expressions. The COALESCEC function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCEC function returns the value of that expression. If all the values of all expressions are null or missing, then the COALESCEC function returns a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

Comparisons

The COALESCEC function searches character expressions, whereas the COALESCE function searches numeric expressions.

Example

The following statements illustrate the COALESCEC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=coalescec('', 'Hello');</td>
<td>Hello</td>
</tr>
<tr>
<td>a=coalescec('', 'Goodbye', 'Hello');</td>
<td>Goodbye</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “COALESCE Function” on page 426

COMB Function

Computes the number of combinations of n elements taken r at a time.

Category: Combinatorial

Returned data type: INTEGER

Syntax

COMB(n, r)

Arguments

n

is a nonnegative integer that represents the total number of elements from which the sample is chosen.

Data type INTEGER
$r$ is a nonnegative integer that represents the number of chosen elements.

Restriction $r \leq n$

Data type INTEGER

Details

The mathematical representation of the COMB function is given by the following equation:

$$COMB(n, r) = \binom{n}{r} = \frac{n!}{r! \cdot (n - r)!}$$

In the preceding equation, $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the COMB function.

Example

The following statement illustrates the COMB function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x=\text{comb}(27, 2)$;</td>
<td>351</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “FACT Function” on page 488
- “PERM Function” on page 678

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>CHAR, VARCHAR</td>
</tr>
</tbody>
</table>

Syntax

COMPARE($\text{string-1, string-2[, modifiers]}$)
Arguments

string-1
specifies a character constant, variable, or expression.
Data type  CHAR, VARCHAR

string-2
specifies a character constant, variable, or expression.
Data type  CHAR, VARCHAR

modifiers
specifies a character string that can modify the action of the COMPARE function.
You can use one or more of the following characters as a valid modifier:

i or I  ignores the case in string-1 and string-2.
l or L  removes leading blanks in string-1 and string-2 before comparing the values.
n or N  removes quotation marks from any argument that is a name literal and ignores the case of string-1 and string-2. A name literal is a name token that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter n. Name literals enable you to use special characters (including blanks) that are not otherwise allowed in table or variable names. For COMPARE to recognize a string as a name literal, the first character must be a quotation mark.

:  (colon) truncates the longer of string-1 or string-2 to the length of the shorter string, or to one, whichever is greater. If you do not specify this modifier, the shorter string is padded with blanks to the same length as the longer string.

Data type  CHAR, VARCHAR

Tip  COMPARE ignores blanks that are used as modifiers.

Details

The Basics
The order in which the modifiers appear in the COMPARE function is relevant.

• “LN” first removes leading blanks from each string, and then removes quotation marks from name literals.
• “NL” first removes quotation marks from name literals, and then removes leading blanks from each string.

In the COMPARE function, if string-1 and string-2 do not differ, COMPARE returns a value of zero. If the arguments differ, then the following apply:

• The sign of the result is negative if string-1 precedes string-2 in a sort sequence, and positive if string-1 follows string-2 in a sort sequence.
• The magnitude of the result is equal to the position of the leftmost character at which the strings differ.
**DBCS Compatibility**

The DBCS equivalent function is KCOMPARE. There are minor differences between the COMPARE and KCOMPARE functions. Both functions accept varying numbers of arguments, but usage of the third argument is not compatible. The following example shows the differences in the syntax:

```
COMPARE(string-1, string-2[, modifiers])
KCOMPARE(string-1[, position[, count]], string-2)
```

For more information, see the “KCOMPARE Function” in *SAS National Language Support (NLS): Reference Guide.*

**Examples**

**Example 1: Understanding the Order of Comparisons When Comparing Two Strings**

The following example compares two strings by using the COMPARE function.

```sas
data test;
  dcl char string1 string2 modifiers having informat $char8. format $char8.;
  method init();
    string1='12345678'; string2='12345678'; output;
    string1='123'; string2='abc'; output;
    string1='abc'; string2='abx'; output;
    string1='xyz'; string2='abcde'; output;
    string1='aBc'; string2='abc'; output;
    string1='aBc'; string2='AbC'; modifiers='i'; output;
    string1=' abc '; string2='AbC'; modifiers='i'; output;
    string1=' abc '; string2='abc'; modifiers='i'; output;
    string1=' abc '; string2='abx'; modifiers='i'; output;
    string1=' abc '; string2=' abc '; modifiers='i'; output;
end;
enddate;
run;

data test_out;
  method run();
    set test;
    result=compare(string1, string2, modifiers);
    put 'String 1= ' string1 'String 2= ' string2 'Modifier= ' modifiers
       'Result= ' result;
  end;
run;
proc print data=test_out noobs;run;quit;
```
Example 2: Truncating Strings Using the COMPARE Function

The following example uses the : (colon) modifier to truncate strings.

```sas
data test2;
  dcl double pad1 pad2 truncate1 truncate2 blank;
  method run();
      pad1=compare('abc','abc            ');
      pad2=compare('abc','abcdef         ');
      truncate1=compare('abc','abcdef',':');
      truncate2=compare('abcdef','abc',':');
      blank=compare('', 'abc',          ':');
      put pad1 pad2 truncate1 truncate2 blank;
  end;
enddata;
run;
proc print data=test2 noobs;run;quit;
quit;
```
COMPBL Function

Removes multiple blanks from a character string.

**Category:** Character

**Returned data type:** NCHAR

**Syntax**

\[ \text{COMPBL}(\text{character-expression}) \]

**Arguments**

**character-expression**

specifies any valid expression that evaluates to a character string and that specifies the character string to compress.

**Data type** NCHAR

**See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

**Comparisons**

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string, the COMPRESS function is similar to the COMPBL function. However, the COMPRESS function removes all blanks from the source string. The COMPBL function compresses multiple blanks to a single blank and has no effect on a single blank.

**Example**

The following statements illustrate the COMPBL function.
<table>
<thead>
<tr>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=compbl('January Status');</td>
</tr>
<tr>
<td>January Status</td>
</tr>
<tr>
<td>string='125 E. Main St.';</td>
</tr>
<tr>
<td>street=compbl(string);</td>
</tr>
<tr>
<td>125 E. Main St.</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “COMPRESS Function” on page 437

**COMPFUZZ Function**

Performs a fuzzy comparison of two numeric values.

**Category:** Mathematical

**Returned data type:** DOUBLE

**Syntax**

`COMPFUZZ(expression-1, expression-2[, fuzz[, scale]])`

**Arguments**

**expression-1**

specifies any valid expression that evaluates to a numeric value.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**expression-2**

specifies any valid expression that evaluates to a numeric value.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**fuzz**

is a nonnegative numeric value that specifies the relative threshold for comparisons. Values that are greater than or equal to one are treated as multiples of the machine precision.

- **Default:** 1024
- **Data type:** DOUBLE
scale
specifies the scale factor.

<table>
<thead>
<tr>
<th>Default</th>
<th>MAX (ABS (expression-1), ABS (expression-2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Details
The COMPFUZZ function returns the following values if you specify all four arguments:

- 1 if expression-1 < expression-2 – threshold
- 0 if ABS(expression-1 - expression-2) ≤ threshold
- 1 if expression-1 > expression-2 + threshold

The following relationships exist:

- threshold = fuzz * ABS(scale) if 0 ≤ fuzz < 1
- threshold = fuzz * ABS(scale) * CONSTANT('MACEPS') if 1 ≤ fuzz < 1 / CONSTANT('MACEPS')

COMPFUZZ avoids floating-point underflow or overflow.

Comparisons
The COMPFUZZ function compares two floating-point numbers and returns a value based on the comparison. The ROUND function rounds an argument to a value that is very close to a multiple of a second argument. The result might not be an exact multiple of the second argument.

Example
In floating-point arithmetic, the value of a sum sometimes depends on the order in which the numbers are added. One approximate bound for the floating-point error in the computation of a sum of n numbers, x1 through xn, is expressed by the following formula:

\[ n \times \text{machine\_precision} \times \sum (\text{abs}(x_1) + \ldots + \text{abs}(x_n)) \]

To compare sums of n floating-point numbers with the COMPFUZZ function, you can therefore use n as the fuzz value and the sum of the absolute values as the scale factor, as shown in the following DATA step:

```plaintext
data test (overwrite=yes);
dcl double x1 x2 x3 x4 sum1 sum2 diff compfuzz1 compfuzz2 scale;
method run();
x1 = -1./3.;
x2 = 22./7.;
x3 = -1234567891.;
x4 = 1234567890.;
/* Add the numbers in two different orders. */
sum1 = x1 + x2 + x3 + x4;
sum2 = x4 + x3 + x2 + x1;
diff = abs(sum1 - sum2);
put sum1=;
put sum2=;
put diff=;
```
Using only a fuzz value gives the wrong result. The fuzz value is 8 because there are four numbers in each sum, for a total of eight numbers.

```sas
compfuzz1 = compfuzz(sum1, sum2, 8);
put 'fuzz only (wrong):        ' compfuzz1=;
```

Using a fuzz factor and a scale value gives the correct result.

```sas
scale = abs(x1) + abs(x2) + abs(x3) + abs(x4);
compfuzz2 = compfuzz(sum1, sum2, 8, scale);
put 'fuzz and scale (correct): ' compfuzz2=;
end;
enddata;
runch
```

The following lines are written to the SAS log:

```sas
sum1=1.80952382087707
sum2=1.8095238095238
diff=1.1353265660929E-8
fuzz only (wrong): compfuzz1=1
fuzz and scale (correct): compfuzz2=0
```

See Also

Functions:
- “FUZZ Function” on page 509
- “ROUND Function” on page 756

COMPOUND Function

Returns compound interest parameters.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

```
COMPOUND(a, f, r, n)
```

**Arguments**

- `a`
  - specifies the initial amount.
  - **Range:** \( a \geq 0 \)
  - **Data type:** DOUBLE

- `f`
  - specifies the future amount (at the end of \( n \) periods).
  - **Range:** \( f \geq 0 \)
The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by the following equation:

\[ f = a(1 + r)^n \]

One missing argument must be provided. A compound interest parameter is then calculated from the remaining three values. No adjustment is made to convert the results to round numbers.

If \( n=0 \), then
\[ f = a \]

and
\[ (1 + r)^n \]
is equal to 1.

Note: If you choose \( r \) as your missing value, then COMPOUND returns an error.

Example

The accumulated value of an investment of $2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be expressed as follows:

\[
\text{future} = \text{compound}(2000, ., 0.09/12, 30);
\]

The value returned is 2502.544. The second argument has been set to missing, indicating that the future amount is to be calculated. The 9% nominal annual rate has been converted to a monthly rate of 0.09/12. The rate argument is the fractional (not the percentage) interest rate per compounding period.

COMPRESS Function

Returns a character string with specified characters removed from the original string.

Category: Character
Returned data type: CHAR, NCHAR

Syntax

COMPRESS(character-expression[, character-list-expression])

Arguments

character-expression

specifies any valid expression that evaluates to a character expression and from which specified characters will be removed.

Requirement
Enclose a literal string of characters in single quotation marks.

Data type
CHAR, NCHAR

See
Chapter 13, “DS2 Expressions,” on page 93

character-list-expression

specifies a variable or any valid expression that initializes a list of characters. By default, the characters in this list are removed from character-expression.

Requirement
Enclose a literal string of characters in single quotation marks.

Data type
CHAR, NCHAR

See
Chapter 13, “DS2 Expressions,” on page 93

Details

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of zero.

Based on the number of arguments, the COMPRESS functions works as follows:

<table>
<thead>
<tr>
<th>Number of Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only the first argument, source</td>
<td>All blanks have been removed. If the argument is completely blank, then the result is a string with a length of zero. If you assign the result to a character variable with a fixed length, then the value of that variable will be padded with blanks to fill its defined length.</td>
</tr>
<tr>
<td>Two arguments, source and chars</td>
<td>All characters that appear in the second argument are removed from the result.</td>
</tr>
</tbody>
</table>

To remove digits and plus or minus signs, you could use the following function call:

COMPRESS(source, "1234567890+-");
Examples

Example 1: Compressing Blanks
These examples show how to remove blanks from a character string.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcl char a b; a='AB C D '; b=compress(a);</td>
<td>ABCD</td>
</tr>
<tr>
<td>dcl char a b; a='AB C D'; b=compress(a,'A ');</td>
<td>BCD</td>
</tr>
</tbody>
</table>

Example 2: Compressing Vowels
These examples show how to remove characters from a string.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcl char(32) x; dcl char(32) y; x='123-4567-8901 e 234-5678-9012 i'; y=compress(x,'aeiou');</td>
<td>123-4567-8901 234-5678-9012</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “COMPBL Function” on page 433
- “LEFT Function” on page 602
- “TRIM Function” on page 821

CONSTANT Function
Computes machine and mathematical constants.

Category: Mathematical
Returned data type: INTEGER, DOUBLE

Syntax
CONSTANT(constant[, parameter])
Arguments

**constant**

is a character constant, variable, or expression that identifies the constant to be returned. Valid constants are as follows:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E'</td>
<td>The natural base</td>
</tr>
<tr>
<td>'EULER'</td>
<td>Euler constant</td>
</tr>
<tr>
<td>'PI'</td>
<td>Pi</td>
</tr>
<tr>
<td>'EXACTINT' [,nbytes]</td>
<td>Exact integer</td>
</tr>
<tr>
<td>'BIG'</td>
<td>The largest double-precision number</td>
</tr>
<tr>
<td>'LOGBIG' [,base]</td>
<td>The log with respect to base of BIG</td>
</tr>
<tr>
<td>'SQRTBIG'</td>
<td>The square root of BIG</td>
</tr>
<tr>
<td>'SMALL'</td>
<td>The smallest double-precision number</td>
</tr>
<tr>
<td>'LOGSMALL' [,base]</td>
<td>The log with respect to base of SMALL</td>
</tr>
<tr>
<td>'SQRTSMALL'</td>
<td>The square root of SMALL</td>
</tr>
<tr>
<td>'MACEPS'</td>
<td>Machine precision constant</td>
</tr>
<tr>
<td>'LOGMACEPS' [,base]</td>
<td>The log with respect to base of MACEPS</td>
</tr>
<tr>
<td>'SQRTMACEPS'</td>
<td>The square root of MACEPS</td>
</tr>
</tbody>
</table>

**parameter**

is a numeric parameter that can be used as an optional argument with some of the constants specified in **constant**. When used, **parameter** alters the functionality of the **CONSTANT** function.

Details

Overview

**CAUTION:**

In some operating environments, the run-time library might have limitations that prevent the use of the full range of floating-point numbers that the hardware provides. In such cases, the **CONSTANT** function attempts to return values that are compatible with the limitations of the run-time library. For example, if the run-time library cannot compute \( \exp(\log(\text{CONSTANT('BIG')})) \), then \( \text{CONSTANT('LOGBIG')} \) will not return the same value as \( \log(\text{CONSTANT('BIG')}) \), but will return a value such that \( \exp(\text{CONSTANT('LOGBIG')}) \) can be computed.
The Natural Base
CONSTANT('E')

The natural base is described by the following equation:

\[
\lim_{x \to 0} \frac{1}{(1 + x)^x} \approx 2.718281828459045
\]

Euler Constant
CONSTANT('EULER')

Euler’s constant is described by the following equation:

\[
\lim_{n \to \infty} \left\{ \sum_{j = 1}^{n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860
\]

Pi
CONSTANT('PI')

Pi is the ratio between the circumference and the diameter of a circle. Many expressions exist for computing this constant. One such expression for the series is described by the following equation:

\[
4 \sum_{j = 0}^{\infty} \frac{(-1)^j}{2j + 1} \approx 3.14159265358979323846
\]

Exact Integer
CONSTANT('EXACTINT[, nbytes])

Arguments

nbytes

is a numeric value that is the number of bytes.

Range \(2 \leq \text{nbytes} \leq 8\)

Default \(8\)

The exact integer is the largest integer \(k\) such that all integers less than or equal to \(k\) in absolute value have an exact representation in a SAS numeric variable of length \(\text{nbytes}\). This information can be useful to know before you trim a SAS numeric variable from the default 8 bytes of storage to a lower number of bytes to save storage.

The Largest Double-Precision Number
CONSTANT('BIG')

This case returns the largest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of BIG
CONSTANT('LOGBIG[, base]')

Arguments

base

is a numeric value that is the base of the logarithm.

Default the natural base, E
Restriction  The base that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to base of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given base raised to a power less than or equal to CONSTANT('LOGBIG', base) by using the power operation (**) without causing any overflows.

It is safe to exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

The Square Root of BIG
CONSTANT('SQRTBIG')

This case returns the square root of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to square any floating-point number less than or equal to CONSTANT('SQRTBIG') without causing any overflows.

The Smallest Double-Precision Number
CONSTANT('SMALL')

This case returns the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of SMALL
CONSTANT('LOGSMALL', base)

Arguments
base
is a numeric value that is the base of the logarithm.

Default  the natural base, E

Restriction  The base that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to base of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given base raised to a power greater than or equal to CONSTANT('LOGSMALL', base) by using the power operation (**) without causing any underflows or 0.

It is safe to exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

The Square Root of SMALL
CONSTANT('SQRTSMALL')

This case returns the square root of the smallest double-precision floating-point number (8-bytes) that is representable on the computer.
It is safe to square any floating-point number greater than or equal to CONSTANT('SQRTBIG') without causing any underflows or 0.

**Machine Precision**
CONSTANT('MACEPS')

This case returns the smallest double-precision floating-point number (8-bytes) $\varepsilon = 2^{-j}$ for some integer $j$, such that $1 + \varepsilon > 1$.

This constant is important in finite precision computations.

**The Logarithm of MACEPS**
CONSTANT('LOGMACEPS[, base])

**Arguments**

*base*

is a numeric value that is the base of the logarithm.

**Default**

the natural base, $E$

**Restriction**

The base that you specify must be greater than the value of $1+\text{SQRTMACEPS}$.

This case returns the logarithm with respect to base of CONSTANT('MACEPS').

**The Square Root of MACEPS**
CONSTANT('SQRTMACEPS')

This case returns the square root of CONSTANT('MACEPS').

**Example**

The following example uses the CONSTANT function to return values for various constants.

```plaintext
data test;
  /* dcl double a b c d; */
  method run();
    a=constant('E');
    b=constant('EULER');
    c=constant('PI');
    d=constant('EXACTINT');
    e=constant('BIG');
    f=constant('LOGBIG');
    g=constant('SQRTBIG');
    h=constant('SMALL');
    i=constant('LOGSMALL');
    j=constant('SQRTSMALL');
    k=constant('MACEPS');
    l=constant('LOGMACEPS');
    m=constant('SQRTMACEPS');
    put 'a= ' a;
    put 'b= ' b;
    put 'c= ' c;
    put 'd= ' d;
    put 'e= ' e;
```
CONVX Function

Returns the convexity for an enumerated cash flow.

Category: Financial

Returned data type: DOUBLE

Syntax

CONVX(y, f, c(1), ..., c(k))

Arguments

y
specifies the effective per-period yield-to-maturity.

Range 0 < y < 1

Data type DOUBLE

Tip If you express y as a fraction, the dividend must be written as a decimal value. In DS2, integer division results in a value of zero. Zero is converted to a DOUBLE and is passed as the first argument to the CONVX function. The CONVX function returns missing when a zero is passed as the first parameter.
$f$
specifies the frequency of cash flows per period.

<table>
<thead>
<tr>
<th>Range</th>
<th>$f &gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

c(1), ..., c(k)
specifies a list of cash flows.

| Data type | DOUBLE |

**Details**
The CONVX function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{f(k)}} \left(\frac{1}{y}\right)^{f(k)}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{f(k)}}$$

**Example: Using the CONVX Function**

data test;
dcl double c;
method run();
c=convx(1./20,.1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);
put c;
end;
enddata;
run;

SAS writes the following output to the log:

```
42.37760672321
```

**See Also**

**Functions:**
- “CONVXP Function” on page 445

---

**CONVXP Function**

Returns the convexity for a periodic cash flow stream, such as a bond.

**Category:** Financial
Returned data type: DOUBLE

Syntax
CONVXP(A, c, n, K, k₀, y)

Arguments

A
specifies the par value.
Range  A > 0
Data type  DOUBLE

c
specifies the nominal per-period coupon rate, expressed as a decimal.
Range  0 ≤ c < 1
Data type  DOUBLE

n
specifies the number of coupons per period.
Range  n > 0
Data type  INTEGER

K
specifies the number of remaining coupons.
Range  K > 0
Data type  INTEGER

k₀
specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.
Range  0 < k₀ ≤ \( \frac{1}{n} \)
Data type  DOUBLE

y
specifies the nominal per-period yield-to-maturity.
Range  y > 0
Data type  DOUBLE

Details
The CONVXP function returns the value from the following equation.
\[
C = \frac{1}{n^2}\left( \sum_{k=1}^{K} t_k (k + 1) \frac{c(k)}{(1 + \frac{y}{n})^{t_k}} \right)
\]

The following relationships apply to the preceding equation:

\[ t_k = nk_0 + k - 1 \]

\[ c(k) = \frac{c}{n} \quad \text{for} \quad k = 1, \ldots, K - 1 \]

\[ c(K) = \left(1 + \frac{c}{n}\right)A \]

The following relationship applies to the preceding equation:

\[
P = \sum_{k=1}^{K} \frac{c(k)}{(1 + \frac{y}{n})^{t_k}}
\]

**Example: Computing the Convexity of a Bond**

In the following example, the CONVXP function returns the convexity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the annual yield-to-maturity is 0.08.

```sas
data test;
  dcl double c;
  method run();
  c=convxp(1000,.01,4,14,.33/2,.08);
  put c;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
11.7290019868346
```

**See Also**

**Functions:**

- “CONVX Function” on page 444

---

**COS Function**

Returns the cosine in radians.

**Category:** Trigonometric
Syntax

COS(expression)

Arguments

expression

is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Example

The following statements illustrate the COS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=cosh(0.5);</td>
<td>0.87758256189037</td>
</tr>
<tr>
<td>x=cosh(0);</td>
<td>1</td>
</tr>
<tr>
<td>x=cosh(3.14159/3);</td>
<td>0.50000076602519</td>
</tr>
</tbody>
</table>

COSH Function

Returns the hyperbolic cosine in radians.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

COSH(expression)

Arguments

expression

is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93
Details
The COSH function returns the hyperbolic cosine of the argument, given by the following equation.

\[
\frac{e^{\text{argument}} + e^{-\text{argument}}}{2}
\]

Example
The following statements illustrate the COSH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = cosh(0);</td>
<td>1</td>
</tr>
<tr>
<td>x = cosh(-5.0);</td>
<td>74.2099485247878</td>
</tr>
<tr>
<td>x = cosh(4.37);</td>
<td>39.5281414700662</td>
</tr>
<tr>
<td>x = cosh(0.5);</td>
<td>1.12762596520638</td>
</tr>
</tbody>
</table>

**COUNT Function**
Counts the number of times that a specified substring appears within a character string.

- **Category:** Character
- **Returned data type:** INTEGER

**Syntax**

\[
\text{COUNT}(\text{string, substring[, modifiers]})
\]

**Arguments**

- **string**
  - specifies a character constant, variable, or expression in which substrings are to be counted.
  - **Data type:** CHAR, VARCHAR
  - **Tip:** Enclose a literal string of characters in quotation marks.

- **substring**
  - specifies the character constant, variable, or expression to be counted in string.
  - **Data type:** CHAR, VARCHAR
  - **Tip:** Enclose a literal string of characters in quotation marks.
modifiers is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

i or I ignores character case during the count. If this modifier is not specified, COUNT only counts character substrings with the same case as the characters in substring.

t or T trims trailing blanks from string and substring.

Data type CHAR, VARCHAR

Tip If modifiers is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. Modifiers can also be expressed as a variable or an expression.

Details

The Basics
The COUNT function searches string, from left to right, for the number of occurrences of the specified substring, and returns that number of occurrences. If the substring is not found in string, COUNT returns a value of 0.

CAUTION: If two occurrences of the specified substring overlap in the string, the result is undefined. For example, COUNT('boobooboo', 'booboo') might return either a 1 or a 2.

Comparisons
The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

Example
The following statements illustrate the COUNT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz='This is a thistle? Yes, this is a thistle.'; howmanythis=count(xyz,'this'); put howmanythis;</td>
<td>3</td>
</tr>
<tr>
<td>xyz='This is a thistle? Yes, this is a thistle.'; howmanyis=count(xyz,'is'); put howmanyis;</td>
<td>6</td>
</tr>
<tr>
<td>howmanythis_i=count('This is a thistle? Yes, this is a thistle.' , 'this','i'); put howmanythis_i;</td>
<td>4</td>
</tr>
</tbody>
</table>
variable1='This is a thistle? Yes, this is a thistle.';
variable2='is ';
variable3='i';
howmanyis_i=count(variable1,variable2,variable3);
put howmanyis_i;

expression1='This is a thistle? '||'Yes, this is a thistle.';
expression2=kscan('This is',2)||'     ';
expression3=compress('i     '||'     t');
howmanyis_it=count(expression1,expression2,expression3);
put howmanyis_it;

See Also

Functions:

• “COUNTC Function” on page 451
• “COUNTW Function” on page 454
• “FIND Function” on page 489
• “INDEX Function” on page 533
• “KCOUNT Function” on page 588

COUNTC Function

Counts the number of characters in a string that appear or do not appear in a list of characters.

**Category:** Character

**Returned data type:** INTEGER

**Syntax**

\[
\text{COUNTC}(\text{string, charlist[, modifiers]})
\]

**Arguments**

**string**

specifies a character constant, variable, or expression in which characters are counted.

| Data type | CHAR, VARCHAR |
| Tip       | Enclose a literal string of characters in quotation marks. |

**charlist**

specifies a character constant, variable, or expression that initializes a list of characters. COUNTC counts characters in this list, provided that you do not specify the V modifier in the modifiers argument. If you specify the V modifier, then all
characters that are not in this list are counted. You can add more characters to the list by using other modifiers.

**Data type**  CHAR, VARCHAR

**Tips**

Enclose a literal string of characters in quotation marks.

If there are no characters in the list after processing the modifiers, COUNTC returns 0.

**modifiers**

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTC function. Blanks are ignored. The following characters, in uppercase or lowercase, can be used as modifiers:

- **blank** is ignored.
- **a or A** adds alphabetic characters to the list of characters.
- **b or B** scans *string* from right to left, instead of from left to right.
- **c or C** adds control characters to the list of characters.
- **d or D** adds digits to the list of characters.
- **f or F** adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
- **g or G** adds graphic characters to the list of characters.
- **h or H** adds a horizontal tab to the list of characters.
- **i or I** ignores case.
- **l or L** adds lowercase letters to the list of characters.
- **n or N** adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
- **o or O** processes the *charlist* and *modifier* arguments only once, at the first call to this instance of COUNTC. If you change the value of *charlist* or *modifier* in subsequent calls, the change might be ignored by COUNTC.
- **p or P** adds punctuation marks to the list of characters.
- **s or S** adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
- **t or T** trims trailing blanks from *string* and *chars*. If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the COUNTC function with the T modifier.
- **u or U** adds uppercase letters to the list of characters.
- **v or V** counts characters that do not appear in the list of characters. If you do not specify this modifier, then COUNTC counts characters that do appear in the list of characters.
- **w or W** adds printable characters to the list of characters.
- **x or X** adds hexadecimal characters to the list of characters.
Tip

If modifier is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks.

Details

The COUNTC function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. If there are no characters in the list of characters to be counted, COUNTC returns zero.

Comparisons

The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Example

The following example uses the COUNTC function with and without modifiers to count the number of characters in a string.

```sas
data test;
  dcl char(24) string a b_i abc_i abc_iv abc_ivt;
  method run();
    string = 'Baboons Eat Bananas     ';
    a= countc(string, 'a');
    b= countc(string,'b');
    b_i= countc(string,'b','i');
    abc_i= countc(string,'abc','i');
    /* Scan string for characters that are not "a", "b", */
    /* and "c", ignore case, (and include blanks). */
    abc_iv  = countc(string,'abc','iv');
    /* Scan string for characters that are not "a", "b", */
    /* and "c", ignore case, and trim trailing blanks. */
    abc_ivt = countc(string,'abc','ivt');
  end;
enddata;
run;
```

Output 23.5  Results from Using the COUNTC Function with and without Modifiers

<table>
<thead>
<tr>
<th>string</th>
<th>a</th>
<th>b</th>
<th>b_i</th>
<th>abc_i</th>
<th>abc_iv</th>
<th>abc_ivt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baboons Eat Bananas</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>22</td>
<td>11</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “ANYALNUM Function” on page 359
COUNTW Function

Counts the number of words in a character string.

**Category:** Character

**Returned data type:** INTEGER

**Syntax**

```
COUNTW(string[, chars[, modifiers]])
```
Arguments

string
specifies a character constant, variable, or expression in which words are counted.

Data type  CHAR, VARCHAR

chars
specifies an optional character constant, variable, or expression that initializes a list of characters. The characters in this list are the delimiters that separate words, provided that you do not use the K modifier in the modifier argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to the list by using other modifiers.

Data type  CHAR, VARCHAR

modifiers
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTW function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank  is ignored.
a or A  adds alphabetic characters to the list of characters.
b or B  counts from right to left instead of from left to right. Right-to-left counting makes a difference only when you use the Q modifier and the string contains unbalanced quotation marks.
c or C  adds control characters to the list of characters.
d or D  adds digits to the list of characters.
f or F  adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAMES=V7) to the list of characters.
g or G  adds graphic characters to the list of characters.
h or H  adds a horizontal tab to the list of characters.
i or I  ignores the case of the characters.
k or K  causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.
l or L  adds lowercase letters to the list of characters.
m or M  specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the string argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the string argument are ignored.
n or N  adds digits, an underscore, and English letters (that is, the characters that can appear after the first character in a SAS variable name using VALIDVARNAMES=V7) to the list of characters.
o or O  processes the chars and modifier arguments only once, rather than every time the COUNTW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure,
can make COUNTW run faster when you call it in a loop where \textit{chars} and \textit{modifier} arguments do not change.

\begin{itemize}
  \item **p or P** adds punctuation marks to the list of characters.
  \item **q or Q** ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of \textit{string} contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.
  \item **s or S** adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
  \item **t or T** trims trailing blanks from the \textit{string} and \textit{chars} arguments.
  \item **u or U** adds uppercase letters to the list of characters.
  \item **w or W** adds printable characters to the list of characters.
  \item **x or X** adds hexadecimal characters to the list of characters.
\end{itemize}

\textbf{Data type} \texttt{CHAR, VARCHAR}

\section*{Details}

\textbf{Definition of “Word”}

In the COUNTW function, “word” refers to a substring that has one of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters, except if you use the \texttt{Q} modifier and the delimiters are within substrings that have quotation marks

\textit{Note}: The definition of “word” is the same in both the SCAN function and the COUNTW function.

Delimiter refers to any of several characters that you can specify to separate words.

\section*{Using the COUNTW Function in ASCII and EBCDIC Environments}

If you use the COUNTW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:
  \begin{verbatim}
  blank ! $ % & ( ) * + , - . / ; < ^ | \\
  \end{verbatim}

  In ASCII environments that do not contain the \texttt{^} character, the SCAN function uses the \texttt{~} character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:
  \begin{verbatim}
  blank ! $ % & ( ) * + , - . / ; < ^ | \texttt{¬} \texttt{|} \\
  \end{verbatim}

\section*{Using Null Arguments}

The COUNTW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.
Using the M Modifier
If you do not use the M modifier, then a word must contain at least one character. If you use the M modifier, then a word can have a length of zero. In this case, the number of words is one plus the number of delimiters in the string, not counting delimiters inside strings that are enclosed in quotation marks when you use the Q modifier.

Example
The following example shows how to use the COUNTW function with the M and P modifiers.

The explanation for the value of \texttt{mp} for each string is as follows:

- The period is the delimiter and the m modifier causes the period at the end to refer to a subsequent word with zero length, but never the less, a word. So there is one word before the period and one word after the period for a total of two words.
- No delimiters, so there is only one word.
- The p modifier adds punctuation as a delimiter therefore 3 words.
- The p modifier adds punctuation, so / is a delimiter. The m modifier causes the leading / to refer to a word at beginning with zero length for a total of six words.
- The first \ is an escape character. The second \ is a delimiter, so there are six words.

```sas
data test;
  dcl char(60) string1 having informat $char60. format $char60.;;
  method init();
    string1='The quick brown fox jumps over the lazy dog.'; output;
    string1=' Leading blanks'; output;
    string1='2+2=4'; output;
    string1='/unix/path/names/use/slashes'; output;
    string1='\Windows\Path\Names\Use\Backslashes'; output;
  end;
  enddata;
end;

data test_out;
  dcl double default blanks mp;
  method run();
    set test;
    default = countw(string1);
    blanks = countw(string1, ' ');
    mp = countw(string1, '.', 'mp');
    put 'String= ' string1 'Default= ' default 'Blanks= ' blanks 'MP= ' mp;
  end;
  enddata;
end;
run;
```
Results from Using the COUNTW Function with the M and P Modifiers

<table>
<thead>
<tr>
<th>default</th>
<th>blanks</th>
<th>mp</th>
<th>string1</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
<td>The quick brown fox jumps over the lazy dog.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Leading blanks</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2+2=4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>6</td>
<td>/unix/path/names/useslashes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>\Windows\Path\Names\Use\Backslashes</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “COUNT Function” on page 449
- “COUNTC Function” on page 451
- “FINDW Function” on page 499
- “SCAN Function” on page 772

CSS Function

Returns the corrected sum of squares.

**Category:** Descriptive Statistics

**Returned data type:** DOUBLE

**Syntax**

CSS(expression[, …expression])

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value.

**Requirement**

At least one non-null or nonmissing expression is required.

**Data type**

DOUBLE

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the CSS function:
### CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end period.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Financial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

#### Syntax

\[
\text{CUMIPMT}(rate, \text{number-of-periods}, \text{principal-amount}[, \text{start-period}][, \text{end-period}][, \text{type}])
\]

#### Arguments

- **rate**
  - specifies the interest rate per payment period.
  - Data type: DOUBLE

- **number-of-periods**
  - specifies the number of payment periods. \textit{Number-of-periods} must be a positive integer value.
  - Data type: INTEGER

- **principal-amount**
  - specifies the principal amount of the loan. Zero is assumed if a missing value is specified.
  - Data type: DOUBLE

- **start-period**
  - specifies the start period for the calculation.
  - Data type: INTEGER

- **end-period**
  - specifies the end period for the calculation.
Data type INTEGER

**type**
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if `type` is omitted or if a missing value is specified.

Data type INTEGER

**Example**

- The cumulative interest that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

  ```python
  data test;
  dcl double TotalInterest having format dollar10.2;
  method run();
  TotalInterest= CUMIPMT(0.09/12, 360, 125000, 13, 24, 0);
  put 'Total Interest=' TotalInterest;
  end;
  enddata;
  run;
  ```

  This computation returns a value of $11,135.23.

- The interest that is paid on the first period of the same loan is computed in the following way:

  ```python
  data test;
  dcl double first_period_interest having format dollar10.2;
  method run();
  first_period_interest= CUMIPMT(0.09/12, 360, 125000, 1, 1, 0);
  put 'Total Interest=' first_period_interest;
  end;
  enddata;
  run;
  ```

  This computation returns a value of $937.50.

**See Also**

Functions:

- “CUMPRINC Function” on page 460

**CUMPRINC Function**

Returns the cumulative principal paid on a loan between the start and end period.

- **Category:** Financial
- **Returned data type:** DOUBLE
Syntax

CUMPRINC(rate, number-of-periods, principal-amount[, start-period][, end-period][, type])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement Number-of-periods must be a positive integer value.

Data type INTEGER

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing or null value is specified.

start-period
specifies the start period for the calculation.

Data type INTEGER

end-period
specifies the end period for the calculation.

Data type INTEGER

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if type is omitted or if a missing value is specified.

Data type INTEGER

Example

• The cumulative principal that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

  data test;
    dcl double PrincipalYear2 having format dollar10.2;
    method run();
      PrincipalYear2=CUMPRINC(0.09/12, 360, 125000, 12, 24, 0);
      put 'Principal Year 2 EOP=' PrincipalYear2;
    end;
  enddata;
run;

This computation returns a value of $1008.23.
• The principal that is paid on the second year of the same loan with beginning-of-period payments is computed as follows:

```plaintext
data test;
  dcl double PrincipalYear2b having format dollar10.2;
  method run();
  PrincipalYear2b = CUMPRINC(0.09/12, 360, 125000, 12, 24, 1);
  put 'Principal Year 2 BOP=' PrincipalYear2b;
end;
enddata;
run;
```

This computation returns a value of $1000.73.

See Also

Functions:
• “CUMIPMT Function” on page 459

---

**CV Function**

Returns the coefficient of variation.

**Category:** Descriptive Statistics  
**Returned data type:** DOUBLE

**Syntax**

```plaintext
CV(expression-1, expression-2 [, …expression-n])
```

**Arguments**

- **expression**
  - specifies any valid expression that evaluates to a numeric value.
  - Requirement: At least two arguments are required.
  - Data type: DOUBLE

**See**

- Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the CV function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=CV(5,9,3,6);</td>
<td>43.4782608695652</td>
</tr>
<tr>
<td>x2=CV(5,8,9,6,..)</td>
<td>26.082026547865</td>
</tr>
</tbody>
</table>
DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

**Category:** Date and Time

**Returned data type:** INTEGER

**Syntax**

\[
\text{DATDIF}(sdate, edate, basis)
\]

**Arguments**

\textit{sdate}

specifies a SAS date value that identifies the starting date.

- **Data type:** DATE
- **Tip:** If \textit{sdate} falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

\textit{edate}

specifies a SAS date value that identifies the ending date.

- **Data type:** DATE
- **Tip:** If \textit{edate} falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

\textit{basis}

specifies a character string that represents the day count basis. The following values for \textit{basis} are valid:

- '30/360'
  
specifies a 30-day month and a 360-day year, regardless of the actual number of calendar days in a month or year.
  
  A security that pays interest on the last day of a month will either always make its interest payments on the last day of the month, or it will always make its payments on the numerically same day of a month, unless that day is not a valid day of the month, such as February 30. For more information, see “Method of Calculation for Day Count Basis (30/360)” in \textit{SAS Functions and CALL Routines: Reference}.

- '360'

**Example:**

\[
x3=\text{CV}(8, 9, 6, .);
\]

Results:

\[
19.9242421519819
\]
'ACT/ACT'
uses the actual number of days between dates. Each month is considered to have
the actual number of calendar days in that month, and each year is considered to
have the actual number of calendar days in that year.

Alias  'Actual'

'ACT/360'
uses the actual number of calendar days in a particular month, and 360 days as
the number of days in a year, regardless of the actual number of days in a year.

Tip  ACT/360 is used for short-term securities.

'ACT/365'
uses the actual number of calendar days in a particular month, and 365 days as
the number of days in a year, regardless of the actual number of days in a year.

Tip  ACT/365 is used for short-term securities.

Data type  CHAR, VARCHAR

Details

The Basics
The DATDIF function has a specific meaning in the securities industry, and the method
of calculation is not the same as the actual day count method. Calculations can use
months and years that contain the actual number of days. Calculations can also be based
on a 30-day month or a 360-day year. For more information about standard securities
calculation methods, see the References section at the bottom of this function.

Note:  When counting the number of days in a month, DATDIF always includes the
starting date and excludes the ending date.

Method of Calculation for Day Count Basis (30/360)
To calculate the number of days between two dates, use the following formula:

$$\text{Number of days} = [(Y_2 - Y_1) \times 360] + [(M_2 - M_1) \times 30] + (D_2 - D_1)$$

Arguments

Y2
specifies the year of the later date.

Y1
specifies the year of the earlier date.

M2
specifies the month of the later date.

M1
specifies the month of the earlier date.

D2
specifies the day of the later date.

D1
specifies the day of the earlier date.
Because all months can contain only 30 days, you must adjust for the months that do not contain 30 days. Do this before you calculate the number of days between the two dates.

The following rules apply:

- If the security follows the End-of-Month rule, and D2 is the last day of February (28 days in a non-leap year, 29 days in a leap year), and D1 is the last day of February, then change D2 to 30.
- If the security follows the End-of-Month rule, and D1 is the last day of February, then change D1 to 30.
- If the value of D2 is 31 and the value of D1 is 30 or 31, then change D2 to 30.
- If the value of D1 is 31, then change D1 to 30.

Example

In the following example, DATDIF returns the actual number of days between two dates, as well as the number of days based on a 30-day month and a 360-day year.

```
data test (overwrite=yes);
method run();
dcl date sdate edate;
dcl double actual days360;
sdate= date'1978-10-16';
edate= date'1996-02-16';
sasedate=to_double(sdate);
sasedate=to_double(edate);
actual=datdif(sasedate, sasedate, 'act/act');
days360=datdif(sasedate, sasedate, '30/360');
put 'Actual=' actual;
put 'Days 360=' days360;
end;
enddata;
run;
```

The following lines are written to the SAS log.

```
Actual= 6332
Days 360= 6240
```

See Also

Functions:

- “YRDIF Function” on page 847

References

DATE Function

Returns the current date as a SAS date value.

- **Category:** Date and Time
- **Alias:** TODAY
- **Returned data type:** DOUBLE

### Syntax

`DATE( )`

### Without Arguments

The DATE function has no arguments.

### Comparisons

The DATE function does not take any arguments. The SAS date value returned is the number of days from January 1, 1960 to the current date.

For more information about how DS2 handles dates, see Chapter 14, “Dates and Times in DS2,” on page 111.

### Example

The following statement illustrates the DATE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=date();</code></td>
<td>18773</td>
</tr>
</tbody>
</table>

### See Also

- **Functions:**
  - “TODAY Function” on page 813

---

DATEJUL Function

Converts a Julian date to a SAS date value.

- **Category:** Date and Time
- **Returned data type:** DOUBLE
Syntax

DATEJUL(julian-date)

Arguments

julian-date

specifies any valid expression that evaluates to a numeric value and that represents a Julian date. A Julian date is a date in the form yyddd or yyyyddd, where yy or yyyy is a two-digit or four-digit integer that represents the year and ddd is the number of the day of the year. The value of ddd must be between 1 and 365 (or 366 for a leap year).

Data type: DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

A SAS date value is the number of days from January 1, 1960 to a specified date. The DATEJUL function returns the number of days from January 1, 1960 to the Julian date specified in julian-date.

For more information about how dates are handled in DS2, see Chapter 14, “Dates and Times in DS2,” on page 111.

Example

The following statements illustrate the DATEJUL function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=datejul(11365);</td>
<td>18992</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “JULDATE Function” on page 586

DATEPART Function

Extracts the date from a SAS datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

DATEPART(datetime)
Arguments

datetime

specifies any valid expression that represents a SAS datetime value.

Data type: DOUBLE

See: Chapter 13, “DS2 Expressions,” on page 93

Details

A SAS datetime value is the number of seconds between January 1, 1960 and the hour, minute, and seconds within a specific date. The DATEPART function determines the date portion of the SAS datetime value and returns the date as a SAS date value, which is the number of days from January 1, 1960.

Example

The following statement illustrates the DATEPART function where the variable dtvalue, a SAS datetime value, has a value of 1652165417:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dp=put(datepart(dtvalue),date9.);</td>
<td>09MAY2012</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

- “DATETIME Function” on page 468
- “TIMEPART Function” on page 803

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

DATETIME()

Comparisons

The DATETIME function does not take any arguments. The SAS datetime value returned is the number of seconds from January 1, 1960 to the current date and time.
Example
The following statement illustrates the DATETIME function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt=datetime();</td>
<td>1622021468</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:
- “DATE Function” on page 466
- “TIME Function” on page 802

DAY Function
Returns the day of the month from a SAS date value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
`DAY(date)`

Arguments
- `date`
specifies any valid expression that represents a SAS date value.

Data type: DOUBLE

See
- Chapter 13, “DS2 Expressions,” on page 93

Details
The DAY function produces an integer from 1 to 31 that represents the day of the month. A SAS date value is the number of days from January 1, 1960 to a specific date.

Example
The following statement illustrates the DAY function where `dayvalue`, the SAS date value, has a value of 17531, which is December 31, 2007:
DEQUOTE Function

Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.

**Category:** Character

**Returned data type:** NCHAR

**Syntax**

`DEQUOTE(expression)`

**Arguments**

`expression`

specifies any valid expression that evaluates to a character string.

**Data type** NCHAR

**See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The value that is returned by the DEQUOTE function depends on the first character or the first two characters in `expression`:

- If the first character of `expression` is not a quotation mark, DEQUOTE returns a syntax error.

- If the first character of `expression` is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans `expression` from left to right, looking for more single quotation marks or double quotation marks.

  All paired single quotation marks are reduced to a single quotation mark.

  All paired double quotation marks are retained.

  If a double quotation mark is the second character, DEQUOTE removes the double quotation mark from the result. DEQUOTE then scans `expression` from left to right.
If a matching double quotation mark is found, the text between the double quotation marks is returned. Any text to the right of the closing double quotation mark, to the end of expression is removed from the result.

The first non-paired single quotation mark in expression is the closing single quotation mark and is removed.

If a close parentheses follows the close single quotation mark, the function returns the dequoted string. If characters exist to the right of the close single quotation mark, the function results in a syntax error and the error is printed in the SAS log.

- If expression is enclosed in double quotation marks, the DEQUOTE function returns a null or missing value.

Note: If expression is a constant enclosed in quotation marks, those quotation marks are not part of the value of expression. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Example

The following statements illustrate the DEQUOTE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>string=dequote(No quotation marks);</td>
<td>ERROR:   [HY000] Parse error.</td>
</tr>
<tr>
<td></td>
<td>Expecting ')' in statement x:</td>
</tr>
<tr>
<td></td>
<td>char(x) string; string=dequote(no =&gt;'quotation.</td>
</tr>
<tr>
<td></td>
<td>(0x817ff05c)</td>
</tr>
<tr>
<td>string=dequote(No 'leading'</td>
<td>ERROR:   [HY000] Parse error.</td>
</tr>
<tr>
<td>quotation marks);</td>
<td>Expecting ')' in statement x:</td>
</tr>
<tr>
<td></td>
<td>char(yy) string; string=dequote(No =&gt;'leading'.</td>
</tr>
<tr>
<td></td>
<td>(0x817ff05c)</td>
</tr>
<tr>
<td>string=dequote('Single matched</td>
<td>Single matched quotation marks are removed</td>
</tr>
<tr>
<td>quotation marks are removed');</td>
<td></td>
</tr>
<tr>
<td>string=dequote(&quot;Matched double</td>
<td>.</td>
</tr>
<tr>
<td>quotation marks result in a null or</td>
<td></td>
</tr>
<tr>
<td>missing value&quot;);</td>
<td></td>
</tr>
<tr>
<td>string=dequote('Paired 'single'</td>
<td>Paired 'single' quotation marks are reduced</td>
</tr>
<tr>
<td>quotation marks are reduced');</td>
<td></td>
</tr>
<tr>
<td>string=dequote(&quot;Double quotation</td>
<td>&quot;Double quotation marks&quot;</td>
</tr>
<tr>
<td>marks&quot; within &quot;single quotation marks&quot;,</td>
<td>with space before open quotation mark'</td>
</tr>
<tr>
<td>with space before open quotation mark&quot;);</td>
<td></td>
</tr>
<tr>
<td>string=dequote(&quot;Double quotation</td>
<td>Double quotation marks</td>
</tr>
<tr>
<td>marks&quot; within single quotation marks,</td>
<td>without space before open quotation mark'</td>
</tr>
<tr>
<td>without space before open quotation</td>
<td></td>
</tr>
<tr>
<td>mark&quot;);</td>
<td></td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
string=dequote('"Text after closing double quotation mark" is removed') | Text after closing double quotation mark

string=dequote('No matching quotation mark'); | Statement execution does not complete. Submit the following characters to complete the execution:
| '};

string=dequote('Identifiers after close quotation mark' results in a syntax error); | ERROR: [HY000]Parse error. Expecting ')' in statement x: string=dequote('Identifiers after close quotation mark' => results. (0x817ff05c)

### DEVIANCE Function

Returns the deviance based on a probability distribution.

**Category:** Mathematical  
**Returned data type:** DOUBLE

### Syntax

DEVIANCE(distribution, variable, shape-parameter(s)[, ε])

### Arguments

**distribution**

is a character constant, variable, or expression that identifies the distribution. Valid distributions are listed in the following table:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli (p. 473)</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Binomial (p. 473)</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Gamma (p. 474)</td>
<td>'GAMMA'</td>
</tr>
<tr>
<td>Inverse Gauss (Wald) (p. 474)</td>
<td>'IGAUSS'</td>
</tr>
<tr>
<td>Normal (p. 475)</td>
<td>'NORMAL'</td>
</tr>
<tr>
<td>Poisson (p. 475)</td>
<td>'POISSON'</td>
</tr>
</tbody>
</table>
**variable**

is a numeric constant, variable, or expression.

**shape-parameter(s)**

are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

**ε**

is an optional numeric small value used for all of the distributions, except for the normal distribution.

## Details

### The Bernoulli Distribution

**DEVIANCE('BERNOULLI', variable, p[, ε])**

**Arguments**

**variable**

is a binary numeric random variable that has the value of 1 for success and 0 for failure.

**p**

is a numeric probability of success with $ε \leq p \leq 1 – ε$.

ε

is an optional positive numeric value that is used to bound $p$. Any value of $p$ in the interval $0 \leq p \leq ε$ is replaced by $ε$. Any value of $p$ in the interval $1 – ε \leq p \leq 1$ is replaced by $1 – ε$.

The DEVIANCE function returns the deviance from a Bernoulli distribution with a probability of success $p$, where success is defined as a random variable value of 1. The equation follows:

$$\text{DEVIANCE('BERN', variable, p, ε)} = \begin{cases} -2\log(1 - p) & x = 0 \\ -2\log(p) & x = 1 \\ . & \text{otherwise} \end{cases}$$

### The Binomial Distribution

**DEVIANCE('BINO', variable, μ, n[, ε])**

**Arguments**

**variable**

is a numeric random variable that contains the number of successes.

Range $0 \leq \text{variable} \leq 1$

**μ**

is a numeric mean parameter.

Range $nε \leq \mu \leq n(1 – ε)$

**n**

is an integer number of Bernoulli trials parameter.

Range $n \geq 0$
ε is an optional positive numeric value that is used to bound μ. Any value of μ in the interval \(0 \leq μ \leq ne\) is replaced by \(ne\). Any value of μ in the interval \(n(1 - ε) \leq μ \leq n\) is replaced by \(n(1 - ε)\).

The DEVIANCE function returns the deviance from a binomial distribution, with a probability of success \(p\), and a number of independent Bernoulli trials \(n\). The following equation describes the DEVIANCE function for the Binomial distribution, where \(x\) is the random variable:

\[
DEVIANCE('BINO', x, μ, n) = \begin{cases} 
2 \left( x \log \left( \frac{x}{μ} \right) + (n - x) \log \left( \frac{n - x}{n - μ} \right) \right) & 0 \leq x \leq n \\
\cdot & x < 0 \\
\cdot & x > n 
\end{cases}
\]

**The Gamma Distribution**

\[
DEVIANCE('GAMMA', variable, μ[, ε])
\]

**Arguments**

**variable**

- is a numeric random variable.
- Range \(variable \geq ε\)

**μ**

- is a numeric mean parameter.
- Range \(μ \geq ε\)

**ε**

- is an optional positive numeric value that is used to bound \(variable\) and \(μ\). Any value of \(variable\) in the interval \(0 \leq variable \leq ε\) is replaced by \(ε\). Any value of \(μ\) in the interval \(0 \leq μ \leq ε\) is replaced by \(ε\).

The DEVIANCE function returns the deviance from a gamma distribution with a mean parameter \(μ\). The following equation describes the DEVIANCE function for the gamma distribution, where \(x\) is the random variable:

\[
DEVIANCE('GAMMA', x, μ) = \begin{cases} 
2 \left( - \log \left( \frac{x}{μ} \right) + \frac{x - μ}{μ} \right) & x, μ \geq ε, \ μ \geq ε \\
\cdot & x < 0 \\
\cdot & x > n 
\end{cases}
\]

**The Inverse Gauss (Wald) Distribution**

\[
DEVIANCE('IGAUSS' | 'WALD', variable, μ[, ε])
\]

**Arguments**

**variable**

- is a numeric random variable.
- Range \(variable \geq ε\)

**μ**

- is a numeric mean parameter.
- Range \(μ \geq ε\)
ε is an optional positive numeric value that is used to bound variable and μ. Any value of variable in the interval 0 ≤ variable ≤ ε is replaced by ε. Any value of μ in the interval 0 ≤ μ ≤ ε is replaced by ε.

The DEVIANCE function returns the deviance from an inverse Gaussian distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the inverse Gaussian distribution, where x is the random variable:

\[
\text{DEVIANCE}(\text{IGAUSS}, x, \mu) = \begin{cases} 
\frac{(x-\mu)^2}{\mu^2x} & x \geq \varepsilon, \mu \geq \varepsilon \\
. & x < 0 
\end{cases}
\]

**The Normal Distribution**

\[
\text{DEVIANCE}(\text{NORMAL} | \text{GAUSSIAN}, \text{variable, } \mu)
\]

**Arguments**

- **variable**
  - is a numeric random variable.

- **μ**
  - is a numeric mean parameter.

The DEVIANCE function returns the deviance from a normal distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the normal distribution, where x is the random variable:

\[
\text{DEVIANCE}(\text{NORMAL}, x, \mu) = (x - \mu)^2
\]

**The Poisson Distribution**

\[
\text{DEVIANCE}(\text{POISSON}, \text{variable, } \mu[\varepsilon])
\]

**Arguments**

- **variable**
  - is a numeric random variable.

  - Range \( \text{variable} \geq 0 \)

- **μ**
  - is a numeric mean parameter.

  - Range \( \mu \geq \varepsilon \)

- ε
  - is an optional positive numeric value that is used to bound μ. Any value of μ in the interval 0 ≤ μ ≤ ε is replaced by ε.

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the Poisson distribution, where x is the random variable:

\[
\text{DEVIANCE}(\text{POISSON}, x, \mu) = \begin{cases} 
2\left(x\log\left(\frac{x}{\mu}\right) - (x - \mu)\right) & x \geq 0, \mu \geq \varepsilon \\
. & x < 0 
\end{cases}
\]
DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

**Category:** Date and Time

**Returned data type:** DOUBLE

**Syntax**

\[ \text{DHMS}(date, \text{hour}, \text{minute}, \text{second}) \]

**Arguments**

- **date**
  - specifies any valid expression that represents a SAS date value.
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **hour**
  - specifies a numeric expression that represents an integer from 1 through 12.
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **minute**
  - specifies a numeric expression that represents an integer from 1 through 59.
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **second**
  - specifies a numeric expression that represents an integer from 1 through 59.
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93

**Details**

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

**Examples**

**Example 1: Using the DHMS Function**

The following statements illustrate the DHMS function:
**Example 2: Combining Date and Time Values**

The following statements illustrate how to combine a SAS date value with a SAS time value into a SAS datetime value, using the current day and time.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtval=dhms(mdy(12,31,2012),12,01,01);</code></td>
<td><code>1672574461</code></td>
</tr>
<tr>
<td><code>day=date();</code></td>
<td><code>1622021987.6</code></td>
</tr>
<tr>
<td><code>time=time();</code></td>
<td></td>
</tr>
<tr>
<td><code>dt=dhms(day,0,0,time);</code></td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Functions:**

- “HMS Function” on page 528

---

**DIGAMMA Function**

Returns the value of the digamma function.

**Category:** Mathematical

**Returned data type:** DOUBLE

**Syntax**

\[
\text{DIGAMMA}(expression)
\]

**Arguments**

`expression`

specifies any valid expression that evaluates to a numeric value.

**Restriction**

Zero and negative integers are not valid.

**Data type**

DOUBLE

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The DIGAMMA function returns the ratio that is given by the following equation.

\[
\Psi(x) = \Gamma'(x)/\Gamma(x)
\]
\( \Gamma(.) \) and \( \Gamma'(.) \) denote the Gamma function and its derivative, respectively. For \( \text{expression}>0 \), the DIGAMMA function is the derivative of the LGAMMA function.

**Example**

The following statement illustrates the DIGAMMA function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x=\text{digamma}(1.0); )</td>
<td>(-0.57721566490153)</td>
</tr>
</tbody>
</table>

**DIM Function**

Returns the number of elements in an array.

**Category:** Array

**Returned data type:** INTEGER

**Syntax**

\[
\text{DIM}(\text{array-name}[, \text{bound-n}])
\]

**Arguments**

- **array-name**
  specifies the name of a temporary or a variable array.

- **bound-n**
  is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the number of elements. If no \( \text{bound-n} \) value is specified, the DIM function returns the number of elements in the first dimension of the array.

  \( \text{Bound-n} \) evaluates to an integral value.

**Data type** INTEGER

**Details**

The DIM function returns the number of elements in a one-dimensional array, or the number of elements in a specified dimension of a multidimensional array.

If the DIM function is called with a \( \text{bound-n} \) dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

**Comparisons**

- DIM returns the number of elements in an array dimension.
- HBOUND returns the value of the upper bound of an array dimension.
LBOUND returns the value of the lower bound of an array dimension.
NDIMS returns the number of dimensions in an array.

Example

The following example shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```sas
data _null_;  
  method init();  
    declare char(15) a1[4];  
    declare double a2[2,3,4] sum;  
    a1 := ('red' 'yellow' 'green' 'blue');  
    a2 := (24*2.0);  
    do i = 1 to dim(a1);  
      put a1[i];  
    end;  
    numelems = 0;  
    do i = 1 to ndims(a2);  
      numelems = numelems + dim(a2, i);  
    end;  
    sum = 0;  
    do i = lbound(a2, 1) to hbound(a2, 1);  
      do j = lbound(a2, 2) to hbound(a2, 2);  
        do k = lbound(a2, 3) to hbound(a2, 3);  
          sum = sum + a2[i,j,k];  
        end;  
      end;  
    end;  
    put sum=;  
  end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

See Also

Functions:
- “HBOUND Function” on page 526
- “LBOUND Function” on page 600
DIVIDE Function

Returns the result of a division that handles special missing values for ODS output.

**Category:** Arithmetic  
**Returned data type:** DOUBLE

---

**Syntax**

\[ \text{DIVIDE}(x, y) \]

**Arguments**

\( x \)
- specifies any valid expression that evaluates to a numeric value.  
  - Data type: DOUBLE  
  - See: Chapter 13, “DS2 Expressions,” on page 93

\( y \)
- specifies any valid expression that evaluates to a numeric value.  
  - Data type: DOUBLE  
  - See: Chapter 13, “DS2 Expressions,” on page 93

---

**Details**

The DIVIDE function divides two numbers and returns a result that is compatible with ODS conventions. The function handles special missing values for ODS output. The following list shows how certain special missing values are interpreted in ODS:

- .I as infinity
- .M as minus infinity
- _ as a blank

The following table shows the values that are returned by the DIVIDE function, based on the values of \( x \) and \( y \).
Figure 23.1 Values That Are Returned by the DIVIDE Function

The DIVIDE function never writes a note to the SAS log regarding missing values, division by zero, or overflow.

<table>
<thead>
<tr>
<th>x</th>
<th>positive</th>
<th>zero</th>
<th>negative</th>
<th>.I</th>
<th>.M</th>
<th>__</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>x/y or .I</td>
<td>0</td>
<td>x/y or .M</td>
<td>.I</td>
<td>.M</td>
<td>__</td>
<td>x</td>
</tr>
<tr>
<td>.I</td>
<td>.I</td>
<td>.M</td>
<td>.I</td>
<td>.M</td>
<td>__</td>
<td>__</td>
<td>x</td>
</tr>
<tr>
<td>.M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>__</td>
<td>__</td>
<td>x</td>
</tr>
<tr>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
<td>__</td>
</tr>
<tr>
<td>other</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>__</td>
<td>__</td>
<td>x</td>
</tr>
</tbody>
</table>

Note: The DIVIDE function never writes a note to the SAS log regarding missing values, division by zero, or overflow.

Example

The following example shows the results of using the DIVIDE function.

```sas
data test (overwrite=yes);
dcl double a b c d;
method run();
a=divide(1, 0);
put a='(infinity)';
b=divide(2, .I);
put b=;
c=divide(.I, -1);
put c='(minus infinity)';
d=divide(constant('big'), constant('small'));
put d='(infinity because of overflow)';
end;
enddata;
run;
```

The following lines are written to the SAS log:

```
a=I  (infinity)
b=0
 c=M  (minus infinity)
d=I  (infinity because of overflow)
```

DUR Function

Returns the modified duration for an enumerated cash flow.

Category: Financial

Returned data type: DOUBLE
Syntax

\[ \text{DUR}(y, f, c(1), \ldots, c(k)) \]

Arguments

\( y \)

specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range \( y > 0 \)

Data type \( \text{DOUBLE} \)

\( f \)

specifies the frequency of cash flows per period.

Range \( f > 0 \)

Data type \( \text{DOUBLE} \)

\( c(1), \ldots, c(k) \)

specifies a list of cash flows.

Data type \( \text{DOUBLE} \)

Details

The DUR function returns the value from the following equation.

\[
C = \sum_{k=1}^{K} \left( \frac{c(k)}{k} \right) \frac{1}{(1+y)^f} \frac{1}{(P(1+y))^f}
\]

The following relationship applies to the preceding equation:

\[
P = \sum_{k=1}^{K} \frac{c(k)}{k} \frac{1}{(1+y)^f}
\]

Example: Using the DUR Function

```sas
data test;
  dcl double d;
  method run();
    d=dur(.05,1,.33,.44,.49,.55,.50,.22,.4,.8,.01,.36,.2,.4);
    put d;
  end;
enddata;
run;
```

SAS writes the following output to the log:

5.28402498798216
See Also

Functions:

- “DURP Function” on page 483

---

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

\[ \text{DURP}(A, c, n, K, k_0, y) \]

**Arguments**

**A**

specifies the par value.

Range \( A > 0 \)

Data type DOUBLE

**c**

specifies the nominal per-period coupon rate, expressed as a fraction.

Range \( 0 \leq c < 1 \)

Data type DOUBLE

**n**

specifies the number of coupons per period.

Range \( n > 0 \) and is an integer

Data type DOUBLE

**K**

specifies the number of remaining coupons.

Range \( K > 0 \) and is an integer

Data type DOUBLE

**k_0**

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range \( 0 < k_0 \leq 1/n \)
Data type: DOUBLE

\( y \)
specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range: \( y > 0 \)

Data type: DOUBLE

**Details**

The DURP function returns the value from the following equation.

\[
D = \frac{1}{n} \sum_{k=1}^{K} \frac{c(k) t_k}{(1 + \frac{y}{n})^{t_k}}
\]

The following relationships apply to the preceding equation:

- \( t_k = nk_0 + k - 1 \)
- \( c(k) = \frac{c}{n^k}A \) for \( k = 1, \ldots, K - 1 \)
- \( c(K) = (1 + \frac{c}{n^k})A \)

The following relationship applies to the preceding equation:

\[
P = \sum_{k=1}^{K} \frac{c(k) t_k}{(1 + \frac{y}{n})^{t_k}}
\]

**Example: Using the DURP Function**

```sas
data test;
  dcl double d;
  method run();
    d=durp(1000,1/100,4,14,.33/2,.10);
    put d;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
3.33170731707317
```

**See Also**

Functions:
- “DUR Function” on page 481
EFFRATE Function

Returns the effective annual interest rate.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

\[
\text{EFFRATE}(\text{compounding-interval, rate})
\]

**Arguments**

- **compounding-interval**
  is a SAS interval. This value represents how often rate compounds.
  
  **Data type** CHAR

- **rate**
  is numeric. Rate is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.
  
  **Data type** DOUBLE

**Details**

The EFFRATE function returns the effective annual interest rate. The function computes the effective annual interest rate that corresponds to a nominal annual interest rate.

The following details apply to the EFFRATE function:

- The values for rates must be at least −99.
- In considering a nominal interest rate and a compounding interval, if compounding-interval is 'CONTINUOUS', then the value that is returned by EFFRATE equals \( e^{\frac{\text{rate}}{100} - 1} \).
  
  If compounding-interval is not 'CONTINUOUS', and \( m \) compounding intervals occur in a year, the value that is returned by EFFRATE equals \( (1 + \left[ \frac{\text{rate}}{100} \right]^{\frac{1}{m}})^{m-1} \).
- The following values are valid for compounding-interval:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then \( m=365 \).
Example

The following examples show how the effective rate is calculated:

- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded monthly can be expressed as
  \[
  \text{effective-rate}_1 = \text{EFFRATE('MONTH', 10)};
  \]

- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded quarterly can be expressed as
  \[
  \text{effective-rate}_2 = \text{EFFRATE('QUARTER', 10)};
  \]

ERF Function

Returns the value of the (normal) error function.

**Categories:**
- Mathematical
- Arithmetic

**Returned data type:**
DOUBLE

**Syntax**

\[
\text{ERF}(\text{expression})
\]

**Arguments**

- **expression**
  - specifies any valid expression that evaluates to a numeric value.
  - **Data type**: DOUBLE

**See**
Chapter 13, “DS2 Expressions,” on page 93

**Details**

The ERF function returns the integral, given by the following:

\[
\text{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-z^2} dz
\]

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation will take on a value less than X. For example, the quantity that is given by the following statement is equivalent to PROBNORM(X):

\[
p = .5 + .5 \cdot \text{erf}(x/\sqrt{2})
\]

**Example**

The following statements illustrate the ERF function:
ERFC Function

Returns the value of the complementary (normal) error function.

**Categories:** Mathematical, Arithmetic

**Restriction:** DOUBLE

**Syntax**

\[
\text{ERFC(expression)}
\]

**Arguments**

- **expression**: specifies any valid expression that evaluates to a numeric value.
  - **Data type**: DOUBLE

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The ERFC function returns the complement to the ERF function (that is, \(1 - \text{ERF}(\text{argument})\)).

**Example**

The following statements illustrate the ERFC function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x=\text{erfc}(1.0);)</td>
<td>(0.157299)</td>
</tr>
<tr>
<td>(x=\text{erfc}(-1.0);)</td>
<td>(1.842701)</td>
</tr>
</tbody>
</table>
EXP Function

Returns the value of the e constant raised to a specified power.

**Category:** Mathematical

**Returned data type:** DOUBLE

### Syntax

EXP(expression)

### Arguments

**expression**

specifies any valid expression that evaluates to a numeric value.

**Data type** DOUBLE

### Details

The EXP function raises the constant $e$, which is approximately given by 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a double decimal value on the computer.

### Example

The following statements illustrate the EXP function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=exp(1.0);</td>
<td>2.71828182845904</td>
</tr>
<tr>
<td>a=exp(0);</td>
<td>1</td>
</tr>
</tbody>
</table>

FACT Function

Computes a factorial.

**Category:** Mathematical
Syntax

FACT(expression)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type: INTEGER

See Chapter 13, “DS2 Expressions,” on page 93

Details

The mathematical representation of the FACT function is given by the following equation:

\[ FACT(n) = n! \]

In this equation, \( n \geq 0 \).

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the FACT function.

Example

The following statement illustrates the FACT function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=fact(5);</td>
<td>120</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “COMB Function” on page 428
- “PERM Function” on page 678

FIND Function

Searches for a specific substring of characters within a character string.

Category: Character

Returned data type: CHAR

Syntax

\[
\text{FIND}(\text{string, substring[, modifier(s)][, startpos]})
\]

Arguments

\textbf{string}

specifies a character constant, variable, or expression that will be searched for substrings.

Data type: CHAR

Tip: Enclose a literal string of characters in quotation marks.

\textbf{substring}

is a character constant, variable, or expression that specifies the substring of characters to search for in \textit{string}.

Data type: CHAR

Tip: Enclose a literal string of characters in quotation marks.

\textbf{modifier(s)}

is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

\begin{itemize}
  \item \textit{I} or \textit{I} ignores character case during the search. If this modifier is not specified, FIND only searches for character substrings with the same case as the characters in \textit{substring}.
  \item \textit{T} or \textit{T} trims trailing blanks from \textit{string} and \textit{substring}.
\end{itemize}

Note: If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the FIND function with the \textit{T} modifier.

Data type: CHAR

Tip: If \textit{modifier} is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. \textit{Modifier} can also be expressed as a variable or an expression.

\textbf{startpos}

is a numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction of the search.

Data type: INTEGER
Details

The FIND function searches string for the first occurrence of the specified substring, and returns the position of that substring. If the substring is not found in string, FIND returns a value of 0.

If startpos is not specified, FIND starts the search at the beginning of the string and searches the string from left to right. If startpos is specified, the absolute value of startpos determines the position at which to start the search. The sign of startpos determines the direction of the search.

<table>
<thead>
<tr>
<th>Value of startpos</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater than 0</td>
<td>starts the search at position startpos and the direction of the search is to the right. If startpos is greater than the length of string, FIND returns a value of 0.</td>
</tr>
<tr>
<td>less than 0</td>
<td>starts the search at position –startpos and the direction of the search is to the left. If –startpos is greater than the length of string, the search starts at the end of string.</td>
</tr>
<tr>
<td>equal to 0</td>
<td>returns a value of 0.</td>
</tr>
</tbody>
</table>

Comparisons

- The FIND function searches for substrings of characters in a character string, whereas the FINDC function searches for individual characters in a character string.
- The FIND function and the INDEX function both search for substrings of characters in a character string. However, the INDEX function does not have the modifiers nor the startpos arguments.

Example

The following statements illustrate the FIND function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>whereisshe=find('She sells seashells? Yes, she does.','she ');</td>
<td>27</td>
</tr>
<tr>
<td>put whereisshe;</td>
<td></td>
</tr>
<tr>
<td>variable1='She sells seashells? Yes, she does.';</td>
<td>1</td>
</tr>
<tr>
<td>variable2='she ';</td>
<td></td>
</tr>
<tr>
<td>variable3='i';</td>
<td></td>
</tr>
<tr>
<td>whereisshe_i=find(variable1,variable2,variable3);</td>
<td></td>
</tr>
<tr>
<td>put whereisshe_i;</td>
<td></td>
</tr>
<tr>
<td>expression1='She sells seashells? '</td>
<td></td>
</tr>
<tr>
<td>expression2=kscan('he or she',3)</td>
<td></td>
</tr>
<tr>
<td>expression3=trim('t ');</td>
<td></td>
</tr>
<tr>
<td>whereisshe_t=find(expression1,expression2,expression3);</td>
<td></td>
</tr>
<tr>
<td>put whereisshe_t;</td>
<td></td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
xyz='She sells seashells? Yes, she does.';
startposvar=22;
whereisshe_22=find(xyz, 'she', startposvar);
put whereisshe_22;

27

xyz='She sells seashells? Yes, she does.';
startposexp=1-23;
whereisShe_ineg22=find(xyz, 'She', 'i', startposexp);
put whereisShe_ineg22;

14

See Also

Functions:

- “COUNT Function” on page 449
- “FINDC Function” on page 492
- “FINDW Function” on page 499
- “INDEX Function” on page 533

FINDC Function

Searches a string for any character in a list of characters.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Syntax

FINDC(\textit{string[, charlist]})

FINDC(\textit{string, charlist[, modifier]})

FINDC(\textit{string, charlist, modifier[, startpos]})

FINDC(\textit{string, charlist[, startpos][, modifier]})

Arguments

\textit{string}

is a character constant, variable, or expression that specifies the character string to be searched.

Data type | CHAR
---|---

Tip | Enclose a literal string of characters in quotation marks.
charlist

is a constant, variable, or character expression that initializes a list of characters. FINDC searches for the characters in this list provided that you do not specify the K modifier in the modifier argument. If you specify the K modifier, FINDC searches for all characters that are not in this list of characters. You can add more characters to the list by using other modifiers.

Data type  CHAR

Tip  Enclose a literal string of characters in quotation marks.

modifier

is a character constant, variable, or expression in which each character modifies the action of the FINDC function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank  is ignored.

a or A  adds alphabetic characters to the list of characters.

b or B  searches from right to left, instead of from left to right, regardless of the sign of the startpos argument.

c or C  adds control characters to the list of characters.

d or D  adds digits to the list of characters.

f or F  adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.

g or G  adds graphic characters to the list of characters.

h or H  adds a horizontal tab to the list of characters.

i or I  ignores character case during the search.

k or K  searches for any character that does not appear in the list of characters. If you do not specify this modifier, then FINDC searches for any character that appears in the list of characters. The V and K modifiers perform the same function.

l or L  adds lowercase letters to the list of characters.

n or N  adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O  processes the charlist and the modifier arguments only once, rather than every time the FINDC function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDC run faster when you call it in a loop where the charlist and the modifier arguments do not change.

p or P  adds punctuation marks to the list of characters.

s or S  adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T  trims trailing blanks from the string and charlist arguments. Note that if you want to remove trailing blanks from just one character argument instead of both (or all) character arguments, use the TRIM function instead of the FINDC function with the T modifier.
u or U adds uppercase letters to the list of characters.
v or V causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.
w or W adds printable characters to the list of characters.
x or X adds hexadecimal characters to the list of characters.

Data type CHAR

Tip If modifier is a constant, then enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression.

Startpos is an optional numeric constant, variable, or expression having an integer value that specifies the position at which the search should start and the direction in which to search.

Data type INTEGER

Details

The FINDC function searches string for the first occurrence of the specified characters, and returns the position of the first character found. If no characters are found in string, then FINDC returns a value of 0.

The FINDC function allows character arguments to be null. Null arguments are treated as character strings that have a length of zero. Numeric arguments cannot be null.

If startpos is not specified, FINDC begins the search at the end of the string if you use the B modifier, or at the beginning of the string if you do not use the B modifier.

If startpos is specified, the absolute value of startpos specifies the position at which to begin the search. If you use the B modifier, the search always proceeds from right to left. If you do not use the B modifier, the sign of startpos specifies the direction in which to search. The following table summarizes the search directions:

<table>
<thead>
<tr>
<th>Value of startpos</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater than 0</td>
<td>search begins at position startpos and proceeds to the right. If startpos is greater than the length of the string, FINDC returns a value of 0.</td>
</tr>
<tr>
<td>less than 0</td>
<td>search begins at position –startpos and proceeds to the left. If startpos is less than the negative of the length of the string, the search begins at the end of the string.</td>
</tr>
<tr>
<td>equal to 0</td>
<td>returns a value of 0.</td>
</tr>
</tbody>
</table>

Comparisons

- The FINDC function searches for individual characters in a character string, whereas the FIND function searches for substrings of characters in a character string.
• The FINDC function and the INDEXC function both search for individual characters in a character string. However, the INDEXC function does not have the `modifier` nor the `startpos` arguments.

• The FINDC function searches for individual characters in a character string, whereas the VERIFY function searches for the first character that is unique to an expression. The VERIFY function does not have the `modifier` nor the `startpos` arguments.

Examples

**Example 1: Searching for Characters in a String**
This example searches a character string and returns the characters that are found.

```sas
data test;
  method run();

  j=0;
  do until(j=0);
    j = findc('Hi, ho!','hi',j+1);
    if j= 0 then put 'The End';
    else do;
      c = substr('Hi, ho!', j, 1);
      put j= c=;
    end;
  end;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=2 c=i
j=5 c=h
The End
```

**Example 2: Searching for Characters in a String and Ignoring Case**
This example searches a character string and returns the characters that are found. The `i` modifier is used to ignore the case of the characters.

```sas
data test;
  method run();

  string='Hi, ho!';
  charlist='ho';
  j=0;
  do until(j=0);
    j=findc(string,charlist,j+1,'i');
    if j=0 then put 'The End';
    else do;
      c=substr(string, j, 1);
      put j= c=;
    end;
  end;
end;
enddata;
run;
```
SAS writes the following output to the log:

```
j=1  c=H
j=5  c=h
j=6  c=o
The End
```

**Example 3: Searching for Characters and Using the K Modifier**
This example searches a character string and returns the characters that do not appear in the character list.

```
data test;
  method run();
    string='Hi, ho!';
    charlist='hi';
    j=0;
    do until(j = 0);
      j = findc(string,charlist,'k',j+1);
      if j=0 then put 'The End';
      else do;
        c = substr(string,j,1);
        put j= c=;
      end;
    end;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1  c=H
j=3  c=,
j=4  c=
j=6  c=o
j=7  c=!
The End
```

**Example 4: Searching for the Characters h, i, and Blank**
This example searches for the three characters h, i, and blank. The characters h and i are in lowercase. The uppercase characters H and I are ignored in this search.

```
data test;
  method run();
    whereishi=0;
    do until(whereishi=0);
      whereishi=findc('Hi there, Ian!','hi ',whereishi+1);
      if whereishi=0 then put 'The End';
      else do;
        whatfound=substr('Hi there, Ian!',whereishi,1);
        put whereishi= whatfound=;
      end;
    end;
  end;
enddata;
run;
```
SAS writes the following output to the log:

```
whereishi=2 whatfound=i
whereishi=3 whatfound=
whereishi=5 whatfound=h
whereishi=10 whatfound=
The End
```

**Example 5: Searching for the Characters h and i While Ignoring Case**

This example searches for the four characters h, i, H, and I. FINDC with the i modifier ignores character case during the search.

```sas
data test;
  method run();
  whereishi=0;
  do until(whereishi=0);
    whereishi=findc('Hi there, Ian!','hi ',whereishi+1);
    if whereishi=0 then put 'The End';
    else do;
      whatfound=substr('Hi there, Ian!',whereishi,1);
      put whereishi= whatfound=;
    end;
  end;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
whereishi_i=1 whatfound=H
whereishi_i=2 whatfound=i
whereishi_i=5 whatfound=h
whereishi_i=11 whatfound=I
The End
```

**Example 6: Searching for the Characters h and i with Trailing Blanks Trimmed**

This example searches for the two characters h and i. FINDC with the t modifier trims trailing blanks from the string argument and the characters argument.

```sas
data test;
  method run();
  whereishi_t=0;
  do until(whereishi_t=0);
    expression1='Hi there, '||'Ian!';
    expression2=kscan('bye or hi',3)||'  ';
    expression3=trim('t   ');
    whereishi_t=findc(expression1,expression2,expression3,whereishi_t+1);
    if whereishi_t=0 then put 'The End';
    else do;
      whatfound=substr(expression1,whereishi_t,1);
      put whereishi_t= whatfound=;
    end;
  end;
end;
enddata;
```
run;
SAS writes the following lines output to the log:

```sas
whereishi_t=2 whatfound=i
whereishi_t=5 whatfound=h
The End
```

**Example 7: Searching for All Characters, Excluding h, i, H, and I**
This example searches for all of the characters in the string, excluding the characters h, i, H, and I. FINDC with the v modifier counts only the characters that do not appear in the characters argument. This example also includes the i modifier and therefore ignores character case during the search.

data test (overwrite=yes);
  method run();
    whereishi_iv=0;
    do until(whereishi_iv=0);
      xyz='Hi there, Ian!';
      whereishi_iv=findc(xyz,'hi',whereishi_iv+1,'iv');
      if whereishi_iv=0 then put 'The End';
      else do;
        whatfound=substr(xyz,whereishi_iv,1);
        put whereishi_iv= whatfound=;
      end;
    end;
  enddata;
run;
quit;
SAS writes the following output to the log:

```sas
whereishi_iv=3 whatfound=
whereishi_iv=4 whatfound=t
whereishi_iv=6 whatfound=e
whereishi_iv=7 whatfound=r
whereishi_iv=8 whatfound=e
whereishi_iv=9 whatfound=,
whereishi_iv=10 whatfound=
whereishi_iv=12 whatfound=n
whereishi_iv=13 whatfound=n
whereishi_iv=14 whatfound=! 
The End
```

**See Also**

**Functions:**
- “ANYALNUM Function” on page 359
- “ANYALPHA Function” on page 362
- “ANYCNTRL Function” on page 364
- “ANYDIGIT Function” on page 366
- “ANYGRAPH Function” on page 370
- “ANYLOWER Function” on page 372
FINDW Function

Returns the character position of a word in a string, or returns the number of the word in a string.

**Category:** Character

**Returned data type:** INTEGER

**Syntax**

FINDW\( (\text{string, word[, chars]}))\)

FINDW\( (\text{string, word, chars, modifier(s)[, startpos]}))\)

FINDW\( (\text{string, word, chars, startpos[, modifier(s)]})\)

FINDW\( (\text{string, word, startpos[, chars[, modifier(s)]]})\)\)

**Arguments**

**string**

is a character constant, variable, or expression that specifies the character string to be searched.

**Data type** CHAR
Tip
Enclose a literal string of characters in quotation marks.

**word**

is a character constant, variable, or expression that specifies the word to be searched.

**Data type** CHAR

Tip
Enclose a literal string of characters in quotation marks.

**chars**

is an optional character constant, variable, or expression that initializes a list of characters.

The characters in this list are the delimiters that separate words, provided that you do not specify the K modifier in the modifier argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to this list by using other modifiers.

**Data type** CHAR

Tip
Enclose a literal string of characters in quotation marks.

**startpos**

is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should begin and the direction in which to search.

**Data type** INTEGER

'**modifier(s)**'

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function.

You can use the following characters as modifiers:

**blank** is ignored.

**a or A** adds alphabetic characters to the list of characters.

**b or B** searches from right to left, instead of from left to right, regardless of the sign of the startpos argument.

**c or C** adds control characters to the list of characters.

**d or D** adds digits to the list of characters.

**e or E** counts the words that are scanned until the specified word is found, instead of determining the character position of the specified word in the string. Fragments of words are not counted.

**f or F** adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.

**g or G** adds graphic characters to the list of characters.

**h or H** adds a horizontal tab to the list of characters.

**i or I** ignores the case of the characters.

**k or K** causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list
of characters are treated as delimiters. The K and V modifiers perform the same function.

l or L adds lowercase letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the string argument, refer to words that have a length of zero.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the chars and the modifier arguments only once, rather than every time the FINDW function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDW run faster when you call it in a loop where the chars and the modifier arguments do not change.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the string argument contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.

r or R removes leading and trailing delimiters from the word argument.

s or S adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.

t or T trims trailing blanks from the string, word, and chars arguments.

u or U adds uppercase letters to the list of characters.

v or V causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.

w or W adds printable characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

**Data type** CHAR

**Tip** If you use the modifier argument, then it must be positioned after the chars argument.

---

**Details**

**Definition of "Delimiter"**

"Delimiter" refers to any of several characters that are used to separate words. You can specify the delimiters by using the chars argument, the modifier argument, or both. If you specify the Q modifier, then the characters inside substrings that are enclosed in quotation marks are not treated as delimiters.

**Definition of "Word"**

"Word" refers to a substring that has both of the following characteristics:

- bounded on the left by a delimiter or the beginning of the string
• bounded on the right by a delimiter or the end of the string

*Note:* A word can contain delimiters. In this case, the FINDW function differs from the SCAN function, in which words are defined as not containing delimiters.

**Searching for a String**

If the FINDW function fails to find a substring that both matches the specified word and satisfies the definition of a word, then FINDW returns a value of 0.

If the FINDW function finds a substring that both matches the specified word and satisfies the definition of a word, the value that is returned by FINDW depends on whether the E modifier is specified:

• If you specify the E modifier, then FINDW returns the number of complete words that were scanned while searching for the specified word. If \textit{startpos} specifies a position in the middle of a word, then that word is not counted.

• If you do not specify the E modifier, then FINDW returns the character position of the substring that is found.

If you specify the \textit{startpos} argument, then the absolute value of \textit{startpos} specifies the position at which to begin the search. The sign of \textit{startpos} specifies the direction in which to search:

<table>
<thead>
<tr>
<th>Value of \textit{startpos}</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater than 0</td>
<td>search begins at position \textit{startpos} and proceeds to the right. If \textit{startpos} is greater than the length of the string, then FINDW returns a value of 0.</td>
</tr>
<tr>
<td>less than 0</td>
<td>search begins at position –\textit{startpos} and proceeds to the left. If \textit{startpos} is less than the negative of the length of the string, then the search begins at the end of the string.</td>
</tr>
<tr>
<td>equal to 0</td>
<td>FINDW returns a value of 0.</td>
</tr>
</tbody>
</table>

If you do not specify the \textit{startpos} argument or the B modifier, then FINDW searches from left to right starting at the beginning of the string. If you specify the B modifier, but do not use the \textit{startpos} argument, then FINDW searches from right to left starting at the end of the string.

**Using the FINDW Function in ASCII and EBCDIC Environments**

If you use the FINDW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

• If your computer uses ASCII characters, then the default delimiters are as follows:
  blank ! $ % & ( ) * + , - . / ; < ^ |
  In ASCII environments that do not contain the ^ character, the FINDW function uses the ¬ character instead.

• If your computer uses EBCDIC characters, then the default delimiters are as follows:
  blank ! $ % & ( ) * + , - . / ; < ¬ | ©
Using Null Arguments
The FINDW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Examples

Example 1: Searching a Character String for a Word
The following example searches a character string for the word “she”, and returns the position of the beginning of the word.

```sas
data _null_; 
  method run(); 
    whereisshe=findw('She sells sea shells? Yes, she does.','she'); 
    put whereisshe=; 
  end; 
enddata; 
run; 
```

SAS writes the following output to the log:

```
whereisshe=28
```

Example 2: Searching a Character String and Using the Chars and Startpos Arguments
The following example contains two occurrences of the word “rain.” Only the second occurrence is found by FINDW because the search begins in position 25. The `chars` argument specifies a space as the delimiter.

```sas
data _null_; 
  method run(); 
    result = findw('At least 2.5 meters of rain falls in a rain forest.','rain',' ', 25); 
    put result=; 
  end; 
enddata; 
run; 
```

SAS writes the following output to the log:

```
result=40
```

Example 3: Searching a Character String and Using the I Modifier and the Startpos Argument
The following example uses the I modifier and returns the position of the beginning of the word. The I modifier disregards case, and the `startpos` argument identifies the starting position from which to search.

```sas
data _null_; 
  method run(); 
    string='Artists from around the country display their art at an art festival.'; 
    result=findw(string, 'Art',' ', 'i', 10); 
    put result=; 
  end; 
enddata; 
```
run;
quit;

SAS writes the following output to the log:

```
result=47
```

**Example 4: Searching a Character String and Using the E Modifier**
The following example uses the E modifier and returns the number of complete words that are scanned while searching for the word "art."

```sas
data _null_;  
  method run();  
  string='Artists from around the country display their art at an art festival.';
  result=findw(string,'art',' ','E');
  put result=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=8
```

**Example 5: Searching a Character String and Using the E Modifier and the Startpos Argument**
The following example uses the E modifier to count words in a character string. The word count begins at position 50 in the string. The result is 3 because "art" is the third word after the 50th character position.

```sas
data _null_;  
  method run();  
  string='Artists from around the country display their art at an art festival.';
  result=findw(string, 'art',' ','E',50);
  put result=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=3
```

**Example 6: Searching a Character String and Using Two Modifiers**
The following example uses the I and the E modifiers to find a word in a string.

```sas
data _null_;  
  method run();  
  string='The Great Himalayan National Park was created in 1984. Because of its terrain and altitude, the park supports a diversity of wildlife and vegetation.';
  result=findw(string,'park',' ','I E');
  put result=;
  end;
```
enddata;
run;

SAS writes the following output to the log:

```
result=5
```

**Example 7: Searching a Character String and Using the R Modifier**
The following example uses the R modifier to remove leading and trailing delimiters from a word.

```sas
data _null_; method run();
  string='Artists from around the country display their art at an art festival.';
  word='  art  ';
  result=findw(string, word, ' ', 'R');
  put result=;
end;
enddata;
run;

SAS writes the following output to the log:

```
result=47
```

**See Also**

**Functions:**
- “COUNTW Function” on page 454
- “FIND Function” on page 489
- “FINDC Function” on page 492
- “INDEXW Function” on page 536
- “SCAN Function” on page 772

---

**FLOOR Function**

Returns the largest integer less than or equal to a numeric value expression.

- **Category:** Truncation
- **Returned data type:** DECIMAL, DOUBLE, NUMERIC

**Syntax**

```
FLOOR(expression)
```
Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type

DECIMAL, DOUBLE, NUMERIC

See

Chapter 13, “DS2 Expressions,” on page 93

Details

If expression is within 1E-12 of an integer, the function returns that integer. If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

The FLOOR function fuzzes the results so that if the results are within 1E-12 of an integer, the FLOOR function returns that integer. The FLOORZ function uses zero fuzzing. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statement illustrates the FLOOR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>floor(1.95);</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 420
- “FLOORZ Function” on page 506

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Category: Truncation

Returned data type: DOUBLE

Syntax

FLOORZ(expression)
Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type: DOUBLE

See: Chapter 13, “DS2 Expressions,” on page 93

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statements illustrate the FLOORZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=2.1;</td>
<td>2</td>
</tr>
<tr>
<td>a=floorz(var1);</td>
<td></td>
</tr>
<tr>
<td>b=floorz(-2.4);</td>
<td>-3</td>
</tr>
<tr>
<td>c=floorz(-1.6);</td>
<td>-2</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 420
- “FLOOR Function” on page 505

FMTINFO Function

Returns information about a SAS format or informat.

Restriction: This function returns information about formats that are supplied by SAS. It cannot be used for user-defined formats that are created with the FORMAT procedure.

Syntax

FMTINFO('format-name', 'information-type');
Arguments

'format-name'

specifies the name of a SAS format or informat.

Requirement format-name must be enclosed in single quotation marks.

information-type

specifies the type of information that is returned. format-information can be one of the following values:

'CAT'
returns the function category.

See For a complete list, see “Function Categories” on page 343.

'TYPE'
returns whether the format-name is a format, an informat, or both.

'DESC'
returns a short description of the format or informat.

'MIND'
returns the minimum number of digits to the right of the decimal place in the format or informat.

'MAXD'
returns the maximum number of digits to the right of the decimal place in the format or informat.

'DEFD'
returns the default number of digits to the right of the decimal place in the format or informat.

'MINW'
returns the minimum width value of the format or informat.

'MAXW'
returns the maximum width value of the format or informat.

'DEFW'
returns the default width value of the format or informat.

Restriction You can specify only one information-type argument.

Requirement information-type must be enclosed in single quotation marks.

Details

The FMTINFO function returns information about a format or informat. You can return information about a format or informat’s category, the type of language element, a description of the language element, and the minimum, maximum, and default decimal and width values.

You cannot specify multiple arguments with the FMTINFO function.

The FMTINFO function returns a character string for all data values, including the numeric value arguments MIND, MAXD, DEFD, MINW, MAXW, and DEFW.
Example

The following example returns information about the COMMAw. and COMMAw.d informat.

```sas
data _null_;
dcl char(30) fdesc fcat ftype;
dcl double fmind fmaxd fdefd fminw fmaxw fdefw;
method run();
  ftype=fmtinfo('date','type');
  fcat= fmtinfo('date','cat');
  fdesc= fmtinfo('date','desc');
  fmind= fmtinfo('date','mind');
  fmaxd= fmtinfo('date','maxd');
  fdefd= fmtinfo('date','defd');
  fminw= fmtinfo('date','minw');
  fmaxw= fmtinfo('date','maxw');
  fdefw= fmtinfo('date','defw');
  put ftype=;
  put fcat=;
  put fdesc= ;
  put fmind= ;
  put fmaxd= ;
  put fdefd= ;
  put fminw= ;
  put fmaxw= ;
  put fdefw= ;
end;
enddata;
run;
```

The following lines are written to the SAS log.

```
ftype=BOTH
fcat=date
fdesc=date value
fmind=0
fmaxd=8
fdefd=0
fminw=5
fmaxw=11
fdefw=7
```

**FUZZ Function**

Returns the nearest integer if the argument is within 1E-12 of that integer.

- **Category:** Truncation
- **Returned data type:** DOUBLE

**Syntax**

```
FUZZ(expression)
```
Arguments

data: DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The FUZZ function returns the nearest integer value if the expression is within 1E-12 of the integer (that is, if the absolute difference between the integer and argument is less than 1E-12). Otherwise, the expression is returned.

Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=5.9999999999999; x=put(fuzz(var1),16.14);</td>
<td>6</td>
</tr>
<tr>
<td>x=put(fuzz(5.99999999), 16.14);</td>
<td>5.99999999</td>
</tr>
</tbody>
</table>

GAMINV Function

Returns a quantile from the gamma distribution.

Category: Quantile

Returned data type: DOUBLE

Syntax

GAMINV(p, a)

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range 0 ≤ p < 1

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

a

specifies any valid expression that evaluates to a numeric shape parameter.

Range a > 0
The GAMINV function returns the $p$th quantile from the gamma distribution, with shape parameter $a$. The probability that an observation from a gamma distribution is less than or equal to the returned quantile is $p$.

Note: GAMINV is the inverse of the PROBGAM function.

### Example

The following statements illustrate the GAMINV function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1=gaminv(0.5, 9);</td>
<td>8.668951</td>
</tr>
<tr>
<td>q2=gaminv(0.1, 2.1);</td>
<td>0.584193</td>
</tr>
</tbody>
</table>

### See Also

Functions:
- “PROBGAM Function” on page 694
Details
The GAMMA function returns the integral, which is given by the following equation.

\[
\text{GAMMA}(x) = \int_0^\infty t^{x-1} e^{-t} \, dt.
\]

For positive integers, \( \text{GAMMA}(x) \) is \((x - 1)!\). This function is commonly denoted by \( \Gamma(x) \).

Example
The following statement illustrates the GAMMA function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x=\text{gamma(6)}; )</td>
<td>120</td>
</tr>
</tbody>
</table>

GARKHCLPRC Function
Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Financial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
\( \text{GARKHCLPRC}(E, t, S, R_d, R_f, \sigma) \)

Arguments
\( E \)

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify \( E \) and \( S \) in the same units.

Data type DOUBLE

\( t \)

is a nonmissing value that specifies the time to maturity.

Data type DOUBLE

\( S \)

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify \( S \) and \( E \) in the same units.

Data type DOUBLE
\( R_d \)

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period \( t \).

**Requirement**

Specify a value for \( R_d \) for the same time period as the unit of \( t \).

**Data type**

DOUBLE

\( R_f \)

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period \( t \).

**Requirement**

Specify a value for \( R_f \) for the same time period as the unit of \( t \).

**Data type**

DOUBLE

**sigma**

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

**Requirement**

Specify a value for \( \sigma \) for the same time period as the unit of \( t \).

**Data type**

DOUBLE

**Details**

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

\[
\text{CALL} = \text{SN}(d_1)e^{-R_f t} - EN(d_2)e^{-R_d t}
\]

**Arguments**

\( S \)

specifies the spot currency price.

\( N \)

specifies the cumulative normal density function.

\( E \)

specifies the exercise price of the option.

\( t \)

specifies the time to expiration.

\( R_d \)

specifies the risk-free domestic interest rate for period \( t \).

\( R_f \)

specifies the risk-free foreign interest rate for period \( t \).

\[
d_1 = \frac{\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t}{\sigma\sqrt{t}}
\]

\[
d_2 = d_1 - \sigma\sqrt{t}
\]

The following arguments apply to the preceding equation:
\( \sigma \)

specifies the volatility of the underlying asset.

\( \sigma^2 \)

specifies the variance of the rate of return.

For the special case of \( t=0 \), the following equation is true:

\[
\text{CALL} = \max(S - E, 0)
\]

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

Comparisons

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following statements illustrate the GARKHCLPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=garkhclprc(40, .5, 38, .06, .04, .2);</td>
<td>1.44942510595479</td>
</tr>
<tr>
<td>c=garkhclprc(19, .25, 20, .05, .03, .09);</td>
<td>1.1304209447635</td>
</tr>
</tbody>
</table>

See Also

Functions:

• “GARKHPTPRC Function” on page 514

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Financial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

GARKHPTPRC\((E, t, S, R_p, R_r, \text{sigma})\)
Arguments

$E$

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify $E$ and $S$ in the same units.

Data type DOUBLE

$t$

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

$S$

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify $S$ and $E$ in the same units.

Data type DOUBLE

$R_d$

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period $t$.

Requirement Specify a value for $R_d$ for the same time period as the unit of $t$.

Data type DOUBLE

$R_f$

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period $t$.

Requirement Specify a value for $R_f$ for the same time period as the unit of $t$.

Data type DOUBLE

$\sigma$

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Data type DOUBLE

Details

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$PUT = CALL - S (e^{-R_f t}) + E (e^{-R_d t})$$

Arguments

$S$

specifies the spot currency price.

$E$

specifies the exercise price of the option.
specifies the time to expiration, in years.

\( R_d \)
specifies the risk-free domestic interest rate for period \( t \).

\( R_f \)
specifies the risk-free foreign interest rate for period \( t \).

\[
d_1 = \frac{\ln\left(\frac{S}{E}\right) + \left( R_d - R_f + \frac{\sigma^2}{2}\right)t}{\sigma \sqrt{t}}
\]

\[
d_2 = d_1 - \sigma \sqrt{t}
\]

The following arguments apply to the preceding equation:

\( \sigma \)
specifies the volatility of the underlying asset.

\( \sigma^2 \)
specifies the variance of the rate of return.

For the special case of \( t=0 \), the following equation is true:

\[ \text{PUT} = \max\{E - S, 0\} \]

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

**Comparisons**

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

**Example**

The following statements illustrate the GARKHPTPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=garkhptprc(50, .7, 55, .05, .04, .2) );</td>
<td>1.4050880844848</td>
</tr>
<tr>
<td>( b=garkhptprc(32, .3, 33, .05, .03, .3) );</td>
<td>1.56473205137371</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “GARKHCLPRC Function” on page 512
GCD Function

Returns the greatest common divisor for a set of integers.

**Category:** Mathematical  
**Returned data type:** DOUBLE

**Syntax**

\[ \text{GCD}(\text{expression-1, expression-2 [,...expression-n]}) \]

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value.

**Requirement**  
At least two arguments are required.

**Data type**  
DOUBLE

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**Details**

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

**Example**

The following statements illustrate the GCD function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{gcd}(5, 15) )</td>
<td>5</td>
</tr>
<tr>
<td>( x = \text{gcd}(36, 45) )</td>
<td>9</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “LCM Function” on page 601

GEODIST Function

Returns the geodetic distance between two latitude and longitude coordinates.
Category: Distance  
Returned data type: DOUBLE

Syntax

GEODIST(latitude-1, longitude-1, latitude-2, longitude-2 [option(s)])

Arguments

latitude

is a numeric constant, variable, or expression that specifies the coordinate of a given position north or south of the equator. Coordinates that are located north of the equator have positive values; coordinates that are located south of the equator have negative values.

Restriction  If the value is expressed in degrees, it must be between 90 and –90. If the value is expressed in radians, it must be between pi/2 and –pi/2.

Data type  DOUBLE

longitude

is a numeric constant, variable, or expression that specifies the coordinate of a given position east or west of the prime meridian, which runs through Greenwich, England. Coordinates that are located east of the prime meridian have positive values; coordinates that are located west of the prime meridian have negative values.

Restriction  If the value is expressed in degrees, it must be between 180 and –180. If the value is expressed in radians, it must be between pi and –pi.

Data type  DOUBLE

option(s)

specifies a character constant, variable, or expression that contains any of the following characters:

M  specifies distance in miles.
K  specifies distance in kilometers. K is the default value for distance.
D  specifies that input values are expressed in degrees. D is the default for input values.
R  specifies that input values are expressed in radians.

Data type  CHAR

Details

The GEODIST function computes the geodetic distance between any two arbitrary latitude and longitude coordinates. Input values can be expressed in degrees or in radians.
Examples

Example 1: Calculating the Geodetic Distance in Kilometers
The following example shows the geodetic distance in kilometers between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W). The program uses the default K option.

```sas
data _null_; method run();
   distance=geodist(30.68, -88.25, 35.43, -82.55);
   put 'Distance= ' distance ' kilometers';
end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance= 748.652914703181 kilometers
```

Example 2: Calculating the Geodetic Distance in Miles
The following example uses the M option to compute the geodetic distance in miles between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W).

```sas
data _null_; method run();
   distance=geodist(30.68, -88.25, 35.43, -82.55, 'M');
   put 'Distance = ' distance ' miles';
end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance = 465.290810878298 miles
```

Example 3: Calculating the Geodetic Distance with Input Measured in Degrees
The following example uses latitude and longitude values that are expressed in degrees to compute the geodetic distance between two locations. Both the D and the M options are specified in the program.

```sas
data _null_; method run();
dcl double distance lat1 long1 lat2 long2;
lats=35.2;
longs=-78.1;
lats=37.6;
longs=-79.8;
pi = constant('pi');
lats = (pi*lats)/180;
longs = (pi*longs)/180;
Distance = geodist(lat1,long1,lat2,long2,'DM');
put 'Distance= ' Distance ' miles';
```

GEODIST Function
Example 4: Calculating the Geodetic Distance with Input Measured in Radians

The following example uses latitude and longitude values that are expressed in radians to compute the geodetic distance between two locations. The program converts degrees to radians before executing the GEODIST function. Both the R and the M options are specified in this program.

```sas
data _null_; method run();
  dcl double distance pi lat1 long1 lat2 long2;
  lat1=35.2;
  long1=-78.1;
  lat2=37.6;
  long2=-79.8;
  pi = constant('pi');
  lat1 = (pi*lat1)/180;
  long1 = (pi*long1)/180;
  lat2 = (pi*lat2)/180;
  long2 = (pi*long2)/180;
  Distance = geodist(lat1,long1,lat2,long2,'RM');
  put 'Distance= ' Distance ' miles';
end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance = 190.724742819706 miles
```

References


**GEOMEAN Function**

Returns the geometric mean.

<table>
<thead>
<tr>
<th>Category</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
Syntax

GEOMEAN(expression [,... expression])

Arguments

expression

is any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let \( n \) be the number of arguments with non-null or nonmissing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The geometric mean is the \( n^{th} \) root of the product of the values:

\[
\sqrt[n]{x_1 \times x_2 \times \cdots \times x_n}
\]

Equivalently, the geometric mean is shown in this equation.

\[
\exp\left(\frac{\log(x_1) + \log(x_2) + \cdots + \log(x_n)}{n}\right)
\]

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEAN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=geomean(1,2,2,4);</td>
<td>2</td>
</tr>
<tr>
<td>x2=geomean(.,2,4,8);</td>
<td>4</td>
</tr>
</tbody>
</table>
**See Also**

**Functions:**
- “GEOMEANZ Function” on page 522
- “HARMEAN Function” on page 523
- “HARMEANZ Function” on page 525
- “MEAN Function” on page 625

---

**GEOMEANZ Function**

*Returns the geometric mean, using zero fuzzing.*

**Category:** Descriptive Statistics

**Returned data type:** DOUBLE

**Syntax**

```
GEOMEANZ(expression [,...expression])
```

**Arguments**

- `expression` specifies any valid expression that evaluates to a nonnegative numeric value.

  **Data type** DOUBLE

  **See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let \( n \) be the number of arguments with non-null or nonmissing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The geometric mean is the \( n^{th} \) root of the product of the values:

\[
\sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n}
\]

Equivalently, the geometric mean is shown in this equation.

\[
\exp \left( \frac{\log(x_1) + \log(x_2) + \ldots + \log(x_n)}{n} \right)
\]
Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEANZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1 = \text{geomeanz}(1, 2, 2, 4));</td>
<td>2</td>
</tr>
<tr>
<td>(x_2 = \text{geomeanz}(., 2, 4, 8));</td>
<td>4</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “GEOMEAN Function” on page 520
- “HARMEAN Function” on page 523
- “HARMEANZ Function” on page 525
- “MEAN Function” on page 625

HARMEAN Function

Returns the harmonic mean.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

\[
\text{HARMEAN}(expression [, ...expression])
\]

Arguments

\(expression\)

specifies any valid expression that evaluates to a nonnegative numeric value.

<table>
<thead>
<tr>
<th>Data type</th>
<th>DOUBLE</th>
</tr>
</thead>
</table>

See

Chapter 13, “DS2 Expressions,” on page 93
Details
If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let \( n \) be the number of arguments with non-null or nonmissing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The harmonic mean is shown in this equation.

\[
\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}
\]

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons
The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example
The following statements illustrate the HARMEAN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x1 = \text{harmean}(1, 2, 4, 4) );</td>
<td>2</td>
</tr>
<tr>
<td>( x2 = \text{harmean}(., 4, 12, 24) );</td>
<td>8</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “GEOMEAN Function” on page 520
- “GEOMEANZ Function” on page 522
- “HARMEANZ Function” on page 525
- “MEAN Function” on page 625
HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Category:  Descriptive Statistics
Returned data type:  DOUBLE

Syntax

HARMEANZ(expression [,...expression])

Arguments

expression

specifies any valid expression that evaluates to a nonnegative numeric value.

Data type  DOUBLE

See  Chapter 13, “DS2 Expressions,” on page 93

Details

If any argument is negative, then the result is a null or value. A message appears in the log that the negative argument is invalid. If all the arguments are null or values, then the result is a null or value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let $n$ be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEANZ function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEANZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x1=harmeanz(1,2,4,4)$;</td>
<td>2</td>
</tr>
</tbody>
</table>
See Also

Functions:
- “GEOMEAN Function” on page 520
- “GEOMEANZ Function” on page 522
- “HARMEAN Function” on page 523
- “MEAN Function” on page 625

HBOUND Function

Returns the upper bound of an array.

Category: Array

Returned data type: INTEGER

Syntax

HBOUND(array-name[, bound-n])

Arguments

array-name
specifies the name of a temporary or a variable array.

Data type: CHAR

bound-n
is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the upper bound. If no bound-n value is specified, the HBOUND function returns the upper bound of the first dimension of the array.

Bound-n evaluates to an integral value.

Data type: INTEGER

Details

The HBOUND function returns the upper bound of a one-dimensional array, or the upper bound of a specified dimension of a multidimensional array.

HBOUND and LBOUND can be used together to return the values of the upper and lower bounds of an array dimension.
If the HBOUND function is called with a dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

**Comparisons**

- **DIM** returns the number of elements in an array dimension.
- **HBOUND** returns the value of the upper bound of an array dimension.
- **LBOUND** returns the value of the lower bound of an array dimension.
- **NDIMS** returns the number of dimensions in an array.

**Example**

The following example shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```plaintext
data _null_;  
method init();  
declare char(15) a1[4];  
declare double a2[2,3,4] sum;  
a1 := ('red' 'yellow' 'green' 'blue');  
a2 := (24*2.0);  
do i = 1 to dim(a1);  
   put a1[i];  
end;  
numelems = 0;  
do i = 1 to ndims(a2);  
   numelems = numelems + dim(a2, i);  
end;  
sum = 0;  
do i = lbound(a2, 1) to hbound(a2, 1);  
   do j = lbound(a2, 2) to hbound(a2, 2);  
      do k = lbound(a2, 3) to hbound(a2, 3);  
         sum = sum + a2[i,j,k];  
      end;  
   end;  
   end;  
   sum = sum=;  
end;  
enddata;  
run;
```
SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

**See Also**

**Functions:**
- “DIM Function” on page 478
- “LBOUND Function” on page 600
- “NDIMS Function” on page 638

---

**HMS Function**

Returns a SAS time value from hour, minute, and second values.

**Category:** Date and Time  
**Returned data type:** DOUBLE

**Syntax**

\[ \text{HMS}(\text{hour}, \text{minute}, \text{second}) \]

**Arguments**

**hour**
- specifies a numeric expression that represents an integer from 1 through 12.  
  Data type: DOUBLE  
  See: Chapter 13, “DS2 Expressions,” on page 93

**minute**
- specifies a numeric expression that represents an integer from 1 through 59.  
  Data type: DOUBLE  
  See: Chapter 13, “DS2 Expressions,” on page 93

**second**
- specifies a numeric expression that represents an integer from 1 through 59.  
  Data type: DOUBLE  
  See: Chapter 13, “DS2 Expressions,” on page 93
Details

The HMS function returns a numeric value that represents a SAS time value. A SAS time value is a number that represents the number of seconds since midnight of the current day.

Example

The following statements illustrate the HMS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=hms(12,45,10);</td>
<td>45910</td>
</tr>
<tr>
<td>b=put(a, time.);</td>
<td>12:45:10</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

- “DHMS Function” on page 476
- “HOUR Function” on page 532
- “MINUTE Function” on page 628
- “SECOND Function” on page 775

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

**Category:** Date and Time

**Returned data type:** INTEGER

**Syntax**

HOLIDAY('holiday', year)

**Arguments**

'holiday'

is a character constant, variable, or expression that specifies one of the values listed in the following table.

Values for holiday can be in uppercase or lowercase.
<table>
<thead>
<tr>
<th>Holiday Value</th>
<th>Description</th>
<th>Date Celebrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOXING</td>
<td>Boxing Day</td>
<td>December 26</td>
</tr>
<tr>
<td>CANADA</td>
<td>Canadian Independence Day</td>
<td>July 1</td>
</tr>
<tr>
<td>CANADAOBSERVED</td>
<td>Canadian Independence Day observed</td>
<td>July 1, or July 2 if July 1 is a Sunday</td>
</tr>
<tr>
<td>CHRISTMAS</td>
<td>Christmas</td>
<td>December 25</td>
</tr>
<tr>
<td>COLUMBUS</td>
<td>Columbus Day</td>
<td>2nd Monday in October</td>
</tr>
<tr>
<td>EASTER</td>
<td>Easter Sunday</td>
<td>date varies</td>
</tr>
<tr>
<td>FATHERS</td>
<td>Father's Day</td>
<td>3rd Sunday in June</td>
</tr>
<tr>
<td>HALLOWEEN</td>
<td>Halloween</td>
<td>October 31</td>
</tr>
<tr>
<td>LABOR</td>
<td>Labor Day</td>
<td>1st Monday in September</td>
</tr>
<tr>
<td>MLK</td>
<td>Martin Luther King, Jr. ’s birthday</td>
<td>3rd Monday in January beginning in 1986</td>
</tr>
<tr>
<td>MEMORIAL</td>
<td>Memorial Day</td>
<td>last Monday in May (since 1971)</td>
</tr>
<tr>
<td>MOTHERS</td>
<td>Mother's Day</td>
<td>2nd Sunday in May</td>
</tr>
<tr>
<td>NEWYEAR</td>
<td>New Year's Day</td>
<td>January 1</td>
</tr>
<tr>
<td>THANKSGIVING</td>
<td>U.S. Thanksgiving Day</td>
<td>4th Thursday in November</td>
</tr>
<tr>
<td>THANKSGIVINGCANADA</td>
<td>Canadian Thanksgiving Day</td>
<td>2nd Monday in October</td>
</tr>
<tr>
<td>USINDEPENDENCE</td>
<td>U.S. Independence Day</td>
<td>July 4</td>
</tr>
<tr>
<td>USPRESIDENTS</td>
<td>Abraham Lincoln's and George Washington's birthdays observed</td>
<td>3rd Monday in February (since 1971)</td>
</tr>
<tr>
<td>VALENTINES</td>
<td>Valentine's Day</td>
<td>February 14</td>
</tr>
<tr>
<td>VETERANS</td>
<td>Veterans Day</td>
<td>November 11</td>
</tr>
<tr>
<td>VETERANSUSG</td>
<td>Veterans Day - U.S. government-observed</td>
<td>U.S. government-observed date for Monday–Friday schedule</td>
</tr>
</tbody>
</table>
The HOLIDAY function computes the date on which a specific holiday occurs in a specified year. Only certain common U.S. and Canadian holidays are defined for use with this function. (See “Holiday Values and Their Descriptions” in SAS Functions and CALL Routines: Reference for a list of valid holidays.)

The definition of many holidays has changed over the years. In the U.S., Executive Order 11582, issued on February 11, 1971, fixed the observance of many U.S. federal holidays.

The current holiday definition is extended indefinitely into the past and future, although many holidays have a fixed date at which they were established. Some holidays have not had a consistent definition in the past.

The HOLIDAY function returns a SAS date value. To convert the SAS date value to a calendar date, use any valid SAS date format, such as the DATE9. format.

Comparisons

In some cases, the HOLIDAY function and the NWKDOM function return the same result. For example, the statement `holiday('thanksgiving', 2012);` returns the same value as `nwkdom(4, 5, 11, 2012);`.

In other cases, the HOLIDAY function and the MDY function return the same result. For example, the statement `holiday('christmas', 2012);` returns the same value as `mdy(12, 25, 2012);`.

Example

The following statements illustrate the HOLIDAY function:

```
year
```
Statements | Results
---|---
dcl double thanks having format date9.;
thanks = holiday('thanksgiving', 2013);
put thanks; | 28NOV2013

dcl double boxing having format date9.;
boxing = holiday('boxing', 2013);
put boxing; | 26DEC2013

dcl double easter having format date9.;
easter = holiday('easter', 2013);
put easter; | 31MAR2013

dcl double canada having format date9.;
canada = holiday('canada', 2013);
put canada; | 01JUL2013

dcl double fathers having format date9.;
fathers = holiday('fathers', 2013);
put fathers; | 16JUN2013

dcl double valentines having format date9.;
valentines = holiday('valentines', 2013);
put valentines; | 14FEB2013

dcl double victoria having format date9.;
victoria = holiday('victoria', 2013);
put victoria; | 20MAY2013

See Also

Functions:
- “MDY Function” on page 624
- “NWKDOM Function” on page 673

**HOUR Function**

Returns the hour from a SAS time or datetime value.

**Category:** Date and Time

**Returned data type:** DOUBLE

**Syntax**

```
HOUR(time | datetime)
```
**Arguments**

*time*

specifies any valid expression that represents a SAS time value.

Data type **DOUBLE**

See Chapter 13, “DS2 Expressions,” on page 93

**datetime**

specifies any valid expression that represents a SAS datetime value.

Data type **DOUBLE**

See Chapter 13, “DS2 Expressions,” on page 93

**Details**

The HOUR function returns a numeric value that represents the hour from a SAS time or datetime value. Numeric values can range from 0 through 23. HOUR always returns a positive number.

**Example**

The following statement illustrates the HOUR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=hour(time());</td>
<td>10</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Functions:**

- “MINUTE Function” on page 628
- “SECOND Function” on page 775

---

**INDEX Function**

Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

\`INDEX(target-expression, search-expression)\`
Arguments

target-expression
specifies any valid expression that evaluates to a character string.

Data type NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

search-expression
specifies any valid expression that evaluates to a character string to search for in target-expression.

Data type NCHAR

Tip Enclose a literal string of characters in single quotation marks.

See Chapter 13, “DS2 Expressions,” on page 93

Details
The INDEX function searches target-expression, from left to right, for the first occurrence of the string specified in search-expression, and returns the position in target-expression of the string's first character. If the string is not found in target-expression, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

Comparisons
The VERIFY function returns the position of the first character in target-expression that does not contain search-expression where the INDEX function returns the position of the first occurrence of search-expression that is present in target-expression.

Example
The following statements illustrate the INDEX statement:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='ABC.DEF (X=Y)' ;</td>
<td>10</td>
</tr>
<tr>
<td>b='X=Y' ;</td>
<td></td>
</tr>
<tr>
<td>c=index(a,b) ;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

- “INDEXC Function” on page 535
- “INDEXW Function” on page 536
- “VERIFY Function” on page 828
INDEXC Function

Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.

**Category:** Character  
**Returned data type:** DOUBLE

**Syntax**

```
INDEXC(target-expression, search-expression[,...search-expression])
```

**Arguments**

- **target-expression**
  - Specifies any valid expression that evaluates to a character string that is searched.  
  - **Data type:** NCHAR
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **search-expression**
  - Specifies the characters to search for in **target-expression**.  
  - **Data type:** NCHAR
  - **Tip:** Enclose a literal string of characters in single quotation marks.
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The INDEXC function searches **target-expression**, from left to right, for the first occurrence of any character present in the search expressions and returns the position in **target-expression** of that character. If none of the characters in the search expressions are found in **target-expression**, INDEXC returns a value of 0.

**Comparisons**

The INDEXC function searches for the first occurrence of any individual character that is present within the search expression, whereas the INDEX function searches for the first occurrence of the search expression as a pattern.

**Example**

The following statements illustrate the INDEXC function:
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 'ABC.DEF (X2=Y1)'</td>
<td>4</td>
</tr>
<tr>
<td>b = '('  ').'</td>
<td>8</td>
</tr>
<tr>
<td>c = '..'</td>
<td>4</td>
</tr>
<tr>
<td>b = indexc(a, '0123',  ';,.')</td>
<td></td>
</tr>
<tr>
<td>b = indexc(a, b)</td>
<td></td>
</tr>
<tr>
<td>b = indexc(a, b, c)</td>
<td></td>
</tr>
<tr>
<td>c = 'have a good day';</td>
<td>2</td>
</tr>
<tr>
<td>d = indexc(c, 'pleasant', 'very')</td>
<td></td>
</tr>
</tbody>
</table>

### See Also

#### Functions:
- “INDEX Function” on page 533
- “INDEXW Function” on page 536

### INDEXW Function

Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.

**Category:** Character  
**Returned data type:** DOUBLE

### Syntax

\[ \text{INDEXW} (\text{target-expression}, \text{search-expression} [\,, \text{delimiter}]) \]

### Arguments

- **target-expression**
  - specifies any valid expression that evaluates to a character string that is searched.
  - **Data type:** NCHAR
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **search-expression**
  - specifies any valid expression that evaluates to a character string and that is searched for in target-expression. SAS removes the leading and trailing delimiters from search-expression.
  - **Data type:** NCHAR
  - **See:** Chapter 13, “DS2 Expressions,” on page 93
**delimiter**

specifies a character expression that you want INDEXW to use as a word separator in the character strings. The default delimiter is the blank character.

**Data type**

NCHAR

**Tip**

If the blank character is a delimiter, order it so that it is not the last character in *delimiter*. Trailing blanks are ignored because *delimiter* is trimmed of trailing blanks.

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The INDEXW function searches *target-expression*, from left to right, for the first occurrence of *search-expression* and returns the position in *target-expression* of the substring's first character. If the substring is not found in *target-expression*, then INDEXW returns a value of 0. If there are multiple occurrences of the string, then INDEXW returns only the position of the first occurrence.

The substring pattern must begin and end on a word boundary. For INDEXW, word boundaries are delimiters, the beginning of *target-expression*, and the end of *target-expression*.

**T I P**

INDEXW has the following behavior when *search-expression* contains blank spaces or has a length of 0:

- If both *target-expression* and *search-expression* contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If *search-expression* contains only blank spaces or has a length of 0, and *target-expression* contains character or numeric data, then INDEXW returns a value of 0.

**Comparisons**

The INDEXW function searches for strings that are words, whereas the INDEX function searches for patterns as separate words or as parts of other words. INDEXC searches for any characters that are present in the excerpts.

**Example**

The following statements illustrate the INDEXW function.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='The power to know.'; word='power'; c=indexw(a,word);</td>
<td>5</td>
</tr>
<tr>
<td>a='The power to know.'; b=indexw(a,'know');</td>
<td>0</td>
</tr>
<tr>
<td>a='The power to know.','.'); b=indexw(a,'know','.');</td>
<td>14</td>
</tr>
</tbody>
</table>
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| `a='abc,def@xyz';`  
`b=indexw(a,',' '@');` | 0 |
| `a="abc,def@xyz";`  
`b=indexw(a,'def', '@');` | 5 |
| `x="abc,def@xyz";`  
`xyz=indexw(x,' xyz', '@');` | 10 |

### See Also

**Functions:**
- “INDEX Function” on page 533
- “INDEXC Function” on page 535

### INPUTC Function

Enables you to specify a character informat at run time.

**Category:** Special

**Returned data type:** CHAR

### Syntax

\[ \text{INPUTC}(\text{source}, \text{informat}[, \text{w}]) \]

### Arguments

**source**

specifies a character constant, variable, or expression to which you want to apply the informat.

Data type: CHAR

**informat**

is a character constant, variable, or expression that contains the character informat that you want to apply to source.

Data type: CHAR

**w**

specifies any valid expression that evaluates to a numeric width to apply to the informat.

Interaction: If you specify a width here, it overrides any width specification in the informat.
Data type   INTEGER

See   Chapter 13, “DS2 Expressions,” on page 93

Details
If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons
The INPUTN function enables you to specify a numeric informat at run time.

Example
This example shows how to specify character informat.

data _null_
  dcl char(10) type type2;
  method init();
  type=inputc('positive', '$upcase15.');
  type2=inputc('positive', '$upcase15.', 3);
  put type=
  put type2=
  end;
enddata;
run;

The following line is written to the SAS log.

  type=POSITIVE
  type2=POS

See Also

Functions:
•  “INPUTN Function” on page 539
•  “PUT Function” on page 725
Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR

informat
is a character constant, variable, or expression that contains the numeric informat that you want to apply to source.

Data type CHAR

w
is a numeric constant, variable, or expression that specifies a width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type INTEGER

d
is a numeric constant, variable, or expression that specifies the number of decimal places to use.

Interaction If you specify a number here, it overrides any decimal-place specification in the informat.

Data type INTEGER

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the PUT function is faster because you specify the informat at compile time.

Example

This example shows how to specify numeric informats.

```sas
data _null_;  
  method init();
  declare double salary;
  salary = inputn('20,000.00', 'comma10.2');
  put salary=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
20000
```

See Also

Functions:
INT Function

Returns the integer value, fuzzed to avoid unexpected floating-point results.

**Category:** Truncation

**Returned data type:** DOUBLE

### Syntax

```
INT(expression)
```

### Arguments

- **expression** specifies any expression that evaluates to a numeric value.

**Data type**: DOUBLE

### Details

The INT function returns the integer portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of an integer, the function results in that integer. If the value of `expression` is positive, the INT function has the same result as the FLOOR function. If the value of `expression` is negative, the INT function has the same result as the CEIL function.

### Comparisons

Unlike the INTZ function, the INT function fuzzes the result. If the argument is within 1E-12 of an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

### Example

The following statements illustrate the INT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var1=2.1;</code></td>
<td></td>
</tr>
<tr>
<td><code>a=int(var1);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>a=int(-2.4);</code></td>
<td>-2</td>
</tr>
<tr>
<td><code>a=int(1+1.e-11);</code></td>
<td>1</td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
a=int(-1.6); | -1

See Also

Functions:

- “CEIL Function” on page 419
- “FLOOR Function” on page 505
- “INTZ Function” on page 581
- “MOD Function” on page 631
- “MODZ Function” on page 633
- “ROUND Function” on page 756

INTCINDEX Function

Returns the cycle index when a date, time, or timestamp interval and value are specified.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Syntax

INTCINDEX(interval[multiple][.shift-index], date-time-value)

Arguments

interval[multiple][.shift-index]

specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

<table>
<thead>
<tr>
<th>Data type:</th>
<th>CHAR</th>
</tr>
</thead>
</table>

Note

The possible values of interval are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.

Tip

Interval can appear in uppercase or lowercase.

Example

YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.
**Data type**  INTEGER

**See**  “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

**Example**  YEAR2 specifies a two-year, or biennial, interval type.

**shift-index**  specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

**Restrictions**  The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

**Data type**  INTEGER

**See**  “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

**Example**  YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

**date-time-value**  specifies a date, time, or timestamp value that represents a time period of a specified interval.

**Data type**  DOUBLE

**Details**

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a DATE, TIME, or TIMESTAMP value. For example, if the interval is MONTH, each observation in the data corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12. WEEK is the seasonal cycle for an interval that is equal to DAY. This example returns a value of 36 because September 1, 2013, is the sixth day of the 35th week of the year.

```sas
sasdate1 = to_double(date'2013-09-01');
cycle_index1 = intcindex('day', sasdate1);
```
For more information about working with date and time intervals, see “Date and Time Intervals” in SAS Functions and CALL Routines: Reference.

The INTCINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For a list of these intervals, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Comparisons

The INTCINDEX function returns the cycle index, whereas the INTINDEX function returns the seasonal index.

In this example, the INTCINDEX function returns the week of the year.

```sas
sasdate1=to_double(date'04apr2013');
cycle_index1 = intcindex('day', sasdate1);
```

In this example, the INTINDEX function returns the day of the week.

```sas
sasdate1=to_double(date'04apr2013');
index1 = intindex('day', '04APR2013'd);
```

In this example, the INTCINDEX function returns the hour of the day.

```sas
sasts=to_double(timestamp '2012-09-01 00:00:00');
a= intcindex('minute', sasts);
```

In this example, the INTINDEX function returns the minute of the hour.

```sas
sasts=to_double(timestamp '2012-09-01 00:00:00');
a= intindex('minute', sasts);
```

In the example `intseas(intcycle('interval'))`, the INTSEAS function returns the maximum number that could be returned by `intcindex('interval', date)`.

Example

The following statements illustrate the INTCINDEX function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasdate1=to_double(date'2013-09-01');</td>
<td>36</td>
</tr>
<tr>
<td>cycle_index1 = intcindex('day', sasdate1);</td>
<td></td>
</tr>
<tr>
<td>put cycle_index1;</td>
<td></td>
</tr>
<tr>
<td>sascydate=to_double(timestamp '2013-05-23 05:03:01');</td>
<td>1</td>
</tr>
<tr>
<td>cycle_index1 = intcindex('dtqtr', sascydate);</td>
<td></td>
</tr>
<tr>
<td>put cycle_index1;</td>
<td></td>
</tr>
<tr>
<td>sascydate=to_double(date'2013-12-13');</td>
<td>1</td>
</tr>
<tr>
<td>cycle_index1 = intcindex('tenday', sascydate);</td>
<td></td>
</tr>
<tr>
<td>put cycle_index1;</td>
<td></td>
</tr>
<tr>
<td>sascydate=to_double(time '23:13:02');</td>
<td>24</td>
</tr>
<tr>
<td>cycle_index1 = intcindex('minute', sascydate);</td>
<td></td>
</tr>
<tr>
<td>put cycle_index1;</td>
<td></td>
</tr>
</tbody>
</table>
Statements

```
sascydate=to_double(timestamp '2013-05-05 10:54:03');
var1='semimonth';
cycle_index1 = intcindex(var1, sascydate);
put cycle_index1;
```

Results

1

See Also

Functions:

- “INTCYCLE Function” on page 552
- “INTINDEX Function” on page 560
- “INTSEAS Function” on page 572

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.

**Syntax**

```
INTCK(interval[,multiple][.shift-index], start-date, end-date[, 'method'])
INTCK(start-date, end-date[, 'method'])
```

**Arguments**

`interval[,multiple][.shift-index]`

specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

`interval`

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR

**Note**

The possible values of `interval` are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.

**Tip**

`Interval` can appear in uppercase or lowercase.

**Example**

YEAR specifies year-based intervals.
**multiple**

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type INTEGER

See “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

**shift-index**

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type INTEGER

See “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

**start-date**

specifies an expression that represents the starting SAS date, time, or timestamp value.

Data type DOUBLE

**end-date**

specifies an expression that represents the ending SAS date, time, or timestamp value.

Data type DOUBLE

*method*

specifies that intervals are counted using either a discrete or a continuous method.

You must enclose *method* in quotation marks. *Method* can be one of these values:
CONTINUOUS specifies that continuous time is measured. The interval is shifted based on the starting date.

For example, the distance in months between January 15, 2013, and February 15, 2013, is one month.

Alias C or CONT

DISCRETE specifies that discrete time is measured. The discrete method counts interval boundaries (for example, end of month).

The default discrete method is useful to sort time series observations into bins for processing. For example, daily data can be accumulated to monthly data for processing as a monthly series.

For the DISCRETE method, the distance in months between January 31, 2013, and February 1, 2013, is one month.

Alias D or DISC

Default DISCRETE

Data type CHAR

Details

**Calendar Interval Calculations**

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2013 and January 15, 2013 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2013 and ends on January 31, 2013. You can use the date for the beginning of the interval (January 1, 2013) or the date for the end of the interval (January 31, 2013) to identify the interval. These dates represent all of the dates within the monthly interval.

In the following example, the start-date (Jan. 14, 2013) is equivalent to the first quarter of 2013.

```sas
sasdate1=to_double(date'2013-01-14');
sasdate2 = to_double(date'2013-09-02');
qtr=intck('qtr', sasdate1, sasdate2);
```

The end-date (September 2, 2013) is equivalent to the third quarter of 2013. The interval count, that is, the number of times the beginning of an interval is reached in moving from the start-date to the end-date is 2.

The INTCK function using the default discrete method counts the number of times the beginning of an interval is reached in moving from the first date to the second. It does not count the number of complete intervals between two dates:

- The following example returns 0, because the two dates are within the same month.
  ```sas
  sasdate1=to_double(date'2013-01-01');
sasdate2 = to_double(date'2013-01-31');
month=intck('month', sasdate1, sasdate2);
put month;
  ```

- The following example returns 1, because the two dates lie in different months that are one month apart.
The following example returns –1 because the first date is in a later discrete interval than the second date. (INTCK returns a negative value whenever the first date is later than the second date and the two dates are not in the same discrete interval.)

```
sasdate1 = to_double(date'2013-02-01');
sasdate2 = to_double(date'2013-01-31');
month = intck('month', sasdate1, sasdate2);
put month;
```

Using the discrete method, WEEK intervals are determined by the number of Sundays, the default first day of the week, that occur between the start-date and the end-date, and not by how many seven-day periods fall between those dates. To count the number of seven-day periods between start-date and end-date, use the continuous method.

Both the multiple and the shift-index arguments are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years.

For more information about working with date and time intervals, see “Date and Time Intervals” in *SAS Functions and CALL Routines: Reference*.

### Intervals by Category

**Table 23.2** Intervals Used with Date and Time Functions

<table>
<thead>
<tr>
<th>Category</th>
<th>Interval</th>
<th>Definition</th>
<th>Default Starting Point</th>
<th>Shift Period</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>DAY</td>
<td>Daily intervals</td>
<td>Each day</td>
<td>Days</td>
<td>DAY3</td>
<td>Three-day intervals starting on Sunday</td>
</tr>
<tr>
<td>WEEK</td>
<td></td>
<td>Weekly intervals of seven days</td>
<td>Each Sunday</td>
<td>Days (1=Sunday … 7=Saturday)</td>
<td>WEEK.7</td>
<td>Weekly with Saturday as the first day of the week</td>
</tr>
<tr>
<td>WEEKDAY &lt;daysW&gt;</td>
<td>Daily intervals with Friday-Saturday-Sunday</td>
<td>Each day</td>
<td>Days</td>
<td>WEEKDAY1W</td>
<td>Six-day week with Sunday as a weekend day</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Interval</td>
<td>Definition</td>
<td>Default Starting Point</td>
<td>Shift Period</td>
<td>Example</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------</td>
<td>--------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>counted as the same day</td>
<td>(five-day work week with a Saturday-Sunday weekend). days identifies the weekend days by number (1=Sunday ... 7=Saturday). By default, days=17.</td>
<td></td>
<td></td>
<td>WEEKDAY35W</td>
<td>Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)</td>
</tr>
<tr>
<td>TENDAY</td>
<td>Ten-day intervals (a U.S.</td>
<td>First, eleventh, and twenty-first of each month</td>
<td>Ten-day periods</td>
<td></td>
<td>TENDAY4.2</td>
<td>Four ten-day periods starting at the second TENDAY period</td>
</tr>
<tr>
<td></td>
<td>automobile industry convention)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEMIMONTH</td>
<td>Half-month intervals</td>
<td>First and sixteenth of each month</td>
<td>Semi-monthly periods</td>
<td></td>
<td>SEMIMONTH2.2</td>
<td>Intervals from the sixteenth of one month through the fifteenth of the next month</td>
</tr>
<tr>
<td>MONTH</td>
<td>Monthly intervals</td>
<td>First of each month</td>
<td>Months</td>
<td></td>
<td>MONTH2.2</td>
<td>February-March, April-May, June-July, August-September, October-November, and December-January of the following year</td>
</tr>
<tr>
<td>QTR</td>
<td>Quarterly (three-month)</td>
<td>January 1</td>
<td>Months</td>
<td>QTR3.2</td>
<td>Three-month intervals starting on April 1, July 1, October 1, and January 1</td>
<td></td>
</tr>
</tbody>
</table>
## Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in *SAS Language Reference: Concepts*.

## Example

The following statements illustrate the INTCK function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasdate1 = to_double(date'2013-01-10'); sasdate2 = to_double(date'2013-07-01'); qtr=intck('qtr', sasdate1, sasdate2); put qtr;</td>
<td>2</td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2012-12-31'); sasdate2 = to_double(date'2013-01-01'); year=intck('year', sasdate1, sasdate2); put year;</td>
<td>1</td>
</tr>
<tr>
<td>Statements</td>
<td>Results</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>`sasdate1 = to_double(date'2013-01-01');</td>
<td>0</td>
</tr>
<tr>
<td>`sasdate2 = to_double(date'2013-12-31');</td>
<td></td>
</tr>
<tr>
<td>`year=intck('year', sasdate1, sasdate2);</td>
<td></td>
</tr>
<tr>
<td>`put year;</td>
<td></td>
</tr>
<tr>
<td>`sasdate1 = to_double(date'2010-01-01');</td>
<td>6</td>
</tr>
<tr>
<td>`sasdate2 = to_double(date'2013-01-01');</td>
<td></td>
</tr>
<tr>
<td>`semiyear=intck('semiyear', sasdate1, sasdate2);</td>
<td></td>
</tr>
<tr>
<td>`put semiyear;</td>
<td></td>
</tr>
<tr>
<td>`sasdate1 = to_double(date'2013-01-07');</td>
<td>6</td>
</tr>
<tr>
<td>`sasdate2 = to_double(date'2013-04-01');</td>
<td></td>
</tr>
<tr>
<td>`weekvar=intck('week2.2', sasdate1, sasdate2);</td>
<td></td>
</tr>
<tr>
<td>`put weekvar;</td>
<td></td>
</tr>
<tr>
<td>`sasdate1 = to_double(date'2013-01-01');</td>
<td>27</td>
</tr>
<tr>
<td>`sasdate2 = to_double(date'2013-02-01');</td>
<td></td>
</tr>
<tr>
<td>`wdvar=intck('weekday7w', sasdate1, sasdate2);</td>
<td></td>
</tr>
<tr>
<td>`put wdvar;</td>
<td></td>
</tr>
<tr>
<td>`y='year';</td>
<td>10</td>
</tr>
<tr>
<td>`sasdate1 = to_double(date'2003-09-01');</td>
<td></td>
</tr>
<tr>
<td>`sasdate2 = to_double(date'2013-09-01');</td>
<td></td>
</tr>
<tr>
<td>`newyears=intck(y, sasdate1, sasdate2);</td>
<td></td>
</tr>
<tr>
<td>`put newyears;</td>
<td></td>
</tr>
</tbody>
</table>

In the second example, INTCK returns a value of 1 even though only one day has elapsed. This result is returned because the interval from December 31, 2012, to January 1, 2013, contains the starting point for the YEAR interval. However, in the third example, a value of 0 is returned even though 364 days have elapsed. This result is because the period between January 1, 2013, and December 31, 2013, does not contain the starting point for the interval.

In the fourth example, SAS returns a value of 6 because January 1, 2010, through January 1, 2013, contains six semiyearly intervals. (Note that if the ending date were December 31, 2012, SAS would count five intervals.) In the fifth example, SAS returns a value of 6 because there are six two-week intervals beginning on a first Monday during the period of January 7, 2013, through April 1, 2013. In the sixth example, SAS returns the value 27. That indicates that beginning with January 1, 2013, and counting only Saturdays as weekend days through February 1, 2013, the period contains 27 weekdays.

In the seventh example, the use of variables for the arguments is illustrated.

**See Also**

**Functions:**
- “INTDT Function” on page 555
- “INTNX Function” on page 564
- “INTTS Function” on page 580

**Other References:**


INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or
datetime interval is specified.

**Category:** Date and Time

**Returned data type:** VARCHAR, NVARCHAR

**Syntax**

`INTCYCLE(interval[multiple][.shift-index][, seasonality])`

**Arguments**

- `interval[multiple][.shift-index]`
  
  specifies a basic or complex interval. Multipliers and shift indexes can be used with
  the basic interval names to construct more complex interval specifications. The three
  parts of the interval name are as follows:

  - **interval**
    
    specifies a character constant, a variable, or an expression that contains an
    interval name such as WEEK, MONTH, or QTR.
    
    **Data type** CHAR

  - **Note**
    
    The possible values of `interval` are listed in “Intervals Used with
    Date and Time Functions” in *SAS Language Reference: Concepts*.

  - **Tip**
    
    `Interval` can appear in uppercase or lowercase.

  - **Example**
    
    YEAR specifies year-based intervals.

  - **multiple**
    
    specifies an optional multiplier that sets the interval equal to a multiple of the
    period of the basic interval type.
    
    **Data type** INTEGER

  - **See**
    
    “Incrementing Dates and Times By Using Multipliers and By
    Shifting Intervals” in *SAS Functions and CALL Routines: Reference*
    for more information.

  - **Example**
    
    YEAR2 specifies a two-year, or biennial, interval type.

- `shift-index`
  
  specifies an optional shift index that shifts the interval to start at a specified
  subperiod starting point.

- **Restrictions**
  
  The shift index cannot be greater than the number of subperiods in
  the whole interval. For example, you could use YEAR2.24, but
YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type: INTEGER

See: “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

Example: YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

 seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default: 52

Data type: INTEGER, CHAR

Example: The seasonality argument in the following example

INTCYCLE('MONTH', 3);

causes the function call to return the value QTR. The function call

INTCYCLE('MONTH');

does not have a seasonality argument and returns the value YEAR.

Details

The Basics

The INTCYCLE function returns the interval of the seasonal cycle, depending on a date, time, or datetime interval. For example, INTCYCLE('MONTH') returns the value YEAR because the months from January through December constitute a yearly cycle.

INTCYCLE('DAY'); returns the value WEEK because the days from Sunday through Saturday constitute a weekly cycle.

For information about multipliers and shift indexes, see “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference. For information about how intervals are calculated, see “Commonly Used Time Intervals” in SAS Functions and CALL Routines: Reference.

For more information about working with date and time intervals, see “Date and Time Intervals” in SAS Functions and CALL Routines: Reference.
The INTCYCLE function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Functions and CALL Routines: Reference.

**Seasonality**
Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTCYCLE function uses the concept of seasonality and returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the SAS/ETS User's Guide.

**Example**
The following statements illustrate the INTCYCLE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle_year=intcycle('year');</td>
<td>YEAR</td>
</tr>
<tr>
<td>put cycle_year;</td>
<td></td>
</tr>
<tr>
<td>cycle_quarter=intcycle('qtr');</td>
<td>YEAR</td>
</tr>
<tr>
<td>put cycle_quarter;</td>
<td></td>
</tr>
<tr>
<td>cycle_3=intcycle('month', 3);</td>
<td>QTR</td>
</tr>
<tr>
<td>put cycle_3;</td>
<td></td>
</tr>
<tr>
<td>cycle_month=intcycle('month');</td>
<td>YEAR</td>
</tr>
<tr>
<td>put cycle_month;</td>
<td></td>
</tr>
<tr>
<td>cycle_weekday=intcycle('weekday');</td>
<td>WEEK</td>
</tr>
<tr>
<td>put cycle_weekday;</td>
<td></td>
</tr>
<tr>
<td>cycle_weekday2=intcycle('weekday', 5);</td>
<td>WEEK</td>
</tr>
<tr>
<td>put cycle_weekday2;</td>
<td></td>
</tr>
<tr>
<td>cycle_day=intcycle('day');</td>
<td>WEEK</td>
</tr>
<tr>
<td>put cycle_day;</td>
<td></td>
</tr>
<tr>
<td>cycle_day2=intcycle('day', 10);</td>
<td>TENDAY</td>
</tr>
<tr>
<td>put cycle_day2;</td>
<td></td>
</tr>
<tr>
<td>var1='second';</td>
<td></td>
</tr>
<tr>
<td>cycle_second=intcycle(var1);</td>
<td>DTMINUTE</td>
</tr>
<tr>
<td>put cycle_second;</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

Functions:
INTDT Function

Specifies the number of days to add to a DATE value.

Category: Date and Time
Returned data type: DATE

Syntax

INTDT(expression, increment)

Arguments

expression
specifies any valid expression that represents a DATE value.
Data type: DATE

See Chapter 13, “DS2 Expressions,” on page 93

increment
specifies a negative, positive, or zero integer that represents the number of days to add to the date.
Data type: INTEGER

Details

The INTDT function increments a DATE value by the number of days that you specify.

Comparisons

The INTNX function increments a SAS date, time, or datetime value that is encoded as a DOUBLE value.

Example

The following statements illustrate the INTDT function:
### INTFIT Function

Returns a time interval that is aligned between two dates.

**Category:** Date and Time  
**Returned data type:** CHAR

**Syntax**

```
INTFIT(expression-1, expression-2, 'type')
```

**Arguments**

- **expression**
  - specifies any valid expression that represents a SAS date or datetime value.  
  - **Data type:** DOUBLE  
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **'type'**
  - specifies whether the arguments are SAS date values, datetime values, or a row.  
  - The following values for type are valid:
    - **d** specifies that expression-1 and expression-2 are date values.  
    - **dt** specifies that expression-1 and expression-2 are datetime values.  
    - **obs** specifies that expression-1 and expression-2 are rows.  
  - **Data type:** CHAR
Details

The INTFIT function returns the most likely time interval based on two dates, datetime values, or rows that have been aligned within an interval. INTFIT assumes that the alignment value is SAME, which specifies that the date is aligned to the same calendar date with the corresponding interval increment. For more information about the alignment argument, see “INTNX Function” in SAS Functions and CALL Routines: Reference.

If the arguments that are used with INTFIT are rows, you can determine the cycle of an occurrence by using row numbers. In the following example, the first two arguments of INTFIT are row numbers, and the type argument is obs. If Jason used the gym the first time and the 25th time that a researcher recorded data, you could determine the interval by using the following statement: interval=intfit(1, 25, 'obs');. In this case, the value of interval is OBS24.2.

For information about time series, see the SAS/ETS 9.3 User’s Guide.

The INTFIT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Example

The following example shows the interval that is aligned between two dates. The type argument in this example identifies the input as date values.

```sas
data test;
  dcl char(10) c;
  dcl double sasdate1 sasdate2;
  method run();
    sasdate1=to_double(date'2013-08-01');
    sasdate2=to_double(date'2013-09-01');
    c=intfit(sasdate1, sasdate2, 'd');
    put c;
  end;
enddata;
run;
```

The following line is written to the SAS log.

MONTH

See Also

Functions:

- “INTCK Function” on page 545
- “INTNX Function” on page 564

INTGET Function

Returns a time interval based on three date or datetime values.

Category: Date and Time
Returned data type: NCHAR, NVARCHAR

Syntax

INTGET(\textit{date-1}, \textit{date-2}, \textit{date-3})

Argument

\textit{date}

specifies any valid expression that evaluates to a SAS date or datetime value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

\textbf{INTGET Function Intervals}

The INTGET function returns a time interval based on three date or datetime values. The function first determines all possible intervals between the first two dates, and then determines all possible intervals between the second and third dates. If the intervals are the same, INTGET returns that interval. If the intervals for the first and second dates differ, and the intervals for the second and third dates differ, INTGET compares the intervals. If one interval is a multiple of the other, then INTGET returns the smaller of the two intervals. Otherwise, INTGET returns a missing value. INTGET works best with dates generated by the INTNX function whose alignment value is BEGIN.

In the following example, INTGET returns the interval DAY2:

\begin{verbatim}
interval=intget('01mar00'd, '03mar00'd, '09mar00'd);
\end{verbatim}

The interval between the first and second dates is DAY2, because the number of days between March 1, 2000, and March 3, 2000, is two. The interval between the second and third dates is DAY6, because the number of days between March 3, 2000, and March 9, 2000, is six. DAY6 is a multiple of DAY2. INTGET returns the smaller of the two intervals.

In the following example, INTGET returns the interval MONTH4:

\begin{verbatim}
interval=intget('01jan00'd, '01may00'd, '01may01'd);
\end{verbatim}

The interval between the first two dates is MONTH4, because the number of months between January 1, 2000, and May 1, 2000, is four. The interval between the second and third dates is YEAR. INTGET determines that YEAR is a multiple of MONTH4 (there are three MONTH4 intervals in YEAR), and returns the smaller of the two intervals.

In the following example, INTGET returns a missing value:

\begin{verbatim}
interval=intget('01Jan2006'd, '01Apr2006'd, '01Dec2006'd);
\end{verbatim}

The interval between the first two dates is MONTH3, and the interval between the second and third dates is MONTH8. INTGET determines that MONTH8 is not a multiple of MONTH3, and returns a missing value.

The intervals that are returned are valid SAS intervals, including multiples of the intervals and shift intervals. Valid SAS intervals are listed in “Intervals Used with Date and Time Functions” in \textit{SAS Language Reference: Concepts}.
Note: If INTGET cannot determine a matching interval, then the function returns a missing value. No message is written to the SAS log.

Retail Calendar Intervals
The INTGET function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Example
The following statements illustrate the INTGET function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasdate1=to_double(date'2013-01-01'); sasdate2=to_double(date'2014-01-01'); sasdate3=to_double(date'2014-05-01'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>MONTH4</td>
</tr>
<tr>
<td>sasdate1=to_double(date'2012-02-29'); sasdate2=to_double(date'2014-02-28'); sasdate3=to_double(date'2016-02-29'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>YEAR2.2</td>
</tr>
<tr>
<td>sasdate1=to_double(date'2013-02-01'); sasdate2=to_double(date'2013-02-16'); sasdate3=to_double(date'2013-03-01'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>SEMIMONTH</td>
</tr>
<tr>
<td>sasdate1=to_double(date'2013-01-02'); sasdate2=to_double(date'2014-02-02'); sasdate3=to_double(date'2015-03-02'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>MONTH13.13</td>
</tr>
<tr>
<td>sasdate1=to_double(date'2013-02-10'); sasdate2=to_double(date'2013-02-19'); sasdate3=to_double(date'2013-02-28'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>DAY9.5</td>
</tr>
<tr>
<td>sasdate1=to_double(timestamp'2014-04-01 01:03:00.0000'); sasdate2=to_double(timestamp'2014-04-01 01:04:00.0000'); sasdate3=to_double(timestamp'2014-04-01 01:05:00.0000'); c=intget(sasdate1, sasdate2, sasdate3); put c;</td>
<td>MINUTE</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “INTFIT Function” on page 556
INTINDEX Function

Returns the seasonal index when a date, time, or timestamp interval and value are specified.

**Category:** Date and Time  
**Returned data type:** DOUBLE

**Syntax**

\[
\text{INTINDEX}(\text{interval}[\text{multiple}][,\text{shift-index}],\text{date-value}[,,\text{seasonality}])
\]

**Arguments**

\(\text{interval}[\text{multiple}][,\text{shift-index}]\)

specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

- \(\text{interval}\) specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.
  
  **Data type** CHAR
  
  **Note** The possible values of \(\text{interval}\) are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.
  
  **Tip** Interval can appear in uppercase or lowercase.
  
  **Example** YEAR specifies year-based intervals.

\(\text{multiple}\)

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

**Data type** INTEGER

**See** “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

**Example** YEAR2 specifies a two-year, or biennial, interval type.

\(\text{shift-index}\)

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

**Restrictions** The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.
If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

**Data type**
INTEGER

**See**
“Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

**Example**
YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-value
specifies a date, time, or timestamp value that represents a time period of the given interval.

**Data type**
DOUBLE

**seasonality**
specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

**Data type**
INTEGER, CHAR

**Example**
In this example, the following functions produce the same result.

```
INTINDEX('MONTH', sasdate, 3);
INTINDEX('MONTH', sasdate, 'QTR');
```

*Seasonality* in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

**Details**

*INTINDEX Function Intervals*
The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or timestamp value. The seasonal index is a number that represents the position of the date, time, or timestamp value in the seasonal cycle of the specified interval. This example returns a value of 12 because there are 12 months in a yearly cycle and December is the 12th month of the year.

```
sasdate=to_double(date'2012-12-01');
x=intindex('month', sasdate);
put x;
```
In the following examples, INTINDEX returns the same value (1) because both statements have dates that occur in the first quarter of the year 2013.

```sas
sasdate=to_double(date'2013-01-01');
x=intindex('qtr', sasdate);
put x;

sasdate=to_double(date'2013-03-31');
y=intindex('qtr', sasdate);
put y;
```

The following example returns a value of 6 because daily data is weekly periodic and December 7, 2012, is a Friday, the sixth day of the week.

```sas
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

**How Interval and Date-Time-Value Are Related**

To correctly identify the seasonal index, the interval should agree with the date, time, or timestamp value. For example, `intindex('month', 01DEC2012'd)` returns a value of 12 because there are 12 months in a yearly interval and December is the 12th month of the year. The MONTH interval requires a SAS date value. The following example returns a value of 6 because there are seven days in a weekly interval and December 7, 2012, is a Friday, the sixth day of the week.

```sas
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

The DAY interval requires a SAS date value.

This example returns a missing value because the QTR interval expects the date to be a SAS date value rather than a TIMESTAMP value.

```sas
sasdate=to_double(timestamp'2013-01-01 00:00:00');
x=intindex('qtr', sasdate);
put x;
```

This example returns a value of 12. The DTMONTH interval requires a TIMESTAMP value.

```sas
sasdate=to_double(timestamp'2013-12-01 00:00:00');
x=intindex('dtmonth', sasdate);
put x;
```

For more information about working with date and time intervals, see “Date and Time Intervals” in *SAS Functions and CALL Routines: Reference*.

**Retail Calendar Intervals**

The INTINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in *SAS Functions and CALL Routines: Reference*.

**Seasonality**

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in
winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTINDEX function uses the concept of seasonality and returns the seasonal index when a date, time, or timestamp interval and value are specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the SAS/ETS User’s Guide.

Comparisons
The INTINDEX function returns the seasonal index whereas the INTCINDEX function returns the cycle index.

In the following example, the INTINDEX function returns 5 because April 4, 2013 is on a Thursday, the fifth day of the week.

```
sasdate=to_double(date'2013-04-04');
x = intindex('day', sasdate);
put x;
```

Using the same date, the INTCINDEX function returns 14 because April 4, 2013 is the fourteenth week of the year.

```
sasdate=to_double(date'2013-04-04');
x = intcindex('day', sasdate);
put x;
```

In this example, the INTINDEX function returns the minute of the hour.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
x = intindex('minute', sasdate);
put x;
```

Using the same date and time, the INTCINDEX function returns the hour of the day.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
y = intcindex('minute', sasdate);
put y;
```

In the example `intseas('interval')`, INTSEAS returns the maximum number that could be returned by `intindex('interval', date);`.

Example
The following statements illustrate the INTINDEX function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasdate1=to_double(date'2013-08-14'); interval1 = intindex('qtr', sasdate1); put interval1;</td>
<td>3</td>
</tr>
<tr>
<td>sastime1=to_double(time'09:05:15'); interval3 = intindex('hour', sastime1); put interval3;</td>
<td>10</td>
</tr>
<tr>
<td>sastime1=to_double(time'09:05:15'); interval3 = intindex('hour', sastime1); put interval3;</td>
<td>10</td>
</tr>
<tr>
<td>sastime1=to_double(time'09:05:15'); interval3 = intindex('hour', sastime1); put interval3;</td>
<td>10</td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
`sasdate1=to_double(date '2013-02-26'); interval4 = intindex('month', sasdate1); put interval4;` | 2
`sastsl=to_double(timestamp'2013-05-28 05:15:00'); interval5 = intindex('dtmonth', sastsl); put interval5;` | 5
`sasdate1=to_double(date '2013-09-09'); interval6 = intindex('week', sasdate1); put interval6;` | 37
`sasdate1=to_double(date '2013-04-16'); interval7 = intindex('tenday', sasdate1); put interval7;` | 11

See Also

Functions:

- “INTCINDEX Function” on page 542
- “INTCYCLE Function” on page 552
- “INTSEAS Function” on page 572

Other References:

- *SAS/ETS User's Guide*

**INTNX Function**

Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

**Category:** Date and Time  
**Returned data type:** DOUBLE

**Syntax**

```
INTNX(interval[multiple][.shift-index], start-from, increment[, 'alignment'])
INTNX(start-from, increment[, 'alignment'])
```

**Arguments**

`interval[multiple][.shift-index]`  
specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:
**interval**

A character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

- **Data type**: CHAR
- **Note**: The possible values of interval are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.
- **Tip**: Interval can appear in uppercase or lowercase.
- **Example**: YEAR specifies year-based intervals.

**multiple**

An optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

- **Data type**: INTEGER
- **See**: “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.
- **Example**: YEAR2 specifies a two-year, or biennial, interval type.

**shift-index**

An optional shift index that shifts the interval to start at a specified subperiod starting point.

- **Restrictions**: The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

  If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

- **Data type**: INTEGER
- **See**: “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.
- **Example**: YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

**start-from**

An expression that represents a SAS date, time, or datetime value encoded as a DOUBLE and that identifies a starting point.
**Data type**  
**DOUBLE**

**increment**  
specifies a negative, positive, or zero integer that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

**Data type**  
**INTEGER**

**alignment**  
controls the position of SAS dates within the interval. You must enclose *alignment* in quotation marks. *Alignment* can be one of these values:

- **BEGINNING**  
specifies that the returned date or datetime value is aligned to the beginning of the interval.
  
  **Alias**  
  **B**

- **MIDDLE**  
specifies that the returned date or datetime value is aligned to the midpoint of the interval, which is the average of the beginning and ending alignment values.
  
  **Alias**  
  **M**

- **END**  
specifies that the returned date or datetime value is aligned to the end of the interval.
  
  **Alias**  
  **E**

- **SAME**  
specifies that the date that is returned has the same alignment as the input date.
  
  **Aliases**  
  **S**

**SAME**

**See**  
“SAME Alignment” on page 567

**Default**  
**BEGINNING**

**Data type**  
**CHAR**

**See**  
“Aligning SAS Date Output within Its Intervals” on page 567

### Details

**The Basics**

The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE, or a custom interval that you define. The increment is based on a starting date, time, or datetime value, and on the number of time intervals that you specify.

The INTNX function returns the SAS date value for the beginning date, time, or datetime value of the interval that you specify in the *start-from* argument. (To convert the date value to a calendar date, use any valid DS2 date format, such as the DATE9.)
format.) The following example shows how to determine the date of the start of the week that is six weeks from the week of October 17, 2011.

```sas
sasdate=to_double(date'2011-10-17');
x=intnx('week', sasdate, 6);
put x date9.;
```

INTNX returns the value 27NOV2011.

For more information about working with date and time intervals, see “Date and Time Intervals” in SAS Functions and CALL Routines: Reference.

**Aligning SAS Date Output within Its Intervals**

SAS date values are typically aligned with the beginning of the time interval that is specified with the `interval` argument.

You can use the optional `alignment` argument to specify the alignment of the date that is returned. The values BEGINNING, MIDDLE, or END align the date to the beginning, middle, or end of the interval, respectively.

**SAME Alignment**

If you use the SAME value of the `alignment` argument, then INTNX returns the same calendar date after computing the interval increment that you specified. The same calendar date is aligned based on the interval's shift period, not the interval. To view the valid shift periods, see “Intervals by Category” on page 548.

Most of the values of the shift period are equal to their corresponding intervals. The exceptions are the intervals WEEK, WEEKDAY, QTR, SEMIYEAR, YEAR, and their DT counterparts. WEEK and WEEKDAY intervals have a shift period of DAYS; and QTR, SEMIYEAR, and YEAR intervals have a shift period of MONTH. When you use SAME alignment with YEAR, for example, the result is same-day alignment based on MONTH, the interval's shift period. The result is not aligned to the same day of the YEAR interval. If you specify a multiple interval, then the default shift interval is based on the interval, and not on the multiple interval.

When you use SAME alignment for QTR, SEMIYEAR, and YEAR intervals, the computed date is the same number of months from the beginning of the interval as the input date. The day of the month matches as closely as possible. Because not all months have the same number of days, it is not always possible to match the day of the month.

For more information about shift periods, see “Intervals by Category” on page 548.

**Alignment Intervals**

Use the SAME value of the `alignment` argument if you want to base the alignment of the computed date on the alignment of the input date.

```sas
/*** returns 22MAR2011 ***/
dcl double x having format date9.;
sasdate=to_double(date'2011-03-15');
x=intnx('week', sasdate, 1, 'same');
put x;

/*** returns 22MAR11:08:45:00 ***/
dcl double y having format datetime.;
method init();
sasdt=to_double(timestamp'2011-03-15 08:45:00');
y=intnx('dtweek', sasdt, 1, 'same');
put y;
```
/*** returns 15MAR2016 ***/
dcl double z having format date9.;
sasdate=to_double(date'2011-03-15');
z=intnx('year', sasdate, 5, 'same');
put z;

Adjusting Dates
The INTNX function automatically adjusts for the date if the date in the interval that is
ingcremented does not exist. Here is an example:

/*** returns 15AUG2011 ***/
dcl double a having format date9.;
sasdate=to_double(date'2011-03-15');
a=intnx('month', sasdate, 5, 'same');
put a;

/*** returns 28FEB2014 ***/
dcl double b having format date9.;
sasdate=to_double(date'2012-02-29');
b=intnx('year', sasdate, 2, 'same');
put b;

/*** returns 30SEP2011 ***/
dcl double c having format date9.;
sasdate=to_double(date'2011-08-31');
c=intnx('month', sasdate, 1, 'same');
put c;

/*** returns 01MAR2012 (the 1st day of the 3rd month of the year) ***/
dcl double d having format date9.;
sasdate=to_double(date'2011-03-01');
d=intnx('year', sasdate, 1, 'same');
put d;

/*** returns 29FEB2012 (the 60th day of the year) ***/
dcl double d having format date9.;
sasdate=to_double(date'2011-03-01');
d=intnx('year', sasdate, 1, 'same', 'day');
put d;

In the following example, the INTNX function returns the value 01JAN2014, which is
the beginning of the year two years from the starting date (29FEB2012).

dcl double a having format date9.;
sasdate=to_double(date'2012-02-29');
a=intnx('year', sasdate, 2); 
put a;

In this example , the INTNX function returns the value 28FEB2014. In this case, the
starting date begins in the year 2012, the year is two years later (2014), the month is the
same (February), and the date is the 28th, because that is the closest date to the 29th in
February 2014.

dcl double b having format date9.;
Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Examples

Example 1: Using the INTNX Function

The following statements illustrate the INTNX function.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasdate1 = to_double(date'2013-02-05');</td>
<td></td>
</tr>
<tr>
<td>yr=intnx('year', sasdate1, 3);</td>
<td>20454</td>
</tr>
<tr>
<td>put yr;</td>
<td>01Jan16</td>
</tr>
<tr>
<td>put yr date7.;</td>
<td></td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2013-01-05');</td>
<td></td>
</tr>
<tr>
<td>x=intnx('month', sasdate1, 0);</td>
<td>19359</td>
</tr>
<tr>
<td>put x;</td>
<td>01JAN13</td>
</tr>
<tr>
<td>put x date7.;</td>
<td></td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2013-01-01');</td>
<td></td>
</tr>
<tr>
<td>next=intnx('semiyear', sasdate1, 1);</td>
<td>19540</td>
</tr>
<tr>
<td>put next;</td>
<td>01JUL13</td>
</tr>
<tr>
<td>put next date7.;</td>
<td></td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2012-08-01');</td>
<td></td>
</tr>
<tr>
<td>past=intnx('month2', sasdate1, -1);</td>
<td>19114</td>
</tr>
<tr>
<td>put past;</td>
<td>01MAY12</td>
</tr>
<tr>
<td>put past date7.;</td>
<td></td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2013-04-01');</td>
<td></td>
</tr>
<tr>
<td>sm=intnx('seimonth2.2', sasdate1, 4);</td>
<td>19555</td>
</tr>
<tr>
<td>put sm;</td>
<td>16JUL13</td>
</tr>
<tr>
<td>put sm date7.;</td>
<td></td>
</tr>
<tr>
<td>x='month';</td>
<td></td>
</tr>
<tr>
<td>sasdate1 = to_double(date'2013-06-01');</td>
<td></td>
</tr>
<tr>
<td>nextmon=intnx(x, sasdate1, 1);</td>
<td>19540</td>
</tr>
<tr>
<td>put nextmon;</td>
<td>01JUL13</td>
</tr>
<tr>
<td>put nextmon date7.;</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: Using the ALIGNMENT Argument

The following examples show the results of advancing a date by using the optional \textit{alignment} argument.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m1='month '}; \texttt{m2=trim(m1)}; \texttt{sasdate1 = to_double(date'2012-06-15')}; \texttt{x=intnx(m2, sasdate1, 1)}; put x; put x date7.;</td>
<td>19175 01JUL12</td>
</tr>
<tr>
<td>\texttt{sasdate1 = to_double(date'2013-01-01')}; \texttt{x=intnx('month', sasdate1, 5, 'beginning')}; put x; put x date7.;</td>
<td>19510 01JUN13</td>
</tr>
<tr>
<td>\texttt{sasdate1 = to_double(date'2013-01-01')}; \texttt{x=intnx('month', sasdate1, 5, 'middle')}; put x; put x date7.;</td>
<td>19524 15JUN13</td>
</tr>
<tr>
<td>\texttt{sasdate1 = to_double(date'2013-01-01')}; \texttt{x=intnx('month', sasdate1, 5, 'end')}; put x; put x date7.;</td>
<td>19539 30JUN13</td>
</tr>
<tr>
<td>\texttt{sasdate1 = to_double(date'2013-01-01')}; \texttt{x=intnx('month', sasdate1, 5, 'sameday')}; put x; put x date7.;</td>
<td>19510 01JUN13</td>
</tr>
<tr>
<td>\texttt{sasdate1 = to_double(date'2013-03-15')}; \texttt{x=intnx('month', sasdate1, 5, 'same')}; put x; put x date7.;</td>
<td>19585 15AUG13</td>
</tr>
<tr>
<td>\texttt{interval='month'; align='m'; sasdate1 = to_double(date'2013-09-01')}; \texttt{x=intnx(interval, sasdate1,2, align)}; put x; put x date7.;</td>
<td>19667 15NOV13</td>
</tr>
<tr>
<td>\texttt{m1='month '}; \texttt{m2=trim(m1)}; \texttt{sasdate1 = to_double(date'2012-09-01')}; \texttt{x=intnx(m2, sasdate1, 2, 'm')}; put x; put x date7.;</td>
<td>19312 15NOV12</td>
</tr>
</tbody>
</table>
INTRR Function

Returns the internal rate of return as a decimal value.

**Syntax**

\[ \text{INTRR}(freq, c_0, c_1, ..., c_n) \]

**Arguments**

- \( freq \)
  
  is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.
  
  **Range** \( freq > 0 \)
  
  **Data type** \( \text{DOUBLE} \)
  
  **Tip**
  
  The case \( freq = 0 \) is a flag to allow continuous compounding.

- \( c_0, c_1, ..., c_n \)
  
  are numeric, the optional cash payments.
  
  **Data type** \( \text{DOUBLE} \)

**Details**

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments \( c_0, c_1, ..., c_n \). The time intervals between any two consecutive payments are assumed to be equal. The argument \( freq > 0 \) describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value. (See the “NPV Function” on page 669.) It is given by the following equation.

---

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Functions:**

- “INTCK Function” on page 545
- “INTDT Function” on page 555
- “INTSHIFT Function” on page 576
- “INTTS Function” on page 580
\[ r = \begin{cases} \frac{1}{\text{freq}} - 1 & \text{freq} > 0 \\ -\log_x(e) & \text{freq} = 0 \end{cases} \]

In this equation, \( x \) is the real root of the polynomial.

\[ \sum_{i=0}^{n} c_i x^i = 0 \]

In the case of multiple roots, one real root is returned and a warning is issued concerning the non-uniqueness of the returned internal rate of return. Depending on the value of payments, a root for the equation does not always exist. In that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When \( \text{freq} > 0 \), the computed rate of return is the effective rate over the specified base period. To compute a quarterly internal rate of return (the base period is three months) with monthly payments, set \( \text{freq} \) to 3.

If \( \text{freq} \) is 0, continuous compounding is assumed and the base period is the time interval between two consecutive payments. The computed internal rate of return is the nominal rate of return over the base period. To compute with continuous compounding and monthly payments, set \( \text{freq} \) to 0. The computed internal rate of return will be a monthly rate.

**Comparisons**

The IRR function is identical to INTRR, except for in the IRR function, the internal rate of return is a percentage.

**Example**

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return can be expressed as

\[ \text{rate} = \text{intrr}(1, -400, 100, 200, 300); \]

The value that is returned is 0.19437709962747.

**See Also**

**Functions:**

- “IRR Function” on page 585

---

**INTSEAS Function**

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

**Category:** Date and Time

**Returned data type:** DOUBLE
Syntax

\texttt{INTSEAS(interval}[\texttt{multiple}][.\texttt{shift-index}][, \texttt{seasonality}])

\textbf{Arguments}

\texttt{interval}[\texttt{multiple}][.\texttt{shift-index}]  
specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

\texttt{interval}  
specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

\textbf{Data type} CHAR

\textbf{Note}  
The possible values of \texttt{interval} are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.

\textbf{Tip}  
\texttt{Interval} can appear in uppercase or lowercase.

\textbf{Example}  
\texttt{YEAR} specifies year-based intervals.

\texttt{multiple}  
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

\textbf{Data type} INTEGER

\textbf{See}  
“Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

\textbf{Example}  
\texttt{YEAR2} specifies a two-year, or biennial, interval type.

\texttt{shift-index}  
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

\textbf{Restrictions}  
The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use \texttt{YEAR2.24}, but \texttt{YEAR2.25} would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because \texttt{MONTH} type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two \texttt{MONTH} intervals in each \texttt{MONTH2} interval. For example, the interval name \texttt{MONTH2.2} specifies bimonthly periods starting on the first day of even-numbered months.

\textbf{Data type} INTEGER
See “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

Example
YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type INTEGER, CHAR

Example The seasonality argument in the following example
INTSEAS('MONTH', 'qtr');
causes the function call to return the value 3. The function call
INTSEAS('MONTH');
does not have a seasonality argument and returns the value 12.

Details

The Basics

The INTSEAS function returns the number of intervals in a seasonal cycle. For example, when the interval for a time series is described as monthly, then many procedures use the option INTERVAL=MONTH. Each observation in the data then corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12.

Quarterly data is also considered to be periodic for a one-year period. A year contains four quarters, so the number of intervals in a seasonal cycle is four.

The periodicity is not always one year. For example, INTERVAL=DAY is considered to have a period of one week. Because there are seven days in a week, the number of intervals in the seasonal cycle is seven.

For more information about working with date and time intervals, see “Date and Time Intervals” in SAS Functions and CALL Routines: Reference.

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in
winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTSEAS function uses the concept of seasonality and returns the length of the seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and forecasting, see the *SAS/ETS User’s Guide*.

**Example**

The following statements illustrate the INTCYCLE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle_years = intseas('year');</td>
<td>1</td>
</tr>
<tr>
<td>put cycle_years;</td>
<td></td>
</tr>
<tr>
<td>cycle_semiyears = intseas('semiyear');</td>
<td>2</td>
</tr>
<tr>
<td>put cycle_semiyears;</td>
<td></td>
</tr>
<tr>
<td>cycle_quarters = intseas('quarter');</td>
<td>4</td>
</tr>
<tr>
<td>put cycle_quarters;</td>
<td></td>
</tr>
<tr>
<td>cycle_number = intseas('month', 'qtr');</td>
<td>3</td>
</tr>
<tr>
<td>put cycle_number;</td>
<td></td>
</tr>
<tr>
<td>cycle_months = intseas('month');</td>
<td>12</td>
</tr>
<tr>
<td>put cycle_months;</td>
<td></td>
</tr>
<tr>
<td>cycle_semimonths = intseas('semimonth');</td>
<td>24</td>
</tr>
<tr>
<td>put cycle_semimonths;</td>
<td></td>
</tr>
<tr>
<td>cycle_tendays = intseas('tenday');</td>
<td>36</td>
</tr>
<tr>
<td>put cycle_tendays;</td>
<td></td>
</tr>
<tr>
<td>cycle_weeks = intseas('week');</td>
<td>52</td>
</tr>
<tr>
<td>put cycle_weeks;</td>
<td></td>
</tr>
<tr>
<td>cycle_wkdays = intseas('weekday');</td>
<td>5</td>
</tr>
<tr>
<td>put cycle_wkdays;</td>
<td></td>
</tr>
<tr>
<td>cycle_hours = intseas('hour');</td>
<td>24</td>
</tr>
<tr>
<td>put cycle_hours;</td>
<td></td>
</tr>
<tr>
<td>cycle_minutes = intseas('minute');</td>
<td>60</td>
</tr>
<tr>
<td>put cycle_minutes;</td>
<td></td>
</tr>
<tr>
<td>cycle_month2 = intseas('month2.2');</td>
<td>6</td>
</tr>
<tr>
<td>put cycle_month2;</td>
<td></td>
</tr>
<tr>
<td>cycle_week2 = intseas('week2');</td>
<td>26</td>
</tr>
<tr>
<td>put cycle_week2;</td>
<td></td>
</tr>
<tr>
<td>var1 = 'month4.3';</td>
<td>3</td>
</tr>
<tr>
<td>cycle_var1 = intseas(var1);</td>
<td></td>
</tr>
<tr>
<td>put cycle_var1;</td>
<td></td>
</tr>
</tbody>
</table>
cycle_day1 = intseas('day1');
put cycle_day1;

7

See Also

Functions:

- “INTCYCLE Function” on page 552
- “INTINDEX Function” on page 560

Other References:

- SAS/ETS User’s Guide

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Category: Date and Time

Returned data type: NCHAR, NVARCHAR

Syntax

INTSHIFT(interval[multiple][.shift-index])

Arguments

interval[multiple][.shift-index]

specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR

Note

The possible values of interval are listed in “Intervals Used with Date and Time Functions” in SAS Language Reference: Concepts.

Tip

Interval can appear in uppercase or lowercase.

Example

YEAR specifies yearly intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.
Data type INTEGER

See “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

Example YEAR2 consists of two-year, or biennial, periods.

shift-index specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type INTEGER

See “Incrementing Dates and Times By Using Multipliers and By Shifting Intervals” in SAS Functions and CALL Routines: Reference for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The INTSHIFT function returns the shift interval that corresponds to the base interval. For custom intervals, the value that is returned is the base custom interval name.

INTSHIFT ignores multiples of the interval and interval shifts.

The INTSHIFT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in SAS Language Reference: Concepts.

Example

The following statements illustrate the INTSHIFT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift1=intshift('year');</td>
<td>MONTH</td>
</tr>
<tr>
<td>put shift1;</td>
<td></td>
</tr>
</tbody>
</table>
Statements           Results
shift2=intshift('dtyear');  DTMONTH
     put shift2;

shift3=intshift('minute');  DTMINUTE
     put shift3;

interval='weekdays';
     shift4 = intshift(interval);
     put shift4;
             WEEKDAY

shift5=intshift('weekday5.4');  WEEKDAY
     put shift5;

shift6=intshift('qtr');
     put shift6;
             MONTH

shift7=intshift('dttenday');  DTTENDAY
     put shift7;

INTTEST Function
Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Category: Date and Time
Returned data type: INTEGER

Syntax
INTTEST(interval[multiple][.shift-index])

Arguments
interval[multiple][.shift-index]
specifies a basic or complex interval. Multipliers and shift indexes can be used with
the basic interval names to construct more complex interval specifications. The three
parts of the interval name are as follows:

interval
specifies a character constant, a variable, or an expression that contains an
interval name such as WEEK, MONTH, or QTR.

Data type  CHAR

Note  The possible values of interval are listed in “Intervals Used with
      Date and Time Functions” in SAS Language Reference: Concepts.

Tip  Interval can appear in uppercase or lowercase.

Example  YEAR specifies year-based intervals.
The INTTEST function checks for a valid interval name. This function is useful when checking for valid values of `multiple` and `shift-index`. For more information about multipliers and shift indexes, see “Multiunit Intervals” in *SAS Language Reference: Concepts*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in *SAS Language Reference: Concepts*.

**Example**

In the following examples, SAS returns a value of 1 if the `interval` argument is valid, and 0 if the interval argument is invalid.
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>test1 = inttest('month');</code></td>
<td>1</td>
</tr>
<tr>
<td><code>put test1;</code></td>
<td></td>
</tr>
<tr>
<td><code>test2 = inttest('week6.13');</code></td>
<td>1</td>
</tr>
<tr>
<td><code>put test2;</code></td>
<td></td>
</tr>
<tr>
<td><code>test3 = inttest('tenday');</code></td>
<td>1</td>
</tr>
<tr>
<td><code>put test3;</code></td>
<td></td>
</tr>
<tr>
<td><code>test4 = inttest('twoweeks');</code></td>
<td>0</td>
</tr>
<tr>
<td><code>put test4;</code></td>
<td></td>
</tr>
<tr>
<td><code>var1 = 'hour2.2';</code></td>
<td></td>
</tr>
<tr>
<td><code>test5 = inttest(var1);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>put test5;</code></td>
<td></td>
</tr>
</tbody>
</table>

### INTTS Function

Specifies the number of seconds to add to a TIMESTAMP value.

**Category:** Date and Time  
**Returned data type:** TIMESTAMP

### Syntax

`INTTS(expression, increment)`

### Arguments

- **expression**
  - specifies any valid expression that represents a TIMESTAMP value.  
  - Data type: TIMESTAMP
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **increment**
  - specifies a negative, positive, or zero integer that represents the number of seconds to add to the time.  
  - Data type: INTEGER

### Details

The INTTS function increments a TIMESTAMP value by the number of seconds that you specify.
Comparisons

The INTNX function increments a SAS date, time, or datetime value encoded as a DOUBLE value.

Example

The following statements illustrate the INTDT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = timestamp '2011-03-01 16:51:36.00'; z = intts(y, 43);</td>
<td>y = 2011-03-01 16:51:36  z = 2011-03-01 16:52:19</td>
</tr>
<tr>
<td>y = timestamp '2011-05-01 02:58:17.00'; z = intdt(y, -2500);</td>
<td>y = 2011-05-01 02:58:17  z = 2011-05-01 02:16:37</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

- “INTCK Function” on page 545
- “INTDT Function” on page 555
- “INTNX Function” on page 564

INTZ Function

Returns the integer portion of the argument, using zero fuzzing.

**Category:** Truncation  
**Returned data type:** DOUBLE

**Syntax**

`INTZ(expression)`

**Arguments**

`expression` specifies any valid expression that evaluates to a numeric value.

**Data type**  
DOUBLE

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**Details**

The following rules apply:
• If the value of the argument is an exact integer, INTZ returns that integer.
• If the argument is positive and not an integer, INTZ returns the largest integer that is less than the argument.
• If the argument is negative and not an integer, INTZ returns the smallest integer that is greater than the argument.

Comparisons
Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example
The following statements illustrate the INTZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=2.1; a=intz(var1);</td>
<td>2</td>
</tr>
<tr>
<td>a=intz(-2.4);</td>
<td>-2</td>
</tr>
<tr>
<td>a=intz(1+1.e11);</td>
<td>1</td>
</tr>
<tr>
<td>a=intz(-1.6);</td>
<td>-1</td>
</tr>
</tbody>
</table>

See Also

Functions:
• “CEIL Function” on page 419
• “CEILZ Function” on page 420
• “FLOOR Function” on page 505
• “FLOORZ Function” on page 506
• “INT Function” on page 541
• “MOD Function” on page 631
• “MODZ Function” on page 633
• “ROUND Function” on page 756
• “ROUNDDZ Function” on page 767

IPMT Function
Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
Syntax

IPMT(\(\text{rate}, \text{period}, \text{number-of-periods}, \text{principal-amount}, [\text{future-amount}], [\text{type}]\))

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

period
specifies the payment period for which the interest payment is computed.

Requirement \(\text{Period} \) must be a positive integer value that is less than or equal to the value of \(\text{number-of-periods}\).

Data type INTEGER

number-of-periods
specifies the number of payment periods.

Requirement \(\text{Number-of-periods} \) must be a positive integer value.

Data type INTEGER

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing value is specified.

future-amount
specifies the future amount.

Data type DOUBLE

Notes \(\text{Future-amount} \) can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings.

Zero is assumed if \(\text{future-amount} \) is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments.

Data type INTEGER
Note 0 is assumed if type is omitted or if a missing value is specified.

Example
The interest payment on the first periodic payment of an $8,000 loan, where the nominal annual interest rate is 10% and the end-of-period monthly payments are 36, is computed as follows:

\[ \text{InterestPaid1} = \text{ipmt}(0.1/12, 1, 36, 8000); \]

This computation returns a value of 66.6666666666666.

If the same loan has beginning-of-period payments, then the interest payment can be computed as follows:

- \[ \text{InterestPaid2} = \text{ipmt}(0.1/12, 1, 36, 8000, 0, 1); \]
  This computation returns a value of 0.
- \[ \text{InterestPaid3} = \text{ipmt}(0.1, 3, 3, 8000); \]
  This computation returns a value of 292.447129909366.
- \[ \text{InterestPaid4} = \text{ipmt}(0.09/12, 359, 360, 125000, 0, 1); \]
  This computation returns a value of 14.8075736630449.

IQR Function
Returns the interquartile range.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

\[ \text{IQR(expression [, …expression])} \]

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Data type</th>
<th>DOUBLE</th>
</tr>
</thead>
</table>

See Chapter 13, “DS2 Expressions,” on page 93

Details
If all arguments have null or missing values, the result is a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

Otherwise, the result is the interquartile range of the non-null or nonmissing values. The formula for the interquartile range is the same as the one that is used in the Base SAS.
UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following statement illustrates the IQR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=iqr(2,4,1,3,999999);</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “MAD Function” on page 615
- “PCTL Function” on page 676

IRR Function

Returns the internal rate of return as a percentage.

Syntax

\[
\text{IRR}(freq, c1, c2, \ldots, cn)
\]

Arguments

\(freq\)

is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range \(freq > 0\).

Data type DOUBLE

Tip The case \(freq = 0\) is a flag to allow continuous compounding.

\(c1, c2, \ldots, cn\)

are numeric, the optional cash payments.

Requirement At minimum, two cash payment values are required.

Data type DOUBLE
Details
The IRR function returns the internal rate of return over a specified base period of time for the set of cash payments \(c_1, c_2, \ldots, c_n\). The time intervals between any two consecutive payments are assumed to be equal. The argument \(freq > 0\) describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

Comparisons
The IRR function is identical to INTRR, except that in the IRR function, the internal rate of return is a percentage.

Example
For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return as a percentage can be expressed as
\[
rate = \text{irr}(1, -400, 100, 200, 300);
\]
The value that is returned is 19.437709962747.

See Also
Functions:
- “INTRR Function” on page 571

JULDATE Function
Returns the Julian date from a SAS date value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
JULDATE(date)

Arguments
date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details
A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If
date falls within the 100-year span defined by the system option YEARCUTOFF=, the
result has three, four or five digits: In a five-digit result, the first two digits represent the
year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap
years). As leading zeros are dropped from the result, the year portion of a Julian date can
be omitted (for years ending in 00) or it can have only one digit (for years ending 01–
09). Otherwise, the result has seven digits: the first four digits represent the year, and the
next three digits represent the day of the year.

For years that end between 00–09, you can format the five-digit Julian date by using the
Z5. format.

For more information about how DS2 handles dates, see Chapter 14, “Dates and Times
in DS2,” on page 111.

Comparisons
The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns
a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the
need to consider the implications of a two-digit year.

Example
The following statements illustrate the JULDATE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>julian=juldate(mdy(12,31,2013));</td>
<td>7365</td>
</tr>
<tr>
<td>julian=put(juldate(mdy(12,31,2013)),Z5.);</td>
<td>07365</td>
</tr>
<tr>
<td>julian=juldate(mdy(9,1,1999));</td>
<td>99244</td>
</tr>
<tr>
<td>julian=juldate(mdy(7,1,1886));</td>
<td>1886182</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “DATEJUL Function” on page 466
- “JULDATE7 Function” on page 587
Syntax

JULDATE7(date)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type: DOUBLE

See: Chapter 13, “DS2 Expressions,” on page 93

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE7 function returns a seven-digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

For more information about how DS2 handles dates, see Chapter 14, “Dates and Times in DS2,” on page 111.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE7 function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>julian=juldate7(mdy(12,31,2006));</td>
<td>2007365</td>
</tr>
<tr>
<td>julian=juldate7(mdy(12,31,2016));</td>
<td>2016366</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “JULDATE Function” on page 586

KCOUNT Function

Returns the number of double-byte characters in an expression.

Category: DBCS

Returned data type: INTEGER
### Syntax

\[ \text{KCOUNT(}'\text{source}'\text{'}) \]

### Arguments

'\text{source}'

specifies any valid expression that evaluates to a character string.

- **Data type**: NCHAR
- **Tip**: Enclose a literal string of characters in quotation marks.

### Details

For restrictions and more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS): Reference Guide*.

### Example

The following example uses Japanese characters.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| proc ds2; data _null_; method run(); c = kcount('Abc def'); put c=; c = kcount('SAS漢字'); put c=; end; enddata; run; quit; | c=0  
c=2 |

### See Also

- “COUNT Function” on page 449

---

**KSTRCAT Function**

Concatenates two or more character expressions.

- **Category**: DBCS
- **Returned data type**: NCHAR
Syntax

KSTRCAT(expression-1, expression-2[, … expression-n])

Arguments

expression
  specifies any single-byte or double-byte character expression.

Requirement  At least two expressions are required.

Data type  NCHAR

Details

For restrictions and more information, see “Internationalization Compatibility for SAS String Functions” in SAS National Language Support (NLS): Reference Guide.

KSTRCAT concatenates two or more single-byte or double-byte character expressions. It also removes unnecessary shift out/shift in escape code (SO/SI) pairs between the expressions.

Example

The following example uses Japanese characters.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc ds2;</td>
<td></td>
</tr>
<tr>
<td>data <em>null</em>;</td>
<td>a=SAS漢字SAS</td>
</tr>
<tr>
<td>method run();</td>
<td></td>
</tr>
<tr>
<td>dcl nchar(8) a;</td>
<td></td>
</tr>
<tr>
<td>a = kstrcat('SAS', '漢字', 'SAS');</td>
<td></td>
</tr>
<tr>
<td>put a=;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
<tr>
<td>enddata;</td>
<td></td>
</tr>
<tr>
<td>run;</td>
<td></td>
</tr>
<tr>
<td>quit;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:
  • “CAT Function” on page 409

KSTRIP Function

Returns a character string with all leading and trailing blanks removed.

Category: DBCS
The KSTRIP function returns the argument with all leading and trailing single-byte character set (SBCS) blanks removed. If the argument is blank, KSTRIP returns a string with a length of zero.

Assigning the results of KSTRIP to a variable does not affect the length of the receiving variable. If the value that is trimmed is shorter than the length of the receiving variable, DS2 pads the value with new trailing blanks.

Example

The following example shows how the KSTRIP function deletes leading and trailing blanks, and how the double-byte character set (DBCS) character is truncated. This example uses the Japanese Shift_JIS encoding.
See Also

Functions
- “STRIP Function” on page 790

KUPDATE Function

Inserts, deletes, and replaces character value contents.

<table>
<thead>
<tr>
<th>Category:</th>
<th>DBCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

**Syntax**

\[
\text{KUPDATE}(["expression"]\[,\] \text{position} \[,\] n \[,\] \text{characters-to-replace}])
\]

**Arguments**

- \["expression"]
  - specifies any valid expression that evaluates to a character string.
  - Data type: NCHAR
  - Tip: Enclose a literal string of characters in quotation marks.
**position**

specifies a numeric expression that is the beginning character position.

Data type: INTEGER

**n**

specifies a numeric expression that is the length of the substring to be replaced.

Restrictions: 

- \( n \) cannot be larger than the length of the expression that remains in \( expression \) after \( position \).
- \( n \) is optional, but you cannot omit both \( n \) and \( characters-to-replace \) from the function.

Data type: INTEGER

Tip: If you omit \( n \), SAS uses all of the characters in \( characters-to-replace \) to replace the values of \( expression \).

\[ \text{ Characters-to-replace } \]

specifies an expression that evaluates to a character string that replaces the contents of \( expression \).

Restriction: \( characters-to-replace \) is optional, but you cannot omit both \( characters-to-replace \) and \( n \) from the function.

Data type: NCHAR

Tip: Enclose a literal string of characters in quotation marks.

**Details**

For restrictions and more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS): Reference Guide*.

The KUPDATE function replaces the value of \( expression \) with the expression in \( characters-to-replace \). KUPDATE replaces \( n \) characters starting at the character that you specify in \( position \).

If you omit \( characters-to-replace \), \( n \) characters are removed from the string.

If you specify more \( characters-to-replace \) than \( n \), all characters are used as replacement. \( n \) is ignored.

If you specify fewer \( characters-to-replace \) than \( n \), only the first \( n \) characters are replaced. The other characters are deleted.

Here are some examples.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>kupdate(&quot;123456&quot;,2,3);</td>
<td>1 56</td>
</tr>
<tr>
<td>kupdate(&quot;123456&quot;,2,3,abcd);</td>
<td>1abc56</td>
</tr>
<tr>
<td>kupdate(&quot;123456&quot;,2,3, &quot;ab&quot;);</td>
<td>1ab 56</td>
</tr>
</tbody>
</table>
Comparisons

The KUPDATE function tries to update the target string (expression) using the actual length of the last argument (characters-to-replace), in spite of the third argument (n). The KUPDATES function tries to update the target string (expression) based on the third argument (n).

Example

The following example uses Japanese characters.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc ds2;</td>
<td></td>
</tr>
<tr>
<td>data <em>null</em>;</td>
<td></td>
</tr>
<tr>
<td>method run();</td>
<td></td>
</tr>
<tr>
<td>str='北京賽仕軟件';</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3);</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3,'abcd');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3,'ab');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,'ab');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>str='北京賽仕軟件';</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3);</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3,'分析数据');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,3,'分析');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>r=kupdate(str,2,'分析');</td>
<td></td>
</tr>
<tr>
<td>put r;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
<tr>
<td>enddata;</td>
<td></td>
</tr>
<tr>
<td>run;</td>
<td></td>
</tr>
<tr>
<td>quit;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:
KUPDATES Function

Inserts, deletes, and replaces character value contents.

**Category:** DBCS

**Returned data type:** NCHAR

**Syntax**

\[
\text{KUPDATE}(['expression'], \text{position}, n [ [, 'characters-to-replace']])
\]

\[
\text{KUPDATE}(['expression'], \text{position}, n)
\]

**Arguments**

- **['expression']**
  Specifies any valid expression that evaluates to a character string.
  
  **Data type:** NCHAR
  
  **Tip:** Enclose a literal string of characters in quotation marks.

- **position**
  Specifies a numeric expression that is the beginning character position.
  
  **Data type:** INTEGER

- **n**
  Specifies a numeric expression that is the length of the substring to be replaced.
  
  **Restrictions:**
  
  - \( n \) cannot be larger than the length of the expression that remains in \( expression \) after \( \text{position} \).
  
  - \( n \) is optional, but you cannot omit both \( n \) and \( \text{characters-to-replace} \) from the function.
  
  **Data type:** INTEGER
  
  **Tip:** If you omit \( n \), SAS uses all of the characters in \( \text{characters-to-replace} \) to replace the values of \( expression \).

- **['characters-to-replace']**
  Specifies an expression that evaluates to a character string that replaces the contents of \( expression \).
  
  **Restriction:** \( \text{characters-to-replace} \) is optional, but you cannot omit both \( \text{characters-to-replace} \) and \( n \) from the function.
  
  **Data type:** NCHAR
  
  **Tip:** Enclose a literal string of characters in quotation marks.
Details

For restrictions and more information, see “Internationalization Compatibility for SAS String Functions” in SAS National Language Support (NLS): Reference Guide.

The KUPDATE function replaces the value of expression with the expression in characters-to-replace. KUPDATE replaces \( n \) characters starting at the character that you specify in position.

If you omit characters-to-replace, \( n \) blank characters are used to replace the existing value.

If you specify more characters-to-replace than \( n \), only the first \( n \) characters are used as replacement.

If you specify fewer characters-to-replace than \( n \), blank characters are added at the end until the number of characters replaced is equal to \( n \).

Here are some examples.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>kupdate(&quot;123456&quot;,2,3);</td>
<td>1 56</td>
</tr>
<tr>
<td>kupdate(&quot;123456&quot;,2,3,&quot;abcd&quot;);</td>
<td>labc56</td>
</tr>
<tr>
<td>kupdate(&quot;123456&quot;,2,3,&quot;ab&quot;);</td>
<td>lab 56</td>
</tr>
</tbody>
</table>

Comparisons

The KUPDATES function tries to update the target string (expression) based on the third argument (\( n \)). The KUPDATE function tries to update the target string (expression) using the actual length of the last argument (characters-to-replace), in spite of the third argument (\( n \)).

Example

The following example uses Japanese characters.
KURTOSIS Function

Returns the kurtosis.

<table>
<thead>
<tr>
<th>Category</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

See Also

Function:
- “KUPDATE Function” on page 592

proc ds2;
data _null_
   method run();
   str='北京赛仕软件';
   r=kupdates(str,2,3);
   put r=
   r=kupdates(str,2,3,'abcd');
   put r=
   r=kupdates(str,2,3,'ab');
   put r=
   r=kupdates(str,2,'ab');
   put r=
   str='北京赛仕软件';
   r=kupdates(str,2,3);
   put r=
   r=kupdates(str,2,3,'分析数据');
   put r=
   r=kupdates(str,2,3,'分析');
   put r=
   r=kupdates(str,2,'分析');
   put r=
end;
enddata;
run;
quit;
Syntax
KURTOSIS(expression-1, expression-2, expression-3, expression-4 [,..., expression-n])

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

Requirement
At least four non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type
DOUBLE

See
Chapter 13, “DS2 Expressions,” on page 93

Details
Kurtosis is primarily a measure of the heaviness of the tails of a distribution. Large kurtosis values indicate that the distribution has heavy tails.

Null values and missing values are ignored and are not included in the computation.

If all non-null or nonmissing arguments have equal values, the kurtosis is mathematically undefined and the KURTOSIS function returns a null or missing value.

Example
The following statements illustrate the KURTOSIS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=kurtosis(5,9,3,6);</td>
<td>0.92799999999999</td>
</tr>
<tr>
<td>b=kurtosis(5,8,9,6,);</td>
<td>-3.3</td>
</tr>
<tr>
<td>c=kurtosis(8,9,6,1);</td>
<td>1.5</td>
</tr>
<tr>
<td>d=kurtosis(8,1,6,1);</td>
<td>-4.48337950138504</td>
</tr>
</tbody>
</table>

LARGEST Function
Returns the kth largest non-null or nonmissing value.

Category: Descriptive Statistics
Returned data type: DOUBLE
Syntax

LARGEST(k, expression [, ...expression])

Arguments

k
specifies any valid expression that evaluates to a numeric value that represents the largest value to return. For example, if k is 2, the LARGEST function returns the second largest value from the list of expressions.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value and that is to be searched.

Data type DOUBLE

Details

If k is null or missing, less than zero, or greater than the number of values, the result is a null or missing value. Otherwise, if k is greater than the number of non-null or nonmissing values, the result is a null or missing value.

Example

The following statements illustrate the LARGEST function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=1;</td>
<td></td>
</tr>
<tr>
<td>largest1=largest(k, 456, 789, .Q, 123);</td>
<td>789</td>
</tr>
<tr>
<td>k=2;</td>
<td></td>
</tr>
<tr>
<td>largest2=largest(k, 456, 789, .Q, 123);</td>
<td>456</td>
</tr>
<tr>
<td>k=3;</td>
<td></td>
</tr>
<tr>
<td>largest3=largest(k, 456, 789, .Q, 123);</td>
<td>123</td>
</tr>
<tr>
<td>k=4;</td>
<td></td>
</tr>
<tr>
<td>largest4=largest(k, 456, 789, .Q, 123);</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

- “ORDINAL Function” on page 675
- “PCTL Function” on page 676
LBOUND Function

Returns the lower bound of an array.

Category: Array

Returned data type: INTEGER

Syntax

LBOUND(array-name[, bound-n])

Arguments

array-name
 specifies the name of a temporary or a variable array.

bound-n
 is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the lower bound.

If no bound-n value is specified, the LBOUND function returns the lower bound of the first dimension of the array.

Bound-n evaluates to an integral value.

Details

The LBOUND function returns the lower bound of a one-dimensional array, or the lower bound of a specified dimension of a multidimensional array. LBOUND and HBOUND can be used together to return the values of the lower and upper bounds of an array dimension.

If the LBOUND function is called with a dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

Comparisons

• DIM returns the number of elements in an array dimension.
• HBOUND returns the value of the upper bound of an array dimension.
• LBOUND returns the value of the lower bound of an array dimension.
• NDIMS returns the number of dimensions in an array.

Example

The following example shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```plaintext
data _null_;  
    method init();
    declare char[15] a1[4];  
    declare double a2[2,3,4] sum;
```
a1 := ('red' 'yellow' 'green' 'blue');
a2 := (24*2.0);

do i = 1 to dim(a1);
   put a1[i];
end;

numelems = 0;
do i = 1 to ndims(a2);
   numelems = numelems + dim(a2, i);
end;

sum = 0;
do i = lbound(a2, 1) to hbound(a2, 1);
do j = lbound(a2, 2) to hbound(a2, 2);
do k = lbound(a2, 3) to hbound(a2, 3);
   sum = sum + a2[i,j,k];
end;
end;
end;

put sum=;
end;
enddata;
run;

SAS writes the following output to the log:

red
yellow
green
blue
sum=48

See Also

Functions:
- “DIM Function” on page 478
- “HBOUND Function” on page 526
- “NDIMS Function” on page 638

LCM Function

Returns the least common multiple for a set of integers.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Mathematical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
Syntax

\[ \text{LCM}(\text{expression-1}, \text{expression-2} [, ... \text{expression-n}]) \]

Arguments

\text{expression}

specifies any valid expression that evaluates to an integer.

Requirement  
At least two arguments are required.

Data type  
DOUBLE

See  
Chapter 13, “DS2 Expressions,” on page 93

Details

The least common multiple is the smallest number that two or more numbers will divide into evenly.

Example

The following statements illustrate the LCM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = \text{lcm}(1,5,3,0); )</td>
<td>0</td>
</tr>
<tr>
<td>( b = \text{lcm}(25,70,85,130); )</td>
<td>77350</td>
</tr>
<tr>
<td>( c = \text{lcm}(33,78); )</td>
<td>858</td>
</tr>
</tbody>
</table>

See Also

Functions:

•  “GCD Function” on page 517

LEFT Function

Left aligns a character expression.

Category:  Character

Returned data type:  VARCHAR, NVARCHAR

Syntax

\[ \text{LEFT}(\text{expression}) \]
**Arguments**

*expression*

specifies any valid expression that evaluates to a character string.

**Data type** CHAR, NCHAR

**See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

LEFT returns a character string with leading blanks moved to the end of the value.

**Example**

The following statements illustrate the LEFT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=' END-OF-YEAR';</td>
<td>END-OF-YEAR</td>
</tr>
<tr>
<td>b=left(a);</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “COMPRESS Function” on page 437
- “RIGHT Function” on page 754
- “STRIP Function” on page 790
- “TRIM Function” on page 821

---

**LENGTH Function**

Returns the length of a character string, excluding trailing blanks, and returns a 0 for a blank character string.

**Category:** Character

**Alias:** LENGTHN

**Returned data type:** DOUBLE

**Syntax**

LENGTH(expression)
Arguments

data type specifies any valid expression that evaluates to a character string.

Data type  CHAR, NCHAR

See  Chapter 13, “DS2 Expressions,” on page 93

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character or number in expression. If the value of expression is a blank character, LENGTH returns a value of 0. If expression is a numeric expression, LENGTH converts and processes the expression as a character expression.

Comparisons

- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks.
- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string.

Example

The following statements illustrate the LENGTH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=length('ABCDEF');</td>
<td>6</td>
</tr>
<tr>
<td>b=date(); a=length(b);</td>
<td>5</td>
</tr>
<tr>
<td>a=length(' ');</td>
<td>0</td>
</tr>
<tr>
<td>a=length(.);</td>
<td>1</td>
</tr>
</tbody>
</table>

* The data type for b is converted to NCHAR with a value of 16763 whose length is 5.
** The data type for the null or missing value (. ) is converted from DOUBLE to NCHAR with a value of . whose length is 1.

See Also

Functions:

- “LENGTHC Function” on page 605
- “LENGTHM Function” on page 606
- “LENGTHN Function” on page 607
LENGTHC Function

Returns the length of a character string, including trailing blanks.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

LENGTHC(expression)

**Arguments**

- **expression**
  - specifies any valid expression that evaluates to a character string.
  - Data type: NCHAR

**Details**

The LENGTHC function returns an integer that represents the position of the rightmost blank or non-blank character in expression. For fixed-length variables, LENGTHC returns the declared length of the variable. If the value of expression is missing and contains blanks, LENGTHC returns the number of blanks in expression. If expression is a numeric expression, LENGTHC converts and processes the numeric expression as a character expression.

**Comparisons**

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH function returns the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the value returned by LENGTH.

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. For fixed-length character strings, LENGTHC and LENGTHM always return the same value. For varying-length character strings, LENGTHC always returns a value that is less than or equal to the value returned by LENGTHM.

**Example**

The following statements illustrate the LENGTHC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=lengthc('string with trailing blanks ');</td>
<td>32</td>
</tr>
</tbody>
</table>
See Also

Functions:

- “LENGTH Function” on page 603
- “LENGTHM Function” on page 606
- “LENGTHN Function” on page 607

LENGTHM Function

Returns the amount of memory, in characters, that is allocated for a character string.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

LENGTHM(expression)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type: NCHAR

See

Chapter 13, “DS2 Expressions,” on page 93

Details

The LENGTHM function returns an integer that represents the amount of memory in bytes that is allocated for the expression. If expression is a numeric expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHM returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.
Comparisons

The LENGTHM function returns the amount of memory in characters that is allocated for a character string, whereas the LENGTH and LENGTHC functions return the length of a character string. LENGTHM always returns a value that is greater than or equal to the values returned by LENGTH and LENGTHC.

Example

The following statements illustrate the LENGTHM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=lengthm('ABCDEF ');</td>
<td>20</td>
</tr>
<tr>
<td>dcl char(30) string;</td>
<td></td>
</tr>
<tr>
<td>string='The Power to Know. ';</td>
<td></td>
</tr>
<tr>
<td>a=lengthm(string);</td>
<td>60</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “LENGTH Function” on page 603
- “LENGTHC Function” on page 605
- “LENGTHN Function” on page 607

LENGTHN Function

Returns the length of a character string, excluding trailing blanks.

**Category:** Character  
**Returned data type:** INTEGER

**Syntax**

LENGTHN(*string*)

**Arguments**

*expression*

specifies any valid expression that evaluates to a character string.

**Data type**  
CHAR

See  
Chapter 13, “DS2 Expressions,” on page 93
Details
The LENGTHN function returns an integer that represents the position of the rightmost non-blank character in string. If the value of string is blank, LENGTHN returns a value of 0. If string is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons
• The LENGTHN and LENGTH functions return the same value for non-blank character strings. LENGTHN returns a value of 0 for blank character strings, whereas LENGTH returns a value of 1.
• The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHC.
• The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHM.

Example
The following statements illustrate the LENGTHN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>c=lengthn('   abc   '); put c;</td>
<td>6</td>
</tr>
<tr>
<td>d=lengthn('abc   '); put d;</td>
<td>3</td>
</tr>
<tr>
<td>e=lengthn(18); put e;</td>
<td>2</td>
</tr>
<tr>
<td>f=lengthn(' '); put f;</td>
<td>0</td>
</tr>
</tbody>
</table>

See Also

Functions:
• “LENGTH Function” on page 603
• “LENGTHC Function” on page 605
• “LENGTHM Function” on page 606

LGAMMA Function
Returns the natural logarithm of the Gamma function.
**Syntax**

`LGAMMA(expression)`

**Arguments**

- `expression`: specifies any valid expression that evaluates to a numeric value.
  - **Requirement**: Must be a positive number.
  - **Data type**: DOUBLE
  - **See**: Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the `LGAMMA` function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a=lgamma(2);</code></td>
<td>0</td>
</tr>
<tr>
<td><code>a=lgamma(1.5);</code></td>
<td>-0.12078223763524</td>
</tr>
</tbody>
</table>

---

**LOG Function**

Returns the natural logarithm (base e) of a numeric value expression.

- **Category**: Mathematical
- **Returned data type**: DOUBLE

**Syntax**

`LOG(expression)`

**Arguments**

- `expression`: specifies any valid expression that evaluates to a numeric value.
  - **Requirement**: Must be a positive number.
  - **Data type**: DOUBLE
  - **See**: Chapter 13, “DS2 Expressions,” on page 93
Example

The following statements illustrate the LOG function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=log(1.0);</td>
<td>0</td>
</tr>
<tr>
<td>a=log(10.0);</td>
<td>2.30258509299404</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “LOG10 Function” on page 611
- “LOG2 Function” on page 613

LOGBETA Function

Returns the logarithm of the beta function.

Category: Mathematical
Returned data type: DOUBLE

Syntax

LOGBETA(a, b)

Arguments

a
is the first shape parameter, where \( a > 0 \).
Data type \( \text{DOUBLE} \)

b
is the second shape parameter, where \( b > 0 \).
Data type \( \text{DOUBLE} \)

Details

The LOGBETA function is mathematically given by the equation

\[
\log(\beta(a, b)) = \log(\Gamma(a)) + \log(\Gamma(b)) - \log(\Gamma(a + b))
\]

In the equation, \( \Gamma(.) \) is the gamma function.
If the expression cannot be computed, LOGBETA returns a missing value.
Examples

**Example 1**
The following DS2 statements compute the logarithm of the beta function. The first shape parameter is 5 and the second shape parameter is 3.

```sas
data test;
  method run();
    y=logbeta(5,3);
    put y=;
  end;
enddata;
run;
```

The following line is written to the SAS log.

```
y=-4.65396035015752
```

**Example 2**
The following statement illustrates the LOGBETA function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>select logbeta(5,3);</td>
<td>-4.6539603501575</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “BETA Function” on page 394

---

**LOG10 Function**

Returns the base-10 logarithm of a numeric value expression.

**Category:** Mathematical

**Returned data type:** DOUBLE

**Syntax**

```
LOG10(expression)
```

**Arguments**

- `expression` specifies any valid expression that evaluates to a numeric value.

  **Requirement:** Must be a positive number.

  **Data type:** DOUBLE
Example

The following statements illustrate the LOG10 function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=log10(1.0);</td>
<td>0</td>
</tr>
<tr>
<td>a=log10(10.0);</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “LOG Function” on page 609
- “LOG2 Function” on page 613

LOG1PX Function

Returns the log of 1 plus the argument.

Category: Mathematical
Returned data type: DOUBLE

Syntax

\[ \text{LOG1PX}(x) \]

Arguments

\( x \)

specifies a numeric variable, constant, or expression.

Data type: DOUBLE

Details

The LOG1PX function computes the log of 1 plus the argument. The LOG1PX function is mathematically defined by the following equation, where \(-1 < x\):

\[ \text{LOG1PX}(x) = \log(1 + x) \]

When \( x \) is close to 0, \( \text{LOG1PX}(x) \) can be more accurate than \( \text{LOG}(1+x) \).
Examples

**Example 1: Computing the Log with the LOG1PX Function**  
The following example computes the log of 1 plus the value 0.5.

```
data _null_;  
method run();  
x=log1px(0.5);  
put x=;  
end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
x=0.40546510810816
```

**Example 2: Comparing the LOG1PX Function with the LOG Function**  
In the following example, the value of X is computed by using the LOG1PX function. The value of Y is computed by using the LOG function.

```
data _null_;  
method run();  
x=log1px(1.e-5);  
put x= hex16.;  
y=log(1+1.e-5);  
put y= hex16.;  
end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
x=3EE4F8A9AE7317  
y=3EE4F8A9AE7317
```

See Also

Functions:

- “LOG Function” on page 609

LOG2 Function

Returns the base 2 logarithm of a numeric value expression.

**Category:** Mathematical  
**Returned data type:** DOUBLE

**Syntax**

```
LOG2(expression)
```
**Arguments**

*expression* specifies any valid expression that evaluates to a numeric value.

**Requirement**  
Must be a positive number.

**Data type**  
DOUBLE

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the LOG2 function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=log2(8.0);</td>
<td>3</td>
</tr>
<tr>
<td>a=log2(4);</td>
<td>2</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**
- “LOG Function” on page 609
- “LOG10 Function” on page 611

---

**LOWCASE Function**

Converts all letters in a character expression to lowercase.

**Category:**  
Character

**Alias:**  
LOWER

**Returned data type:**  
VARCHAR, NVARCHAR

**Syntax**

LOWCASE(*expression*)

**Arguments**

*expression* specifies any valid expression that evaluates to a character string.

**Requirement**  
Literal character expressions must be enclosed in single quotation marks.

**Data type**  
CHAR, NCHAR
Details

The LOWCASE function copies a character expression, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Comparisons

The UPCASE function converts all letters in an argument to uppercase letters. The LOWCASE function converts all letters in an argument to lowercase letters.

Example

The following statement illustrates the LOWCASE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=lowcase('INTRODUCTION');</td>
<td>introduction</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “UPCASE Function” on page 825

MAD Function

Returns the median absolute deviation from the median.

<table>
<thead>
<tr>
<th>Category</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

MAD(expression-1[, …expression-n])

Arguments

evaluation

specifies any valid expression that evaluates to a numeric value of which the median absolute deviation from the median is to be computed.

Data type DOUBLE

Details

If all arguments have missing or null values, the result is a missing or null value. Otherwise, the result is the median absolute deviation from the median of the
nonmissing or non-null values. The formula for the median is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following statement illustrates the MAD function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>c=mad(2, 4, 1, 3, 5, 999999); put c;</td>
<td>1.5</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “IQR Function” on page 584
- “MEDIAN Function” on page 626
- “PCTL Function” on page 676

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

\[
\text{MARGRCLPRC}(X_1, t, X_2, \text{sigma1}, \text{sigma2}, \text{rho12})
\]

**Arguments**

\(X_1\)

is a nonmissing, positive value that specifies the price of the first asset.

**Requirement**

Specify \(X_1\) and \(X_2\) in the same units.

**Data type**

DOUBLE

\(t\)

is a nonmissing value that specifies the time to expiration, in years.

**Data type**

DOUBLE

\(X_2\)

is a nonmissing, positive value that specifies the price of the second asset.
Requirement Specify $X_2$ and $X_1$ in the same units.

Data type DOUBLE

$\sigma_1$
is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

$\sigma_2$
is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

$\rho_{12}$
specifies the correlation between the first and second assets, $\rho_{x_1, x_2}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

\[
\text{CALL} = X_1 N(d_1) - X_2 N(d_2)
\]

Arguments

$X_1$
specifies the price of the first asset.

$X_2$
specifies the price of the second asset.

$N$
specifies the cumulative normal density function.

\[
d_1 = \frac{\ln \left( \frac{X_1}{X_2} \right) + \left( \frac{\sigma_1^2 + \sigma_2^2}{2} \right) t}{\sigma_{x_1, x_2} \sqrt{t}}
\]

\[
d_2 = d_1 - \sigma_{x_1, x_2} \sigma_{x_1} \sigma_{x_2}
\]

The following arguments apply to the preceding equation:

$t$
specifies the time to expiration.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.
\( \sigma_{x_1} \)

specifies the volatility of the first asset.

\( \sigma_{x_2} \)

specifies the volatility of the second asset.

\( \rho_{x_1, x_2} \)

specifies the correlation between the first and second assets.

For the special case of \( t=0 \), the following equation is true:

\[
\text{CALL} = \max\{X_1 - X_2, 0\}
\]

**Note:** This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see “Using Pricing Functions” in *SAS Functions and CALL Routines: Reference.*

**Comparisons**

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

**Example**

The following statements illustrate the MARGRCLPRC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{a=margrclprc(15, .5, 13, .06, .05, 1);} )</td>
<td>2</td>
</tr>
<tr>
<td>\text{put a;}</td>
<td></td>
</tr>
<tr>
<td>( \text{b=margrclprc(2, .25, 1, .3, .2, 1);} )</td>
<td>1</td>
</tr>
<tr>
<td>\text{put b;}</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “MARGRPTPRC Function” on page 618

**MARGRPTPRC Function**

Calculates put prices for European options on stocks, based on the Margrabe model.

**Category:** Financial

**Returned data type:** DOUBLE
Syntax

MARGRPTPRC($X_1$, $t$, $X_2$, $\sigma_1$, $\sigma_2$, $\rho_{12}$)

Arguments

$X_1$
- is a nonmissing, positive value that specifies the price of the first asset.
- Requirement: Specify $X_1$ and $X_2$ in the same units.
- Data type: DOUBLE

$t$
- is a nonmissing value that specifies the time to expiration, in years.
- Data type: DOUBLE

$X_2$
- is a nonmissing, positive value that specifies the price of the second asset.
- Requirement: Specify $X_2$ and $X_1$ in the same units.
- Data type: DOUBLE

$\sigma_1$
- is a nonmissing, positive fraction that specifies the volatility of the first asset.
- Data type: DOUBLE

$\sigma_2$
- is a nonmissing, positive fraction that specifies the volatility of the second asset.
- Data type: DOUBLE

$\rho_{12}$
- specifies the correlation between the first and second assets, $\rho_{X_1X_2}$.
- Range: between $-1$ and $1$
- Data type: DOUBLE

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

\[
PUT = X_2 N(pd_1) - X_1 N(pd_2)
\]

Arguments

$X_1$
- specifies the price of the first asset.

$X_2$
- specifies the price of the second asset.
\( N \) specifies the cumulative normal density function.

\[
pd_1 = \frac{\ln \left( \frac{N_1}{N_2} \right) + \left( \frac{\sigma^2 t}{2} \right)}{\sigma \sqrt{t}}
\]

\[
pd_2 = pd_1 - \sigma \sqrt{t}
\]

\[
\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2 \rho_{x_1, x_2} \sigma_{x_1} \sigma_{x_2}
\]

The following arguments apply to the preceding equation:

\( t \)

is a nonmissing value that specifies the time to expiration, in years.

\( \sigma_{x_1}^2 \)

specifies the variance of the first asset.

\( \sigma_{x_2}^2 \)

specifies the variance of the second asset.

\( \sigma_{x_1} \)

specifies the volatility of the first asset.

\( \sigma_{x_2} \)

specifies the volatility of the second asset.

\( \rho_{x_1, x_2} \)

specifies the correlation between the first and second assets.

To view the corresponding CALL relationship, see the “MARGRCLPRC Function” on page 616.

For the special case of \( t=0 \), the following equation is true:

\[
PUT = \max(X_2 - X_1, 0)
\]

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see “Using Pricing Functions” in SAS Functions and CALL Routines: Reference.

Comparisons

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRPTPRC function:
Variables and Results

\[
a = \text{margrptprc}(2, .25, 3, .06, .2, 1);
\]
\[
\text{put } a;
\]
\[
1.00000000009729
\]

\[
b = \text{margrptprc}(3, .25, 4, .05, .3, 1);
\]
\[
\text{put } b;
\]
\[
1.00157624907711
\]

See Also

Functions:
- “MARGRCLPRC Function” on page 616

MAX Function

Returns the largest value from a list of arguments.

**Syntax**

\[
\text{MAX}(expression-1, expression-2\ [, \ldots expression-n])
\]

**Arguments**

- **expression**
  - is any valid expression that evaluates to a numeric value.
  - **Requirement**: At least two arguments are required.
  - **Data type**: BIGINT, DECIMAL, DOUBLE, NUMERIC

**Details**

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

**Comparisons**

The MAX function returns the largest value from a list of arguments. The MAX operator (\(<\>)\) returns the largest of two operands.

The MAX function returns a null or missing value only if all arguments are null or missing. The MAX operator (\(<\>)\) returns a null or missing value only if both operands
are null or missing. In this case, it returns the value of the operand that is higher in the sort order for null or missing values.

Example

The following statements illustrate the MAX function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=max(8,3);</td>
<td>8</td>
</tr>
<tr>
<td>x=max(2, 6, .);</td>
<td>6</td>
</tr>
<tr>
<td>x=max(2, -3, 1, -1);</td>
<td>2</td>
</tr>
<tr>
<td>x=max(3, ., -3);</td>
<td>3</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “MIN Function” on page 627

MD5 Function

Returns the result of the message digest of a specified string in binary format.

Category: Character
Restriction: Use CHAR character arguments only. MD5 does not accept NCHAR character arguments.

Returned data type: BINARY

Syntax

\[ \text{MD5(string)} \]

Arguments

string

specifies a character constant, variable, or expression.

Data type BINARY, CHAR

Tips

Enclose a literal string of characters in single quotation marks.

For scalar character variable arguments, the initial character set encoding that is specified in the DECLARE statement is used to transcode the variable before it is passed to the MD5 function. For binary arguments, the
binary value is converted to a character string and the session encoding is used to transcode the value it is passed to the MD5 function.

Details

The Basics
The MD5 function converts a string, based on the MD5 algorithm, into a 128-bit hash value. This hash value is referred to as a message digest (digital signature), and it is nearly unique for each string that is passed to the function.

The MD5 function does not format its own output. Use the $BINARYw. or the $HEXw. formats to view readable results.

The Message Digest Algorithm
A message digest results from manipulating and compacting an arbitrarily long stream of binary data. An ideal message digest algorithm never generates the same result for two different sets of input. However, generating such a unique result would require a message digest as long as the input itself. Therefore, MD5 generates a message digest of modest size (16 bytes), created with an algorithm that is designed to make a nearly unique result.

Using the MD5 Function
You can use the MD5 function to track changes in your tables. The MD5 function can generate a digest of a set of column values in a record in a table. This digest could be treated as the signature of the record, and be used to keep track of changes that are made to the record. If the digest from the new record matches the existing digest of a record in a table, then the two records are the same. If the digest is different, then a column value in the record has changed. The new changed record could then be added to the table along with a new surrogate key because it represents a change to an existing keyed value.

The MD5 function can be useful when developing shell scripts or Perl programs for software installation, for file comparison, and for detection of file corruption and tampering.

You can also use the MD5 function to create a unique identifier for observations to be used as the key of a hash package. For more information, see “Using the Hash Package” on page 145.

Example
The following is an example of how to generate results that are returned by the MD5 function.

```plaintext
data _null_;  
  method init();  
dcl char(16) y z having format $hex32.;  
y = md5('abc');  
z = md5('access method');  
put y= ;  
put z= ;  
end;  
enddata;  
run;
```
SAS writes the following results to the log:

\[
y=900150983CD24FB0D6963F7D28E17F72 \\
z=53128C19421A8E6C7F6436D06A026537
\]

---

### MDY Function

Returns a SAS date value from month, day, and year values.

**Category:** Date and Time  
**Returned data type:** DOUBLE

---

#### Syntax

\[
\text{MDY}(\text{month}, \text{day}, \text{year})
\]

#### Arguments

- **month**
  - specifies a numeric expression that represents an integer from 1 through 12.
  - Data type: DOUBLE  
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **day**
  - specifies a numeric expression that represents an integer from 1 through 31.
  - Data type: DOUBLE  
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **year**
  - specifies a numeric expression that represents a two-digit or four-digit year. The YEARCUTOFF= system option defines the year value for two-digit dates.
  - Data type: DOUBLE  
  - See: Chapter 13, “DS2 Expressions,” on page 93

---

#### Example

The following statements illustrate the MDY function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>mn=8; dy=27; yr=12;</td>
<td>08/27/2012</td>
</tr>
<tr>
<td>birthday= mdy(mn,dy,yr);</td>
<td></td>
</tr>
<tr>
<td>put birthday;</td>
<td></td>
</tr>
</tbody>
</table>
mn=7; dy=11; yr=12;
anniversary= mdy(mn,dy,yr);
put anniversary;

11JUL2012

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:
- “DAY Function” on page 469
- “MONTH Function” on page 635
- “YEAR Function” on page 845

MEAN Function
Returns the arithmetic mean (average) of the non-null or nonmissing arguments.

Category: Descriptive Statistics
Returned data type: DOUBLE

Syntax
MEAN(expression-1[, ...expression-n])

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

Requirement
At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type
DOUBLE

See
Chapter 13, “DS2 Expressions,” on page 93

Comparisons
The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, whereas the MEAN function returns the arithmetic mean (average).

Example
The following statements illustrate the MEAN function:
### MEDIAN Function

Returns the median value.

- **Category:** Descriptive Statistics
- **Returned data type:** DOUBLE

#### Syntax

\[
\text{MEDIAN(expression-1[, \ldots, expression-n])}
\]

#### Arguments

- **expression**
  - Specifies any valid expression that evaluates to a numeric value.
  - **Data type:** DOUBLE

#### See

Chapter 13, “DS2 Expressions,” on page 93

#### Details

The MEDIAN function returns the median of the nonmissing or nonnull values. If all arguments have missing or null values, the result is a missing or null value.

*Note:* The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures. For more information, see SAS Elementary Statistics Procedures.

#### Comparisons

The MEDIAN function returns the median of nonmissing or nonnull values, whereas the MEAN function returns the arithmetic mean (average).
Example

The following statements illustrate the MEDIAN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = median(2, 4, 1, 3);</td>
<td>2.5</td>
</tr>
<tr>
<td>y = median(5, 8, 0, 3, 4);</td>
<td>4</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “MEAN Function” on page 625

MIN Function

Returns the smallest value.

Category: Descriptive Statistics

Returned data type: BIGINT, DECIMAL, DOUBLE, NUMERIC

Syntax

\[ \text{MIN(expression-1, expression-2 [,...expression-n])} \]

Arguments

expression specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type BIGINT, DECIMAL, DOUBLE, NUMERIC

See Chapter 13, “DS2 Expressions,” on page 93

Details

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Comparisons

The MIN function returns the smallest value from a list of values. The MIN operator (\(\leq\)) returns the smallest value of two operands.
The MIN function returns a null or missing value only if all arguments are null or missing. The MIN operator returns a null or missing value only if either operand is null or missing. In this case, it returns the value of the operand that is lower in the sort order for null or missing values.

Example

The following statements illustrate the MIN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=min(2,.,6);</td>
<td>2</td>
</tr>
<tr>
<td>a=min(2,-3,1,-1);</td>
<td>-3</td>
</tr>
</tbody>
</table>

MINUTE Function

Returns the minute from a SAS time or datetime value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

MINUTE(time | datetime)

Arguments

time

specifies any valid expression that represents a SAS time value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

datetime

specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The MINUTE function returns an integer that represents a specific minute of the hour. MINUTE always returns a positive number in the range of 0 through 59. Null or missing values are ignored.
Example

The following statement illustrates the MINUTE function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=minute(time());</td>
<td>19</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

- “HOUR Function” on page 532
- “SECOND Function” on page 775

MISSING Function

Returns a number that indicates whether the argument contains a missing value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Syntax

MISSING(expression)

Arguments

expression specifies any valid expression that evaluates to a value.

Data type  All data types

Note

If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.

See

Chapter 13, “DS2 Expressions,” on page 93

Details

- The MISSING function checks if a value is a null or missing value and returns a numeric result. If the argument does not contain a missing value, SAS returns a value of 0. If the argument contains a missing value, SAS returns a value of 1.
- In SAS mode, a blank-filled character value is defined to be the SAS missing value. In ANSI mode, a blank-filled character value is defined as nonmissing and non-null.
- In SAS mode, a DOUBLE value could be a SAS missing value (., .A through .Z). The other numeric types do not support SAS missing values.
• The MISSING function returns a 1 if a package instance does not exist. That is, the package variable is a missing package reference. The MISSING function returns a 0 if the package variable references a package instance.

Comparisons

The MISSING function can have only one argument. The NMISS function requires numeric arguments and returns the number of missing values in the list of arguments.

Note: Missing values and null values are treated differently in SAS mode versus ANSI mode. Missing and null values might be converted dependent on mode.

Examples

Example 1: Using the MISSING Function

The following example illustrates the MISSING function.

data _null_; method init();
   dcl int a[3];
   dcl double i;
   a[1]=2;
   a[2]=4;
   do i = 1 to 3;
      if missing(a[i]) then put 'Missing';
      else put 'Not Missing';
   end;
end;
enddata;
run;

The following lines are written to the SAS log.

Not Missing
Not Missing
Missing

Example 2: MISSING Function with SAS Mode and ANSI Mode

This example illustrates how a DS2 program with a MISSING function can return different results based on mode.

data _null_; method init();
   declare char(1) a[3];
   declare double b[3];
   declare int c[3];
   declare double i;

   a := ('a', '', NULL);
   b := (1, ., NULL);
   c := (1, NULL, NULL);

   do i = 1 to 3;
      if (missing(a[i])) then put a[i]= 'missing';
      else put a[i]= 'not missing';
   end;
if (missing(b[i])) then put b[i]= 'missing';
    else put b[i]= 'not missing';

    if (missing(c[i])) then put c[i]= 'missing';
    else put c[i]= 'not missing';

    end;
    end;
enddata;
run;

In SAS mode, the following lines are written to the SAS log.

```
a[1]=a not missing
b[1]=1 not missing
c[1]=1 not missing
a[2]= missing
b[2]=. missing
c[2]= missing
a[3]= missing
b[3]=. missing
c[3]= missing
```

In ANSI mode, the following lines are written to the SAS log.

```
a[1]=a not missing
b[1]=1 not missing
c[1]=1 not missing
a[2]= not missing
b[2]= missing
c[2]= missing
a[3]= missing
b[3]= missing
c[3]= missing
```

See Also

- Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81

Functions:

- “NMISS Function” on page 641
- “N Function” on page 637
- “NULL Function” on page 670

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

**Category:** Mathematical

**Returned data type:** DOUBLE
**Syntax**

\[
\text{MOD}(\text{dividend-expression}, \text{divisor-expression})
\]

**Arguments**

**dividend-expression**

specifies a dividend that is any valid expression that evaluates to a numeric value.

- Data type: DOUBLE
- See: Chapter 13, “DS2 Expressions,” on page 93

**divisor-expression**

specifies a divisor that is any valid expression that evaluates to a numeric value.

- Restriction: \(\text{divisor-expression} \text{ cannot be } 0\)
- Data type: DOUBLE
- See: Chapter 13, “DS2 Expressions,” on page 93

**Details**

The MOD function returns the remainder from the division of \(\text{dividend-expression}\) by \(\text{divisor-expression}\). When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case, the following occurs:

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

**Comparisons**

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

**Example**

The following statements illustrate the MOD function:
MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Syntax

\[ \text{MODZ}(\text{dividend-expression}, \text{divisor-expression}) \]

Arguments

- **dividend-expression**
  - specifies a dividend that is any valid expression that evaluates to a numeric value.
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93

- **divisor-expression**
  - specifies a divisor that is any valid expression that evaluates to a numeric value.
  - Restriction: \( \text{divisor-expression} \) cannot be 0
  - Data type: DOUBLE
  - See: Chapter 13, “DS2 Expressions,” on page 93
Details
The MODZ function returns the remainder from the division of \(\text{dividend-expression}\) by \(\text{divisor-expression}\). When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representation.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

Comparisons
Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example
The following statements illustrate the differences between the MOD and MODZ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = \text{mod}(10, 3));</td>
<td>1</td>
</tr>
<tr>
<td>(b = \text{modz}(10, 3));</td>
<td>1</td>
</tr>
<tr>
<td>(a = \text{mod}(.35, -.1));</td>
<td>0.05</td>
</tr>
<tr>
<td>(b = \text{modz}(.35, -.1));</td>
<td>0.4999999999999</td>
</tr>
<tr>
<td>(a = \text{mod}(17, 3));</td>
<td>2</td>
</tr>
<tr>
<td>(b = \text{modz}(17, 3));</td>
<td>2</td>
</tr>
<tr>
<td>(a = \text{mod}(.3, -.9));</td>
<td>0.3</td>
</tr>
<tr>
<td>(b = \text{modz}(.3, -.9));</td>
<td>0.3</td>
</tr>
</tbody>
</table>

See Also
Functions:

- “INT Function” on page 541
- “INTZ Function” on page 581
- “MOD Function” on page 631
MONTH Function

Returns a number that represents the month from a SAS date value.

Category: Date and Time
Returned data type: DOUBLE

Syntax

MONTH(date)

Arguments

date

specifies any valid expression that represents a SAS date value.

Range 1–12
Data type DOUBLE
See Chapter 13, “DS2 Expressions,” on page 93

Example

The following statement illustrates the MONTH function when the month is November:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=month(date());</td>
<td>11</td>
</tr>
</tbody>
</table>

See Also

• Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

• “DAY Function” on page 469
• “YEAR Function” on page 845

MORT Function

Returns amortization parameters.

Category: Financial
Returned data type: DOUBLE
Syntax

MORT\((a, p, r, n)\)

Arguments

\(a\)

specifies any valid expression that evaluates to the initial amount.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

\(p\)

specifies any valid expression that evaluates to the periodic payment.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

\(r\)

specifies any valid expression that evaluates to the periodic interest rate that is expressed as a fraction.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

\(n\)

specifies any valid expression that evaluates to the number of compounding periods.

Range \(n \geq 0\)

Data type INTEGER

See Chapter 13, “DS2 Expressions,” on page 93

Details

Calculating Results

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by the following equation:

\[
p = \frac{ar(1 + r)^n}{(1 + r)^n - 1}
\]

One missing argument must be provided. The value is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Restrictions in Calculating Results

The MORT function returns an invalid argument note to the SAS log and sets _ERROR_ to 1 if one of the following argument combinations is true:

- rate < –1 or n < 0
- principal <= 0 or payment <= 0 or n <= 0
• principal <= 0 or payment <= 0 or rate <= –1
• principal * rate > payment
• principal > payment * n

Example

In the following example, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10% compounded monthly.

data test (overwrite=yes);
  dcl double payment;
  method run();
  payment=mort(50000, . , .10/12, 30*12);
  put payment;
end;
enddata;
run;

The value that is returned is 438.79 (rounded). The second argument is set to missing, which indicates that the periodic payment is to be calculated. The 10% nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted to 360 months.

---

**N Function**

Returns the number of non-null or nonmissing numeric values.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DECIMAL, DOUBLE, NUMERIC</td>
</tr>
</tbody>
</table>

**Syntax**

\[ N(expression [, ...expression]) \]

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>At least one argument is required.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DECIMAL, DOUBLE, NUMERIC</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

**Details**

Null values are converted to missing values and are counted as missing values.
Comparisons

The N function counts non-null and nonmissing values, whereas the NMISS function counts missing values. The N function requires numeric arguments.

Example

The following statements illustrate the N function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=n(1,0,.,2,5,.);</td>
<td>4</td>
</tr>
<tr>
<td>a=n(1,2);</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also

Functions:

• “NMISS Function” on page 641

NDIMS Function

Returns the number of dimensions in an array.

Category: Array

Returned data type: INT

Syntax

NDIMS(array-name)

Arguments

array-name

specifies the name of a temporary or a variable array.

Details

The NDIMS function returns the number of dimensions of a multidimensional array, or returns 1 for a one-dimensional array.

Comparisons

• DIM returns the number of elements in an array dimension.
• HBOUND returns the value of the upper bound of an array dimension.
• LBOUND returns the value of the lower bound of an array dimension.
• NDIMS returns the number of dimensions in an array.
Example: Using Array Functions

The following example shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```sas
data _null_
method init();
  declare char(15) a1[4];
  declare double   a2[2,3,4] sum;

  a1 := ('red' 'yellow' 'green' 'blue');
  a2 := (24*2.0);

  do i = 1 to dim(a1);
    put a1[i];
  end;

  numelems = 0;
  do i = 1 to ndims(a2);
    numelems = numelems + dim(a2, i);
  end;

  sum = 0;
  do i = lbound(a2, 1) to hbound(a2, 1);
    do j = lbound(a2, 2) to hbound(a2, 2);
      do k = lbound(a2, 3) to hbound(a2, 3);
        sum = sum + a2[i,j,k];
      end;
    end;
  end;

  put sum=;
end;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

See Also

Functions:
- “DIM Function” on page 478
- “HBOUND Function” on page 526
- “LBOUND Function” on page 600
NETPV Function

Returns the net present value as a percent.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

\[ \text{NETPV}(r, freq, c0, c1, ..., cn) \]

**Arguments**

- **\( r \)** is numeric, the interest rate over a specified base period of time expressed as a fraction.
  - Range: \( r \geq 0 \)
  - Data type: DOUBLE

- **\( freq \)** is numeric, the number of payments during the base period of time that is specified with the rate \( r \).
  - Range: \( freq > 0 \)
  - Data type: DOUBLE

- **\( c0, c1, ..., cn \)** are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ..., \( n \). These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The \( c0 \) argument and the \( c1 \) argument are required.
  - Data type: DOUBLE

**Details**

The \( \text{NETPV} \) function returns the net present value at time 0 for the set of cash payments \( c0, c1, ..., cn \), with a rate \( r \) over a specified base period of time. The argument \( freq > 0 \) describes the number of payments that occur over the specified base period of time.

The net present value is given by the equation:

\[ \text{NETPV}(r, freq, c0, c1, ..., cn) = \sum_{i=0}^{n} c_i x^i \]

The following relationship applies to the preceding equation:
$x = \begin{cases} 
\frac{1}{(1+r)^{1/freq}} & freq > 0 \\
\varepsilon^{-r} & freq = 0 
\end{cases}$

Missing values in the payments are treated as 0 values. When $freq > 0$, the rate $r$ is the effective rate over the specified base period. To compute with a quarterly rate (the base period is three months) of 4% with monthly cash payments, set $freq$ to 3 and set $r$ to .04.

If $freq$ is 0, continuous discounting is assumed. The base period is the time interval between two consecutive payments, and the rate $r$ is a nominal rate.

To compute with a nominal annual interest rate of 11% discounted continuously with monthly payments, set $freq$ to 0 and set $r$ to .11/12.

**Example**

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding 6 years and an annual discount rate of 10%, the net present value of the investment can be expressed as follows:

```sas
data _null_
  method run();
  value=netpv(.10,.5,-500,200,300,400);
  put value;
end;
enddata;
run;
```

The value that is returned is 95,982,864,829,379.

**See Also**

**Functions:**

- “NPV Function” on page 669

---

**NMISS Function**

Returns the number of null and SAS missing numeric values.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

**Syntax**

`NMISS(expression [, …expression])`

**Arguments**

- `expression`
  - specifies any valid expression that evaluates to a numeric value.

  **Requirement** At least one argument is required.
Details
Null values are converted to SAS missing values and are counted as missing values.

Comparisons
The NMISS function returns the number of null or SAS missing values, whereas the N function returns the number of non-null and nonmissing values. NMISS requires numeric values and works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

Example
The following statements illustrate the NMISS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=nmiss(1,0,,2,5,.);</td>
<td>2</td>
</tr>
<tr>
<td>a=nmiss(1,0);</td>
<td>0</td>
</tr>
</tbody>
</table>

See Also
- Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81

Functions:
- “MISSING Function” on page 629
- “N Function” on page 637
- “NULL Function” on page 670

NOMRATE Function
Returns the nominal annual interest rate.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Financial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
NOMRATE(compounding-interval, rate)
Arguments

**compounding-interval**

is a SAS interval. This value represents how often the returned value is compounded.

Data type: CHAR

**rate**

is numeric. *Rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

Data type: DOUBLE

Details

The NOMRATE function returns the nominal annual interest rate. NOMRATE computes the nominal annual interest rate that corresponds to an effective annual interest rate.

The following details apply to the NOMRATE function:

- The values for rates must be at least –99.
- In considering an effective interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by NOMRATE equals $\log_e(1+[rate/100])$.

If *compounding-interval* is not 'CONTINUOUS', and $m$ intervals occur in a year, the value that is returned by NOMRATE equals the following:

$$m \left(1 + \frac{rate}{100} \right)^{\frac{1}{m}} - 1$$

- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then $m=365$.

Example

- If an effective rate is 10% when compounded monthly, the corresponding nominal rate can be expressed as follows:
  
  ```
  effective_rate1 = NOMRATE('MONTH', 10);
  ```

- If an effective rate is 10% when compounded quarterly, the corresponding nominal rate can be expressed as follows:

  ```
  effective_rate2 = NOMRATE('QUARTER', 10);
  ```
NOTALNUM Function

Searches a character string for a non-alphanumeric character, and returns the first position at which the character is found.

**Category:** Character  
**Returned data type:** DOUBLE

**Syntax**

```
NOTALNUM('expression'[, start])
```

**Arguments**

- **expression**
  - Specifies any valid expression that evaluates to a character string.  
  - **Data type:** CHAR, NCHAR  
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **start**
  - Is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.  
  - **Data type:** INTEGER

**Details**

The results of the NOTALNUM function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

NOTALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of `start` is greater than the length of the string.
• The value of \( start = 0 \).

**Comparisons**

The NOTALNUM function searches a character string for a non-alphanumeric character. The ANYALNUM function searches a character string for an alphanumeric character.

**Example**

The following example uses the NOTALNUM function to search a string from left to right for non-alphanumeric characters.

```sas
data _null_;  
dcl nchar(16) string c;  
dcl double j i;  
method run();  
    string='Next = Last + 1;';  
    j=0;  
    do until(j=0);  
        j=notalnum(string, j+1);  
        if j=0 then put 'The end';  
        else do;  
            c=substr(string, j, 1);  
            put j= c=;  
        end;  
    end;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
j=5 c=  
j=6 c==  
j=7 c=  
j=12 c=  
j=13 c==  
j=14 c=  
j=16 c=;  
The end
```

**See Also**

**Functions:**

• “ANYALNUM Function” on page 359

---

**NOTALPHA Function**

Searches a character string for a nonalphabetic character, and returns the first position at which the character is found.

- **Category:** Character
- **Returned data type:** DOUBLE
Syntax

`NOTALPHA(expression[, start])`

Arguments

`expression`

specifies any valid expression that evaluates to a character string.

Data type: CHAR, NCHAR

See: Chapter 13, “DS2 Expressions,” on page 93

`start`

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type: INTEGER

Details

The results of the NOTALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns the position in the string of that character. If no such character is found, NOTALPHA returns a value of 0.

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of `start` is greater than the length of the string.
- The value of `start` = 0.

Comparisons

The NOTALPHA function searches a character string for a nonalphabetic character. The ANYALPHA function searches a character string for an alphabetic character.
Examples

**Example 1: Searching a String for Nonalphabetic Characters**

The following example uses the NOTALPHA function to search a string from left to right for nonalphabetic characters.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=notalpha(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;  
    end;  
  end;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
j=5  c=  
j=6  c==  
j=7  c=  
j=8  c=_  
j=10  c=_  
j=11  c=  
j=12  c=+  
j=13  c=  
j=14  c=1  
j=15  c=2  
j=17  c=3  
j=18  c=;  
The end
```

**Example 2: Identifying Control Characters By Using the NOTALPHA Function**

You can execute the following program to show the control characters that are identified by the NOTALPHA function.

```sas
data test;  
dcl nchar(3) byte1 hex1;  
dcl double dec notalpha1;  
method run();  
  do dec=0 to 255;  
    byte1=byte(dec);  
    hex1=put(dec,hex2.);  
    notalpha1=notalpha(byte1);  
    output;  
  end;  
end;  
enddata;
```
See Also

Functions:

• “ANYALPHA Function” on page 362

NOTCNTRL Function

Searches a character string for a character that is not a control character, and returns the first position at which that character is found.

**Syntax**

\[
\text{NOTCNTRL('expression'[, start])}
\]

**Arguments**

*expression*

specifies any valid expression that evaluates to a character string.

Data type: CHAR, NCHAR

See: Chapter 13, “DS2 Expressions,” on page 93

*start*

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type: INTEGER

**Details**

The results of the NOTCNTRL function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTCNTRL function searches a string for the first occurrence of a character that is not a control character. If such a character is found, NOTCNTRL returns the position in the string of that character. If no such character is found, NOTCNTRL returns a value of 0.

If you use only one argument, NOTCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

* If the value of *start* is positive, the search proceeds to the right.
* If the value of *start* is negative, the search proceeds to the left.
• If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTCNTRL returns a value of zero when one of the following is true:
• The character that you are searching for is not found.
• The value of start is greater than the length of the string.
• The value of start = 0.

Comparisons
The NOTCNTRL function searches a character string for a character that is not a control character. The ANYCNTRL function searches a character string for a control character.

Example
You can execute the following program to show the control characters that are identified by the NOTCNTRL function.

data test (overwrite=yes);
  dcl double dec notcntrl1;
  dcl char byte1 hex1;
  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec, hex2.);
      notcntrl1=notcntrl(byte1);
      output;
    end;
  end;
enddata;
run;

See Also
Functions:
• “ANYCNTRL Function” on page 364

NOTDIGIT Function
Searches a character string for any character that is not a digit, and returns the first position at which that character is found.

Category:            Character
Returned data type:  DOUBLE

Syntax
NOTDIGIT('expression'[, start])
Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type  CHAR, NCHAR

See  Chapter 13, “DS2 Expressions,” on page 93

start
is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type  INTEGER

Details

The results of the NOTDIGIT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

• If the value of start is positive, the search proceeds to the right.
• If the value of start is negative, the search proceeds to the left.
• If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when one of the following is true:

• The character that you are searching for is not found.
• The value of start is greater than the length of the string.
• The value of start = 0.

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Example

The following example uses the NOTDIGIT function to search for a character that is not a digit.

```sas
data _null_;
  dcl nchar(18) string c;
  dcl double j i;
  method run();
    string='Next = _n_ + 12E3;';
    j=0;
```
do until(j=0);
    j=notdigit(string, j+1);
    if j=0 then put 'The end';
    else do;
        c=substr(string, j, 1);
        put j= c=;
    end;
end;
enddata;
run;

SAS writes the following output to the log:

j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=16 c=E
j=18 c=;
The end

See Also

Functions:

• “ANYDIGIT Function” on page 366

NOTFIRST Function

Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character
Returned data type: DOUBLE

Syntax

NOTFIRST(’expression’[, start])

Arguments

expression
    specifies any valid expression that evaluates to a character string.
**Data type**
CHAR, NCHAR

**See**
Chapter 13, “DS2 Expressions,” on page 93

**start**
is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

**Data type**
INTEGER

**Details**
The NOTFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of `start` is greater than the length of the string.
- The value of `start` = 0.

**Comparisons**
The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

**Example**
The following example uses the NOTFIRST function to search a string for any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

```sas
data _null_
  dcl nchar(18) string c;
  dcl double j i;
  method run();
  string='Next = _n_ + 12E3;';
  j=0;
```
do until(j=0);
    j=notfirst(string, j+1);
    if j=0 then put 'The end';
    else do;
        c=substr(string, j, 1);
        put j= c=;
    end;
    end;
enddata;
run;

SAS writes the following output to the log:

<table>
<thead>
<tr>
<th>j</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>=</td>
</tr>
<tr>
<td>6</td>
<td>==</td>
</tr>
<tr>
<td>7</td>
<td>=</td>
</tr>
<tr>
<td>11</td>
<td>=</td>
</tr>
<tr>
<td>12</td>
<td>+=</td>
</tr>
<tr>
<td>13</td>
<td>=</td>
</tr>
<tr>
<td>14</td>
<td>=1</td>
</tr>
<tr>
<td>15</td>
<td>=2</td>
</tr>
<tr>
<td>17</td>
<td>=3</td>
</tr>
<tr>
<td>18</td>
<td>=;</td>
</tr>
</tbody>
</table>

The end

See Also

Functions:

• “ANYFIRST Function” on page 368

NOTGRAPH Function

Searches a character string for a non-graphical character, and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

\[
\text{NOTGRAPH('expression'[, start])}
\]

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93
**start**

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

**Data type**  
INTEGER

**Details**

The results of the NOTGRAPH function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTGRAPH function searches a string for the first occurrence of a non-graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, NOTGRAPH returns the position in the string of that character. If no such character is found, NOTGRAPH returns a value of 0.

If you use only one argument, NOTGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, **start**, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of **start** is positive, the search proceeds to the right.
- If the value of **start** is negative, the search proceeds to the left.
- If the value of **start** is less than the negative length of the string, the search begins at the end of the string.

NOTGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of **start** is greater than the length of the string.
- The value of **start** = 0.

**Comparisons**

The NOTGRAPH function searches a character string for a non-graphical character. The ANYGRAPH function searches a character string for a graphical character.

**Examples**

**Example 1: Searching a String for Non-Graphical Characters**

The following example uses the NOTGRAPH function to search a string for a non-graphical character.

```sas
data _null_;  
dcl nchar(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=notgraph(string, j+1);  
    if j=0 then put 'The end';  
    else do;  
      c=substr(string, j, 1);  
      put j= c=;
```

Example 2: Identifying Control Characters By Using the NOTGRAPH Function

You can execute the following program to show the control characters that are identified by the NOTGRAPH function.

```sas
data test (overwrite=yes);
  dcl nchar byte1 hex1;
  dcl double dec notgraph1;

  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec,hex2.);
      notgraph1=notgraph(byte1);
      output;
    end;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
The end
```

See Also

Functions:

- “ANYGRAPH Function” on page 370

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.

**Category:** Character  
**Returned data type:** DOUBLE

**Syntax**

```
NOTLOWER('expression'[, start])
```
Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type \text{CHAR, NCHAR}

See \text{Chapter 13, “DS2 Expressions,” on page 93}

start

is a numeric constant, variable, or expression that specifies the position at which the
search should start and the direction in which to search.

Data type \text{INTEGER}

Details

The results of the NOTLOWER function depend directly on the translation table that is
in effect (see “\text{TRANTAB= System Option}” in \text{SAS National Language Support (NLS): Reference Guide}) and indirectly on the \text{ENCODING} and the \text{LOCALE} system options.

The NOTLOWER function searches a string for the first occurrence of any character that
is not a lowercase letter. If such a character is found, NOTLOWER returns the position
in the string of that character. If no such character is found, NOTLOWER returns a value
of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the
string. If you use two arguments, the absolute value of the second argument, start,
specifies the position at which to begin the search. The direction in which to search is
determined in the following way:

\begin{itemize}
  \item If the value of \text{start} is positive, the search proceeds to the right.
  \item If the value of \text{start} is negative, the search proceeds to the left.
  \item If the value of \text{start} is less than the negative length of the string, the search begins at
        the end of the string.
\end{itemize}

NOTLOWER returns a value of zero when one of the following is true:

\begin{itemize}
  \item The character that you are searching for is not found.
  \item The value of \text{start} is greater than the length of the string.
  \item The value of \text{start} = 0.
\end{itemize}

Comparisons

The NOTLOWER function searches a character string for a character that is not a
lowercase letter. The ANYLOWER function searches a character string for a lowercase
letter.

Example

The following example uses the NOTLOWER function to search a string for any
character that is not a lowercase letter.

\begin{verbatim}
data _null_
  dcl nchar(18) string c;
  dcl double j i;
  method run()
  \end{verbatim}
string='Next = _n_ + 12E3;';
j=0;
do until(j=0);
   j=notlower(string, j+1);
   if j=0 then put 'The end';
   else do;
      c=substr(string, j, 1);
      put j= c=;
   end;
end;
enddata;
run;

SAS writes the following output to the log:

j=1 c=N
j=5 c=
j=6 c=*
j=7 c=
j=8 c=*
j=9 c=_
j=10 c=_
j=11 c=
j=12 c=*
j=13 c=
j=14 c=*
j=15 c=*
j=16 c=E
j=17 c=3
j=18 c=;
The end

See Also

Functions:

- “ANYLOWER Function” on page 372

NOTNAME Function

Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character

Returned data type: DOUBLE

Syntax

NOTNAME('expression'[, start])

Arguments

expression

specifies any valid expression that evaluates to a character string.
**Data type**  
CHAR, NCHAR

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**start**  
is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

**Data type**  
INTEGER

**Details**

The NOTNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \( \text{start} \), specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \( \text{start} \) is positive, the search proceeds to the right.
- If the value of \( \text{start} \) is negative, the search proceeds to the left.
- If the value of \( \text{start} \) is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of \( \text{start} \) is greater than the length of the string.
- The value of \( \text{start} = 0 \).

**Comparisons**

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

**Example**

The following example uses the NOTNAME function to search a string for any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

```sas
data _null_
  dcl nchar(18) string c;
  dcl double j i;
  method run()
  
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
```
j=notname(string, j+1);
if j=0 then put 'The end';
else do;
c=substr(string, j, 1);
put j= c=;
end;
end;
enddata;
run;

SAS writes the following output to the log:

```
j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c==
j=13 c==
j=18 c=;
The end
```

See Also

Functions:

- “ANYNAME Function” on page 374

---

**NOTPRINT Function**

Searches a character string for a nonprintable character, and returns the first position at which that character is found.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

```
NOTPRINT('expression'[, start])
```

**Arguments**

- **expression** specifies any valid expression that evaluates to a character string.

  **Data type**: CHAR, NCHAR

  **See**: Chapter 13, “DS2 Expressions,” on page 93

- **start** is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.
Details

The results of the NOTPRINT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTPRINT function searches a string for the first occurrence of a non-printable character. If such a character is found, NOTPRINT returns the position in the string of that character. If no such character is found, NOTPRINT returns a value of 0.

If you use only one argument, NOTPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

• If the value of start is positive, the search proceeds to the right.
• If the value of start is negative, the search proceeds to the left.
• If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTPRINT returns a value of zero when one of the following is true:

• The character that you are searching for is not found.
• The value of start is greater than the length of the string.
• The value of start = 0.

Comparisons

The NOTPRINT function searches a character string for a non-printable character. The ANYPRINT function searches a character string for a printable character.

Example

You can execute the following program to show the control characters that are identified by the NOTPRINT function.

data test (overwrite=yes);
  dcl double dec notprint1;
  dcl nchar byte1 hex1;
  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec, hex2.);
      notprint1=notprint(byte1);
      output;
    end;
  end;
enddata;
run;

See Also

Functions:
NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.

Category: Character

Returned data type: DOUBLE

Syntax

NOTPUNCT('expression'[, start])

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

start

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type INTEGER

Details

The results of the NOTPUNCT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in SAS National Language Support (NLS): Reference Guide) and indirectly on the ENCODING and the LOCALE system options.

The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
The value of \textit{start} is greater than the length of the string.

The value of \textit{start} = 0.

**Comparisons**

The NOTPUNCT function searches a character string for a character that is not a punctuation character. The ANYPUNCT function searches a character string for a punctuation character.

**Examples**

**Example 1: Searching a String for Characters That Are Not Punctuation Characters**

The following example uses the NOTPUNCT function to search a string for characters that are not punctuation characters.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='\texttt{Next = \_n\_ + 12E3;}';  
j=0;  
do until(j=0);  
j=notpunct(string, j+1);  
  if j=0 then put 'The end';  
  else do;  
    c=substr(string, j, 1);  
    put j= c=;  
  end;  
end;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
j=1 c=N  
j=2 c=e  
j=3 c=x  
j=4 c=t  
j=5 c=  
j=6 c==  
j=7 c=  
j=9 c=n  
j=11 c=  
j=12 c=+  
j=13 c=  
j=14 c=1  
j=15 c=2  
j=16 c=E  
j=17 c=3  
The end
```
Example 2: Identifying Control Characters By Using the NOTPUNCT Function
You can execute the following program to show the control characters that are identified by the NOTPUNCT function.

```sas
data test;
  dcl nchar(3) byte1 hex1;
  dcl double dec notpunct1;

  method run();
    do dec=0 to 255;
      byte1=byte(dec);
      hex1=put(dec,hex2.);
      notpunct1=notpunct(byte1);
      output;
    end;
  end;
enddata;
run;
quit;
proc print data=test;
run;
```

See Also

Functions:
- “ANYPUNCT Function” on page 378

NOTSPACE Function

Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

\[
\text{NOTSPACE}(\text{expression}[\text{, start}])
\]

Arguments

- **expression**
  - specifies any valid expression that evaluates to a character string.

  Data type: CHAR, NCHAR

See

Chapter 13, “DS2 Expressions,” on page 93
**start**

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the NOTSPACE function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

**Comparisons**

The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

**Examples**

**Example 1: Searching a String for a Character That Is Not a Whitespace Character**

The following example uses the NOTSPACE function to search a string for a character that is not a whitespace character.

```sas
data _null_;
dcl char(18) string c;
dcl double j i;
method run();
  string='Next = _n_ + 12E3;';
j=0;
do until(j=0);
```
j=notspace(string, j+1);
if j=0 then put 'The end';
else do;
   c=substr(string, j, 1);
   put j= c=;
end;
end;
enddata;
run;

SAS writes the following output to the log:

j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=_
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end

Example 2: Identifying Control Characters By Using the NOTSPACE Function
You can execute the following program to show the control characters that are identified by the NOTSPACE function.

data test (overwrite=yes);
   dcl nchar(3) byte1 hex1;
   dcl double dec notspace1;
   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         notspace1=notspace(byte1);
         output;
      end;
   end;
enddata;
run;

See Also

Functions:
• “ANYSPACE Function” on page 381
NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

NOTUPPER(expression[, start])

**Arguments**

expression

specifies any valid expression that evaluates to a character string.

Data type: CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

start

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type: INTEGER

**Details**

The results of the NOTUPPER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
• The value of \( \text{start} = 0 \).

**Comparisons**

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

**Example**

The following example uses the NOTUPPER function to search a string for any character that is not an uppercase letter.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
  string='Next = _n_ + 12E3;';  
j=0;  
do until(j=0);  
  j=notupper(string, j+1);  
  if j=0 then put 'The end';  
  else do;  
    c=substr(string, j, 1);  
    put j= c=;  
  end;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
  j=2 c=e
  j=3 c=x
  j=4 c=t
  j=5 c=
  j=6 c==
  j=7 c=
  j=8 c=_
  j=9 c=n
  j=10 c=_
  j=11 c=
  j=12 c==
  j=13 c=
  j=14 c=1
  j=15 c=2
  j=17 c=3
  j=18 c=;
  The end
```

**See Also**

**Functions:**

• “ANYUPPER Function” on page 383
NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

NOTXDIGIT('expression', start)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type: CHAR, NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

start

is a numeric constant, variable, or expression that specifies the position at which the search should start and the direction in which to search.

Data type: INTEGER

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of start is greater than the length of the string.
- The value of start = 0.
Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character. The ANYXDIGIT function searches a character string for a character that is a hexadecimal character.

Example

The following example uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal character.

```sas
data _null_;  
dcl char(18) string c;  
dcl double j i;  
method run();  
   string='Next = _n_ + 12E3;';  
   j=0;  
   do until(j=0);  
      j=notxdigit(string, j+1);  
      if j=0 then put 'The end';  
      else do;  
         c=substr(string, j, 1);  
         put j= c=;  
      end;  
   end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
j=1 c=N  
j=3 c=x  
j=4 c=t  
j=5 c=  
j=6 c==  
j=7 c=  
j=8 c=_  
j=9 c=n  
j=10 c=_  
j=11 c=  
j=12 c=+  
j=13 c=  
j=18 c=;  
The end
```

See Also

Functions:

- “ANYXDIGIT Function” on page 385

NPV Function

Returns the net present value with the rate expressed as a percentage.

Category: Financial
Returned data type: DOUBLE

Syntax

NPV(r, freq, c0, c1, ..., cn)

Arguments

r
is numeric, the interest rate over a specified base period of time expressed as a percentage.

freq
is numeric, the number of payments during the base period of time specified with the rate r.

Range
freq > 0

Note
The case freq = 0 is a flag to allow continuous discounting.

c0, c1, ..., cn
are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ..., n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The c0 argument and the c1 argument are required.

Comparisons

The NPV function is identical to NETPV, except that the r argument is provided as a percentage.

See Also

Functions:
- “NETPV Function” on page 640

NULL Function

Returns a 1 if the argument is null and a 0 if the argument is not null.

Category: Special

Returned data type: INTEGER
Syntax

**NULL(expression)**

**Arguments**

*expression*

specifies any valid expression.

**Data type**

All data types

**Note**

If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2\_SASMISSING environment variable is set to TRUE.

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The NULL function returns a 1 only for a null value. It returns a 0 for any non-null value, including a SAS missing value.

The NULL function returns a 1 if a package instance does not exist, that is the package variable is a null package reference. The NULL function returns a 0 if the package variable references a package instance.

**Note:** Missing values and null values are treated differently in SAS mode versus ANSI mode. Missing and null values might be converted dependent on mode.

**Example**

The following example illustrates how null can differ is SAS mode and in ANSI mode.

```sas
proc ds2;
  data _null_;  
    method init();
      declare char(1) a[3];
      declare double  b[3];
      declare int c[3];
      declare int i;

      a := ('a', '', NULL);
      b := (1, ., NULL);
      c := (1, NULL, NULL);

      do i = 1 to 3;
        if (null(a[i])) then put a[i]= 'null';
        else put a[i]= 'not null';

        if (null(b[i])) then put b[i]= 'null';
        else put b[i]= 'not null';

        if (null(c[i])) then put c[i]= 'null';
        else put c[i]= 'not null';
      end;
  enddata;
```
run;
quit;

proc ds2 ansi mode;
data _null_;  
method init();
declare char(1) a[3];
declare double b[3];
declare int c[3];
declare int i;
  a := ('a', '', NULL);
b := (1, ., NULL);
c := (1, NULL, NULL);
  do i = 1 to 3;
    if (null(a[i])) then put a[i] = 'null';
    else put a[i] = 'not null';
    if (null(b[i])) then put b[i] = 'null';
    else put b[i] = 'not null';
    if (null(c[i])) then put c[i] = 'null';
    else put c[i] = 'not null';
  end;
enddata;
run;
quit;

In SAS mode, the following lines are written to the SAS log.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]=a not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[1]=1 not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[1]=1 not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2]= not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[2]=. not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[2]= NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[3]= not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[3]=. not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[3]= NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In ANSI mode, the following lines are written to the SAS log.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]=a not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[1]=1 not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[1]=1 not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2]= not null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[2]= null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[2]= null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[3]= null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[3]= null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c[3]= null</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that in ANSI mode, b[2] is null because the SAS missing value (.) is converted to null before being assigned to b[2] in b := (1, ., NULL);. In SAS mode, a[3] and b[3] are not null because the null value is converted to a SAS missing value (blank-filled string for a[3] and missing . for b[3]) before being assigned to a[3] and b[3]
\begin{verbatim}
in if (null(a[i])) then put a[i] = 'null'; and else put a[i] = 'not null';
\end{verbatim}

**See Also**
- Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81

**Functions:**
- “MISSING Function” on page 629
- “N Function” on page 637
- “NMISS Function” on page 641

---

### NWKDOM Function

Returns the date for the nth occurrence of a weekday for the specified month and year.

- **Category:** Date and Time
- **Returned data type:** DOUBLE

### Syntax

**NWKDOM**(\(n, \textit{weekday}, \textit{month}, \textit{year}\))

#### Arguments

- **\(n\)**
  - Specifies the numeric week of the month that contains the specified day.
  - Range: 1–5
  - Data type: INTEGER
  - Tip: \(N=5\) indicates that the specified day occurs in the last week of that month. Sometimes \(n=4\) and \(n=5\) produce the same results.

- **\textit{weekday}**
  - Specifies the number that corresponds to the day of the week.
  - Range: 1–7
  - Data type: INTEGER
  - Tip: Sunday is considered the first day of the week and has a \textit{weekday} value of 1.

- **\textit{month}**
  - Specifies the number that corresponds to the month of the year.
  - Range: 1–12
  - Data type: INTEGER
The NWKDOM function returns a SAS date value for the \( n \)th weekday of the month and year that you specify. Use any valid SAS date format, such as the DATE9. format, to display a calendar date. You can specify \( n=5 \) for the last occurrence of a particular weekday in the month.

Sometimes \( n=5 \) and \( n=4 \) produce the same result. These results occur when there are only four occurrences of the requested weekday in the month. For example, if the month of January begins on a Sunday, there will be five occurrences of Sunday, Monday, and Tuesday, but only four occurrences of Wednesday, Thursday, Friday, and Saturday. In this case, specifying \( n=5 \) or \( n=4 \) for Wednesday, Thursday, Friday, or Saturday will produce the same result.

If the year is not a leap year, February has 28 days and there are four occurrences of each day of the week. In this case, \( n=5 \) and \( n=4 \) produce the same results for every day.

**Comparisons**

In the NWKDOM function, the value for weekday corresponds to the numeric day of the week beginning on Sunday. This value is the same value that is used in the WEEKDAY function, where Sunday \( =1 \), and so on. The value for month corresponds to the numeric month of the year beginning in January. This value is the same value that is used in the MONTH function, where January \( =1 \), and so on.

You can use the NWKDOM function to calculate events that are not defined by the HOLIDAY function. For example, if a university always schedules graduation on the first Saturday in June, then you can use the following statement to calculate the date:

```sas
UnivGrad = nwkdom(1, 7, 6, year);
```

**Examples**

**Example 1: Returning Date Values**

The following example uses the NWKDOM function and returns the date for specific occurrences of a weekday for a specified month and year.

```sas
data _null_;  
method run();  
    /* Return the date of the third Monday in May 2012. */  
a=nwkdom(3, 2, 5, 2012);  
    /* Return the date of the fourth Wednesday in November 2012. */  
b=nwkdom(4, 4, 11, 2012);  
    /* Return the date of the fourth Saturday in November 2012. */  
c=nwkdom(4, 7, 11, 2012);  
    /* Return the date of the first Sunday in January 2013. */  
d=nwkdom(1, 1, 1, 2013);  
    /* Return the date of the second Tuesday in September 2012. */  
e=nwkdom(2, 3, 9, 2012);  
    /* Return the date of the fifth Thursday in December 2012. */  
f=nwkdom(5, 5, 12, 2012);  
put a= weekdatx.;  
put b= weekdatx.;
```
Example 2: Returning the Date of the Last Monday in May
The following example returns the date that corresponds to the last Monday in the month of May in the year 2012.

```
data _null_;  
  method init();  
    /* The last Monday in May. */  
    x=nwkdom(5,2,5,2012);  
    put x date9.;  
  end;  
enddata;  
run;  
```

SAS writes the following output to the log:

```
a= Monday, 21 May 2012  
b= Wednesday, 28 November 2012  
c= Saturday, 24 November 2012  
d= Sunday, 6 January 2013  
e= Tuesday, 11 September 2012  
f= Thursday, 27 December 2012  
```

See Also

Functions:
- “HOLIDAY Function” on page 529
- “INTNX Function” on page 564
- “MONTH Function” on page 635
- “WEEKDAY Function” on page 842

ORDINAL Function
Orders a list of values, and returns a value that is based on a position in the list.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
Syntax

ORDINAL(position, expression-1, expression-2 [, ...expression-n])

Arguments

position
specifies an integer that is less than or equal to the number of elements in the list of arguments.

Requirement  position must be a positive number.

Data type  DOUBLE

expression
specifies any valid expression that evaluates to a numeric value.

Requirement  At least two arguments are required.

Data type  DOUBLE

See  Chapter 13, “DS2 Expressions,” on page 93

Details

The ORDINAL function sorts the list and returns the argument in the list that is specified by position. Missing values are sorted low and are placed before any numeric values.

Comparisons

The ORDINAL function counts both null, missing, non-null, and nonmissing values, whereas the SMALLEST function counts only non-null and nonmissing values.

Example

The following statement illustrates the ORDINAL function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=ordinal{4,1,.,2,3,-4,5,6,7}</td>
<td>2</td>
</tr>
</tbody>
</table>

PCTL Function

Returns the percentile that corresponds to the percentage.

Category:  Descriptive Statistics

Returned data type:  DOUBLE

Syntax

PCTL[n](percentage, expression[, ...expression])
Arguments

\( n \)

is a digit from 1 to 5 that specifies the definition of the percentile to be computed.

Default: definition 5

Data type: DOUBLE


percentage

specifies the percentile to be computed.

Data type: DOUBLE

Tip: percentage is numeric where, \(0 \leq \text{percentage} \leq 100\).

expression

specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type: DOUBLE

See: Chapter 13, “DS2 Expressions,” on page 93

Details

The PCTL function returns the percentile of the non-null or nonmissing values corresponding to the percentage. If percentage is null or missing, less than zero, or greater than 100, the PCTL function generates an error message.

Example

The following statements illustrate the PCTL function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower_quartile=PCTL(25,2,4,1,3);</td>
<td>1.5</td>
</tr>
<tr>
<td>percentile_def2=PCTL2(25,2,4,1,3);</td>
<td>1</td>
</tr>
<tr>
<td>lower_tertile=PCTL(100/3,2,4,1,3);</td>
<td>2</td>
</tr>
<tr>
<td>percentile_def3=PCTL3(100/3,2,4,1,3);</td>
<td>2</td>
</tr>
<tr>
<td>median=PCTL(50,2,4,1,3);</td>
<td>2.5</td>
</tr>
<tr>
<td>upper_tertile=PCTL(200/3,2,4,1,3);</td>
<td>3</td>
</tr>
<tr>
<td>upper_quartile=PCTL(75,2,4,1,3);</td>
<td>3.5</td>
</tr>
</tbody>
</table>
PERM Function

Computes the number of permutations of $n$ items that are taken $r$ at a time.

**Category:** Combinatorial

**Returned data type:** INTEGER

### Syntax

PERM($n$, $r$)

### Arguments

$n$

 specifies any valid expression that represents the total number of elements from which the sample is chosen.

**Data type:** INTEGER

See Chapter 13, “DS2 Expressions,” on page 93

$r$

 Specifies any valid expression that represents the number of chosen elements.

**Restriction:** $r \leq n$

**Data type:** INTEGER

**Note:** If $r$ is omitted, the function returns the factorial of $n$.

See Chapter 13, “DS2 Expressions,” on page 93

### Details

The mathematical representation of the PERM function is given by the following equation:

$$PERM(n, r) = \frac{n!}{(n-r)!}$$

with $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the PERM function.

### Example

The following statements illustrate the PERM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x=\text{perm}(5, 1);$</td>
<td>5</td>
</tr>
</tbody>
</table>
### Statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=perm(5);</td>
<td>120</td>
</tr>
<tr>
<td>x=perm(5, 2)</td>
<td>20</td>
</tr>
</tbody>
</table>

### See Also

**Functions:**
- “COMB Function” on page 428
- “FACT Function” on page 488

---

**PMT Function**

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

**Category:** Financial  
**Returned data type:** DOUBLE

#### Syntax

\[
PMT(rate, number-of-periods, principal-amount[, future-amount][, type])
\]

#### Arguments

- **rate**
  - Specifies the interest rate per payment period.  
  - Data type: DOUBLE

- **number-of-periods**
  - Specifies the number of payment periods.  
  - Requirement: Number-of-periods must be a positive integer value.  
  - Data type: INTEGER

- **principal-amount**
  - Specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.  
  - Data type: DOUBLE

- **future-amount**
  - Specifies the future amount. Future-amount can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if future-amount is omitted or if a missing value is specified.
Data type: DOUBLE

$type$

Specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if $type$ is omitted or if a null or missing value is specified.

Data type: INTEGER

Example

- The monthly payment for a $10,000 loan with a nominal annual interest rate of 8% and 10 end-of-month payments can be computed in the following ways:
  \[
  \text{Payment}_1 = \text{PMT}(0.08/12., 10, 10000, 0, 0);
  \]
  \[
  \text{Payment}_1 = \text{PMT}(0.08/12., 10, 10000);
  \]
  These computations return a value of 1037.03208935915.

- If the same loan has beginning-of-period payments, then payment can be computed as follows:
  \[
  \text{Payment}_2 = \text{PMT}(0.08/12., 10, 10000, 0, 1);
  \]
  This computation returns a value of 1030.16432717796.

- The payment for a $5,000 loan earning a 12% nominal annual interest rate, that is to be paid back in five monthly payments, is computed as follows:
  \[
  \text{Payment}_3 = \text{PMT}(0.01/12., 5, 5000);
  \]
  This computation returns a value of 1002.50138831008.

- The payment for monthly periodic savings that accrue more than 18 years at a 6% nominal annual interest rate, and which accumulates $50,000 at the end of the 18 years, is computed as follows:
  \[
  \text{Payment}_4 = \text{PMT}(0.06/12., 216, 0, 50000, 0);
  \]
  This computation returns a value of -129.081160867993.

**POISSON Function**

Returns the probability from a Poisson distribution.

| Category: | Probability |
| Returned data type: | DOUBLE |

**Syntax**

\[
\text{POISSON}(m, n)
\]

**Arguments**

$m$

specifies any valid expression that evaluates to a numeric mean parameter.
The POISSON function returns the probability that an observation from a Poisson distribution, with mean \( m \), is less than or equal to \( n \). To compute the probability that an observation is equal to a given value, \( n \), compute the difference of two probabilities from the Poisson distribution for \( n \) and \( n-1 \).

### Example

The following statement illustrates the POISSON function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{poisson}(1, 2); )</td>
<td>0.9196986029286</td>
</tr>
</tbody>
</table>

---

**POWER Function**

Returns the value of a numeric value expression raised to a specified power.

- **Category:** Mathematical
- **Returned data type:** DOUBLE

### Syntax

\[
\text{POWER}(\text{numeric-expression}, \text{integer-expression})
\]

### Arguments

- **numeric-expression**
  - specifies any valid expression that evaluates to a numeric value.
  - **Data type:** DOUBLE
  - **See:** Chapter 13, “DS2 Expressions,” on page 93
\textit{integer-expression} specifies any valid expression that evaluates to an integer value.

<table>
<thead>
<tr>
<th>Data type</th>
<th>INTEGER, DOUBLE</th>
</tr>
</thead>
</table>

See Chapter 13, “DS2 Expressions,” on page 93

**Details**

If \textit{numeric-expression} is null, then the POWER function returns null. If the result is a number that does not fit into the range of the argument's data type, the POWER function fails.

**Example**

The following statement illustrates the POWER function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a=power(5*3, 2)}</td>
<td>225</td>
</tr>
</tbody>
</table>

**PPMT Function**

Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

- **Category:** Financial
- **Returned data type:** DOUBLE

**Syntax**

\texttt{PPMT(rate, period, number-of-periods, principal-amount[, future-amount][, type])}

**Arguments**

- **rate** specifies the interest rate per payment period.
  - **Data type:** DOUBLE

- **period** specifies the payment period for which the principal payment is computed.
  - **Requirement:** \textit{Period} must be a positive integer value that is less than or equal to the value of \textit{number-of-periods}
  - **Data type:** INTEGER

- **number-of-periods** specifies the number of payment periods.
Requirement: *Number-of-periods* must be a positive integer value.

**Data type:** INTEGER

**principal-amount**

specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

**Data type:** DOUBLE

**future-amount**

specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a null or missing value is specified.

**Data type:** DOUBLE

**type**

specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

**Data type:** INTEGER

**Example**

- The principal payment amount of the first monthly periodic payment for a 2-year, $2,000 loan with a nominal annual interest rate of 10%, is computed as follows:

\[
\text{PrincipalPayment} = \text{PPMT}(0.1/12., 1, 24, 2000);
\]

This computation returns a value of 75.6231860083663.

- The principal payment for a 3-year, $20,000 loan with beginning-of-month payments is computed as follows:

\[
\text{PrincipalPayment2} = \text{PPMT}(0.1/12., 1, 36, 20000, 0, 1);
\]

This computation returns a value of 640.010324505867 as the principal that was paid with the first payment.

- The principal payment of an end-of-month payment loan with an outstanding balance of $5,000 at the end of three years, is computed as follows:

\[
\text{PrincipalPayment3} = \text{PPMT}(0.1/12., 1, 36, 20000, 5000, 0);
\]

This computation returns a value of 359.007807907562 as the principal that was paid with the first payment.

---

**PROBBETA Function**

Returns the probability from a beta distribution.

**Category:** Probability

**Returned data type:** DOUBLE
Syntax

PROBBETA\((x, a, b)\)

Arguments

\(x\)
is a numeric random variable.
Range  \(0 \leq x \leq 1\)
Data type  DOUBLE

\(a\)
is a numeric shape parameter.
Range  \(a > 0\)
Data type  DOUBLE

\(b\)
is a numeric shape parameter.
Range  \(b > 0\)
Data type  DOUBLE

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters \(a\) and \(b\), is less than or equal to \(x\).

Example

The following statement illustrates the PROBBETA function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{probbeta}(0.2, 3, 4));</td>
<td>0.09888</td>
</tr>
</tbody>
</table>

PROBBNML Function

Returns the probability from a binomial distribution.

Category: Probability

Returned data type: DOUBLE
Syntax

PROBBNML\((p, n, m)\)

Arguments

\(p\)
- is a numeric probability of success parameter.
- Range: \(0 \leq p \leq 1\)
- Data type: DOUBLE

\(n\)
- is an integer number of independent Bernoulli trials parameter.
- Range: \(n > 0\)
- Data type: INTEGER

\(m\)
- is an integer number of successes random variable.
- Range: \(0 \leq m \leq n\)
- Data type: INTEGER

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success \(p\), number of trials \(n\), and number of successes \(m\), is less than or equal to \(m\). To compute the probability that an observation is equal to a given value \(m\), compute the difference of two probabilities from the binomial distribution for \(m\) and \(m-1\) successes.

Example

The following statement illustrates the PROBBNML function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{probbnl}(0.5, 10, 4);)</td>
<td>(0.376953125)</td>
</tr>
</tbody>
</table>

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

- **Category:** Probability
- **Returned data type:** DOUBLE
Syntax

PROBBNRM(x, y, r)

Arguments

x
specifies a numeric constant, variable, or expression.
Data type: DOUBLE

y
specifies a numeric constant, variable, or expression.
Data type: DOUBLE

r
is a numeric correlation coefficient.
Range: -1 ≤ r ≤ 1
Data type: DOUBLE

Details

The PROBBNRM function returns the probability that an observation (X, Y) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient r, is less than or equal to (x, y). That is, it returns the probability that X ≤ x and Y ≤ y. The following equation describes the PROBBNRM function, where u and v represent the random variables x and y, respectively:

\[
\text{PROBBNRM}(x, y, r) = \frac{1}{2 \pi \sqrt{1 - r^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp \left\{ -\frac{u^2 - 2ruv + v^2}{2(1 - r^2)} \right\} dv du
\]

Example

The following statement illustrates the PROBBNRM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=probbnrm(.4, -.3, .2);</td>
<td>0.27831833451901</td>
</tr>
</tbody>
</table>

PROBCHI Function

Returns the probability from a chi-square distribution.

Category: Probability
Returned data type: DOUBLE
Syntax
PROBCHI(x, df[, nc])

Arguments
x
is a numeric random variable.
Range \( x \geq 0 \)
Data type DOUBLE

df
is a numeric degrees of freedom parameter.
Range \( df > 0 \)
Data type DOUBLE

nc
is an optional numeric noncentrality parameter.
Range \( nc \geq 0 \)
Data type DOUBLE

Details
The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom \( df \) and noncentrality parameter \( nc \), is less than or equal to \( x \). This function accepts a noninteger degrees of freedom parameter \( df \). If the optional parameter \( nc \) is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example
The following statement illustrates the PROBCHI function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probchi(11.264,11);</td>
<td>0.5785813293173</td>
</tr>
</tbody>
</table>

PROBDF Function
Calculates significance probabilities for Dickey-Fuller tests for unit roots in time series.

Category: Probability
Returned data type: DOUBLE
Syntax

PROBDF(x, n[, d[, type]])

Arguments

x
is the test statistic.

Data type    DOUBLE

n
is the sample size. The minimum value of n that is allowed depends on the value that is specified for the third argument, d. For d in the set (1,2,4,6,12), n must be an integer greater than or equal to max (2d, 5). For other values of d the minimum value of n is 24.

Data type    INTEGER

d
is an integer that gives the degree of the unit root that is tested for. For tests of a simple unit root, (1–B), specify d=1. For tests for a seasonal unit root, specify that d is equal to the seasonal cycle length for tests. The default value of d is 1. That is, a test for a simple unit root is assumed if d is not specified. The maximum value of d is 12.

Data type    INTEGER

Type

is a character argument that specifies the type of test statistic that is used. The values of type are the following:

RSM
specifies the regression test statistic for the single mean (intercept) case.

RTR
specifies the regression test statistic for the deterministic time trend case.

RZM
specifies the regression test statistic for the zero mean (no intercept) case.

SSM
specifies the studentized test statistic for the single mean (intercept) case.

STR
specifies the studentized test statistic for the deterministic time trend case.

SZM
specifies the studentized test statistic for the zero mean (no intercept) case.

Default    SZM

Restriction The values STR and RTR are allowed only when d=1.

Data type    CHAR
Details

Theoretical Background

When a time series has a unit root, the series is nonstationary and the ordinary least squares (OLS) estimator is not normally distributed. Dickey (1976) and Dickey and Fuller (1979) studied the limiting distribution of the OLS estimator of autoregressive models for time series with a simple unit root. Dickey, Hasza, and Fuller (1984) obtained the limiting distribution for time series with seasonal unit roots. This section introduces the nonseasonal tests, and lists references for the nonseasonal tests.

In the Dicky-Fuller regression, the null hypothesis states that there is an autoregressive unit root \( H_0 : \alpha = 1 \), and an alternative, \( H_\alpha : |\alpha| < 1 \), where \( \alpha \) is the autoregressive coefficient of the following time series:

\[
y_t = \alpha y_{t-1} + \epsilon_t
\]

This model is referred to as the zero mean (ZM) model. The standard Dickey-Fuller (DF) test assumes that errors are white noise. There are two other types of regression models that include a constant or a time trend:

\[
y_t = \mu + \alpha y_{t-1} + \epsilon_t
\]

\[
y_t = \mu + \beta t + \alpha y_{t-1} + \epsilon_t
\]

These two models are referred to as the constant mean model (SM) and the trend model (TR), respectively. The constant mean model includes a constant mean \( \mu \) of the time series. However, the interpretation of \( \mu \) depends on the stationarity in the following sense: the mean in the stationary case when \( \alpha < 1 \) is the trend in the integrated case when \( \alpha = 1 \). Therefore, the null hypothesis should be the joint hypothesis that \( \alpha = 1 \) and \( \mu = 0 \). However, for the unit root tests, the test statistics are concerned with the null hypothesis of \( \alpha = 1 \). The joint null hypothesis is not commonly used. This issue is address in Bhargava, A. (1986) with a different nesting model.

Under the null of I(1) of the Dickey-Fuller test, the differenced process is not serially correlated. There is a great need for the generalization of this specification. The augmented Dickey-Fuller (ADF) test, originally proposed in Dickey and Fuller (1979), adjusts for the serial correlation in the time series by adding lagged first differences to the autoregressive model as follows. Consider the \( (p+1) \)th order autoregressive time series:

\[
y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_{p+1} y_{t-p-1} + \epsilon_t
\]

The characteristic equation follows:

\[
m^{p+1} - \alpha_1 m^p - \alpha_2 m^{p-1} - \ldots - \alpha_{p+1} = 0
\]

If all the characteristic roots are less than 1 in absolute value, \( y_t \) is stationary. \( y_t \) is nonstationary if there is a unit root. If there is a unit root, the sum of the autoregressive parameters is 1, and therefore you can test for a unit root by testing whether the sum of the autoregressive parameters is 1. The no-intercept model is parameterized as follows.

\[
\nabla y_t = \delta y_{t-1} + \theta_1 \nabla y_{t-1} + \ldots + \theta_p \nabla y_{t-p} + \epsilon_t
\]

In the equation above, the following relationships apply:

\[
\nabla y_t = y_t - y_{t-1}
\]
and
\[ \delta = a_1 + \ldots + a_p + 1 - 1 \]
\[ \theta_k = -a_{k+1} - \ldots - a_{p+1} \]

The estimators are obtained by regressing \( \nabla y_t \) on \( y_{t-1}, \nabla y_{t-1}, \ldots, \nabla y_{t-p} \). The \( t \) statistic of the ordinary least squares estimator of \( \delta \) is the test statistic for the unit root test.

If the type argument value specifies a test for a nonzero mean (intercept case), the autoregressive model includes a mean term \( a_0 \). If the type argument value specifies a test for a time trend, the model also includes a time trend term and the model is as follows:
\[ \nabla y_t = a_0 + \gamma t + \delta y_{t-1} + \theta_1 \nabla y_{t-1} + \ldots + \theta_p \nabla y_{t-p} + \epsilon \]

For testing for a seasonal unit root, consider the multiplicative model.
\[ (1 - \theta_1 \beta - \ldots - \theta_p \beta^p) y_t = \epsilon_t \]

Let \( \nabla^d y_t \equiv y_t - y_{t-d} \). The test statistic is calculated in the following steps:
1. Regress \( \nabla^d y_t \) on \( \nabla^d y_{t-1}, \ldots, \nabla^d y_{t-p} \) to obtain the initial estimators \( \hat{\theta}_i \) and compute residuals \( \hat{\epsilon}_i \). Under the null hypothesis that \( \alpha_d = 1 \), \( \hat{\theta}_i \) are consistent estimators of \( \theta_i \).
2. Regress \( \hat{\epsilon}_i \) on \( (1 - \hat{\theta}_1 \beta - \ldots - \hat{\theta}_p \beta^p) y_{t-d}, \nabla^d y_{t-1}, \ldots, \nabla^d y_{t-p} \) to obtain estimates of \( \delta = \alpha_d - 1 \) and \( \theta_i - \hat{\theta}_i \).

The \( t \) ratio for the estimates of \( \delta \) that are produced by the second step is used as a test statistic for testing for a seasonal unit root. The estimates of \( \theta_i \) are obtained by adding the estimates of \( \theta_i - \hat{\theta}_i \) from the second step to \( \hat{\theta}_i \) from the first step.

The series \( (1 - B^d)y \) is assumed to be stationary, where \( d \) is the value of the third argument to the PROBDF function.

If the series is an ARMA process, then a large value of \( p \) might be desirable in order to obtain a reliable test statistic. To determine an appropriate value for \( p \), see Said and Dickey (1984).

**Test Statistics**

The Dickey-Fuller test is used to test the null hypothesis that the time series exhibits a lag \( d \) unit root against the alternative of stationarity. The PROBDF function computes the probability of observing a test statistic more extreme than \( x \) under the assumption that the null hypothesis is true. You should reject the unit root hypothesis when PROBDF returns a small (significant) probability value.

Consider the Dickey-Fuller regression first. There are several versions of the Dickey-Fuller test. The PROBDF function supports six versions, as selected by the type argument. Specify the type value that corresponds to how you calculated the test statistic \( x \).

The last two characters of the type value specify the type of regression model that is used to compute the Dickey-Fuller test statistic. The meaning of the last two characters of the type value are as follows:
SM
specifies a single mean or intercept case. The test statistic $x$ is assumed to be computed from the following regression model:

$$y_t = \mu + \alpha y_{t-1} + e_t$$

TR
specifies the intercept and deterministic time trend case. The test statistic $x$ is assumed to be computed from the following regression model:

$$y_t = \mu + \gamma t + \alpha y_{t-1} + e_t$$

ZM
specifies the zero mean or no-intercept case. The test statistic $x$ is assumed to be computed from the following regression model:

$$y_t = \alpha y_{t-1} + e_t$$

The first character of the type value specifies whether the regression test statistic or the studentized test statistic is used. Let $\hat{\alpha}$ be the estimated regression coefficient for the lag of the series, and let $se_\alpha$ be the standard error of $\hat{\alpha}$. The meaning of the first character of the type value is as follows:

R
specifies the regression-coefficient-based test statistic. The test statistic follows:

$$\rho = n(\hat{\alpha} - 1)$$

S
specifies the studentized test statistic. The test statistic follows:

$$DF_t = \frac{\hat{\alpha} - 1}{se_\alpha}$$

The equation for the type value of R is also called $\rho$-test. The equation for the type value of S is also called $\tau$-test. For the zero mean model, the asymptotic distributions of the Dickey-Fuller test statistics follow:

$$n(\hat{\alpha} - 1) \Rightarrow \left( \int_0^1 W(r) dW(r) \right) \left( \int_0^1 W(r)^2 dr \right)^{-1}$$

$$DF_t \Rightarrow \left( \int_0^1 W(r) dW(r) \right) \left( \int_0^1 W(r)^2 dr \right)^{-1/2}$$

For the constant mean model, the asymptotic distributions follow:

$$n(\hat{\alpha} - 1) \Rightarrow \left[ W(1)^2 - 1 \right]/2 - W(1) \int_0^1 W(r) dr \left[ \int_0^1 W(r)^2 dr - \left( \int_0^1 W(r) dr \right)^2 \right]^{-1}$$

$$DF_t \Rightarrow \left[ W(1)^2 - 1 \right]/2 - W(1) \int_0^1 W(r) dr \left[ \int_0^1 W(r)^2 dr - \left( \int_0^1 W(r) dr \right)^2 \right]^{-1/2}$$

For the trend model, the asymptotic distributions follow:
\[ n(\hat{\alpha} - 1) \Rightarrow \left[ W(r) dW + 12 \left( \int_0^1 r W(r) \, dr - \frac{1}{2} \int_0^1 W(r) \, dr \right) \left( \int_0^1 W(r) \, dr - \frac{1}{2} W(1) \right) \right. \]
\[ \left. - W(1) \int_0^1 W(r) \, dr \right] D^1 \]

\[ DF_r \Rightarrow \left[ W(r) dW + 12 \left( \int_0^1 r W(r) \, dr - \frac{1}{2} \int_0^1 W(r) \, dr \right) \left( \int_0^1 W(r) \, dr - \frac{1}{2} W(1) \right) \right. \]
\[ \left. - W(1) \int_0^1 W(r) \, dr \right] D^{1/2} \]

The following equation applies to the equations that are shown above:

\[ D = \int_0^1 W(r)^2 \, dr - 12 \left( \int_0^1 W(r) \, dr \right)^2 + 12 \int_0^1 W(r) \, dr \int_0^1 r W(r) \, dr - 4 \left( \int_0^1 W(r) \, dr \right)^2 \]

For more information about the Dickey-Fuller test null distribution, see Dickey and Fuller (1979), Dickey, Hasza, and Fuller (1984), and Hamilton (1994). The preceding formulas are for the basic Dickey-Fuller test. The PROBDF function can also be used for the augmented Dickey-Fuller test, in which the error term is modeled as an autoregressive process. However, the test statistic is computed somewhat differently for the augmented Dickey-Fuller test. For the nonseasonal augmented Dickey-Fuller test, the test statistics can have one of the two forms similar to Dickey-Fuller test. One of the forms is the OLS \( t \) value, \( \frac{\hat{\alpha} - 1}{sd(\alpha)} \), and the other form is \( \frac{n(\hat{\alpha} - 1)}{1 - \hat{\alpha} - \ldots - \hat{\alpha}_p} \)

**Example**

In the following example, the table Test contains 104 observations of the time series variable \( Y \). The program tests the null hypothesis that there exists a lag 4 seasonal unit root in the \( Y \) series. The following statements illustrate how to perform the single-mean Dickey-Fuller regression coefficient test using PROC REG and the PROBDF function.

```
data test1;
  set test;
  y4 = lag4(y);
run;

proc reg data=test1 outest=alpha;
  model y = y4 / noprint;
run;

proc ds2;
data _null_; method run();
  set alpha;
  x = 100 * ( y4 - 1 );
  p = probdf( x, 100, 4, "RSM" );
  put p= pvalue5.3;
end;
enddate;
run;
quit;
```

To perform the augmented Dickey-Fuller test, regress the differences of the series on lagged differences and on the lagged value of the series, and compute the test statistic.
from the regression coefficient for the lagged series. The following statements illustrate how to perform the single-mean augmented Dickey-Fuller studentized test for a simple unit root using PROC REG and the PROBDF function:

```sas
data test1;
  set test;
  yl  = lag(y);
  yd  = dif(y);
  yd1 = lag1(yd); yd2 = lag2(yd);
  yd3 = lag3(yd); yd4 = lag4(yd);
run;

proc reg data=test1 outest=alpha covout;
  model yd = yl yd1-yd4 / noprint;
run;

proc ds2;
  data _null_; method run();
    set alpha;
    retain a;
    if _type_ = 'PARMS' then
      a = yl;
    if _type_ = 'COV' & _NAME_ = 'Y1' then do;
      x = a / sqrt(yl);
      p = probdf( x, 99, 1, "SSM" );
      put p= ;
    end;
  enddata;
  run;
  quit;
```

The %DFTEST macro provides an easier way to perform the Dickey-Fuller tests. The following statements perform the same tests as the preceding example:

```sas
%dftest(test, y, ar=4);
%put p=&dftest;
```

**Note:** When DS2 runs outside of SAS, such as in the SAS Federation Server and in grid computing environments, the SAS macro facility is not available and DS2 programs with macros fail to compile.

---

**PROBF Function**

Returns the probability from an $F$ distribution.

- **Category:** Probability
- **Returned data type:** DOUBLE

**Syntax**

```
PROBF(x, ndf, ddf[, nc])
```
Arguments

\( x \)

is a numeric random variable.

Range \( x \geq 0 \)

Data type \( \text{DOUBLE} \)

\( ndf \)

is a numeric numerator degrees of freedom parameter.

Range \( ndf > 0 \)

Data type \( \text{DOUBLE} \)

\( ddf \)

is a numeric denominator degrees of freedom parameter.

Range \( ddf > 0 \)

Data type \( \text{DOUBLE} \)

\( nc \)

is an optional numeric noncentrality parameter.

Range \( nc \geq 0 \)

Data type \( \text{DOUBLE} \)

Details

The PROBF function returns the probability that an observation from an \( F \) distribution, with numerator degrees of freedom \( ndf \), denominator degrees of freedom \( ddf \), and noncentrality parameter \( nc \), is less than or equal to \( x \). The PROBF function accepts noninteger degrees of freedom parameters \( ndf \) and \( ddf \). If the optional parameter \( nc \) is not specified or has the value 0, the value returned is from the central \( F \) distribution.

The significance level for an \( F \) test statistic is given by the following equation.

\[ p = 1 - \text{probf}(x, ndf, ddf); \]

Example

The following statement illustrates the PROBF function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{probf}(3.32, 2, 3); )</td>
<td>0.82639336022431</td>
</tr>
</tbody>
</table>

PROBGAM Function

Returns the probability from a gamma distribution.

Category: Probability
Returned data type: DOUBLE

Syntax

PROBGAM(x, a)

Arguments

x
is a numeric random variable.
Range  x ≥ 0
Data type  DOUBLE

a
is a numeric shape parameter.
Data type  DOUBLE

Details

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter a, is less than or equal to x.

Example

The following statement illustrates the PROBGAM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probgam(1,3);</td>
<td>0.08030139707139</td>
</tr>
</tbody>
</table>

PROBHYPYPR Function

Returns the probability from a hypergeometric distribution.

Category: Probability
Returned data type: DOUBLE

Syntax

PROBHYPYPR(N, K, n, x[, r])

Arguments

N
is an integer population size parameter.
**Details**

The PROBHYPR function returns the probability that an observation from an extended hypergeometric distribution, with population size \( N \), number of items \( K \), sample size \( n \), and odds ratio \( r \), is less than or equal to \( x \). If the optional parameter \( r \) is not specified or is set to 1, the value returned is from the usual hypergeometric distribution.

**Example**

The following statement illustrates the PROBHYPR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probhypr(200,50,10,2);</td>
<td>0.52367340812167</td>
</tr>
</tbody>
</table>

**PROBIT Function**

Returns a quantile from the standard normal distribution.

- **Category:** Quantile
- **Returned data type:** DOUBLE
Syntax

PROBIT\((p)\)

Arguments

\(p\)

is a numeric probability.

Range \(0 < p < 1\)

Data type DOUBLE

Details

The PROBIT function returns the \(p^\text{th}\) quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is \(p\).

\textbf{CAUTION:}

The result could be truncated to lie between -8.222 and 7.941.

\textit{Note:} PROBIT is the inverse of the PROBNORM function.

Example

The following statements illustrate the PROBIT function:

\begin{tabular}{|l|l|}
\hline
Statements & Results \\
\hline
x=probit(.025); & -1.95996398454005 \\
\hline
x=probit(1.e-7); & -5.19933758219281 \\
\hline
\end{tabular}

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

\begin{itemize}
\item Category: Probability
\item Returned data type: DOUBLE
\end{itemize}

Syntax

PROBMC\((\text{distribution}, q, \text{prob}, df, nparms[], \text{parameters})\)
Arguments
distribution

is a character constant, variable, or expression that identifies the distribution. The following are valid distributions:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of Means</td>
<td>ANOM</td>
</tr>
<tr>
<td>One-sided Dunnett</td>
<td>DUNNETT1</td>
</tr>
<tr>
<td>Two-sided Dunnett</td>
<td>DUNNETT2</td>
</tr>
<tr>
<td>Maximum Modulus</td>
<td>MAXMOD</td>
</tr>
<tr>
<td>Partitioned Range</td>
<td>PARTRANGE</td>
</tr>
<tr>
<td>Studentized Range</td>
<td>RANGE</td>
</tr>
<tr>
<td>Williams</td>
<td>WILLIAMS</td>
</tr>
</tbody>
</table>

Data type CHAR

q

is the quantile from the distribution.

Data type DOUBLE

Restriction Either q or prob can be specified, but not both.

prob

is the left probability from the distribution.

Data type DOUBLE

Restriction Either prob or q can be specified, but not both.

df

is the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms

is the number of treatments.

Data type DOUBLE

Note For DUNNETT1 and DUNNETT2, the control group is not counted.

parameters

is a set of nparms parameters that must be specified to handle the case of unequal sample sizes. The meaning of parameters depends on the value of distribution. If parameters is not specified, equal sample sizes are assumed, which is usually the case for a null hypothesis.
Overview
The PROBMC function returns the probability or the quantile from various distributions with finite and infinite degrees of freedom for the variance estimate.

The prob argument is the probability that the random variable is less than q. Therefore, p-values can be computed as 1 – prob. For example, to compute the critical value for a 5% significance level, set prob= 0.95. The precision of the computed probability is O(10^{-8}) (absolute error); the precision of computed quantile is O(10^{-5}).

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

Note: Williams' test is computed only for equal sample sizes.

Formulas and Parameters
The equations listed here define expressions that are used in equations that relate the probability, prob, and the quantile, q, for different distributions and different situations within each distribution. For these equations, let v be the degrees of freedom, df.

\[ d\mu_v(x) = \frac{v}{\Gamma\left(\frac{v}{2}\right)} x^{v/2 - 1} e^{-x/2} dx \]

\[ \phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \]

\[ \Phi(x) = \int_{-\infty}^{x} \phi(u) du \]

Computing the Analysis of Means
Analysis of Means (ANOM) applies to data that is organized as k (Gaussian) samples, the i\textsuperscript{th} sample being of size n\textsubscript{i}. Let I = \sqrt{-1}. The distribution function \[1, 2, 3, 4, 5\] is the CDF for the maximum absolute of a k-dimensional multivariate vector, with ν degrees of freedom, and an associated correlation matrix \[\rho_{ij} = -\alpha_i \alpha_j\]. This equation can be written as follows.

\[ \text{prob} = r(\left|t_1\right| < h, \left|t_2\right| < h, ..., \left|t_k\right| < h) \]

\[ = \int_{0}^{\infty} \left\{ \int_{0}^{\infty} \prod_{j=0}^{k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_v(s) \]

The following relationship applies to the preceding equation:

\[ g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1 + \alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1 + \alpha_j^2}}\right) \]

In this equation, \(\Gamma(\cdot), \phi(\cdot),\) and \(\Phi(\cdot)\), are the gamma function, the density, and the CDF from the standard normal distribution, respectively.
For $\nu = \infty$, the distribution reduces to this equation.

$$r(t_1 < h, t_2 < h, ..., t_k < h) = \int_0^{\infty} \prod_{j=0}^{k} g(h, y, \alpha_j) \phi(y) dy$$

The following relationship applies to the preceding equation:

$$g(h, y, \alpha_j) = \Phi \left( \frac{h - y\alpha_j}{\sqrt{1 + \alpha_j^2}} \right) - \Phi \left( \frac{-h - y\alpha_j}{\sqrt{1 + \alpha_j^2}} \right)$$

For the balanced case, the distribution reduces to the following equation:

$$r(t_1 < h, t_2 < h, ..., t_n < h) = \int_0^{\infty} f(h, y, \rho) \phi(y) dy$$

The following relationships apply to the preceding equation:

$$f(h, y, \rho) = \Phi \left( \frac{h - y\sqrt{\rho}}{\sqrt{1 + \rho}} \right) - \Phi \left( \frac{-h - y\sqrt{\rho}}{\sqrt{1 + \rho}} \right)$$

$$\rho = \frac{1}{n-1}$$

Here is the syntax for this distribution:

```c
x = probmc('anom', q, p, nu, n[, alpha1, ..., alphak]);
```

### Arguments

- **x** is a numeric value with the returned result.
- **q** is a numeric value that denotes the quantile.
- **p** is a numeric value that denotes the probability. One of $p$ and $q$ must be missing.
- **nu** is a numeric value that denotes the degrees of freedom.
- **n** is a numeric value that denotes the number of samples.
- **alpha_i**, $i = 1, ..., k$ are optional numeric values denoting the alpha values from the first equation of this distribution. See “Computing the Analysis of Means” on page 699.

### Many-One t-Statistics: Dunnett's One-Sided Test

- This case relates the probability, prob, and the quantile, q, for the unequal case with finite degrees of freedom. The parameters are $\lambda_1, ..., \lambda_k$, the value of nparms is set to k, and the value of df is set to v. The equation follows:

$$prob = \int_0^{\infty} \int_{-\infty}^{\infty} \frac{\phi(y)}{\sqrt{1 - z_i^2}} \prod_{i=1}^{k} \phi \left( \frac{y\lambda_i + q\lambda_i}{\sqrt{1 - z_i^2}} \right) dy \, dx$$

- This case relates the probability, prob, and the quantile, q, for the equal case with finite degrees of freedom. No parameters are passed ($\lambda = \sqrt{\frac{1}{2}}$), the value of nparms is set to k, and the value of df is set to v. The equation follows:
\[
prob = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \phi(y) \left( \Phi(y + \sqrt{2qx}) \right)^k \, dy \, du.
\]

- This case relates the probability, \(prob\), and the quantile, \(q\), for the unequal case with infinite degrees of freedom. The parameters are \(\lambda_1, \ldots, \lambda_k\), the value of \(np\) is set to \(k\), and the value of \(df\) is set to missing. The equation follows:

\[
prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \frac{\phi\left(\frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}}\right)}{\Phi\left(\frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}}\right)} \, dy
\]

- This case relates the probability, \(prob\), and the quantile, \(q\), for the equal case with infinite degrees of freedom. No parameters are passed, \(\lambda = \frac{1}{\sqrt{2}}\), the value of \(np\) is set to \(k\), and the value of \(df\) is set to missing. The equation follows:

\[
prob = \int_{-\infty}^{\infty} \phi(y) \left( \Phi(y + \sqrt{2q}) \right)^k \, dy
\]

**Many-One t-Statistics: Dunnett’s Two-sided Test**

- This case relates the probability, \(prob\), and the quantile, \(q\), for the unequal case with finite degrees of freedom. The parameters are \(\lambda_1, \ldots, \lambda_k\), the value of \(np\) is set to \(k\), and the value of \(df\) is set to \(v\). The equation follows:

\[
prob = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \frac{\phi\left(\frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}}\right)}{\Phi\left(\frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}}\right)} \, dy \, du.
\]

- This case relates the probability, \(prob\), and the quantile, \(q\), for the equal case with finite degrees of freedom. No parameters are passed, the value of \(np\) is set to \(k\), and the value of \(df\) is set to \(v\). The equation follows:

\[
prob = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \phi(y) \left( \Phi(y + \sqrt{2q}) - \Phi(y - \sqrt{2q}) \right)^k \, dy \, du.
\]

- This case relates the probability, \(prob\), and the quantile, \(q\), for the unequal case with infinite degrees of freedom. The parameters are \(\lambda_1, \ldots, \lambda_k\), the value of \(np\) is set to \(k\), and the value of \(df\) is set to missing. The equation follows:

\[
prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \frac{\phi\left(\frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}}\right)}{\Phi\left(\frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}}\right)} \, dy
\]

- This case relates the probability, \(prob\), and the quantile, \(q\), for the equal case with infinite degrees of freedom. No parameters are passed, the value of \(np\) is set to \(k\), and the value of \(df\) is set to missing. The equation follows:

\[
prob = \int_{-\infty}^{\infty} \phi(y) \left( \Phi(y + \sqrt{2q}) - \Phi(y - \sqrt{2q}) \right)^k \, dy
\]

**Computing the Partitioned Range**

RANGE applies to the distribution of the studentized range for \(n\) group means. PARTRANGE applies to the distribution of the partitioned studentized range. Let the \(n\) groups be partitioned into \(k\) subsets of size \(n_1 + \ldots + n_k = n\). Then the partitioned range is the maximum of the studentized ranges in the respective subsets. The studentization factor is the same in all cases.
\[ \text{prob} = \int_{0}^{\infty} \prod_{i=1}^{k} \left( \int_{-\infty}^{\infty} k \phi(y)(\Phi(y) - \Phi(y - qx))^k - 1 \, dy \right)^{n_i} d\mu(x) \]

Here is the syntax for this distribution:

\[ x = \text{probmc}(\text{partrange}', q, p, \text{nu}, k, n_1, \ldots, n_k); \]

**Arguments**

\( x \)
- is a numeric value with the returned result (either the probability or the quantile).

\( q \)
- is a numeric value that denotes the quantile.

\( p \)
- is a numeric value that denotes the probability. One of \( p \) and \( q \) must be missing.

\( \text{nu} \)
- is a numeric value that denotes the degrees of freedom.

\( k \)
- is a numeric value that denotes the number of groups.

\( n_i, i=1,\ldots,k \)
- are optional numeric values that denote the \( n \) values from the equation in this distribution. See “Computing the Partitioned Range” on page 701.

**The Studentized Range**

- This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( v \). The equation follows:

\[ \text{prob} = \int_{0}^{\infty} \prod_{i=1}^{k} \left( \int_{-\infty}^{\infty} k \phi(y)(\Phi(y) - \Phi(y - qx))^k - 1 \, dy \right)^{n_i} d\mu(x) \]

- This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The parameters are \( \sigma_1, \ldots, \sigma_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[ \text{prob} = \int_{-\infty}^{\infty} \sum_{j=1}^{k} \left( \prod_{i=1}^{k} \phi\left( \frac{y}{\sigma_i} \right) - \phi\left( \frac{y - q}{\sigma_i} \right) \right) \phi\left( \frac{y}{\sigma_j} \right)^{1/k} \, dy \]

- This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[ \text{prob} = \int_{-\infty}^{\infty} k \phi(y)(\Phi(y) - \Phi(y - q))^k - 1 \, dy \]

**The Studentized Maximum Modulus**

- This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with finite degrees of freedom. The parameters are \( \sigma_1, \ldots, \sigma_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( v \). The equation follows:

\[ \text{prob} = \int_{0}^{\infty} \prod_{i=1}^{k} \left[ 2\phi\left( \frac{q_x}{\sigma_i} \right) - 1 \right] d\mu(x) \]
• This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No parameters are passed, the value of \( nparms \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_0^\infty (2\Phi(qx) - 1)^k d\mu(x)
\]

• This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The parameters are \( \sigma_1, ..., \sigma_k \), the value of \( nparms \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = \prod_{i=1}^k \left[ 2\Phi\left(\frac{q}{\sigma_i}\right) - 1 \right]
\]

• This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No parameters are passed, the value of \( nparms \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = [2\Phi(q) - 1]^k
\]

**Williams’ Test**

PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972). See “References” in SAS Functions and CALL Routines: Reference. The need for the Williams’ Test arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.

**Note:** Williams’ Test is computed only for equal sample sizes.

Let \( X_1, X_2, ..., X_k \) be identical independent \( N(0,1) \) random variables. Let \( Y_k \) denote their average given by the following equation:

\[
Y_k = \frac{X_1 + X_2 + ... + X_k}{k}
\]

It is required to compute the distribution of the following value.

\[
(Y_k - Z)/S
\]

**Arguments**

\( Y_k \)

is as defined previously.

\( Z \)

is an \( N(0,1) \) independent random variable.

\( S \)

is such that \( \frac{1}{2\nu}S^2 \) is a \( \chi^2 \) variable with \( \nu \) degrees of freedom.

As described in Williams (1971), the full computation is extremely lengthy, and is carried out in three stages. See “References” in SAS Functions and CALL Routines: Reference.

1. Compute the distribution of \( Y_k \). It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of \( Y_k \). Let \( U_i \) be defined in this equation.

\[
U_i = \frac{X_1 + X_2 + ... + X_i}{i}, \quad i = 1, 2, ..., k
\]
You can write a recursive expression for the probability of \( Y_k > d \). The value of \( d \) can be any real number.

\[
\Pr(Y_k > d) = \Pr(U_1 > d) \\
+ \Pr(U_2 > d, U_1 < d) \\
+ \Pr(U_3 > d, U_2 < d, U_1 < d) \\
+ \ldots \\
+ \Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d) \\
= \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)
\]

To compute this probability, start from an \( N(0,1) \) density function.

\[
D(U_1 = x) = \phi(x)
\]

And recursively compute the convolution.

\[
D(U_k = x, U_{k-1} < d, \ldots, U_1 < d) = \\
\int_{-\infty}^{d} D(U_{k-1} = y, U_{k-2} < d, \ldots, U_1 < d)(k-1)\phi(kx - (k-1)y)dy
\]

From this sequential convolution, it is possible to compute all the elements of the recursive equation for \( \Pr(Y_k < d) \), shown previously.

2. Compute the distribution of \( Y_k - Z \). This computation involves another convolution to compute the probability.

\[
\Pr((Y_k - Z) > t) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2d + y})\phi(y)dy
\]

3. Compute the distribution of \( (Y_k - Z)/S \). This computation involves another convolution to compute the probability.

\[
\Pr((Y_k - Z) > ts) = \int_{0}^{\infty} \Pr((Y_k - Z) > ty)d\mu_{\nu}(y)
\]

The third stage is not needed when \( \nu = \infty \). Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when \( k \leq 15 \) for both finite and infinite degrees of freedom \( \nu \). For \( k \geq 16 \), the lengthy computation is carried out. It is extremely expensive and very slow due to the complexity of the algorithm.

**Comparisons**

The MEANS statement in the GLM Procedure of SAS/STAT Software computes the following tests:

- Dunnett's one-sided test
- Dunnett's two-sided test
- Studentized Range
Examples

**Example 1: Computing Probabilities By Using PROBMC**

This example shows how to compute probabilities.

```sas
data _null_; method run();
    declare double par[5];
    declare double df q prob;
    declare char(20) test[3];
    declare int i;

    par := (.5 .51 .55 .45 .2);
    df = 40;
    q = 1;
    test := ('dunnett1' 'dunnett2' 'maxmod');
    do i = 1 to dim(test);
        prob=probmc(test[i], q, ., df, 5, par[1], par[2], par[3], par[4],
                       par[5]);
        put test[i] $10. df q e18.13 prob e18.13;
    end;
end;
enddata;
run;
```

SAS writes the following results to the log:

<table>
<thead>
<tr>
<th>Test</th>
<th>df</th>
<th>Probability</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dunnett1</td>
<td>40</td>
<td>1.00000000000E+00</td>
<td>4.82992196083E-01</td>
</tr>
<tr>
<td>dunnett2</td>
<td>40</td>
<td>1.00000000000E+00</td>
<td>1.64023105316E-01</td>
</tr>
<tr>
<td>maxmod</td>
<td>40</td>
<td>1.00000000000E+00</td>
<td>8.02784203408E-01</td>
</tr>
</tbody>
</table>

**Example 2: Computing the Analysis of Means**

```sas
proc ds2;
    data _null_; method run();
        q1=probmc('anom',.,0.9,.,20);
        put q1=;
        q2=probmc('anom',.,0.9,20,5,0.1,0.1,0.1,0.1,0.1);
        put q2=;
        q3=probmc('anom',.,0.9,20,5,0.5,0.5,0.5,0.5,0.5);
        put q3=;
        q4=probmc('anom',.,0.9,20,5,0.1,0.2,0.3,0.4,0.5);
        put q4=;
    end;
enddata;
run;
quit;
```

SAS writes the following results to the log:

<table>
<thead>
<tr>
<th>q1</th>
<th></th>
<th>q2</th>
<th></th>
<th>q3</th>
<th></th>
<th>q4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.78950610163346</td>
<td></td>
<td>2.45499619666786</td>
<td></td>
<td>2.45499619666786</td>
<td></td>
<td>2.45323199941123</td>
<td></td>
</tr>
</tbody>
</table>
Example 3: Computing the Partitioned Range

```sas
data _null_;  
method run();  
q1=probmc('partrange',.,0.9,.,4,3,4,5,6);  
put q1=;  
q2=probmc('partrange',.,0.9,12,4,3,4,5,6);  
put q2=;  
end;  
enddata;  
run;  
```

SAS writes the following results to the log:

```
q1=4.1022397989482
q2=4.7888626337511
```

Example 4: Computing Williams’ Test

In the following example, a substance has been tested at seven levels in a randomized block design of eight blocks. The observed treatment means are as follows:

<table>
<thead>
<tr>
<th>Treatment</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>10.4</td>
</tr>
<tr>
<td>X₁</td>
<td>9.9</td>
</tr>
<tr>
<td>X₂</td>
<td>10.0</td>
</tr>
<tr>
<td>X₃</td>
<td>10.6</td>
</tr>
<tr>
<td>X₄</td>
<td>11.4</td>
</tr>
<tr>
<td>X₅</td>
<td>11.9</td>
</tr>
<tr>
<td>X₆</td>
<td>11.7</td>
</tr>
</tbody>
</table>

The mean square, with $(7 - 1)(8 - 1) = 42$ degrees of freedom, is $s^2 = 1.16$.

Determine the maximum likelihood estimates $M_i$ through the averaging process.

- Because $X₀ > X₁$, form $X_{0,1} = (X₀ + X₁)/2 = 10.15$.
- Because $X_{0,1} > X₂$, form $X_{0,1,2} = (X₀ + X₁ + X₂)/3 = (2X_{0,1} + X₂)/3 = 10.1$.
- $X_{0,1,2} < X₃ < X₄ < X₅$
- Because $X₃ > X₅$, form $X_{3,5} = (X₃ + X₅)/2 = 11.8$.

Now the order restriction is satisfied.

The maximum likelihood estimates under the alternative hypothesis are:

- $M₀ = M₁ = M₂ = X_{0,1,2} = 10.1$
- $M₃ = X₃ = 10.6$
\( M_4 = X_4 = 11.4 \)
\( M_5 = M_6 = X_{5,6} = 11.8 \)

Now compute \( t = \left( 11.8 - 10.4 \right) / \sqrt{2s^2 / 8} = 2.60 \), and the probability that corresponds to \( k = 6, \nu = 42, \) and \( t = 2.60 \) is \( .9924467341 \), which shows strong evidence that there is a response to the substance. You can also compute the quantiles for the upper 5% and 1% tails, as shown in the following table.

```
data _null_;  
   method run();  
      prob=probmc('WILLIAMS',2.6,.,42,6);  
      put prob=;  
      quant5=probmc('WILLIAMS',.,95,42,6);  
      put quant5=;  
      quant1=probmc('WILLIAMS',.,99,42,6);  
      put quant1= ;  
   end;  
enddate;  
run;```

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob=probmc('williams',2.6,.,42,6);</td>
<td>0.99244668715827</td>
</tr>
<tr>
<td>quant5=probmc('williams',.,95,42,6);</td>
<td>1.80656253603889</td>
</tr>
<tr>
<td>quant1=probmc('williams',.,99,42,6);</td>
<td>2.49090827298686</td>
</tr>
</tbody>
</table>

**References**


PROBMED Function

Computes cumulative probabilities for the sample median.

**Category:** Probability

**Returned data type:** DOUBLE

**Syntax**

PROBMED(n, x)

**Arguments**

n

specifies the sample size.

Data type DOUBLE

x

is the point of interest. That is, the PROBMED function calculates the probability that the median is less than or equal to x.

Data type DOUBLE

**Details**

The PROBMED function computes the probability that the sample median is less than or equal to x for a sample of n independent, standard normal random variables (mean 0, variance 1).

Let n represent the sample size, and \(x_{(i)}\) represents the ith order statistic. Then, when n is odd, the function makes the following calculation:

\[ \Pr\left[ X_{\left(\frac{n+1}{2}\right)} \leq x \right] = I_{\Phi(x)} \left( \frac{n+1}{2}, \frac{n+1}{2} \right) \]

The following equations refer to the preceding equation:

\[ I_p(a, b) = \frac{1}{B(a, b)} \int_0^b t^{a-1}(1-t)^{b-1} dt \]

In the equation \(B(a, b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)\), \(\Gamma(.)\) is the gamma function. If n is even, the PROBMED function performs the following calculation:

\[ \Pr\left[ \frac{X_{(n/2)} + X_{(n/2) + 1}}{2} \leq x \right] = \]

\[ \frac{2}{B(\frac{n}{2}, \frac{n}{2})} \int_x^\infty \left[ (1 - \Phi(u))^{n/2} - (1 - \Phi(2x-u))^{n/2} \right] \Phi(u) \left( \frac{n}{2} \right) - 1 \phi(u) du \]

In this equation, \(B(n/2, n/2) = [\Gamma(n/2)]^2/\Gamma(n)\), and \(\Phi(.)\) and \(\phi(.)\) are the standard normal cumulative distribution function and density function, respectively.
Example

data _null_;  
    method run();  
    b=probmed(5,-0.1);  
    put b;  
    end;  
enddata;  
run;

SAS writes the following output to the log:

0.42563808966747

References


PROBNEGB Function

Returns the probability from a negative binomial distribution.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBNEGB(\(p, n, m\))

Arguments

\(p\)

is a numeric probability of success parameter.

Range \(0 \leq p \leq 1\)

Data type DOUBLE

\(n\)

is an integer number of successes parameter.

Range \(n \geq 1\)

Data type INTEGER

\(m\)

is a positive integer random variable, the number of failures.

Range \(m \geq 0\)

Data type INTEGER
Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success $p$ and number of successes $n$, is less than or equal to $m$.

To compute the probability that an observation is equal to a given value $m$, compute the difference of two probabilities from the negative binomial distribution for $m$ and $m-1$.

Example

The following statement illustrates the PROBNEGB function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probnegb(0.5,2,1);</td>
<td>0.5</td>
</tr>
</tbody>
</table>

PROBNORM Function

Returns the probability from the standard normal distribution.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBNORM(x)

Arguments

$x$

is a numeric random variable.

Data type DOUBLE

Details

The PROBNORM function returns the probability that an observation from the standard normal distribution is less than or equal to $x$.

Note: PROBNORM is the inverse of the PROBIT function.

Example

The following statement illustrates the PROBNORM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probnorm(1.96);</td>
<td>0.97500210485177</td>
</tr>
</tbody>
</table>
PROBT Function

Returns the probability from a t distribution.

**Category:** Probability

**Returned data type:** DOUBLE

**Syntax**

\[
\text{PROBT}(x, df[, nc])
\]

**Arguments**

\(x\) is a numeric random variable.

- **Data type:** DOUBLE

\(df\) is a numeric degrees of freedom parameter.

- **Range:** \(df > 0\)
- **Data type:** DOUBLE

\(nc\) is a numeric noncentrality parameter.

- **Data type:** DOUBLE

**Details**

The PROBT function returns the probability that an observation from a Student's t distribution, with degrees of freedom \(df\) and noncentrality parameter \(nc\), is less than or equal to \(x\). This function accepts a noninteger degree of freedom parameter \(df\). If the optional parameter, \(nc\), is not specified or has the value 0, the value that is returned is from the central Student's t distribution.

The significance level of a two-tailed t test is given by the following equation.

\[
p = (1 - \text{probt(abs}(x), df)) \times 2;
\]

**Example**

The following statement illustrates the PROBT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{probt}(0.9, 5));</td>
<td>0.79531439982768</td>
</tr>
</tbody>
</table>
PRXCHANGE Function

Performs a pattern-matching replacement.

**Category:** Character String Matching  
**Returned data type:** CHAR

**Syntax**

PRXCHANGE(perl-regular-expression | regular-expression-id, times, source)

**Arguments**

- **perl-regular-expression**
  - specifies a character constant, variable, or expression with a value that is a Perl regular expression.
  - Data type: CHAR

- **regular-expression-id**
  - specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.
  - Restriction: If you use this argument, you must also use the PRXPARSE function.
  - Data type: INTEGER

- **times**
  - is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.
  - Data type: INTEGER
  - Tip: If the value of times is –1, then matching patterns continue to be replaced until the end of source is reached.

- **source**
  - specifies a character constant, variable, or expression that you want to search.
  - Data type: CHAR

**Details**

**The Basics**

If you use regular-expression-id, the PRXCHANGE function searches the source with the regular-expression-id that is returned by PRXPARSE. It returns the value in source with the changes that were specified by the regular expression. If there is no match, PRXCHANGE returns the unchanged value in source.

If you use perl-regular-expression, PRXCHANGE searches the source with the perl-regular-expression, and you do not need to call PRXPARSE. You can use PRXCHANGE with a perl-regular-expression in a WHERE clause and in PROC SQL.
Note: The following restrictions apply to PRX functions in DS2:

- Only m, i, s, and x can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers p, o, c, a, and l are not supported.
- The matching mode modifier g is supported.

For more information about pattern matching, see “Pattern Matching Using Perl Regular Expressions (PRX)” in SAS Functions and CALL Routines: Reference.

Compiling a Perl Regular Expression

If perl-regular-expression is a constant or if it uses the /o option, then the Perl regular expression is compiled once, and each use of PRXCHANGE reuses the compiled expression. If perl-regular-expression is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXCHANGE.

Note: The compile-once behavior occurs when you use PRXCHANGE in a DS2 environment, in a WHERE clause, or in PROC SQL. For all other uses, the perl-regular-expression is recompiled for each call to PRXCHANGE.

Performing a Match

Perl regular expressions consist of characters and special characters that are called metacharacters. When performing a match, SAS searches a source string for a substring that matches the Perl regular expression that you specify.

To view a short list of Perl regular expression metacharacters that you can use when you build your code, see the table “Tables of Perl Regular Expression (PRX) Metacharacters” in SAS Functions and CALL Routines: Reference. You can find a complete list of metacharacters on the Perl website.

Examples

Example 1: Changing the Order of First and Last Names By Using the DATA Step

The following example changes the order of first and last names.

```sas
/* Create a table that contains a list of names. */
proc ds2;
data names;
dcl char(32) name;
method init();
    name='Jones, Fred'; output;
    name='Kavich, Kate'; output;
    name='Turley, Ron'; output;
    name='Dulix, Yolanda'; output;
end;
enddata;
run;
quit;

/* Reverse last and first names */
proc ds2;
data ReversedNames;
method run();
set names;
```
Example 2: Changing a Matched Pattern to a Fixed Value

This example locates a pattern in a variable and replaces the variable with a predefined value. The example finds the phone numbers and replaces them with an informational message.

```sas
/* Create table that contains confidential information. */
proc ds2;
data a;
  dcl char(95) text;
  method run();
  text='The phone number for Ed is (801)443-9876 but not until tonight.';
  output;
  text='He can be reached at (910)998-8762 tomorrow for testing purposes.';
  output;
end;
enddata;
run;
quit;
proc print data=a;
run;
quit;
/* Locate confidential phone numbers and replace them with message */
/* indicating that they have been removed. */
proc ds2;
data b;
  method run();
  set a;
  text=prxchange('s/\(\[2-9\]\d\d\) \d\d\d\d/\*PHONE NUMBER REMOVED*/', -1, text);
  put text=;
```

Output 23.7  Results from the PRXCHANGE Function

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fred Jones</td>
</tr>
<tr>
<td>2</td>
<td>Kate Kavich</td>
</tr>
<tr>
<td>3</td>
<td>Ron Turley</td>
</tr>
<tr>
<td>4</td>
<td>Yolanda Dulix</td>
</tr>
</tbody>
</table>

Chapter 23 • DS2 Functions
end;
enddata;
run;
quit;

Output 23.8  Results Before Changing a Matched Pattern to a Fixed Value

![](image1)

Output 23.9  Results from Changing a Matched Pattern to a Fixed Value

![](image2)

See Also

Functions:

- “PRXMATCH Function” on page 715
- “PRXPAREN Function” on page 718
- “PRXPARSE Function” on page 719
- “PRXPOSN Function” on page 721

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character String Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Syntax

PRXMATCH(perl-regular-expression, source)

Arguments

perl-regular-expression

specifies a character constant, variable, or expression with a value that is a Perl regular expression.
**Details**

**The Basics**
When you use `perl-regular-expression`, the PRXMATCH function searches `source` with the `perl-regular-expression` and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero.

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL.

*Note:* The following restrictions apply to PRX functions in DS2:

- Only `m`, `i`, `s`, and `x` can be used in the PRX form `/…/…/` that can be preceded or followed by a single character.
- The matching mode modifiers `p`, `o`, `c`, `a`, and `l` are not supported.

For more information about pattern matching, see “Pattern Matching Using Perl Regular Expressions (PRX)” in *SAS Functions and CALL Routines: Reference*.

**Compiling a Perl Regular Expression**
If `perl-regular-expression` is a constant or if it uses the `/o` option, then the Perl regular expression is compiled once and each use of PRXMATCH reuses the compiled expression. If `perl-regular-expression` is not a constant and if it does not use the `/o` option, then the Perl regular expression is recompiled for each call to PRXMATCH.

**Examples**

**Example 1: Finding the Position of a Substring By Using a Perl Regular Expression**
The following example uses a Perl regular expression to search a string (Hello world) for a substring (world) and to return the position of the substring in the string.

```sas
data _null_
  dcl double position;
  method run();
  position=prxmatch('/world/', 'Hello world!');
  put position;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
7
```
Example 2: Extracting a ZIP Code

The following example searches each observation in a table for a nine-digit ZIP code, and writes those observations to the table ZipPlus4.

Note: The backslash (\) must be preceded by another backslash (\) that acts as an escape character.

data ZipCodes (overwrite=yes);
  dcl char(16) name;
  dcl char(10) zip;
  method run();
    name='Jonathan'; zip='32532-2343'; output;
    name='Seth'; zip='85030'; output;
    name='Kim'; zip='39204'; output;
    name='Samuel'; zip='93849-3843'; output;
  end;
  enddata;
run;

proc print data=ZipCodes; run;
quit;

data ZipPlus4 (overwrite=yes);
  method run();
    set zipcodes;
    select;
      when (prxmatch('/\d{5}-\d{4}/', zip))
        output;
    end;
  end;
  enddata;
runc;
quit;

proc print data=ZipPlus4; run;
quit;

Output 23.10  Original ZipCodes Table Output

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jonathan</td>
<td>32532-2343</td>
</tr>
<tr>
<td>2</td>
<td>Seth</td>
<td>85030</td>
</tr>
<tr>
<td>3</td>
<td>Kim</td>
<td>39204</td>
</tr>
<tr>
<td>4</td>
<td>Samuel</td>
<td>93849-3843</td>
</tr>
</tbody>
</table>
Output 23.11  Nine-digit ZIP Code Output

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jonathan</td>
<td>32532-2343</td>
</tr>
<tr>
<td>2</td>
<td>Samuel</td>
<td>93849-3843</td>
</tr>
</tbody>
</table>

PRXPAREN Function

Returns the last bracket match for which there is a match in a pattern.

Category: Character String Matching
Restriction: Use with the PRXPARSE function.
Returned data type: INTEGER

Syntax

PRXPAREN(regular-expression-id)

Arguments

regular-expression-id
specifies a numeric variable with a value that is an identification number that is returned by the PRXPARSE function.

Data type  INTEGER

Details

The PRXPAREN function is useful in identifying which part of a pattern matched.

Note: The following restrictions apply to PRX functions in DS2:

- Only m, i, s, and x can be used in the PRX form /…/…/ that can be preceded or followed by a single character.
- The matching mode modifiers p, o, c, a, and l are not supported.

For more information about pattern matching, see “Pattern Matching Using Perl Regular Expressions (PRX)” in SAS Functions and CALL Routines: Reference.

Comparisons

The Perl regular expression (PRX) functions manipulate strings that match patterns. To see a list and short description of these functions, see the Character String Matching category in “SAS Functions and CALL Routines by Category” in SAS Functions and CALL Routines: Reference.
Example

The following example uses Perl regular expressions and writes the results to the SAS log.

```
proc ds2;
data _null_;  
dcl double position;  
method run();
    ExpressionID=prxparse('/(magazine)|(book)|(newspaper)/');  
    position=prxmatch(ExpressionID, 'find book here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;  
    position=prxmatch(ExpressionID, 'find magazine here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;  
    position=prxmatch(ExpressionID, 'find newspaper here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;
end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
Matched paren 2
Matched paren 1
Matched paren 3
```

See Also

Functions:
- “PRXCHANGE Function” on page 712
- “PRXMATCH Function” on page 715
- “PRXPARSE Function” on page 719
- “PRXPOSN Function” on page 721

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

- **Category:** Character String Matching
- **Restriction:** Use with other Perl regular expressions.
- **Returned data type:** INTEGER

**Syntax**

```
regular-expression-id=PRXPARSE(perl-regular-expression)
```

---

The following example uses Perl regular expressions and writes the results to the SAS log.

```
proc ds2;
data _null_;  
dcl double position;  
method run();
    ExpressionID=prxparse('/(magazine)|(book)|(newspaper)/');  
    position=prxmatch(ExpressionID, 'find book here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;  
    position=prxmatch(ExpressionID, 'find magazine here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;  
    position=prxmatch(ExpressionID, 'find newspaper here');  
    if position then paren=prxparen(ExpressionID);  
        put 'Matched paren ' paren;
end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
Matched paren 2
Matched paren 1
Matched paren 3
```

See Also

Functions:
- “PRXCHANGE Function” on page 712
- “PRXMATCH Function” on page 715
- “PRXPARSE Function” on page 719
- “PRXPOSN Function” on page 721

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

- **Category:** Character String Matching
- **Restriction:** Use with other Perl regular expressions.
- **Returned data type:** INTEGER

**Syntax**

```
regular-expression-id=PRXPARSE(perl-regular-expression)
```
Arguments

regular-expression-id
is a numeric pattern identifier that is returned by the PRXPARSE function.

Data type INTEGER

perl-regular-expression
specifies a character, constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

Details

The Basics
The PRXPARSE function returns a pattern identifier number that is used by other Perl functions to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

PRXPARSE uses metacharacters in constructing a Perl regular expression. To view a table of common metacharacters, see “Tables of Perl Regular Expression (PRX) Metacharacters” in SAS Functions and CALL Routines: Reference.

Note: The following restrictions apply to PRX functions in DS2:

• Only m, i, s, and x can be used in the PRX form /.../ that can be preceded or followed by a single character.

• The matching mode modifiers p, o, c, a, and l are not supported.

For more information about pattern matching, see “Pattern Matching Using Perl Regular Expressions (PRX)” in SAS Functions and CALL Routines: Reference.

Compiling a Perl Regular Expression
If perl-regular-expression is a constant, the Perl regular expression is compiled only once. Successive calls to PRXPARSE will not cause a recompile, but will return the regular-expression-id for the regular expression that was already compiled. This behavior simplifies the code because you do not need to use an initialization block (IF _N_ =1) to initialize Perl regular expressions.

Examples

Example 1: Compiling a Perl Regular Expression
The following example uses PRXPARSE to compile the Perl regular expression.

data _null_
method init();
declare double patternID position;

    /*! Use PRXPARSE to compile the Perl regular expression. */
    patternID=prxparse('/world/');
    /*! Use PRXMATCH to find the position of the pattern match. */
    position=prxmatch(patternID, 'Hello world!');
    put position=
end;
Example 2: Using PRXPARSE to Reverse First and Last Names

data _null_;  
method init();  
  declare double patternID position;  
  declare char(32) names[4];  
  declare int i;  

  names := ('Jones, Fred  
    'Kavich, Kate  
    'Turley, Ron  
    'Dulix, Yolanda');  

  /* Reverse last and first names */  
do i = 1 to dim(names);  
    names[i]=prxchange('s/(\w+), (\w+)/$2 $1/', -1, names[i]);  
    put names[i];  
  end;  
end;  
enddata; run;

Fred Jones
Kate Kavich
Ron Turley
Yolanda Dulix

See Also

Functions:

• “PRXCHANGE Function” on page 712
• “PRXMATCH Function” on page 715
• “PRXPAREN Function” on page 718
• “PRXPOSN Function” on page 721

PRXPOSN Function

Returns a character string that contains the value for a capture buffer.

  Category: Character String Matching
  Returned data type: CHAR

Syntax

PRXPOSN(regular-expression-id, capture-buffer, source)
Arguments

regular-expression-id

specifies a numeric variable with a value that is a pattern identifier that is returned by
the PRXPARSE function.

Restriction

Data type INTEGER

capture-buffer

is a numeric constant, variable, or expression that identifies the capture buffer for
which to retrieve a value:

• If the value of capture-buffer is zero, PRXPOSN returns the entire match.
• If the value of capture-buffer is between 1 and the number of open parentheses in
  the regular expression, then PRXPOSN returns the value for that capture buffer.
• If the value of capture-buffer is greater than the number of open parentheses,
  then PRXPOSN returns a missing value.

Data type INTEGER

source

specifies the text from which to extract capture buffers.

Data type CHAR

Details

The PRXPOSN function uses the results of PRXMATCH or PRXCHANGE to return a
capture buffer. A match must be found by one of these functions for PRXPOSN to return
meaningful information.

Note: The following restrictions apply to PRX functions in DS2:

• Only m, i, s, and x can be used in the PRX form /…/…/ that can be preceded or
  followed by a single character.
  • The matching mode modifiers p, o, c, a, and l are not supported.

For more information about pattern matching, see “Pattern Matching Using Perl Regular
Expressions (PRX)” in SAS Functions and CALL Routines: Reference.

Examples

Example 1: Extracting First and Last Names

The following example uses PRXPOSN to extract first and last names from a table.

```
proc ds2;
data ReversedNames;
dcl char(32) name;
method init();
   name='Jones, Fred'; output;
   name='Kavich, Kate'; output;
   name='Turley, Ron'; output;
   name='Dulix, Yolanda'; output;
end;
```
enddata;
run;
quit;

proc ds2;
data FirstLastNames (overwrite=yes);
  dcl char(16) first last;
  dcl double re;
  keep first last;
  retain re;

  method init();
    dcl varchar(32) expression;
    expression = '/\w+, \w+/';
    re=prxparse(expression);
    if missing( re ) then do;
      put 'ERROR: Invalid expression ' expression;
      stop;
    end;
  end;

  method run();
    set ReversedNames;
    if prxmatch(re, name) then do;
      last=prxposn(re, 1, name);
      first=prxposn(re, 2, name);
    end;
  enddata;
run;
quit;

proc print data=FirstLastNames;
run;
quit;

Figure 23.2  Output from PRXPOSN: First and Last Names

<table>
<thead>
<tr>
<th>Obs</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fred</td>
<td>Jones</td>
</tr>
<tr>
<td>2</td>
<td>Kate</td>
<td>Kavich</td>
</tr>
<tr>
<td>3</td>
<td>Ron</td>
<td>Turley</td>
</tr>
<tr>
<td>4</td>
<td>Yolanda</td>
<td>Dulix</td>
</tr>
</tbody>
</table>
Example 2: Extracting Names When Some Names Are Invalid

The following example creates a table that contains a list of names. Observations that have only a first name or only a last name are invalid. PRXPOSN extracts the valid names from the table, and writes the names to the table NEW.

```
proc ds2;
data old;
dcl char(60) name;
method init();
   name='Judith S Reaveley'; output;
   name='Ralph F. Morgan'; output;
   name='Jess Ennis'; output;
   name='Carol Echols'; output;
   name='Kelly Hansen Huff'; output;
   name='Judith'; output;
   name='Nick'; output;
   name='Jones'; output;
end;
enddata;
run;
quit;
```

```
proc ds2;
data new;
dcl char(40) first middle last;
keep first middle last;
method run();
   re=prxparse('/(\S+)\s+([\^\s]+)?(\S+)/i');
   set old;
   if prxmatch(re, name) then do;
      first=prxposn(re, 1, name);
      middle=prxposn(re, 2, name);
      last=prxposn(re, 3, name);
      output;
   end;
end;
enddata;
run;
quit;
proc print data=new;
run;
quit;
```
PUT Function

Returns a value using a specified format.

**Category:** Special

**Returned data type:** NVARCHAR

**Syntax**

`PUT(expression, format)`

**Arguments**

*expression*

specifies any valid expression.

**Requirement**

`position` must be a positive number.

**Data type**

DOUBLE, DATE, TIME, TIMESTAMP, CHAR, NCHAR

**See**

Chapter 13, “DS2 Expressions,” on page 93

*format.*

specifies either a DS2 format or a user-defined format that you want applied to `expression`.

To override the default alignment, you can add an alignment specification to a `format`:

See Also

Functions:

- “PRXCHANGE Function” on page 712
- “PRXMATCH Function” on page 715
- “PRXPAREN Function” on page 718
- “PRXPARSE Function” on page 719

---

**Figure 23.3** Output of Valid Names

<table>
<thead>
<tr>
<th>Obs</th>
<th>first</th>
<th>middle</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Judith</td>
<td>S</td>
<td>Reaveley</td>
</tr>
<tr>
<td>2</td>
<td>Ralph</td>
<td>F.</td>
<td>Morgan</td>
</tr>
<tr>
<td>3</td>
<td>Jess</td>
<td></td>
<td>Ennis</td>
</tr>
<tr>
<td>4</td>
<td>Carol</td>
<td>Echols</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Kelly</td>
<td>Hansen</td>
<td>Huff</td>
</tr>
</tbody>
</table>
Details

If a value is not specified for the format width or decimal specification, DS2 uses the default values for that format.

If expression is not a valid data type for the format type (either numeric or character), DS2 converts expression to a valid data type for format, with these exceptions:

- date and time expressions are converted to a SAS date, time, or datetime DOUBLE value for numeric formats, and converted to NCHAR for character string formats
- when the format is a binary character format such as $BINARY, $HEX or $OCTAL, expressions with a data type of DOUBLE are converted to NCHAR
- an error is issued when an expression with a data type of VARBINARY is used with a numeric format that does not produce a data type of VARBINARY

When DS2 converts an expression's data type in an assignment statement, the result is left-aligned.

You can use the PUT function to convert a numeric value to a character value and to convert a date, time, or timestamp value to a SAS date/time value.

Comparisons

The PUT function and the PUT statement have similar behavior. However, the PUT statement directs its results to the SAS log whereas the PUT function returns an NCHAR value containing the result of formatting its argument.

Example

The following statements illustrate the PUT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>put (17180, date7.);</td>
<td>15JAN07</td>
</tr>
<tr>
<td>put (‘AB’, $binary.);</td>
<td>010000101000010</td>
</tr>
<tr>
<td>a=35436745.3354</td>
<td>35,436,745.3354</td>
</tr>
<tr>
<td>b=put (a, comma20.4 -L);</td>
<td>35,436,745.3354</td>
</tr>
<tr>
<td>b=put (a, comma20.4)</td>
<td>35,436,745.3354</td>
</tr>
</tbody>
</table>
Statements | Results
--- | ---
proc format;
  value abc 1="Yes" 2="No";
run;
proc ds2;
data;
  method init{};
    dcl double d;
    d=1;
    x=put(d,abc.);
    put x=;
    d=2;
    x=put(d,abc.);
    put x=;
  end;
enddata;
run;
quit;

See Also

Functions:
- “INPUTC Function” on page 538
- “INPUTN Function” on page 539

**PVP Function**

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

**Category:** Financial

**Returned data type:** DOUBLE

**Syntax**

\[ PVP(A, c, n, K, k_0, y) \]

**Arguments**

- **\( A \)** specifies the par value.
  - **Range:** \( A > 0 \)
  - **Data type:** DOUBLE

- **\( c \)** specifies the nominal per-year coupon rate, expressed as a fraction.
Range \( 0 \leq c < 1 \)

Data type DOUBLE

\( n \)
specifies the number of coupons per year.

Range \( n > 0 \)

Data type INTEGER

\( K \)
specifies the number of remaining coupons.

Range \( K > 0 \)

Data type INTEGER

\( k_0 \)
specifies the time from the present date to the first coupon date, expressed in terms of the number of years.

Range \( 0 < k_0 \leq \frac{1}{n} \)

Data type DOUBLE

\( y \)
specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Range \( y > 0 \)

Data type DOUBLE

Details

The PVP function is based on the following relationship:

\[
P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}
\]

The following relationships apply to the preceding equation:

- \( t_k = nk_0 + k - 1 \)
- \( c(k) = \frac{c}{n} \) for \( k = 1, \ldots, K - 1 \)
- \( c(K) = \left(1 + \frac{c}{n}\right)A \)

Example

data _null_

method run();

dcl double p;
p=pvp(1000,.01,4,14,.33/2,.10);
put p;
The value that is returned is 743.167613519067.

**QTR Function**

Returns the quarter of the year from a SAS date value.

**Syntax**

\[
\text{QTR}(date)
\]

**Arguments**

*date*

specifies any valid expression that represents a SAS date value.

Data type: DOUBLE

**Details**

The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

For more information about how DS2 handles date and time values, see Chapter 14, “Dates and Times in DS2,” on page 111.

**Example**

The following statements illustrate the QTR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=17180;</td>
<td>14JAN07</td>
</tr>
<tr>
<td>b=put(a,date7.);</td>
<td></td>
</tr>
<tr>
<td>c=qtr(a);</td>
<td>1</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

- “YYQ Function” on page 850
QUOTE Function

Adds double quotation marks to a character value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

**Syntax**

QUOTE(expression)

**Arguments**

expression

specifies any valid expression that evaluates to a character string.

<table>
<thead>
<tr>
<th>Data type</th>
<th>NCHAR</th>
</tr>
</thead>
</table>

See Chapter 13, “DS2 Expressions,” on page 93

**Details**

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the SAS log.

A string of characters enclosed in double quotation marks is a DS2 identifier and not a character constant. The double quotation marks become part of the identifier. Quoted identifiers cannot be used to create column names in an output table.

**Example**

The following statements illustrate the QUOTE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='A&quot;B'; b=quote(a);</td>
<td>&quot;A&quot;&quot;B&quot;</td>
</tr>
<tr>
<td>a='A'B'; b=quote(a);</td>
<td>&quot;A'B&quot;</td>
</tr>
</tbody>
</table>
RANBIN Function

Returns a random variate from a binomial distribution.

Category: Random Number

Returned data type: DOUBLE

Syntax

RANBIN(seed, n, p)

Arguments

seed

is a numeric constant, variable, or expression. If seed ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31} - 1$

Data type INTEGER

See “Seed Values” in SAS Functions and CALL Routines: Reference for more information about seed values

n

is a numeric constant, variable, or expression that specifies the number of independent Bernoulli trials parameter.

Range $n > 0$

Data type INTEGER

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 < p < 1$

Data type DOUBLE

Details

The RANBIN function returns a variate that is generated from a binomial distribution with mean $np$ and variance $np(1-p)$. If $n \leq 50$, $np \leq 5$, or $n(1-p) \leq 5$, an inverse transform method applied to a RANUNI uniform variate is used. If $n > 50$, $np > 5$, and
\( n(1-p) > 5 \), the normal approximation to the binomial distribution is used. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

**See Also**

Functions:

- “RAND Function” on page 733

---

**RANCAU Function**

Returns a random variate from a Cauchy distribution.

**Category:** Random Number

**Returned data type:** DOUBLE

**Syntax**

RANCAU(seed)

**Arguments**

*seed* is a numeric constant, variable, or expression.

<table>
<thead>
<tr>
<th>Range</th>
<th>seed &lt; ( 2^{31} -1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

**Note** If \( seed \leq 0 \), the time of day is used to initialize the seed stream.

**See** “Seed Values” in *SAS Functions and CALL Routines: Reference* for more information about seed values

**Details**

The RANCAU function returns a variate that is generated from a Cauchy distribution with location parameter 0 and scale parameter 1. An acceptance-rejection procedure applied to RANUNI uniform variates is used. If \( u \) and \( v \) are independent uniform \((-1/2, 1/2)\) variables and \( u^2 + v^2 \leq 1/4 \), then \( u/v \) is a Cauchy variate. A Cauchy variate \( X \) with location parameter \( \alpha \) and scale parameter \( \beta \) can be generated:

\[
x = \alpha + \beta \cdot \text{rancau(seed)};
\]

**See Also**

Functions:

- “RAND Function” on page 733
RAND Function

Generates pseudo-random numbers from a distribution that you specify.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Random Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

RAND(dist, parm-1, ...parm-k)

Arguments

*dist*

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli (p. 735)</td>
<td>BERNOULLI</td>
</tr>
<tr>
<td>Beta (p. 735)</td>
<td>BETA</td>
</tr>
<tr>
<td>Binomial (p. 735)</td>
<td>BINOMIAL</td>
</tr>
<tr>
<td>Cauchy (p. 736)</td>
<td>CAUCHY</td>
</tr>
<tr>
<td>Chi-Square (p. 736)</td>
<td>CHISQUARE</td>
</tr>
<tr>
<td>Erlang (p. 737)</td>
<td>ERLANG</td>
</tr>
<tr>
<td>Exponential (p. 737)</td>
<td>EXPONENTIAL</td>
</tr>
<tr>
<td>F (p. 737)</td>
<td>F</td>
</tr>
<tr>
<td>Gamma (p. 738)</td>
<td>GAMMA</td>
</tr>
<tr>
<td>Geometric (p. 738)</td>
<td>GEOMETRIC</td>
</tr>
<tr>
<td>Hypergeometric (p. 739)</td>
<td>HYPERGEOMETRIC</td>
</tr>
<tr>
<td>Lognormal (p. 739)</td>
<td>LOGNORMAL</td>
</tr>
<tr>
<td>Negative Binomial (p. 740)</td>
<td>NEGBINOMIAL</td>
</tr>
<tr>
<td>Normal (p. 740)</td>
<td>NORMAL</td>
</tr>
<tr>
<td>Poisson (p. 741)</td>
<td>POISSON</td>
</tr>
</tbody>
</table>
### Distribution

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>T (p. 741)</td>
<td>T</td>
</tr>
<tr>
<td>Tabled (p. 742)</td>
<td>TABLE</td>
</tr>
<tr>
<td>Triangular (p. 742)</td>
<td>TRIANGLE</td>
</tr>
<tr>
<td>Uniform (p. 743)</td>
<td>UNIFORM</td>
</tr>
<tr>
<td>Weibull (p. 743)</td>
<td>WEIBULL</td>
</tr>
</tbody>
</table>

**Data type**: CHAR

**Note**: Except for T and F, you can minimally identify any distribution by its first four characters.

**parm-1, …parm-k**

are *shape, location, or scale* parameters that are appropriate for the specific distribution.

**See**: “Details” on page 734

### Details

**Generating Pseudo-Random Numbers**

The RAND function generates pseudo-random numbers from various continuous and discrete distributions. Wherever possible, the simplest form of the distribution is used.

The RAND function uses the Mersenne-Twister pseudo-random number generator (RNG) that was developed by Matsumoto and Nishimura (1998). The pseudo-random number generator has a very long period ($2^{19937} - 1$) and very good statistical properties. The period is a Mersenne prime, which contributes to the naming of the RNG. The algorithm is a twisted generalized feedback shift register (TGFSR) that explains the latter part of the name. The TGFSR gives the RNG a very high order of equidistribution (623-dimensional with 32-bit accuracy), which means that there is a very small correlation between successive vectors of 623 pseudo-random numbers.

The RAND function is started with a single seed. However, the state of the process cannot be captured by a single seed. You cannot stop and restart the generator from its stopping point.

If the initial seed is exactly divisible by 8192, the RAND function uses the 2002 initialization algorithm (Matsumoto and Nishimura, 2002) (See “References” in SAS Functions and CALL Routines: Reference). Otherwise, RAND uses the 1998 initialization algorithm.

**Reproducing a Pseudo-Random Number Stream**

If you want to create reproducible streams of pseudo-random numbers, then use the STREAMINIT function to specify a seed value for pseudo-random number generation. Use the STREAMINIT function once per data program before any invocation of the RAND function. For more information, see the example in “STREAMINIT Function” on page 789.
**Duplicate Values in the Mersenne-Twister RNG Algorithm**

The Mersenne-Twister RNG algorithm has an extremely long period, but this does not imply that large random samples are devoid of duplicate values. The RAND function returns at most $2^{32}$ distinct values. In a random uniform sample of size $10^5$, the chance of drawing at least one duplicate is greater than 50%. The expected number of duplicates in a random uniform sample of size $M$ is approximately $M^2/2^{32}$ when $M$ is much less than $2^{32}$. For example, you should expect about 115 duplicates in a random uniform sample of size $M=10^6$. These results are consequences of the famous “birthday matching problem” in probability theory.

**Bernoulli Distribution**

$x = \text{RAND('BERNOULLI', } p\text{)}$

**Arguments**

$x$

is an observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} 
1 & p = 0, x = 0 \\
 p^x(1-p)^{1-x} & 0 < p < 1, x = 0, 1 \\
 1 & p = 1, x = 1
\end{cases}$$

Range $x = 0, 1$

$p$

is a numeric probability of success.

Range $0 \leq p \leq 1$

**Beta Distribution**

$x = \text{RAND('BETA', } a, b\text{)}$

**Arguments**

$x$

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

Range $0 < x < 1$

$a$

is a numeric shape parameter.

Range $a > 0$

$b$

is a numeric shape parameter.

Range $b > 0$

**Binomial Distribution**

$x = \text{RAND('BINOMIAL', } p, n\text{)}$
Arguments

\( x \)

is an integer observation from the distribution with the following probability density function:

\[
 f(x) = \begin{cases} 
 1 & p = 0, x = 0 \\
 \binom{n}{x} p^x (1-p)^{n-x} & 0 < p < 1, x = 0, \ldots, n \\
 1 & p = 1, x = n 
\end{cases}
\]

Range \( x = 0, 1, \ldots, n \)

\( p \)

is a numeric probability of success.

Range \( 0 \leq p \leq 1 \)

\( n \)

is an integer parameter that counts the number of independent Bernoulli trials.

Range \( n = 1, 2, \ldots \)

Note: You can achieve the same results by using the RANBIN function. The following examples use a seed value of 45 and are equivalent:

- \text{streaminit}(45); \\
a=rand('binomial',.75,10); \\
a=ranbin(45,.75,10);

**Cauchy Distribution**

\( x = \text{RAND}('\text{CAUCHY}') \)

Arguments

\( x \)

is an observation from the distribution with the following probability density function:

\[
 f(x) = \frac{1}{\pi \left(1 + x^2\right)}
\]

Range \( -\infty < x < \infty \)

Note: You can achieve the same results by using the RANCAU function. The following examples use a seed value of 45 and are equivalent:

- \text{streaminit}(45); \\
a=rand('cauchy'); \\
a=rancau(45);

**Chi-Square Distribution**

\( x = \text{RAND}('\text{CHISQUARE}', \text{df}) \)

Arguments
\[ x \]

is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{\Gamma\left(\frac{df}{2}\right)^{\frac{df}{2}}}{\Gamma\left(\frac{df}{2}ight)} x^{\frac{df}{2} - 1} e^{-x/2}
\]

Range \( x > 0 \)

\( df \)

is a numeric degrees of freedom parameter.

Range \( df > 0 \)

**Erlang Distribution**

\( x = \text{RAND}('ERLANG', a) \)

**Arguments**

\( x \)

is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}
\]

Range \( x > 0 \)

\( a \)

is an integer numeric shape parameter.

Range \( a = 1, 2, ... \)

**Exponential Distribution**

\( x = \text{RAND}('EXPONENTIAL') \)

**Arguments**

\( x \)

is an observation from the distribution with the following probability density function:

\[
f(x) = e^{-x}
\]

Range \( x > 0 \)

**Note:** You can achieve the same results by using the RANEXP function. The following examples use a seed value of 45 and are equivalent:

- \( \text{streaminit}(45); \)
  - \( a = \text{rand('exponential')}; \)

- \( a = \text{ranexp}(45); \)

**F Distribution**

\( x = \text{RAND}('F', n, d) \)
Arguments

\( x \)

is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{\Gamma\left(\frac{n+d}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\Gamma\left(\frac{d}{2}\right)} x^{n/2} d^{d/2} (n+d)^{n/2 - 1}
\]

Range \( x > 0 \)

\( n \)

is a numeric numerator degrees of freedom parameter.

Range \( n > 0 \)

\( d \)

is a numeric denominator degrees of freedom parameter.

Range \( d > 0 \)

**Gamma Distribution**

\( x = \text{RAND}('\text{GAMMA}', a) \)

Arguments

\( x \)

is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}
\]

Range \( x > 0 \)

\( a \)

is a numeric shape parameter.

Range \( a > 0 \)

**Note:** You can achieve the same results by using the \text{RANGAM} function. The following examples use a seed value of 45 and are equivalent:

- \text{streaminit}(45); \n  a=\text{rand}('\text{gamma}',7.25);
- a=\text{rangam}(45,7.25);

**Geometric Distribution**

\( x = \text{RAND}('\text{GEOMETRIC}', p) \)

Arguments

\( x \)

is an integer count that denotes the number of trials that are needed to obtain one success. \( x \) is an integer observation from the distribution with the following probability density function:
\[
f(x) = \begin{cases} 
(1 - p)^{x-1} p & 0 < p < 1, x = 1, 2, \ldots \\
1 & p = 1, x = 1 
\end{cases}
\]

\[P\]
is a numeric probability of success.

\[0 < p \leq 1\]

**Hypergeometric Distribution**

\[x = \text{RAND}('HYPER', N, R, n)\]

**Arguments**

\[x\]
is an integer observation from the distribution with the following probability density function:

\[
f(x) = \binom{R}{x} \binom{N-R}{n-x} \binom{N}{n}
\]

\[N\]
is an integer population size parameter.

\[N = 1, 2, \ldots\]

\[R\]
is an integer number of items in the category of interest.

\[R = 0, 1, \ldots, N\]

\[n\]
is an integer sample size parameter.

\[n = 1, 2, \ldots, N\]

The hypergeometric distribution is a mathematical formalization of an experiment in which you draw \(n\) balls from an urn that contains \(N\) balls, \(R\) of which are red. The hypergeometric distribution is the distribution of the number of red balls in the sample of \(n\).

**Lognormal Distribution**

\[x = \text{RAND}('LOGNORMAL')\]

**Arguments**

\[x\]
is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{e^{-\ln^2(x)/2}}{x\sqrt{2\pi}}
\]
Negative Binomial Distribution

\( x = \text{RAND}(\text{NEGBINOMIAL}', p, k) \)

**Arguments**

- \( x \) is an integer observation from the distribution with the following probability density function:

\[
f(x) = \begin{cases} 
\frac{(x + k - 1)}{(k - 1)} (1 - p)^x p^k & 0 < p < 1, x = 0, 1, \ldots \\
1 & p = 1, x = 0
\end{cases}
\]

Range \( x > 0 \)

\( k \) is an integer parameter that is the number of successes. However, non-integer \( k \) values are allowed as well.

Range \( k = 1, 2, \ldots \)

\( p \) is a numeric probability of success.

Range \( 0 < p \leq 1 \)

The negative binomial distribution is the distribution of the number of failures before \( k \) successes occur in sequential independent trials, all with the same probability of success, \( p \).

Normal Distribution

\( x = \text{RAND}(\text{NORMAL}', [\theta, \lambda]) \)

**Arguments**

- \( x \) is an observation from the normal distribution with a mean of \( \theta \) and a standard deviation of \( \lambda \) that has the following probability density function:

\[
f(x) = \frac{1}{\sqrt{2\pi} \lambda} \exp\left(-\frac{(x - \theta)^2}{2\lambda^2}\right)
\]

Range \( -\infty < x < \infty \)

\( \theta \) is the mean parameter.

Default \( 0 \)

\( \lambda \) is the standard deviation parameter.

Default \( 1 \)
Range  \( \lambda > 0 \)

**Note:** You can achieve the same results by using the RANNOR function. The following examples use a seed value of 45 and are equivalent:

- `streaminit(45); a=rand('normal');`
- `a=rannor(45);`

**Poisson Distribution**

\[ x = \text{RAND}('POISSON', m) \]

**Arguments**

- \( x \) is an integer observation from the distribution with the following probability density function:
  \[
  f(x) = \frac{m^x e^{-m}}{x!}
  \]
  Range  \( x = 0, 1, ... \)

- \( m \) is a numeric mean parameter.
  Range  \( m > 0 \)

**Note:** You can achieve the same results by using the RANPOI function. The following examples use a seed value of 45 and are equivalent:

- `streaminit(45); a=rand('poisson',.75);`
- `a=ranpoi(45,.75);`

**T Distribution**

\[ x = \text{RAND}('T', df) \]

**Arguments**

- \( x \) is an observation from the distribution with the following probability density function:
  \[
  f(x) = \frac{1}{\sqrt{df} \pi r^{(df+1)/2}} \left(1 + \frac{x^2}{r} \right)^{-\frac{df+1}{2}}
  \]
  Range  \(-\infty < x < \infty\)

- \( df \) is a numeric degrees of freedom parameter.
  Range  \( df > 0 \)
**Tabled Distribution**

\[ x = \text{RAND}('\text{TABLE}', p_1, p_2, ...) \]

**Arguments**

- **x**
  - is an integer observation from one of the following distributions:

  If \( \sum_{i=1}^{n} p_i < 1 \), then \( x \) is an observation from this probability density function:

  \[ f(i) = p_i, \quad i = 1, 2, ..., n \]
  
  and

  \[ f(n + 1) = 1 - \sum_{i=1}^{n} p_i \]

  If \( \sum_{i=1}^{n} p_i \geq 1 \) for some index \( n \), then \( x \) is an observation from this probability density function:

  \[ f(i) = p_i, \quad i = 1, 2, ..., n - 1 \]
  
  and

  \[ f(n) = 1 - \sum_{i=1}^{n-1} p_i \]

- **p1, p2, ...**
  - are numeric probability values.

**Range**

\[ 0 \leq p_1, p_2, ... \leq 1 \]

**Restriction**

The maximum number of probability parameters depends on your operating environment, but the maximum number of parameters is at least 32,767.

The tabled distribution takes on the values 1, 2, ..., \( n \) with specified probabilities.

**Note:** By using the FORMAT statement, you can map the set \{1, 2, ..., \( n \}\) to any set of \( n \) or fewer elements.

**Note:** You can achieve the same results by using the RANTBL function. The following examples use a seed value of 45 and are equivalent:

- \( \text{streaminit}(45); \)
  - \( a = \text{rand}('\text{table}', .2, .5); \)
- \( a = \text{rantbl}(45, .2, .5); \)

**Triangular Distribution**

\[ x = \text{RAND}('\text{TRIANGLE}', h) \]

**Arguments**

- **x**
  - is an observation from the distribution with the following probability density function:
In this equation, $0 \leq h \leq 1$.

### Range

$0 \leq x \leq h$

### Note

The distribution can be easily shifted and scaled.

$h$

is the horizontal location of the peak of the triangle.

### Range

$0 \leq h \leq 1$

### Uniform Distribution

$x = \text{RAND('UNIFORM')}$

### Arguments

$x$

is an observation from the distribution with the following probability density function:

$$f(x) = 1$$

### Range

$0 < x < 1$

The uniform pseudo-random number generator that the RAND function uses is the Mersenne-Twister (Matsumoto and Nishimura 1998). This generator has a period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy. This algorithm underlies the generators for the other available distributions in the RAND function.

### Note

You can achieve the same results by using the RANUNI function. The following examples use a seed value of 45 and are equivalent:

- `streaminit(45); a=rand('uniform');`
- `a=ranuni(45);`

### Weibull Distribution

$x = \text{RAND('WEIBULL', a, b)}$

### Arguments

$x$

is an observation from the distribution with the following probability density function:
\[ f(x) = \frac{a}{b^a} x^{a-1} e^{-\left(\frac{x}{b}\right)^a} \]

Range \( x \geq 0 \)

\( a \)

is a numeric shape parameter.

Range \( a > 0 \)

\( b \)

is a numeric scale parameter.

Range \( b > 0 \)

---

**RANEXP Function**

Returns a random variate from an exponential distribution.

**Category:** Random Number

**Returned data type:** DOUBLE

**Syntax**

\[ \text{RANEXP}(seed) \]

**Arguments**

\( seed \)

is a numeric constant, variable, or expression with an integer value. If \( seed \leq 0 \), the time of day is used to initialize the seed stream.

Range \( seed < 2^{31} - 1 \)

Data type INTEGER

See “Seed Values” in *SAS Functions and CALL Routines: Reference*

**Details**

The RANEXP function returns a variate that is generated from an exponential distribution with parameter 1. An inverse transform method applied to a RANUNI uniform variate is used.

An exponential variate \( X \) with parameter \( \text{LAMBDA} \) can be generated:

\[ x = \text{ranexp}(seed) / \text{lambda}; \]

An extreme value variate \( X \) with location parameter \( \text{ALPHA} \) and scale parameter \( \text{BETA} \) can be generated:

\[ x = \text{alpha} - \text{beta} \times \log(\text{ranexp}(seed)); \]
A geometric variate $X$ with parameter $P$ can be generated as follows:

$$x = \text{floor}\left(\frac{-\text{ranexp}(\text{seed})}{\log(1-p)}\right);$$

**See Also**

Functions:

- “RAND Function” on page 733

---

**RANGAM Function**

Returns a random variate from a gamma distribution.

**Category:** Random Number  
**Returned data type:** DOUBLE

**Syntax**

$\text{RANGAM}(\text{seed}, a)$

**Arguments**

**seed**  
is a numeric constant, variable, or expression.  
**Range** $seed < 2^{31}-1$  
**Data type** INTEGER  
**Note** If $seed \leq 0$, the time of day is used to initialize the seed stream.  
**See** “Seed Values” in *SAS Functions and CALL Routines: Reference* for more information about seed values

**a**  
is a numeric constant, variable, or expression that specifies the shape parameter.  
**Range** $a > 0$

**Details**

The RANGAM function returns a variate that is generated from a gamma distribution with parameter $a$. For $a > 1$, an acceptance-rejection method due to Cheng (1977) is used. (See “References” in *SAS Functions and CALL Routines: Reference*). For $a \leq 1$, an acceptance-rejection method due to Fishman is used (1978, Algorithm G2) (See “References” in *SAS Functions and CALL Routines: Reference*).

A gamma variate $X$ with shape parameter $\alpha$ and scale $\beta$ can be generated:

$$x = \beta \times \text{rangam}(\text{seed}, \alpha);$$

If $2\alpha$ is an integer, a chi-square variate $X$ with $2\alpha$ degrees of freedom can be generated:
\[ x = 2 \cdot \text{rangam}(\text{seed}, \alpha); \]

If \( N \) is a positive integer, an Erlang variate \( X \) can be generated:

\[ x = \beta \cdot \text{rangam}(\text{seed}, N); \]

It has the distribution of the sum of \( N \) independent exponential variates whose means are \( \beta \).

And finally, a beta variate \( X \) with parameters \( \alpha \) and \( \beta \) can be generated:

\[
\begin{align*}
  y_1 &= \text{rangam}(\text{seed}, \alpha); \\
  y_2 &= \text{rangam}(\text{seed}, \beta); \\
  x &= y_1 / (y_1 + y_2);
\end{align*}
\]

### See Also

**Functions:**

- “RAND Function” on page 733

---

**RANGE Function**

Returns the difference between the largest and the smallest values.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

\[ \text{RANGE}(\text{expression}[, \ldots \text{expression}]) \]

**Arguments**

**expression**

specifies any valid expression that evaluates to a numeric value.

**Requirement**

At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

**Data type**

DOUBLE

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The RANGE function returns the difference between the largest and the smallest of the non-null or nonmissing arguments.

**Example**

The following statements illustrate the RANGE function:
RANK Function

Returns the position of a character in the ASCII or EBCDIC collating sequence.

**Category:** Character  
**Returned data type:** DOUBLE

**Syntax**

\[
\text{RANK(expression)}
\]

**Arguments**

*expression*

specifies any valid expression that evaluates to a character string.

Data type: NCHAR  
See: Chapter 13, “DS2 Expressions,” on page 93

**Details**

The RANK function returns an integer that represents the position of the first character in the character expression.

**Example**

The following statement illustrates the RANK function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=rank('A');</td>
<td>65</td>
<td>65</td>
<td>193</td>
</tr>
</tbody>
</table>
See Also

Functions:

• “BYTE Function” on page 408

RANNOR Function

Returns a random variate from a normal distribution.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Random Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

RANNOR(seed)

Arguments

seed

is a numeric constant, variable, or expression.

Range               seed < 2^{31}−1
Data type           INTEGER
Note                If seed ≤ 0, the time of day is used to initialize the seed stream.
See                 “Seed Values” in SAS Functions and CALL Routines: Reference for more information about seed values

Details

The RANNOR function returns a variate that is generated from a normal distribution with mean 0 and variance 1. The Box-Muller transformation of RANUNI uniform variates is used.

A normal variate X with mean MU and variance S2 can be generated with this code:

\[ x = MU + \sqrt{S2} \times \text{rannor}(seed); \]

A lognormal variate X with mean \( \exp(MU + S2/2) \) and variance \( \exp(2*MU + 2*S2) - \exp(2*MU + S2) \) can be generated with this code:

\[ x = \exp(MU + \sqrt{S2} \times \text{rannor}(seed)); \]

See Also

Functions:

• “RAND Function” on page 733
RANPOI Function

Returns a random variate from a Poisson distribution.

**Category:** Random Number

**Returned data type:** DOUBLE

**Syntax**

RANPOI(seed, m)

**Arguments**

*seed*

is a numeric constant, variable, or expression.

- **Range:** $seed < 2^{31} - 1$
- **Data type:** INTEGER

**Note**

If $seed \leq 0$, the time of day is used to initialize the seed stream.

**See**

“Seed Values” in SAS Functions and CALL Routines: Reference for more information about seed values

*m*

is a numeric constant, variable, or expression that specifies the mean of the distribution.

- **Range:** $m \geq 0$
- **Data type:** DOUBLE

**Details**

The RANPOI function returns a variate that is generated from a Poisson distribution with mean $m$. For $m < 85$, an inverse transform method applied to a RANUNI uniform variate is used (Fishman 1976). (See “References” in SAS Functions and CALL Routines: Reference.) For $m \geq 85$, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new $m$).

**See Also**

**Functions:**

- “RAND Function” on page 733
RANTBL Function

Returns a random variate from a tabled probability distribution.

Category: Random Number
Returned data type: DOUBLE

Syntax

RANTBL(seed, p1...pi...pn)

Arguments

seed

is a numeric constant, variable, or expression.
- Range: seed < 2^{31}−1
- Data type: INTEGER
- Note: If seed ≤ 0, the time of day is used to initialize the seed stream.
- See: For more information about seed values, see “Seed Values” in SAS Functions and CALL Routines: Reference.

pi

is a numeric constant, variable, or expression.
- Range: 0 ≤ pi ≤ 1 for 0 < i ≤ n
- Data type: DOUBLE

Details

The RANTBL function returns a variate that is generated from the probability mass function defined by p1 through pn. An inverse transform method applied to a RANUNI uniform variate is used. RANTBL returns

1 with probability p1
2 with probability p2
...

n with probability pn

n + 1 with probability 1 − \sum_{i=1}^{n} pi if \sum_{i=1}^{n} pi ≤ 1
If, for some index \( j < n \), \( \sum_{i=1}^{j} p_i \geq 1 \), RANTBL returns only the indices 1 through \( j \) with the probability of occurrence of the index \( j \) equal to \( 1 - \sum_{i=1}^{j-1} p_i \).

Let \( n=3 \) and \( P_1, P_2, \) and \( P_3 \) be three probabilities with \( P_1+P_2+P_3=1 \), and \( M_1, M_2, \) and \( M_3 \) be three variables. The variable \( X \) in these statements

```plaintext
array m{3} m1-m3;
x=m{rantbl(seed, of p1-p3)};
```

will be assigned one of the values of \( M_1, M_2, \) or \( M_3 \) with probabilities of occurrence \( P_1, \) \( P_2, \) and \( P_3, \) respectively.

### See Also

#### Functions:

- “RAND Function” on page 733

---

**RANTRI Function**

Returns a random variate from a triangular distribution.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Random Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

#### Syntax

RANTRI\((seed, h)\)

#### Arguments

**seed**

- is a numeric constant, variable, or expression.

  - **Range**: \( seed < 2^{31}-1 \)
  - **Data type**: INTEGER
  - **Note**: If \( seed \leq 0 \), the time of day is used to initialize the seed stream.
  - **See**: “Seed Values” in *SAS Functions and CALL Routines: Reference* for more information about seed values

**h**

- is a numeric constant, variable, or expression that specifies the mode of the distribution.

  - **Range**: \( 0 < h < 1 \)
  - **Data type**: DOUBLE
Details

The RANTRI function returns a variate that is generated from the triangular distribution on the interval (0,1) with parameter \( h \), which is the modal value of the distribution. An inverse transform method applied to a RANUNI uniform variate is used.

A triangular distribution \( X \) on the interval \((A,B)\) with mode \( C \), where \( A \leq C \leq B \), can be generated:

\[
x = (b-a) \cdot \text{rantri}(seed, (c-a)/(b-a)) + a;
\]

See Also

Functions:

- “RAND Function” on page 733

RANUNI Function

Returns a random variate from a uniform distribution.

**Category:** Random Number

**Returned data type:** DOUBLE

**Syntax**

\[
\text{RANUNI}(seed)
\]

**Required Argument**

\( seed \)

is a numeric constant, variable, or expression.

**Range**

\( seed < 2^{31}-1 \)

**Data type**

INTEGER

**Note**

If \( seed \leq 0 \), the time of day is used to initialize the seed stream.

**See**

“Seed Values” in SAS Functions and CALL Routines: Reference for more information about seed values

Details

The RANUNI function returns a number that is generated from the uniform distribution on the interval (0,1) using a prime modulus multiplicative generator with modulus \( 2^{31} - 1 \) and multiplier 397204094 (Fishman and Moore 1982) (See “References” in SAS Functions and CALL Routines: Reference).

You can use a multiplier to change the length of the interval and an added constant to move the interval. For example,

\[
\text{random_variate} = a \cdot \text{ranuni}(seed) + b;
\]

returns a number that is generated from the uniform distribution on the interval \((b,a+b)\).
REPEAT Function

Repeats a character expression.

**Category:** Character

**Returned data type:** VARCHAR, NVARCHAR

**Syntax**

\[
\text{REPEAT}(\text{expression}, \text{n})
\]

**Arguments**

- **expression** specifies any valid expression that evaluates to a character string.
  - **Data type:** CHAR, NCHAR
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **n** specifies the number of times to repeat *expression*.
  - **Restriction:** \( n \geq 0 \)
  - **Data type:** INTEGER

**Details**

The REPEAT function returns a character value consisting of the first argument repeated \( n \) times. Thus, the first argument appears \( n+1 \) times in the result.

**Example**

The following statement illustrates the REPEAT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a=repeat('ONE',2);</code></td>
<td>ONEONEONE</td>
</tr>
</tbody>
</table>
**REVERSE Function**

Reverses a character expression.

**Category:** Character  
**Returned data type:** NCHAR

**Syntax**

\[ \text{REVERSE}(\text{expression}) \]

**Arguments**

expression  
 specifies any valid expression that evaluates to a character string.

**Data type**  
NCHAR

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**Details**

The REVERSE function returns a character value with the last character in the expression is the first character in the result, the next-to-last character in the expression is the second character in the result, and so on.

*Note:* Trailing blanks in the expression become leading blanks in the result.

**Example**

The following statement illustrates the REVERSE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=reverse('xyz ');</td>
<td>zyx</td>
</tr>
</tbody>
</table>

**RIGHT Function**

Right aligns a character expression.

**Category:** Character  
**Returned data type:** NCHAR
**Syntax**

```none
RIGHT(expression)
```

**Arguments**

`expression`

specifies any valid expression that evaluates to a character string.

**Data type**

NCHAR

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Details**

The RIGHT function returns an argument with trailing blanks moved to the start of the value. The argument's length does not change.

**Example**

The following statements illustrate the RIGHT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='Due Date  ';</td>
<td>Due Date</td>
</tr>
<tr>
<td>b=put(a,$10.);</td>
<td>Due Date</td>
</tr>
<tr>
<td>c=put(right(a),$10.);</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “COMPRESS Function” on page 437
- “LEFT Function” on page 602
- “TRIM Function” on page 821

---

**RMS Function**

Returns the root mean square.

**Category:**

Descriptive Statistics

**Returned data type:**

DOUBLE

**Syntax**

```none
RMS(expression [, ...expression])
```
**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value.

- **Data type**: DOUBLE
- **See**: Chapter 13, “DS2 Expressions,” on page 93

**Details**

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the root mean square of the non-null or nonmissing values.

Let \( n \) be the number of arguments with non-null or nonmissing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The root mean square is calculated as follows.

\[
\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}
\]

**Example**

The following statements illustrate the RMS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x1 = \text{rms}(1,7); )</td>
<td>5</td>
</tr>
<tr>
<td>( x2 = \text{rms}(.,1,5,11); )</td>
<td>7</td>
</tr>
</tbody>
</table>

**ROUND Function**

Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

- **Category**: Truncation
- **Returned data type**: DOUBLE

**Syntax**

\[
\text{ROUND}(\text{expression} [, \text{rounding-unit}])
\]

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value, to be rounded.

- **Data type**: DOUBLE
**Details**

**Basic Concepts**
The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The results might not be an exact multiple of the second argument.

**Differences between Binary and Decimal Arithmetic**
Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic. As the following example shows, if \(a\) is a float and \(b\) is a REAL, there is a difference between the two values.

```sas
data _null_;
dcl float a diff;
dcl real b;
method run();
a=0.3;
b=3*0.1;
diff=a-b;
put a=;
prompt a=;
put diff=;
end;
enddata;
run;
```

The following lines are written to the SAS log:

```
a=0.3
b=0.3
diff=-1.192092896618E-8
```

**Operating Environment Information**
The example above was executed in the Windows environment. If you use other operating environments, the results will be slightly different.

**The Effects of Rounding**
Rounding by definition finds an exact multiple of the rounding unit that is closest to the value to be rounded. For example, 0.33 rounded to the nearest tenth equals 3*0.1 or 0.3 in decimal arithmetic. In binary arithmetic, 0.33 rounded to the nearest tenth equals 3*0.1, and not 0.3, because 0.3 is not an exact multiple of one tenth in binary arithmetic.
The ROUND function returns the value that is based on decimal arithmetic, even though this value is sometimes not the exact, mathematically correct result. In the example ROUND(0.33, 0.1), ROUND returns 0.3 and not 3*0.1.

**Expressing Binary Values**

If the characters "0.3" appear as a constant in a DS2 program, the value is computed as 3/10. To be consistent with the standard informat, ROUND(0.33, 0.1) computes the result as 3/10, and the following statement produces the results that you would expect.

```plaintext
if round(x, 0.1) = 0.3 then
    ... more DS2 statements ...
```

However, if you use the variable Y instead of the constant 0.3, as the following statement shows, the results might be unexpected depending on how the variable Y is computed.

```plaintext
if round(x, 0.1) = y then
    ... more DS2 statements ...
```

If ROUND reads Y as the characters "0.3" using the standard informat, the result is the same as if a constant 0.3 appeared in the IF statement. If ROUND reads Y with a different informat, or if a program other than SAS reads Y, then there is no guarantee that the characters "0.3" would produce a value of exactly 3/10. Imprecision can also be caused by computation involving numbers that do not have exact binary representations, or by porting tables from one operating environment to another that has a different floating-point representation.

If you know that Y is a decimal number with one decimal place, but are not certain that Y has exactly the same value as would be produced by the standard informat, it is better to use the following statement:

```plaintext
if round(x, 0.1) = round(y, 0.1) then
    ... more DS2 statements ...
```

**Testing for Approximate Equality**

You should not use the ROUND function as a general method to test for approximate equality. Two numbers that differ only in the least significant bit can round to different values if one number rounds down and the other number rounds up. Testing for approximate equality depends on how the numbers have been computed. If both numbers are computed to high relative precision, you could test for approximate equality by using the ABS and the MAX functions, as the following example shows.

```plaintext
if abs(x-y) <= 1e-12 * max( abs(x), abs(y) ) then
    ... more DS2 statements ...
```

**Producing Expected Results**

In general, ROUND(expression, rounding-unit) produces the result that you expect from decimal arithmetic if the result has no more than nine significant digits and any of the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. ¹
- The result that you expect from decimal arithmetic has no more than four decimal places.

¹ If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a power of 10 in at most the three or four least significant bits.
For example:

data rounding;
  method run();
    d1 = round(1234.56789,100) - 1200;
    d2 = round(1234.56789,10) - 1230;
    d3 = round(1234.56789,1) - 1235;
    d4 = round(1234.56789,.1) - 1234.6;
    d5 = round(1234.56789,.01) - 1234.57;
    d6 = round(1234.56789,.001) - 1234.568;
    d7 = round(1234.56789,.0001) - 1234.5679;
    d8 = round(1234.56789,.00001) - 1234.56789;
    d9 = round(1234.56789,.1111) - 1234.5432;
    /* d10 has too many decimal places in the value for */
    /* rounding-unit.                                   */
    d10 = round(1234.56789,.11111) - 1234.54321;
  end;
enddata;
run;

The following output shows the results.

Output 23.12  Results of Rounding Based on the Value of the Rounding Unit

<table>
<thead>
<tr>
<th>Obs</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.012345</td>
</tr>
</tbody>
</table>

Operating Environment Information
The example above was executed in a z/OS environment. If you use other operating environments, the results will be slightly different.

When the Rounding Unit Is the Reciprocal of an Integer
When the rounding unit is the reciprocal of an integer, the ROUND function computes the result by dividing by the integer. Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit. Here is an example:

data rounding2;
  drop pi unit;
  method run();
    pi = arcos(-1);
    unit=1/7.;
    d1=round(pi,unit) - 22/7.;
    d2=round(pi, unit) - 22*unit;
  end;
enddata;
run;

The following output shows the results.

---

1  ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.
The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

**Computing Results in Special Cases**

The ROUND function computes the result by multiplying an integer by the rounding unit when all of the following conditions are true:

- The rounding unit is not an integer.
- The rounding unit is not a power of 10.
- The rounding unit is not the reciprocal of an integer.
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```sas
data _null_
  method run();
  difference=round(1234.56789,.11111) - 11111*.11111;
  put difference=;
end;
enddata;
run;
```

The following line is written to the SAS log:

```
  difference=0
```

Operating Environment Information

The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

**Computing Results When the Value Is Halfway between Multiples of the Rounding Unit**

When the value to be rounded is approximately halfway between two multiples of the rounding unit, the ROUND function rounds up the absolute value and restores the original sign. For example:

```sas
data test;
  method run ();
  do i=8 to 17;
    value=0.5 - 10**(-i);
    round=round(value);
    output;
  end;
run;
```
end;
do i=8 to 17;
    value=-0.5 + 10**(-i);
    round=round(value);
    output;
    end;
end;
enddata;
run;

The following output shows the results.
**Output 23.14**  Results of Rounding When Values Are Halfway between Multiples of the Rounding Unit

<table>
<thead>
<tr>
<th>Obs</th>
<th>I</th>
<th>VALUE</th>
<th>ROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>0.50000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0.50000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0.50000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>0.50000</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>0.50000</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>0.50000</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>0.50000</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>0.50000</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0.50000</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>0.50000</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>-0.50000</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>-0.50000</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>-0.50000</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>-0.50000</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>-0.50000</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>13</td>
<td>-0.50000</td>
<td>-1</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>-0.50000</td>
<td>-1</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>-0.50000</td>
<td>-1</td>
</tr>
<tr>
<td>19</td>
<td>16</td>
<td>-0.50000</td>
<td>-1</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
<td>-0.50000</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Operating Environment Information**

The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

The approximation is relative to the size of the value to be rounded, and is computed in a manner that is shown in the following example. This example code will not always produce results exactly equivalent to the ROUND function.

```sas
data testfile;
  method run();
```
do i = 1 to 17;
  value = 0.5 - 10**(-i);
  epsilon = min(1e-6, value * 1e-12);
  temp = value + .5 + epsilon;
  fraction = modz(temp, 1);
  round = temp - fraction;
  output;
end;
end;
enddata;
run;

proc print data=testfile noobs;
  format value 19.16;
run;

Comparisons

The ROUND function is the same as the ROUNDE function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

The ROUNDEZ function returns a multiple of the rounding unit without trying to make the result match the result that is computed with decimal arithmetic.

Example

The following example compares the results that are returned by the ROUND function with the results that are returned by the ROUNDE function. The output was generated from the UNIX operating environment.

data results;
  method run();
    do x=0 to 4 by .25;
      Rounde=rounde(x);
      Round=round(x);
      output;
    end;
  end;
enddata;
run;

proc print data=results noobs;
run;

The following output shows the results.
## The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>ROUND</th>
<th>ROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1.50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1.75</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2.25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>2.50</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>2.75</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>3.00</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>3.25</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>3.50</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>3.75</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>4.00</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

## See Also

### Functions:
- “CEIL Function” on page 419
- “CEILZ Function” on page 420
- “FLOOR Function” on page 505
- “FLOORZ Function” on page 506
- “INT Function” on page 541
- “INTZ Function” on page 581
- “ROUND Function” on page 765

### Output 23.15  Results That Are Returned by the ROUND and ROUNDE Functions
ROUNDE Function

Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.

- **Category:** Truncation
- ** Returned data type:** DOUBLE

**Syntax**

ROUNDE(expression [, rounding-unit])

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value and that is to be rounded.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

*rounding-unit*

is a positive, numeric expression that specifies the rounding unit.

- **Default:** 1
- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The ROUNDE function rounds the first argument to the nearest multiple of the second argument.

**Comparisons**

The ROUNDE function is the same as the ROUND function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

**Example**

The following example compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```plaintext
data results
  method run();
    do x=0 to 4 by .25;
```
The following output shows the results.

**Output 23.16  Results That Are Returned by the ROUND and ROUND Functions**

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>ROUND</th>
<th>ROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1.50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1.75</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2.25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>2.50</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>2.75</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>3.00</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>3.25</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>3.50</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>3.75</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>4.00</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “CEIL Function” on page 419
ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

**Category:** Truncation

**Returned data type:** DOUBLE

### Syntax

```
ROUNDZ(expression [, rounding-unit])
```

### Arguments

**expression**

specifies any valid expression that evaluates to a numeric value.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**rounding-unit**

specifies any valid expression that evaluates to a numeric expression and that specifies the rounding unit.

- **Default:** 1
- **Requirement:** Only positive values are valid.
- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

### Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument.

### Comparisons

The ROUNDZ function is the same as the ROUND function with these exceptions:
• ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument. ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples.

• When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUNDZ does not fuzz the result. ROUND performs extra computations, called fuzzing, to try to make the result agree with decimal arithmetic.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions
The following example compares the results that are returned by the ROUNDZ and the ROUND function.

data test;
  method run();
    do i=10 to 17;
      Value=2.5 - 10**(-i);
      Roundz=roundz(value);
      Round=round(value);
      output;
    end;
    do i=16 to 12 by -1;
      value=2.5 + 10**(-i);
      roundz=roundz(value);
      round=round(value);
      output;
    end;
  end;
enddata;
run;

proc print data=test;
  format value 19.16;
quit;

The following output shows the results.
Output 23.17  Results That Are Returned by the ROUNDZ and ROUND Functions

<table>
<thead>
<tr>
<th>Obs</th>
<th>i</th>
<th>Value</th>
<th>Roundz</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2.4999999999999900000000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>2.49999999999999000000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>2.49999999999999000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>2.49999999999999000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2.49999999999999000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>2.49999999999999000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>2.50000000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>2.50000000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>2.50000000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>2.50000000000000000000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2.50000000000000000000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>2.50000000000000000000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>2.50000000000000000000</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Example 2: Sample Output from the ROUNDZ Function

The following statements illustrate the ROUNDZ function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=223.456;</td>
<td>223.000000</td>
</tr>
<tr>
<td>b=roundz(a,1);</td>
<td></td>
</tr>
<tr>
<td>c=roundz(b,9.5);</td>
<td></td>
</tr>
<tr>
<td>var2=223.456;</td>
<td>223.460000</td>
</tr>
<tr>
<td>x=roundz(var2,.01);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td></td>
</tr>
<tr>
<td>x=roundz(223.456,100);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>200.000000</td>
</tr>
</tbody>
</table>
Statement | Results
--- | ---
\[ x = \text{roundz}(223.456,.3); \] \[ \text{put } x \ 9.5; \] | 223.50000

See Also

Functions:
- “ROUND Function” on page 756
- “ROUNDE Function” on page 765

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Category: Financial
Returned data type: DOUBLE

Syntax

\[ \text{SAVINGS}\left(\text{base-date, initial-deposit-date, deposit-amount, deposit-number, deposit-interval, compounding-interval, date-1, rate-1[, ...date-n, rate-n]}\right) \]

Arguments

\textit{base-date}

specifies the value that is returned is the balance of the savings at the base date.

Requirement \( \text{Base-date is a SAS date.} \)

Data type \( \text{DOUBLE} \)

\textit{initial-deposit-date}

specifies the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

Requirement \( \text{Initial-deposit-date is a SAS date.} \)

Data type \( \text{DOUBLE} \)

\textit{deposit-amount}

specifies the value of each deposit. All deposits are assumed constant.

Data type \( \text{DOUBLE} \)

\textit{deposit-number}

specifies the number of deposits.

Data type \( \text{INTEGER} \)
**deposit-interval**

specifies the frequency at which deposits are made.

Requirement  *Deposit-interval* is a SAS interval.

Data type  CHAR

**compounding-interval**

specifies the compounding interval.

Requirement  *Compounding-interval* is a SAS interval.

Data type  CHAR

**date**

specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement  *Date* is a SAS date.

Data type  DOUBLE

**rate**

specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type  DOUBLE

**Details**

The following details apply to the SAVINGS function:

- The values for rates must be between –99 and 120.
- *Deposit-interval* cannot be 'CONTINUOUS'.
- The list of date-rate pairs does not need to be in chronological order.
- When multiple rate changes occur on a single date, the SAVINGS function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the initial-deposit-date and the base-date.

**Example**

- If you deposit $300 monthly for two years into an account that compounds quarterly at an annual rate of 4%, the balance of the account after five years can be expressed as follows:

```sas
data _null_;  
method run();  
bd=to_double(date'2005-01-01');  
idd=to_double(date'2000-01-01');  
d=to_double(date'2000-01-01');  
amount_base1=savings(bd, idd, 300, 24, 'month', 'qtr', d, 4.00);  
put amount_base1;  
end;  
enddata;
```
run;

The following line is written to the SAS log.

8458.79415896917

• If the interest rate increases by a quarter-point each year, then the balance of the account could be expressed as follows:

```sas
data _null_;  
  method run();
  bd= to_double(date'2005-01-01');
  idd= to_double(date'2000-01-01');
  d1= to_double(date'2000-01-01');
  d2= to_double(date'2001-01-01');
  d3= to_double(date'2002-01-01');
  d4= to_double(date'2003-01-01');
  d5= to_double(date'2004-01-01');
  amount_base2 = savings(bd, idd, 300, 24, 'month', 'qtr', d1, 4.00, d2, 4.25, d3, 4.50, d4, 4.75, d5, 5.0);
  put amount_base2;
  end;
enddata;
run;

The following line is written to the SAS log.

8623.09024586998

• To determine the balance after one year of deposits, the following statement sets amount_base3 to the desired balance:

```sas
data _null_;  
  method run();
  bd= to_double(date'2001-01-01');
  idd= to_double(date'2000-01-01');
  d= to_double(date'2000-01-01');
  amount_base3 = savings(bd, idd, 300, 24, 'month', 'qtr', d, 4.00, d2, 4.25, d3, 4.50, d4, 4.75, d5, 5.0);
  put amount_base3;
  end;
enddata;
run;

The following line is written to the SAS log.

3978.69037121739

The SAVINGS function ignores deposits after the base date, so the deposits after the reference date do not affect the value that is returned.

---

**SCAN Function**

Returns the \( n \)th word from a character expression.

- **Category:** Character
- **Returned data type:** NCHAR
Syntax

SCAN(expression, n [, delimiters])

Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

n
is a nonzero numeric expression that specifies the number of the word in the character expression that you want SCAN to select. The following rules apply:

• If n is positive, SCAN counts words from left to right in the character string.
• If n is negative, SCAN counts words from right to left in the character string.
• If n is greater than the number of words in expression, SCAN returns a blank value.

delimiters
specifies any valid expression that evaluates to a character string and that SCAN uses as word separators in the expression.

Default

Requirement If delimiter is a constant, enclose delimiter in single quotation marks.

Interactions ASCII default delimiters are: blank ! $ % & ( ) * + - . / ; < |. In environments without the ^ character, SCAN uses the ~ character instead.

EBCDIC default delimiters are: blank ! $ % & ( ) * + - . / ; < ¬ | ¢.

Data type NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

Details

Leading delimiters before the first word in the expression do not effect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

In DS2, if the SCAN function returns a value to a variable that has not yet been given a length, then that variable is given the length of the first argument. If you need the SCAN function to assign to a variable a value that is different from the length of the first argument, then you should use a DECLARE statement for that variable before the statement that uses the SCAN function.

Example

The following statements illustrate the SCAN function:
Statements | Results
---|---
expr='ABC.DEF (X=y)'; word=scan(expr,3); | X=Y

scan ('ABC.DEF (X=Y) ', -3); | ABC

---

**SEC Function**

Returns the secant.

- **Category:** Trigonometric
- **Returned data type:** DOUBLE

**Syntax**

SEC(*expression*)

**Arguments**

*expression*

specifies any valid expression that evaluates to a numeric value that expressed in radians.

**Restriction**

*expression* cannot be an odd multiple of PI/2.

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Comparisons**

The SEC function is related to the COS function:

\[ \sec(x) = \frac{1}{\cos(x)} \]

**Example**

The following statements illustrate the SEC function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=sec(0.5);</td>
<td>1.13949392732454</td>
</tr>
<tr>
<td>y=sec(0);</td>
<td>1</td>
</tr>
<tr>
<td>z=sec(3.14159/3);</td>
<td>1.99999693590391</td>
</tr>
</tbody>
</table>
SECOND Function

Returns the second from a SAS time or datetime value.

**Category:** Date and Time

**Returned data type:** DOUBLE

**Syntax**

```
SECOND(time | datetime)
```

**Arguments**

- **time**
  - Specifies any valid expression that represents a SAS time value.
  - **Data type:** DOUBLE
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

- **datetime**
  - Specifies any valid expression that represents a SAS datetime value.
  - **Data type:** DOUBLE
  - **See:** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The SECOND function produces a numeric value that represents a specific second of the minute. The result can be any number that is \( \geq 0 \) and \(< 60\).

**Example**

The following statements illustrate the SECOND function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=hms(3,19,24);</td>
<td>11964</td>
</tr>
<tr>
<td>b=second(a);</td>
<td>24</td>
</tr>
</tbody>
</table>
### Statements and Results

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = hms(6, 25, 65);</code></td>
<td>23165</td>
</tr>
<tr>
<td><code>s = second(a);</code></td>
<td>5</td>
</tr>
<tr>
<td><code>a = hms(3, 19, 60);</code></td>
<td>12000</td>
</tr>
<tr>
<td><code>b = second(a);</code></td>
<td>0</td>
</tr>
</tbody>
</table>

### See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

### Functions

- “HOUR Function” on page 532
- “MINUTE Function” on page 628

---

### SHA256HEX Function

Returns the result of the message digest of a specified string and converts the string to hexadecimal representation.

**Category:** Character

**Notes:**
- The SHA256HEX function verifies the data integrity and authentication of a message.
- UTF-8 text is recommended for the SHA256HEX function arguments to ensure consistency across encodings.

**Syntax**

```
SHA256HEX('string', string_indicator);
```

**Arguments**

- **string**
  - specifies any valid expression that evaluates to a character string.
  - **Data type:** CHAR, NCHAR
  - See Chapter 13, “DS2 Expressions,” on page 93

- **string_indicator**
  - indicates whether the argument `string` is regular characters or hexadecimal representation characters.
    - 0 indicates that the expression in the argument `string` is regular characters.
    - 1 indicates that the expression in the argument `string` is hexadecimal representation characters.
Note: There must be an even number of hexadecimal representation characters, and they must all be between 0–9, a–f, A–F. Blanks in the hexadecimal representation string are ignored.

Details

The Basics
The SHA256HEX function converts a string, based on the SHA256 algorithm, to a 256-bit hash value. Then, the function converts the data to a hexadecimal representation format.

z/OS Specifics
In the z/OS operating environment, because the SHA256HEX function might be operating on EBCDIC data, the message digest is different from the ASCII equivalent. For example, SHA256HEX('ABC') on an EBCDIC system means that SHA256HEX receives the bytes 'C1C2C3', and the digest is 5202BF40821662BF1AD79C958056775D9D6BF8AA1C00492BCA8556B02772F. On an ASCII system, 'ABC' is '414243', and the digest is B5D405C3F466FA91FE2CC6AB79232A1A57C9104F7A26E716E0A1E2789DF78.

Using the SHA256HEX Function
You can use the SHA256HEX function to track changes in your data sets. The SHA256HEX function can generate a digest of a set of column values in a table record. This digest could be treated as the signature of the record and be used to track changes that are made to the record. If the digest from the new record matches the existing digest of a table record, then the two records are the same. If the digest is different, then a column value in the record has changed. The new changed record could then be added to the table along with a new surrogate key because the record represents a change to an existing keyed value.

The SHA256HEX function can be useful when you are developing shell scripts or Perl programs for software installation, file comparison, and detection of file corruption and tampering.

Comparisons
The SHA256 function does not format its own output, so you must use the $BINARYw. or $HEXw. formats to view readable results. The SHA256HEX function formats its output, so you do not have to use the $BINARY or $HEX formats.

Example: Generating Results with the SHA256HEX Function
This example generates results that are returned by the SHA256HEX function.

data _null_;  
dcl char y z;  
method run();  
y=sha256hex('abc');  
z=sha256hex('access method');  
put y=;  
put z=;  
end;  
enddata;
run;

For ASCII systems, the following lines are written to the SAS log.

```
y=BA7816BF8F01CFEA414140DE5DAE223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15D8824560EDC471EBB40A83BA6D1259B1605915
```

For EBCDIC systems, the following lines are written to the SAS log.

```
y=B58EA6D31995A4D8CE092EB718DDFA58B6CEF2288B41FAF1D52FF3D6D8FA01
z=D7EE088DAF6B029BADCC2DD01984867FOA3C342D60719DA7847821E3778F63
```

See Also

Functions:
- “SHA256HMACHEX Function” on page 778

### SHA256HMACHEX Function

Returns the result of the message digest of a specified string by using the Hash-based Message Authentication (HMAC) algorithm.

**Category:** Character

**Notes:** The SHA256HMACHEX function verifies the data integrity and authentication of a message. See the following article for more information about the Hash-based message authentication code (HMAC).

UTF-8 text is recommended for the SHA256HMACHEX function arguments to ensure consistency across encodings.

**Syntax**

```
SHA256HMACHEX('key', 'message' [string-indicator]);
```

**Arguments**

- **key**
  - specifies any valid expression that evaluates to a character string.
  - Data type: CHAR, NCHAR
  - See Chapter 13, “DS2 Expressions,” on page 93

- **message**
  - specifies a secret key padded to the right with extra zeros to the input block size of the hash function.
  - Data type: CHAR, NCHAR

- **string_indicator**
  - indicates whether the key and message are provided in hexadecimal representation.
the arguments key and message are not represented in hexadecimal representation.

1. the argument message is represented in hexadecimal representation.

2. the argument key is represented in hexadecimal representation.

3. the arguments key and message are represented in hexadecimal representation.

Note: This argument is useful when the SHA256HMACHEX function is being called repeatedly and the result of a previous call is used as the key in a subsequent call. The following code demonstrates this functionality:

```plaintext
length digest $64;
digest = sha256hmachex('mykey', 'mymessage', 0);
digest = sha256hmachex(digest, 'my new message', 2);
```

Data type: INTEGER

Details

The SHA256HMACHEX function converts a string, based on the SHA256 algorithm, to a 256-bit hash value.

See the following article for more information about the Hash-based message authentication code (HMAC).

Example: Generating Results with the SHA256HMACHEX Function

This example generates results that are returned by the SHA256HMACHEX function.

```plaintext
data _null_;  
  method run();  
    digest = SHA256HMACHEX('key',  
                         'The quick brown fox jumps over the lazy dog', 0);  
    if digest=  
      upcase('f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8')  
        then  
          put 'matched';  
        else  
          put 'not matched';  
    end;  
enddata;  
run;
```

matched

SIGN Function

Returns a number that indicates the sign of a numeric value expression.

Category: Mathematical
Returned data type: DOUBLE

Syntax
SIGN(expression)

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

Data type
All numeric types

See
Chapter 13, “DS2 Expressions,” on page 93

Details
The SIGN function returns the following values:

-1
if expression < 0
0
if expression = 0
1
if expression > 0.

Example
The following statements illustrate the SIGN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=sign(-5);</td>
<td>-1</td>
</tr>
<tr>
<td>a=sign(5);</td>
<td>1</td>
</tr>
<tr>
<td>a=sign(0);</td>
<td>0</td>
</tr>
</tbody>
</table>

SIN Function
Returns the trigonometric sine.

Category: Trigonometric

Returned data type: DOUBLE
Syntax
SIN(expression)

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Example
The following statements illustrate the SIN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=sin(25.6)</td>
<td>0.45044059427538</td>
</tr>
<tr>
<td>a=sin(5);</td>
<td>-0.95892427466313</td>
</tr>
</tbody>
</table>

SINH Function
Returns the hyperbolic sine.

Category: Trigonometric
Returned data type: DOUBLE

Syntax
SINH(expression)

Arguments
expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details
The SINH function returns the hyperbolic sine of the argument, which is given by the following equation.

\[ e^{\text{argument}} - e^{-\text{argument}} / 2 \]
Example

The following statements illustrate the SINH function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=sinh(0);</td>
<td>0</td>
</tr>
<tr>
<td>a=sinh(1);</td>
<td>1.1752011936438</td>
</tr>
<tr>
<td>a=sinh(-1.0);</td>
<td>-1.1752011936438</td>
</tr>
</tbody>
</table>

SKEWNESS Function

Returns the skewness.

**Category:** Descriptive Statistics

**Returned data type:** DOUBLE

**Syntax**

SKEWNESS(expression-1, expression-2, expression-3 [, …expression-n])

**Arguments**

expression  
specifies any valid expression that evaluates to a numeric value.

**Requirement**

At least three non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

**Data type**  
DOUBLE

**See**  
Chapter 13, “DS2 Expressions,” on page 93

**Details**

If all non-null or nonmissing arguments have equal values, the skewness is mathematically undefined and the SKEWNESS function returns a null or missing value.

Example

The following statements illustrate the SKEWNESS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=skeuwness(0,1,1);</td>
<td>-1.73205080756887</td>
</tr>
<tr>
<td>x2=skeuwness(2,4,6,3,1);</td>
<td>0.59012865638436</td>
</tr>
</tbody>
</table>
SLEEP Function

For a specified period of time, suspends the execution of a program that invokes this function.

**Syntax**

$$SLEEP(number-of-time-units[, time-unit])$$

**Arguments**

*number-of-time-units*

specifies any valid expression that evaluates to a numeric value and that specifies the number of units of time for which you want to suspend execution of a program.

- **Range**: \( n \geq 0 \)
- **Data type**: DOUBLE

*time-unit*

specifies the unit of time, as a power of 10, which is applied to *number-of-time-units*. For example, 1 corresponds to a second, and .001 to a millisecond.

- **Default**: 1 in a Windows PC environment, .001 in other environments
- **Data type**: DOUBLE

**Details**

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify. The maximum sleep period for the SLEEP function is 46 days.

**Examples**

**Example 1: Suspending Execution for a Specified Period of Time**

The following example delays the execution for 20 seconds:

```plaintext
data payroll;
...DS2 statements...

time_slept=sleep(20,1);
...more DS2 statements...
enddate;
```

---

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3=skewness(2,0,0);</td>
<td>1.73205080756887</td>
</tr>
</tbody>
</table>
Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution until June 15, 2011, at midnight. DS2 calculates the length of the suspension based on the target date and the date and time that the code begins to execute.

```
data budget;
  ...DS2 statements...;
  sleeptime=dhms(mdy(06,15,2011),00,00,00)-datetime();
  time_calc=sleep(sleeptime,1);
  ...more DS2 statements...;
enddata;
```

---

SMALLEST Function

Returns the \( k \)th smallest non-null or nonmissing value.

**Category:** Descriptive Statistics  

**Returned data type:** DOUBLE

**Syntax**

\[
\text{SMALLEST}(k, \text{expression}[, \ldots \text{expression}])
\]

**Arguments**

\( k \)

specifies any valid expression that evaluates to a numeric value to return.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

\( \text{expression} \)

specifies any valid expression that evaluates to a numeric value to be processed.

- **Data type:** DOUBLE
- **See:** Chapter 13, “DS2 Expressions,” on page 93

**Details**

If \( k \) is null or missing, less than zero, or greater than the number of values, the result is a null or missing value.

**Comparisons**

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores null and missing values, but ORDINAL counts null and missing values.
Example

This example compares the values that are returned by the SMALLEST function with values that are returned by the ORDINAL function.

```plaintext
proc ds2;
  conn="driver=base;
  catalog=mycatalog;
  schema=(name=mylib;primarypath='\\my-primary-path')"
  nolibs;
data comparison;
  dcl double smallest_num having label 'SMALLEST Function';
  dcl double ordinal_num having label 'ORDINAL Function';
method run();
  do k = 1 to 4;
    smallest_num = smallest(k, 456, 789, .Q, 123);
    ordinal_num  = ordinal (k, 456, 789, .Q, 123);
    output;
  end;
end;
enddata;
run;
quit;
proc print data=mylib.comparison label noobs;
  var k smallest_num ordinal_num;
  title 'Results From the SMALLEST and the ORDINAL Functions';
run;
```

Output 23.18 Comparison of Values: The SMALLEST and the ORDINAL Functions

<table>
<thead>
<tr>
<th>Obs</th>
<th>K</th>
<th>SMALLEST Function</th>
<th>ORDINAL Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>123</td>
<td>.Q</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>456</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>789</td>
<td>456</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>.</td>
<td>789</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “LARGEST Function” on page 598
- “ORDINAL Function” on page 675
- “PCTL Function” on page 676
SQLEXEC Function

Executes a FedSQL statement to create, delete, or update a table or to insert rows into a table.

**Category:** Special

**Syntax**

SQLEXEC('sql-text')

**Arguments**

'sql-text'

is a valid FedSQL statement that inserts into, updates, creates, or deletes rows from a table.

**Requirement**

The FedSQL statement must be enclosed in single quotation marks ('').

**Note**

The statement can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

**Details**

The following items apply to the SQLEXEC function:

- Use the SQLEXEC function when a FedSQL statement is to be executed only once.
- Allocate, prepare, execute, and free are performed at run time.
- The SQLEXEC function does not support parameters.
- The SQLEXEC function does not support the return of a result set. It cannot be used with a SELECT statement.
- SQLEXEC is similar to the SQL EXECUTE IMMEDIATE statement or the JDBC Statement.executeUpdate (string) method.

An SQLSTMT package enables you to execute a FedSQL more than one time, to use parameters, and to access a result set. For more information, see “Using the SQLSTMT Package” on page 172.

**Example**

Here is an example of using the SQLEXEC function:

```plaintext
tablename='testdata';
name='Jane Doe';
age=25;

s='create table ' || tablename || ' (name char(50), age int)';
sqlexec(s);

s='insert into ' || tablename || ' values(''' || name || ''',' || age ||')';
sqlexec(s);
```

See Also
“Comparing the SQLSTMT Package and the SQLEXEC Function” on page 175

SQRT Function
Returns the square root of a value.

<table>
<thead>
<tr>
<th>Category</th>
<th>Mathematical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
SQRT(expression)

Arguments
expression specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Example
The following statements illustrate the SQRT function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=sqrt(36);</td>
<td>6</td>
</tr>
<tr>
<td>a=sqrt(25);</td>
<td>5</td>
</tr>
<tr>
<td>a=sqrt(4.4);</td>
<td>2.097617663403</td>
</tr>
</tbody>
</table>

STD Function
Returns the standard deviation.

<table>
<thead>
<tr>
<th>Categories:</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descriptive Statistics</td>
</tr>
<tr>
<td>Returned data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
Syntax

\[ \text{STD}(\text{expression-1, expression-2 [\ldots expression-n]}) \]

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Requirement

At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type

DOUBLE

See

Chapter 13, “DS2 Expressions,” on page 93

Example

The following statements illustrate the STD function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{a=std(2,6);} )</td>
<td>2.82842712474619</td>
</tr>
<tr>
<td>( \text{a=std(2,6, .);} )</td>
<td>2.82842714274619</td>
</tr>
<tr>
<td>( \text{a=std(2,4,6,3,1);} )</td>
<td>1.92353840616713</td>
</tr>
</tbody>
</table>

STDERR Function

Returns the standard error of the mean.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

\[ \text{STDERR}(\text{expression-1, expression-2 [\ldots expression-n]}) \]

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Requirement

At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type

DOUBLE

See

Chapter 13, “DS2 Expressions,” on page 93
Example

The following examples illustrate the STDERR function.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(stderr(2,6));</td>
<td>2</td>
</tr>
<tr>
<td>a(stderr(2,6,.));</td>
<td>2</td>
</tr>
<tr>
<td>a(stderr(2,4,6,3,1));</td>
<td>0.86023252670426</td>
</tr>
</tbody>
</table>

STREAMINIT Function

Specifies a seed value to use for subsequent pseudo-random number generation by the RAND function.

**Category:** Random Number

**Returned data type:** DOUBLE

**Syntax**

STREAMINIT(\textit{seed})

**Arguments**

\textit{seed} is an integer seed value.

- **Range:** \textit{seed} < 2^{31} – 1
- **Data type:** INTEGER

**Tip**

If you specify a nonpositive seed, then the STREAMINIT function is ignored. Any subsequent pseudo-random number generation seeds itself from the system clock.

**Details**

If you want to create reproducible streams of pseudo-random numbers, then specify STREAMINIT before any calls to the RAND pseudo-random number function. For DS2, the STREAMINIT function must be called before the RAND function. Each DS2 program honors one STREAMINIT seed. The prevailing seed value is the one that is specified prior to the first RAND function call. For more information, see “Seed Values” in SAS Functions and CALL Routines: Reference.

**Example**

The following example shows how to specify a seed value with the STREAMINIT function to create a reproducible stream of pseudo-random numbers with the RAND function.
data random (overwrite=yes);
dcl double i x1;
method run()
  streaminit(123);
  do i=1 to 10;
    x1=rand('cauchy');
    output;
  end;
end;
enddata;
run;

proc print data=random;
  id i;
run;
quit;

**Figure 23.4**  Number String Seeded with the STREAMINIT Function

<table>
<thead>
<tr>
<th>i</th>
<th>x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.17593</td>
</tr>
<tr>
<td>2</td>
<td>3.70100</td>
</tr>
<tr>
<td>3</td>
<td>1.23427</td>
</tr>
<tr>
<td>4</td>
<td>0.49095</td>
</tr>
<tr>
<td>5</td>
<td>-0.05094</td>
</tr>
<tr>
<td>6</td>
<td>0.72496</td>
</tr>
<tr>
<td>7</td>
<td>-0.51646</td>
</tr>
<tr>
<td>8</td>
<td>7.61304</td>
</tr>
<tr>
<td>9</td>
<td>0.89784</td>
</tr>
<tr>
<td>10</td>
<td>1.69348</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**
- “RAND Function” on page 733

### STRIP Function

Returns a character string with all leading and trailing blanks removed.

**Category:**  Character
Returned data type: NCHAR

Syntax

\texttt{STRIP(expression)}

\textbf{Arguments}

\textit{expression}

specifies any valid expression that evaluates to a character string.

Data type NCHAR

See Chapter 13, “DS2 Expressions,” on page 93

Details

The \texttt{STRIP} function returns the argument with all leading and trailing blanks removed. If the argument is blank, \texttt{STRIP} returns a string with a length of zero.

If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

\textit{Note:} The \texttt{STRIP} function is useful for concatenation because the concatenation operator does not remove trailing blanks.

Comparisons

The following list compares the \texttt{STRIP} function with the \texttt{TRIM} function:

- For blank character strings, the \texttt{STRIP} and \texttt{TRIM} functions both return a string with a length of zero.
- For strings that lack leading blanks, the \texttt{STRIP} and \texttt{TRIM} functions return the same value.

Example

The following example shows the results of using the \texttt{STRIP} function to delete leading and trailing blanks.

``` SAS
data mycatalog.lengthn;
dcl char(8) string;
method init();
  string='abcd    '; output;
  string='  abcd  '; output;
  string='    abcd'; output;
  string='abcdefgh'; output;
  string='x y x'; output
end;
enddata;
run;
data mycatalog.stripstring;
dcl varchar(10) original;
dcl varchar(10) stripped;
method run();
```

set lengthn;
orignal = '*' || string || '*';
stripped = '*' || strip(string) || '*';
end;
enddata;
run;

Output 23.19  Results from the STRIP Function

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>ORIGINAL</th>
<th>STRIPPED</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>abcd</em></td>
<td><em>abcd</em></td>
<td>abcd</td>
</tr>
<tr>
<td>2</td>
<td>* abcd*</td>
<td><em>abcd</em></td>
<td>abcd</td>
</tr>
<tr>
<td>3</td>
<td><em>abcd</em></td>
<td><em>abcd</em></td>
<td>abcd</td>
</tr>
<tr>
<td>4</td>
<td><em>abcdefg</em></td>
<td><em>abcdefg</em></td>
<td>abcdefg</td>
</tr>
<tr>
<td>5</td>
<td><em>xyx</em></td>
<td><em>xyx</em></td>
<td>x y x</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “CAT Function” on page 409
- “CATS Function” on page 411
- “CATT Function” on page 413
- “CATX Function” on page 415
- “LEFT Function” on page 602
- “TRIM Function” on page 821
- “TRIMN Function” on page 822

**SUBSTR Function**

Returns a substring, allowing a result with a length of zero.

**Category:** Character  
**Returned data type:** VARCHAR, NVARCHAR

**Syntax**

`SUBSTR(character-expression, position-expression [, length-expression])`
Arguments

class expression
specifies any valid expression that evaluates to a character string.

Data type  CHAR, NCHAR

See  Chapter 13, “DS2 Expressions,” on page 93

position-expression
specifies any valid expression that evaluates to an integer and that specifies the
position of the first character in the substring.

Data type  INTEGER

See  Chapter 13, “DS2 Expressions,” on page 93

length-expression
specifies any valid expression that evaluates to an integer and that specifies the
length of the substring. If you do not specify length-expression, the SUBSTR
function returns the substring that extends from the position that you specify to the
end of the string.

Data type  INTEGER

See  Chapter 13, “DS2 Expressions,” on page 93

Details
The following information applies to the SUBSTR function:

• The SUBSTR function returns a missing or null value if character-expression is a
  missing or null value.

• The SUBSTR function returns a missing or empty string if either of the following is
  true: position-expression is a missing or null value, or position-expression is less than
  1 or greater than the length of character-expression.

• If the value for length-expression is not specified, the SUBSTR function returns
  length-expression.

• The SUBSTR function returns the substring from position-expression to the end of
  the character-expression if length-expression meets any of the following conditions:
    • is not specified
    • is less than zero
    • is a null value
    • is greater than the length of the substring from position-expression to the end of
      the character-expression

Example
The following statements illustrate the SUBSTR function:
Statements | Results
---|---
a='chsh234960b3';
substr(a,5); | 234960b3

a='chsh234960b3';
substr(a,5,6); | 234960

a='chsh234960b3';
substr(a,5,15); | 234960b3

a='chsh234960b3';
substr(a,5,-5); | 234960b3

See Also

Functions:

• “SUBSTRN Function” on page 794

SUBSTRN Function

Returns a substring, allowing a result with a length of zero.

Category: Character
Returned data type: VARCHAR, NVARCHAR
Tip: KSUBSTR has the same functionality.

Syntax

\[
\text{SUBSTRN}(\text{expression}, \text{position-expression}[\text{, length-expression}])
\]

Arguments

expression

specifies any valid expression that evaluates to a character string or numeric value.

If expression is numeric, then it is converted to a character value that uses the BEST32. format. Leading and trailing blanks are removed, and no message is sent to the SAS log.

Data type: CHAR, NCHAR, DOUBLE

position-expression

specifies any valid expression that evaluates to an integer and that specifies the position of the first character in the substring.

Data type: INTEGER

See

Chapter 13, “DS2 Expressions,” on page 93
length-expression

specifies any valid expression that evaluates to an integer and that specifies the length of the substring. If you do not specify length-expression, the SUBSTRN function returns the substring that extends from the position that you specify to the end of the string.

Data type

INTEGER

Details

The Basics

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns a string with a length of zero if either position-expression or length-expression has a missing or null value, or if position-expression is a non-positive value.
- If the value for length-expression is non-positive, the SUBSTRN function ignores length-expression.
- If the length that you specify extends beyond the end of the string, the result is truncated at the end, so that the last character of the result is the last character of the string.

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>the value of position-expression is nonpositive</td>
<td>SUBSTRN</td>
<td>returns a result beginning at the first character of the string.</td>
</tr>
<tr>
<td>the value of position-expression is nonpositive</td>
<td>SUBSTR</td>
<td>• writes a note to the log stating that the second argument is invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• sets <em>ERROR</em> = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• returns the substring that extends from the position that you specified to the end of the string.</td>
</tr>
<tr>
<td>the value of length-expression is nonpositive</td>
<td>SUBSTRN</td>
<td>returns a result with a length of zero.</td>
</tr>
<tr>
<td>Condition</td>
<td>Function</td>
<td>Result</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| the value of length-expression is nonpositive | SUBSTR | • writes a note to the log stating that the third argument is invalid.  
• sets _ERROR_ =1.  
• returns the substring that extends from the position that you specified to the end of the string. |
| the substring that you specify extends past the end of the string | SUBSTRN | truncates the result. |
| the substring that you specify extends past the end of the string | SUBSTR | • writes a note to the log stating that the third argument is invalid.  
• sets _ERROR_ =1.  
• returns the substring that extends from the position that you specified to the end of the string. |

**Examples**

**Example 1: Manipulating Strings with the SUBSTRN Function**

The following example shows how to manipulate strings with the SUBSTRN function.

```plaintext
proc ds2;
data test;
dcl char(6) string result;
dcl double position length;
retain string 'abcd';
drop string;
method run();
do position=-1 to 6;
do length=max(-1,-position) to 7-position;
   result=substrn(string, position, length);
   output;
end;
end;
enddata;
enddata;
run;
quit;

proc print data=test;
run;
```
Figure 23.5  Partial Output from the SUBSTRN Function

<table>
<thead>
<tr>
<th>Obs</th>
<th>result</th>
<th>position</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ab</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>abc</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>abcd</td>
<td>-1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>abcd</td>
<td>-1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>abcd</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>abcd</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>abcd</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>ab</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>abc</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>abcd</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>abcd</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>abcd</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>'a'</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Example 2: Comparison between the SUBSTR and SUBSTRN Functions
The following example compares the results of using the SUBSTR function and the SUBSTRN function when the first argument is numeric.

```plaintext
data _null_;  
dcl char substr_result substrn_result;  
method run();  
  substr_result='*' || substr('   1234.5678',2,6) || '*';  
  put substr_result=;  
  substrn_result='*' || substrn('1234.5678',2,6) || '*';  
  put substrn_result=;  
end;
```
SAS writes the following output to the log:

```
substr_result=*  1234*
substrn_result=*234.56*
```

### See Also

**Functions:**
- “SUBSTR Function” on page 792

---

**SUM Function**

Returns the sum of the non-null or nonmissing arguments.

**Category:** Descriptive Statistics

**Returned data type:** BIGINT, DECIMAL, DOUBLE, NUMERIC

---

**Syntax**

```
SUM(expression-1, expression-2 [, … expression-n])
```

**Arguments**

`expression`

specifies any valid expression that evaluates to a numeric value.

**Requirement**

At least two arguments are required.

**Data type**

BIGINT, DECIMAL, DOUBLE, NUMERIC

**See**

Chapter 13, “DS2 Expressions,” on page 93

---

**Details**

Null and missing values are ignored and not included in the computation. If all of the arguments have missing values, the result is a missing value. If all the arguments have a null value, the result is a null value.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

---

**Example**

The following statements illustrate the SUM function.
## SUMABS Function

Returns the sum of the absolute values of the nonmissing arguments.

**Category:** Descriptive Statistics  
**Returned data type:** DOUBLE

### Syntax

\[
\text{SUMABS(value-1, ..., value-n)}
\]

### Arguments

- **value**  
  specifies any valid expression that evaluates to a numeric value.  
  - Data type: DOUBLE

### Details

If all arguments have null or missing values, then the result is a null or missing value. Otherwise, the result is the sum of the absolute values of the nonmissing values.

### Examples

#### Example 1: Calculating the Sum of Absolute Values

The following example returns the sum of the absolute values of the nonmissing arguments.

```sas
data _null_;  
method run();  
x=sumabs(1,.,-2,0,3,.q,-4);  
put x=;  
end;  
enddata;  
run;  
```

SAS writes the following results to the log:

\[
x=10
\]
Example 2: Calculating the Sum of Absolute Values When You Use a Variable List

The following example uses a variable list and returns the sum of the absolute value of the nonmissing arguments.

```
data _null_;  
  method run();  
    x1 = 1;  
    x2 = 3;  
    x3 = 4;  
    x4 = 3;  
    x5 = 1;  
    x = sumabs(x1, x2, x3, x4, x5);  
    put x=;  
  end;  
enddata;  
run;  
```

SAS writes the following results to the log:

```
x=12
```

---

**TAN Function**

Returns the tangent.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Trigonometric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

```
TAN(expression)
```

**Arguments**

`expression`

specifies any valid expression that evaluates to a numeric value in radians.

<table>
<thead>
<tr>
<th>Restriction</th>
<th><code>expression</code> cannot be an odd multiple of π/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**See**

Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the TAN function:
<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=tan(0.5);</td>
<td>0.54630248984379</td>
</tr>
<tr>
<td>a=tan(0);</td>
<td>0</td>
</tr>
<tr>
<td>a=tan(3.14159/3);</td>
<td>1.73204726945457</td>
</tr>
</tbody>
</table>

### See Also

#### Functions:
- “TANH Function” on page 801

---

**TANH Function**

Returns the hyperbolic tangent.

**Category:** Trigonometric

**Returned data type:** DOUBLE

**Syntax**

TANH(expression)

**Arguments**

expression

specifies any valid expression that evaluates to a numeric value.

**Restriction**

expression cannot be an odd multiple of π/2

**Data type**

DOUBLE

### Details

The TANH function returns the hyperbolic tangent of the argument, which is given by the following equation.

\[
\frac{e^{\text{argument}} - e^{-\text{argument}}}{e^{\text{argument}} + e^{-\text{argument}}}
\]

**Example**

The following statements illustrate the TANH function.
<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=tanh(0);</td>
<td>0</td>
</tr>
<tr>
<td>a=tanh(0.5);</td>
<td>0.46211715726</td>
</tr>
<tr>
<td>a=tanh(-0.5);</td>
<td>-0.46211715726</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “TAN Function” on page 800

TIME Function

Returns the current time of day as a numeric SAS time value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

TIME()

Details

The TIME function does not take any arguments. It produces the current time in the form of a SAS time value.

Example

The following statements illustrate the TIME function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=time();</td>
<td>56526.0399990081</td>
</tr>
<tr>
<td>a=put(time(),time.);</td>
<td>15:42:06</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:
- “DATE Function” on page 466
- “DATETIME Function” on page 468
**TIMEPART Function**

Extracts a time value from a SAS datetime value.

- **Category:** Date and Time
- **Returned data type:** DOUBLE

**Syntax**

\[ \text{TIMEPART} \left( \text{datetime} \right) \]

**Arguments**

- **datetime**
  - specifies any valid expression that represents a SAS datetime value.
  - **Data type:** DOUBLE

**Example**

The following statements illustrate the TIMEPART function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dttm=datetime();</code></td>
<td>1450175146.88599</td>
</tr>
<tr>
<td><code>tm=put(timepart(dttm),time.);</code></td>
<td>10:25:47</td>
</tr>
</tbody>
</table>

**See Also**

- Chapter 14, “Dates and Times in DS2,” on page 111

**Functions:**

- “DATEPART Function” on page 467

---

**TIMEVALUE Function**

Returns the equivalent of a reference amount at a base date by using variable interest rates.

- **Category:** Financial
- **Returned data type:** DOUBLE
Syntax

\[
\text{TIMEVALUE}(\text{base-date}, \text{reference-date}, \text{reference-amount}, \text{compounding-interval}, \text{date-1}, \text{rate-1}, \ldots, \text{date-n}, \text{rate-n})
\]

Arguments

**base-date**

specifies the time value of the reference-amount at the base-date.

Requirement  
_Base-date_ is a SAS date.

Data type  
DOUBLE

**reference-date**

specifies the date of reference-amount.

Requirement  
_Reference-date_ is a SAS date.

Data type  
DOUBLE

**reference-amount**

specifies the amount at the reference-date.

Data type  
DOUBLE

**compounding-interval**

specifies the compounding interval.

Requirement  
_Compounding-interval_ is a SAS interval.

Data type  
CHAR

**date**

specifies the time at which rate takes effect. Each date is paired with a rate.

Requirement  
_Date_ is a SAS date.

Data type  
DOUBLE

**rate**

specifies the interest rate as numeric percentage that starts on _date_. Each rate is paired with a date.

Data type  
DOUBLE

Details

The following details apply to the TIMEVALUE function:

- The values for rates must be between –99 and 120.
- The list of date-rate pairs does not need to be sorted by date.
- When multiple rate changes occur on a single date, the TIMEVALUE function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
• There must be a valid date-rate pair whose date is at or prior to both the reference-date and the base-date.

Example

• You can express the accumulated value of an investment of $1,000 at a nominal interest rate of 10% compounded monthly for one year as the following:

```plaintext
data _null_;  
method run();  
    bd= to_double(date'2001-01-01');  
    rd= to_double(date'2000-01-01');  
    d= to_double(date'2000-01-01');  
    amount_base1 = timevalue(bd, rd, 1000, 'month', d, 10);  
    put amount_base1;  
end;  
enddata;  
run;
```

• If the interest rate jumps to 20% halfway through the year, the resulting calculation would be as follows:

```plaintext
data _null_;  
method run();  
    bd= to_double(date'2001-01-01');  
    rd= to_double(date'2000-01-01');  
    d1= to_double(date'2000-01-01');  
    d2= to_double(date'2000-07-01');  
    amount_base2 = timevalue(bd, rd, 1000, 'month', d1, 10, d2, 20);  
    put amount_base2;  
end;  
enddata;  
run;
```

• The date-rate pairs do not need to be sorted by date. This flexibility allows amount_base2 and amount_base3 to assume the same value:

```plaintext
data _null_;  
method run();  
    bd= to_double(date'2001-01-01');  
    rd= to_double(date'2000-01-01');  
    d1= to_double(date'2000-01-01');  
    d2= to_double(date'2000-07-01');  
    amount_base3 = timevalue(bd, rd, 1000, 'month', d1, 20, d2, 10);  
    put amount_base3;  
end;  
enddata;  
run;
```

---

**TINV Function**

Returns a quantile from the *t* distribution.

**Category:** Quantile

**Returned data type:** DOUBLE
Syntax
TINV(\(p, df[, nc]\))

Arguments

\(p\)
specifies any valid expression that evaluates to a numeric probability.

Range \(0 < p < 1\)

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

\(df\)
specifies any valid expression that evaluates to a numeric degrees of freedom parameter.

Range \(df > 0\)

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

\(nc\)
specifies any valid expression that evaluates to a numeric noncentrality parameter.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The TINV function returns the \(p^{th}\) quantile from the Student's \(t\) distribution with degrees of freedom \(df\) and a noncentrality parameter \(nc\). The probability that an observation from a \(t\) distribution is less than or equal to the returned quantile is \(p\).

TINV accepts a noninteger degree of freedom parameter \(df\). If the optional parameter \(nc\) is not specified or is 0, the quantile from the central \(t\) distribution is returned.

CAUTION:

For large values of \(nc\), the algorithm can fail. In that case, a missing value is returned.

Comparisons

TINV is the inverse of the PROBT function.

Example
The following statements illustrate the TINV function:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=tinv(.95, 2);</td>
<td>2.91998558035372</td>
</tr>
<tr>
<td>x=tinv(.95, 2.5, 3);</td>
<td>11.0338336251942</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “PROBT Function” on page 711

TO_DATE Function

Returns a DATE value from a DOUBLE value that specifies a SAS date value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DATE</td>
</tr>
</tbody>
</table>

Syntax

TO_DATE(date)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type: DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

A DOUBLE value that specifies a SAS date value represents the number of days between January 1, 1960, and a specified date. SAS can perform calculations on dates ranging from A.D. 1582 to A.D. 19,900. Dates before January 1, 1960, are negative numbers; dates after January 1, 1960, are positive numbers.

- SAS date values account for all leap year days, including the leap year day in the year 2000.
- SAS date values can reliably tell you what day of the week a particular day fell on as far back as September 1752, when the calendar was adjusted by dropping several days. SAS day-of-the-week and length-of-time calculations are accurate in the future to A.D. 19,900.
Example
The following example converts a DOUBLE value that specifies a SAS date value, to a
DATE value.

```sas
/* SAS date to date */
data _null_;  
dcl date da;  
dcl double d;  
method init();  
  d = 19358;  
  da = to_date(d);  
  put d=YYMMDD10. da=;  
end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
d=2012-12-31 da=2012-12-31
```

See Also

Functions:

- “TO_DOUBLE Function” on page 808
- “TO_TIME Function” on page 811
- “TO_TIMESTAMP Function” on page 812

TO_DOUBLE Function

Returns a DOUBLE value that specifies a SAS date, time, or datetime value, from a DATE, TIME, or
TIMESTAMP value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

`TO_DOUBLE (date | time | timestamp)`

Arguments

- **date**
  is a date constant, variable, or expression.

- **time**
  is a time constant, variable, or expression.

- **timestamp**
  is a timestamp constant, variable, or expression.
Details

The following list describes the values that can be returned by the TO_DOUBLE function:

- A SAS date value represents the number of days between January 1, 1960, and a specified date. SAS can perform calculations on dates ranging from A.D. 1582 to A.D. 19,900. Dates before January 1, 1960, are negative numbers; dates after January 1, 1960, are positive numbers.
- SAS date values account for all leap year days, including the leap year day in the year 2000.
- SAS date values can reliably tell you what day of the week a particular day fell on as far back as September 1752, when the calendar was adjusted by dropping several days. SAS day-of-the-week and length-of-time calculations are accurate in the future to A.D. 19,900.
- A SAS time value represents the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.
- A timestamp is a record of the date and time at which a certain event occurred.
- A SAS datetime value represents the number of seconds between January 1, 1960, and an hour/minute/second within a specified date.

Examples

Example 1: Converting a TIMESTAMP Value to a DOUBLE Value

The following example converts a TIMESTAMP value to a DOUBLE value that specifies a SAS datetime value.

```sas
/* Timestamp to SAS datetime */
data _null_;    
dcl timestamp ts;    
dcl double d;    
method init();    
    ts = timestamp '2012-10-19 16:51:36.0625';    
    d = to_double(ts);    
    put d=DATETIME28.9 ts=;    
end;    
enddata;    
run;
```

SAS writes the following output to the log:

```
```

Example 2: Converting a Date Value to a SAS Date Value

The following example converts a DATE value to a DOUBLE value that specifies a SAS date value:

```sas
/* Date to SAS date */
data _null_;    
dcl date da;    
dcl double d;    
method init();    
    da = date '2012-10-19';
```

Example 3: Converting a Time Value to a SAS Time Value

The following example converts a TIME value to a SAS time value:

```sas
/* Time to SAS time */
data _null_
   dcl time t;
   dcl double d;
   method init();
   t = time '16:51:36.0625';
   d = to_double(t);
   put d=TIME18.9 t=;
end;
enddata;
runclass;
```

SAS writes the following output to the log:

```
d=16:51:36.062500000 t=16:51:36.062500000
```

Example 4: Converting a NULL Timestamp to a SAS Datetime Value

The following example converts a NULL TIMESTAMP to a SAS datetime value:

```sas
/* NULL timestamp to SAS datetime */
data _null_
   dcl timestamp ts;
   dcl double d;
   method init();
   ts = null;
   d = to_double(ts);
   put d=DATETIME28.9 ts=;
end;
enddata;
runclass;
```

SAS writes the following output to the log:

```
d=                           . ts=
```

See Also

Functions:
- “TO_DATE Function” on page 807
- “TO_TIME Function” on page 811
- “TO_TIMESTAMP Function” on page 812
**TO_TIME Function**

Returns a TIME value from a DOUBLE value that specifies a SAS time value.

<table>
<thead>
<tr>
<th>Category</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>TIME</td>
</tr>
</tbody>
</table>

**Syntax**

\[
\text{TO_TIME}(date)\]

**Arguments**

*date*

specifies any valid expression that represents a SAS time value.

Data type **DOUBLE**

**See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

A SAS time value represents the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

**Example**

The following program converts a SAS time value to a formatted time value.

```sas
/* SAS time to time */
data _null_
  dcl time t;
  dcl double d;
  method init();
  d = 45911.68;
  t = to_time(d);
  put d=TIME18.9 t=;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=12:45:11.680000000 t=12:45:11.680000000
```

**See Also**

**Functions:**

- “TO_DATE Function” on page 807
- “TO_DOUBLE Function” on page 808
TO_TIMESTAMP Function

Returns a TIMESTAMP value from a DOUBLE value that specifies a SAS time value.

Category: Date and Time

Returned data type: TIMESTAMP

Syntax

TO_TIMESTAMP(date)

Arguments

date

specifies any valid expression that represents a SAS datetime value.

Data type: DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

A SAS datetime value is a DOUBLE value that represents the number of seconds between January 1, 1960, and an hour/minute/second within a specified date.

Examples

Example 1: Converting a SAS Datetime Value to a Timestamp Value

The following example converts a SAS datetime value to a timestamp value.

```sas
/* SAS datetime to timestamp */
data _null_;
dcl timestamp ts;
dcl double d;
method init();
d = 1614773470.3;
ts = to_timestamp(d);
put d=DATETIME28.9 ts=;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
```

Example 2: Converting a SAS Datetime Value That Is Missing

The following example shows how SAS handles the conversion of a missing SAS datetime value.
/* Missing SAS datetime to timestamp */
data _null_
  dcl timestamp ts;
  dcl double d;
  method init();
    d = .;
    ts = to_timestamp(d);
    put d=DATETIME28.9 ts=;
  end;
enddata;
run;

SAS writes the following output to the log:

d=                           . ts=

See Also

Functions:
  • “TO_DATE Function” on page 807
  • “TO_DOUBLE Function” on page 808
  • “TO_TIME Function” on page 811

TODAY Function

Returns the current date as a numeric SAS date value.

Syntax

TODAY()

Details

The TODAY function does not take any arguments. It produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

For more information about how DS2 handles dates, see Chapter 14, “Dates and Times in DS2,” on page 111.

Example

The following statement illustrates the TODAY function:
TRANSLATE Function

Replaces specific characters in a character expression.

**Category:** Character
**Returned data type:** NCHAR

**Syntax**

\[ \text{TRANSLATE}(\text{expression}, \text{to-characters}, \text{from-characters}) \]

**Arguments**

- **expression**
  specifies any valid expression that evaluates to a character string. \textit{expression} contains the original character value.
  
  Data type: CHAR

  See Chapter 13, “DS2 Expressions,” on page 93

- **to-characters**
  specifies the characters that you want \text{TRANSLATE} to use as substitutes.

  Data type: NCHAR

- **from-characters**
  specifies the characters that you want \text{TRANSLATE} to replace.

  Data type: NCHAR

**Details**

Values of \textit{to-characters} and \textit{from-characters} correspond on a character-by-character basis; \text{TRANSLATE} changes the first character in \textit{from-characters} to the first character in \textit{to-characters}, and so on. If \textit{to-characters} has fewer characters than \textit{from-characters}, \text{TRANSLATE} changes the extra \textit{from-characters} characters to blanks. If \textit{to-characters} has more characters than \textit{from-characters}, \text{TRANSLATE} ignores the extra \textit{to-characters}.

**Comparisons**

The \text{TRANWRD} function differs from the \text{TRANSTRN} function because \text{TRANSTRN} allows the replacement string to have a length of zero. \text{TRANWRD} uses a single blank instead when the replacement string has a length of zero.
The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Example

The following statement illustrates the TRANSLATE function.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=translate('XYZW','AB','VW');</td>
<td>XYZB</td>
</tr>
<tr>
<td>string1='AABBAABABB';</td>
<td></td>
</tr>
<tr>
<td>a=translate(string1,'12','AB');</td>
<td>1122112122</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “TRANSTRN Function” on page 815
- “TRANWRD Function” on page 818

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

**Category:** Character  
**Returned data type:** VARCHAR, NVARCHAR

**Syntax**

TRANSTRN(source-expression, target-expression, replacement-expression)

**Arguments**

**source-expression**  
 specifies any valid expression that evaluates to a character string, whose characters you want to translate.

**Data type**  
 NCHAR

**See**  
 Chapter 13, “DS2 Expressions,” on page 93
**target-expression**
specifies any valid expression that evaluates to a character string, whose characters are searched for in **source-expression**.

**Requirement**
The length for **target-expression** must be greater than zero.

**Data type**
NCHAR

**See**
Chapter 13, “DS2 Expressions,” on page 93

**replacement-expression**
specifies any valid expression that evaluates to a character string and that replaces **target-expression**.

**Data type**
NCHAR

**See**
Chapter 13, “DS2 Expressions,” on page 93

**Details**
The TRANSTRN function replaces or removes all occurrences of a given substring within a character string. The TRANSTRN function does not remove trailing blanks in the **target-expression** string and the **replacement-expression** string. To remove all occurrences of **target**, specify **replacement-expression** as TRIMN(“ ”).

**Comparisons**
The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANSTRN function differs from TRANSLATE in that TRANSTRN scans for substrings and replaces those substrings with a second substring.

**Examples**

**Example 1: Replacing All Occurrences of a Word**
In this example, the TRANSTRN function is used to replace Mrs. and Miss with Ms.

```plaintext
data _null_;  
dcl char(20) name;  
method run();  
   name='Mrs. Joan Smith';  
   name=transtrn(name, 'Mrs.', 'Ms.');  
   put name;  
   name='Miss Alice Cooper';  
   name=transtrn(name, 'Miss', 'Ms.');  
   put name;  
   end;  
enddata;
```
The following lines are written to the SAS log:

Ms. Joan Smith
Ms. Alice Cooper

**Example 2: Removing Blanks from the Search String**

In this example, the TRIM function is used with `target` to exclude blanks. If you did not include the TRIM function, the TRANSTRN function would not replace the source string because the target string contains blanks.

```sas
data test (overwrite=yes);
  dcl char(10) target;
  dcl char(3) replacement;
  dcl char(8) salelist salelist2;
  method run();
    salelist='CATFISH';
    target='FISH';
    replacement='NIP';
    salelist2=transtrn(salelist, trim(target), replacement);
    put salelist2;
  end;
enddata;
run;
```

The following is written to the SAS log:

CATNIP

**Example 3: Zero Length in the Third Argument of the TRANSTRN Function**

The following example shows the results of the TRANSTRN function when the third argument, `replacement`, has a length of zero. In DS2, a character constant that consists of two quotation marks with a blank in between them represents a single blank, and not a zero-length string. In the following example, the results for `string1` are different from the results for `string2`.

```sas
data _null_
  dcl char string1 string2;
  method run();
    string1='*' || transtrn('abcxabc', 'abc', trimn(' ')) || '*';
    put string1=;
    string2='*' || transtrn('abcxabc', 'abc', ' ') || '*';
    put string2=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
string1=*x*
string2=* x *
```

**See Also**

**Functions:**
TRANWRD Function

Replaces or removes all occurrences of a word in a character string.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

**Syntax**

TRANWRD(source-expression, target-expression, replacement-expression)

**Arguments**

*source-expression*

specifies any valid expression that evaluates to a character string, whose characters you want to replace.

- **Data type**: NCHAR
- **See**: Chapter 13, “DS2 Expressions,” on page 93

*target-expression*

specifies any valid expression that evaluates to a character string and that is searched for in source-expression.

- **Requirement**: The length of the target-expression must be greater than zero.
- **Data type**: NCHAR
- **See**: Chapter 13, “DS2 Expressions,” on page 93

*replacement-expression*

specifies any valid expression that evaluates to a character string and that replaces target-expression.

- **Data type**: NCHAR
- **See**: Chapter 13, “DS2 Expressions,” on page 93

**Details**

The TRANWRD function replaces or removes all occurrences of a given word (or a pattern of characters) within a character string. The TRANWRD function does not remove trailing blanks in the target-expression string and the replacement-expression string.
Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Examples

Example 1: Replacing All Occurrences of a Word
The following statements illustrate replacing all occurrences of a word using the TRANWRD function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>text='Whatever you so, so it with all your might. -P.T. Barnum';</td>
<td>Whatever you do, do it with all your might. -P.T. Barnum</td>
</tr>
<tr>
<td>fix=tranwrd(text,'so','do');</td>
<td></td>
</tr>
</tbody>
</table>

Example 2: Removing Blanks from the Search String
This example illustrates incorrect data type declarations. The TRANWRD function does not replace the source string because TARGET is declared as char(10) and the TRANWRD function searches for the character string 'pail ' and not 'pail'.

```
data _null_;
dcl char(100) text finaltext;
dcl char(10) target;
dcl varchar(10) rplc;
method run();
text='Believe and act as if it were impossible to pail. -Charles F. Kettering';
target='pail';
rplc='fail';
finaltext=tranwrd(text, target, rplc);
put finaltext=;                          
end;
enddata;
run;
```

This line is written to the SAS log.

finaltext=Believe and act as if it were impossible to pail. -Charles F. Kettering
By changing the data type declaration to `VARCHAR(10)`, trailing blanks are excluded from the search:

```sas
data _null_;  
dcl char(100) text finaltext;  
dcl varchar(10) target;  
dcl varchar(10) rplc;  
method run();  
   text='Believe and act as if it were impossible to pail. -Charles F. Kettering';  
   target='pail';  
   rplc='fail';  
   finaltext=tranwrd(text, target, rplc);  
   put finaltext=;  
end;  
enddata;  
run ;
```

This line is written to the SAS log.

```
finaltext=Believe and act as if it were impossible to fail. -Charles F. Kettering
```

### See Also

**Functions:**

- “TRANSLATE Function” on page 814
- “TRANSTRN Function” on page 815

---

### TRIGAMMA Function

Returns the value of the trigamma function.

- **Category:** Mathematical
- **Returned data type:** DOUBLE

#### Syntax

`TRIGAMMA(expression)`

#### Arguments

**expression**

specifies any valid expression that evaluates to a numeric value.

- **Restriction:** Nonpositive integers are invalid.
- **Data type:** DOUBLE

**See**

Chapter 13, “DS2 Expressions,” on page 93
Details

The TRIGAMMA function returns the derivative of the DIGAMMA function. For $expression > 0$, the TRIGAMMA function is the second derivative of the LGAMMA function.

Example

The following statement illustrates the TRIGAMMA function:

```
Statements          Results
a=trigamma(3);      0.39493406684822
```

See Also

Functions:
- “DIGAMMA Function” on page 477
- “LGAMMA Function” on page 608

TRIM Function

Removes trailing blanks from a character expression, and returns one blank if the string is missing.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias:</td>
<td>TRIMN</td>
</tr>
<tr>
<td>Returned data type:</td>
<td>VARCHAR, NVARCHAR</td>
</tr>
</tbody>
</table>

Syntax

```
TRIM('expression')
```

Arguments

`expression`

specifies any valid expression that evaluates to a character string.

Data type: CHAR, NCHAR

See

Chapter 13, “DS2 Expressions,” on page 93

Details

The TRIM function copies a character argument, removes trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns one blank. TRIM is useful for concatenating because concatenation does not remove trailing blanks.
Comparisons

The TRIMN and TRIM functions are similar. TRIMN returns a string with a length of zero for a blank string. TRIM returns one blank for a blank string.

Example

The following statements illustrate the TRIM function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>string='Testscore   '; results=trim(string)</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:
- “COMPRESS Function” on page 437
- “LEFT Function” on page 602
- “STRIP Function” on page 790
- “TRIMN Function” on page 822

TRIMN Function

Removes trailing blanks from character expressions, and returns a string with a length of zero if the expression is missing.

- **Category:** Character
- **Returned data type:** VARCHAR, NVARCHAR

Syntax

TRIMN(expression)

Arguments

expression

specifies any valid expression that evaluates to a character string.

- **Data type:** CHAR, NCHAR

See

Chapter 13, “DS2 Expressions,” on page 93
Details

Length of Returned Variable
Assigning the results of TRIMN to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns it to the variable.

The Basics
TRIMN copies a character argument, removes all trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIMN returns a string with a length of zero. TRIMN is useful for concatenating because concatenation does not remove trailing blanks.

Comparisons
The TRIMN and TRIM functions are similar. TRIMN returns a string with a length of zero for a blank string. TRIM returns one blank for a blank string.

Example
The following statements illustrate the TRIMN function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=&quot;A&quot;</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td></td>
</tr>
<tr>
<td>x=&quot;   &quot;;</td>
<td></td>
</tr>
<tr>
<td>z=&quot;&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td>put z;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:
- “COMPRESS Function” on page 437
- “LEFT Function” on page 602
- “RIGHT Function” on page 754
- “TRIM Function” on page 821

TRUNC Function
Truncates a numeric value to a specified length.

Category: Truncation
Returned data type: DOUBLE
Syntax

TRUNC(expression, length-expression)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

length-expression

specifies any valid expression that evaluates to a numeric value.

Range 3–8

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The TRUNC function truncates a full-length numeric expression (stored as a DOUBLE) to a smaller number of bytes, as specified in length-expression and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

Example

The following statements illustrate the TRUNC function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=trunc(3.1,3);</td>
<td>3.099609375</td>
</tr>
<tr>
<td>x=trunc(3.1,4);</td>
<td>3.09999847412109</td>
</tr>
<tr>
<td>x=trunc(3.1,5);</td>
<td>3.0999999403953</td>
</tr>
<tr>
<td>x=trunc(3.1,6);</td>
<td>3.0999999997671</td>
</tr>
<tr>
<td>x=trunc(3.1,7);</td>
<td>3.0999999999999</td>
</tr>
<tr>
<td>x=trunc(3.1,8);</td>
<td>3.1</td>
</tr>
</tbody>
</table>

UNIFORM Function

Returns a random variate from a uniform distribution.

Category: Random Number

Alias: RANUNI

Returned data type: DOUBLE

See: “Uniform Distribution” on page 743
UPCASE Function

Converts all letters in an argument to uppercase.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias:</td>
<td>UPPER</td>
</tr>
<tr>
<td>Returned data type:</td>
<td>VARCHAR, NVARCHAR</td>
</tr>
</tbody>
</table>

Syntax

UPCASE(expression)

Arguments

eexpression

specifies any valid expression that evaluates to a character string.

Data type

CHAR, NCHAR

See

Chapter 13, “DS2 Expressions,” on page 93

Details

The UPCASE function copies a character expression, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Comparisons

The LOWCASE function converts all letters in an argument to lowercase letters. The UPCASE function converts all letters in an argument to uppercase letters.

Example

The following statement illustrates the UPCASE function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>name=upcase('John B. Smith');</td>
<td>JOHN B. SMITH</td>
</tr>
</tbody>
</table>

See Also

Functions:

•  “LOWCASE Function” on page 614
**USS Function**

Returns the uncorrected sum of squares.

**Category:** Descriptive Statistics

**Returned data type:** DOUBLE

**Syntax**

USS(expression [, ...expression-n])

**Arguments**

expression

specifies any valid expression that evaluates to a numeric value.

**Requirement**

At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

**Data type**

DOUBLE

See

Chapter 13, “DS2 Expressions,” on page 93

**Example**

The following statements illustrate the USS function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=uss(4,2,3.5,6);</td>
<td>68.25</td>
</tr>
<tr>
<td>a=uss(4,2,3.5,6,);</td>
<td>68.25</td>
</tr>
</tbody>
</table>

**See Also**

**Functions:**

- “CSS Function” on page 458

---

**UUIDGEN Function**

Returns the short form of a Universally unique identifier (UUID).

**Category:** Special

**Returned data type:** CHAR
Syntax

UUIDGEN()

Without Arguments
The UUIDGEN function has no arguments.

Details
The UUIDGEN function returns a UUID (a unique value) for each call. The default result is 36 characters long and it looks like this:
Sab6fa40-426b-4375–bb22–2d0291f43319

Example
The following example returns a UUID. Note that a variable declaration of 36 characters is required.

data _null_;  
dcl char(36) x;  
method run();  
x=uuidgen();  
put x;  
end;  
enddata;  
run;

The following value is written to the SAS log. Each UUID is unique.

25C752D5-AFA1-4932-BEE6-39E4006C2AAB

See Also

Other References:
• “Universal Unique Identifiers” in SAS Language Reference: Concepts

VAR Function
Returns the variance.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Descriptive Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax
VAR(expression-1, expression-2 [,...expression-n])
**Arguments**

**expression**

specifies any valid expression that evaluates to a numeric value. The argument list can consist of a variable list.

**Requirement**
At least two non-null or nonmissing arguments are required.
Otherwise, the function returns a null or missing value.

**Data type**
DOUBLE

**See**
Chapter 13, “DS2 Expressions,” on page 93

---

**Example**

The following statements illustrate the VAR function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1=\text{var}(4, 2, 3.5, 6);$</td>
<td>2.7291666667</td>
</tr>
<tr>
<td>$x_2=\text{var}(4, 6, .);$</td>
<td>2</td>
</tr>
</tbody>
</table>

---

**VERIFY Function**

Returns the position of the first character that is unique to an expression.

**Category:** Character

**Returned data type:** DOUBLE

**Syntax**

VERIFY(target-expression, search-expression)

**Arguments**

**target-expression**

specifies any valid expression that evaluates to a character string that is to be searched.

**Requirement**
Literal character strings must be enclosed in single quotation marks.

**Data type**
NCHAR

**See**
Chapter 13, “DS2 Expressions,” on page 93

**search-expression**

specifies any valid expression that evaluates to a character string.

**Requirement**
Literal character strings must be enclosed in single quotation marks.
Data type: NCHAR
See: Chapter 13, “DS2 Expressions,” on page 93

Details
The VERIFY function returns the position of the first character in target-expression that is not present in search-expression. If there are no characters in target-expression that are unique from those in search-expression, VERIFY returns a 0.

Comparisons
The INDEX function returns the position of the first occurrence of search-expression that is present in target-expression where the VERIFY function returns the position of the first character in target-expression that does not contain search-expression.

Example
The following statements illustrate the VERIFY function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='abc'; y='abcdef'; z=verify(y,x);</td>
<td>4</td>
</tr>
<tr>
<td>x='abcdef'; y='abcdef'; z=verify(y,x);</td>
<td>0</td>
</tr>
</tbody>
</table>

See Also
Functions:
- “INDEX Function” on page 533

VFORMAT Function
Returns the format that is associated with the specified variable.

Category: Variable Information
Returned data type: NCHAR

Syntax
VFORMAT(variable | variable-list[i] | variable-array[i])
Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction  You cannot use an expression as an argument.

Data type  All data types

variable-list
specifies a collection of DS2 variables.

See  “Variable Lists” on page 59

variable-array
specifies a collection of homogenous DS2 variables.

See  “Variable Arrays” on page 123

i
specifies the element number of the named variable list or variable array.

Details

VFORMAT returns the complete format name, which includes the width and the period (for example, $CHAR20.).

Example

The following example returns the format of a character variable.

```
proc ds2;
data _null_;    
    declare varchar(7) str having format $char7.;
    method run();
    a = vformat(str);
    put a=;  
end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
a=$char7.
```

See Also

Functions:

- “VINARRAY Function” on page 831
- “VINFORMAT Function” on page 832
- “VLABEL Function” on page 833
- “VLENGTH Function” on page 835
VINARRAY Function

Returns a value that indicates whether the specified variable is a member of an array.

**Category:** Variable Information  
**Returned data type:** INTEGER

**Syntax**

VINARRAY(variable | variable-list[i] | variable-array[i])

**Arguments**

*variable*

specifies a variable that is expressed as a scalar or as an array reference.

*variable-list*

specifies a collection of DS2 variables.

*variable-array*

specifies a collection of homogenous DS2 variables.

*i*

specifies the element number of the named variable list or variable array.

**Details**

VINARRAY returns a value of 1 if the given variable is a member of a variable array. Otherwise, VINARRAY returns a value of 0.

**Example**

The following example returns a value (0 or 1) depending on whether the specified variable is a member of an array:

```plaintext
data _null_;  
declare int a b c d;  
vararray int x[3] a b c;  
method init();  
declare int c_in;  
declare int d_in;  
c_in=vinarray(c);  /* c is in array x */```
d_in=vinarray(d);  /* d is not in an array */
put c_in=;
put d_in=;
end;
enddata;
run;

SAS writes the following output to the log:

c_in=1
d_in=0

See Also

Functions:

• “VFORMAT Function” on page 829
• “VINFORMAT Function” on page 832
• “VLABEL Function” on page 833
• “VLENGTH Function” on page 835
• “VNAME Function” on page 836
• “VTYPE Function” on page 837

VINFORMAT Function

Returns the informat that is associated with the specified variable.

Category: Variable Information
Returned data type: NCHAR

Syntax

VINFORMAT(variable | variable-list[i] | variable-array[i])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction
You cannot use an expression as an argument.

Data type
All data types

variable-list
specifies a collection of DS2 variables.

See “Variable Lists” on page 59

variable-array
specifies a collection of homogenous DS2 variables.
See  “Variable Arrays” on page 123

\[ i \]

specifies the element number of the named variable list or variable array.

Details

VINFORMAT returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Example

The following example returns an informat that is associated with the specified variable.

```sas
data _null_;  
declare varchar(10) str having informat $char5.;  
method run();  
a=vinformat(str);  
   put a=;  
end;  
enddata;  
run;  
```

SAS writes the following output to the log:

\[ a=$char5. \]

See Also

Functions:

- “VINARRAY Function” on page 831
- “VFORMAT Function” on page 829
- “VLABEL Function” on page 833
- “VLENGTH Function” on page 835
- “VNAME Function” on page 836
- “VTYPE Function” on page 837


### VLABEL Function

Returns the label that is associated with the specified variable.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Variable Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

**Syntax**

\[
\text{VLABEL}(\text{variable} \mid \text{variable-list}[i] \mid \text{variable-array}[i])
\]
Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See “Variable Lists” on page 59

variable-array
specifies a collection of homogenous DS2 variables.

See “Variable Arrays” on page 123

i
specifies the element number of the named variable list or variable array.

Details

VLABEL returns the label that is associated with the specified variable. If there is no label, VLABEL returns the variable name.

Example

The following example returns a label for a specified variable.

```
data _null_;    
declare varchar(10) fname having label 'First Name';
method run();
a=vlabel(fname);
put a=;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=First Name
```

See Also

Functions:

- “VINARRAY Function” on page 831
- “VFORMAT Function” on page 829
- “VINFORMAT Function” on page 832
- “VLENGTH Function” on page 835
- “VNAME Function” on page 836
- “VTYPE Function” on page 837
VLENGTH Function

Returns the size of the specified variable.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Variable Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>

Syntax

VLENGTH(\(variable \mid variable-list[i] \mid variable-array[i]\))

Arguments

variable

specifies a value that is expressed as a scalar or as an array reference.

Restriction

You cannot use an expression as an argument.

Data type

All data types

variable-list

specifies a collection of DS2 variables.

See “Variable Lists” on page 59

variable-array

specifies a collection of homogenous DS2 variables.

See “Variable Arrays” on page 123

i

specifies the element number of the named variable list or variable array.

Details

The length of numeric data types is defined as the maximum number of digits used by the data type of the column, or the precision of the data. For character types, this is the length in characters of the data; for binary data types, this is the length in bytes of the data. For the TIME, TIMESTAMP, and all interval data types, this is the number of characters in the character representation of this data.

Example

The following example returns the length of the specified variable.

```
data _null_;  
declare char (7) str;  
method run();  
    str='World';  
    a=vlength {str};  
    put a=;  
end;  
run;```
SAS writes the following output to the log:

\[ a = 7 \]

See Also

Functions:
- “VINARRAY Function” on page 831
- “VFORMAT Function” on page 829
- “VINFORMAT Function” on page 832
- “VLABEL Function” on page 833
- “VNAME Function” on page 836
- “VTYPE Function” on page 837

VNAME Function

Returns the name of the specified variable.

<table>
<thead>
<tr>
<th>Category</th>
<th>Variable Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

Syntax

\[ \text{VNAME}(\text{variable} \mid \text{variable-list}[i] \mid \text{variable-array}[i]) \]

Arguments

\textit{variable}

specifies a variable that can be expressed as a scalar or as an array reference.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>You cannot use an expression as an argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>All data types</td>
</tr>
</tbody>
</table>

\textit{variable-list}

specifies a collection of DS2 variables.

See “Variable Lists” on page 59

\textit{variable-array}

specifies a collection of homogenous DS2 variables.

See “Variable Arrays” on page 123

\textit{i}

specifies the element number of the named variable list or variable array.
Example

The following example returns the name of the specified variable.

```sas
data _null_;  
  vararray int x[3] a b c;  

  method run();  
    y=vname(x[1]);  
    put y=;  
  end;  
enddata;  
run;
```

SAS writes the following output to the log:

```
y=a
```

See Also

Functions:
- “VINARRAY Function” on page 831
- “VFORMAT Function” on page 829
- “VINFORMAT Function” on page 832
- “VLABEL Function” on page 833
- “VLENGTH Function” on page 835
- “VTYPE Function” on page 837

---

**VTYPE Function**

Returns the full name of the data type that is associated with a variable.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Variable Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>NCHAR</td>
</tr>
</tbody>
</table>

**Syntax**

```
VTYPE( variable | variable-list[i] | variable-array[i] )
```

**Arguments**

`variable`

specifies a variable that is expressed as a scalar or as an array reference.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>You cannot use an expression as an argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>All data types</td>
</tr>
</tbody>
</table>

`variable-list`

specifies a collection of DS2 variables.
variable-array

specifies a collection of homogenous DS2 variables.

See “Variable Arrays” on page 123

i

specifies the element number of the named variable list or variable array.

Details

VTYPE returns the data type name for the data type of the variable.

Example

The following example returns the name of the data type that is associated with the specified variable.

```sas
data _null_; declare double d; declare timestamp t; declare nvarchar(10) n; method run(); declare char(20) a b c; a=vtype(d); put a=; b=vtype(t); put b=; c=vtype(n); put c=; end; enddata; run;
```

SAS writes the following output to the log:

```
a=double
b=timestamp
c=nvarchar
```

See Also

Functions:

- “VINARRAY Function” on page 831
- “VFORMAT Function” on page 829
- “VINFORMAT Function” on page 832
- “VLABEL Function” on page 833
- “VLENGTH Function” on page 835
- “VNAME Function” on page 836
**WEEK Function**

Returns the week-number value.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Date and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

**Syntax**

```
WEEK([sas-date], ['descriptor'])
```

**Arguments**

- **sas-date**
  specifies the SAS date value. If the `sas-date` argument is not specified, the WEEK function returns the week-number value of the current date.
  
  Data type: DOUBLE

- **descriptor**
  specifies the value of the descriptor. The following descriptors can be specified in uppercase or lowercase characters.
  
  **U**
  specifies the number-of-the-week within the year. Sunday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'u')` is 53. U is the default value.
  
  **Tip**
  The U and W descriptors are similar, except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.
  
  **See**
  “The U Descriptor” on page 840

- **V**
  specifies the number-of-the-week whose value is represented as a decimal number in the range 1–53. Monday is considered the first day of the week and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year.
  
  **See**
  “The V Descriptor” on page 840

- **W**
  specifies the number-of-the-week within the year. Monday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'w')` is 53.
  
  **Tip**
  The U and W descriptors are similar except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.
The WEEK function reads a SAS date value and returns the week number. The WEEK function is not dependent on locale, and uses only the Gregorian calendar in its computations.

The U Descriptor
The WEEK function with the U descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year is represented as 05.

Sunday is considered the first day of the week. For example, the value of week('01jan2013'd, 'u') is 0.

The V Descriptor
The WEEK function with the V descriptor reads a SAS date value and returns the week number. The number-of-the-week is represented as a decimal number in the range 01–53. The decimal number has a leading zero and a maximum value of 53. Weeks begin on a Monday, and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. In the following example, 01jan2014 and 31dec2013 occur in the same week. The first day (Monday) of that week is 30dec2013. Therefore, week('01jan2014'd, 'v') and week('30dec2013'd, 'v') both return a value of 53. This means that both dates occur in week 53 of the year 2013.

The W Descriptor
The WEEK function with the W descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year would be represented as 05.

Monday is considered the first day of the week. Therefore, the value of week('30dec2013'd, 'w') is 1.

Comparisons of Descriptors
U is the default descriptor. Its range is 0-53, and the first day of the week is Sunday. The V descriptor has a range of 1-53 and the first day of the week is Monday. The W descriptor has a range of 0-53 and the first day of the week is Monday.

The following list describes the descriptors and an associated week:

- Week 0:
**WEEK Function**

- **U** indicates the days in the current Gregorian year before week 1.
- **V** does not apply.
- **W** indicates the days in the current Gregorian year before week 1.

- **Week 1:**
  - **U** begins on the first Sunday in a Gregorian year.
  - **V** begins on the Monday between December 29 of the previous Gregorian year and January 4 of the current Gregorian year. The first ISO week can span the previous and current Gregorian years.
  - **W** begins on the first Monday in a Gregorian year.

- **End of Year Weeks:**
  - **U** specifies that the last week (52 or 53) in the year can contain less than 7 days. A Sunday to Saturday period that spans 2 consecutive Gregorian years is designated as 52 and 0 or 53 and 0.
  - **V** specifies that the last week (52 or 53) of the ISO year contains 7 days. However, the last week of the ISO year can span the current Gregorian and next Gregorian year.
  - **W** specifies that the last week (52 or 53) in the year can contain less than 7 days. A Monday to Sunday period that spans two consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

**Example**

The following example shows the values of the U, V, and W descriptors for the date August 16, 2013.

```sas
data _null_
  dcl double sasdate x y z;
  method run();
    sasdate=to_double(date'2013-08-16');
    x=week(sasdate, 'u');
    y=week(sasdate, 'v');
    z=week(sasdate, 'w');
    put x;
    put y;
    put z;
  end;
enddata;
run;
```

The following lines are written to the SAS log.

```
32
33
32
```

**See Also**

**Functions:**

- “INTNX Function” on page 564
WEEKDAY Function

From a SAS date value, returns an integer that corresponds to the day of the week.

**Category:** Date and Time  
**Returned data type:** DOUBLE

**Syntax**

WEEKDAY(expression)

**Arguments**

- **expression** specifies any valid expression that represents a SAS date value.
- **Data type** DOUBLE
- **See** Chapter 13, “DS2 Expressions,” on page 93

**Details**

The WEEKDAY function produces an integer that represents the day of the week, where 1 = Sunday, 2 = Monday, …, 7 = Saturday.

For information about how DS2 handles date and time values, see Chapter 14, “Dates and Times in DS2,” on page 111.

**Example**

The following statement illustrates the WEEKDAY function when the current day is Sunday:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=weekday(today());</td>
<td>1</td>
</tr>
</tbody>
</table>

WHICHC Function

Returns the first position of a character string from a list of character strings.

**Category:** Character
Returned data type: DOUBLE

Syntax

WHICHC(search-expression, expression-list-item-1, expression-list-item-2 [, …expression-list-item-n])

Arguments

search-expression
 specifies any valid expression that evaluates to a character string that is compared with a list of character string expressions.

Requirements
 Literal character strings must be enclosed in single quotation marks.

Data type: NCHAR

See
 Chapter 13, “DS2 Expressions,” on page 93

expression-list-item
 specifies any valid expression that evaluates to a character string and that is a member of a list of character string expressions.

Requirements
 Literal character strings must be enclosed in single quotation marks.

At least two expressions are required in the list.

Data type: NCHAR

See
 Chapter 13, “DS2 Expressions,” on page 93

Details

The WHICHC function searches the character expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHC returns its position in the expression list. If none of the expressions match the search expression, WHICHC returns a value of 0.

Example

In the following example, 'Spain' appears twice in the list. The WHICHC function return the first position of 'Spain' in the list:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=whichc('Spain', 'Denmark', 'Germany', 'Austria', 'Spain', 'China', 'Egypt', 'Spain', 'France');</td>
<td>4</td>
</tr>
</tbody>
</table>

See Also

Functions:

- “WHICHN Function” on page 844
WHICHN Function

Returns the first position of a number from a list of numbers.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Mathematical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned data type:</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Syntax

WHICHN(search-expression, expression-list-item-1, expression-list-item-2
       [, ...expression-list-item-n])

Arguments

search-expression

specifies any valid expression that evaluates to a number and that is compared with a list of numeric expressions.

<table>
<thead>
<tr>
<th>Data type</th>
<th>DOUBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

expression-list-item

specifies any valid expression that evaluates to a number and is part of a list.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>At least two expressions are required in the list.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>See</td>
<td>Chapter 13, “DS2 Expressions,” on page 93</td>
</tr>
</tbody>
</table>

Details

The WHICHN function searches the numeric expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHN returns its position in the expression list. If none of the expressions match the search expression, WHICHN returns a value of 0. Arguments for the WHICHN functions can be any numeric data type.

Example

In the following example, 4.5 appears two times in the list. The WHICHN function return the first position of 4.5 in the list:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=whchn(4.5,7.3, 8.6, 4.5, 4.5, 2.1, 6.4);</td>
<td>3</td>
</tr>
</tbody>
</table>
YEAR Function

Returns the year from a SAS date value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

YEAR(date)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type: DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93

Details

The YEAR function produces a four-digit numeric value that represents the year.

Example

The following statement illustrates the YEAR function when the year is 2007.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date=today();</td>
<td></td>
</tr>
<tr>
<td>y=year(date);</td>
<td>2007</td>
</tr>
</tbody>
</table>

See Also

- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:

- “DAY Function” on page 469
- “MONTH Function” on page 635
YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

**Category:** Financial

**Returned data type:** DOUBLE

---

**Syntax**

\[ \text{YIELDP}(A, c, n, K, k_0, p) \]

**Arguments**

\( A \)

specifies the face value.

Range \( A > 0 \)

Data type DOUBLE

\( c \)

specifies the nominal annual coupon rate, expressed as a fraction.

Range \( 0 \leq c < 1 \)

Data type DOUBLE

\( n \)

specifies the number of coupons per year.

Range \( n > 0 \)

Data type INTEGER

\( K \)

specifies the number of remaining coupons from settlement date to maturity.

Range \( K > 0 \)

Data type INTEGER

\( k_0 \)

specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Range \( 0 < k_0 \leq \frac{1}{n} \)

Data type DOUBLE

\( p \)

specifies the price with accrued interest.
The YIELDP function is based on the following relationship:

\[ P = \sum_{k=1}^{K} c(k) \left( \frac{1}{1 + \frac{y}{n}} \right)^{t_k} \]

The following relationships apply to the preceding equation:

- \( t_k = nk_0 + k - 1 \)
- \( c(k) = \frac{c}{n} A \) for \( k = 1, \ldots, K - 1 \)
- \( c(K) = \left( 1 + \frac{c}{n} \right) A \)

The YIELDP function solves for \( y \).

Example

In the following example, the YIELDP function returns the yield-to-maturity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the price with accrued interest is 800. The value returned is 0.0775031248.

```sas
data _null_; 
method run(); 
y=yieldp(1000,.01,4,14,.165,800); 
put y; 
end; 
enddata; 
run;
```

SAS writes the following output to the log:

0.0775031247735

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

**Category:** Date and Time  
**Returned data type:** INTEGER  

**Syntax**

\[ \text{YRDIF}(\text{start-date}, \text{end-date}[, \text{basis}]) \]
Arguments

start-date
specifies a SAS date value that identifies the starting date.

Data type  DOUBLE

end-date
specifies a SAS date value that identifies the ending date.

Data type  DOUBLE

basis
identifies a character constant or variable that describes how SAS calculates a date
difference or a person’s age. The following character strings are valid:

'30/360'
specifies a 30-day month and a 360-day year in calculating the number of years.
Each month is considered to have 30 days, and each year 360 days, regardless of
the actual number of days in each month or year.

Alias  '360'
Tip  If either date falls at the end of a month, it is treated as if it were the last
day of a 30-day month.

'ACT/ACT'
uses the actual number of days between dates in calculating the number of years.
SAS calculates this value as the number of days that fall in 365-day years divided
by 365 plus the number of days that fall in 366-day years divided by 366.

Alias  'Actual'

'ACT/360'
uses the actual number of days between dates in calculating the number of years.
SAS calculates this value as the number of days divided by 360, regardless of the
actual number of days in each year.

'ACT/365'
uses the actual number of days between dates in calculating the number of years.
SAS calculates this value as the number of days divided by 365, regardless of the
actual number of days in each year.

'AGE'
specifies that a person’s age will be computed.

If you do not specify a third argument, AGE becomes the default value for basis.

Data type  CHAR

Details

Using YRDIF in Financial Applications

The Basics
The YRDIF function can be used in calculating interest for fixed income securities when
the third argument, basis, is present. YRDIF returns the difference between two dates
according to specified day count conventions.
Calculations That Use ACT/ACT Basis
In YRDIF calculations that use the \textit{ACT/ACT} basis, both a 365-day year and 366-day year are taken into account. For example, if \(n_{365}\) equals the number of days between the start and end dates in a 365-day year, and \(n_{366}\) equals the number of days between the start and end dates in a 366-day year, the YRDIF calculation is computed as \[ \text{YRDIF} = \frac{n_{365}}{365.0} + \frac{n_{366}}{366.0}. \] This calculation corresponds to the commonly understood \textit{ACT/ACT} day count basis that is documented in the financial literature. The values for \textit{basis} also includes \textit{30/360}, \textit{ACT/360}, and \textit{ACT/365}. Each has well-defined meanings that must be conformed to in calculating interest payments for specific financial instruments.

Computing a Person’s Age
The YRDIF function can compute a person’s age. The first two arguments, \textit{start-date} and \textit{end-date}, are required. If the value of \textit{basis} is \textit{AGE}, then YRDIF computes the age. The age computation takes into account leap years. No other values for \textit{basis} are valid when computing a person’s age.

Examples

\textbf{Example 1: Calculating a Difference in Years Based on Basis}
In the following example, YRDIF returns the difference in years between two dates based on each of the options for \textit{basis}.

```sas
data _null_;    method run();
  sdate= to_double(date'1998-10-16');
  edate= to_double(date'2010-02-06');
  y30360=yrdif(sdate, edate, '30/360');
  yactact=yrdif(sdate, edate, 'ACT/ACT');
  yact360=yrdif(sdate, edate, 'ACT/360');
  yact365=yrdif(sdate, edate, 'ACT/365');
  put y30360=;
  put yactact=;
  put yact360=;
  put yact365=;
end;
enddata;
run;
```

SAS writes the following results to the log:

\begin{verbatim}
y30360=11.3055555555555
yactact=11.3095890410958
yact360=11.475
yact365=11.317808219178
\end{verbatim}

\textbf{Example 2: Calculating a Person’s Age}
You can calculate a person’s age by using three arguments in the YRDIF function. The third argument, \textit{basis}, must have a value of \textit{AGE}:

```sas
data _null_;    method run();
  sdate= to_double(date'1998-10-16');
  edate= to_double(date'2010-02-16');
  age=yrdif(sdate, edate, 'AGE');
```

put age= 'years';
end;
enddata;
run;

SAS writes the following results to the log:

age=11.3369863013698 years

See Also

Functions:

• “DATDIF Function” on page 463

References


YYQ Function

Returns a SAS date value from year and quarter year values.

Category: Date and Time

Returned data type: DOUBLE

Syntax

YYQ(year, quarter)

Arguments

year

specifies any valid expression that evaluates to a two-digit or four-digit integer that represents the year.

Interaction The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” on page 93
quarter
specifies the quarter of the year (1, 2, 3, or 4).

Data type
DOUBLE

See
Chapter 13, “DS2 Expressions,” on page 93

Details
The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either `year` or `quarter` is null or missing, or if the quarter value is not valid, the result is a null or missing value.

Example
The following statements illustrate the YYQ function:

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateValue = yyq(2006,3);</td>
<td>16983</td>
</tr>
<tr>
<td>Date7Value = put(DateValue, date7.);</td>
<td>01JUL06</td>
</tr>
<tr>
<td>Date9Value = put(DateValue, date9.);</td>
<td>01JUL2006</td>
</tr>
<tr>
<td>StartOfQuarter = yyq(2006,4);</td>
<td>17075</td>
</tr>
<tr>
<td>StartOfQuarter9 = put(StartOfQuarter, date9.);</td>
<td>01OCT2006</td>
</tr>
</tbody>
</table>

See Also
- Chapter 14, “Dates and Times in DS2,” on page 111

Functions:
- “QTR Function” on page 729
- “YEAR Function” on page 845
Chapter 24

DS2 Informats

Overview of Informats

An informat is an instruction that determines how values are read into a column. For example, the following value contains a dollar sign and commas:

$1,000,000

To remove the dollar sign ($) and commas (,) before storing the numeric value 1000000 in a column, read this value with the COMMA11. informat.

General Informat Syntax

DS2 informats have the following form:

[ $ ] informat [w] [d]

Here is an explanation of the syntax:

$ indicates a character informat; its absence indicates a numeric informat.

informat names the informat. The informat is a SAS informat or a user-defined informat that was previously defined with the INVALUE statement in PROC FORMAT. For more information about user-defined informats, see PROC FORMAT in the Base SAS Procedures Guide.
w
specifies the informat width, which for most informats is the number of columns in
the input data.

d
specifies an optional decimal scaling factor in the numeric informats. SAS divides
the input data by 10 to the power of d.

Note: Even though SAS can read up to 32 decimal places when you specify some
numeric informats, floating-point numbers with more than 15 decimal places might
lose precision due to the limitations of the eight-byte floating-point representation
used by most computers.

Informs always contain a period (.) as a part of the name. If you omit the w and the d
values from the informat, SAS uses default values. If the data contains decimal points,
SAS ignores the d value and reads the number of decimal places that are actually in
the input data.

For more information about how informats work and a complete list of informats, see the
SAS Formats and Informs: Reference.

---

How Informs Are Used in DS2

DS2 supports SAS informats as follows.

- Both informats supplied by SAS and user-defined informats can be associated with a
column. For information about how to create your own informat in SAS, see PROC
FORMAT in the Base SAS Procedures Guide.

  Note: To create and access user-defined informats, a Base SAS session must be
  available in order to access the SAS catalog file that stores the SAS informat
definitions.

- Only the SAS data set and SPD data sets support storing and retrieving an informat
  with a column.

- Informs can be associated with all data types, but all data types will be converted to
  either CHAR or DOUBLE.

- You can associate SAS informats with a column by using the HAVING clause of the
  DS2 DECLARE statement. For more information, see “How to Specify Informs in
  DS2” on page 854.

For more information and a complete list of informats supplied by SAS, see the section
on informats in the SAS Formats and Informs: Reference.

---

How to Specify Informs in DS2

In DS2, specify informats as an attribute in the HAVING clause of the DECLARE
statement. For example, in the following statement, the column y is declared with the
IEEE8.2 format and the BITS5.2 informat.

dcl double y having format ieee8.2 informat bits5.2;

Note: In DS2, an informat for a column cannot be changed or removed.
Validation of DS2 Informats

Informats are not validated by a data source or applied to a column until execution time. When metadata for a column is requested, the informat name will be returned without validation.

DS2 Informat Examples

dcl char(10) y having label 'varchar' format $5. informat $charzb4.3;
dcl double ssn having format best 10.4 informat comma10.4;
dcl double(6) salary having informat uscurrency.;
dcl char(12) site having informat $city.;
Chapter 25
DS2 Operators

Dictionary

.NEW_ Operator

Constructs an instance of a package.

Note: The escape character ( \ ) before the bracket indicates that the bracket is required in the syntax.

See: The _NEW_ operator for predefined DS2 packages is documented in the reference section for each package.

Syntax

```plaintext
package-variable=_NEW_ [ [THIS\] | [package-instance\] ]
package-name ([constructor-arguments]);
```

Arguments

- **package-variable**
  - specifies a name that can reference an instance of the package.

- **[THIS]**
  - specifies that the package instance has global scope.

  See “Packages and Scope” on page 137

- **[package-instance]**
  - specifies that the new package instance has the same scope as package-instance. package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

  See “Package-Specific Scope” on page 139
**package-name**

specifies the name of the package.

** Requirement**  
package-name must be a DS2 predefined package type or created with a PACKAGE statement before the _NEW_ operator is executed.

**constructor-arguments**

specifies any constructor arguments that are passed to the constructor of the package instance.

**Details**

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

After you have stored methods and variables in a package by using the PACKAGE statement, you can access them by declaring and instantiating the package. If you use the _NEW_ operator to instantiate the package, you must first use the DECLARE PACKAGE statement to declare the package variable.

```
declare package package-name variable-name;
variable-name = _new_ package-name();
```

For example, in the following lines of code, the DECLARE PACKAGE statement tells SAS that C is a variable of type COMPLEX package. The _NEW_ operator constructs an instance of the COMPLEX package and assigns it to the package variable C.

```
declare package complex c;
c = _new_ complex();
```

If you want to initialize package data, you can use a constructor. A constructor is a method that is used to instantiate a package and to initialize the package data. For example, you can provide initialization data by using parameters in the constructor syntax for the hash and hash iterator package.

```
declare package hash h();
h = _new_ hash(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

**Note:** You can use the DECLARE PACKAGE statement with constructor arguments to declare and instantiate a package in one step. The example shown above would look like this.

```
declare package hash h(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

For more information, see Chapter 16, “DS2 Packages,” on page 135 and “DECLARE PACKAGE Statement” on page 884.

**Comparisons**

You can use the DECLARE PACKAGE statement and the _NEW_ operator, or the DECLARE PACKAGE statement alone to declare and instantiate an instance of a package.

**Example**

See “User-Defined Packages” on page 143 and “Predefined DS2 Packages” on page 143.

**See Also**

- “Package Constructors and Destructors” on page 142
• “Packages and Scope” on page 137

**Operators:**

• “_NEW_ Operator, FCMP Package” on page 1001
• “_NEW_ Operator, Hash Package” on page 1044
• “_NEW_ Operator, Hash Iterator Package” on page 1048
• “_NEW_ Operator, Logger Package” on page 1130
• “_NEW_ Operator, Matrix Package” on page 1181
• “_NEW_ Operator, SQLSTMT Package” on page 1228

**Statements:**

• “DECLARE PACKAGE Statement” on page 884
Chapter 26
DS2 Statements

Overview of Statements ................................................. 862
Block Statements ...................................................... 862
  Overview of Block Statements .................................. 862
  Program Block Statements ...................................... 863
  Program Subblock Statements .................................. 864
Global Declaration Statements ...................................... 864
Local Statements ..................................................... 865
Dictionary ............................................................. 866
  ARRAY Assignment Statement .................................. 866
  Assignment Statement ........................................... 866
  BY Statement ..................................................... 868
  CONTINUE Statement ............................................ 873
  DATA Statement .................................................. 874
  DECLARE Statement ............................................. 877
  DECLARE PACKAGE Statement .................................. 884
  DECLARE THREAD Statement .................................... 886
  DO Statement ..................................................... 887
  DROP Statement .................................................. 891
  DROP PACKAGE Statement ...................................... 893
  DROP THREAD Statement ........................................ 893
  DS2_OPTIONS Statement .......................................... 894
  ENDDATA Statement .............................................. 896
  ENDPACKAGE Statement .......................................... 896
  ENDTTHREAD Statement ......................................... 897
  FORWARD Statement ............................................. 898
  GOTO Statement .................................................. 899
  IF Statement, Subsetting ....................................... 900
  IF-THEN/ELSE Statement ....................................... 900
  KEEP Statement .................................................. 902
  Labels Statement ................................................ 903
  LEAVE Statement ................................................ 904
  MERGE Statement ............................................... 906
  METHOD Statement .............................................. 910
  Null Statement ................................................... 915
  OUTPUT Statement ............................................... 916
  PACKAGE Statement ............................................. 921
  PUT Statement ................................................... 925
  RENAME Statement .............................................. 929
  RETAIN Statement ............................................... 930
Overview of Statements

A DS2 statement is a series of items that can include keywords, identifiers, special characters, and operators. A DS2 statement can perform the following actions:

- perform variable assignments
- influence program control
- perform input and output of data
- create methods
- call methods and functions

All DS2 statements end with a semicolon.

There are three categories of statements:

Block
  use to create a programming blocks. For more information, see “Programming Blocks” on page 52.

Global
  use anywhere in the global section of a programming block created by a DATA, PACKAGE, or THREAD statement

Local
  use only inside a DS2 method

Note that some statements belong to multiple categories. For example, the DECLARE statement can be both a global and a local statement.

Block Statements

Overview of Block Statements

There are five DS2 statements that differ from other statements in that they are used to group other statements in programming blocks. The block statements are as follows.

- DO...END
- METHOD...END
- PACKAGE...ENDPACKAGE
- DATA...ENDDATA
Block statements can be divided further into two categories: statements that create program blocks and statements that create program subblocks. Program block statements include PACKAGE...ENDPACKAGE, DATA...ENDDATA, and THREAD...ENDTHREAD. Program subblock statements include METHOD...END and DO...END.

In this documentation, these terms are used for programming blocks.

- A data programming block or data program refers to code bounded by DATA…ENDDATA statements.
- A package programming block or package refers to the stored library of variables and methods bounded by PACKAGE…ENDPACKAGE statements. The variables and methods of a package can be used by DS2 programs, threads, or other packages.
- A thread programming block, or thread program, refers to a stored program bounded by a THREAD…ENDTHREAD statements. The thread program can be called by the SET FROM statement in a DS2 program or package.
- A DO programming block, or DO loop, refers to a subblock of programming statements bounded by DO...END statements.
- A method programming block or method block refers to a subblock of programming statements bounded by a METHOD...END statements.

Each data program must have one and only one program block statement. A data program can and often will have multiple subblock statements. For more information about programming blocks, see “Programming Blocks” on page 52.

**Program Block Statements**

Program blocks are created with the DATA, PACKAGE, and THREAD statements. These statements, and their concluding END statements, are used outside of any other statement, and the program blocks that they define can contain other statements. All other statements must be used inside one of these blocks.

A data program, a package, or a thread program contains two sections: a section of global declarations followed by a section of METHOD statements. This is an example of a data program.

```plaintext
data t;
  ...global declarations
  ...METHOD statement;
enddata;
```

All global declarations must precede the first METHOD statement in the program. A syntax error results if a global declaration is found after a METHOD statement in a program.

The DATA statement creates a data program. A data program consists of the global declaration list and the METHOD statement list contained within a program created by the DATA...ENDDATA statements. For information about compiling and executing a data program, see “DS2” in Base SAS Procedures Guide. For more information about the DATA statement, see the “DATA Statement” on page 874.

A thread program is created by the THREAD...ENDTHREAD statements. A thread program consists of the global declaration list and the METHOD statement list contained within the THREAD...ENDTHREAD statements. The structure of a thread program is essentially the same as that of a data program, but is used to execute several threads in parallel. When you use a DECLARE statement in another data program or package to
reference the thread, the thread program is loaded into memory. You can then use the SET FROM statement in a subsequent data program to run the program in one or more operating system threads. For more information, see the “THREAD Statement” on page 947.

The package is defined by the PACKAGE…ENDPACKAGE statements. A package is a collection of variables and methods that can be called by a data program, a thread program, or another package. A package consists of the global declaration list and the METHOD statement list contained within a programming block created by the PACKAGE…ENDPACKAGE statements. A package is compiled and stored for later use. When you declare the package in a DS2 program, a thread program or another package, the stored package is loaded into memory. You can then access the methods and variables in the package. For more information, see the “PACKAGE Statement” on page 921.

**Program Subblock Statements**

Program subblock statements are created with the METHOD or DO statements. A DS2 program normally contains several subblocks of programming statements.

Each subblock contains two sections: a section of global declaration statements followed by a section of other local statements. This is an example of a METHOD subblock.

```plaintext
method m;
    ...global declarations
    ...local statements;
end;
```

All global declaration statements must precede all other local statements in the subblock. A syntax error will result if a global declaration statement is placed after any other type of statement in a programming subblock.

**Global Declaration Statements**

Global declaration statements are statements that must be in the declaration section of a program created by a DATA, PACKAGE, or THREAD statement. Global declaration statements generally provide information for your DS2 program or request information or data. Generally, global declaration statements are not executable; they take effect as soon as DS2 compiles program statements.

The following table lists DS2 statements that are allowed in the declaration section of a DS2 program.

*Note:* The DECLARE and GLOBAL statements can be used both globally and within a method.

- DECLARE
- DROP
- DROP PACKAGE
- DROP THREAD
- FORWARD
- KEEP
- RENAME
Local Statements

Local statements are statements that you can use inside a programming block created with a METHOD statement.

The following table lists DS2 method statements.

*Note:* All global declaration statements must proceed all other local statements in a method programming block.

*Note:* A METHOD statement is not a local statement. Therefore, a METHOD statement cannot be nested inside another METHOD statement.

- Assignment
- BY
- CONTINUE
- DECLARE
- DECLARE PACKAGE
- DECLARE THREAD
- DO
- GOTO
- IF, Subsetting
- IF-THEN/ELSE
- Labels
- LEAVE
- Null
- OUTPUT
- PUT
- RETURN
- SELECT
- SET
- SET FROM
- STOP
- Sum
Dictionary

ARRAY Assignment Statement

Assigns either a temporary array or a constant list to a temporary array.

Syntax

\[
\text{array-name} := \text{array-name}; \\
\text{array-name} := (\text{constant-list});
\]

Arguments

- `array-name` specifies the name of the array.
- `constant-list` specifies a list that define the array elements.

Details

You can assign either a temporary array or a constant list to a temporary array.

When you assign one array to another array, the data types of the two arrays must be compatible (either the same or convertible). The number of dimensions do not have to be the same for the two arrays, and the total number of elements in each array do not have to be the same.

Example

The following statements are examples of array assignments.

\[
\begin{align*}
\text{ar1} & := (\text{"sales", "inv", "profit"}); \\
\text{ar2} & := (3*3.14159, 2*'5', 2*(1,2), 99); \\
\text{ar7} & := \text{ar2};
\end{align*}
\]

See Also

“Array Assignment” on page 128

Assignment Statement

Evaluates an expression and stores the result in a variable.

Category: Local
Syntax

```
variable = expression;
```

Arguments

**variable**

names a new or existing variable.

**Range**

`variable` can be a variable name, array reference, or SUBSTR function.

**Tip**

Variables that are created by the Assignment statement are not automatically retained.

**expression**

is any valid DS2 expression.

**Tip**

`expression` can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign.

See Chapter 13, “DS2 Expressions,” on page 93

Details

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

The following type conversions take place with the Assignment statement:

- If the variable has a data type and the expression’s data type does not match and can be converted, the expression is converted to the variable's data type. If the expression cannot be converted, an error occurs.

- If the variable does not have a data type, then one of the following actions occurs:
  - If the expression's value is not null and has a data type of CHAR, BINARY, DATE, or TIME, then the variable is given the data type of the expression.
  - If the expression's value is not null and of numeric type, then the variable is given a data type of DOUBLE. If the expression's value is null, then the variable is given a data type of DOUBLE.
  - If the expression's type is CHAR or BINARY, then the variable is given the length of the expression's value. If the expression's value cannot be determined at compile time (for example, for VARCHAR strings), the variable is given the default length of 200.
  - If the expression's type is TIME or TIMESTAMP, then the variable is given the expression's precision.

- If an assignment statement is `a = b = 5`, then `b = 5` is an expression. If `b` is a value other than 5, then `b = 5` is evaluated to 0. Therefore, `a` is assigned a value of 0. The first equal sign (=) is an assignment operator and the second equal sign is a logical equality operator. For more information, see “Example 2: Using an Expression with Multiple Equals Signs” on page 868.

Note: DS2 supports using `eq` as well as the equal sign. For example, `x = y < z < w;` is equivalent to `x = y < z & z < w;`. Another example is that `a = b = c = d;` equates to `a = ((b = c) & (c = d))`;.
Examples

Example 1: Different Types of Expressions
These assignment statements use different types of expressions.

- name='Nagasaki';
- FullName='Mr. '||name;
- price=price+markup;
- declare int i;
  declare double d;
  declare character(200) c;
  i = 5;
  d = 1.2345;
  d = d + i;
  c = 'abc';
  c = d;
  c = '123' || '456';
  i = c;

Example 2: Using an Expression with Multiple Equals Signs
The result of these assignment statements is a=0. The values of b, c, and d are not changed.

```sql
proc ds2;
data;
dcl double a b c d;
method init();
a = b = c = d = 5;
pCLU_all_cu;
end;
enddata;
run;
quit;
```

See Also
Appendix 2, “DS2 Type Conversions for Expression Operands,” on page 1299

BY Statement
Controls the operation of a MERGE or SET statement in a DS2 program and sets up special grouping variables.

Category: Local
Restriction: The BY statement must immediately follow a MERGE or SET statement. The BY statement is optional when using a SET statement.
Tip: Trailing blanks are always ignored when combining tables with a SET or MERGE statement.
Syntax

BY [DESCENDING] column… [DESCENDING] column;

Arguments

DESCENDING
specifies that the tables are sorted in descending order by the variable that is specified. DESCENDING means largest to smallest numerically, or reverse alphabetical for character variables.

Restriction
You cannot use the DESCENDING option with tables that are indexed because indexes are always stored in ascending order.

column(s)
names each column by which the table is sorted. These columns are referred to as BY variables.

Requirement
If you designate a name literal as the BY variable in BY-group processing and you want to refer to the corresponding FIRST. or LAST. temporary variables, you must include the FIRST. or LAST. portion of the two-level variable name within single quotation marks. For example:

data sedanTypes;
method run();
set cars;
by 'Sedan Types'n;
if first.'Sedan Types'n then type=1;
end;
enddata;
run;

Tip
The table can be sorted by more than one column.

Details

How DS2 Indicates the Beginning and End of a BY Group
DS2 indicates the beginning and end of a BY group by creating two temporary variables for each BY variable: FIRST.variable and LAST.variable. The value of these variables is either 0 or 1. DS2 sets the value of FIRST.variable to 1 when it reads the first observation in a BY group, and sets the value of LAST.variable to 1 when it reads the last observation in a BY group. These temporary variables are available for DS2 programming but are not added to the result set.

For a complete explanation of how SAS processes grouped data and of how to prepare your data, see Chapter 20, “Combining Tables,” on page 191.

In a Data Program
The BY statement applies only to the SET or MERGE statement that precedes it in a data program, and only one BY statement can accompany each of these statements in a data program. An error occurs if the BY statement appears anywhere else in the data program.

Note: The BY statement honors the linguistic collation of data that is sorted by using the SORT procedure with the SORTSEQ=LINGUISTIC option.
For more information, see Chapter 20, “Combining Tables,” on page 191.

**Processing BY Groups**

SAS assigns the following values to FIRST.var and LAST.var:

- **FIRST.var** has a value of 1 under the following conditions:
  - when the current row is the first row that is read from the table.
  - FIRST.var has a value of 1 for any preceding variable in the BY statement.

  In all other cases, FIRST.var has a value of 0.

- **LAST.var** has a value of 1 under the following conditions:
  - when the current row is the last row that is read from the table.
  - LAST.var has a value of 1 for any preceding variable in the BY statement.

  In all other cases, LAST.var has a value of 0.

**Examples**

**Example 1: Specifying One or More BY Variables**

- Observations are in ascending order of the variable DEPT:
  
  ```
  by dept;
  ```

- Observations are in alphabetical (ascending) order by CITY and, within each value of CITY, in ascending order by ZIPCODE:
  
  ```
  by city zipcode;
  ```

**Example 2: Specifying Sort Order**

- Observations are in ascending order of SALESREP and, within each SALESREP value, in descending order of the values of JANSALES:
  
  ```
  by salesrep descending jansales;
  ```

- Observations are in descending order of BEDROOMS, and, within each value of BEDROOMS, in descending order of PRICE:
  
  ```
  by descending bedrooms descending price;
  ```

**Example 3: Using a BY Statement When Combining Tables with a SET Statement**

The following example creates two tables and uses a SET statement to combine the tables using the **common** column.

```r
data mrg01a(overwrite=yes);
  dcl varchar(10) common animal;
  method init();
    common='a'; animal='Ant'; output;
    common='b'; animal='Bird'; output;
    common='c'; animal='Cat'; output;
    common='d'; animal='Dog'; output;
    common='e'; animal='Eagle'; output;
    common='f'; animal='Frog'; output;
  end;
enddata;
```
run;

data mrg01b(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Apple'; output;
      common='b'; plant='Banana'; output;
      common='c'; plant='Coconut'; output;
      common='d'; plant='Dewberry'; output;
      common='e'; plant='Eggplant'; output;
      common='f'; plant='Fig'; output;
      common='g'; plant='Grapefruit'; output;
   end;
enddata;
run;

/* set concatenates */
data;
   method run();
      set mrg0la mrg01b; by common;
   end;
enddata;
run;

The following table is output.

<table>
<thead>
<tr>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>a</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>b</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>c</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>f</td>
<td>Frog</td>
<td>Fig</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>Grapefruit</td>
</tr>
</tbody>
</table>
Example 4: Using a BY Statement When Combining Tables with a MERGE Statement

The following example creates two tables and uses a MERGE statement to combine the tables using the common column.

```plaintext
data mrg01a(overwrite=yes);
  dcl char(10) common animal;
  method init();
    common='a'; animal='Ant'; output;
    common='b'; animal='Bird'; output;
    common='c'; animal='Cat'; output;
    common='d'; animal='Dog'; output;
    common='e'; animal='Eagle'; output;
    common='f'; animal='Frog'; output;
  end;
enddata;
run;

data mrg01b(overwrite=yes);
  dcl char(10) common plant;
  method init();
    common='a'; plant='Apple'; output;
    common='b'; plant='Banana'; output;
    common='c'; plant='Coconut'; output;
    common='d'; plant='Dewberry'; output;
    common='e'; plant='Eggplant'; output;
    common='f'; plant='Fig'; output;
    common='g'; plant='Grapefruit'; output;
  end;
enddata;
run;

/* match merge */
data;
  method run();
    merge mrg01a mrg01b; by common;
  end;
enddata;
run;

The following table is output.
CONTINUE Statement

Stops processing the current DO loop iteration and resumes with the next iteration.

**Category:** Local

**Syntax**

CONTINUE;

**Without Arguments**

The CONTINUE statement has no arguments. It resumes processing statements with the next iteration of the DO loop.

**Details**

The CONTINUE statement can appear only in the statement list of an iterative DO loop (for example, DO i=, DO WHILE, or DO UNTIL).

**Comparisons**

- The CONTINUE statement stops the processing of the current iteration of a DO statement and resumes program execution with the next iteration of the current DO statement.
- The LEAVE statement stops the processing of the current DO statement and resumes program execution outside of the current DO statement.

**See Also**

- Chapter 20, “Combining Tables,” on page 191
- “MERGE Statement” on page 906
- “SET Statement” on page 937
Example

This example illustrates the use of the CONTINUE statement. The DO loop iterates and prints the incremented value of \texttt{ctr}. When \texttt{ctr} is equal to 3, the CONTINUE statement causes execution to jump to the next iteration of the DO loop, and prevents \texttt{ctr} from printing.

```sas
data _null_;
dcl int ctr;
method init();
do ctr = 1 to 5;
   if ctr = 3 then continue;
   put ctr;
end;
end;
enddata;
```

The following lines are written to the SAS log.

```
1
2
4
5
```

See Also

Statements:

- “DO Statement” on page 887
- “LEAVE Statement” on page 904

DATA Statement

Begins a DS2 program and provides names for any output tables.

Category: Block

Alias: TABLE

Syntax

```
DATA [ <table-expression> ] [...] <table-expression> ];
   ... program-body ...
[ ENDDATA ; ]
<table-expression>::=
   table (table-options)
   | _ROWSET_ (table-options)
   | _NULL_
```

Without Arguments

If you do not specify any table names with the DATA statement, then the DS2 program returns table rows to the client application and no tables are created.
Arguments

**table**

specifies the name of the table. *table* can be one of these forms.

- `catalog.schema.table-name`
- `schema.table-name`
- `catalog.table-name`
- `table-name`

*catalog* is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

*schema* is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

*table-name* is the name of the table.

**Note**

If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stores as "MyTable". This is important if table and schema names in your data source are case-sensitive.

**CAUTION**

Using the PRESERVE_TAB_NAMES=no option on your LIBNAME statement can cause unexpected results.

**_ROWSET_**

specifies that the DATA statement should not create a table, but it should instead return table rows to the client application.

**_NULL_**

specifies that the DATA statement should not create a table or return rows to the client application.

*table-options*

specifies optional arguments that the DS2 program applies when it writes rows to the output table. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

**Tip**  _NULL_ can be useful in debugging programs when using PUT statements.

Details

A DS2 program begins with the DATA statement and ends with the ENDDATA statement.
A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

If you specify no table names in the DATA statement, or you specify the keyword _ROWSET_, then the DS2 program returns table rows to the client application and no tables are created. If you specify no table names in the DATA statement, at least one global variable is required.

No rows are ever written to the _NULL_ table name. Therefore, if _NULL_ is the only table name present in the DATA statement, the DS2 program does not return any rows.

If any other table names are present, then the program creates a table for each table, and table rows will be written to those tables. For more information, see the “OUTPUT Statement” on page 916.

A warning is issued for tables with delimited column names that are submitted to data sources that are not case sensitive. Data sources that are not case-sensitive will remove the quotation marks and treat the column name as not delimited.

**Comparisons**

For a comparison between packages, DS2 programs, and threads, see “Block Statements” on page 862.

**Examples**

**Example 1: Creating an Output Table**

Use the DATA statement to create one or more output tables. You can use table options to customize the output table. The following DS2 program creates two output tables, EXAMPLE1 and EXAMPLE2. It uses the table option DROP to prevent the column IDNumber from being written to the EXAMPLE2 table.

``` SAS
data example1 example2 (drop=(IDnumber));
  set sample;
  . . .more statements . .
enddata;
```

**Example 2: When Not Creating a Table**

Usually, the DATA statement specifies at least one table name to create an output table. Using the keyword _NULL_ as the table name causes the DS2 program to execute without writing rows to a table. This example writes to the log the value of NAME and ID for each row. An output table is not created.

``` SAS
data _null_;
  set sample;
  put Name ID;
```
DECLARE Statement

Declares one or more DS2 variables or temporary arrays.

Categories:
- Global
- Local

Note: Square brackets in the syntax convention indicate optional arguments. The escape character (\) before a square bracket indicates that the square bracket is required in the syntax. Array bounds must be contained by square brackets ([ ]).

Syntax

```sql
DECLARE <data-type> <variable-list> [having-clause] ;
```

- `<data-type>` ::= 
  `<exact-numeric-type>` | `<approximate-numeric-type>` | `<binary-string-type>` | `<string-type>` | `<date-type>`
  `<exact-numeric-type>` ::= 
    `{ INT | BIGINT | SMALLINT | TINYINT | DECIMAL [(precision [,scale])] | NUMERIC [(precision [,scale])] }`
  `<approximate-numeric-type>` ::= 
    `{ DOUBLE | DOUBLE PRECISION | FLOAT | REAL }`
  `<binary-string-type>` ::= 
    `BINARY(length) | VARBINARY(length)`
  `<string-type>` ::= 
    `NCHAR [(character-length)] | NVARCHAR [(character-length)] | CHAR [(character-length)] [CHARACTER SET character-set-identifier] | VARCHAR [(character-length)] [CHARACTER SET character-set-identifier]`
  `<date-type>` ::= 
    `{ TIME | TIMESTAMP [(precision)] | DATE }`

- `<variable-list>` ::= 
  `{ variable [..variable] }`
  `[ variable <array-declaration> [variable..<array-declaration>] ]`
  `<array-declaration>` ::= 
    `[(<array-bound> [..<array-bound>] )]`
  `<array-bound>` ::= 
    `{ [dim-lower:] dim-upper ] | [dim-lower:] {DIM (a[, n]) | *] }`

See Also

Statements:
- “OUTPUT Statement” on page 916
- “SET Statement” on page 937
<having-clause>::= 
   HAVING <having-option> [...] <having-option> ]
<having-option>::= 
   LABEL 'string' | n'string' 
   | FORMAT format 
   | INFORMAT format

Arguments

INT | BIGINT | SMALLINT | TINYINT
specifies an integer variable or array.

Alias INTEGER for INT

See Chapter 9, “DS2 Data Types,” on page 71

| DECIMAL[precision [, scale]] | NUMERIC[precision [, scale]]
specifies an exact numeric variable or array.

precision
specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

Note Not all data sources can support a precision of 52 digits.

scale
specifies the maximum number of decimal digits that can be stored to the right of the decimal point

Range 0–precision

Note scale is less than or equal to precision.

See Chapter 9, “DS2 Data Types,” on page 71

DOUBLE | DOUBLE PRECISION | FLOAT | REAL
specifies a floating-point variable or array.

See Chapter 9, “DS2 Data Types,” on page 71

BINARY(length)
specifies a binary variable or array.

Requirement If you specify BINARY, you must also specify the length of the variable or array in bytes.

See Chapter 9, “DS2 Data Types,” on page 71

VARBINARY(length)
specifies a varying-length binary variable or array.

Alias BINARY VARYING

Requirement If you specify VARBINARY, you must also specify the length of the binary variable or array in bytes.
NCHAR | NVARCHAR | CHAR | VARCHAR
specifies a character variable or array.

**Aliases**
- NATIONAL CHARACTER, NATIONAL CHAR for NCHAR
- NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR
- CHARACTER for CHAR
- CHARACTER VARYING for VARCHAR

**character-length**
specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.

**Default** 8

**Tip** The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character, for example, UTF-8, then CHAR and NCHAR are identical types and both use NCHAR.

**CHARACTER SET character-set-identifier**
specifies character set encoding information for CHAR and VARCHAR data types.

**Default** Default encoding depends on your operating system and locale.

**Tip** You can use a character string literal or a simple string for character set names. For example, you can specify "ibm-866" or 'ibm-866'.

**See** For a complete list of character set encoding values, see “Character Sets for Encoding in NLS” in the SAS National Language Support (NLS): Reference Guide.

**TIME**
specifies a time variable or array.

**TIMESTAMP**
specifies both a date and time variable or array.

**precision**
specifies the precision for a TIME or TIMESTAMP data type.

**Defaults**
- 0 for time
- 6 for timestamp

**DATE**
specifies a date variable or array.
variable
specifies the scalar variable or array name. You can specify one or more variables or arrays. However, variable can only be of the type specified in data-type. You can mix scalar and array variables of the same type.

dim-lower and dim-upper
specifies a positive or negative integer used to define the number and size of the array boundary.

Tip If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

DIM(a[, n])
specifies that the size of the upper bounds of the array is determined by the number of elements in a dimension of a previously declared array by using a DIM function call.

a
specifies the name of a previously declared array.

n
specifies the dimension, in a multidimensional array, for which you want to know the number of elements.

Tip If no n value is specified, the DIM function returns the number of elements in the first dimension of the array.

Restriction The DIM function is the only function that you can use to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

LABEL 'string' | n'string'
assigns a descriptive label to the variable or array. The label can be a CHAR literal (string) or NCHAR literal (nstring).

See Chapter 9, “DS2 Data Types,” on page 71

FORMAT format
Associates any valid DS2 format with the variable or array.

See Chapter 22, “DS2 Formats,” on page 225

INFORMAT informat
Associates any valid SAS informat with the variable or array.

See Chapter 24, “DS2 Informats,” on page 853

Details

Overview of the DECLARE Statement
The DECLARE statement can be used to specify scalar variables and temporary arrays. More than one variable and array can be specified in a DECLARE statement. For
example, the following DECLARE statement specifies two scalar variables named \( x \) and \( y \) and two temporary arrays named \( a \) and \( b \).

```
declare double a[10] x y b[20];
```

A DECLARE statement associates a data type with each variable in a variable list or an array. In the previous example, \( x \), \( y \), \( a \), and \( b \) have a data type of DOUBLE.

In DS2, the DECLARE statement is also used for package and thread declarations. For more information about package and thread declaration, see the “DECLARE PACKAGE Statement” on page 884 and the “DECLARE THREAD Statement” on page 886.

By default, you receive a warning for any variable that is not declared. If you use a variable without declaring it, DS2 assigns the variable a data type (implicit declaration). The data type for an undeclared variable on the left side of an assignment statement is determined by the data type of the value on the right side of the assignment statement. However, you can use the DS2SCOND system option or the SCOND option on the DS2 procedure to control how DS2 handles an undeclared variable. You can use these options to require the declaration of all table columns and variables. If you specify DS2SCOND=ERROR or SCOND=ERROR, you must use a DECLARE statement for each column or variable. Declaration by assignment does not occur. For more information, see the “DS2SCOND= System Option” on page 966, “DS2” in Base SAS Procedures Guide, and “Variable Declaration” on page 58.

**Scalar Variable Declarations**

Scalar declarations can be used for numeric, character, date, or time data types. You can specify the maximum number of characters a string can contain. Here is an example.

```
declare char(200) s;
```

DS2 imposes no limit on this number, but, in practice, there might be some restriction due to machine limitation. The default length or precision for a particular data type depends on the data source.

For fixed-length character variables, the maximum length is used as the initial (and only) allocation for the string memory. For varying character strings, memory is allocated on an as-needed basis up to the maximum length. There might be an execution-time advantage to using varying character variables because they do not require blank-padding after an operation as fixed-length character variables do.

The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character, for example, UTF-8, then CHAR and NCHAR are identical types and both use NCHAR.

An error occurs if a variable is declared more than once in the same scope, and the declarations are not identical.

If you use a variable without declaring it, it receives the type of the value assigned to it. If no value is assigned, DS2 assigns the variable as type DOUBLE and assigns the value as a missing or null value.

**Temporary Array Variable Declaration**

You use the DECLARE statement to create a temporary array. The elements of a temporary array are temporary in that they are not located in the PDV and therefore do not appear in the result table.
Array declarations are similar to scalar declarations. In addition to the data type and name you also specify the number and size of the array bounds.

Array bounds are given as a signed integer pair, \([l:h]\), where \(l\) represents the lowest index for the given bound and \(h\) represents the highest index for the given bound. An error is returned if \(h<l\). If you specify an array bound with only one integer, then that integer is interpreted as the highest index. The default lowest index is 1.

This example declares an array \(a\) of type double, with five elements indexed from 1 to 5.
\[
declare 
double a[5];
\]

Multiple bounds (or dimensions) are specified using comma separators. This example declares a two dimensional character array \(b\) with 5 elements in the first dimension and 10 elements in the second dimension for a total of 50 elements in the array.
\[
declare 
char b[5,10];
\]

This example declares an array \(c\) with two elements. The array is indexed with a lower bound of 0 and an upper bound of 1.
\[
declare 
int c[0:1];
\]

Temporary arrays exist only for the duration of the DS2 program. For more information, see Chapter 15, “DS2 Arrays,” on page 121 and “Temporary Arrays” on page 122.

**HAVING Clause**

You can associate label, format, and informat attributes with one or more scalar variables or an array. The HAVING clause functions the same as the FORMAT, INFORMAT, and LABEL statements in Base SAS. However, in DS2, the attributes must be specified in the declaration statement of the variable or array.

For more information about how DS2 handles formats and informats, see “Using Formats in DS2” on page 228 and “How Informats Are Used in DS2” on page 854.

For more information about arrays and the HAVING clause, see “Declaring Arrays with a HAVING Clause” on page 126.

**Note:** If variables are declared with a HAVING clause in a thread program and the variables are redeclared in a data program with a HAVING clause, the HAVING clause in the data program is used instead of the HAVING clause in the thread program. If there is no HAVING clause in the DECLARE statement in the data program, the HAVING clause in the thread program is not used.

**Examples**

**Example 1: Declaring Variables**
The following examples illustrate the DECLARE statement.

- declare bigint b2 b3;
- declare double d;
- declare char(200) c1 c2;
- declare varchar vc;
- declare nchar(100) wc;
- declare time(4) tm;
- declare date dt;
• declare varbinary(10) b;
• dcl double x having label 'Amount' format ieee8.2;
• dcl char(10) y having label 'varchar' format $quote.;

**Example 2: Declaring Temporary Arrays**
The following examples illustrate the DECLARE statement for temporary arrays.

• declare double darr[-5:4];
• declare char carr[1:2, 0:3];
• declare int iarr[10] having format octal7.;

**Example 3: Temporary Array Dimensions**
The following table contains examples of statements that specify temporary arrays and the dimensions of those arrays.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Number of Dimensions</th>
<th>Range of Each Dimension</th>
<th>Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>declare double a[100];</td>
<td>1</td>
<td>1:100</td>
<td>100</td>
</tr>
<tr>
<td>declare double a[10, 20, 30];</td>
<td>3</td>
<td>1:10 1:20 1:30</td>
<td>10x20x30 = 6000</td>
</tr>
<tr>
<td>declare double a[5:10];</td>
<td>1</td>
<td>5:10</td>
<td>6</td>
</tr>
<tr>
<td>declare double a[-3:3, 5, 7:9, 10];</td>
<td>4</td>
<td>-3:3 1:5 7:9 1:10</td>
<td>7x5x3x10 = 1050</td>
</tr>
<tr>
<td>declare double a[DIM(u)];</td>
<td>1</td>
<td>1:DIM(u)</td>
<td>DIM(u)</td>
</tr>
<tr>
<td>declare double a[DIM(u,1), 0:DIM(u,2)];</td>
<td>2</td>
<td>1:DIM(u,1) 0:DIM(u,2)</td>
<td>DIM(u,1)x(DIM(u,2)+1)</td>
</tr>
</tbody>
</table>

**See Also**

• “Variable Declaration” on page 58
• Chapter 15, “DS2 Arrays,” on page 121
• “Temporary Arrays” on page 122
• “Declaring Arrays with a HAVING Clause” on page 126

**Statements:**

• “DECLARE PACKAGE Statement” on page 884
• “DECLARE THREAD Statement” on page 886
• “VARARRAY Statement” on page 951

**System Options:**

• “DS2SCOND= System Option” on page 966
DECLARE PACKAGE Statement

Creates a package variable and gives you the option to create an instance of the package.

**Category:** Local

**See:** The DECLARE PACKAGE statement for the predefined DS2 packages is documented in the reference section for each package.

---

**Syntax**

```
DECLARE PACKAGE package [(table-options)] variable [(constructor-arguments)] [...variable [(constructor-arguments)]];  
```

**Arguments**

- `package` specifies the package name. `package` can be one of these forms.
  - `catalog.schema.package`
  - `schema.package`
  - `catalog.package`
  - `package`

- `catalog` is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

- `schema` is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

- `package` is the name of the package.

**Requirements** The package name must match the name of a package created in a PACKAGE statement or be a predefined DS2 package, or an error will occur.

Package naming conventions are based on the data source. For more information, see the documentation for your data source.

**See**

- “PACKAGE Statement” on page 921

**table-options** specifies optional arguments that the DS2 program applies when it creates a package. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.
variable
   specifies a name that can reference an instance of the package.

constructor-arguments
   specifies any constructor arguments that are passed to the constructor of the package instance.

Details
A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

An instance of a package can be constructed either with the _NEW_ operator or with the DECLARE PACKAGE statement. The DECLARE PACKAGE statement with constructor arguments creates a package variable and constructs a package instance:

```plaintext
declare package complex c();
```

The above statement is equivalent to the following two statements:

```plaintext
   declare package complex c; 1
   c = _new_ complex(); 2
```

1 Creates a package COMPLEX variable C that is a null package reference.
2 Constructs a package COMPLEX instance and sets package variable C to reference the constructed package instance.

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also
- “User-Defined Packages” on page 143
- “Package Constructors and Destructors” on page 142

Operators:
- “_NEW_ Operator” on page 857

Statements:
- “DECLARE PACKAGE Statement, FCMP Package” on page 997
- “DECLARE PACKAGE Statement, Hash Package” on page 1012
- “DECLARE PACKAGE Statement, Hash Iterator Package” on page 1020
- “DECLARE PACKAGE Statement, Logger Package” on page 1125
- “DECLARE PACKAGE Statement, Matrix Package” on page 1155
- “DECLARE PACKAGE Statement, SQLSTMT Package” on page 1200
- “PACKAGE Statement” on page 921
DECLARE THREAD Statement

Creates an instance of a thread.

**Category:** Local

### Syntax

```sql
DECLARE THREAD thread [(table-options)] instance( argument ) [... instance ( argument )];
```

### Arguments

- **thread**
  - Specifies the thread name. `thread` can be one of these forms.
  - `catalog.schema.thread`
  - `schema.thread`
  - `catalog.thread`
  - `thread`

- **catalog**
  - An implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

- **schema**
  - An implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

- **thread**
  - The name of the thread.

### Requirements

The thread name must match the name of a thread created in a THREAD statement, or an error will occur.

Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

### See

- “Overview of Threaded Processing” on page 179

### table-options

Specifies optional arguments that the DS2 program applies when it creates a thread. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

### instance

Specifies a name that identifies an instance of the thread.
argument

specifies arguments used with instance.

Details

When a thread is declared, the variable representing the thread can be considered an instance of the thread. This means that two different thread variables represent two completely separate copies of a thread.

Thread variables can appear only in global scope, otherwise an error will be returned. If the thread named in the DECLARE statement does not exist, an error will be returned.

In this example, the thread work.t is instantiated and named thread_name:

```
decare thread work.t thread_name;
```

Once an instance of a thread has been created, you can use it in a SET FROM statement. For more information, see the “THREAD Statement” on page 947. For more information about threads, see “Overview of Threaded Processing” on page 179.

Note: The SAS In-Database Code Accelerator enables you to publish a thread program to the database and execute that thread program in parallel inside the database. For more information about using the SAS In-Database Code Accelerator, see SAS In-Database Products: User’s Guide.

See Also

- Chapter 17, “Threaded Processing,” on page 179

Statements:

- “SET FROM Statement” on page 941
- “THREAD Statement” on page 947

DO Statement

Specifies a group of statements to be executed as a unit.

Category: Local

Syntax

```
DO [ < index-variable-clause > ] [ < conditional-clause > ] ;
    ....statement-list...
END [ end-label ] ;
< index-variable-clause > ::= index-variable = start TO stop [ BY increment ]
< conditional-clause > ::= WHILE ( expression ) | UNTIL ( expression )
```

Without Arguments

The DO statement without clauses is the simplest form of DO group processing. The statements between the DO and END statements are called a DO group. In a simple DO
loop, statements in the DO group are executed one time only. You can nest DO statements within DO groups. A simple DO statement is often used within IF-THEN/ELSE statements to designate a group of statements to be executed depending on whether the IF condition is true or false.

**Arguments**

*statement-list*

specifies any valid DS2 statements.

*end-label*

The END statement closes the DO loop. The optional *end-label* argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the DO statement, or an error will occur. For more information, see the “Labels Statement” on page 903.

**Clauses**

< *index-variable-clause*>  

specifies a numeric scalar expression that determines the number of times that the DO group will be executed.

*index-variable*

names a variable that is used as an index counter for the loop.

**CAUTION:**  

_Avoid changing the index variable within the DO group._ If you modify the index variable within the iterative DO group, you might cause infinite looping.

**Requirement**  
The variable must resolve to a numeric value.

**Tips**  
If the variable is not declared as a local variable, it will end up in the table that is being created unless it is explicitly dropped. For more information about local variables, see “Scope of DS2 Identifiers” on page 52.

The index variable can be a THIS expression. For more information, see “THIS Expression” on page 101.

*start*

specifies the initial value of the index variable. *start* can be any expression that resolves to a numeric value.

**Requirement**

**Notes**  
The DO group is executed first with *index-variable* equal to *start*. The value of *start* is evaluated before the first execution of the loop.

If *index-variable* is an integral type and *start* is floating-point type, the value of *start* is converted to INTEGER type with possible loss of precision.

*TO*  

*stop*

specifies the ending value of the index variable. *stop* can be any expression that resolves to a numeric value.
Notes Execution continues based on the value of increment until one of the following conditions is met: the value of index-variable passes the value of stop, until a WHILE or UNTIL clause that is specified in the DO statement is satisfied, or until a statement in the DO group directs execution out of the loop. The value of stop is evaluated before the first execution of the loop.

If index-variable is an integral type and stop is floating-point type, the value of stop is converted to INTEGER type with possible loss of precision. Any change to stop made within the DO group does not affect the number of iterations.

BY increment specifies a positive or negative value that controls the incrementing or decrementing of index-variable. increment can be any expression that resolves to a numeric value.

Notes The value of increment is evaluated before the execution of the loop. When increment is positive, start must be the lower bound and stop must be the upper bound of the loop. If increment is negative, start must be the upper bound and stop must be the lower bound of the loop. If no increment is specified, the index variable is incremented by 1.

If index-variable is an integral type and increment is floating-point type, the value of increment is converted to INTEGER type with possible loss of precision. Any change to the increment made within the DO group does not affect the number of iterations.

<conditional-clause> specifies a clause that returns true or false.

WHILE ( expression ) causes DO group statements to execute repetitively while a condition is true.

Note A WHILE clause is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. If the expression is false the first time it is evaluated, the DO loop does not iterate even once.

See Chapter 13, “DS2 Expressions,” on page 93

UNTIL ( expression ) causes DO group statements to execute repetitively until a condition is true.

Note An UNTIL clause is evaluated after each execution of the loop, so that the statements inside the group are executed repetitively until the expression is true. The DO loop always iterates at least once.

See Chapter 13, “DS2 Expressions,” on page 93

Details

The DO statement allows a group of statements to be executed as a unit. If iterative or conditional clauses are specified, this group of statements can be executed multiple times.
DO loop iteration with an integral index variable is performed using INTEGER arithmetic. Otherwise, DO loop iteration is performed using DOUBLE arithmetic. Because the representation of the DOUBLE type is a binary number, the accumulation of an imprecise number can introduce enough error to prevent execution of the DO loop the expected number of times. For example, this loop might not execute the expected number of times.

```
do i = 0.001 to 10 by 0.001;
```

For more information, see “Numerical Accuracy in SAS Software” in *SAS Language Reference: Concepts*.

A DO statement defines a scoping block so that any variables declared in the DO statement have scope local to the scope of the DO statement.

There are three forms of the DO statement:

- The DO statement without clauses is the simplest form of DO-group processing. In this form, a group of statements is executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement can execute statements between DO and END statements repetitively, based on the values of an index variable.
- The DO statement can execute statements in a DO loop repetitively while a conditional clause is true, checking the condition before each iteration of the DO loop. The DO UNTIL form evaluates the condition at the bottom of the loop; the DO WHILE form evaluates the condition at the top of the loop.

The final value of the loop is, at most, the specified TO value. For example, `do i = 1 to 10` executes the loop 10 times. This is different from other languages (for example, C, where the last iteration is less than the TO value).

The CONTINUE statement can be used within the DO group to cause execution to immediately continue with the next iteration of the DO statement.

The LEAVE statement can be used within the DO statement to transfer execution to either the statement immediately following a specified target DO statement or the current DO statement.

**Examples**

**Example 1: DO Statement with No Clauses**

```
do;
    dcl int i j;
    i = 2;
    j = i + 5
end;
```

**Example 2: DO with an Index Variable Clause**

- `do j = 1 to 10 by 2;
  i + j;
end;`
- `do k = 11 to 0 by -3;
  i + k;
end;`
- `x = -2;
y = -1;`
do k = 11 to 0 by x + y;
   dcl int i;
   if k < 5 then i = k;
   else
      i = k - 1;
   end;

Example 3: DO Statement with a WHILE Clause
k = 1;
do while(k <= 5);
   k = k + 1;
end;

Example 4: DO Statement with Both an Index Variable and a WHILE Clause
j = -2;
do k = 1 to 10 by 2 while (j <= 0);
   j = j + 1;
end;

Example 5: DO Statement with an UNTIL Clause
k = 1;
do until(k > 6);
   k = k + 1;
end;

See Also

Statements:
• “CONTINUE Statement” on page 873
• “Labels Statement” on page 903
• “LEAVE Statement” on page 904

DROP Statement
Excludes columns from output tables.

Category: Global
Note: This statement cannot be used within a method.

Syntax
DROP column-list;
Arguments

column-list

specifies the name of one or more columns to omit from the output tables.

Restriction  Numbered range lists in the format col{1–coln} and name prefix lists in the format col: are not supported.

Details

The DROP statement applies to all the tables that are created within the same DS2 program and can appear only in the global statements section of a data, thread, or package program. The columns in the DROP statement are available for processing in the DS2 program. If no DROP or KEEP statement appears, all tables that are created in the DS2 program contain all columns. Do not use both DROP and KEEP statements within the same DS2 program.

Comparisons

• The DROP statement applies to all output tables that are named in the DATA statement. To exclude columns from some tables but not from others, use the DROP= table option in the DATA statement.

• The KEEP statement is a parallel statement that specifies a list of columns to write to output tables. Use the KEEP statement instead of the DROP statement if the number of columns to include is significantly smaller than the number to omit.

• The KEEP and DROP statements select columns to include in or exclude from output tables. The subsetting IF statement selects rows.

Example

• These examples show the correct syntax for listing columns with the DROP statement:

  • drop time shift batchnum;

  • drop grade1 grade2 grade3 grade4;

• In this example, the columns PURCHASE and REPAIR are used in processing but are not written to the output table INVENTORY:

  data inventory;
  drop purchase repair;
  method run();
  set table-specification;
  totcost=sum(purchase,repair);
  end;
enddata;

See Also

Statements:

• “KEEP Statement” on page 902

Table Options:
DROP PACKAGE Statement

Deletes a DS2 package.

**Category:** Global

**Restriction:**
This statement must be used outside of any other programming block. Programming blocks are delimited by the DATA...ENDDATA, THREAD...ENDTHREAD, or PACKAGE...ENDPACKAGE statements.

**Syntax**

```
DROP PACKAGE package [(table-options)];
RUN;
```

**Arguments**

*package*

specifies the name of the package to be deleted.

*table-options*

specifies optional arguments that the DS2 program applies when it deletes a package. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

**Details**

The DROP PACKAGE statement drops, or deletes, the table that contains the code for the specified DS2 package. The table must have been previously created with a PACKAGE statement.

*Note:* The RUN statement is required after the DROP PACKAGE statement.

**Example**

These examples show the syntax for dropping a DS2 package:

```
drop package mypkg;
run;
drop package mypkg (pw='n1234');
run;
```

**See Also**

**Statements:**

- “PACKAGE Statement” on page 921

DROP THREAD Statement

Deletes a DS2 program thread.
Category:  Global
Restriction:  This statement must be used outside of any other programming block. Programming blocks are delimited by the DATA...ENDDATA, THREAD...ENDTHREAD, or PACKAGE...ENDPACKAGE statements.

Syntax

DROP THREAD thread [table-options];

Arguments

thread
specifies the name of the thread to be deleted.

table-options
specifies optional arguments that the DS2 program applies when it deletes a thread. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

Details

The DROP THREAD statement drops, or deletes, the table that contains the code for the specified DS2 thread. The table must have been previously created with a THREAD statement.

Example

These examples show the syntax for dropping a DS2 program thread:

drop thread mythread;
drop thread mythread (pw='n1234');

See Also

Statements:
•  “THREAD Statement” on page 947

DS2_OPTIONS Statement

Specifies or changes the default behavior of a DS2 program.

Requirement:  The DS2_OPTIONS statement must appear at the top level of the DS2 program and applies only to the next DATA, PACKAGE, or THREAD statement.

Syntax

DS2_OPTIONS option(s);

Arguments

option(s)
specifies one or more DS2 options. option(s) can be one of the following values:
DIVBYZERO=ERROR | IGNORE
specifies how DS2 processes a division by zero operation.

ERROR
halts row processing and writes an error to the SAS log.

IGNORE
writes a missing or null value to the result set. No message is written to the SAS log.

Default ERROR

MISSING_NOTE
writes a note to the SAS log when an invalid function argument generates a missing value.

Default An error message is written to the SAS log when an invalid function argument generates a missing value.

SAS
specifies that nonexistent values are processed as SAS missing values. This option overrides the ANSIMODE system or DS2 procedure option.

Default By default, DS2 processes nonexistent values as SAS missing values.

Notes You can also specify ANSIMODE in the PROC DS2 statement. For more information, see “PROC DS2 Statement” in Base SAS Procedures Guide.

For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

SCOND
specifies the level of messages that is displayed in the SAS log for the DS2 variable declaration strict mode, which requires that every variable must be declared in the DS2 program. For more information about the DS2 variable declaration strict mode, see “Variable Declaration” on page 58.

WARNING
writes warning messages to the SAS log.

NONE
no messages are written to the SAS log.

NOTE
writes notes to the SAS log.

ERROR
writes error messages to the SAS log.

Default The default is determined by the DS2SCOND= system option. The default for DS2SCOND= is WARNING. For more information, see “DS2SCOND= System Option” on page 966.

Note You can also specify SCOND in the PROC DS2 statement. For more information, see “PROC DS2 Statement” in Base SAS Procedures Guide.

TRACE
provides a trace of what statements are executed.
Requirement  You must have Write permission to the current directory.

Note    Because this option creates a trace of every statement that is executed, there could be a significant performance impact.

Tip     This option is useful when debugging DS2 or in-database operations.

TYPEWARN prints a warning to the SAS log when an implicit type conversion occurs.

Example
Here are some examples:

ds2_options typewarn trace;
ds2_options scond=error;
ds2_options divbyzero=ignore;

ENDDATA Statement
Marks the end of a DATA statement.

Category: Block
Alias: ENDTABLE

Syntax
ENDDATA;

Details
A DS2 program can have multiple package subprograms followed by an optional data program. The following restrictions apply:

• There can be only one data program and the data program must be the last subprogram.

• The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:

• “DATA Statement” on page 874

ENDPACKAGE Statement
Marks the end of a PACKAGE statement.
ENDPACKAGE;

Details
A DS2 program can have multiple subprograms followed by an optional data program. The following restrictions apply:

• There can be only one data program and the data program must be the last subprogram.

• The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:
• “PACKAGE Statement” on page 921

ENDTHREAD Statement
Marks the end of a THREAD statement.

Syntax
ENDTHREAD;

Details
A DS2 program can have multiple thread subprograms followed by an optional data program. The following restrictions apply:

• There can be only one data program and the data program must be the last subprogram.

• The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:
• “THREAD Statement” on page 947
**FORWARD Statement**

Indicates that the method definition follows the method expression.

**Category:** Global

**Syntax**

\[
\text{FORWARD \hspace{0.5em} method [ \ldots \text{method} ];}
\]

**Arguments**

\[\text{method}\]

specifies the name of the method to be defined.

**Details**

When a method definition appears after any method expression that refers to it, a FORWARD statement for the method must be declared before the method expression. Otherwise, the DS2 compiler cannot determine whether the method expression refers to a method.

**Example**

- In this example, the D method is called inside the RUN method. Because the D method is defined after it is called, a FORWARD statement must be specified before the D method is called.

```plaintext
forward d;
method run();
  d = d();
  d = d(100);
end;
method d() returns double;
  return 99;
end;
method d(int y) returns int;
  return 100 + y;
end;
```

- This example creates a user-defined method, SIN, that masks the system function SIN. The user method calls the system function SIN.

```plaintext
forward sin;
method run()
  d = sin(3.14159/2);
  put 'd= ' d;
end;
method sin(double x) returns double;
  return system.sin(x);
end;
```
GOTO Statement

Transfers execution immediately to a labeled statement.

Category: Local

Syntax

GOTO label;

Arguments

label

specifies a statement label that identifies the GOTO destination.

Details

The destination label for the GOTO statement must be within the same DS2 method. You must specify the label argument or an error will occur. Statement labels are defined by using the Labels statement.

Comparisons

GOTO statements can often be replaced by DO-END and IF-THEN/ELSE programming logic.

Example

In this example, when \( x = 2 \), program execution transfers to the DONE label.

```plaintext
method run();
  x = 1;
  do;
    if x=2 then goto done;
    put x;
    x+1;
  end;
  done:
  put x;
end;
```

See Also

Statements:

- “DO Statement” on page 887
- “Labels Statement” on page 903
- “LEAVE Statement” on page 904
IF Statement, Subsetting

Continues processing only those rows that meet the condition.

Category: Local

Syntax

IF expression;

Arguments

expression
is any valid expression that evaluates to true or false.

See Chapter 13, “DS2 Expressions,” on page 93

Details

The expression in a subsetting IF statement is evaluated to produce a result that is either a nonzero value or zero. A nonzero value causes the expression to be true; a result of zero causes the expression to be false.

The subsetting IF statement is equivalent to this IF-THEN statement:

if not (expression)
then return;

If expression is true, DS2 will continue to execute statements in the program.

Note: In logical operations, including the subsetting IF statement, a SAS missing value and a null value evaluate to zero (or false).

Comparisons

Use the IF-THEN/ELSE statement to process statements when both true and false conditions are present or when more processing is required before values are output.

See Also

Statements:

• “IF-THEN/ELSE Statement” on page 900

IF-THEN/ELSE Statement

Executes a statement for observations that meet specific conditions.

Category: Local
Syntax

`IF expression THEN statement;`

`[ ELSE statement ;]`

**Arguments**

*expression*

is any valid expression that evaluates to true or false and is a required argument.

*See Chapter 13, “DS2 Expressions,” on page 93*

*statement*

can be any executable statement or DO group.

**Details**

The expression in an IF-THEN statement is evaluated to produce a result that is either a nonzero value or zero. A nonzero value causes the expression to be true; a result of zero causes the expression to be false.

*Note:* In logical operations, including the IF-THEN/ELSE statement, a SAS missing value and a null value evaluate to zero (or false). To check for a null value in an IF-THEN-ELSE statement, you must use the NULL function as shown in this example.

```plaintext
method init ();
  x=null;
  if (null(x)) then
    put 'null';
  else
    put 'not null';
end;
```

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a statement. An optional ELSE statement gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement.

Using IF-THEN statements *without* the ELSE statement causes all IF-THEN statements to be evaluated. If the IF clause is true, the statement after THEN is executed, otherwise the statement after ELSE is executed.

*Note:* For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

*Note:* You can use an IF expression to select between two values based on whether a conditional evaluates to true or false. In addition, IF expressions can be nested to select between many values for a multi-way decision. For more information, see “IF Expression” on page 102.

**Comparisons**

- Use a SELECT expression rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions. The SELECT expression is evaluated only once, which could result in improved performance.

- Use subsetting IF statements, without a THEN clause, to continue processing only those expressions that evaluate to nonzero values when the condition indicates that no more processing is required and no output is to be produced.
Example

These examples illustrate the IF-THEN/ELSE statement.

- if a = b then
d = e;
else
d = f;

- if status='OK' and type=3 then count+1;

- if x=0 then
  if y ne 0 then put 'X ZERO, Y NONZERO';
  else put 'X ZERO, Y ZERO';
else put 'X NONZERO';

- if answer=9 then
  do;
    answer=.;
    put 'INVALID ANSWER FOR ' id=;
  end;
else
  do;
    answer=answer10;
    valid+1;
  end;

See Also

- “IF Expression” on page 102

Statements:

- “IF Statement, Subsetting” on page 900

KEEP Statement

Includes columns in output tables.

**Category:** Global

**Note:** This statement cannot be used within a method.

**Syntax**

```
KEEP column-list;
```

**Arguments**

`column-list`

specifies the names of one or more columns to write to the output table.

**Details**

The KEEP statement specifies that all columns in the column list should be included in the creation of output rows. If the KEEP attribute is specified, all columns not included in the KEEP statement will be dropped from the output rows. If no DROP or KEEP
statement appears, all tables that are created in the DS2 program contain all columns. Do not use both DROP and KEEP statements within the same DS2 program.

**Comparisons**

- The KEEP statement applies to all output tables that are named in the DATA statement. To write different columns to different tables, you must use the KEEP= table option.
- The DROP statement is a parallel statement that specifies columns to omit from the output table.
- The KEEP and DROP statements select columns to include in or exclude from output tables. The subsetting IF statement selects rows.
- Do not confuse the KEEP statement with the RETAIN statement. The RETAIN statement holds a row value in a column from one iteration of the DS2 RUN method to the next iteration. The KEEP statement does not affect the row values, but only specifies which columns to include in any output tables.

**Example**

This example uses the KEEP statement to include only the columns NAME and AVG in the output table.

```plaintext
data;
  keep name avg;
  method run();
  set table-specification;
  end;
enddata;
```

**See Also**

**Statements:**

- “DROP Statement” on page 891

**Table Options:**

- “KEEP= Table Option” on page 987

---

**Labels Statement**

Identifies a statement that is referred to by another statement.

**Category:** Local

**Syntax**

```
label: statement; [ ... statement ];
```
**Arguments**

*label*

specifies any identifier, which is followed by a colon (:). You must specify the label argument.

**Restriction**

If the label contains non-Latin characters, you must enclose it in double quotation marks.

*statement*

specifies any executable statement, including a null statement (;). You must specify the statement argument.

**Details**

A label associates an identifier with a given statement so that the statement can be referred to by other statements, such as GOTO and LEAVE. You can have multiple labels for a statement.

**Comparisons**

The LABEL attribute of the DECLARE statement's HAVING clause assigns a descriptive label to a column. A statement label identifies a statement or group of statements that are referred to in the same DS2 program by another statement, such as GOTO or LEAVE.

**Example**

- restock:
  
  ```
  if x > 1 then
  y = 3;
  else
  y = 5;
  ```

- label1:

  ```
  label2:
  do i = 1 to 3;
  j = j * i;
  end;
  ```

**See Also**

Statements:

- “GOTO Statement” on page 899
- “LEAVE Statement” on page 904

---

**LEAVE Statement**

Stops processing the current DO loop and transfers execution to either the statement following the current DO statement, or a labeled DO statement that encloses the current DO statement.

**Category:** Local
Syntax

`LEAVE [identifier];`

*Without Arguments*

The `LEAVE` statement stops the processing of the current DO statement and resumes processing with the next statement following the current DO statement.

*Arguments*

`identifier`

label associated with the target DO statement.

*Details*

You can use the `LEAVE` statement to exit a DO loop prematurely. You can use the `LEAVE` statement either on its own or use it based on a condition (for example, in conjunction with an IF statement). If the `LEAVE` statement is not followed by an identifier, then the target is the DO statement immediately enclosing the `LEAVE` statement. If the `LEAVE` statement is followed by an identifier, then the target is the DO statement with the label specified by the identifier. The target DO statement must enclose the current DO statement. An error occurs if the identifier specifies a statement other than a DO statement, or a DO statement that does not enclose the current DO statement. An error also occurs if the specified label does not exist.

*Comparisons*

- The `LEAVE` statement stops the processing of the current DO statement and resumes program execution outside of the current DO statement.
- The `CONTINUE` statement stops the processing of the current iteration of a DO statement and resumes program execution with the next iteration of the current DO statement.

*Examples*

**Example 1: LEAVE Statement without an Identifier**

This example illustrates the `LEAVE` statement without an identifier.

```sas
data _null_;
dcl int i;
method init();
do i = 1 to 10;
   if i > 4 then leave;
   put i;
end;
end;
enddata;
```

The following lines are written to the SAS log.

```
1 2 3 4
```
Example 2: LEAVE Statement with an Identifier
This example illustrates the LEAVE statement with an identifier.

```sas
data _null_;  
dcl int sum i j k;  
method init();  
  lab1: do i = 1 to 5;  
    sum + 1;  
  lab2: do j = 1 to 3;  
    sum + 1;  
    do k = 1 to 3;  
      put sum i j k;  
      sum + 1;  
      if j = 2 then  
        leave lab2;  
      if i = 2 then  
        leave lab1;  
      if k = 2 then  
        leave ;  
    end;  
  end;  
end;  
enddata;
```

The following lines are written to the SAS log.

```
2 1 1 1
3 1 2 2
5 1 2 1
# 2 1 1
```

See Also

Statements:

- “CONTINUE Statement” on page 873
- “DO Statement” on page 887

MERGE Statement

Joins rows from two or more tables into a single row.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restriction:</td>
<td>Tables with the SPDE or HDMD format do not support the MERGE statement.</td>
</tr>
<tr>
<td>Requirement:</td>
<td>The MERGE statement must be followed by a BY statement.</td>
</tr>
<tr>
<td>Interactions:</td>
<td>The MERGE statement is allowed when using the SAS In-Database Code Accelerator. However, if you are using the SAS In-Database Code Accelerator for Hadoop, Hive .13 or later is required. Setting the PROC DS2 BYPARTITION=NO option does not affect a MERGE statement when you are using in-database processing.</td>
</tr>
</tbody>
</table>
Note: The variables that are read using the MERGE statement are set to either a missing or null value.

Tip: The width of the resulting column is determined by the largest width across all the tables in a single SET statement. Trailing blanks are irrelevant to the MERGE statement.

Syntax

MERGE <table-reference> [...<table-reference>];

< table-reference> ::= 
{ table [ (table-option(s)) ] } 
| \{ sql-text \}

Arguments

table  
specifies the name of the table. table can be one of these forms.

• catalog.schema.table-name  
• schema.table-name  
• catalog.table-name  
• table-name

catalog  is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema  is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name  is the name of the table.

Note  If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in data mytable; the table name is stored as "MYTABLE" and in data "MyTable"; the table name is stored as "MyTable". This is important if table and schema names in your data source are case-sensitive.

CAUTION  Using the PRESERVE_TAB_NAMES=no option in your LIBNAME statement can cause unexpected results.

(table-option(s))  
specifies optional arguments that the DS2 program applies when it writes rows to the output table.
Note: For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

{sql-text}
is any valid FedSQL code that resolves to a set of table rows.

Requirement: The FedSQL query must be enclosed in braces ( { } ).

Note: The FedSQL query is specified in the following form: \{SELECT <select-list> FROM <table-specification> WHERE <search-condition>;} . For more information, see the SELECT statement in SAS FedSQL Language Reference.

Tip: You can use sql-text to merge rows from one or more tables.

Details

The MERGE statement is flexible and has a variety of uses in DS2 programming. This section describes basic uses of MERGE. Other applications include using more than one BY variable, merging more than two tables, and merging a few rows with all rows in another table.

Match-merging combines rows from two or more tables into a single row in a new table according to the values of a common variable. The number of row in the new table is the sum of the largest number of rows in each BY group in all tables. To perform a match-merge, use a BY statement immediately after the MERGE statement. The variables in the BY statement must be common to all tables. Only one BY statement can accompany each MERGE statement in a data program.

For more information, see “Match-Merging” on page 212.

CAUTION:
BY variables in a DS2 merge that have a DECIMAL or NUMERIC data type will be converted to a DOUBLE data type. If matching DECIMAL columns are not BY variables, the DECIMAL columns remain as a DECIMAL data type.

CAUTION:
If there is a type, scale, or precision mismatch between columns with a DECIMAL or NUMERIC data type between tables, the column is converted to a DOUBLE data type.

Note: The MERGE statement does not produce a Cartesian product on a many-to-many match-merge. Instead, it performs a sparse one-to-one merge while there are rows in the BY group in at least one table.

Comparisons

With a SET statement, SAS stops processing before all rows are read from all tables if the number of rows are not equal. In contrast, SAS continues processing all rows in all tables that are named in the MERGE statement.

In comparison to DATA step merging, the DS2 MERGE statement is a subset of the Cartesian product. The DATA step merge is not a subset because it uses a lazy retain strategy to fill in the PDV.
Example

The following example creates two tables and uses a MERGE statement to combine the tables using the common column.

```plaintext
data mrg01a(overwrite=yes);
dcl varchar(10) common animal;
method init();
  common='a'; animal='Ant'; output;
  common='b'; animal='Bird'; output;
  common='c'; animal='Cat'; output;
  common='d'; animal='Dog'; output;
  common='e'; animal='Eagle'; output;
  common='f'; animal='Frog'; output;
end;
enddata;
run;

data mrg01b(overwrite=yes);
dcl varchar(10) common plant;
method init();
  common='a'; plant='Apple'; output;
  common='b'; plant='Banana'; output;
  common='c'; plant='Coconut'; output;
  common='d'; plant='Dewberry'; output;
  common='e'; plant='Eggplant'; output;
  common='f'; plant='Fig'; output;
  common='g'; plant='Avocado'; output;
end;
enddata;
run;

/* match merge */
data;
method run();
  merge mrg01a mrg01b; by common;
end;
enddata;
run;

The following table is output.
```
METHOD Statement

Defines a block of code that can be called and executed multiple times.

**Category:** Block

**Syntax**

```plaintext
METHOD method ([IN_OUT] <parameter> [, [IN_OUT]< parameter>])
[RETURNS data-type];
...method-body ...
END;

<parameter>::=
    <data-type> variable
```

See Also

- “Match-Merging” on page 212

Statements:

- “BY Statement” on page 868
- “SET Statement” on page 937
Arguments

**method**

specifies a name for the method. Method names have global scope.

**IN_OUT**

specifies that the argument is to be manipulated by reference, not by value. The IN_OUT parameter manipulates the argument rather than a copy of the argument.

**Restriction**

If the method contains any IN_OUT parameters, the method must not return a value.

**Requirements**

The argument that is passed to the IN_OUT parameter must be a modifiable value, such as an identifier, not an expression.

The data type of the argument must be the same data type as the IN_OUT parameter.

**Tip**

If the method declaration specifies a length for an IN_OUT parameter, a warning is issued and the length is ignored. The length of the supplied argument is always used.

**<parameter>**

specifies a parameter that is passed to the method. The type can be any valid character, numeric, or date type. Parameters have scope that is local to the method and, by default, parameters are passed by value. Unless specified with the IN_OUT parameter, a parameter is initialized to be a copy of the value of the argument that is specified for the parameter.

**Interaction**

If the IN_OUT parameter is specified, the value of the argument that is passed into the IN_OUT parameter might have changed when the method returns.

**INT | BIGINT | SMALLINT | TINYINT**

specifies an integer variable or array.
DECIMAL[(precision [, scale])] | NUMERIC[(precision [, scale])]

specifies an exact numeric variable or array.

**precision**
- specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

  **Note**  
  Not all data sources can support a precision of 52 digits.

**scale**
- specifies the maximum number of decimal digits that can be stored to the right of the decimal point

  **Range**  
  0–precision

  **Note**  
  scale is less than or equal to precision.

DOUBLE | DOUBLE PRECISION | FLOAT | REAL

specifies a floating-point variable or array.

See  
Chapter 9, “DS2 Data Types,” on page 71

BINARY (length)

specifies a binary variable or array.

**Requirement**
If you specify BINARY, you must also specify the length of the variable or array in bytes.

See  
Chapter 9, “DS2 Data Types,” on page 71

VARBINARY (length)

specifies a varying-length binary variable or array.

**Alias**
BINARY VARYING

**Requirement**
If you specify VARBINARY, you must also specify the length of the binary variable or array in bytes.

See  
Chapter 9, “DS2 Data Types,” on page 71

NCHAR | NVARCHAR | CHAR | VARCHAR

specifies a character variable or array.

**Aliases**
NATIONAL CHARACTER, NATIONAL CHAR for NCHAR

NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR

CHARACTER for CHAR

CHARACTER VARYING for VARCHAR
See Chapter 9, “DS2 Data Types,” on page 71

character-length
specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.

Default 8

Tip The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character, for example UTF-8, then CHAR and NCHAR are identical types and both use NCHAR.

CHARACTER SET character-set-identifier
specifies character set encoding information for CHAR and VARCHAR data types.

Default Default encoding depends on your operating system and locale.

Tip You can use a character string literal or a simple string for character set names. For example, you can specify "ibm-866" or 'ibm-866'

See For a complete list of character set encoding values, see “Character Sets for Encoding in NLS” in the SAS National Language Support (NLS): Reference Guide.

TIME
specifies a time variable or array.

TIMESTAMP
specifies both a date and time variable or array.

precision
specifies the precision for a TIME or TIMESTAMP data type.

Defaults 0 for time
       6 for timestamp

DATE
specifies a date variable or array.

variable
specifies the scalar variable or array name. You can specify one or more variables or arrays. However, variable can only be of the type specified in data-type. You can mix scalar and array variables of the same type.

RETURNS data-type
specifies the data type of the value that the method returns. The type can be any valid character, numeric, or date type.

method-body
comprises the variable declarations and executable DS2 code that runs when the method is called. All variables that are declared in the method body are local to the method.

END
marks the end of the method.
Details

There are two types of methods in DS2: system methods, and user-defined methods. The METHOD statement enables you to create your own user-defined methods. For information about system methods, see Chapter 27, “DS2 System Methods,” on page 957.

If a method returns a value, each RETURN statement appearing in the method must have an associated return expression. For more information, see the “RETURN Statement” on page 932.

When a method definition appears after any method expression that refers to it, a FORWARD statement for the method must appear for the method before the method expression. User-defined methods that are not defined before the DS2 INIT, RUN, or TERM methods must be declared in a FORWARD statement. For more information, see the “FORWARD Statement” on page 898.

Note: TINYINT and SMALLINT method parameters will automatically be promoted to INTEGER, and REAL method parameters will automatically be promoted to DOUBLE. A warning message will be given. For more information, see Chapter 12, “DS2 Type Conversions,” on page 87.

Examples

Example 1: User-defined Methods

In these three examples, M, CONCAT, and ADD are user-defined methods.

• method m(int x, int y) returns int;
  return x + y;
end;

• method concat(char(100) x, char(100) y) returns char(200);
  return trim(x) || y;
end;

• method add(double x, double y);
  this.x = this.x + x;
  this.y = this.y + y;
end;

Example 2: Overloaded Methods

In the following examples, the D method and the CONCAT methods are overloaded. If any two method definitions have the same name, but different type signatures (that is, if the methods have different parameter lists), the method is overloaded.

method d(double x, double y) returns double;
  dcl double temp;
  temp = x * 99;
  return x + y + temp;
end;
method d(double x, int y) returns double;
  return x + y
end;
method d(int x);
  put x;
end;
method concat(char(100) x, char(100) y) returns char(200);
  return "pre" || trim(x) || y;
Example 3: Using the IN_OUT Argument
In the following example, the method `swapper` exchanges argument values. The IN_OUT argument enables the values to be changed where the method is called.

```
package xyzzy;
method swapper(in_out double a, in_out double b);
  declare double x;
  x=a; a=b; b=x;
end;
endpackage;
run;
```

data _null_; 
method init();
  dcl package xyzzy x();
  a=10; b=42;
  put 'before: ' a= b=;
  x.swapper(a,b);
  put 'after: ' a= b=;
end;
enddata;
run;

The following lines are written to the SAS log.

```
before:  a=10  b=42
after:  a=42  b=10
```

See Also
- “Methods” on page 51

Statements:
- “FORWARD Statement” on page 898
- “RETURN Statement” on page 932

Methods:
- “INIT Method” on page 958
- “RUN Method” on page 959
- “TERM Method” on page 963

Null Statement
Creates an empty statement.

Category: Local
Syntax
;

Details
The Null statement consists solely of a semicolon. It creates an empty statement.

Example
This example shows how the Null statement can be used to not execute any statements.

```dss
method init();
    x = 1;
    y = 0;
    z = 1;
    if x & y | not z then
        ;
    else
        put 'else';
end;
```

OUTPUT Statement
Writes the current row to a table.

**Category:** Local

**Syntax**

```
OUTPUT [ { table [ … table ] } | _ROWSET_ | _NULL_ ];
```

**Without Arguments**
Using OUTPUT without arguments causes the current row to be written to all tables that are named in the DATA statement or thread program. If no tables are specified in the DATA statement or thread program, then the row is written to the client application.

**Arguments**

- **table**
  specifies the name of the table to which write rows. *table* can be one of these forms.
  - `catalog.schema.table-name`
  - `schema.table-name`
  - `catalog.table-name`
  - `table-name`

- **catalog** is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to...
provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

**schema** is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

**table-name** is the name of the table.

**Restriction**
All names specified in the OUTPUT statement must also appear in the DATA statement.

**Note**
If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stores as "MyTable". This is important if table and schema names in your data source are case-sensitive.

**Tip**
You can specify up to as many tables in the OUTPUT statement as you specified in the DATA statement for that DS2 program.

**CAUTION**
Using the PRESERVE_TAB_NAMES=no option on your LIBNAME statement can cause unexpected results.

**_ROWSET_**
specifies that the OUTPUT statement should not write rows to a table, but it should instead return table rows to the client application.

**_NULL_**
specifies that the OUTPUT statement should not write rows to either a table or the client application.

**Details**

**When and Where the OUTPUT Statement Writes Rows**
The OUTPUT statement creates an output row, using values for the row that are contained in the global variables when output statement executes. The OUTPUT statement writes the current row to a table immediately, not at the end of the DS2 program. If no table name is specified in the OUTPUT statement, the row is written to the table or tables that are listed in the DATA statement.

DS2 keeps track of the values in the order in which the compiler encounters them within a DS2 program, whether they are read from existing tables or created in the program.

**Implicit versus Explicit Output**
If you do not supply an OUTPUT statement, DS2 adds one implicitly at the end of the RUN method that writes rows to the table or tables that are being created.

Placing an explicit OUTPUT statement in a DS2 program overrides the automatic output, and adds a row to a table only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write a row to any one table, however, there is no implicit OUTPUT statement at the end of the RUN method. In this situation, a DS2
program writes a row to a table only when an explicit OUTPUT executes. You can use the OUTPUT statement alone or as part of an IF-THEN/ELSE or SELECT statement or in DO loop processing.

**Using the OUTPUT Statement in DS2 Program Threads**
OUTPUT statements in thread programs cannot contain any table names. Each output row is returned to the thread program that started the thread.

**Comparisons**
- The OUTPUT statement writes rows to a table or to the client application; the PUT statement writes variable values or text strings to the SAS log.
- To control whenever a row is written to a table, use the OUTPUT statement. To control which columns are written to a table, use the KEEP= or DROP= table option in the DATA statement or use the KEEP or DROP statement.

**Examples**

**Example 1: Sample Uses of OUTPUT**
- This line of code writes the current row to a table.
  
  ```
  output;
  ```
- This line of code writes the current row to a table when a specified condition is true.
  
  ```
  if deptcode gt 2000 then output;
  ```
- This line of code writes a row to the MARKUP table when the PHONE value is missing.
  
  ```
  if phone=. then output markup;
  ```

**Example 2: Creating Multiple Rows from Each Row of Input**
You can create two or more rows from each row of input data. This DS2 program creates three rows in the RESPONSE table for each row in the SULFA table:

```
data response(drop= {time1 time2 time3});
  method run();
  set sulfa;
  time=time1;
  output;
  time=time2;
  output;
  time=time3;
  output;
end;
enddata;
```

**Example 3: Creating Multiple Tables from a Single Input Table**
You can create more than one table from one input table. In this example, OUTPUT writes rows to two tables, EASTERN and WESTERN:

```
data eastern western;
  method run();
  set cities;
  if location = 'east' then output eastern;
```
else output western;
end;
enddata;

**Example 4: Creating One Observation from Several Rows of Input**

You can combine several input rows into one row. In this example, OUTPUT creates one row that totals the values of DEFECTS in the first ten rows of the input table:

```plaintext
data discards;
  drop defects;
  method run();
    set gadgets;
    reps+1;
    if reps=1 then total=0;
    total+defects;
    if reps=10 then do;
      output;
      stop;
    end;
  end;
enddata;
```

**Example 5: Output Using Threads**

The following example generates a result set of 20 random numbers. The data program starts 4 threads. Each thread generates and outputs 5 random numbers with variable x. The data program reads the output of each thread with the SET statement and then writes the input random numbers to the data set `random_data`.

```plaintext
/* Thread generates and outputs 5 random numbers */
thread thread_pgm;
  declare double x;
  method init();
    declare int i;
    streaminit(_threadid_);
    do i = 1 to 5;
      x = ranuni(1);
      output;               /* output variable x */
    end;
  end;
endthread;

data random_data;
  dcl thread thread_pgm t;
  method run();
    /* Start 4 threads and read the output of each thread. */
    set from t threads=4;
  end;
enddata;
run;
```

The following output is generated.
### The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.58602</td>
</tr>
<tr>
<td>2</td>
<td>0.78108</td>
</tr>
<tr>
<td>3</td>
<td>0.03955</td>
</tr>
<tr>
<td>4</td>
<td>0.31368</td>
</tr>
<tr>
<td>5</td>
<td>0.73902</td>
</tr>
<tr>
<td>6</td>
<td>0.45233</td>
</tr>
<tr>
<td>7</td>
<td>0.98123</td>
</tr>
<tr>
<td>8</td>
<td>0.06654</td>
</tr>
<tr>
<td>9</td>
<td>0.10132</td>
</tr>
<tr>
<td>10</td>
<td>0.62904</td>
</tr>
<tr>
<td>11</td>
<td>0.51530</td>
</tr>
<tr>
<td>12</td>
<td>0.88424</td>
</tr>
<tr>
<td>13</td>
<td>0.35661</td>
</tr>
<tr>
<td>14</td>
<td>0.11297</td>
</tr>
<tr>
<td>15</td>
<td>0.16502</td>
</tr>
<tr>
<td>16</td>
<td>0.79072</td>
</tr>
<tr>
<td>17</td>
<td>0.90079</td>
</tr>
<tr>
<td>18</td>
<td>0.79053</td>
</tr>
<tr>
<td>19</td>
<td>0.26467</td>
</tr>
<tr>
<td>20</td>
<td>0.22305</td>
</tr>
</tbody>
</table>

### See Also

**Statements:**

- “PUT Statement” on page 925
PACKAGE Statement

Creates a DS2 package.

Category: Block

Syntax

Form 1:  PACKAGE package [/ENCRYPT=SAS | AES] [/table-options];
         ... package-body ...
         ENDPACKAGE;

Form 2:  PACKAGE fcmp-package-name [/ENCRYPT=SAS | AES] [/table-options]
         LANGUAGE=['']FCMP[''] TABLE='library-name';
         ... package-body ...
         ENDPACKAGE;

Arguments

package  specifies the package name. package can be one of these forms.
          • catalog.schema.package
          • schema.package
          • catalog.package
          • package

catalog  is an implementation of the ANSI SQL standard for an SQL catalog, which is a
data container object that groups logically related schemas. The catalog is the
first-level (top) grouping mechanism in a data organization hierarchy that is used
along with a schema to provide a means of qualifying names. A catalog is a
metadata object in a SAS Metadata Repository.

schema  is an implementation of the ANSI SQL standard for an SQL schema, which is a
data container object that groups files such as tables and views and other objects
supported by a data source such as stored procedures. The schema provides a
grouping object that is used along with a catalog to provide a means of qualifying
names.

package  is the name of the package.

Requirement  Package naming conventions are based on the data source. For more
               information, see the documentation for your data source.

/ENCRYPT=SAS|AES

specifies the encryption algorithm. SAS specifies the SAS Proprietary algorithm.
AES specifies the Advanced Encryption Standard (AES) algorithm.
The ENCRYPT option for the PACKAGE statement is different from and has different values than the ENCRYPT= table option. The ENCRYPT= table option affects only SAS output data sets. For more information, see “ENCRYPT= Table Option” on page 981.

/package-options
specifies optional arguments that the DS2 program applies when it creates a package. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

Note Options that are not recognized by DS2 are passed without error to the underlying data source.

package-body
contains the declarations and methods in the package.

fcmp-package-name
specifies the name of the FCMP package.

See “Using the FCMP Package” on page 144

'library-name'
specifies the name of the library where the FCMP function resides.

Requirement This location must be the library.dataset portion of the location that was specified in the OUTLIB option in the PROC FCMP statement.

See The FCMP procedure in Base SAS Procedures Guide

Details

Package Basics
A package is similar to a DS2 program. The package body consists of a set of global declarations and a list of methods. The main syntactical differences are the PACKAGE and ENDPACKAGE statements. These statements define a block with global scope. For more information about scope, see “Scope of DS2 Identifiers” on page 52.

The package's library of methods can be called from any other DS2 program including another package. Consequently, the INIT, RUN, and TERM methods have no special meaning in a package.

After successful compilation of a package, a copy of the package's source code is stored in the catalog entry that is identified by the package name.

The PACKAGE statement is required for all user-defined packages and for the FCMP package that is supplied by SAS. The hash, hash iterator, logger, matrix, and SQLSTMT packages, which are supplied by SAS, do not require a PACKAGE statement. For more information, see “Introduction to DS2 Packages” on page 135.

Packages are declared for use in a DS2 program or another package by using the DECLARE PACKAGE statement. When you declare a package, the variable representing the package is considered an instance of the package.

By default, DS2 packages are encrypted with SAS encryption. You can override this default and specify AES encryption by using the ENCRYPT=AES table option in the PACKAGE statement. SAS Proprietary is a fixed encoding algorithm that is included
with Base SAS software. It requires no additional SAS product licenses. For more information, see *Encryption in SAS*.

Table options can be specified in the PACKAGE statement. They are specified after the package name and preceded by a slash.

**FCMP Packages**

SAS provides an FCMP package that supports calls to FCMP functions and subroutines from within the DS2 language. For more information, see “Using the FCMP Package” on page 144.

**Comparisons**

For a comparison of packages, DS2 programs, and threads, see “Block Statements” on page 862.

**Examples**

**Example 1: Creating a Complex Number Package**

This example creates a very simple complex number package. Two global variables, \(x\) and \(y\), represent the ordered pair of real numbers that constitute a complex number. This set of methods performs various operations on complex numbers, such as add and multiply.

```sas
package complex;
   dcl double x y;
   method init(double x, double y);
      this.x = x;
      this.y = y;
   end;
   method add(double x, double y);
      this.x = this.x + x;
      this.y = this.y + y;
   end;
   method mult(double x, double y);
      this.x = this.x * x - this.y * y;
      this.y = this.x * y + x * this.y;
   end;
   method norm() returns double;
      return sqrt(x ** 2 + y ** 2);
   end;
   method print();
      put 'x = ' x ' y= ' y;
   end;
endpackage;
run;
```

The following DS2 program instantiates and calls the methods defined in the previous PACKAGE statement.

```sas
data _null_;  
   method init();
      dcl package complex c();
      dcl package complex c2();
      c.x=3;
      c.y=4;
```
d = c.norm();
put 'd= ' d;
c.add(5, 6);
c.print();
c2.x=7;
c2.y=24;
d = c2.norm();
put 'd= ' d;
c2.print();
end;
enddata;
run;

These lines are written to the SAS log:

```
    d=  5
    x =  8  y=  10
    d=  25
    x =  7  y=  24
```

**Example 2: Creating an FCMP Package**

This example creates a square routine in FCMP and uses that routine in a DS2 program. The current directory is used as the "library" of FCMP packages.

```
libname base '.';

* fcmp defines a function, square;
proc fcmp outlib = base.fcmpps.package1;
    function square(a);
        return (a*a);
    endsub;
run;

* define the ds2 package thru which the fcmp functions will be called;
proc ds2;
    package pkg /
        overwrite=yes
        language='fcmp'
        table='base.fcmpsubs';
run;

* call fcmp thru the ds2 wrapper package;
data;
    dcl package pkg p();
    dcl double a b;
    method init();
        do a = 10 to 20;
            b=p.square(a);
            put a= b=;
        end;
    end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.
PUT Statement
Prints the values of program variables, arrays, and constants to the SAS log.

Category: Local

Syntax
PUT <put-list> [ ... <put-list> ] ;
<put-list>::=
  _ALL_
  | 'character-string'
  | ["character-string"]<eq-expression> [=] [:] <format [-L | -C | -R]]
<eq-expression>::=
  identifier
  | array-reference
  | this-expression

Without Arguments
PUT without arguments prints a blank line to the SAS log.

Arguments
_ALL_
prints the values of all variables, which includes predefined variables, to the SAS log.

'character-string'
specifies a string of text that is written to the SAS log.
identifier
    names a variable whose value is written to the SAS log.

array-reference
    specifies an array element. The subscript can be any SAS expression that resolves to an integer value when the PUT statement executes. Use the array subscript asterisk (*) to write all elements of the array.

this-expression
    specifies a THIS expression.

See “THIS Expression” on page 101

=          If an equal sign is added after a variable or array element, then the output is preceded by the variable or array element name and an equal sign.

:          enables you to specify a format that the PUT statement uses to write the variable value. All leading and trailing blanks are deleted, and each value is followed by a single blank.

Restriction If you use “:”, you must specify a format.

format
    specifies a format to use when the data value is written to the SAS log. If you use a colon modifier (:) with the format name, all leading and trailing blanks are deleted and each value is followed by a single blank. To override the default alignment, you can add an alignment specification to a format:

- L    left aligns the value.
- C    centers the value.
- R    right aligns the value.

Tip    Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

Details

How to Use the PUT Statement
The PUT statement consists of the keyword PUT followed by a list of variables and constants. You list the names of the variables whose values you want written, or you specify a character string in quotation marks. If you do not specify a format, the PUT statement writes a variable value with the default format, inserts a single blank, and then writes the next value. If you specify a format, the output is written using the format width. Character values are left-aligned in the field; leading and trailing blanks are removed. Numeric values are right-aligned in the field.

How nonexistent data (SAS missing values or null values) are output depends on whether you are in ANSI mode or SAS mode. A period is output for DOUBLE missing dot and null values when the default format, BEST32., is associated with the DOUBLE. For special missing values (.a-.z and ._), the character after the period is output when using BEST32.. For example, if a variable held the value .a, A would be output. INTEGER and other non-DOUBLE numeric types cannot be missing. However, they can be null in which case nothing is output by the PUT statement when the value is null.
For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

**How List Output Is Spaced**

List output uses different spacing methods when it writes variable values and character strings. When a variable is written with list output, SAS automatically inserts a blank space. The output pointer stops at the second column that follows the variable value.

```sas
dcl int a b;
dcl varchar c d;
area = 9924;
city = 1001;
ctry1='Peru';
ctry2='Bolivia';
    put area city ctry1 ctry2;
```

These lines are written to the SAS log.

```
-----+----1----+----2
9924 1001 Peru Bolivia
```

However, when a character string is written, SAS does not automatically insert a blank space. The output pointer stops at the column that immediately follows the last character in the string.

To avoid spacing problems when both character strings and variable values are written, you might want to use a blank space as the last character in a character string.

If you use a colon modifier (:) with the format name, SAS writes the value with the specified format, inserts a blank space, and moves the pointer to the next column.

```sas
dcl int a b;
dcl varchar c d;
area = 9924;
POP = 1000;
ctry1='Peru';
ctry2='Bolivia';
    put area : octal10. pop : comma8.2 ctry1 ctry2;
```

These lines are written to the SAS log.

```
-----+----1----+----2----+----3----+
0000023304 1,000.00 Peru Bolivia
```

**Formatted Output**

You can use a format to control how SAS prints the variable values. The PUT statement uses the format that follows the variable name to write each value. With formatted output, SAS does not automatically add blanks between values. Formatted output moves the pointer the length of the format, even if the value does not fill that length. The pointer moves to the next column; an intervening blank is not inserted. If the value uses fewer columns than specified, character values are left-aligned and numeric values are right-aligned in the field that is specified by the format width.

```sas
dcl int a b;
dcl varchar c d;
pop = 1000
area = 9924;
ctry1='Peru';
ctry2='Bolivia';
    put area octal10. pop comma8.2 ctry1 ctry2;
```
These lines are written to the SAS log.

If no format is specified, the variable's default format is used. You can associate a format with a column by using a HAVING clause in the DECLARE statement. For more information, see “DECLARE Statement” on page 877.

**Comparisons**

The PUT statement and the PUT function have similar behavior. However, the PUT statement directs its results to the SAS log whereas the PUT function returns an NCHAR value containing the result of formatting its argument.

**Examples**

**Example 1: PUT Statements**

This example contains several PUT statements.

```sas
x = 1;
y = 2;
z = 3;
s = 'abc';
a[4] = 99;
put 'x = ' x;
put 'y = ' y;
put z s a[4];
```

*Note:* If an equal sign is added after a variable or array element, then the output is preceded by the variable or array element name and an equal sign. For example, these two code lines are equivalent.

```sas
put x= y= a[2]=;
put 'x=' x ' y=' y ' a[2]=' a[2];
```

**Example 2: Using PUT to Output Arrays**

This example outputs the contents of both temporary and variable arrays.

```sas
data _null_;
declare double a[6] having format 4.2;
vararray double b[2,3];
declare double c[0:1, 2:4, 5:5];

method init();
a := (3 6 9 12 15 18);
b := (3 6 9 12 15 18);
c := (3 6 9 12 15 18);
put a[*]=;
put b[*]=;
put c[*]=;
put 'array a with default format:' a[*];
put 'array a with best3. format:' a[*] best3.;
end;
enddata;
```
Example 3: Using the _ALL_ Argument

This example uses the _ALL_ argument in the PUT statement to print the values of all variables, including the predefined _N_ variable, to the SAS log.

```sas
proc ds2;
data;
dcl double a b c;
method init();
a = 116;
b = 220;
c = 37;
put _all_
end;
enddata;
run;
quit;
```

See Also

- “How to Output Array Content” on page 134

Functions:

- “PUT Function” on page 725

### RENAME Statement

Specifies new names for columns in output tables.

**Category:** Global

**Syntax**

```
RENAME old-name {= | AS} new-name [ ... old-name {= | AS} new-name ];
```
**Arguments**

*old-name*
    specifies the name of a column as it appears in the input table, or in the current DS2 program for newly created columns.

*new-name*
    specifies the name to use in the output table.

**Details**

The RENAME statement enables you to change the names of one or more columns. The new column names are written to the output table only. Use the old column names in programming statements for the current DS2 program. RENAME applies to all output tables. In addition to changing the name of a column, the RENAME statement also changes the label for the column.

**Comparisons**

- The RENAME= table option enables you to specify the columns that you want to rename for each input or output table. Use it in input tables to rename columns before processing.
- If you use the RENAME= table option in an output table, you must continue to use the old column names in programming statements for the current DS2 program. The RENAME= table option affects only that output table. The RENAME statement affects all output tables.
- If you use both the RENAME statement and RENAME= output table option, the RENAME statement has precedence. If X is renamed to Y with a RENAME statement and X is renamed to Z with a RENAME= table option, the RENAME statement takes precedence and X will be renamed to Y.
- The RENAME= table option in the SET statement renames columns in the input table. You must use the new names in programming statements for the current DS2 program.

**Example**

The following examples illustrate the RENAME statement.

- `rename prod=ProductName;`
- `rename street as Address cit as City st as State;`
- `rename oldsalary=newsalary;`

**See Also**

- **Table Option**
  - “RENAME= Table Option” on page 993

---

**RETAIL Statement**

Specifies that all columns or all columns in the column list will have their values retained between executions of the RUN method.
**Syntax**

Form 1: `RETAIN;`

Form 2: `RETAIN column-list;`

Form 3: `RETAIN column-list < constant-value >;`

Form 4: `RETAIN column-list ( < constant-value > … < constant-value > );`

< constant-value > ::= 

- `bit_constant`
- `hex_constant`
- `floating_constant`
- `decimal_constant`
- `sas_missing_value`
- `integer_constant`
- `string_constant`
- `null`
- `DATE character_constant`
- `TIME character_constant`
- `TIMESTAMP character_constant`

**Without Arguments**

(Form 1) If you do not specify any arguments, the RETAIN statement causes the values of all columns to be retained from one execution of the RUN method to the next.

**Arguments**

`column-list`

specifies column names whose values you want retained.

**Details**

**Column Behavior (Form 2)**

The RETAIN statement specifies that all columns in the column list should have their values retained during each execution of the RUN method. Normally, columns in the PDV are set to either a missing or null value before the RUN method executes.

**Assigning Initial Values (Forms 3 and 4)**

Use a RETAIN statement to specify initial values for individual columns, a list of columns, or members of an array. If a value appears in a RETAIN statement, columns that appear before it in the list are set to that value initially.

(Form 3) You can assign one value to all columns. For example, the following statement assigns a value of 100 to columns A, B, and C.

```
retain a b c 100;
```

(Form 4) You can assign different values to each column. For example, the following statement assigns the values 'Vancouver', 'BC', and 'Canada' to columns CITY, PROVINCE, and COUNTRY.
preserve city province country ('Vancouver', 'BC', 'Canada');

Note that you can also use an iterator to assign one value to all columns. For example, the following statement assigns a value of 100 to columns A, B, and C.

```
retain a b c (3* 100);
```

**Redundancy**

It is redundant to name any of these items in a RETAIN statement, because their values are automatically retained from one iteration of the DS2 program to the next:

- columns that are read with a SET statement
  
  *Note:* It might not be redundant if the SET statement has multiple tables associated with it. Assume that table A defines variable X but table B does not. Without a RETAIN statement, variable X is set to missing when records are read from table B. With a RETAIN statement, variable X has whatever value was last assigned to it by table A or by DS2 program logic.

- a column whose value is assigned in a sum statement

- data elements that are specified in an array

**Comparisons**

The RETAIN statement specifies columns whose values are preserved. The KEEP statement specifies columns that are to be included in any table that is being created.

**See Also**

**Statements:**

- “KEEP Statement” on page 902

---

**RETURN Statement**

Returns execution from a method to the method caller.

**Category:** Local

**Syntax**

```
RETURN [ expression ];
```

**Without Arguments**

When a RETURN statement does not have an expression, control is transferred back to the caller of the method in which the RETURN statement is located. No value is returned to the caller of the method.

**Arguments**

*expression*

specifies any valid expression that returns a single value. The expression's type is evaluated, and if necessary, converted to the type specified in the METHOD statement's RETURNS clause. The value of *expression* is then passed back to the caller of the method.
Details

When the RETURN statement is executed in the implicit loop, the next iteration of the implicit loop executes. The RETURN statement transfers control and, if expression is present, returns a value back to the caller of the method. Any method that returns a value (in other words, that has a RETURNS clause in the METHOD statement) must have RETURN expression statement as the last statement in the METHOD body. Otherwise, an error occurs. A warning occurs if a method has a RETURN expression statement, but does not have a RETURNS clause.

You can use the STOP statement to terminate the RUN method.

Example

In this example, the CONCAT method returns a concatenated string. The RETURN statement's type is converted to the type of the RETURNS clause. In this example, the return type is CHAR.

```sql
method concat(char(100) x, char(100) y) returns char(200);
    return trim(x) || y;
end;
```

See Also

Statements:

- “METHOD Statement” on page 910

SELECT Statement

Executes one of several statements or groups of statements.

Syntax

SELECT [( select-expression )];
[ < when-list > [ ...< when-list> ] ];
[ OTHERWISE statement-list ];
END [ end-label ];
<when-list>::=
    WHEN ( when-expression ) [ statement-list ]

Arguments

select-expression
specifies an expression that evaluates to a single value of any type other than VARBINARY.

end-label
The END statement closes the SELECT statement. The optional end-label argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the SELECT statement, or an error will occur.
when-expression
specifies any expression.

Requirement You must specify at least one when-expression.

statement-list
can be any executable statement or statements.

Details

Using WHEN Statements in a SELECT Group
The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify DS2 statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Null statements that are used in WHEN statements cause no further action to be taken when the condition is true.

Note: SELECT statements can be nested.

Note: You can use a SELECT expression to select between multiple expressions based on the values of other expressions. For more information, see “SELECT Expression” on page 104.

Evaluating the when-expression When a select-expression Is Included
If the select-expression is present, both the select-expression and when-expression are evaluated. The two are compared for equality and a value of true or false is returned. If the comparison is true, the statement-list is executed and processing exits the SELECT statement. No other conditions are tested.

If the comparison is false, execution proceeds to the next WHEN statement. If no WHEN statements remain, execution proceeds to the OTHERWISE statement, if one is present. If the result of all SELECT-WHEN comparisons is false and no OTHERWISE statement is present, an error is given and the DS2 program continues to execute.

Evaluating the when-expression When a select-expression Is Not Included
If no select-expression is present, the when-expression is evaluated to produce a result of true or false. If the result is true, the statement-list is executed and processing exits the SELECT statement. No other conditions are tested.

If the result is false, execution proceeds to the next WHEN statement, or to the OTHERWISE statement if one is present. (That is, the action that is indicated in the first true WHEN statement is performed.) If the result of all when-expressions is false and no OTHERWISE statement is present, an error message is issued. If more than one WHEN statement has a true when-expression, only the first WHEN statement is used. Once a when-expression is true, no other when-expressions are evaluated.

Evaluating the when-expression When a statement-list Is Not Included
If a when-expression appears without a statement-list, it uses the statement-list of the next when-expression. The following example produces an output of 10.
a = 10;
select(a);
when(10)
    when(20) put a=

However, a *when-expression* without a *statement-list* breaks this behavior. This example produces no output.

a = 10;
select(a);
when(10) ;
when(20) put a=

*Note:* This is not true for a *when-expression* that precedes OTHERWISE. In this case, a *when-expression* without a statement is treated as if it has an empty statement (;).

**Comparisons**

Use IF-THEN/ELSE statements for programs with few statements. Use subsetting IF statements without a THEN clause to continue processing only those observations or records that meet the condition that is specified in the IF clause.

**Examples**

*Example 1: SELECT with a select-expression*

This example illustrates how to use the SELECT statement with a *select-expression*.

```plaintext
select (a);
    when (1) x=x*10;
    when (2);
    when (3) x=x*100;
    when (4) x=x*100;
    when (5) x=x*100;
    otherwise;
end;
```

*Example 2: SELECT without a select-expression*

This is an example of a SELECT statement without a *select-expression*.

```plaintext
select;
    when (mon in ('JUN', 'JUL', 'AUG')
        and temp>70) put 'SUMMER ' mon=;
    when (mon in ('MAR', 'APR', 'MAY'))
        put 'SPRING ' mon=;
    otherwise put 'FALL OR WINTER ' mon=;
end;
```

*Example 3: SELECT without an IF Expression*

This example uses a SELECT statement. You could also use IF expressions. IF expressions allow a more compact representation of calculations.

```plaintext
tempi=.;
select (age);
    when(12) temp3=0;
    when(13) temp3=-14.2769145744513;
    when(14) temp3=-27.3577925153955;
    when(15) temp3=-21.9485551871151;
```
when(16) temp3=-13.1764092846992;
when(17) temp3=-0.63898626243485;
end;
temp4=.;
select(sex);
when('F') temp4=-1.47186167693037;
when('M') temp4=1.47186167693037;
end;
temp5=.;
select(age);
when(12) DO;
select(sex);
when('F') temp5=0;
when('M') temp5=0;
end;
end;
when(13) DO;
select(sex);
when('F') temp5=7.47012474340756;
when('M') temp5=-7.47012474340756;
end;
end;
when(14) DO;
select(sex);
when('F') temp5=4.35656087162482;
when('M') temp5=-4.35656087162482;
end;
end;
when(15) DO;
select(sex);
when('F') temp5=2.40720037896732;
when('M') temp5=-2.40720037896732;
end;
end;
when(16) DO;
select(sex);
when('F') temp5=7.56020843202274;
when('M') temp5=-7.56020843202274;
end;
end;
when(17) DO;
select(sex);
when('F') temp5=0.0520606347702515;
when('M') temp5=0.0520606347702515;
end;
end;
end;
predictedWeight = -182.556181904311 + temp3 + temp4 + 4.85030791094268*height + temp5;

See Also

• “SELECT Expression” on page 104

Statements:
SET Statement

Reads rows from one or more tables.

**Category:** Local

**Interactions:** Only one SET statement is allowed when using the SAS In-Database Code Accelerator. If more than one SET statement is used in the thread program, the thread program is not run inside the database. Instead, the thread program runs on the client.

In the third maintenance release for SAS 9.4, multi-table SET statements and a SET statement with embedded SQL code are allowed when using the SAS In-Database Code Accelerator. However, if you are using the SAS In-Database Code Accelerator for Hadoop, Hive .13 or later is required.

**Note:** Braces in the syntax convention indicate a syntax grouping. The escape character (\) before a brace indicates that the brace is required in the syntax. sql-text must be enclosed in braces ( { } ).

**Tip:** The width of the resulting column is determined by the largest width across all the tables on a single SET statement. Trailing blanks are irrelevant to the SET statement.

**Syntax**

```
SET < table-reference > [ ... < table-reference > ] [INDSNAME=variable] ;
[ BY [ DESCENDING ] column [ ... [ DESCENDING ] column ] ] ;
< table-reference>::=
   { table [ (table-options) ] }
   | \{ sql-text \}
```

**Arguments**

**INDSNAME=variable**

creates and names a variable that stores the name of the table from which the current row is read. The stored name can be a table name or a physical name. The physical name is the name by which the operating environment recognizes the file.

<table>
<thead>
<tr>
<th>Data type</th>
<th>NCHAR</th>
</tr>
</thead>
</table>

**Tips**

Although the INDSNAME variable is automatically declared as NCHAR, you can explicitly declare it as CHAR.

Unless previously defined, the length of the variable is set to 41 characters. If the variable is declared as CHAR with a specific length, that length is not changed. If the value placed into the INDSNAME variable is longer than that length, then the value is truncated.
column

names each column by which the table is sorted.

Tip The table can be sorted by more than one column.

table

specifies the name of the input table. *table* can be one of these forms.

- `catalog.schema.table-name`
- `schema.table-name`
- `catalog.table-name`
- `table-name`

*catalog* is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

*schema* is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

*table-name* is the name of the table.

Note If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stored as "MyTable". This is important if table and schema names in your data source are case-sensitive.

CAUTION Using the `PRESERVE_TAB_NAMES=no` option in your `LIBNAME` statement can cause unexpected results.

table-options

specifies optional arguments that the DS2 program applies when it writes rows to the output table. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

*sql-text*

is any valid FedSQL code that resolves to a set of table rows.

Requirement The FedSQL query must be enclosed in braces `{ }`.

Tip You can use *sql-text* to merge rows from one or more tables.

See The SELECT statement in *SAS FedSQL Language Reference*
DESCENDING

specifies that the tables are sorted in descending order by the column that is specified. DESCENDING means largest to smallest for numeric columns, or reverse alphabetical for character columns.

Details

What SET Does

The SET statement is flexible and has a variety of uses in DS2 programming. These uses are determined by the options and statements that you use with the SET statement:

• reading rows and columns from existing tables for further processing in a DS2 program

• concatenating and interleaving tables, and performing one-to-one reading of tables

Each time the SET statement executes, one row is read into the program data vector. SET reads all columns one row at a time from the input tables unless you specify otherwise. A SET statement can contain multiple tables; a DS2 program can contain multiple SET statements.

Note: A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

Note: The SET statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

Examples

Example 1: Reading a Table

In this example, each row in the table NC.MEMBERS is read into the program data vector. Only those rows whose value of CITY is Raleigh are output to the new table RALEIGH.MEMBERS:

data raleigh.members;
  method run();
    set nc.members;
      if city='Raleigh';
    end;
enddata;
run;

Example 2: Concatenating Tables

If more than one table name appears in the SET statement, the resulting output table is a concatenation of all the tables that are listed. SAS reads all observations from the first table, and then all from the second table, and so on, until all observations from all the tables have been read. This example concatenates the three tables into one output table named FITNESS:

data fitness;
  method run();
    set health exercise well;
  end;
enddata;
run;
Example 3: Interleaving Tables
To interleave two or more tables, use a BY statement after the SET statement:

```sas
data april;
   method run();
   set payable recvable;
   by account;
   end;
enddata;
run;
```

Example 4: Combining a Single Row with All Rows in a Table
A row to be combined into an existing table can be one that is created by another DS2 program. In this example, the table AVGSALES has only one row:

```sas
data national;
   method init ();
   set avgsales;
   end;
   method run();
   set totsales;
   end;
enddata;
run;
```

Example 5: Reading from the Same Table More Than Once
In this example, SAS treats each SET statement independently. That is, it reads from one table as if it were reading from two separate tables:

```sas
data drugxyz;
   method run();
   set trial5(keep=(sample));
   if sample>2;
      set trial5;
   end;
enddata;
run;
```

For each iteration of the DS2 program, the first SET statement reads one row. The next time the first SET statement is executed, it reads the next row. Each SET statement can read different rows with the same iteration of the DS2 program.

See Also
- “BY-Group Processing When Running Thread Programs inside the Database” in SAS In-Database Products: User's Guide
- “Reading Data Using the SET Statement” on page 186
- “Combining DS2 Tables: Methods” on page 200

Statements:
- “BY Statement” on page 868
- “DATA Statement” on page 874
- “DECLARE PACKAGE Statement, Matrix Package” on page 1155
- “MERGE Statement” on page 906
Table Option:

- “IN= Table Option” on page 986

## SET FROM Statement

Runs a DS2 program as one or more threads.

<table>
<thead>
<tr>
<th>Category:</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restriction:</td>
<td>Multiple SET FROM statements are not allowed in a data program. Otherwise, an error occurs.</td>
</tr>
</tbody>
</table>

### Syntax

```plaintext
SET FROM thread [ THREADS = threads ];
```

### Arguments

- `thread`
  
  Specifies the thread name that is executed by the SET statement. `thread` can be one of these forms.
  
  - `catalog.schema.thread`
  
  - `schema.thread`
  
  - `thread`

- `catalog`
  
  Is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

- `schema`
  
  Is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

- `thread`
  
  Is the name of the thread.

### Requirements

- The thread name must match the name of a thread created in a THREAD statement and the thread must be created before the SET FROM statement is executed, or an error will occur.

- Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

### See

- “Overview of Threaded Processing” on page 179

- `THREADS= threads`
  
  Specifies the number of threads that are run for `thread`. 

---

**SET FROM Statement**

941
Restriction
This argument has no effect if the SAS In-Database Code Accelerator is used to access a database table. The SAS In-Database Code Accelerator always uses either the number of reducers specified in the config file with any maximum limitations that are set by the administrator or one reducer when processing the data program that is forced down to one reducer.

Requirement
threads must be an integer value.

Tip
If threads is not present, the thread runs as a single thread.

See
“THREADS= Argument and the SAS In-Database Code Accelerator” on page 942

Details

The Basics
The SET FROM statement enables a DS2 program to run as a single thread or as multiple threads. The thread name specified in a SET FROM statement references a DS2 program thread that has been created by a THREAD statement.

Note: The SET FROM statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

THREADS= Argument and the SAS In-Database Code Accelerator
If the thread program is run inside the database, the number of threads is set by the SAS In-Database Code Accelerator. The THREADS= argument has no effect if the SAS In-Database Code Accelerator for Greenplum, Hadoop, or Teradata is used to access a database table.

For more information about using the SAS In-Database Code Accelerator, see SAS In-Database Products: User's Guide.

Comparisons
After the thread specified in SET FROM begins execution, the SET FROM statement executes similarly to the SET statement.

Similarities to note are:

• The PDV information for the thread is read by the DS2 program in which the SET FROM statement appears, so that all the output variables from the thread are declared automatically with correct types in the DS2 program.

• SET FROM and SET both loop through input until there are no more rows to read.

Here are differences to notice:

• Instead of reading from tables, the SET FROM statement reads the output from each of the threads.

• In general, the SET FROM statement's input consists of the stream of output produced by all the running threads, via the thread's OUTPUT statement. Because the execution order of threads is unpredictable, the input is not read sequentially like the SET statement reads tables. If the thread contains a SET statement that reads rows from tables, the rows are asynchronously divided among the threads. If a thread is not using a SET statement to read data, then the SET FROM statement's input is
similar to reading one or more copies of a table, but with no given order on the incoming rows.

Examples

Example 1: Running a Single Thread
In this example, threads X and Y are automatically declared in the DS2 program with the appropriate types. When the SET FROM statement executes, T is started as a single thread, its rows are read, and a simple calculation is done for each.

```plaintext
thread work.t;
dcl int x;
dcl double y;
method init();
dcl int i;
do i = 1 to 5;
x = i;
y = i * 2.5;
output;
end;
end;
endthread;
data;
dcl thread work.t t;
method run();
set from t;
sum = x + y;
put ' x= ' x ' y= ' y ' sum= ' sum;
end;
enddata;
The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>x=</th>
<th>y= 2.5</th>
<th>sum=</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>x=</td>
<td>3</td>
<td>7.5</td>
<td>10.5</td>
</tr>
<tr>
<td>x=</td>
<td>4</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>x=</td>
<td>5</td>
<td>12.5</td>
<td>17.5</td>
</tr>
</tbody>
</table>
```

Example 2: Running Multiple Threads
This example modifies a thread, T, to run multiple threads by adding the THREADS option to the SET FROM statement.

```plaintext
thread work.t;
dcl int x;
dcl double y;
method init();
dcl int i;
do i = 1 to 5;
x = i;
y = i * 2.5;
output;
end;
end;
endthread;
data;
```
This runs two threads for T. These lines are written to the SAS log.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7.5</td>
<td>10.5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
<td>17.5</td>
</tr>
<tr>
<td>1</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7.5</td>
<td>10.5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
<td>17.5</td>
</tr>
</tbody>
</table>

In this case, the output is sequential, although there is no guarantee that will happen consistently.

**Example 3: Accumulating Thread Values**

In this example, the thread T outputs a value of 1. In the DS2 program, four threads are started, and all output values are summed and printed in the TERM method.

```sas
thread t;
dcl int x;
method init();
   x = 1;
   output;
end;
endthread;
data;
dcl thread t t;
dcl int sum;
method init();
   sum = 0;
end;
method run();
   set from t threads=4;
   sum + x;
end;
method term();
   put 'sum= ' sum;
end;
enddata;
```

The following line is written to the SAS log.

```
sum=  4
```

**See Also**

- Chapter 17, “Threaded Processing,” on page 179
STOP Statement

Stops execution of the current DS2 program.

**Category:** Local

**Syntax**

```
STOP;
```

**Without Arguments**

The STOP statement causes processing of the current DS2 program to stop immediately and resume processing statements after the end of the current DS2 program.

**Details**

If DS2 outputs a table, the row being processed when STOP executes is not added. The STOP statement can be used alone or in an IF-THEN/ELSE statement or SELECT group.

The TERM method will always execute regardless of the method in which the STOP statement is executed. If you use the STOP statement in the TERM method, the TERM method will stop at the point where the STOP statement is executed.

If the STOP statement is executed in the INIT method or any method that is called from the INIT method, the RUN method will not execute.

Sum Statement

Adds or subtracts the result of an expression to an accumulator variable.

**Category:** Local

**Note:** The Sum statement can be used only with global variables. If you use the Sum statement with local variables, the values are not retained.

**Syntax**

```
variable + expression;
variable – expression;
```

**Arguments**

`variable`

specifies the name of the accumulator variable, which contains a numeric value.
Restrictions

If you use an undeclared array in the Sum statement, an error occurs and a message is written to the SAS log.

The variable name cannot be “x”.

Tips

The variable is automatically set to 0 before DS2 reads the first row. The variable's value is retained from one iteration to the next, as if it had appeared in a RETAIN statement.

To initialize a sum variable to a value other than 0, include it in a RETAIN statement with an initial value.

expression

is any valid DS2 expression.

Tip DS2 treats an expression that produces a missing or null value as zero.

See Chapter 13, “DS2 Expressions,” on page 93

Details

expression is evaluated and the result added to the accumulator variable. When the plus sign (+) is used, the result is added to the accumulator variable. When a minus sign (–) is used, the negative result is added to, in essence subtracted from, the accumulator variable.

Comparisons

The Sum statement is equivalent to using the SUM function and the RETAIN statement, as shown in this example.

```
retain variable 0;
variable=sum(variable,expression);
```

Examples

**Example 1:**
Here are examples of the Sum statement.

```
i + 2;
balance - debit;
umvalid + (not missing(x));
```

**Example 2: Using the Sum Statement with Global and Local Variables**
The following example uses both global and local variables in a Sum statement. Note that the value of the local variable, x, does not change.

```
data _null_
   dcl int y;
   method m()
      dcl int x;
      x = i;
      x + 2;
      y + 4;
```
The following lines are written to the SAS log:

```
x=3 y=4
x=3 y=8
x=3 y=12
```

**THREAD Statement**

Creates a DS2 program thread.

**Category**: Block

**Syntax**

```
THREAD thread [(data-type variable [ , ... data-type variable ] ) ] [/ENCRYPT=SAS | AES][/table-options];
... thread-body ...
ENDTHREAD;
```

**Arguments**

- `thread`
  
specifies the thread name. `thread` can be one of these forms.
  
  - `catalog.schema.thread`
  - `schema.thread`
  - `catalog.thread`
  - `thread`

- `catalog`
  
is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

- `schema`
  
is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a
grouping object that is used along with a catalog to provide a means of qualifying names.

**thread**

is the name of the thread.

**Requirement**

Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

**data-type**

is an optional data type declaration. For more information, see Chapter 9, “DS2 Data Types,” on page 71.

**variable**

names an optional variable that identifies the parameter.

**/ENCRYPT=SAS|AES**

specifies the encryption algorithm. SAS specifies the SAS Proprietary algorithm. AES specifies the Advanced Encryption Standard (AES) algorithm.

**Default**

SAS

**Interaction**

The ENCRYPT option for the PACKAGE statement is different from and has different values than the ENCRYPT= table option. The ENCRYPT= table option affects only SAS output data sets. For more information, see “ENCRYPT= Table Option” on page 981.

**/table-options**

specifies optional arguments that the DS2 program applies when it creates a thread. For more information about table options, see Chapter 29, “DS2 Table Options,” on page 969.

**thread-body**

contains the declarations and methods in the thread.

**Details**

A DS2 thread begins with the THREAD statement and ends with the ENDT THREAD statement. These statements define a block with global scope. For more information about global scope, see “Scope of DS2 Identifiers” on page 52.

The thread body consists of a set of global declarations and a list of methods. You can specify the number of threads used by a thread program by using the SET FROM statement.

A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.
For more information about threads, see “Overview of Threaded Processing” on page 179.

A DS2 thread must be given a name. This name identifies a catalog entry in which the thread's source code is stored after it successfully compiles. Other DS2 programs and threads can then read and execute the thread by using the SET FROM statement.

When a thread is declared, a table is created with the name of the thread. A note is written to the SAS log that indicates that a table was created and where it was created, typically to the Work library. In most situations, a single-level named table does not persist after a SAS session ends. However, some single-level named tables do persist. Tables with multi-level names always persist after a SAS session. If a thread persists, it can be executed multiple times without having to redeclare the thread. You can add a DROP THREAD statement to your program to clean up unwanted tables.

Threads are declared for use in a DS2 program by using the DECLARE THREAD statement. When you declare a thread, the variable representing the thread is considered an instance of the thread. Thread variables can appear only in global scope. Otherwise, an error occurs. For more information about instantiating a thread, see the DECLARE THREAD statement.

Note: If variables are declared with a HAVING clause in a thread program and the variables are redeclared in a data program with a HAVING clause, the HAVING clause in the data program is used instead of the HAVING clause in the thread program. If there is no HAVING clause in the DECLARE statement in the data program, the HAVING clause in the thread program is not used.

Threads can have parameters, as in this example:

```plaintext
thread work.t (double d, char (100) sp);
```

When you are using parameterized threads, the parameter names and their types are specified in the THREAD statement. The DS2 program that calls the thread must initialize the thread's parameters by calling the SETPARMS method. In this example, the parameter D is initialized with a value of 99 and the parameter SP is initialized with a value of 'ijk'.

```plaintext
t.setparms(99, 'ijk');
```

By default, DS2 threads are encrypted with SAS encryption. You can override this default and specify AES encryption by using the ENCRYPT=AES table option in the THREAD statement. SAS Proprietary is a fixed encoding algorithm that is included with Base SAS software. It requires no additional SAS product licenses. For more information, see Encryption in SAS.

Table options can be specified in the THREAD statement. They are specified after the package name and preceded by a slash.

Note: The SAS In-Database Code Accelerator enables you to publish a thread program to the database and execute that thread program in parallel inside the database. For more information about using the SAS In-Database Code Accelerator, see SAS In-Database Products: User’s Guide.

**Comparisons**

For a comparison between packages, DS2 programs, and threads, see “Block Statements” on page 862.
Examples

**Example 1: Simple Thread**
In this example, a single thread is created by using the THREAD statement.

```sas
thread t;
  dcl int x;
  method init();
    x = 99;
    output;
  end;
endthread;
```

**Example 2: Running Multiple Threads**
This example modifies a thread, T, to run multiple threads by adding the THREADS option to the SET FROM statement.

```sas
thread work.t;
  dcl int x;
  dcl double y;
  method init();
    dcl int i;
    do i = 1 to 5;
      x = i;
      y = i * 2.5;
      output;
    end;
  end;
endthread;
data;
  dcl thread work.t t;
  method run();
    set from t threads=2;
    sum = x + y;
    put ' x= ' x ' y= ' y ' sum= ' sum;
  end;
enddate;
```

This runs two threads for T. These lines are written to the SAS log.

```
x= 1  y=  2.5  sum=  3.5
x= 2  y=  5    sum=  7
x= 3  y=  7.5  sum= 10.5
x= 4  y=  10   sum= 14
x= 5  y= 12.5  sum= 17.5
x= 1  y=  2.5  sum=  3.5
x= 2  y=  5    sum=  7
x= 3  y=  7.5  sum= 10.5
x= 4  y=  10   sum= 14
x= 5  y= 12.5  sum= 17.5
```

In this case, the output is sequential, although there is no guarantee that will happen consistently.

**See Also**
- Chapter 17, “Threaded Processing,” on page 179
VARARRAY Statement

Declares one or more DS2 variable arrays.

Syntax

VARARRAY <data-type> array-name <array-declaration> [ <variable-list> ] [ <having-clause> ];

<data-type>::=

<exact-numeric-type> | <approximate-numeric-type> | <binary-string-type> | <string-type> | <date-type>
<exact-numeric-type>::=

{INT | BIGINT | SMALLINT | TINYINT
 | DECIMAL [ (precision [, scale] ) ] | NUMERIC [ (precision [, scale] ) ] }
<approximate-numeric-type>::=

{ DOUBLE | DOUBLE PRECISION | FLOAT | REAL }
<binary-string-type>::=

BINARY (length) | VARBINARY (length)
<string-type>::=

NCHAR [ ( character-length ) ]
 | NVARCHAR [ ( character-length ) ]
 | CHAR [ ( character-length ) ] [ CHARACTER SET character-set-identifier ]
 | VARCHAR [ ( character-length ) ] [ CHARACTER SET character-set-identifier ]
<date-type>::=

{ TIME | TIMESTAMP } [ ( precision ) ] | DATE
<array-declaration>::=[array-bound]>[ ... <array-bound> ]
<array-bound>::=[ [dim-lower:]dim-upper] | {[dim-lower:] {DIM(a[n])} [*]}

<variable-list>::=

name-varlist |
 | numbered-range-varlist
 | name-range-list
 | name-prefix-list
 | type-varlist
 | special-name-list
<having-clause>::=  
  HAVING <having-option> […] <having-option> ]

<having-option>::=  
  LABEL 'string' | n'string'  
  | FORMAT format  
  | INFORMAT format

**Arguments**

**INT | BIGINT | SMALLINT | TINYINT**  
specifies an integer array.

**Alias** INTEGER for INT

**See** [Chapter 9, “DS2 Data Types,” on page 71](#)

**| DECIMAL[ (precision [, scale]) ] | NUMERIC[ (precision [, scale]) ]**  
specifies an exact numeric variable or array.

**precision**  
specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

**Note** Not all data sources can support a precision of 52 digits.

**scale**  
specifies the maximum number of decimal digits that can be stored to the right of the decimal point

**Range** 0–precision

**Note** scale is less than or equal to precision.

**See** [Chapter 9, “DS2 Data Types,” on page 71](#)

**DOUBLE | DOUBLE PRECISION | FLOAT | REAL**  
specifies a floating-point array.

**See** [Chapter 9, “DS2 Data Types,” on page 71](#)

**BINARY (length)**  
specifies a binary variable or array.

**Requirement** If you specify BINARY, you must also specify the length of the variable or array in bytes.

**See** [Chapter 9, “DS2 Data Types,” on page 71](#)

**VARBINARY (length) | BINARY (length)**  
specifies a fixed-length or varying-length binary array.

**Alias** BINARY VARYING

**See** [Chapter 9, “DS2 Data Types,” on page 71](#)

**NCHAR | NVARCHAR | CHAR | VARCHAR**  
specifies a character array.
### Aliases

NATIONAL CHARACTER, NATIONAL CHAR for NCHAR  
NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR  
CHARACTER for CHAR  
CHARACTER VARYING for VARCHAR  

See Chapter 9, “DS2 Data Types,” on page 71

---

<table>
<thead>
<tr>
<th>character-length</th>
<th>specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>8</td>
</tr>
</tbody>
</table>

CHARACTER SET character-set-identifier  
specifies character set encoding information for CHAR and VARCHAR data types.  

<table>
<thead>
<tr>
<th>Default</th>
<th>Default encoding depends on your operating system and locale.</th>
</tr>
</thead>
</table>

Tip You can use a character string literal or a simple string for character set names. For example, you can specify "ibm-866" or 'ibm-866'

See For a complete list of character set encoding values, see “Character Sets for Encoding in NLS” in the SAS National Language Support (NLS): Reference Guide.

---

**TIME**  
specifies a time array.

**TIMESTAMP**  
specifies both a date and time array.

**precision**  
specifies the precision for a TIME or TIMESTAMP data type.

| Defaults | 0 for time  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 for timestamp</td>
</tr>
</tbody>
</table>

**DATE**  
specifies a date array.

**dim-lower and dim-upper**  
specifies a positive or negative integer used to define the number and size of the array boundary.

Tip If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

See “Variable Array Declaration” on page 123

**DIM(a[, n])**  
specifies that the size of the upper bounds of the array is determined by the number of elements in a dimension of a previously declared array by using a DIM function call.


\( a \)

specifies the name of a previously declared array.

\( n \)

specifies the dimension, in a multidimensional array, for which you want to know the number of elements.

**Tip** If no \( n \) value is specified, the DIM function returns the number of elements in the first dimension of the array.

**Restriction** The DIM function is the only function that you can use to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

**See** “DIM Function” on page 478

\( * \)

specifies a one-dimensional array in which the lower bound is 1 and the upper bound is the number of variables in the variable list.

**Requirement** You must specify at least one variable in the variable list.

\(<\text{variable-list}>\)

specifies the name of the variable(s) that is to be referenced by the elements of the array.

**Requirement** \textit{variable} must be the same type specified in \textit{data-type}.

**Tip** You can specify one or more variables.

**See** “Variable Lists” on page 59

\textbf{LABEL 'string' | n'string'}

assigns a descriptive label to the variable array. The label can be a CHAR literal \textit{(string)} or NCHAR literal \textit{(nstring)}.

**See** Chapter 9, “DS2 Data Types,” on page 71s

\textbf{FORMAT format}

Associates any valid DS2 format with the variable or array.

**See** Chapter 22, “DS2 Formats,” on page 225

\textbf{INFORMAT informat}

Associates any valid SAS informat with the variable or array.

**See** Chapter 24, “DS2 Informats,” on page 853

**Details**

You use the VARARRAY statement to create a variable array. A variable array is a temporary grouping of global variables. Only one array can be specified in a VARARRAY statement.

Variable arrays exist only for the duration of the DS2 program.

The different forms of variable lists can be mixed within a single variable list specification. For example, \texttt{vararray double a[*] u x1-x3 u:;} is a valid statement.
The above variable list would expand to \( u \ x_1 \ x_2 \ x_3 \ u \ u_1 \ u_2 \). Therefore, a seven-element variable array would be constructed. Note that a single variable can be referenced by multiple elements of a variable array.

For more information, see Chapter 15, “DS2 Arrays,” on page 121 and “Variable Arrays” on page 123.

For information about how to create a temporary array, see “DECLARE Statement” on page 877.

Example

The following table contains examples of statements that specify variable arrays and the dimensions of those arrays.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Number of Dimensions</th>
<th>Range of Each Dimension</th>
<th>Number of Elements</th>
<th>Referenced Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{vararray double a[100];}</td>
<td>1</td>
<td>1:100</td>
<td>100</td>
<td>( a_1 \ldots a_{100} )</td>
</tr>
<tr>
<td>\texttt{vararray double a[2, 2];}</td>
<td>2</td>
<td>1:2 1:2</td>
<td>4</td>
<td>( a_1 \ a_2 \ a_3 \ a_4 )</td>
</tr>
<tr>
<td>\texttt{vararray double a[-3:3, 5, 7:9, 10];}</td>
<td>4</td>
<td>-3:3 1:5 7:9 1:10</td>
<td>7\times5\times3\times10 = 1050</td>
<td>( a_1 \ldots a_{1050} )</td>
</tr>
<tr>
<td>\texttt{vararray double a[3] x y z;}</td>
<td>1</td>
<td>1:3</td>
<td>3</td>
<td>( x \ y \ z )</td>
</tr>
<tr>
<td>\texttt{vararray double a[3] c3-c1;}</td>
<td>1</td>
<td>1:3</td>
<td>3</td>
<td>( c_3 \ c_2 \ c_1 )</td>
</tr>
<tr>
<td>\texttt{vararray double a[2, 2] t u v w;}</td>
<td>2</td>
<td>1:2 1:2</td>
<td>4</td>
<td>( t \ u \ v \ w )</td>
</tr>
<tr>
<td>\texttt{vararray double a[2, 2] u v2-v4 w1-w3 x;}</td>
<td>3</td>
<td>1:2 1:2 1:2</td>
<td>8</td>
<td>( u \ v_2 \ v_3 \ v_4 \ w_1 \ w_2 \ w_3 \ x )</td>
</tr>
<tr>
<td>\texttt{vararray double a[*] x y z;}</td>
<td>1</td>
<td>1:3</td>
<td>13</td>
<td>( x \ y \ z )</td>
</tr>
<tr>
<td>\texttt{vararray double a[*] a1-a10;}</td>
<td>1</td>
<td>1:10</td>
<td>10</td>
<td>( a_1 \ldots a_{10} )</td>
</tr>
</tbody>
</table>

See Also

- Chapter 15, “DS2 Arrays,” on page 121
- “Variable Arrays” on page 123

Statements:

- “DECLARE Statement” on page 877

VARLIST Statement

Creates a named variable list.

Syntax

\texttt{VARLIST list-name [variable-list];}
Arguments

`list-name` specifies the name of the variable list.

`[variable-list]` specifies the variables that are to be referenced by the list.

**Requirement** The `variable-list` must be enclosed in brackets `[[]]`.

Details

Note that the VARLIST statement is limited to the global scope of the DS2 package or program. The VARLIST statement cannot be used to create a local variable list.

Example

In this example, the VARLIST statement creates a variable list named `allvars`, which contains all the PDV variables in the DS2 program.

```
varlist allvars [_all_];
```

See Also

“Creating Named Variable Lists” on page 62
Overview of System Methods

Methods are basic program execution units. A method defines a scoping block, so any parameters and any variable declarations in the method body are local to the method. In DS2, all program code must reside in some method.

System methods have a preset meaning in DS2. There are three system methods: INIT, RUN, and TERM. These methods cannot be overloaded. There is one optional system method that is used only with threads: SETPARMS.

The following table lists and summarizes the purpose of each DS2 system method.

<table>
<thead>
<tr>
<th>System Method</th>
<th>Execution Details</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT( )</td>
<td>Automatically executes one time, as the first method of a program.</td>
<td>As the name implies, INIT( ) is a good place to initialize global program variables. Most global variables are not initialized by the system. However, the system does initialize predefined variables, such as <em>N</em> and <em>N</em> , and variables that are used in Sum and RETAIN statements. If your program does not require the capabilities of the other system methods, you can code your entire program in the INIT( ) method. Just add DS2 statements, including but not limited to the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DECLARE statements to create method-scope local variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Calls to one or more user-defined methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DS2 statements that perform variable assignments, call DS2 functions, execute loops or other logic, and so on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “INIT Method” on page 958.</td>
</tr>
<tr>
<td>System Method</td>
<td>Execution Details</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| RUN( )        | Automatically executes after INIT( ) completes. | The RUN( ) method is the functional equivalent of the DATA step. That is, if your RUN( ) method contains a SET statement, the method runs as an implicit loop. You can also use RUN( ) to read rows from a thread program using the SET FROM statement. 
  *Note:* You are not required to include code that leverages the implicit loop capabilities. 
  If appropriate, you can code your entire program in the RUN( ) method, as described for INIT( ). 
  For more information, see “RUN Method” on page 959. |
| TERM( )       | Automatically executes one time, as the last method of a program. | As the name implies, TERM( ) is where final processing takes place, before the program exits. 
  TERM( ) automatically resets global variables to uninitialized values, with the following exceptions: 
  - predefined variables, such as _N and _N_ 
  - accumulator variables that were used in Sum statements 
  - variables that were used in a RETAIN statement 
  - package variables 
  If appropriate, you can code your entire program in the TERM( ) method, as described for INIT( ). 
  For more information, see “TERM Method” on page 963. |
| SETPARMS( )   | Executes one time, when called from a data program, to initialize the values of a parameterized thread. | SETPARMS( ) initializes the values of a parameterized thread. Because only parameterized thread programs require this, SETPARMS( ) is the only system method that must be called. 
  *Note:* Do not write a SETPARMS( ) method in your thread program. The system supplies the method for you. 
  *Note:* Do not call SETPARMS( ) more than once. The initialization only works the first time. 
  For more information, see “SETPARMS Method” on page 961. |

For complete information about how methods work in DS2, see “Methods” on page 51.

### Dictionary

#### INIT Method

Calls a DS2 system method where program initializations can take place.

**Syntax**

```plaintext
METHOD INIT();
END;
```
**Without Arguments**
The METHOD INIT statement has no arguments. If you try to pass arguments, an error will occur.

**Details**
Typically, the INIT method will contain any initialization code such as variable initialization or opening of tables. Code in the INIT method will run once at the beginning of the DS2 program.

Every DS2 program will contain, either implicitly or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD INIT statement, DS2 will automatically provide one.

For more information about the INIT method and how DS2 programs work, see “Methods” on page 51.

**Example**

```
method init();
dcl int i;
dcl double d;
d = 99;
do i = 1 to 3;
d = d + i;
   output d;
end;
end;
```

**See Also**
- “Methods” on page 51

**Methods:**
- “RUN Method” on page 959
- “TERM Method” on page 963

---

**RUN Method**
Calls a DS2 system method where DS2 program code can run in an implicit loop.

**Syntax**
```
METHOD RUN();
END;
```

**Without Arguments**
The METHOD RUN statement has no arguments. If you try to pass arguments, an error will occur.
Details

Typically, the RUN method will contain the main DS2 program code. The RUN method has the same feature of automatic, implicit looping as the Base SAS DATA step. After the RUN method has been executed one time, the RUN method either runs again or control is passed to the TERM method.

Every DS2 program will contain, either implicitly or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD INIT or METHOD TERM statement, DS2 will automatically provide one.

After the INIT method runs and before the RUN method is executed, variables in the program data vector, which have not been retained (by using the RETAIN statement), will be set to either SAS missing values or null values depending on whether you are in SAS mode or ANSI mode.

Local variables in the RUN method completely cease to exist between invocations of RUN in the implicit loop. For each invocation of the RUN method, all local variables are constructed at the start of execution of the method and destroyed at end of execution of the method. All global variables, except column variables input by SET statement, are set to missing or null between each iteration (RUN method invocation) of the implicit loop. To retain state data through the implicit loop, you must create a global variable AND also specify that the variable's value be retained across executions of the RUN method with the RETAIN statement. The RUN method will be executed $x+1$ times for a table with $x$ rows. If a SET statement is executed and finds no more rows, then the implicit looping of the RUN method ceases.

For more information about SAS and ANSI mode, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

For more information about the RUN method and how DS2 programs work, see “Methods” on page 51.

Comparisons

In Base SAS, the entire DATA step represents the implicit loop. In the DS2 language, the implicit loop is represented by the RUN method.

Example

DS2's flow of execution is to call the INIT method once, then the RUN method until the input tables are completely read, then the TERM method. The RUN method is where the implicit loop exists. The following program demonstrates this flow of control by finding the minimum of values in a table. The INIT method initializes the columns used to find the current minimum, the RUN method compares input values with the current minimum, and the TERM method outputs the minimums to an output table.

data xy_data;
  dcl double x y;
  method init();
    do x = 1 to 5;
      y = 2*x;
      output;
    end;
  end;
enddata;
run;
/* Find the minimum value for x and y */
data xy_mins;
dcl double min_x min_y;
retain min_x min_y;
keep min_x min_y;
method init();
  min_x = 999999;
  min_y = 999999;
end;
method run();
  set xy_data;
  if x < min_x then min_x = x;
  if y < min_y then min_y = y;
end;
method term();
  output;
end;
enddata;
run;

See Also
• “Methods” on page 51

Methods:
• “INIT Method” on page 958
• “TERM Method” on page 963

SETPARMS Method
Initializes parameters for an instance of a DS2 thread.

Restriction: The SETPARMS method cannot be used with the SAS In-Database Code Accelerator.

Syntax

\[
\text{thread}.\text{SETPARMS}(\text{parameter-value}, \ldots)\]

Arguments

\textit{thread}
  specifies an instance of the thread.

\textit{parameter-value}
  specifies the initial value of the thread parameter.
Details

When using parameterized threads, the parameter names and their types are specified in the THREAD statement. The DS2 program that invokes the thread must initialize the thread's parameters by calling the SETPARMS method. In this example, assume you have an instance of the thread, T, that takes two parameters, INV and PROD.

```plaintext
thread work.t (double inv, char (30) prod);
```

Using the SETPARMS method, the parameter INV is initialized with a value of 38824 and the parameter PROD is initialized with a value of 'rice'.

```plaintext
t.setparms(38824, 'rice');
```

Each argument is passed by value to the corresponding thread parameter. All arguments will be converted, if necessary, to the data type of the corresponding parameter. If the SETPARMS method is called for a thread, which has no parameters, an error will occur.

The SETPARMS method must be called to initialize parameters for a thread before the thread's SET FROM statement executes, or the parameters will initialize with SAS missing values or null values, depending on whether you are in SAS mode or ANSI mode. For more information, see Chapter 11, “How DS2 Processes Nulls and SAS Missing Values,” on page 81.

Example

This example illustrates how to use threads with parameters.

```plaintext
thread work.t (double d, char (100) sp);
dcl int x;
dcl double y;
dcl nchar(20) s;
dcl char(30) c;
method init();
dcl int i;
s = 'abc' || sp;
c = 'uvwxyz' || sp;
do i = 1 to 100;
x = i;
y = i * 2.5 + d;
output;
end;
end;
endthread;
run;
data;
dcl thread work.t t;
method init();
t.setparms(99, 'ijk');
end;
method run();
set from t;
anwer = x + y;
put 's= ' s ' x= ' x ' y= ' y ' c= ' c ' answer= ' answer;
end;
enddata;
run;
```

This is a partial listing of lines that are written to the SAS log:
TERM Method

Calls a DS2 system method where program finalizations can take place.

Syntax

METHOD TERM ();
END;

Without Arguments

The METHOD TERM statement has no arguments. If you try to pass arguments, an error will occur.

Details

Typically, the TERM method will contain any finalization code such as writing data to the SAS log. Code in the TERM method will run once at the end of the DS2 program.

Every DS2 program will contain, either implicitly or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD TERM statement, DS2 will automatically provide one.

For more information about the TERM method and how DS2 programs work, see “Methods” on page 51.

See Also

• “Methods” on page 51

Methods:

• “INIT Method” on page 958
• “RUN Method” on page 959
Overview of System Options

System options are instructions that affect the processing of an entire SAS program or interactive SAS session from the time the option is specified until it is changed.

Here is the syntax for specifying system options in an OPTIONS statement:

OPTIONS options(s);

Here is an explanation of the syntax:

option
    specifies one or more SAS system options that you want to change.

The following example show how to use the system option DS2SCOND in an OPTIONS statement.

options ds2scond=none;

Dictionary

DS2ACCEL= System Option

Specifies whether DS2 code is enabled for parallel processing in supported environments using the SAS In-Database Code Accelerator.

Valid in:
- Configuration file
- SAS invocation
- OPTIONS statement
- SAS System Options window
- Greenplum
- Hadoop
- Teradata

PROC OPTIONS GROUP=

LANGUAGECONTROL

Default:
NONE
DS2ACCEL= System Option

 Specifies the level of messages that PROC DS2 displays in the SAS log for the DS2 variable declaration strict mode, which requires that every variable must be declared in the DS2 program.

**Syntax**

```
DS2SCOND=ERROR | NONE | NOTE | WARNING
```

**Arguments**

- **ERROR**
  - Enables DS2 code to execute in supported parallel environments.

- **NONE**
  -disabled DS2 code from executing in supported parallel environments.

**Details**

The SAS In-Database Code Accelerator enables you to publish a DS2 thread program to the database and execute the thread program in parallel inside the database. If you are using the SAS In-Database Code Accelerator for Teradata or Hadoop, the DS2 data program is also published and executed inside the database.

The DS2ACCEL= system option controls whether DS2 code is executed inside the database.

You can override the DS2ACCEL= system option by specifying the DS2ACCEL= option in the PROC DS2 statement.

**See Also**

- “Using the DS2ACCEL Option to Control In-Database Processing” in *SAS In-Database Products: User’s Guide*

**Procedures:**

- “DS2” in *Base SAS Procedures Guide*
Arguments

ERROR
  writes Error messages to the SAS log.

   Alias   ERR

NONE
  no messages are written to the SAS log.

NOTE
  writes Notes to the SAS log.

WARNING
  writes Warning messages to the SAS log. This is the default.

   Alias   WARN

Details

You can override the DS2SCOND system option by specifying the SCOND= option in the PROC DS2 statement.

See Also

• “Variable Declaration” on page 58

Procedures:

• “DS2” in Base SAS Procedures Guide
Overview of Table Options

Table options are analogous to data set options in Base SAS.

Table options specify actions that apply only to the tables with which they appear. These are some of the operations that table options enable you to perform:

- rename variables
- specify passwords
- specify options for bulk loading data
drop variables from processing or from the result table

Note: Some table options are data source specific. Table options that are not recognized by DS2 are passed without error to the underlying table driver.

Using Table Options in DS2

Table options can be used on these DS2 statements:

- DECLARE PACKAGE
- DECLARE THREAD
- DROP PACKAGE
- DROP THREAD
- PACKAGE
- SET
- DATA
- THREAD

Some table options can apply to packages and threads.

Most table options can apply to either input or output tables. If a table option is associated with an input table, the action applies to the table that is being read. If the option appears in the DATA statement, SAS applies the action to the output table. In DS2, table options for output tables must appear in the DATA statement, not in any OUTPUT statements that might be present.

Some table options, such as COMPRESS=, are meaningful only when you create a SAS data set because they set attributes that exist for the duration of the data set. To change or cancel most table options, you must re-create the table.

When table options appear in both input and output tables in the same DS2 program, first SAS applies table options to input tables. Then SAS evaluates programming statements or applies table options to output tables. Likewise, table options that are specified for the table being created are applied after programming statements are processed. For example, when using the RENAME= table option, the new names are not associated with the columns until the DS2 program is compiled.

In some instances, table options conflict when they are used in the same statement. For example, you cannot specify both the DROP= and KEEP= table options for the same variable in the same statement. Timing can also be an issue in some cases. For example, if you are using KEEP= and RENAME= in a table that is specified in the SET statement, KEEP= must use the original column names. SAS processes KEEP= before the table is read. The new names that are specified in RENAME= apply to the programming statements that follow the SET statement.

Table options are applicable whenever you are reading or writing a table that contains data. Therefore, table options work with DS2 packages and threads because packages and threads are stored in tables.
How to Specify Table Options in DS2

Table options are either enclosed in parentheses or preceded by a forward slash (/), depending on which statement they are used in. Table options should be placed at the end of the statement when they are preceded by the forward slash.

Table options are enclosed in parentheses when used in these statements:

- DECLARE PACKAGE
- DECLARE THREAD
- DROP PACKAGE
- DROP THREAD
- SET
- DATA

If the table option is enclosed in parentheses and the option value can be several items separated by spaces, the option values are also enclosed in parentheses. For examples, see “DS2 Table Option Examples” on page 971.

Table options are preceded by a forward slash (/) when used in these statements:

- PACKAGE
- THREAD

DS2 Table Option Examples

The following are examples of table options that are enclosed in parentheses:

data a {bufno=10};
data prod (drop={price sales});
declare package (pw=lk34890f) sales;
drop thread (write=24klj) complex;

The following are examples of table options that are preceded by a forward slash (/):

package invent /overwrite=yes;
thread work.t (double d, char (100) sp) /read=44kl7;
Dictionary

**ALTER= Table Option**

Assigns an ALTER password to a data set that prevents users from replacing or deleting the file, and enables access to a Read- and Write-protected file.

**Data source:** SAS data set

**Note:** Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts.*

**Syntax**

`ALTER=alter-password`

**Arguments**

`alter-password` must be a valid SAS name.

**Details**

The ALTER= option applies only to a SAS data set. You can use this option to assign a password or to access a read-protected, write-protected, or alter-protected file. When you replace a data set that is protected with an ALTER password, the new data set inherits the ALTER password.

The password is blotted out when the code is written in the SAS log. Here is an example:

```
set a(alter=XXXXXXX);
```

**Note:** A SAS password does not control access to a SAS file beyond the SAS System. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

**BUFNO= Table Option**

Specifies the number of buffers to be allocated for processing a SAS data set.

**Data source:** SAS data set

**Syntax**

`BUFNO= n | nK | hexX | MIN | MAX`
**Arguments**

$n | nK$

specifies the number of buffers in multiples of 1 (bytes); 1,024 (kilobytes). For example, a value of 8 specifies 8 buffers, and a value of 1k specifies 1024 buffers.

`hex`

specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, the value 2dx sets the number of buffers to 45 buffers.

**MIN**

sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default.

**MAX**

sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is $2^{31}$-1, or approximately 2 billion.

**Details**

The buffer number is not a permanent attribute of the data set; it is valid only for the current SAS session or job.

BUFNO= applies to SAS data sets that are opened for input, output, or update.

A larger number of buffers can speed up execution time by limiting the number of input and output (I/O) operations that are required for a particular SAS data set. However, the improvement in execution time comes at the expense of increased memory consumption.

To reduce I/O operations on a small data set as well as speed execution time, allocate one buffer for each page of data to be processed. This technique is most effective if you read the same observations several times during processing.

**Comparisons**

- If the BUFNO= table option is not specified, then the value of the BUFNO= system option is used. If both are specified in the same SAS session, the value specified for the BUFNO= table option overrides the value specified for the BUFNO= system option.

- To request that SAS allocate the number of buffers based on the number of data set pages and index file pages, use the SASFILE global statement.

---

**BUFSIZE= Table Option**

Specifies the size of a permanent buffer page for an output SAS data set.

**Restriction:**

Use with output data sets only.

**Data source:**

SAS data set

**Syntax**

`BUFSIZE= n | nK | nM | nG | hexX | MAX`
**Arguments**

\[ n \mid nK \mid nM \mid nG \]

specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of 8 specifies a page size of 8 bytes, and a value of 4k specifies a page size of 4096 bytes.

**hexX**

specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, the value 2dx sets the page size to 45 bytes.

**MAX**

sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is \(2^{31}-1\), or approximately 2 billion bytes.

**Details**

The page size is the amount of data that can be transferred for a single I/O operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

To change the page size, use the COPY procedure to copy the data set and either specify a new page or use the SAS default. To reset the page size to the default value in your operating environment, use BUFSIZE=0.

**Note:** If you use the COPY procedure to copy a data set to another library that is allocated with a different engine, the specified page size of the data set is not retained.

**Operating Environment Information**

The default value for BUFSIZE= is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=. For the default setting and possible settings for direct access, see the BUFSIZE= option in the SAS documentation for your operating environment.

**Comparisons**

If the BUFSIZE= table option is not specified, then the value of the BUFSIZE= system option is used. If both are specified in the same SAS session, the BUFSIZE= table option overrides the value specified for the BUFSIZE= system option.

---

**BULKLOAD= Table Option**

Loads rows of data as one unit.

**Default:** NO

**Data source:** Aster, DB2 UNIX and PC, Greenplum, Hadoop, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
Syntax

BULKLOAD=YES | NO

Arguments

YES
calls a DBMS-specific bulk-load facility in order to insert or append rows to a DBMS table.

NO
uses the dynamic SAS/ACCESS engine to insert or append data to a DBMS table.

Details

Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

When BULKLOAD=YES, the first error encountered causes the remaining rows (including the erroneous row) in the buffer to be rejected. No other errors within the same buffer will be detected. In addition, all rows before the error are committed, even if DBCOMMIT= is set larger than the number of the erroneous row.

COMPRESS= Table Option

Specifies how observations are compressed in a new output SAS data set.

Restriction: Use with output data sets only.
Data source: SAS data set

Syntax

COMPRESS=NO | YES | CHAR | BINARY

Arguments

NO
specifies that the observations in a newly created SAS data set are uncompressed (fixed-length records).

YES | CHAR
specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RLE (Run Length Encoding). RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations.

Alias ON

Note COMPRESS=CHAR is accepted by SAS 7 and later.

Tip Use this compression algorithm for character data.

BINARY
specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file.
Tip  This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

Details
Compressing a file is a process that reduces the number of bytes required to represent each observation. Advantages of compressing a file include reduced storage requirements for the file and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed file (because of the overhead of uncompressing each observation), and there are situations where the resulting file size might increase rather than decrease.

After a file is compressed, the setting is a permanent attribute of the file, which means that to change the setting, you must re-create the file. That is, to uncompress a file, copy the file and specify COMPRESS=NO.

Comparisons
The COMPRESS= table option overrides the COMPRESS= option in the LIBNAME statement.

When you create a compressed file, you can also specify REUSE=YES table option in order to track and reuse space. With REUSE=YES, new observations are inserted in space freed when other observations are updated or deleted. When the default REUSE=NO is in effect, new observations are appended to the existing file.

**DBCREATE_TABLE_OPTS= Table Option**

Specifies DBMS-specific options to be added to the DATA statement.

**Data source:** Aster, DB2 UNIX, Greenplum, Hadoop, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

**Syntax**

\[ DBCREATE_TABLE_OPTS= 'DBMS-option(s)' \]

**Arguments**

\[ DBMS-option(s) \]

specifies one or more valid DBMS-specific options. If more than one option is specified, the options should be separated in the same way as options are separated in the DBMS.

**Details**

**Basics**

This option enables you to add DBMS-specific options to the DATA statement. The interface passes the DATA statement and its clauses to the DBMS, which executes the statement and creates the DBMS table. An example of this would be to use the COLUMN-DELIMITER= option to specify a column delimiter for Hadoop files.
Note: If the SAS ACCESS DBCREATE_TABLE_OPTS LIBNAME option and the DBCREATE_TABLE_OPTS table option are used, the table option setting takes precedence.

Quoting DBMS Options
The DBMS-options must be enclosed with single-quotation marks. Elements within DBMS-options that are quoted should use double the quotation marks around the element. For example, if single quotation marks are used, you use two single quotation marks. If double quotation marks are used, you use two sets of double quotation marks. Here are some examples:

/* Using double quotes around DBMS-option element causes an error */
DBCREATE_TABLE_OPTS="FIELDS TERMINATED BY '\012'"

/* Using single quotes around DBMS-option element works */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY '\012' '

/* You can double the quote character to insert one into a quoted value*/

/* doubled single quote passes single quote to database */
/* trailing space after last set of single quotes added for emphasis */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY ''\012'' '

/* doubled double quotes passes double quotes to database */
/* trailing space after last set of double quotes added for emphasis */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY '""\012""'

Examples

Example 1
In the following example, the Teradata table Teralib is created with the value of the primary index () option appended to the DATA statement.

libname teralib teradata server=terasvr database=model user=myid password=xxxx;
proc delete data=teralib.outtable;
proc ds2;
data teralib.outtable(overwrite=yes dbcreate_table_opts='primary index(i)');
   retain x 0;
   method run();
   do i=1 to 10 by 1;
      x=i;
      output;
   end;
end;
enddata;
run;
quit;

Example 2
In the following example, the Hive table Hivelib.Dzoutput is created and has the partition by option appended to the DATA statement. In this example, Col1 exists in the table in the SET statement, Hivelib.Dzpt, although this is not required.

options set=SAS_HADOOP_JAR_PATH="jar-path";
options set=SAS_HADOOP_CONFIG_PATH="config-path";
libname hivelib hadoop server="hostname" user=username password=xxxx
schema=ds2ip
dbmax_text=300;

proc ds2;
data hivelib.dzoutput (dbcreate_table_opts="partitioned by (col1 int)"
overwrite=yes);
  method run();
    set hivelib.dzpt;
    x+1;
    output; output;
  end;
enddata;
run;
quit;

DBKEY= Table Option

Specifies a key column to optimize DBMS retrieval. Can improve performance when you are processing a
join that involves a large DBMS table and a small SAS data set or DBMS table.

**Default:** none

**Data source:** Aster, DB2 UNIX and PC, Greenplum, MySQL, Netezza, ODBC, Oracle,
PostgreSQL, SAP HANA, Sybase IQ, Teradata

**Syntax**

```
DBKEY=([ ' ] column [ ' ] [ . . . [ ' ] column [ ' ] ])
```

**Arguments**

*column*

used by SAS to build an internal WHERE clause to search for matches in the DBMS
table based on the key column. Here is an example:

```
select * from sas.a, dbms.b(dbkey=x) where a.x=b.x;
```

In this example, DBKEY= specifies column x, which matches the key column
designated in the WHERE clause. However, if the DBKEY= column does NOT
match the key column in the WHERE clause, then DBKEY= is not used.

**Details**

When processing a join that involves a large DBMS table and a relatively small SAS
data set, you might be able to use DBKEY= to improve performance.

When you specify DBKEY=, it is **strongly** recommended that you ensure that an index
exists for the key column in the underlying DBMS table. Performance can be severely
degraded without an index.

**CAUTION:**

Improper use of this option can decrease performance.
**DBNULL= Table Option**

Indicates whether NULL is a valid value for the specified columns when a table is created.

**Default:** DBMS-specific

**Data source:** Aster, DB2 UNIX and PC, Greenplum, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

**Syntax**

```
DBNULL= { _ALL_ = YES | NO } | ([column]=YES | NO [ ... column]=YES | NO ] )
```

**Arguments**

- `_ALL_` specifies that the YES or NO applies to all columns in the table. (This is valid in the interfaces to Informix, Oracle, Sybase, and Teradata only.)
- `YES` specifies that the NULL value is valid for the specified columns in the DBMS table.
- `NO` specifies that the NULL value is not valid for the specified columns in the DBMS table.
- `column` specifies the name of a column.

**Details**

This option is valid only for creating DBMS tables. If you specify more than one column name, the names must be separated with spaces.

The DBNULL= option processes values from left to right, so if you specify a column name twice, or if you use the _ALL_ value, the last value overrides the first value that is specified for the column.

---

**DROP= Table Option**

For an input table, excludes the specified columns from processing; for an output table, excludes the specified columns from being written to the table.

**Data source:** All

**Syntax**

```
DROP= (column-list)
```

**Arguments**

- `column-list` specifies the names of the columns to omit from the output list.
Restriction

Numbered range lists in the format `col1–col5` and name prefix lists in the format `col:` are not supported.

Details

The `DROP=` table option specifies that all columns in the `column-list` should not be included in the creation of output rows. Normally, all columns in the program data vector are included in the output rows. If the drop attribute is specified, all columns not included in the drop statement will be used to create columns in the output rows.

If the `DROP=` table option is associated with an input table, the columns are not available for processing during program execution.

Comparisons

The `DROP=` table option differs from the `DROP` statement in these ways:

- In DS2 programs, the `DROP=` table option can apply to both input and output tables. The `DROP` statement applies only to output tables.
- In DS2 programs, when you create multiple output tables, use the `DROP=` table option to write different columns to different tables. The `DROP` statement applies to all output tables.
- The `KEEP=` table option specifies a list of columns to be included in processing or to be written to the output table.

Examples

Example 1: Excluding Columns from Output Tables

In this example, the variables `SALARY` and `GENDER` are not included in processing and they are not written to either output tables.

```plaintext
data plan1 plan2;
  method run ()
  
    set payroll(drop=(salary gender));
    if hired<"01jan07"d then output plan1;
    else output plan2;
  
  end;
enddata;
```

You cannot use `SALARY` or `GENDER` in any logic in the DS2 program because `DROP=` prevents the `SET` statement from reading them from `PAYROLL`.

Example 2: Processing Columns without Writing Them to the Output Table

In this example, `SALARY` and `GENDER` are not written to `PLAN2`, but they are written to `PLAN1`.

```plaintext
data plan1 plan2(drop=(salary gender));
  method run ()
  
    set payroll;
    if hired<"01jan07"d then output plan1;
    else output plan2;
  
  end;
enddata;
```
**See Also**

**Statements:**
- “DROP Statement” on page 891

**Table Options:**
- “KEEP= Table Option” on page 987

---

### ENCRYPT= Table Option

Specifies whether to encrypt an output SAS data set.

**Restriction:** Use with output data sets only.

**Data source:** SAS data set

---

### Syntax

\[ \text{ENCRYPT}= \text{AES} | \text{NO} | \text{YES} \]

---

### Arguments

- **AES**
  - Encrypts the file using the AES (Advanced Encryption Standard) algorithm.
  
  **Requirement**
  - You must specify the ENCRYPTKEY= table option when using ENCRYPT=AES.

  **Interaction**
  - AES provides enhanced encryption by using SAS/SECURE software. SAS/SECURE is a product within the SAS System. In SAS 9.4, SAS/SECURE is included with Base SAS software. In prior releases, SAS/SECURE was an add-on product that was licensed separately. This change makes strong encryption available in all deployments (except where prohibited by import restrictions). SAS/SECURE must be installed on each computer that runs a client and a server that uses the encryption algorithms.

- **NO**
  - Does not encrypt the data set.

- **YES**
  - Encrypts the data set using the SAS Proprietary algorithm.

---

**See**

- “ENCRYPTKEY= Table Options” on page 983

---

**CAUTION**

**Record all ENCRYPTKEY= values when using ENCRYPT=AES.**

If you forget the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value. The following note will be written to the log:

Note: If you lose or forget the ENCRYPTKEY= value, there will be no way to open the file or recover the data.
**CAUTION:**

If you forget the passwords, you cannot reset them.

**Interaction**

This encryption method uses passwords that are stored in the data set. At a minimum, you must specify the READ= table option or the PW= table option at the same time that you specify ENCRYPT=YES. Because the encryption method uses passwords, you cannot change any password on an encrypted file without re-creating the table.

---

**CAUTION** Record all passwords when using ENCRYPT=YES. If you forget the passwords, you cannot reset them.

---

**Details**

When you use ENCRYPT=SAS, the following rules apply:

- You can use the ENCRYPT= option only when you are creating a SAS data file.
- In order to copy an encrypted data file, the output engine must support encryption. Otherwise, the data file is not copied.
- Encrypted files work only in SAS 6.11 or later.
- If the data file is encrypted, all associated indexes are also encrypted.
- Encryption requires approximately the same amount of CPU resources as compression.
- You cannot use PROC CPORT in SAS Proprietary encrypted SAS data files.

When you use ENCRYPT=AES, the following rules apply:

- You must use the ENCRYPTKEY= table option when creating a table.
- When you copy an encrypted AES data file, the output engine must support AES encryption. Otherwise, the data file is not copied.
- Encrypted AES data files are available beginning with SAS 9.4.
- You must have SAS/SECURE software to use AES encryption.
- You cannot change the ENCRYPTKEY= value in an AES encrypted data file without re-creating the data file.

*Note:* You can create an encrypted DS2 package or thread program by using the ENCRYPT argument in the PACKAGE and THREAD statements.

**Example**

This example creates an encrypted SAS data set:

```sas
table salary(encrypt=SAS);
```

**See Also**

**Statements:**

- “PACKAGE Statement” on page 921
- “THREAD Statement” on page 947
Table Options:

- “ENCRYPTKEY= Table Options” on page 983

---

**ENCRYPTKEY= Table Options**

Specifies a key value for AES encryption.

**Restriction:** Use only with AES encrypted data files.

**Requirement:** SAS/SECURE must be in use

**Interaction:** You cannot change the key value on an AES encrypted data set without re-creating the data set.

**Data source:** SAS data set

**Note:** Check your log after this operation to ensure that the encrypt key values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in the *SAS Language Reference: Concepts*.

---

**Syntax**

```
ENCRYPTKEY=[" | 'key-value" | ']
```

**Arguments**

`key-value`

assigns an encrypt key value. The key value can be up to 64-bytes long. You are able to create an ENCRYPTKEY= key value with or without quotation marks using the following rules:

**no quotation marks**

- alphanumeric characters and underscores only
- up to 64 bytes
- uppercase and lowercase letters
- must start with a letter
- no blank spaces
- is not case-sensitive

**Example**

```
encryptkey=key-value
encryptkey=key-value1
```

**single quotation marks**

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- blank spaces, but not all blanks
- is case-sensitive

**Example**

```
encryptkey='key-value'
encryptkey='1234*#mykey'
```
double quotation marks
• alphanumeric, special, and DBCS characters
• up to 64 bytes
• uppercase and lowercase letters
• enables macro resolution
• blank spaces, but not all blanks
• is case-sensitive

Example
encryptkey="key-value"
encryptkey="1234*#mykey"

%let mykey=abcdefghi12;
encryptkey=&key-value

Requirement
If you use ENCRYPT=AES, you must specify the ENCRYPTKEY= table option.

Note
When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```sas
data _null_
  key=length(unicodec('key-value','UTF8'));
  put 'key length=' key;
run;
```

Details

**CAUTION:** Record the key value. If you forget the ENCRYPTKEY= key value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= key value because the key value is not stored with the table. The following warning will be written to the log:

```
WARNING: If you lose or forget the ENCRYPTKEY= value, there will be no way to open the file or recover the data.
```

You must use the ENCRYPTKEY= option when you are creating or accessing a SAS data set with AES encryption.

The ENCRYPTKEY= table option will not protect the table from deletion or replacement. Encrypted tables can be deleted by using any of the following scenarios without having to specify an ENCRYPTKEY= key value:

- the KILL option in PROC DATASETS
- the DROP statement in PROC SQL
- the DELETE procedure

The ENCRYPTKEY= option only prevents access to the contents of the table. To protect the table from deletion or replacement, the file must also contain an ALTER= password.

The following DATASETS procedure statements require you to specify the ENCRYPTKEY= key value when working with protected files: AGE, AUDIT, APPEND, CHANGE, CONTENTS, MODIFY, REBUILD, and REPAIR statements.
append base=name data=name(encryptkey=key-value);
run;

The option can be specified either in parentheses after the name of the SAS data file or after a forward slash.

It is possible to use a macro variable as the ENCRYPTKEY= key value. When you specify a macro variable for the ENCRYPTKEY= key value, you must enclose the macro variable in double quotation marks. If you do not use the double quotation marks, unpredictable results can occur. The following example defines a macro variable and uses the macro variable as the ENCRYPTKEY= key value:

%let secret=myvalue;
data my.dsname(encrypt=aes encryptkey="&secret");

The following example uses the COPY statement from the DATASETS procedure and the SELECT statement:
copy in=OldLib out=NewLib;
   select salary(encryptkey=key-value);
run;

The option can be specified either in parentheses after the name of the table or after a forward slash.

CAUTION:

When using referential integrity constraints, all primary key and foreign key tables that reference each other must use the same encryption key. For more information, see the topic on encryption and integrity constraints in SAS Language Reference: Concepts.

Note: When DS2 runs outside of SAS, such as in the SAS Federation Server and in grid computing environments, the SAS macro facility is not available and DS2 programs with macros fail to compile.

Example

This example uses the ENCRYPT=AES option.

data salary (encrypt=aes encryptkey=green overwrite=yes);
   dcl char(8) name;
   dcl double yrsal;
   dcl double bonuspct;
   method init();
      name='Muriel'; yrsal=34567; bonuspct=3.2;
      name='Bjorn'; yrsal=74644; bonuspct=2.5;
      name='Freda'; yrsal=38755; bonuspct=4.1;
      name='Benny'; yrsal=29855; bonuspct=3.5;
      name='Agnetha'; yrsal=70998; bonuspct=4.1;
   end;
run;

When you run the CONTENTS procedure, you will be prompted to specify the ENCRYPTKEY= key value.

proc contents data=salary;
run;
See Also

Table Options:

- “ENCRYPT= Table Option” on page 981
- “SAS Data File Encryption” in *SAS Language Reference: Concepts*

---

**IN= Table Option**

Creates a Boolean variable that indicates whether the table contributed data to the current row.

**Restriction:** Use with the SET and DATA statements only.

**Data source:** All

---

**Syntax**

\[ \text{IN=variable} \]

**Arguments**

\[ \text{variable} \]

names the new variable whose value indicates whether that input table contributed data to the current row. Within a DS2 program, the value of the variable is 1 if the table contributed to the current row, and 0 otherwise.

**Interaction:** If the variable is not explicitly declared, it is automatically declared in the local scope of the SET or DATA statement as INTEGER.

**Data type:** BIGINT, INTEGER, SMALLINT, TINYINT

---

**Details**

Specify the IN= table option in parentheses after a table name in the SET or DATA statements. Values of IN= variables are available to program statements during the DS2 program, but the variables are not included in the table that is being created, unless they are assigned to a new variable.

When you use IN= with BY-group processing with the SET statement, and when a table contributes a row for the current BY group, the IN= value is 1. The value remains as long as that BY group is still being processed and the value is not reset by programming logic.

---

**Example**

The following example illustrates the IN table option.

```sas
data _null_
  dcl int gro;
  method run();
  dcl smallint gpo;
  dcl tinyint gpf;
  set gas_price_option (in=gpo) gas_rbid_option (in=gro)
    gas_price_forward (in=gpf);
```

KEEP= Table Option

For an input table, specifies the columns to process; for an output table, specifies the columns to write to the table.

Data source: All

Syntax

KEEP=(column-list)

Arguments

column-list

specifies the names of the columns to keep in the output table.

Restriction Numbered range lists in the format col1–col5 and name prefix lists in the format col: are not supported.

Details

The KEEP= table option specifies that all columns in the column-list should be included in the creation of output rows. Normally, all columns in the program data vector are included in the output rows. If the keep attribute is specified, all columns not included in the KEEP statement will be dropped from the output rows.

If the KEEP= table option is associated with an input table, only the columns that are specified by the KEEP= table option are available for processing during program execution.

Only global variables, by default, are included in the output. Local variables used for program loops and indexes do not need to be explicitly dropped from the output.

Comparisons

The KEEP= table option differs from the KEEP statement in these ways:

• In DS2 programs, the KEEP= table option can apply to both input and output tables. The KEEP statement applies only to output tables.
• In DS2 programs, when you create multiple output tables, use the KEEP= table option to write different columns to different tables. The KEEP statement applies to all output tables.
The DROP= table option specifies columns to omit during processing or to omit from the output table.

Example
In this example, only IDNUM and SALARY are read from PAYROLL, and they are the only variables in PAYROLL that are available for processing.

```sas
data bonus;
  method run();
    set payroll(keep=(idnum salary));
    bonus=salary*1.1;
  end;
enddata;
```

See Also

Table Options:
- “DROP= Table Option” on page 979

**LABEL= Table Option**

Specifies a label for a table.

Data source: SAS data set

Syntax

```
LABEL='label'
```

Arguments

'label'

specifies a text string of up to 256 characters. If the label text contains single quotation marks, use double quotation marks around the label, or use two single quotation marks in the label text and enclose the string in single quotation marks. To remove a label from a table, assign a label that is equal to a blank that is enclosed in quotation marks.

Details

You can use the LABEL= option on both input and output tables. When you use LABEL= on input tables, it assigns a label for the table for the duration of the DS2 program. When it is specified for an output table, the label becomes a permanent part of that table.

A label assigned to a table remains associated with that table when you update a table in place. However, a label is lost if you use a table with a previously assigned label to create a new table in the DS2 program. For example, a label previously assigned to table ONE is lost when you create the new output table ONE:

```sas
data one;
  set one;
enddata;
```
Comparisons

The LABEL= option in the HAVING clause of the DECLARE statement also enables you to assign labels to variables.

Example

These examples assign labels to tables:

```
data w2 (label = '2009 W2 Info, Hourly');
data new (label = 'Sales'' list');
data acct (label = "Hillside''s Daily Account");
data sales (label = 'May (South)');
```

---

**LOCKTABLE= Table Option**

Places shared or exclusive locks on tables.

**Data source:** SAS data set

**Syntax**

```
LOCKTABLE=SHARE | EXCLUSIVE
```

**Arguments**

**SHARE**

locks a table in shared mode, allowing other users or processes to read data from the tables, but preventing users from updating data.

**EXCLUSIVE**

locks a table exclusively, preventing other users from accessing any table that you open.

**Details**

You can lock tables only if you are the owner or have been granted the necessary privilege.

If you use PROC DS2, the default value for the LOCKTABLE option is EXCLUSIVE.

---

**OVERWRITE= Table Option**

For a table, drops the output table before the replacement output table is populated with rows; for packages and threads, drops the existing package or thread if a package or thread by the same name exists.

**Data source:** All

**Syntax**

```
OVERWRITE=YES | NO
```
Arguments

YES | NO

specifies whether the output table is deleted before a replacement output table is created or whether a package or thread is dropped.

CAUTION:
For tables, use the OVERWRITE=YES statement only with data that is backed up or with data that you can reconstruct. Because the output table is deleted first, data will be lost if a failure occurs while the output table is being written.

Default NO

Details

Details for Tables
By default, in DS2, a table is not overwritten unless the OVERWRITE= table option is set to YES. If the output table exists and the OVERWRITE= table option is set to NO, an error will occur because the existing table will not be overwritten.

Details for Packages
If you set OVERWRITE=YES in a PACKAGE statement and a DS2 thread or a regular table exists with the same name as the package being created, the table will not be dropped. Only the package is dropped.

Details for Threads
If you set OVERWRITE=YES in a THREAD statement and a DS2 package or a regular table exists with the same name as the thread being created, the table will not be dropped. Only the thread is dropped.

Examples

Example 1: Using DROP, KEEP, and OVERWRITE
The following example uses the DROP=, KEEP=, and OVERWRITE= table options for tables a and b.

data
  a(keep=x overwrite=yes)
  _rowset_(drop=(x z))
  b(keep=z overwrite=yes);
  dcl double x y z;
  method init();
    do x = 1 to 10;
      y = 2*x;
      z = 3*x;
      output;
    end;
  end;
  enddata;
run;
data;
  method run();
Example 2: Overwriting a Table

The following example creates a table, and then attempts to overwrite it without OVERWRITE=YES.

data a;
  method init();
    do x = 1 to 10;
        y = 2*x;
        z = 3*x;
        output;
    end;
  end;
enddata;
run;

/* This program fails because it is impossible to overwrite table A */
data a;
  method run();
    set a;
    x = y + z;
  end;
enddata;
run;

/* This program deletes (drops) table A before attempting to create it, */
/* so the program executes without error. If there is an error during */
/* execution, the old version of A is lost. */
table a(overwrite=yes);
  method run();
    set a;
    x = y + z;
  end;
enddata;
run;

PW= Table Option

Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected SAS file.

Data source: SAS data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in SAS Language Reference: Concepts.
Syntax

\texttt{PW=\textit{password}}

\textbf{Arguments}

\textit{password} must be a valid SAS name.

\textbf{Details}

The \texttt{PW=} option applies to all types of SAS files. You can use this option to assign a password to a SAS file or to access a password-protected SAS file.

When you replace a SAS data set that is protected by an ALTER password, the new data set inherits the ALTER password. When the code is written to the SAS log, the password is blotted out. Here is an example:

\begin{verbatim}
   drop thread job2 (pw=xxxxxxx);
\end{verbatim}

\textit{Note:} A SAS password does not control access to a SAS file beyond the SAS System. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

\textbf{See Also}

- “ENCRYPT= Table Option” on page 981
- “ALTER= Table Option” on page 972
- “WRITE= Table Option” on page 995

\section*{READ= Table Option}

Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.

\begin{tabular}{ll}
\textbf{Data source:} & SAS data set \\
\textbf{Note:} & Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in \textit{SAS Language Reference: Concepts}.
\end{tabular}

\textbf{Syntax}

\texttt{READ=\textit{read-password}}

\textbf{Arguments}

\textit{read-password} must be a valid SAS name.
Details

The READ= option applies to all types of SAS files except catalogs. You can use this option to assign a password to a SAS file or to access a Read-protected SAS file. When the code is written to the SAS log, the password is blotted out. Here is an example:

```
declare package sales (read=XXXXXXXX);
```

Note: A SAS password does not control access to a SAS file beyond the SAS System. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

See Also

Table Options:

- “ENCRYPT= Table Option” on page 981
- “PW= Table Option” on page 991
- “WRITE= Table Option” on page 995

**RENAME= Table Option**

Changes the name of a column.

**Data source:** All

**Syntax**

```
RENAME=(old-name { = | AS } new-name [...old-name { = | AS } new-name])
```

**Arguments**

- `old-name` the column that you want to rename.
- `new-name` the new name of the column. It must be a valid name for the data source.

**Details**

The RENAME= table option enables you to change the names of one or more columns.

If you use RENAME= when you create a table, the new column name is included in the output table. If you use RENAME= on an input table, the new name is used in DS2 programming statements.

If you use RENAME= in the same DS2 program with either the DROP= or the KEEP= table option, the DROP= and the KEEP= table options are applied before RENAME=.

You must use the old name in the DROP= and KEEP= table options. You cannot drop and rename the same column in the same statement.

In addition to changing the name of a column, RENAME= also changes the label for the column.
Comparisons

• The RENAME= table option differs from the RENAME statement in the following ways.
  • The RENAME statement applies to all output tables. If you want to rename different columns in different tables, you must use the RENAME= table option.
  • The RENAME= table option enables you to specify the columns that you want to rename for each input or output table. Use it in input tables to rename columns before processing.
  • If you use both the RENAME statement and RENAME= output table option, the RENAME statement has precedence. If X is renamed to Y with a RENAME statement and X is renamed to Z with a RENAME= table option, the RENAME statement takes precedence and X will be renamed to Y.
  • Use the RENAME statement or the RENAME= table option when program logic requires that you rename columns such as two input tables that have columns with the same name.

Examples

Example 1: Renaming a Column at Time of Output
This example uses RENAME= in the DATA statement to show that the column is renamed when it is written to the output table. The column keeps its original name, X, during DS2 processing.

```sas
data two(rename=(x=keys))
  method run();
  set one;
  z=x+y;
run;
enddata;
```

Example 2: Renaming a Column at Time of Input
This example renames column X to a column named KEYS in the SET statement, which is a rename before DS2 processing. KEYS, not X, is the name to use for the column for DS2 processing.

```sas
data three;
  method run();
  set one(rename=(x AS keys));
  z=keys+y;
run;
enddata;
```

See Also

Statements:
• “RENAME Statement” on page 929
**TYPE= Table Option**

Specifies the data set type for a specially structured SAS data set.

**Data source:** SAS data set

---

**Syntax**

TYPE=\textit{data-set-type}

**Arguments**

\textit{data-set-type}

specifies the special type of the data set.

---

**Details**

Use the \texttt{TYPE=} table option in a DS2 program to create a special SAS data set in the proper format, or to identify the special type of the SAS data set in a procedure statement.

You can use the CONTENTS procedure to determine the type of a data set.

Most SAS data sets do not have a specified type. However, there are several specially structured SAS data sets that are used by some SAS/STAT procedures. These SAS data sets contain special variables and observations, and they are usually created by SAS statistical procedures.

Other values are available in other SAS software products and are described in the appropriate documentation.

\textit{Note:} If you use a DS2 program with a \texttt{SET} statement to modify a special SAS data set, you must specify the \texttt{TYPE=} option in the \texttt{DATA} statement. The \textit{data-set-type} is not automatically copied to the data set that is created.

---

**See Also**

**Statements:**

- “\texttt{SET Statement}” on page 937

---

**WRITE= Table Option**

Assigns a WRITE password to a SAS file that prevents users from writing to a file, unless they enter the password.

**Data source:** SAS data set

**Note:** Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in \textit{SAS Language Reference: Concepts}.
Syntax

`WRITE=write-password`

Arguments

`write-password`

must be a valid SAS name.

Details

The `WRITE=` option applies to all types of SAS files except catalogs. You can use this option to assign a password to a SAS file or to access a Write-protected SAS file. When the code is written to the SAS log, the password is blotted out. Here is an example:

```r
drop thread job2a (write=XXXXXXX);
```

*Note:* A SAS password does not control access to a SAS file beyond the SAS System. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

See Also

Table Options:

- “`ENCRYPT= Table Option`” on page 981
- “`PW= Table Option`” on page 991
- “`READ= Table Option`” on page 992
Chapter 30
DS2 FCMP Package Methods, Operators, and Statements

Dictionary

DECLARE PACKAGE Statement, FCMP Package
Creates a package variable and gives you the option to create an instance of the FCMP package.

**Category:** Local

**Requirement:** The PACKAGE statement is required before you use the DECLARE PACKAGE statement.

---

**Syntax**

```
DECLARE PACKAGE fcmp-package-name variable ( );
```

**Arguments**

- `fcmp-package-name`
  - specifies the name of the FCMP package.
  - **Requirement**
    - The package name must match the name of a package created in a PACKAGE statement, or an error will occur.
  - **See**
    - “PACKAGE Statement” on page 921

- `variable`
  - specifies a name that can reference an instance of the package.

**Details**

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.
You use an FCMP package to support calls to functions and subroutines that are available or are created with the FCMP procedure. The FCMP package is predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

You create an FCMP package by using the PACKAGE statement then declare the FCMP package by using the DECLARE PACKAGE statement. This associates an FCMP package with an FCMP name. After you declare the new FCMP package, you can call the functions and subroutines that are created with the FCMP procedure.

There are two ways to construct an instance of an FCMP package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:

```
declare package fcmp pharma;
pharma = _new_ fcmp();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package fcmp pharma();
```

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

For more information about FCMP packages, see “Using the FCMP Package” on page 144.

Examples

**Example 1: Using FCMP OUTARGS Parameters**

The following example creates FCMP subroutine named `package1` that uses OUTARGS parameters. The FCMP procedure’s OUTARGS parameter is treated as an IN_OUT parameter in the METHOD statement.

```
libname base '.';
proc fcmp outlib = base.fcmpsubs.package1;
  subroutine swapper(a,b);
    outargs a,b;
    t1 = b; b = a; a = t1;
  endsub;
run;
quit;

proc ds2;
  package pkg / overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

data _null_
  dcl package pkg p();
method init();
  dcl double x y;
  x=10;
  y=42;
```
put 'before:' x= y=;
p.swapper(x,y);
put 'after:' x= y=;
end;
enddata;
run;
quit;

The following lines are written to the SAS log.

before: x=10 y=42
after: x=42 y=10

**Example 2: FCMP Package Using DOUBLE Arguments**

This example walks through creation of a square routine in FCMP and using that routine from a DS2 program. The current directory is used as the "library" of FCMP packages.

libname base '.';

* fcmp defines a function, square;
proc fcmp outlib = base.fcmpsubs.package1;
  function square(a);
    return (a*a);
  endsub;
run;
quit;

* define the ds2 package thru which the fcmp functions will be called;
proc ds2;
  package pkg /overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

* demonstration of calling fcmp thru the ds2 wrapper package;
data _null_;    
dcl package pkg p();    
dcl double a b;    
method init();    
do a = 10 to 20;    
  b=p.square(a);    
  put a= b=;
end;
end;
enddata;
run;
quit;

The following lines are written to the SAS log.
Example 3: FCMP Package with Character Arguments

libname base '.';

proc fcmp outlib = base.fcmpsubs.package1;
  function f(a $) $ 10;
    return (trim(a) !! trim(a));
  endsub;
run;

proc ds2;
  package pkg /overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

data _null_;   dcl package pkg p();
  method runone(double arg, double expected);
    dcl double actual;
    actual=p.f(arg);
    if (actual ~= expected) then put 'ERROR:' arg= expected= actual=;
  end;
  method init();
    runone(5, 55);
    runone(345, 345345);
    runone(10, 1010);
    runone(4.2, .); * can't convert back to double w/ two '.' chars;
    runone(. , .);
  end;
enddata;
run;
quit;

See Also

- “Using the FCMP Package” on page 144
- “Package Constructors and Destructors” on page 142

Operators:

- “_NEW_ Operator, FCMP Package” on page 1001

Statements:

- “PACKAGE Statement” on page 921
DELETE Method, FCMP Package

Deletes an FCMP package.

**Note:** The DELETE method is not required. When an FCMP package goes out of scope, the package is deleted.

### Syntax

```
package.Delete( );
```

### Arguments

- **package**
  
  specifies the name of the FCMP package variable.

### Details

When you no longer need the FCMP package, delete it by using the DELETE method. If you attempt to use an FCMP package after you delete it, an error will be written to the log.

### See Also

“Package Constructors and Destructors” on page 142

.NEW_ Operator, FCMP Package

Constructs an instance of an FCMP package.

**Note:** The escape character ( \ ) before the bracket indicates that the bracket is required in the syntax.

### Syntax

```
package-variable =.NEW_[THIS] | [package-instance]]; fcmp-package-name( );
```

### Arguments

- **package-variable**
  
  specifies a name that can reference an instance of the package.

- **[THIS]**
  
  specifies that the package instance has global scope.

- **[package-instance]**
  
  specifies that the new package instance has the same scope as package-instance. package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

See “Packages and Scope” on page 137
See “Package-Specific Scope” on page 139

fcmp-package-name

specifies the name of the package.

Requirement fcmp-package-name must be a predefined FCMP package created with a PACKAGE statement.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You create an FCMP package by using the PACKAGE statement then declare and instantiate the FCMP package by using the DECLARE PACKAGE statement. This associates an FCMP package with an FCMP package variable name. After you declare the new FCMP package, you can call the functions and subroutines that are created with the FCMP procedure.

There are two ways to construct an instance of an FCMP package.

• Use the DECLARE PACKAGE statement along with the _NEW_ operator:

```plaintext
declare package pharma pkg;
pkg = _new_ pharma();
```

• Use the DECLARE PACKAGE statement along with its constructor syntax:

```plaintext
declare package pharma pkg();
```

*Note:* Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

• “Using the FCMP Package” on page 144
• “Package Constructors and Destructors” on page 142

Statements:

• “DECLARE PACKAGE Statement, FCMP Package” on page 997
• “PACKAGE Statement” on page 921
Chapter 31

DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements
ADD Method, Hash Package

Adds key values, data values, or both to the hash package.

**Apply to:** Hash package

### Syntax

Form 1:  
```plaintext
package.ADD();
```

Form 2:  
```plaintext
package.ADD([keys], [data]);
```

Form 3:  
```plaintext
package.ADD([keys]);
```

### Arguments

- **package**  
  Specifies an instance of the hash package variable.

- **[keys]**  
  Specifies the key values by using a variable list.

  **Restriction**  
  If you specify only keys, the ADD method only works for key-only hash packages.

  **See**  
  “Variable Lists” on page 59

- **[data]**  
  Specifies the variables into which to add the hash data.

  **See**  
  “Variable Lists” on page 59

### Details

You can store key and data values in the hash package using the ADD method.

There are two ways to pass keys and data values to the ADD method:

- **implicit variable method** (Form 1)
  The key and data variables are implied in the ADD method invocation and do not have to be specified.

- **variable list method** (Forms 2 and 3)
  The specified key and data variables are passed explicitly to the ADD method. If the hash package contains only keys, use Form 3.

**Note:**

- If you add a key that is already in the hash package, then the ADD method returns a nonzero value to indicate that the key is already in the hash package. Use the REPLACE method to replace the data that is associated with the specified key with new data. However, if you set the DUPLICATE constructor
parameter or method to ADD when you create the hash package, the ADD method returns a zero.

- If you do not specify the data variables with the DEFINEDATA method, the data variables are automatically assumed to be same as the keys.
- The ADD method does not set the value of the data variable to the value of the data item. It only sets the value in the hash package.

**Examples**

**Example 1: Using the Implicit Variable Method**
The following example uses the implicit variable method to define the key and data item.

```plaintext
data _null_;  
declare char(20) d;  
declare char(20) k;  
declare double rc;  
declare package hash h(4);  
method init();  
 (rc = h.defineKey ('k'));  
 rc = h.defineData ('d');  
 rc = h.defineDone ();  
/* Define constant value for key and data */  
k='Homer';  
d='Odyssey';  
/* Use the ADD method to add the key and data to the hash package */  
rc = h.add();  
/* Define constant value for key and data */  
k='Joyce';  
d='Ulysses';  
/* Use the ADD method to add the key and data to the hash package */  
rc = h.add();  
end;  
enddata;  
run;
```

**Example 2: Adding Key and Data Values Using the Variable List Method**
The following example uses the implicit variable method to define the key and data item.

```plaintext
data _null_;  
declare char(20) d;  
declare char(20) k;  
declare double rc;  
method init();  
  declare package hash h([k], [d]);  
  /* Define constant value for key and data */  
k='Homer';  
d='Odyssey';  
/* Use the ADD method to add the key and data to the hash package */  
rc = h.add([k], [d]);  
/* Define constant value for key and data */  
k='Joyce';  
d='Ulysses';
```
Example 3: Using the ADD and FIND Methods

The following example uses the ADD method to store the data in the hash package and associate the data with the key. The FIND method is then used to retrieve the data that is associated with the key value 'Homer'.

data _null_;    
declare char(20) d;    
declare char(20) k;    
declare double rc;    
method init();    
declare package hash h(\[k\], \[d\], 4);    
/* Define constant value for key and data */    
k='Homer';    
put k=;    
d='Odyssey';    
put d=;    
/* Use the ADD method to add the key and data to the hash package */    
rc = h.add([k], [d]);    
/* Define constant value for key and data */    
k='Joyce';    
d='Ulysses';    
/* Use the ADD method to add the key and data to the hash package */    
rc = h.add([k], [d]);    
k='Homer';    
/* Use the FIND method to retrieve the data associated with 'Homer' key */    
if (h.find([k], [d]) = 0) then    
put d=;    
else    
put 'Key Homer not found.';    
end;    
enddata;    
run;

The FIND method assigns the data value 'Odyssey', which is associated with the key value 'Homer', to the variable D.

The following lines are written to the SAS log:

```
k=Homer
d=Odyssey
d=Odyssey
```

See Also

- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “DEFINEDATA Method” on page 1021
- “DEFINEKEY Method” on page 1024
CHECK Method

Checks whether the specified key is stored in the hash package.

**Applies to:** Hash package

---

**Syntax**

Form 1: `package.CHECK();`

Form 2: `package.CHECK([keys]);`

**Arguments**

- `package` specifies an instance of the hash package variable.
- `[keys]` specifies the key values by using a variable list.

See “Variable Lists” on page 59

---

**Details**

You use the CHECK method to determine whether a key exists in the hash table but the data variable is not updated. The CHECK method returns a zero value if the key is found in the hash table and a nonzero value if the key is not found.

There are two ways to pass keys and data variables to the CHECK method:

- implicit variable method (Form 1)
  - The key variables are implied in the CHECK method invocation and do not have to be specified.

- variable list method (Form 2)
  - The specified key variables are passed explicitly to the CHECK method.

---

**Comparisons**

The CHECK method only returns a value that indicates whether the key is in the hash package. The data variable that is associated with the key is not updated. The FIND method also returns a value that indicates whether the key is in the hash package. However, if the key is in the hash package, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

---

**Example**

In the following example, the CHECK method is used to determine whether the data is associated with the key value 'Homer'.

```plaintext
data _null_;  
declare char(20) d;  
declare char(20) k;
```

---

• “REF Method” on page 1056
declare double rc;
method init();
   declare package hash h([k], [d]);
   /* Define constant value for key and data */
   k='Homer';
   put k=;
   d='Odyssey';
   put d=;
   /* Use the ADD method to add the key and data to the hash package */
   rc = h.add([k], [d]);
   /* Define constant value for key and data */
   k='Joyce';
   d='Ulysses';
   /* Use the ADD method to add the key and data to the hash package */
   rc = h.add([k], [d]);
   k='Homer';
   /* Use the CHECK method to verify the data associated with 'Homer' key */
   if (h.check([k]) = 0) then
      put 'Key Homer is found.';
   else
      put 'Key Homer not found.';
   end;
enddata;
run;

The following lines are written to the SAS log.

```
k=Homer
d=Odyssey
Key Homer is found
```

See Also

- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “DEFINEKEY Method” on page 1024
- “FIND Method” on page 1027
- “KEYS Method” on page 1041

### CLEAR Method

Removes all items from a hash package without deleting the hash package instance.

**Applies to**: Hash package

**Syntax**

```
package.CLEAR();
```
**Arguments**

*package*

specifies an instance of the hash package variable.

**Details**

The CLEAR method removes the items from within the hash package but leaves the hash package instance so that it can be reused. To remove a hash package completely, use the DELETE method.

To clear all items from the hash package MYHASH, use the following code:

```java
rc = myhash.clear();
```

**See Also**

Methods:

- “DELETE Method, Hash, and Hash Iterator Package” on page 1025

---

### DATA Method

Specifies the data variables to be stored in the hash package by using a variable list.

**Applies to:** Hash package

#### Syntax

```java
package.DATA([data]);
```

#### Arguments

*package*

specifies an instance of the hash package variable.

*data*

specifies the name of the data variables by using a variable list.

**See** “Variable Lists” on page 59

#### Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You use a variable list to specify the data variables for a hash package using the DATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

*Note:* Alternatively, you could use the DEFINEDATA method or constructors in the DECLARE PACKAGE statement to specify the data variables.

Keys and data consist of any number of character or numeric variables.
Example
This example creates a hash package that contains two data variables and one key variable. The output is sorted in descending order.

```sas
data _null_;
  dcl double x;
  dcl double rc;
  dcl date d;
  /* The output will be in descending order. */
  dcl package hash h(8, '', 'descending', '', '');
  dcl package hiter hi('h');

  method init();
      rc = h.keys([d]);
      rc = h.data([d]);
      rc = h.data([x]);
      rc = h.definedone();
      d = date '1929-08-24'; x = 1; h.add();
      d = date '1930-09-25'; x = 2; h.add();
      d = date '1930-10-26'; x = 3; h.add();
      d = date '1930-10-27'; x = 4; h.add();
      d = date '1933-12-28'; x = 5; h.add();
      d = date '1999-12-01'; x = 999;
      do while (hi.next() = 0);
          put d= x=;
          d = date '1999-12-01'; x = 999;
      end;
  enddata;
run;
```

The following lines are written to the SAS log.

```
d=1933-12-28 x=5  
d=1930-10-27 x=4  
d=1930-10-26 x=3  
d=1930-09-25 x=2  
d=1929-08-24 x=1  
```

See Also
- “Defining Key and Data Variables” on page 147
- “Implicit Variable and Variable List Methods” on page 149

Methods:
- “DEFINEDATA Method” on page 1021
- “DEFINEDONE Method” on page 1023
- “KEYS Method” on page 1041

Statements:
- “DECLARE PACKAGE Statement, Hash Package” on page 1012
**DATASET Method**

Specifies the name of a table to load into the hash package.

**Applies to:** Hash package

---

**Syntax**

```sql
package.DATASET(['data-source'] | '{sql-text'});
```

**Arguments**

- `package` specifies an instance of the hash package variable.
- `data-source` specifies the name of a table.

**Tip** The name of the table can be a string literal or a character variable. If a literal is used, the table name must be enclosed in single quotation marks.

```
{sql-text}
```

is any valid FedSQL code that resolves to a set of table rows.

**Requirement** The FedSQL query must be enclosed in braces ( `{ } `).

**Note** The FedSQL query is specified in the following form: `{SELECT <select-list> FROM <table-specification>;}`. For more information, see the SELECT statement in the *SAS FedSQL Language Reference*.

---

**Details**

You can specify the table to load into the hash package by using the DATASET method.

*Note:* Alternatively, you can use the `datasource` parameter in the DECLARE PACKAGE statement or the `_NEW_` operator to specify the table.

**See Also**

- “Storing and Retrieving Data” on page 151
- “Providing Initialization Data for a Hash Package” on page 148

**Operators:**

- “_NEW_ Operator, Hash Package” on page 1044

**Statements:**

- “DECLARE PACKAGE Statement, Hash Package” on page 1012
DECLARE PACKAGE Statement, Hash Package

Creates a package variable and gives you the option of creating an instance of the hash package.

Category: Local
Tip: The PACKAGE statement is not required for a hash package.

Syntax

Form 1: DECLARE PACKAGE HASH variable ( );
Form 2: DECLARE PACKAGE HASH variable (hashexp, {datasource | \sql-text}, 'ordered', 'duplicate', 'suminc', 'multidata');
Form 3: DECLARE PACKAGE HASH variable ([keys], [data], [hashexp, {datasource | \sql-text}, 'ordered', 'duplicate', 'suminc', 'multidata']);
Form 4: DECLARE PACKAGE HASH variable ([keys], [hashexp, {datasource | \sql-text}, 'ordered', 'duplicate', 'suminc', 'multidata']);

Arguments

`variable`

specifies a variable that can reference an instance of the hash package.

`[keys]`

specifies the key values by using a variable list.

See “Variable Lists” on page 59

`[data]`

specifies the data variables by using a variable list and associates them with the specified keys.

See “Variable Lists” on page 59

`hashexp`

is the hash package's internal table size, where the size of the hash table is \(2^n\).

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of \(2^4\), or 16. The maximum value for hashexp is 16, which equates to a hash table size of \(2^{16}\).

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different hashexp values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.
Range 0–20

Requirement A value for hashexp must be entered. If a value less than 0 is entered, then a default value of 8, which equates to a hash table size of $2^8$ or 256, is used. If a value greater than 20 is entered, then a default value of 20 is used.

Data type INTEGER

'datasource' is the name of a table to load into the hash package.

The name of the table can be a literal or a character variable. The table name must be enclosed in single quotation marks.

Requirement Either a placeholder of an empty string in quotation marks (""") or a value for datasource must be entered. If the place holder is entered, then the hash is not loaded from a data source.

{sql-text} is any valid FedSQL SELECT statement that resolves to a set of table rows.

Requirement The FedSQL query must be enclosed in braces ({}).

Note The FedSQL query is specified in the following form: {SELECT <select-list> FROM <table-specification>;}. For more information, see the SELECT statement in the SAS FedSQL Language Reference.

'ordered' specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method.

ordered can be one of the following values:

'ASCENDING' | 'A'
Data is returned in ascending key-value order. Specifying 'ASCENDING' is the same as specifying 'YES'.

'DESCENDING' | 'D'
Data is returned in descending key-value order.

'YES'
Data is returned in ascending key-value order. Specifying 'YES' is the same as specifying 'ASCENDING'.

'NO'
Data is returned in an undefined order.

Requirement Either a placeholder of an empty string in quotation marks ("") or a value for ordered must be entered. If the place holder is entered, then a default ordering of 'NO' is used.

'duplicate' determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates.

duplicate can be one of the following values:
'REPLACE'
stores the last duplicate key record.

'ERROR'
reports an error to the log if a duplicate key is found.

'ADD'
stores the first key record found and not any of the duplicates.

Requirement: Either a placeholder of an empty string in quotation marks ("" or a value for `duplicate` must be entered. If the placeholder is entered, then a default of 'ADD' is used.

See: “Example 3: Using the Duplicate Parameter with a Hash Package” on page 1017

'suminc'
specifies a variable that maintains a summary count of hash package keys.

Requirement: Either a placeholder of an empty string in quotation marks ("" or a value for `suminc` must be entered.

Data type: INTEGER

Note: This variable holds the sum increment—that is, how much to add to the key summary for each reference to the key. The `suminc` value treats a missing or null value as zero, like the SUM function. For example, a key summary changes using the current value of the variable.

'multidata'
specifies whether multiple data items are allowed for each key.

`multidata` can be one of the following values:

'YES' | 'Y' | 'MULTIDATA'
allows multiple data items for each key

'NO' | 'N' | 'SINGLEDATA'
allows only one data items for each key

Requirement: Either a placeholder of an empty string in quotation marks ("" or a value for `multidata` must be entered. If the placeholder is entered, then a default of 'NO' is used.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

A particular hash package instance is defined by a set of key variables, a set of data variables, and optional initialization data. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls. If key variables and data variables are specified when the hash package instance is constructed (Forms 3 and 4), then the hash instance is constructed as fully defined. If key variables and data variables are not provided when the hash package instance is constructed (Form 2), then additional definition of the hash instance can be specified with a subsequent series of method calls followed by a single DEFINEDONE method call. The DEFINEDONE method indicates that specification of key variables, data variables, and other initialization data is complete. A hash package instance is not
constructed if no variables or initialization data is provided (Form 1). In this instance, the hash instance is constructed with the _NEW_ operator or additional method calls followed by a single DEFINEDONE method call.

In the following example, hash \texttt{h1} and hash \texttt{h2} have equivalent definition. Hash \texttt{h1} is fully defined when the hash is constructed because key and data variables are provided as constructor arguments. Hash \texttt{h2} is only partially defined when the hash is constructed, and the hash definition is completed through a series of method calls followed by a single DEFINEDONE method call.

\begin{verbatim}
declare package hash h1([key1], [data1 data2 data3], 0, 'testdata', '', '', '', 'multidata');
declare package hash h2();

h2.keys([key1]);
h2.data([data1 data2 data3]);
h2.dataset('testdata');
h2.multidata();
h2.defineDone();
\end{verbatim}

A DECLARE PACKAGE statement can create a hash package variable that is a null package reference. The hash package variable can then be set to reference a hash package instance constructed by a subsequent call of the _NEW_ operator.

\begin{verbatim}
declare package hash hashgnp;
hashgnp = _new_ hash(10, 'testpkg', 'yes');
\end{verbatim}

(Optional) A DECLARE PACKAGE statement can both create the hash package variable and construct the hash package instance.

\begin{verbatim}
declare package hash hashgnp(10, 'testpkg', 'yes');
\end{verbatim}

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

For more information about the hash package, see “Using the Hash Package” on page 145.

Examples

**Example 1: Storing and Retrieving Data with a Hash Package**
The following example declares a hash package named H that will store several key/value pairs and uses an iterator to output the keys in sorted order.

\begin{verbatim}
data _null_;  
dcl double x;  
dcl date d;
/* The output will be in descending order. */  
dcl package hash h(8, '', 'descending', '', '');  
dcl package hiter hi('h');  
method init();  
rc = h.defineKey('d');  
r...
The following lines are written to the SAS log:

d=1933-12-28 x=5
d=1930-10-27 x=4
d=1930-10-26 x=3
d=1930-09-25 x=2
d=1929-08-24 x=1

Example 2: Loading a Table into a Hash Package
Assume that the table SMALL contains two numeric variables K (key) and S (data) and another table, LARGE, contains a corresponding key variable K. The following code loads the SMALL table into the hash package, and then searches the hash package for key matches on the variable K from the LARGE table.

```sas
/* create small table */
data small(overwrite=yes);
dcl char(8) k s;
method init();
dcl integer i;
do i = 1 to 10;
k = put(i, BEST8.);
s = put(2*i, BEST8.);
output;
end;
end;
enddata;
run;
/* create large table */
data large(overwrite=yes);
dcl char(8) k;
method init();
dcl integer i;
do i = -20 to 20;
k = put(i, BEST8.);
output;
end;
end;
enddata;
run;
/*load SMALL table into the hash package */
data myhash(overwrite=yes);
declare char(8) k;
declare char(8) s;
declare package hash h(8,'small');
```
/* define SMALL table variable K as key and S as value */

method init();
    rc = h.defineKey('k');
    rc = h.defineData('s');
    rc = h.defineDone();
end;

/* use the SET statement to iterate over the LARGE table using */
/* keys in the LARGE table to match keys in the hash package */

method run();
    set large;
    if (h.find() = 0) then output;
end;
enddata;
run;

Example 3: Using the Duplicate Parameter with a Hash Package
The following is an example of using the ADD, REPLACE, and ERROR options with
the DUPLICATE parameter.

data dups(overwrite=yes);
    dcl double x y;
    method init();
        do x = 1 to 5;
            y = 2*x;
            output;
        end;
        x = 3; y = 99; output;
        x = 4; y = 100; output;
    end;
enddata;
run;

data _null_;
    dcl double x y;
    dcl int rc1 rc2 rc3 rc4;
    dcl package hash h(8, 'dups', 'yes');
    dcl package hiter hi;
    method init();
        rc = h.defineKey('x');
        rc = h.defineData('x');
        rc = h.defineData('y');
        rc = h.defineDone();
        hi = _new_ hiter('h');
        do while (hi.next() = 0);
            put x= y=;
        end;
        put;
    end;
enddata;
run;
data _null_;  
dcl double x y;  
dcl int rc1 rc2 rc3 rc4;  
dcl package hash h(8, 'dups', 'yes', 'add');  
dcl package hiter hi;  
method init();  
  rc = h.defineKey('x');  
  rc = h.defineData('x');  
  rc = h.defineData('y');  
  rc = h.defineDone();  
hi = _new_ hiter('h');  
  do while (hi.next() = 0);  
    put x= y=;  
  end;  
  put;  
end;  
enddata;  
run;

data _null_;  
dcl double x y;  
dcl int rc1 rc2 rc3 rc4;  
dcl package hash h(8, 'dups', 'yes', 'replace');  
dcl package hiter hi;  
method init();  
  rc = h.defineKey('x');  
  rc = h.defineData('x');  
  rc = h.defineData('y');  
  rc = h.defineDone();  
hi = _new_ hiter('h');  
  do while (hi.next() = 0);  
    put x= y=;  
  end;  
  put;  
end;  
enddata;  
run;

data _null_;  
dcl double x y;  
dcl int rc1 rc2 rc3 rc4;  
dcl package hash h(8, 'dups', 'yes', 'error');  
dcl package hiter hi;  
method init();  
  rc = h.defineKey('x');  
  rc = h.defineData('x');  
  rc = h.defineData('y');  
  rc = h.defineDone();  
hi = _new_ hiter('h');  
  do while (hi.next() = 0);  
    put x= y=;  
  end;  
  put;
end;
enddata;
run;

The following lines are written to the SAS log when using the ADD option:

```sas
x=1 y=2
x=2 y=4
x=3 y=6
x=4 y=8
x=5 y=10

NOTE: Execution succeeded. No rows affected.
```

The following lines are written to the SAS log when using the REPLACE option:

```sas
x=1 y=2
x=2 y=4
x=3 y=99
x=4 y=100
x=5 y=10

NOTE: Execution succeeded. No rows affected.
```

When using the ERROR option, an error message is written to the SAS log:

```sas
x=1 y=2
x=2 y=4
x=3 y=6
x=4 y=8
x=5 y=10

NOTE: Execution succeeded. No rows affected.
ERROR: Duplicate key found when loading hash package from data source "dups".
ERROR: Hash data source load failed.
```

**Example 4: Using the ORDERED Parameter with a Hash Package**

The following example sets an ascending order for the hash package H and a descending order for the hash package H2.

```sas
data _null_;  
dcl double x y;  
dcl package hash h(8, '', 'ascending');  
dcl package hash h2(8, '', 'descending');  
dcl package hiter hi('h');  
dcl package hiter hi2('h2');  
method init();  
  rc = h.defineKey('x');  
  rc = h.defineKey('y');  
  rc = h.defineDone();  
  rc = h2.defineKey('y');  
  rc = h2.defineKey('x');  
  rc = h2.defineDone();  

do x = 1 to 10;  
  y = 2 * x;  
  rc = h.add();
```

---

**DECLARE PACKAGE Statement, Hash Package** 1019
rc = h2.add();
end;

hi.first(); put y=
do while (hi.next() = 0);
    put y=
end;
put;

hi2.first(); put x=
do while (hi2.next() = 0);
    put x=
end;
end;
enddata;
runc;

See Also
• “Using the Hash Package” on page 145
• “Providing Initialization Data for a Hash Package” on page 148
• “Package Constructors and Destructors” on page 142

Operators:
• “_NEW_ Operator, Hash Package” on page 1044

Statements:
• “DECLARE PACKAGE Statement, Hash Iterator Package” on page 1020

DECLARE PACKAGE Statement, Hash Iterator Package
Creates a hash iterator package variable and gives you the option of creating an instance of the hash
iterator package.

Category: Local
Tip: The PACKAGE statement is not required for a hash iterator package.

Syntax
DECLARE PACKAGE HITER variable [("hash-name" | hash-package-instance)];

Arguments
variable
    specifies a name that can reference an instance of the hash iterator package variable.

"hash-name"
    specifies the name of the hash package with which the hash iterator is associated.

hash-package-instance
    specifies the instance of the hash package with which the hash iterator is associated.
Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use the hash and hash iterator packages to quickly and efficiently store, search, and retrieve data based on unique lookup keys. The hash and hash iterator packages are predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

You declare a hash iterator package by using the DECLARE PACKAGE statement. This associates a hash iterator package with a hash and hash iterator name.

Note: You must declare and instantiate a hash package before you create a hash iterator package.

There are two ways to construct an instance of a hash iterator package.

• Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  
  declare package hiter myhiter;
  myhiter = _new_ hiter('h');

• Use the DECLARE PACKAGE statement along with its constructor syntax:
  
  declare package hiter myiter('h');

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

• “Using the Hash Iterator Package” on page 157
• “Using the Hash Package” on page 145
• “Package Constructors and Destructors” on page 142

Operators:

• “_NEW_ Operator, Hash Iterator Package” on page 1048

Statements:

• “DECLARE PACKAGE Statement, Hash Package” on page 1012

---

**DEFINEDATA Method**

Defines data variables for the hash package using implicit variables.

**Applies to:** Hash package
Syntax

package.DEFINEDATA('data');

Arguments

package
  specifies an instance of the hash package variable.
'data'
  specifies the name of the data variable.

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You call the DEFINEDATA method for each data variable that you create. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you could use the DATA variable list method or constructors in the DECLARE PACKAGE statement to specify the data variables.

Keys and data consist of any number of character or numeric variables.

Example

This example creates a hash package that contains two data variables and one key variable. The output is sorted in descending order.

data _null_;  
dcl double x;  
dcl double rc;  
dcl date d;  
/* The output will be in descending order. */  
dcl package hash h(8, '', 'descending', '', '');  
dcl package hiter hi('h');  
method init();  
  rc = h.definekey('d');  
  rc = h.definedata('d');  
  rc = h.definedata('x');  
  rc = h.defineDone();  
  d = date '1929-08-24'; x = 1; h.add();  
  d = date '1930-09-25'; x = 2; h.add();  
  d = date '1930-10-26'; x = 3; h.add();  
  d = date '1930-10-27'; x = 4; h.add();  
  d = date '1933-12-28'; x = 5; h.add();  
  d = date '1999-12-01'; x = 999;  
  do while (hi.next() = 0);  
    put d= x=;  
  d = date '1999-12-01'; x = 999;  
  end;  
end;  
enddata;  
run;
The following lines are written to the SAS log.

```
d=1933-12-28 x=5
d=1930-10-27 x=4
d=1930-10-26 x=3
```

**See Also**

- “Defining Key and Data Variables” on page 147
- “Implicit Variable and Variable List Methods” on page 149

**Methods:**

- “DATA Method” on page 1009
- “DEFINEDONE Method” on page 1023

**Statements:**

- “DECLARE PACKAGE Statement, Hash Package” on page 1012

---

**DEFINEDONE Method**

Indicates that all key and data definitions are complete.

**Applies to:** Hash package

**Syntax**

```
package.Definedone();
```

**Arguments**

`package`

specifies an instance of the hash package variable.

**Details**

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You can define the key and data variables in one of three ways.

- Use the implicit variable methods DEFINEDATA and DEFINEKEY.
- Use the variable list methods DATA and KEYS.
- Use key and data variable lists as constructors in the DECLARE PACKAGE statement.

If the hash package is not completely defined using constructors in the DECLARE PACKAGE statement, you must call the DEFINEDONE method to complete initialization of the hash package.
Example

The following example creates a hash package, defines the key and data variables, and completes the initialization of the hash package:

```
/* definedone with definedata method */
declare hash h(h);
rc = h.defineKey('k');
rc = h.defineData('d');
rc = h.defineDone();

/* same package using definedone with data method */
declare hash h;
rc=h.keys([k]);
rc=h.data([d]);
rc=definedone();
```

See Also

- “Defining Key and Data Variables” on page 147
- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “DEFINEDATA Method” on page 1021
- “DEFINEKEY Method” on page 1024

Statements:

- “DECLARE PACKAGE Statement, Hash Package” on page 1012

---

**DEFINEKEY Method**

Defines key variables for the hash package using implicit variables.

**Applies to:** Hash package

**Syntax**

```
package.DEFINEKEY('key');
```

**Arguments**

- **package**
  - specifies an instance of the hash package variable.
- **'key'**
  - specifies the name of the key variable.

**Details**

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.
You call the DEFINEKEY method for each key variable that you create. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you could use the KEYS variable list method or constructors in the DECLARE PACKAGE statement to specify the key variables.

Keys and data consist of any number of character or numeric variables.

Example

The following example creates a hash package and defines the key and data variables:

```
dec1are hash h();
rc = h.definekey('k');
rc = h.definedata('d');
rc = h.definedone();
end;
```

See Also

- “Defining Key and Data Variables” on page 147
- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “DEFINEDONE Method” on page 1023
- “KEYS Method” on page 1041

Statements:

- “DECLARE PACKAGE Statement, Hash Package” on page 1012

---

**DELETE Method, Hash, and Hash Iterator Package**

Deletes a hash or hash iterator package.

**Applies to:** Hash and hash iterator packages

**Note:** The DELETE method is not required. When a hash or hash iterator package goes out of scope, the package is deleted.

**Syntax**

```
package.DELETE();
```

**Arguments**

- `package`
  
  specifies an instance of the hash or hash iterator package variable.
Details

When you no longer need the hash or hash iterator package, delete it by using the DELETE method. If you attempt to use a hash or hash iterator package after you delete it, an error will be written to the log.

If you want to delete all the items from within a hash package and save the hash package to use again, use the CLEAR method.

See Also

“Package Constructors and Destructors” on page 142

DUPLICATE Method

Determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates.

Applies to: Hash package

Syntax

\[ \text{package.DUPLICATE('option');} \]

Arguments

package

specifies an instance of the hash package variable.

'option'

option can be one of the following values:

'REPLACE'

stores the last duplicate key record.

'ERROR'

reports an error to the log if a duplicate key is found.

'ADD'

stores the first key record found and not any of the duplicates.

Default ADD

Details

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE method. The DUPLICATE method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.
Note: Alternatively, you can use the duplicate parameter as a constructor in the DECLARE PACKAGE statement or the _NEW_ operator to specify duplicate keys.

See Also

- “Non-Unique Key and Data Pairs” on page 150
- “Providing Initialization Data for a Hash Package” on page 148

Methods:

- “MULTIDATA Method” on page 1043

Operators:

- “_NEW_ Operator, Hash Package” on page 1044

Statements:

- “DECLARE PACKAGE Statement, Hash Package” on page 1012

FIND Method

Determines whether the specified key is stored in the hash package.

 Applies to: Hash package

Syntax

Form 1:  

Form 2:  

Form 3:  

Arguments

 package

 specifies an instance of the hash package variable.

 [keys]

 specifies the key values by using a variable list.

 Restriction

 If you specify only keys, the FIND method only works for key-only hash packages.

 See

 “Variable Lists” on page 59

[data]

 specifies the variables into which to copy the hash data.

 See  “Variable Lists” on page 59

Details

You use the key variable values to determine whether a key exists in the hash table. If
the key exists, the data values are copied into the data variables. The FIND method
returns a zero value if the key is found in the hash table and a nonzero value if the key is not found.

There are two ways to pass key and data variables to the FIND method:

• implicit variable method (Form 1)
  The key and data variables are implied in the FIND method invocation and do not have to be specified.

• variable list method (Forms 2 and 3)
  The specified key and data variables are passed explicitly to the FIND method. If the hash package contains only keys, use Form 3.

**Comparisons**

The FIND method returns a value that indicates whether the key is in the hash package. If the key is in the hash package, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call. The CHECK method only returns a value that indicates whether the key is in the hash package. The data variable is not updated.

**Example**

See “Example 3: Using the ADD and FIND Methods” on page 1006.

**See Also**

• “Implicit Variable and Variable List Methods” on page 149
• “Non-Unique Key and Data Pairs” on page 150
• “Storing and Retrieving Data” on page 151

**Methods:**

• “CHECK Method” on page 1007
• “KEYS Method” on page 1041
• “DEFINEKEY Method” on page 1024
• “FIND_NEXT Method” on page 1028
• “FIND_PREV Method” on page 1031

**FIND_NEXT Method**

Sets the current list item to the next item in the current key's multiple item list and sets the data for the corresponding data variables.

**Applies to:** Hash package

**Syntax**

Form 1: `package.FIND_NEXT( );`
Form 2: `package.FIND_NEXT([data]);`
Arguments

package
specifies an instance of the hash package variable.

[data]
specifies the variables into which to copy the data associated with the current key.

See “Variable Lists” on page 59

Details

The FIND method determines whether the key exists in the hash package.

The HAS_NEXT method determines whether the key has multiple data items associated with it. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_NEXT and FIND_NEXT methods to traverse the list.

There are two ways to pass data variables to the FIND_NEXT method:

- implicit variable method (Form 1)
  The data variables are implied in the FIND_NEXT method invocation and do not have to be specified.
- variable list method (Form 2)
  The specified data variables are passed explicitly to the FIND_NEXT method.

Example

This example uses the FIND_NEXT method to iterate through a table where several keys have multiple data items.

data testcases;
dcl double k;
dcl double expected;
method init();
k=0; expected=14; output; /* magic number */

k=1; expected=1; output;
k=2; expected=2; output;
k=3; expected=1; output;
k=4; expected=3; output;
k=5; expected=2; output;
k=6; expected=1; output;
k=7; expected=1; output;
k=8; expected=1; output;
k=9; expected=1; output;
end;
enddata;
run;

data inp;
dcl double k v;
method init();
do k = 1 to 10; v = k * k; output; end;
data _null_

dcl double k v;
dcl package hash h(8, 'inp', 'ascending', '', '', 'multidata');
method init();
    h.defineKey('k');
    h.defineData('k');
    h.defineData('v');
    h.defineDone();
end;
method run();
dcl double actual;
/*******************************
set testcases;

rc = h.find();
if (rc ~= 0) then
    actual = h.get_num_items();
else do;
    put k= rc=;
    actual = 0;
    do while (rc = 0);
        actual+1;
        rc = h.find_next();
    put k= rc=;
end;
end;
enddata;
run;

The following lines are written to the SAS log.
See Also

- “Implicit Variable and Variable List Methods” on page 149
- “Non-Unique Key and Data Pairs” on page 150
- “Storing and Retrieving Data” on page 151

Methods:

- “FIND Method” on page 1027
- “FIND_PREV Method” on page 1031
- “HAS_NEXT Method” on page 1037

---

**FIND_PREV Method**

Sets the current list item to the previous item in the current key's multiple item list and sets the data for the corresponding data variables.

**Applies to:** Hash package

**Syntax**

Form 1: `package.FIND_PREV();`

Form 2: `package.FIND_PREV([data]);`

**Arguments**

`package`

specifies an instance of the hash package variable.

`[data]`

specifies the variables into which to copy the data associated with the current key.
Details

The FIND method determines whether the key exists in the hash package.

The HAS_PREV method determines whether the key has multiple data items associated with it. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

There are two ways to pass data variables to the FIND_PREV method:

- implicit variable method (Form 1)
  The data variables are implied in the FIND_PREV method invocation and do not have to be specified.
- variable list method (Form 2)
  The specified data variables are passed explicitly to the FIND_PREV method.

Example

See “Example: Retrieving a Summary Value” on page 1070.

See Also

- “Implicit Variable and Variable List Methods” on page 149
- “Non-Unique Key and Data Pairs” on page 150
- “Storing and Retrieving Data” on page 151

Methods:

- “FIND Method” on page 1027
- “FIND_NEXT Method” on page 1028
- “HAS_PREV Method” on page 1039

FIRST Method

Returns the first value in the underlying hash package.

Applies to: Hash iterator package

Syntax

Form 1: `package.FIRST( );`
Form 2: `package.FIRST([data]);`
Arguments

package
specifies an instance of the hash iterator package variable.

[data]
specifies the variables into which to copy the data associated with the first hash item.

See “Variable Lists” on page 59

Details

The FIRST method returns the first data item in the hash package. If you specified YES or ASCENDING in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character). This occurs because the data items are sorted in ascending key-value order in the hash package. Repeated calls to the NEXT method will iteratively traverse the hash package and return the data items in ascending key order.

Conversely, if you specified DESCENDING in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character). This occurs because the data items are sorted in descending key-value order in the hash package. Repeated calls to the NEXT method will iteratively traverse the hash package and return the data items in descending key order.

Use the LAST method to return the last data item in the hash package.

Note: The FIRST method sets the data variable to the value of the data item so that it is available for use after the method call.

There are two ways to pass data variables to the FIRST method:

- implicit variable method (Form 1)
  The data variables are implied in the FIRST method invocation and do not have to be specified.

- variable list method (Form 2)
  The specified data variables are passed explicitly to the FIRST method.

Example

The following example uses the FIRST, NEXT, PREV, and LAST methods when starting a new iteration at a different location within the hash package:

```plaintext
data _null_;  
dcl double x y rc;
  dcl package hash h([x], [y]);
  dcl package hiter hi('h');
  method init();
    do x = 1 to 10;
      y = 2*x;
      rc = h.add([x], [y]);
    end;
    do while (hi.next([y]) = 0);
      put y=;
    end;
  end;
  put;
```
The following lines are written to the SAS log.
See Also

Methods:
- “LAST Method” on page 1042
- “ORDERED Method” on page 1052

Operators:
- “_NEW_ Operator, Hash Package” on page 1044

Statements:
- “DECLARE PACKAGE Statement, Hash Package” on page 1012
HASHEXP Method

Defines the hash package’s internal table size. The size of the hash table is $2^n$.

**Applies to:** Hash package

**Syntax**

```plaintext
package.HASHEXP(exponent);
```

**Arguments**

- `package` specifies an instance of the hash package variable.
- `exponent` specifies the power-of-2 for the internal table size.

**Default**

8

**Data type** INTEGER

**Details**

The value specified for the HASHEXP method is used as a power-of-two exponent to create the hash table size. For example, a value of 4 equates to a hash table size of $2^4$, or 16. The maximum value for `exponent` is 16, which equates to a hash table size of $2^{16}$.

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different `exponent` values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.

*Note:* Alternatively, you can use the `hashexp` parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify the hash table size.

**See Also**

- “Providing Initialization Data for a Hash Package” on page 148

**Operators:**

- “_NEW_ Operator, Hash Package” on page 1044

**Statements:**

- “DECLARE PACKAGE Statement, Hash Package” on page 1012
HAS_NEXT Method

Determine whether there is a next item in the current key's multiple data item list.

Applies to: Hash package

Syntax

package.HAS_NEXT();

Arguments

package

specifies an instance of the hash package variable.

Details

If a key has multiple data items, you can use the HAS_NEXT method to determine whether there is a next item in the current key's multiple data item list. If there is another item, the method will return a nonzero value in the numeric variable \( R \). Otherwise, it will return a zero.

The FIND method determines whether the key exists in the hash package. The HAS_NEXT method determines whether the key has multiple data items associated with it. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example: Finding Data Items

This example creates a hash package where several keys have multiple data items. It uses the HAS_NEXT to find all the data items.

data testcases;
   dcl double k;
   dcl double expected;
   method init();
      k=0; expected=14; output; /* magic number */
      k=1; expected=1; output;
      k=2; expected=2; output;
      k=3; expected=1; output;
      k=4; expected=3; output;
      k=5; expected=2; output;
      k=6; expected=1; output;
      k=7; expected=1; output;
      k=8; expected=1; output;
      k=9; expected=1; output;
end;
enddata;
run;
The following lines are written to the SAS log.
HAS_PREV Method

Determines whether there is a previous item in the current key's multiple data item list.

**Applies to:** Hash package

**Syntax**

```package.HAS_PREV();```

**Arguments**

`package`

specifies an instance of the hash package variable.

**Details**

If a key has multiple data items, you can use the HAS_PREV method to determine whether there is a previous item in the current key's multiple data item list. If there is a previous item, the method will return a nonzero value in the numeric variable R. Otherwise, it will return a zero.

The FIND method determines whether the key exists in the hash package. The HAS_NEXT method determines whether the key has multiple data items associated with
it. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example
See “Example: Retrieving a Summary Value” on page 1070.

See Also
- “Non-Unique Key and Data Pairs” on page 150

Methods:
- “FIND Method” on page 1027
- “FIND_PREV Method” on page 1031
- “HAS_NEXT Method” on page 1037

ITEM_SIZE Attribute
Returns the size (in bytes) for an item in a hash package.

Applies to: Hash package

Syntax
```
variable-name=package.ITEM_SIZE;
```

Arguments
- `variable-name`
  - specifies the name of the variable that contains the size of the item in the hash package after the method is complete.
- `package`
  - specifies an instance of the hash package variable.

Details
The ITEM_SIZE attribute returns the size (in bytes) of an item, as well as the key and data variables and some internal information. You can set an estimate of how much memory the hash package is using with the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash package requires, nor does it take into account any necessary internal alignments. Therefore, the use of ITEM_SIZE does not provide exact memory usage, but it does return a good approximation.

Example
For an example, see the “NUM_ITEMS Attribute” on page 1050.
**KEYS Method**

Defines the key variables for the hash package using a variable list.

**Applies to:** Hash package

**Syntax**

```plaintext
package.KEYS([keys]);
```

**Arguments**

- `package` specifies an instance of the hash package variable.
- `[keys]` specifies the names of the key variables using a variable list.

See “Variable Lists” on page 59

**Details**

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You can use a variable list to specify the key variables for a hash package using the KEYS method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

*Note:* Alternatively, you can use the DEFINEKEYS method or constructors in the DECLARE PACKAGE statement to specify key variables.

*Note:* You can have a hash package that contains only key variables and no data variables. This is a keys-only hash package.

Keys and data consist of any number of character or numeric variables.

**Example**

The following example creates a hash package and defines the key and data variables:

```plaintext
declare hash h();
r = h.keys([k]);
r = h.data([d]);
r = h.definedata();
end;
```
LAST Method

Returns the last value in the underlying hash package.

**Applies to:** Hash iterator package

**Syntax**

Form 1:  
```
package.LAST();
```

Form 2:  
```
package.LAST([data]);
```

**Arguments**

**package**

specifies an instance of the hash iterator package variable.

**[data]**

specifies the variables into which to copy the data associated with the first hash item.

See  
“Variable Lists” on page 59

**Details**

The LAST method returns the last data item in the hash package. If you specified YES or ASCENDING in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in ascending key-value order in the hash package.

Conversely, if you specified DESCENDING in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in descending key-value order in the hash package.

Use the FIRST method to return the first data item in the hash package.

There are two ways to pass data variables to the LAST method:

• implicit variable method (Form 1)
The data variables are implied in the LAST method invocation and do not have to be specified.

- variable list method (Form 2)
  The specified data variables are passed explicitly to the LAST method.

**Example**

For an example, see the “FIRST Method” on page 1032.

**See Also**

- “Variable Lists” on page 59

**Methods:**

- “FIRST Method” on page 1032
- “ORDERED Method” on page 1052

**Operators:**

- “_NEW_ Operator, Hash Package” on page 1044

**Statements:**

- “DECLARE PACKAGE Statement, Hash Package” on page 1012

---

**MULTIDATA Method**

Specifies whether multiple data items are allowed for each key.

**Applies to:** Hash package

**Syntax**

```
package.MULTIDATA(['option']);
```

**Arguments**

- `package`
  specifies an instance of the hash package variable.

- `'option'`
  `option` can be one of the following values:

  - `'Y'` | `'YES'` | `'MULTIDATA'`
    allows multiple data items for each key.

  - `'N'` | `'NO'` | `'SINGLEDATA'`
    reports an error to the log if a duplicate key is found.

**Default**

`NO`
Details

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE method. The DUPLICATE method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

Note: Alternatively, you can use the multidata parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify whether multiple data items are allowed for each key.

See Also

• “Non-Unique Key and Data Pairs” on page 150
• “Providing Initialization Data for a Hash Package” on page 148

Methods:

• “DUPLICATE Method” on page 1026

Operators:

• “_NEW_ Operator, Hash Package” on page 1044

Statements:

• “DECLARE PACKAGE Statement, Hash Package” on page 1012

.NEW_ Operator, Hash Package

Constructs an instance of a hash package.

Note: The escape character ( \ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1:  
`package-variable=_NEW_[THIS|package-instance]] HASH (hashexp,'datasource','ordered','duplicate','suminc','multidata');`

Form 2:  
`package-variable=_NEW_[THIS|package-instance]] HASH ([keys],[data],[hashexp,'datasource','ordered','duplicate','suminc','multidata']);`

Form 3:  
`package-variable=_NEW_[THIS|package-instance]] HASH ([keys],[hashexp,'datasource'|sql-text],'ordered','duplicate','suminc','multidata');`
Arguments

*package-variable*

specifies a name that can reference an instance of the package.

**[THIS]**

specifies that the package instance has global scope.

See “Packages and Scope” on page 137

**[package-instance]**

specifies that the new package instance has the same scope as *package-instance*. *package-instance* must be an existing package instance, and the type of *package-instance* can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

**[keys]**

specifies the key values by using a variable list.

See “Variable Lists” on page 59

**[data]**

specifies the data variables by using a variable list and associates them with the specified keys.

See “Variable Lists” on page 59

*hashexp*

is the hash package's internal table size, where the size of the hash table is $2^n$.

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of $2^4$, or 16. The maximum value for hashexp is 16, which equates to a hash table size of $2^{16}$.

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different hashexp values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.

Requirement Either a placeholder of –1 or a value for *hashexp* must be entered. If the place holder is entered, then a default value of 8, which equates to a hash table size of $2^8$ or 256, is used.

Data type INTEGER

*datasource*

is the name of a table to load into the hash package.

The name of the table can be a literal or a character variable. The table name must be enclosed in single quotation marks.
Requirement Either a placeholder of an empty string in quotation marks ("\") or a value for `datasource` must be entered. If the placeholder is entered, then the hash is not loaded from a data source.

### `{sql-text}`

is any valid FedSQL SELECT statement that resolves to a set of table rows.

Requirement The FedSQL query must be enclosed in braces ( `{ } `).

Note The FedSQL query is specified in the following form: `{SELECT <select-list> FROM <table-specification>;}`. For more information, see the SELECT statement in the *SAS FedSQL Language Reference*.

### `'ordered'`

specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method. Here are the valid values:

- `'ASCENDING'` | `'A'`
  Data is returned in ascending key-value order. Specifying `'ASCENDING'` is the same as specifying `'YES'`.

- `'DESCENDING'` | `'D'`
  Data is returned in descending key-value order.

- `'YES'`
  Data is returned in ascending key-value order. Specifying `'YES'` is the same as specifying `'ASCENDING'`.

- `'NO'`
  Data is returned in an undefined order.

Requirement Either a placeholder of an empty string in quotation marks ("\") or a value for `ordered` must be entered. If the placeholder is entered, then a default ordering of `'NO'` is used.

### `'duplicate'`

determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates. Here are the valid values:

- `'REPLACE'`
  stores the last duplicate key record.

- `'ERROR'`
  reports an error to the log if a duplicate key is found.

- `'ADD'`
  stores the first key record found and not any of the duplicates.

Requirement Either a placeholder of an empty string in quotation marks ("\") or a value for `duplicate` must be entered. If the placeholder is entered, then a default ordering of `'ADD'` is used.

See “Example 3: Using the Duplicate Parameter with a Hash Package” on page 1017

### `'suminc'`

specifies a variable that maintains a summary count of hash package keys.
Requirement: Either a placeholder of an empty string in quotation marks ("") or a value for `suminc` must be entered.

Note: This variable holds the sum increment—that is, how much to add to the key summary for each reference to the key. The `suminc` value treats a missing or null value as zero, like the SUM function. For example, a key summary changes using the current value of the variable.

`multidata` specifies whether multiple data items are allowed for each key. Here are the valid values:

- `'YES' | 'Y' | 'MULTIDATA'` allows multiple data items for each key
- `'NO' | 'N' | 'SINGLEDATA'` allows only one data item for each key

Requirement: Either a placeholder of an empty string in quotation marks ("") or a value for ordered must be entered. If the placeholder is entered, then a default ordering of `"NO"` is used.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a hash package is declared, the variable representing the package can be considered an instance of the package. This means that two different package variables represent two completely separate copies of a package.

You declare a hash package using the DECLARE PACKAGE statement. After you declare the new hash package, you can use the `_NEW_` operator to instantiate the package.

```
declare package hash myhash();
myhash = _new_ hash();
```

A particular hash package instance is defined by a set of key variables, a set of data variables, and optional initialization data. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls. If key variables and data variables are specified when the hash package instance is constructed (Forms 2 and 3), then the hash instance is constructed as fully defined. If key variables and data variables are not provided when the hash package instance is constructed (Form 1), then additional definition of the hash instance can be specified with a subsequent series of method calls followed by a single DEFINEDONE method call. The DEFINEDONE method indicates that specification of key variables, data variables, and other initialization data is complete.

For example, you can provide initialization data by using parameters in the constructor syntax for the hash package

```
declare package hash h();
h = _new_ hash(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

Note: You can use the DECLARE PACKAGE statement constructor to declare and instantiate a hash or hash iterator package in one step. For more information, see “Defining a Hash Instance By Using Constructor” on page 146 and the “DECLARE PACKAGE Statement, Hash Package” on page 1012.
Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

• “Using the Hash Package” on page 145
• “Using the Hash Iterator Package” on page 157
• “Package Constructors and Destructors” on page 142

Operators:

• “_NEW_ Operator, Hash Iterator Package” on page 1048

Statements:

• “DECLARE PACKAGE Statement, Hash Package” on page 1012

.NEW_ Operator, Hash Iterator Package

Creates an instance of a hash iterator package.

Note: The escape character ( \ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

\~\texttt{package-variable} = \~\texttt{NEW} \left[ \texttt{\{THIS\}} | \texttt{\{package-instance\}} \right] \texttt{HITER} ('\texttt{hash-name}');

Arguments

\texttt{package-variable}

specifies a name that can reference an instance of the package.

[THIS]

specifies that the package instance has global scope.

See “Packages and Scope” on page 137

[package-instance]

specifies that the new package instance has the same scope as package-instance. package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

'hash-name'

specifies the hash package that is associated with the hash iterator package.

Requirement You must declare and instantiate a hash package before you create a hash iterator package.
Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a hash iterator package is declared, the variable representing the package can be considered an instance of the package. This means that two different package variables represent two completely separate copies of a package.

You declare a hash iterator package using the DECLARE PACKAGE statement. After you declare the new hash iterator package, use the _NEW_ operator to instantiate the package.

```plaintext
declare package hiter myiter;
myiter = _new_ hiter('myhash');
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a hash iterator package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```plaintext
declare package hiter myiter('myhash');
```

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

- “Using the Hash Iterator Package” on page 157
- “Using the Hash Package” on page 145
- “Package Constructors and Destructors” on page 142

Operators:

- “_NEW_ Operator, Hash Package” on page 1044

Statements:

- “DECLARE PACKAGE Statement, Hash Iterator Package” on page 1020

NEXT Method

Returns the next value in the underlying hash package.

Applies to: Hash iterator package

Syntax

Form 1: `package.NEXT();`
Form 2: `package.NEXT([data]);`

Arguments

`package`

specifies an instance of the hash iterator package variable.
[data]
specifies the variables into which to copy the data associated with the next hash item.

See “Variable Lists” on page 59

Details

Use the NEXT method iteratively to traverse the hash package and return the data items in key order. The FIRST method returns the first data item in the hash package. You can use the PREV method to return the previous data item in the hash package.

There are two ways to pass data variables to the NEXT method:

• implicit variable method (Form 1)
  The data variables are implied in the NEXT method invocation and do not have to be specified.

• variable list method (Form 2)
  The specified data variables are passed explicitly to the NEXT method.

Example

For an example, see the “FIRST Method” on page 1032.

See Also

Methods:

• “FIRST Method” on page 1032
• “PREV Method” on page 1055

NUM_ITEMS Attribute

Returns the number of items in the hash package.

Applies to: Hash package

Syntax

\[
\text{variable-name}=\text{package.NUM_ITEMS};
\]

Arguments

\text{variable-name}
  specifies the name of a variable that contains the number of items in the hash package after the method is complete.

\text{package}
  specifies an instance of the hash package variable.

Details

The NUM_ITEMS attribute returns the number of key/data pairs stored in the hash table.
Example

The following example uses the NUM_ITEMS attribute to count the number of items within the hash package and the ITEM_SIZE attribute to report the size of an item in the hash package.

data _null_
   dcl int item_size num_items;
   dcl double x;
   dcl timestamp t;
   dcl package hash h(8, '', 'yes');
   dcl package hiter hi('h');
   method init();
      rc = h.defineKey('t');
      rc = h.defineData('t');
      rc = h.defineData('x');
      rc = h.defineDone();
   item_size = h.item_size;
   num_items = h.num_items;
   put item_size=;
   put num_items=;
   put;
   t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();
   t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();
   t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();
   t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();
   t = timestamp '1929-09-25 12:51:36.00'; x = 1; h.add();
   t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();
   t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();
   t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();
   t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();
   t = timestamp '1930-09-25 13:52:37.01'; x = 1; h.add();
   num_items = h.num_items;
   put num_items=;
   do while (hi.next() = 0);
      put t= x=;
   end;
   put;
   t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.remove();
   t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.remove();
   num_items = h.num_items;
   put num_items=;
   do while (hi.next() = 0);
      put t= x=;
   end;
   enddata;
run;
The following lines are written to the SAS log:

```
item_size=72
num_items=0

num_items=11
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=1
t=1929-09-25 13:52:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1

num_items=9
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
```

### See Also

Attributes:

- “ITEM_SIZE Attribute” on page 1040

### ORDERED Method

Specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method.

Applies to: Hash package

#### Syntax

```
package.ORDERED(‘option’);
```

#### Arguments

- **package**
  - specifies an instance of the hash package variable.

- **option**
  - can be one of the following values:
    - `ASCENDING` | ‘A’
      - Data is returned in ascending key-value order. Specifying `ASCENDING` is the same as specifying `YES`. 
'DESCENDING' | 'D'
Data is returned in descending key-value order.

'YES'
Data is returned in ascending key-value order. Specifying 'YES' is the same as specifying 'ASCENDING'.

'NO'
Data is returned in an undefined order.

Default  NO

Details

If you specify YES or ASCENDING in the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character). This occurs because the data items are sorted in ascending key-value order in the hash package. Repeated calls to the NEXT method will iteratively traverse the hash package and return the data items in ascending key order.

Conversely, if you specify DESCENDING parameter in the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character). This occurs because the data items are sorted in descending key-value order in the hash package. Repeated calls to the NEXT method will iteratively traverse the hash package and return the data items in descending key order.

Use the FIRST method returns the first data item in the hash package. Use the LAST method to return the last data item in the hash package.

Note: Alternatively, you can use the ordered parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify whether the data is returned in key-value order.

See Also

• “Implicit Variable and Variable List Methods” on page 149

Methods:

• “FIRST Method” on page 1032
• “LAST Method” on page 1042

Operators:

• “_NEW_ Operator, Hash Package” on page 1044

Statements:

• “DECLARE PACKAGE Statement, Hash Package” on page 1012

OUTPUT Method

Creates a table that contains the data in the hash package.
Applies to: Hash package

Syntax

```plaintext
package.OUTPUT (['output-table']);
```

Arguments

- `package` specifies an instance of the hash iterator package variable.
- `['output-table']` specifies the name of the output table.

Tip: The name of the table can be a literal or a character variable. If a literal is used, the table name must be enclosed in single quotation marks.

Details

Hash package keys are not automatically stored as part of the output table. The keys must be defined as data items by using the DEFINEDATA method, the DATA method, or the DECLARE PACKAGE statement to be included in the output table.

Example

```plaintext
data a(overwrite=yes);
  dcl double x;
  method init();
    do x = 1 to 10;
      output;
    end;
  end;
enddata;
run;
data _null_; 
  method init(); 
    dcl package hash h(4, 'a');
    rc = h.defineData('x');
    rc = h.defineKey('x');
    rc = h.defineDone();
    x = 11;
    h.add();
    x = 12;
    h.add();
    x = 13;
    h.add();
    x = 14;
    h.add();
    rc = h.output('out');
  end;
enddata;
run;
```
PREV Method

Returns the previous value in the underlying hash package.

**Applies to:** Hash iterator package

**Syntax**

Form 1: `package.PREV();`

Form 2: `package.PREV(data);`

**Arguments**

- `package` specifies an instance of the hash iterator package variable.

- `[data]` specifies the variables into which to copy the data associated with the previous hash item.

  **See** “Variable Lists” on page 59

**Details**

Use the PREV method iteratively to traverse the hash package and return the data items in reverse key order. The FIRST method returns the first data item in the hash package. The LAST method returns the last data item in the hash package. You can use the NEXT method to return the next data item in the hash package.

There are two ways to pass data variables to the PREV method:

- implicit variable method (Form 1)

  The data variables are implied in the PREV method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the PREV method.

**Example**

For an example, see the “FIRST Method” on page 1032.
See Also

Methods:

• “FIRST Method” on page 1032
• “LAST Method” on page 1042
• “NEXT Method” on page 1049

REF Method

Consolidates a FIND and ADD methods into a single method call.

Applies to: Hash package

Syntax

Form 1: `package.REF();`
Form 2: `package.REF([keys], [data]);`
Form 3: `package.REF([keys]);`

Arguments

`package`

specifies an instance of the hash package variable.

`[keys]`

specifies the key variables by using a variable list.

Restriction

If you specify only keys, the FIND method only works for key-only hash packages.

See

“Variable Lists” on page 59

`[data]`

specifies the variables into which to add the hash data.

See

“Variable Lists” on page 59

Details

You can consolidate FIND and ADD methods into a single REF method.

There are two ways to pass key and data variables to the REF method:

• implicit variable method (Form 1)

  The key and data variables are implied in the REF method invocation and do not have to be specified.

• variable list method (Forms 2 and 3)

  The specified key and data variables are passed explicitly to the REF method. If the hash package contains only keys, use Form 3.
Example

data _null_;  
dcl double x y;  
dcl int rc;  
dcl package hash h([x], [x y]);  
dcl package hiter hi('h');  
method init();  
  x = 7; y = 13;  
  rc = h.add([x], [x y]);  
  put x= rc=;  
  do x = 5 to 10;  
    y = 2*x;  
    rc = h.ref([x], [x y]);  
    put x= rc=;  
  end;  
  do while (hi.next([x y]) = 0);  
    put x= y=;  
  end;  
end;  
enddata;  
run;

The following lines are written to the SAS log.

| x=7 | rc=0 |
| x=5 | rc=0 |
| x=6 | rc=0 |
| x=7 | rc=0 |
| x=8 | rc=0 |
| x=9 | rc=0 |
| x=10 | rc=0 |
| x=9 | y=18 |
| x=5 | y=10 |
| x=7 | y=13 |
| x=10 | y=20 |
| x=6 | y=12 |
| x=8 | y=16 |

See Also

- “Implicit Variable and Variable List Methods” on page 149
- “Storing and Retrieving Data” on page 151

Methods:

- “ADD Method, Hash Package” on page 1004
- “CHECK Method” on page 1007
- “FIND Method” on page 1027

REMOVE Method

Removes the data that is associated with the specified key from the hash package.

Applies to: Hash package
Syntax

Form 1:  
```plaintext
package.REMOVE( );
```

Form 2:  
```plaintext
package.REMOVE([keys]);
```

Arguments

package  
specifies an instance of the hash package variable.

[keys]  
specifies the key values by using a variable list.

See “Variable Lists” on page 59

Details

The REMOVE method uses the values in the key variables to find and remove an existing key in a hash table.

You specify the key and then use the REMOVE method to remove the key and data in a hash package.

There are two ways to pass key variables to the REMOVE method:

- implicit variable method (Form 1)
  
  The key variables are implied in the REMOVE method invocation and do not have to be specified.

- variable list method (Form 2)
  
  The specified key variables are passed explicitly to the REMOVE method.

Note: The REMOVE method does not modify the value of data variables. It only removes the value in the hash package.

Note: If you specify YES the DECLARE PACKAGE statement, the _NEW_ operator, or the MULTIDATA method when you instantiate the hash package, the REMOVE method will remove all data items for the specified key.

Example

```plaintext
data _null_
   dcl double x rc;
   dcl timestamp t;
   dcl package hash h(0, '', 'yes');
   dcl package hiter hi('h');
   method init();
      rc = h.Keys([t]);
      rc = h.Data([t x]);
      rc = h.defineDone();
      t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();
```
\begin{verbatim}
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();
t = timestamp '1930-09-25 13:52:37.01'; x = 1; h.add();
t = timestamp '1930-10-25 13:52:37.01'; x = 1; h.add();
t = timestamp '1999-12-01 12:00:00.00'; x = 999;
do while (hi.next([t x]) = 0);
    put t= x=;
    t = timestamp '1999-12-01 12:00:00.00'; x = 999;
end;
put '**************************************';
\end{verbatim}

The following lines are written to the SAS log.

\begin{verbatim}
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=1
t=1929-09-25 13:52:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
**************************************
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
**************************************
\end{verbatim}

See Also

- “Implicit Variable and Variable List Methods” on page 149
- “Replacing and Removing Data” on page 152

Methods:

- “ADD Method, Hash Package” on page 1004
- “DEFINEKEY Method” on page 1024
- “KEYS Method” on page 1041
**REMOVEALL Method**

Removes the data that is associated with all keys from the hash package.

**Applies to:** Hash package

**Syntax**

Form 1:  
```plaintext
package.REMOVEALL();
```

Form 2:  
```plaintext
package.REMOVEALL([keys]);
```

**Arguments**

- `package`: specifies an instance of the hash package variable.
- `[keys]`: specifies the key values by using a variable list.

**See**  
“Variable Lists” on page 59

**Details**

The REMOVEALL method deletes both the keys and the data from the hash package.

There are two ways to pass key variables to the REMOVEALL method:

- **implicit variable method (Form 1)**
  
  The key variables are implied in the REMOVEALL method invocation and do not have to be specified.

- **variable list method (Form 2)**
  
  The specified key variables are passed explicitly to the REMOVEALL method.

**Note:** The REMOVEALL method does not modify the value of data variables. It only removes the value in the hash package.

**See Also**

- “Implicit Variable and Variable List Methods” on page 149
- “Replacing and Removing Data” on page 152

**Methods:**

- “MULTIDATA Method” on page 1043
- “REMOVE Method” on page 1057
- “REMOVEDUP Method” on page 1061

**Operators:**

- “_NEW_ Operator, Hash Package” on page 1044

**Statements:**
REMOVEDUP Method

Removes the data that is associated with the current key's current data item from the hash package.

**Applies to:** Hash package

**Syntax**

```plaintext
package.REMOVEDUP();
```

**Arguments**

- `package`
  - Specifies an instance of the hash package variable.

**Details**

The REMOVEDUP method deletes the current data item from the hash package for keys that have multiple data items.

*Note:* The REMOVEDUP method does not modify the value of data variables. It only removes the value in the hash package.

*Note:* If only one data item is in the key's data item list, the key and data will be removed from the hash package.

**Comparisons**

The REMOVEDUP method removes the data that is associated with the current key's current data item from the hash package. The REMOVE method removes the data that is associated with the specified key from the hash package.

**Example**

This example creates a hash package where several keys have multiple data items. Duplicate data items in the key are removed.

```plaintext
data testdup;
  length key data 8;
  input key data;
datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
```
proc ds2;
data _null_;  
dcl double key "data" k d;  
method init();  
  dcl package hash h([key], [key "data"], 8, 'testdup', 'yes', '', '', 'yes');  
  dcl package hiter i(h);  
  dcl int rc;  
  do k = 1 to 5;  
    do while (h.find([k], [k d]) = 0 and h.has_next() = 0);  
      h.find_next([k d]);  
      h.removedup();  
    end;  
  end;  
  rc = i.first([k d]);  
  do while (rc = 0);  
    put k= d=;  
    rc = i.next([k d]);  
  end;  
enddata;  
run;  
quit;  
The following lines are written to the SAS log.  

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

See Also  
• “Replacing and Removing Data” on page 152

Methods:  
• “REMOVE Method” on page 1057  
• “REMOVEALL Method” on page 1060

REPLACE Method  
Replaces the data that is associated with the specified key with new data.  

| Applies to: | Hash package |

Syntax  
Form 1:  
\texttt{package.REPLACE();}  
Form 2:  
\texttt{package.REPLACE([keys], [data]);}
Form 3:  \texttt{package.REPLACE([keys]);}

\textbf{Arguments}

\textit{package}

specifies an instance of the hash package variable.

\textit{[keys]}

specifies the key values by using a variable list.

\textbf{Restriction}

If you specify only keys, the REPLACE method only works for key-only hash packages.

See \textit{"Variable Lists" on page 59}

\textit{[data]}

specifies the variables for which the data is replaced.

See \textit{"Variable Lists" on page 59}

\textbf{Details}

The REPLACE method uses the values in the key variables to find a key/data pair in the hash table. If a pair is found, the data is replaced with the current value in the data variables (Forms 1 and 2).

For hash packages that have only keys (Form 3), the only effect that the REPLACE method has is that the summary statistics for the keys is updated.

\textbf{Example}

data _null_;  
dcl double x rc;  
dcl timestamp t;  
dcl package hash h(0, '', 'yes');  
dcl package hiter hi('h');  
method init();  
  rc = h.defineKey('t');  
  rc = h.defineData('t');  
  rc = h.defineData('x');  
  rc = h.defineDone();  
t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();  
t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();  
t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();  
t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();  
t = timestamp '1929-09-25 12:51:36.00'; x = 1; h.add();  
t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();  
  do while (hi.next() = 0);  
    put t= x=;  
    t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();  
  end;
The following lines are written to the SAS log.

```
t=1927-08-24 12:51:36 x=1
 t=1928-08-24 12:51:36 x=1
 t=1929-08-24 12:51:36 x=1
 t=1929-09-24 12:51:36 x=1
 t=1929-09-25 12:51:36 x=1
 t=1929-09-25 13:51:36 x=1
 t=1929-09-25 13:52:36 x=1
 t=1929-09-25 13:52:37 x=1
 t=1929-09-25 13:52:37.010000000 x=1
 t=1929-09-25 13:52:37.010000000 x=1
 t=1930-09-25 13:52:37.010000000 x=1
 t=1930-10-25 13:52:37.010000000 x=1
**********************************************************
t=1927-08-24 12:51:36 x=1
 t=1928-08-24 12:51:36 x=1
 t=1929-08-24 12:51:36 x=1
 t=1929-09-24 12:51:36 x=1
 t=1929-09-25 12:51:36 x=1
 t=1929-09-25 13:51:36 x=1
 t=1929-09-25 13:52:36 x=1
 t=1929-09-25 13:52:37 x=1
 t=1929-09-25 13:52:37.010000000 x=1
 t=1930-09-25 13:52:37.010000000 x=1
 t=1930-10-25 13:52:37.010000000 x=1
```

See Also

- “Implicit Variable and Variable List Methods” on page 149
- “Replacing and Removing Data” on page 152

Methods:

- “REPLACEDUP Method” on page 1064

REPLACEDUP Method

Replaces the data that is associated with the current key's current data item with new data.

**Applies to:** Hash package

**Syntax**

```
package.REPLACEDUP();
```
Arguments

*package*

specifies an instance of the hash package variable.

Details

The REPLACEDUP method replaces the current data item from the hash package for keys that have multiple data items.

*Note:* The REPLACEDUP method does not replace the value of the data variable with the value of the data item. It only replaces the value in the hash package.

*Note:* If you call the REPLACEDUP method and the key is not found, then the key and data are added to the hash package.

Comparisons

The REPLACEDUP method replaces the data that is associated with the current key's current data item with new data. The REPLACE method replaces the data that is associated with the specified key with new data.

Example

This example creates a hash package where several keys have multiple data items. When a duplicate data item is found, 300 is added to the value of the data item.

data testdup;
  length key data 8;
  input key data;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
proc ds2;
data _null_;  
dcl double key "data" k d;
  method init();
  dcl package hash h([key], [key "data"], 8, 'testdup','yes', '', '', 'yes');
  dcl package hiter i(h);
  dcl int rc;
  do k = 1 to 5;
    do while (h.find([k], [k d]) = 0);
      put k= d=;
      do while (h.has_next() = 0);
        h.find_next([k d]);
        h.find_next([k d]);
      put 'dup ' k= d=;
      d = d + 300;
The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>key=1 data=10</th>
<th>dup key=1 15</th>
<th>dup key=1 5</th>
<th>key=2 data=11</th>
<th>dup key=2 16</th>
<th>dup key=2 9</th>
<th>key=3 data=20</th>
<th>dup key=3 100</th>
<th>key=4 data=6</th>
<th>key=5 data=5</th>
<th>dup key=5 99</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterating...</td>
<td>key=1 data=10</td>
<td>key=1 data=315</td>
<td>key=1 data=305</td>
<td>key=2 data=11</td>
<td>key=2 data=316</td>
<td>key=2 data=309</td>
<td>key=3 data=20</td>
<td>key=3 data=400</td>
<td>key=4 data=6</td>
<td>key=5 data=5</td>
</tr>
<tr>
<td>key=5 data=5</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
<td>key=5 data=199</td>
</tr>
</tbody>
</table>

**See Also**

- “Replacing and Removing Data” on page 152

**Methods:**

- “REPLACE Method” on page 1062

---

**SETCUR Method**

Specifies a starting key item for iteration.

**Applies to:** Hash iterator package
Syntax

Form 1:  
`package.SETCUR();`

Form 2:  
`package.SETCUR([keys], [data]);`

Arguments

`package`
- specifies the name of the hash iterator package variable.

`[keys]`
- specifies the key variables by using a variable list.
  
  See “Variable Lists” on page 59

`[data]`
- specifies the variables into which to store the data item.
  
  See “Variable Lists” on page 59

Details

The hash iterator enables you to start iteration on any item in the hash package. The SETCUR method sets the starting key for iteration. You reference the starting item with the specified key variables. If the item exists, the data associated with the item is stored in the data variables.

You can use the FIRST or LAST methods to start iteration on the first or last item, respectively.

There are two ways to pass key and data variables to the SETCUR method:

- implicit variable method (Form 1)
  - The key and data variables are implied in the SETCUR method invocation and do not have to be specified.

- variable list method (Form 2)
  - The specified key and data variables are passed explicitly to the SETCUR method.

Example

The following example uses the SETCUR method to start iteration at RA= 18 31.6 instead of the first or last items:

```
declare hiter iter('myhash');
myhash.defineKey('ra');
myhash.defineData('obj', 'ra');
myhash.defineDone();
ra='18 31.6';
rc = iter.setcur();
do while (rc = 0);
    put obj= ra=;
    rc = iter.next();
end;
```
SUM Method

Retrieves the summary value for a given key from the hash table and stores the value in a variable.

 Applies to:  Hash package

Syntax

Form 1:  \( \text{summary-variable} = \text{package}.\text{SUM}(); \)

Form 2:  \( \text{summary-variable} = \text{package}.\text{SUM}([\text{keys}]); \)

Required Arguments

\( \text{summary-variable} \)

specifies a variable that holds the current summary value of the current key.

Note  A return code that specifies success or failure is not returned by the method.

\( \text{package} \)

specifies an instance of the hash package variable.

\([\text{keys}]\)

specifies the key values by using a variable list.

See  “Variable Lists” on page 59

Details

You use the SUM method to retrieve key summaries from the hash package. The SUM method retrieves the summary value for a given key when only one data item exists per key. For more information, see “Maintaining Key Summaries” on page 151.

There are two ways to pass key variables to the SUM method:

- implicit variable method (Form 1)
  
  The key variables are implied in the SUM method invocation and do not have to be specified.

- variable list method (Form 2)
  
  The specified key variables are passed explicitly to the SUM method.

Comparisons

The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key.
Example: Retrieving the Key Summary for a Given Key

The following example uses the SUM method to retrieve the key summary for a given key, 99.

```plaintext
data _null_;    
declare double k count total;    
declare package hash myhash(0, '', 'a', '', 'count');    
method init();    
    myhash.defineKey('k');    
    myhash.defineDone();

k = 99;    
count = 1;    
myhash.add();

    /* COUNT is given the value 2.5 and the */    
    /* FIND sets the summary to 3.5*/    
    count = 2.5;    
    myhash.find();

    /* The COUNT of 3 is added to the FIND and */    
    /* sets the summary to 6.5. */    
    count = 3;    
    myhash.find();

    /* The COUNT of -1 sets the summary to 5.5. */    
    count = -1;    
    myhash.find();

    /* The SUM method gives the current value of */    
    /* the key summary to the variable TOTAL. */    
    total = myhash.sum();

    /* The PUT statement prints total=5.5 in the log. */    
    put total=;

end;
enddata;
run;
```

See Also

- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “SUMDUP Method” on page 1069

**SUMDUP Method**

Retrieves the summary value for the current data item of the current key and stores the value in a variable.

**Applies to:** Hash package
Syntax

Form 1:  

\[ \text{summary-variable} = \text{package.SUMDUP}(); \]

Form 2:  

\[ \text{summary-variable} = \text{package.SUMDUP}([\text{keys}]); \]

Arguments

\textit{summary-variable}

specifies a variable that holds the current summary value for the current data item of the current key.

\textbf{Note} A return code that specifies success or failure is not returned by the method.

\textit{package}

specifies an instance of the hash package variable.

\textit{[keys]}

specifies the key values by using a variable list.

\textbf{See} “Variable Lists” on page 59

Details

You use the SUMDUP method to retrieve key summaries from the hash package when a key has multiple data items. For more information, see “Maintaining Key Summaries” on page 151.

There are two ways to pass key variables to the SUMDUP method:

- implicit variable method (Form 1)

  The key variables are implied in the SUMDUP method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the SUMDUP method.

Comparisons

The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key. The SUM method retrieves the summary value for a given key when only one data item exists per key.

Example: Retrieving a Summary Value

The following example uses the SUMDUP method to retrieve the summary value for the current data item.

```plaintext
data hashinp;
dcl double k v;
method init();
k=1; v=2; output;
k=1; v=4; output;
k=1; v=8; output;
k=2; v=2; output;
k=3; v=4; output;
```
k=2; v=8; output;
end;
enddata;
run;

data results(keep=(k v sumres));
dcl double k v si sumres;
dcl package hash h(8, 'hashinp', 'ascending', '', 'si', 'multidata');

method testsuminc(double kval, double suminc);
dcl double rc;
si = suminc;
put 'input:' kval= suminc=;
k=kval; rc=h.find();
if (rc=0) then do;
  rc=h.has_next();
do while(rc=0);
  put 'next:' k= v=;
  h.find_next();
  rc=h.has_next();
end;
  rc=h.has_prev();
do while(rc=0);
  put 'prev:' k= v=;
  h.find_prev();
  rc=h.has_prev();
si=0;
k=kval; rc=h.find();
do while(rc=0);
  sumres = h.sumdup();
  output;
  rc=h.find_next();
end;
end;

method init();
h.defineKey('k');
h.defineData('k');
h.defineData('v');
h.defineDone();
testsuminc(1.0, 2.0);
testsuminc(2.0, 20.0);
testsuminc(3.0, 4.0);
end;

method term();
dcl package hiter hi('h');
rc=hi.first();
do while (rc=0);
  put 'final:' k= v=;
  rc=hi.next();
end;
enddata;
enddata;
run;
data _null_;  
  method run();  
    set results;  
    put k= v= sumres=;  
  end;  
edndata;  
run;  

The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>k=</th>
<th>v=</th>
<th>sumres=</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

See Also

- “Implicit Variable and Variable List Methods” on page 149

Methods:

- “SUM Method” on page 1068

### SUMINC Method

Specifies a variable that maintains a summary count of hash package keys.

**Applies to:** Hash package

**Syntax**

```
package.SUMINC(suminc-variable);
```

**Arguments**

- `package`
  - specifies an instance of the hash package variable.
- `suminc-variable`
  - specifies a variable that maintains a summary count of hash package keys.

**Details**

You can maintain a summary count for a hash package key by using the SUMINC parameter or method. SUMINC instructs the hash package to allocate internal storage in each record to store a summary value in the record each time that the record is used by a FIND, CHECK, or REF method. The SUMINC value is also used to maintain a summary count of hash parameter keys after a FIND, CHECK, or REF method. SUMINC is given a variable, which holds the sum increment, that is, how much to add to the key summary for each reference to the key. The SUMINC value can be greater than, less than, or equal to 0.
The SUMINC value is also used to initialize the summary on an ADD method. Each time the ADD method occurs, the key to the SUMINC value is initialized.

The SUMINC variable treats a missing or null value as zero, like the SUM function. For example, a key summary changes using the current value of the variable.

For more information, see “Maintaining Key Summaries” on page 151.

Note: Alternatively, you can use the suminc parameter in the DECLARE PACKAGE statement or the _NEW_ operator to retrieve the summary value for the current data item of the current key.

Example

See the example in the “SUMDUP Method” on page 1069.

See Also

• “Providing Initialization Data for a Hash Package” on page 148

Methods:

• “CHECK Method” on page 1007
• “FIND Method” on page 1027
• “REF Method” on page 1056

Operators:

• “_NEW_ Operator, Hash Package” on page 1044

Statements:

• “DECLARE PACKAGE Statement, Hash Package” on page 1012
Chapter 32
DS2 HTTP Package Methods, Operators, and Statements

Method Naming Convention
When discussing a similar set of methods, this document uses a short name and an asterisk (*) to designate the set of methods as a whole.

For example, when the discussion involves both the SETREQUESTBODYASBINARY and SETREQUESTBODYASSTRING methods, the documentation reads “the SETREQUESTBODY* methods”.

Dictionary
ABORT Method
ADDERQUESTHEADER Method
CREATEGETMETHOD Method
CREATEHEADMETHOD Method
CREATEPOSTMETHOD Method
DECLARE PACKAGE Statement, HTTP Package
DELETE Method, HTTP Package
EXECUTEMETHOD Method
EXECUTEMETHODSTREAM Method
GETRESPONSEBODYASBINARY Method
GETRESPONSEBODYASSTRING Method
GETRESPONSECONTENTTYPE Method
GETRESPONSEHEADERSASSTRING Method
GETSTATUSCODE Method
_NEW_ Operator, HTTP Package
SETREQUESTBODYASBINARY Method
SETREQUESTBODYASSTRING Method
SETREQUESTCONTENTTYPE Method
SETSOCKETTIMEOUT Method
STREAMRESPONSEBODYASBINARY Method
STREAMRESPONSEBODYASSTRING Method
Dictionary

ABORT Method

Stops the execution of the HTTP method.

Syntax

```
package.ABORT();
```

Arguments

`package`

specifies an instance of the HTTP package variable.

Details

Call the ABORT method to stop the HTTP method that is currently running. The ABORT method is useful when you are streaming data and decide that you do not need to see the entire response body.

See Also

- “Using the HTTP Package” on page 158

Methods:

- “EXECUTEMETHOD Method” on page 1084
- “EXECUTEMETHODSTREAM Method” on page 1085

ADDREQUESTHEADER Method

Adds a header to the HTTP method request.

Syntax

```
package.ADDREQUESTHEADER('name', 'value');
```

Arguments

`package`

specifies an instance of the HTTP package variable.

`'name'`

specifies the name of the header field.

`'value'`

specifies the value of the header field.
Details
The header is appended to the end of the list of headers.

Example
h.addRequestHeader('Cookie', 'SSID=31k3q84095jk791gjf');

See Also
- “Using the HTTP Package” on page 158

Methods:
- “CREATEGETMETHOD Method” on page 1077
- “CREATEHEADMETHOD Method” on page 1081
- “CREATEPOSTMETHOD Method” on page 1082

CREATEGETMETHOD Method
Creates an HTTP GET method to retrieve a resource from a web server.

Syntax
package.CREATEGETMETHOD(‘url’);

Arguments
package
  specifies an instance of the HTTP package variable.

‘url’
  specifies the URL of the resource.

  The URL can include query strings (name/value pairs).

Details
Use the CREATEGETMETHOD method to create an HTTP GET method. After you create the GET method, call one of the EXECUTEMETHOD* methods to request the specified resource from the web server.

Examples

Example 1: Create a GET Method and Retrieve an HTTP Resource
The following example requests an HTTP resource and displays the headers and body of the response from the web server.
data _null_;  method init();  declare package http h();  declare varchar(1024) character set utf8 body headers;
Country code: FR, executeMethod() status: 200
Headers:
HTTP/1.1 200 OK
Content-Length: 627
Content-Type: text/xml; charset=UTF-8
Server: WorldBank
Web Server2
Date: Sat, 22 Mar 2014 00:43:30 GMT
X-Cache: MISS from transproxy
Via: 1.1
transproxy (squid)
Connection: keep-alive

Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:country id="FRA">
    <wb:name>France</wb:name>
    <wb:region id="ECS">Europe &amp; Central Asia (all income levels)</wb:region>
    <wb:adminregion id="" />
    <wb:incomeLevel id="OEC">High income: OECD</wb:incomeLevel>
    <wb:lendingType id="LNX">Not classified</wb:lendingType>
    <wb:capitalCity>Paris</wb:capitalCity>
    <wb:longitude>2.35097</wb:longitude>
    <wb:latitude>48.8566</wb:latitude>
  </wb:country>
</wb:countries>
Example 2: Create GET Methods Using Expressions in the Resource URL

The following example generates a separate GET request for each country code in the input data set. Each code is concatenated into the expression that forms the URL of the resource. Note that the final code, ZZ, is not a valid country code.

```plaintext
proc ds2;
data country_codes /overwrite=yes;
dcl char(2) code;
method init();
  code='ES'; output;
  code='FR'; output;
  code='GB'; output;
  code='ZZ'; output; /* unknown country code */
end;
enddata; run; quit;

proc ds2;
data _null_;method run();
declare package http h();
declare varchar(1024) character set utf8 body;
declare int rc status;
/* Build and send a GET method for each country code */
set country_codes;
h.createGetMethod('http://api.worldbank.org/countries/
    || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');
h.executeMethod();

/* If the resource was returned by the server, show the response */
status = h.getStatusCode(); /* get the HTTP status code */
put 'Country code:' code 'executeMethod() status code:' status;
if status eq 200 then do; /* 200 = OK */
  /* retrieve the body from the response that came from the server */
  h.getResponseBodyAsString(body, rc);
  put 'Body:';
  put body;
end;
put;
enddata; run; quit;
```
The following lines are written to the log:

```xml
Country code: ES executeMethod() status code: 200
Body:
<xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="ES">Spain</wb:country>
    <wb:date>1990</wb:date>
    <wb:value>11880</wb:value>
    <wb:decimal>0</wb:decimal>
  </wb:data>
</wb:data>

Country code= FR executeMethod() status code = 200
Body:
<xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
    <wb:date>1990</wb:date>
    <wb:value>20050</wb:value>
    <wb:decimal>0</wb:decimal>
  </wb:data>
</wb:data>

Country code= GB executeMethod() status code = 200
Body:
<xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="GB">United Kingdom</wb:country>
    <wb:date>1990</wb:date>
    <wb:value>16620</wb:value>
    <wb:decimal>0</wb:decimal>
  </wb:data>
</wb:data>

Country code= ZZ executeMethod() status code = 200
Body:
<xml version="1.0" encoding="utf-8"?>
  <wb:message id="120" key="Parameter 'country' has an invalid value">The provided parameter value is not valid</wb:message>
</wb:error>
```

See Also

- “Using the HTTP Package” on page 158

Methods:

- “EXECUTEMETHOD Method” on page 1084
CREATEHEADMETHOD Method

Creates an HTTP HEAD method to test whether a web resource exists and to retrieve its headers.

Syntax

```plaintext
package.CREATEHEADMETHOD('url');
```

Arguments

- `package` specifies an instance of the HTTP package variable.
- `'url'` specifies the URL of the resource.

**Tip** The URL can include query strings (name/value pairs).

Details

Use the CREATEHEADMETHOD method to create an HTTP HEAD method. After you create the HEAD method, call the EXECUTEMETHOD method to send the request to the web server.

Example

The following example creates a HEAD method, sends the request to the web server, and displays the headers of the response.

```plaintext
data _null_;    method init();    declare package http h();    declare varchar(1024) character set utf8 headers;    declare int rc status;    declare char(2) code;

/* Build and send a HEAD method for a resource */    code = 'GB';    h.createHeadMethod('http://api.worldbank.org/countries/' || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');    h.executeMethod();

/* If the resource headers were returned by the server, show them */    status = h.getStatusCode(); /* get the HTTP status code */    put 'Country code:' code 'executeMethod() status:' status;    if status eq 200 then do; /* 200 = OK */        /* retrieve the headers from the response that came from the server */        h.getResponseHeadersAsString(headers, rc);        put 'Headers:';        put headers;    end;
end;
```
The following lines are written to the log:

```
Country code: GB executeMethod() status: 200
Headers:
HTTP/1.1 200 OK
Content-Length: 0
Content-Type: text/xml; charset=UTF-8
Server: WorldBank Web
Server1
Date: Tue, 25 Mar 2014 21:05:48 GMT
X-Cache: MISS from transproxy
Via: 1.1 transproxy
(squid)
Connection: keep-alive
```

**See Also**

- “Using the HTTP Package” on page 158

**Methods:**

- “EXECUTEMETHOD Method” on page 1084

---

**CREATEPOSTMETHOD Method**

Creates an HTTP POST method to request the web server to accept data for a resource.

**Syntax**

```
package.CREATEPOSTMETHOD('url');
```

**Arguments**

- `package`
  - specifies an instance of the HTTP package variable.

- `'url'`
  - specifies the URL of the resource.
    
    Tip The URL can include query strings (name/value pairs).

**Details**

Use the CREATEPOSTMETHOD method to create an HTTP POST method.

Because the POST method requires a body, complete the POST method by doing the following:

- Add the body content by calling one of the SETREQUESTBODY* methods, depending on the content type.

- Indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the `Content-Type:` header.

Then, call the EXECUTEMETHOD method to send the request to the web server.
Note: The content length is computed by the DS2 HTTP client after transcoding the data.

See Also

• “Using the HTTP Package” on page 158

Methods:

• “ADDREQUESTHEADER Method” on page 1076
• “EXECUTEMETHOD Method” on page 1084
• “SETREQUESTBODYASBINARY Method” on page 1093
• “SETREQUESTBODYASSTRING Method” on page 1094
• “SETREQUESTCONTENTTYPE Method” on page 1095

DECLARE PACKAGE Statement, HTTP Package

Creates a package variable and enables you to create an instance of the HTTP package.

Category: Local

Tip: The PACKAGE statement is not required for an HTTP package.

Syntax

DECLARE PACKAGE HTTP variable( );

Arguments

variable

specifies a variable that can reference an instance of the HTTP package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use an HTTP package to construct an HTTP client to access HTTP web servers. The HTTP package is predefined for DS2 programs.

You declare an HTTP package by using the DECLARE PACKAGE statement. When a package is declared, a variable is created that can reference an instance of the package. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of an HTTP package:

• Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  
  declare package http httpclt;
  httpclt = _new_ http();

• Use the DECLARE PACKAGE statement along with its constructor syntax:
  
  declare package http httpclt();
Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

- “Using the HTTP Package” on page 158
- “Package Constructors and destructors” on page 142

Operators:

- “_NEW_ Operator, HTTP Package” on page 1092

DELETE Method, HTTP Package

Deletes an HTTP package instance and frees its resources.

**Note:** The DELETE method is not required. When an HTTP package goes out of scope, the package is deleted.

**Syntax**

```
package.DELETE();
```

**Arguments**

```
package
```

specifies the name of the HTTP package variable.

**Details**

When you no longer need the HTTP package, delete it by using the DELETE method. If you attempt to use an HTTP package instance after you delete it, an error is written to the log.

**See Also**

“Package Constructors and destructors” on page 142

EXECUTEMETHOD Method

Executes the HTTP method and enables retrieval of the response body as a complete entity.

**Syntax**

```
package.EXECUTEMETHOD();
```

**Arguments**

```
package
```

specifies an instance of the HTTP package variable.
Details

You must create a GET, HEAD, or POST method before you call the EXECUTEMETHOD method to send the request to the web server. The EXECUTEMETHOD method sends the last method that was created.

If the response has a body, the response body must be retrieved as an entire entity by using one of the GETRESPONSEBODYAS* methods.

Note: The EXECUTEMETHOD method does not support streaming of the response body. If you use one of the STREAMRESPONSEBODYAS* methods to retrieve the response body after sending the request with the EXECUTEMETHOD method, a run-time error occurs. Use the EXECUTEMETHODSTREAM method instead.

See Also

• “Using the HTTP Package” on page 158

Methods:

• “EXECUTEMETHODSTREAM Method” on page 1085
• “GETSTATUSCODE Method” on page 1091
• “GETRESPONSECONTENTTYPE Method” on page 1088
• “GETRESPONSEBODYASBINARY Method” on page 1086
• “GETRESPONSEBODYASSTRING Method” on page 1087
• “SETSOCKETTIMEOUT Method” on page 1096

EXECUTEMETHODSTREAM Method

Executes the HTTP method and enables streaming of the response body from the HTTP server.

Syntax

```
package.EXECUTEMETHODSTREAM()
```

Arguments

`package`

specifies an instance of the HTTP package variable.

Details

Use the EXECUTEMETHODSTREAM method when you want to stream the response body from the web server in chunks, using either the STREAMRESPONSEBODYASBINARY or STREAMRESPONSEBODYASSTRING method. This is useful when you want to process the response without waiting for all of the data to arrive from the server.

You must create a GET method before you call the EXECUTEMETHODSTREAM method to send the request to the web server. The EXECUTEMETHODSTREAM method sends the last method that was created.
Note: The EXECUTEMETHODSTREAM method does not support retrieval of the response body as an entire entity. If you use one of the GETRESPONSEBODYAS* methods to retrieve the response body after sending the request with the EXECUTEMETHODSTREAM method, a run-time error occurs. Use the EXECUTEMETHOD method instead.

See Also

- “Using the HTTP Package” on page 158

Methods:

- “EXECUTEMETHOD Method” on page 1084
- “GETSTATUSCODE Method” on page 1091
- “GETRESPONSECONTENTTYPE Method” on page 1088
- “SETSOCKETTIMEOUT Method” on page 1096
- “STREAMRESPONSEBODYASBINARY Method” on page 1096
- “STREAMRESPONSEBODYASSTRING Method” on page 1097

GETRESPONSEBODYASBINARY Method

Returns the entire body from the HTTP response in binary format.

Syntax

```
package.GETRESPONSEBODYASBINARY(variable, rc);
```

Arguments

- **package**
  - specifies an instance of the HTTP package variable.
- **variable**
  - specifies the binary variable to hold the entire response body.
- **rc**
  - specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEBODYASBINARY method to retrieve the response body in binary format. The response body is returned as one entity.

You can call the GETRESPONSEBODYASBINARY method after using the EXECUTEMETHOD method to retrieve the response body returned by the web server.

Note: The GETRESPONSEBODYASBINARY method does not return until the web client has received all the data from the web server.

Note: The response body is not transcoded.

Note: The EXECUTEMETHOD method does not support streaming of the response body.
See Also

• “Using the HTTP Package” on page 158

Methods:

• “CREATEGETMETHOD Method” on page 1077
• “EXECUTEMETHOD Method” on page 1084
• “GETRESPONSEBODYASSTRING Method” on page 1087

GETRESPONSEBODYASSTRING Method

Returns the entire body from the HTTP response in character string format.

Syntax

```plaintext
package.GETRESPONSEBODYASSTRING(variable, rc);
```

Arguments

- `package` specifies an instance of the HTTP package variable.
- `variable` specifies the variable to hold the entire response body.
- `rc` specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEBODYASSTRING method to retrieve the response body in character string format. The response body is returned as one entity.

You can call the GETRESPONSEBODYASSTRING method after using the EXECUTEMETHOD method to retrieve the response body returned by the web server.

Note: The GETRESPONSEBODYASSTRING method does not return until the web client has received all the data from the web server.

Note: The response body is transcoded to the encoding of the string if the encodings are different.

Note: The EXECUTEMETHOD method does not support streaming of the response body.

Example

```plaintext
data _null_; method init(); declare package http h(); declare varchar(1024) character set utf8 body; declare int rc status; declare char(2) code;
```
/* Build and send a GET method for a resource */

code = 'FR';
h.createGetMethod('http://api.worldbank.org/countries/'
    || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1991');
h.executeMethod();

/* If the resource was returned by the server, show the response */

status = h.getStatusCode();   /* get the HTTP status code */

put 'Requested resource for country code:' code 'executeMethod() status:' status;
if status eq 200 then do;     /* 200 = OK */
    /* retrieve the body from the response that came from the server */
    h.getResponseBodyAsString(body, rc);
    put 'Body:';
    put body;
end;
enddata; run;

The following lines are written to the log.

```
Requested resource for country code: FR executeMethod() status: 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="2"
    xmlns:wb="http://www.worldbank.org">
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
    <wb:date>1991</wb:date>
    <wb:value>20880</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
<wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
    <wb:date>1990</wb:date>
    <wb:value>20050</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
</wb:data>
```

See Also

- “Using the HTTP Package” on page 158

Methods:

- “CREATEGETMETHOD Method” on page 1077
- “EXECUTEMETHOD Method” on page 1084
- “GETRESPONSEBODYASBINARY Method” on page 1086

**GETRESPONSECONTENTTYPE Method**

Returns the content type from the HTTP response.
Syntax

```
content-type-variable=package.GETRESPONSECONTENTTYPE( );
```

Arguments

- **content-type-variable**: specifies the variable to hold the content type value of the HTTP response.
  
  **Note**: Consult an HTTP reference for a list of possible content types.

- **package**: specifies an instance of the HTTP package.

Details

Use the GETRESPONSECONTENTTYPE method to retrieve the content type from the latest response message.

Example

```plaintext
data _null_;  
method init();  
declare package http h();  
declare varchar(1024) character set utf8 body contentType headers;  
declare int rc status;  
  
  /* Build and send a HEAD method for the 'ES' resource */  
h.createHeadMethod('http://api.worldbank.org/countries/ES');  
h.executeMethod();  
  
  /* If the resource was returned by the server, show the response */  
status = h.getStatusCode(); /* get the HTTP status code */  
put 'Country code: ES, executeMethod() status:' status;  
if status eq 200 then do; /* 200 = OK */  
  /* retrieve the content type from the response that came from the server */  
  contentType = h.getResponseContentType();  
  put 'Content type:' contentType;  
end;  
end;  
enddata; run;
```

The following lines are written to the log.

```
Country code: ES, executeMethod() status: 200  
Content type: text/xml; charset=UTF-8
```

See Also

- “Using the HTTP Package” on page 158

Methods:

- “EXECUTEMETHOD Method” on page 1084
- “EXECUTEMETHODSTREAM Method” on page 1085
GETRESPONSEHEADERSASSTRING Method

Returns the response headers from the HTTP method in character string format.

Syntax

```plaintext
package.GETRESPONSEHEADERSASSTRING(variable, rc);
```

Arguments

- `package` specifies an instance of the HTTP package variable.
- `variable` specifies the variable to hold the response headers.
- `rc` specifies the variable to hold the return code value.

**Note:** A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEHEADERSASSTRING method to retrieve all headers from the HTTP response. You can use the GETRESPONSEHEADERSASSTRING method after using either the EXECUTEMETHOD method or the EXECUTEMETHODSTREAM method to send any HTTP request to the server.

Example

The following example creates and sends a HEAD method and uses the GETRESPONSEHEADERSASSTRING method to display the headers of the response from the web server.

```plaintext
data _null_; method init();
declare package http h();
declare varchar(1024) character set utf8 headers;
declare int rc status;
declare char(2) code;
/* Build and send a HEAD method for a resource */
code = 'GB';
h.createHeadMethod('http://api.worldbank.org/countries/' || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');
h.executeMethod();
/* If the resource headers were returned by the server, show them */
status = h.getStatusCode(); /* get the HTTP status code */
put 'Country code:' code 'executeMethod() status:' status;
if status eq 200 then do; /* 200 = OK */
   /* retrieve the headers from the response that came from the server */
   h.getResponseHeadersAsString(headers, rc);
   put 'Headers:';
```

Chapter 32 • DS2 HTTP Package Methods, Operators, and Statements
The following lines are written to the log:

<table>
<thead>
<tr>
<th>Country code: GB</th>
<th>executeMethod() status: 200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers:</td>
<td></td>
</tr>
<tr>
<td>HTTP/1.1 200 OK</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 0</td>
<td></td>
</tr>
<tr>
<td>Content-Type: text/xml; charset=UTF-8</td>
<td></td>
</tr>
<tr>
<td>Server: WorldBank Web</td>
<td></td>
</tr>
<tr>
<td>Server1</td>
<td></td>
</tr>
<tr>
<td>Date: Tue, 25 Mar 2014 21:05:48 GMT</td>
<td></td>
</tr>
<tr>
<td>X-Cache: MISS from transproxy</td>
<td></td>
</tr>
<tr>
<td>Via: 1.1 transproxy</td>
<td></td>
</tr>
<tr>
<td>{squid}</td>
<td></td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td></td>
</tr>
</tbody>
</table>

See Also

- “Using the HTTP Package” on page 158

Methods:

- “CREATEGETMETHOD Method” on page 1077
- “CREATEHEADMETHOD Method” on page 1081
- “CREATEPOSTMETHOD Method” on page 1082
- “EXECUTEMETHOD Method” on page 1084
- “EXECUTEMETHODSTREAM Method” on page 1085

GETSTATUSCODE Method

Returns the HTTP status code from the most recently executed HTTP method.

Syntax

```
status-code-variable=package.GETSTATUSCODE( );
```

Arguments

`status-code-variable`  
specifies the variable to hold the HTTP status code value.  

*Note:* Consult an HTTP reference for possible status codes.

`package`  
specifies an instance of the HTTP package variable.

Details

Use the GETSTATUSCODE method to retrieve the HTTP status code from the most recently executed HTTP method.
Example

data _null_;  
  method init();  
    declare package http h();  
    declare int status;  

    /* Build and send a HEAD method for a resource */  
    h.createHeadMethod('http://support.sas.com/documentation/');  
    h.executeMethod();  

    /* If the resource was returned by the server, show the response */  
    status = h.getStatusCode();  /* get the HTTP status code */  
    put 'HEAD method created for resource:';  
    put 'http://support.sas.com/documentation/';  
    put 'executeMethod() status:' status;  
end;  
enddata; run;  

The following lines are written to the log.

HEAD method created for resource:
http://support.sas.com/documentation/
executeMethod() status: 200

See Also

• “Using the HTTP Package” on page 158

Methods:

• “EXECUTEMETHOD Method” on page 1084
  • “EXECUTEMETHODSTREAM Method” on page 1085

_NEW_ Operator, HTTP Package

Constructs an instance of an HTTP package.

**Note:** The escape character (\) before the bracket indicates that the bracket is required in the syntax.

**Syntax**

\[
\text{package-variable} = \_NEW_ \{[\text{THIS}] | \{\text{package-instance}\}\} \text{HTTP}();
\]

**Arguments**

package-variable

specifies a name that can reference an instance of the package.

[THIS]

specifies that the package instance has global scope.

See “Packages and Scope” on page 137
[package-instance] specifies that the new package instance has the same scope as package-instance. package-instance must be an existing package instance and is referenced by its package variable name. The package type of package-instance does not have to be the same as the instance that is being instantiated.

Note: Associating the scope of one package instance with another is useful when you want instances to be automatically deleted at the same time.

See “Package-Specific Scope” on page 139

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When an HTTP package is declared, the variable representing the package can be considered an instance of the package. This means that two different HTTP package variables represent two completely separate copies of the package.

You declare an HTTP package by using the DECLARE PACKAGE statement. After you declare the new HTTP package, use the _NEW_ operator to instantiate the package.

declare package http h;
h = _new_ http();

As an alternative to the two-step process of using the DECLARE PACKAGE statement and the _NEW_ operator to declare and instantiate an HTTP package, you can declare and instantiate a package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

declare package http h();

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

• “Using the HTTP Package” on page 158
• “Package Constructors and Destructors” on page 142

Statements:

• “DECLARE PACKAGE Statement, HTTP Package” on page 1083
• “PACKAGE Statement” on page 921

SETREQUESTBODYASBINARY Method

Adds the specified body to the HTTP method request in binary format.

Syntax

package.SETREQUESTBODYASBINARY(variable);
Arguments

package
  specifies an instance of the HTTP package variable.

variable
  specifies the binary variable that contains the request body data.

Details

Use the SETREQUESTBODYASBINARY method to add the request body, in binary format, to the HTTP method.

To complete the HTTP method, indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the Content-Type: header. Then, call the EXECUTEMETHOD method to send the request to the web server.

Note: The content length is computed by the DS2 HTTP client after transcoding the data.

See Also

• “Using the HTTP Package” on page 158

Methods:

• “ADDREQUESTHEADER Method” on page 1076
• “CREATEPOSTMETHOD Method” on page 1082
• “EXECUTEMETHOD Method” on page 1084
• “SETREQUESTBODYASBINARY Method” on page 1094
• “SETREQUESTCONTENTTYPE Method” on page 1095

SETREQUESTBODYASSTRING Method

Adds the specified body to the HTTP method request in character string format.

Syntax

package.SETREQUESTBODYASSTRING(variable);

Arguments

package
  specifies an instance of the HTTP package variable.

variable
  specifies the string variable that contains the request body data.

Details

Use the SETREQUESTBODYASSTRING method to add the request body, in character string format, to the HTTP method.
To complete the HTTP method, indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the *Content-Type:* header. Then, call the EXECUTEMETHOD method to send the request to the web server.

*Note:* The content length is computed by the DS2 HTTP client after transcoding the data.

*Note:* If the encoding of the provided request data differs from the specified content type, the data is transcoded to the encoding that is specified by content type.

**See Also**

- “Using the HTTP Package” on page 158

**Methods:**

- “ADDREQUESTHEADER Method” on page 1076
- “CREATEPOSTMETHOD Method” on page 1082
- “EXECUTEMETHOD Method” on page 1084
- “SETREQUESTBODYASBINARY Method” on page 1093
- “SETREQUESTCONTENTTYPE Method” on page 1095

---

**SETREQUESTCONTENTTYPE Method**

Specifies the content type of the body of the HTTP method request.

**Syntax**

```
package.SETREQUESTCONTENTTYPE(content-type);
```

**Arguments**

- `content-type`
  
  specifies the variable that contains the content type value.
  
  *Note:* Consult an HTTP reference for possible content types.

- `package`
  
  specifies an instance of the HTTP package variable.

**Details**

Use the SETREQUESTCONTENTTYPE method to specify the content type of the body of the HTTP method.

**See Also**

- “Using the HTTP Package” on page 158

**Methods:**

- “CREATEPOSTMETHOD Method” on page 1082
- “EXECUTEMETHOD Method” on page 1084
SETSOCKETTIMEOUT Method
Specifies the socket time-out value to wait for a response from an HTTP web server.

Syntax

```
package.SETSOCKETTIMEOUT(time-out-value);
```

Arguments

- `package` specifies an instance of the HTTP package variable.
- `time-out-value` specifies the default socket time-out, in milliseconds, to wait for a response from the web server.

Details

Use the SETSOCKETTIMEOUT method to specify how long to wait for a response from the web server.

The SETSOCKETTIMEOUT method can be set for each new CREATE method when the default time-out is too long or too short.

Example

```
h. setSocketTimeout(1000);
```

See Also

“Using the HTTP Package” on page 158

STREAMRESPONSEBODYASBINARY Method
Streams the body, in chunks, from the HTTP response in binary format.

Syntax

```
package.STREAMRESPONSEBODYASBINARY(variable, rc);
```

Arguments

- `package` specifies an instance of the HTTP package variable.
- `variable` specifies the variable that will contain the response data chunk.
rc specifies the variable to hold the return code value.

_Note:_ A return code value of 0 indicates success; a value of 1 indicates failure.

**Details**

Use the STREAMRESPONSEBODYASBINARY method to retrieve the response body in binary format. The response body is streamed, in chunks.

You can call the STREAMRESPONSEBODYASBINARY method after using the EXECUTEMETHODSTREAM method.

If you do not want to complete the streaming of the body data, call the ABORT method to stop the execution of the method.

_Note:_ The EXECUTEMETHODSTREAM method does not support retrieval of the response body as one entity.

**See Also**

- “Using the HTTP Package” on page 158

**Methods:**

- “ABORT Method” on page 1076
- “EXECUTEMETHODSTREAM Method” on page 1085
- “STREAMRESPONSEBODYASSTRING Method” on page 1097

---

**STREAMRESPONSEBODYASSTRING Method**

Streams the body, in chunks, from the HTTP response in character string format.

**Applies to:** HTTP package

**Syntax**

```
package.STREAMRESPONSEBODYASSTRING(variable, rc);
```

**Arguments**

- **package**
  specifies an instance of the HTTP package variable.

- **variable**
  specifies the variable that will contain the response data chunk.

- **rc**
  specifies the variable to hold the return code value.

  _Note:_ A return code value of 0 indicates success; a value of 1 indicates failure.

**Details**

Use the STREAMRESPONSEBODYASSTRING method to retrieve the response body in character string format. The response body is streamed, in chunks.
You can call the STREAMRESPONSEBODYASBINARY method after using the EXECUTEMETHODSTREAM method.

If you do not want to complete the streaming of the body data, call the ABORT method to stop the execution of the method.

*Note:* The EXECUTEMETHODSTREAM method does not support retrieval of the response body as one entity.

**See Also**

- “Using the HTTP Package” on page 158

**Methods:**

- “ABORT Method” on page 1076
- “EXECUTEMETHODSTREAM Method” on page 1085
- “STREAMRESPONSEBODYASBINARY Method” on page 1096
Chapter 33
DS2 JSON Package Methods, Operators, and Statements

Dictionary

CREATEPARSER Method .................................................................................. 1099
CREATEWRITER Method .............................................................................. 1100
DECLARE PACKAGE Statement, JSON Package ............................................ 1101
DESTROYPARSER Method .......................................................................... 1102
DESTROYWRITER Method ........................................................................... 1103
GETNEXTTOKEN Method ............................................................................. 1103
ISBOOLEANFALSE Method ............................................................................ 1105
ISBOOLEANTRUE Method ............................................................................... 1106
ISFLOAT Method .......................................................................................... 1106
ISINTEGER Method ...................................................................................... 1107
ISLABEL Method .......................................................................................... 1108
ISLEFTBRACE Method .................................................................................. 1108
ISLEFTBRACKET Method .............................................................................. 1109
ISNULL Method ............................................................................................ 1109
ISNUMERIC Method ...................................................................................... 1110
ISPARTIAL Method ........................................................................................ 1110
ISRIGHTRACE Method .................................................................................. 1111
ISRIGHTRACKET Method ............................................................................. 1112
ISSTRING Method ........................................................................................ 1112
_NEW_ Operator, JSON Package ................................................................. 1113
SETPARSERINPUT Method .......................................................................... 1114
WRITEARRAYOPEN Method ....................................................................... 1114
WRITEBOOLEANFALSE Method ................................................................. 1115
WRITEBOOLEANTRUE Method ..................................................................... 1116
WRITECLOSE Method .................................................................................... 1116
WRITEDOUBLE Method ................................................................................. 1117
WRITENULL Method ...................................................................................... 1120
WRITEOBJOPEN Method ............................................................................. 1120
WRITERGETTEXT Method ............................................................................ 1121
WRITESTRING Method .................................................................................. 1121

Dictionary

CREATEPARSER Method

Creates a JSON Parser instance.
Syntax

Form 1:  

```
package.CREATEPARSER ();
```

Form 2:  

```
package.CREATEPARSER (json-text, tipping-size);
```

Form 3:  

```
package.CREATEPARSER (json-text);
```

Form 4:  

```
package.CREATEPARSER (tipping-size);
```

Arguments

- **package**: specifies an instance of the JSON package.
- **json-text**: specifies the input JSON text to be parsed.
- **tipping-size**: specifies the minimum number of characters of output JSON text to accumulate before calling the string call-back routine for strings longer than tipping-size.

Default: 0, which indicates that only complete strings are returned to the string callback, regardless of length.

Restriction: The maximum number of characters that can be returned is (tipping-size + 4) when tipping-size is set.

Details

If you use Form 1 or Form 4 of the CREATEPARSER method syntax, you should subsequently call the SETPARSERINPUT method to provide the JSON text to be parsed.

See Also

- “Using the JSON Package” on page 161

Methods:

- “DESTROYPARSER Method” on page 1102
- “SETPARSERINPUT Method” on page 1114

CREATEWRITER Method

Creates a JSON writer instance.

Syntax

```
package.CREATEWRIITER ([flags='PRETTY' | 'NOPRETTY']);
```
**Arguments**

**package**

specifies an instance of the JSON package.

**flags**='PRETTY' | 'NOPRETTY'

specifies how to format the JSON output. flags can be one of the following values:

'PRETTY'

creates a more human-readable format that uses indentation to illustrate the JSON container structure.

'NOPRETTY'

writes the output in a single line

**Default** NOPRETTY

**Note** More flags might be available in future releases.

**Details**

The DS2 JSON package's writer currently does not support streaming. Instead, the Write instances are gathered in memory until retrieved by calling the WRITERGETTEXT method.

**Example**

For an example, see “WRITEARRAYOPEN Method” on page 1114.

**See Also**

- “Using the JSON Package” on page 161

**Methods:**

- “DESTROYWRITER Method” on page 1103
- “WRITERGETTEXT Method” on page 1121

---

**DECLARE PACKAGE Statement, JSON Package**

Creates a package variable and enables you to create an instance of the JSON package.

**Category:** Local

**Syntax**

DECLARE PACKAGE JSON *variable* ( );

**Arguments**

*variable*

specifies a name that can reference an instance of the JSON package.
Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a JSON package to create and parse JSON text. The JSON package is predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a JSON package.

• Use the DECLARE PACKAGE statement along with the _NEW_ operator:

```plaintext
declare package json j;
j = _new_ json();
```

• Use the DECLARE PACKAGE statement along with its constructor syntax:

```plaintext
declare package json j();
```

See Also

• “Using the JSON Package” on page 161
• “Package Constructors and Destructors” on page 142

Operators:

• “_NEW_ Operator, JSON Package” on page 1113

DESTROYPARSER Method

Destroys a JSON Parser instance.

Syntax

```plaintext
package.DESTROYPARSER ();
```

Arguments

`package`

specifies an instance of the JSON package.

See Also

• “Using the JSON Package” on page 161

Methods:

• “CREATEPARSER Method” on page 1099
DESTROYWRITER Method

Destroys a JSON writer instance.

Syntax

```
package.DESTROYWRITER();
```

Arguments

- `package` specifies an instance of the JSON package.

Example

For an example, see “WRITEARRAYOPEN Method” on page 1114.

See Also

- “Using the JSON Package” on page 161

Methods:

- “CREATEGENERATOR Method” on page 1100

GETNEXTTOKEN Method

Returns the next validated JSON language item or element from the JSON text.

Syntax

```
1. package.GETNEXTTOKEN (rc, token-type, parse-flags);
2. package.GETNEXTTOKEN (rc, token, token-type, parse-flags);
3. package.GETNEXTTOKEN (rc, token, token-type, parse-flags, line-number, column-number);
```

Arguments

- `package` specifies an instance of the JSON package.
- `rc` specifies the variable to hold the return code value. Possible return code values are as follows:
  - 0 Success
  - 100 The output token argument's maximum length was not large enough and truncation occurred.
Done. Depending on the use case, this might or might not be expected.

End of text. Depending on the use case, this might or might not be expected.

An error occurred while parsing the text.

token-type

token-type can be one of the following values:

4  Boolean true
8  Boolean false
16 Left bracket ( [ )
32 Right bracket ( ] )
64 Left brace ( { )
128 Right brace ( } )
256 String
512 Numeric
1024 Null

parse-flags

parse-flags output value can be an integer flag set consisting of one or more of the following flags:

0x00000001 token is a label in an object
0x00000002 token is not complete
0x00000003 token is an integral numeric
0x00000004 token is a floating point number

token

is the next token or string.

line-number

Updates the given integer variable argument with the line number within the text where the token is located.

Tip You can use the line number to help determine the location of the token within the text.

column-number

Updates the given integer variable argument with the column number within the text where the token is located.

Tip You can use the column number to help determine the location of the token within the text.

Details

All of the arguments to the GETNEXTTOKEN method are passed by reference.

You can use the IS* methods to test the token type.
See Also

Methods:
- “ISBOOLEANFALSE Method” on page 1105
- “ISBOOLEANTRUE Method” on page 1106
- “ISFLOAT Method” on page 1106
- “ISINTEGER Method” on page 1107
- “ISLABEL Method” on page 1108
- “ISLEFTBRACE Method” on page 1108
- “ISLEFTBRACKET Method” on page 1109
- “ISNULL Method” on page 1109
- “ISNUMERIC Method” on page 1110
- “ISPARTIAL Method” on page 1110
- “ISRIGHTBRACE Method” on page 1111
- “ISRIGHTBRACKET Method” on page 1112
- “ISSTRING Method” on page 1112

ISBOOLEANFALSE Method

Returns true if the token is false.

Syntax

`package.ISBOOLEANFALSE (token-type);`

Arguments

`package`

specifies an instance of the JSON package.

`token-type`

specifies the token type that was obtained from the GETNEXTTOKEN method.

Details

You can use the ISBOOLEANFALSE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- “Using the JSON Package” on page 161

Methods:
- “GETNEXTTOKEN Method” on page 1103
- “ISBOOLEANTRUE Method” on page 1106
ISBOOLEANTRUE Method

Returns true if the token is true.

Syntax

```plaintext
package.ISBOOLEANTRUE (token-type);
```

Arguments

- `package`
  - Specifies an instance of the JSON package.
- `token-type`
  - Specifies the token type that was obtained from the GETNEXTTOKEN method.

Details

You can use the ISBOOLEANTRUE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- “Using the JSON Package” on page 161

Methods:

- “GETNEXTTOKEN Method” on page 1103
- “ISBOOLEANFALSE Method” on page 1105
- “WRITEBOOLEANTRUE Method” on page 1116

ISFLOAT Method

Returns true if the token is a floating point number in text form.

Syntax

```plaintext
package.ISFLOAT (token-type, parse-flags);
```

Arguments

- `package`
  - Specifies an instance of the JSON package.
- `token-type`
  - Specifies the token type that was obtained from the GETNEXTTOKEN method.
- `parse-flags`
  - Specifies the parse flags that were obtained from the GETNEXTTOKEN method.
Details
You can use the ISFLOAT method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also
• “Using the JSON Package” on page 161

Methods:
• “GETNEXTTOKEN Method” on page 1103
• “ISINTEGER Method” on page 1107
• “ISNUMERIC Method” on page 1110

ISINTEGER Method
Returns true if the token is an integer in text form.

Syntax
package.ISINTEGER (token-type, parse-flags);

Arguments
package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

parse-flags
specifies the parse flags that were obtained from the GETNEXTTOKEN method.

Details
You can use the ISINTEGER method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also
• “Using the JSON Package” on page 161

Methods:
• “GETNEXTTOKEN Method” on page 1103
• “ISFLOAT Method” on page 1106
• “ISNUMERIC Method” on page 1110
ISLABEL Method

Returns true if the token is an object label.

Syntax

\texttt{package.ISLABEL (token-type, parse-flags);}

Arguments

\texttt{package}

specifies an instance of the JSON package.

\texttt{token-type}

specifies the token type that was obtained from the GETNEXTTOKEN method.

\texttt{parse-flags}

specifies the parse flags that were obtained from the GETNEXTTOKEN method.

Details

You can use the ISLABEL method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- “Using the JSON Package” on page 161

Methods:

- “GETNEXTTOKEN Method” on page 1103

ISLEFTBRACE Method

Returns true if the token is a left brace \{ \}.

Syntax

\texttt{package.ISLEFTBRACE (token-type);}

Arguments

\texttt{package}

specifies an instance of the JSON package.

\texttt{token-type}

specifies the token type that was obtained from the GETNEXTTOKEN method.

Details

You can use the ISLEFTBRACE method to test the token type that was obtained from the GETNEXTTOKEN method.
See Also
• “Using the JSON Package” on page 161

Methods:
• “GETNEXTTOKEN Method” on page 1103
• “ISRIGHBRACE Method” on page 1111

ISLEFTBRACKET Method
Returns true if the token is a left bracket ( [ ).

Syntax
package.ISLEFTBRACKET (token-type);

Arguments
package
specifies an instance of the JSON package.
token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Details
You can use the ISLEFTBRACKET method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also
• “Using the JSON Package” on page 161

Methods:
• “GETNEXTTOKEN Method” on page 1103
• “ISRIGHBRACKET Method” on page 1112

ISNULL Method
Returns true if the token is null.

Syntax
package.ISNULL (token-type);

Arguments
package
specifies an instance of the JSON package.
ISNUMERIC Method

Returns true if the token is numeric in text form.

Syntax

package.ISNUMERIC (token-type);

Arguments

package

specifies an instance of the JSON package.

token-type

receives the token type that was obtained from the GETNEXTTOKEN method.

Details

You can use the ISNULL method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- “Using the JSON Package” on page 161

Methods:

- “GETNEXTTOKEN Method” on page 1103
- “Writenull Method” on page 1120

ISPARTIAL Method

Returns true if the token is incomplete because of tipping.
**Syntax**

```
package.ISPARTIAL (parse-flags);
```

**Arguments**

- `package` specifies an instance of the JSON package.
- `parse-flags` receives the parse flags that were obtained from the GETNEXTTOKEN method.

**Details**

You can use the ISPARTIAL method to test the token type that was obtained from the GETNEXTTOKEN method.

**See Also**

- “Using the JSON Package” on page 161

**Methods:**

- “GETNEXTTOKEN Method” on page 1103

---

**ISRIGHBRACE Method**

Returns true if the token is a right brace ( }).

**Syntax**

```
package.ISRIGHTBRACE (token-type);
```

**Arguments**

- `package` specifies an instance of the JSON package.
- `token-type` receives the token type that was obtained from the GETNEXTTOKEN method.

**Details**

You can use the ISRIGHTBRACE method to test the token type that was obtained from the GETNEXTTOKEN method.

**See Also**

- “Using the JSON Package” on page 161

**Methods:**

- “GETNEXTTOKEN Method” on page 1103
ISRIGHTBRACKET Method

Returns true if the token is a right bracket (}).

Syntax

```plaintext
package.ISRIGHTBRACKET (token-type);
```

Arguments

- **package**
  - specifies an instance of the JSON package.
- **token-type**
  - receives the token type that was obtained from the GETNEXTTOKEN method.

Details

You can use the ISRIGHTBRACKET method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- “Parsing JSON Text” on page 163

Methods:

- “GETNEXTTOKEN Method” on page 1103
- “ISLEFTBRACKET Method” on page 1109

ISSTRING Method

Returns true if the token is a string.

Syntax

```plaintext
package.ISSTRING (token-type);
```

Arguments

- **package**
  - specifies an instance of the JSON package.
- **token-type**
  - receives the token type that was obtained from the GETNEXTTOKEN method.
Details
You can use the ISSTRING method to test the token type that was obtained from the
GETNEXTTOKEN method.

See Also
• “Using the JSON Package” on page 161

Methods:
• “GETNEXTTOKEN Method” on page 1103

_NEW_ Operator, JSON Package
Constructs an instance of a JSON package.

Note: The escape character ( \ ) before the bracket indicates that the bracket is required in
the syntax.

Syntax
package-variable = _NEW_ [ [THIS] | \[package-instance\] ] JSON( );

Arguments
package-variable
specifies a name that can reference an instance of the package.

[THIS]
specifies that the package instance has global scope.

See “Packages and Scope” on page 137

[package-instance]
specifies that the new package instance has the same scope as package-instance.
package-instance must be an existing package instance, and the type of package-
instance can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

Details
A DS2 package is a collection of variables and methods of which particular instances
can be constructed and used in other DS2 programs.

When a JSON package is declared, the variable representing the package can be
considered an instance of the package. This means that two different package variables
represent two completely separate copies of a package.

You declare a JSON package using the DECLARE PACKAGE statement. After you
declare the new JSON package, use the _NEW_ operator to instantiate the package.

    declare package json jsontxt;
    jsontxt = _new_ json( );
As an alternative to the two-step process of using the DECLARE PACKAGE and the \_NEW\_ operator to declare and instantiate a JSON package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

declare package json jsontxt();

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also
- “Using the JSON Package” on page 161
- “Package Constructors and Destructors” on page 142

Statements:
- “DECLARE PACKAGE Statement, JSON Package” on page 1101

SETPARSERINPUT Method
Provides JSON text to the parser when it needs more text.

Restriction: This method is valid only if the parser does not have any text; an error is returned if it does.

Syntax
package.\texttt{SETPARSERINPUT}(\texttt{[json-text,]});

Arguments

\texttt{package}
- specifies an instance of the JSON package.

\texttt{json-text}
- specifies the JSON text for the parser.

See Also
- “Using the JSON Package” on page 161

Methods:
- “CREATEPARSER Method” on page 1099

WRITEARRAYOPEN Method
Writes the open bracket ( [ ) signifying the beginning of an array.
Syntax

`package.WRITEARRAYOPEN();`

Arguments

`package`

specifies an instance of the JSON package.

Details

The WRITEARRAYOPEN method explicitly opens an array container, which you must explicitly close with the WRITECLOSE method.

Example

The following example creates a writer instance and writes a numeric value to a JSON array container.

```sas
data _null_;  
  method init();  
    dcl package json j();  
    dcl double dblVal;  
    dcl int rc;  
    dcl nvarchar(15) jsontxt;  

    rc = j.createWriter();  
    rc = j.writeArrayOpen();  
    dblVal = 12345678.1234;  
    rc = j.writeDouble( dblVal,13, 5 );  
    rc = j.writeClose();  
    j.writerGetText( rc, jsontxt);  
    put jsontxt=;  
    rc = j.destroywriter();  
  end;  
enddata;  
run;
```

The following line is written to the SAS log:

```
jsontxt=[1.2346e+07   ]
```

See Also

- “Using the JSON Package” on page 161

Methods:

- “WRITECLOSE Method” on page 1116

WRITEBOOLEANFALSE Method

Writes a Boolean false value to the text.
Syntax

`package.WRITEBOOLEANFALSE ( );`

Arguments

`package`

specifies an instance of the JSON package.

See Also

- “Using the JSON Package” on page 161

Methods:

- “ISBOOLEANFALSE Method” on page 1105
- “WRITEBOOLEANTRUE Method” on page 1116

WRITEBOOLEANTRUE Method

Writes a Boolean true value to the text.

Syntax

`package.WRITEBOOLEANTRUE ( );`

Arguments

`package`

specifies an instance of the JSON package.

See Also

- “Using the JSON Package” on page 161

Methods:

- “ISBOOLEANTRUE Method” on page 1106
- “WRITEBOOLEANFALSE Method” on page 1115

WRITECLOSE Method

Closes the corresponding object ( } ) or array ( ] ).

Syntax

`package.WRITECLOSE ( );`
Arguments

package

specifies an instance of the JSON package.

Details

The WRITECLOSE method closes the most recently opened container of either type that was explicitly opened with the WRITARRAYOPEN or WRITOBJOPEN method. You should call the WRITECLOSE method for containers only if you explicitly opened the container with a WRITE*OPEN method.

Example

For an example, see “WRITARRAYOPEN Method” on page 1114.

See Also

• “Using the JSON Package” on page 161

Methods:

• “WRITOBJOPEN Method” on page 1120
• “WRITARRAYOPEN Method” on page 1114

WRITEDOUBLE Method

Writes a DOUBLE value in text form.

Syntax

Form 1: package.WRITEDOUBLE (double-value);
Form 2: package.WRITEDOUBLE (double-value, width);
Form 3: package.WRITEDOUBLE (double-value, width, precision);
Form 4: package.WRITEDOUBLE (double-value, width, precision, options);

Arguments

package

specifies an instance of the JSON package.

double-value

specifies the value to be written.

width

specifies the width of the formatted value.

Default 0

precision

indicates the number of digits that appear after the radix character.
Default 15

**options** specifies a flag for formatting. *options* can be one of the following values:

**BESTFIT**
formats the value using decimal notation in the form of \([-]\)dddd.ddd or scientific notation in the form of \([-]\)d.ddde±dd. The formatting style depends on the value.

**Notes**
Valid width is 1–32 characters.

Precision is ignored.

**BESTFITBIG**
formats the value using decimal notation in the form of \([-]\)dddd.ddd or scientific notation in the form of \([-]\)d.ddde±dd. The formatting style depends on the value.

**Notes**
Valid width is 1–32 characters.

Precision is ignored.

**SASBEST**
Conforms to the BESTw. format rules. The value is formatted within the specified width. Decimal notation is produced if possible. Otherwise, scientific notation is produced in the style \([-]\)ddd.ddde\([-]\)dd

**Note**
Trailing zeros after the radix character are suppressed.

**SASEW**
Conforms to the E\(_w\). format rules. The value is formatted within the specified width. Scientific notation is always produced in the style \([-]\)ddd.ddde\(±\)dd.

**Notes**
Valid width is 7–32 characters.

Precision is ignored.

**SASEWD**
Conforms to the SAS XP Services %\(w.d\)e rules. The value is formatted within the specified width. Scientific notation is always produced in the style \([-]\)ddd.ddde\(±\)dd.

**Notes**
Valid width is 7–32 digits.

Valid precision is 0–31 digits.

**SASWD**
Conforms to the \(w.d\) format rules.

**Note**
The value is formatted within the specified width. If width is too small, it reverts to the behavior indicated by SASBEST.

**DECIMAL**
Formats the value using decimal notation in the style \([-]\)ddde.dd, using *precision* to determine the number of digits after the radix.

**Note**
The radix character does not appear if there are no digits to display after it or if *precision* is set to zero.
FRACTION
Formats the fractional part of the value in the style of \(-0.dddd\).

**Note** The digit before the radix character is always zero.

INTEGER
Formats the fractional part of the value in the style of \(-dddd\).

**Note** The fractional part of the value is ignored and no radix character is added to the result.

SNOTE
formats the value using scientific notation in the form of \(-dd.dde+dd\), using the lowercase ‘e’ to precede the exponent.

SNOTEBIG
formats the value using scientific notation in the form of \(-dd.dddE+dd\), using the uppercase ‘E’ to precede the exponent.

Default BESTFIT

Example
The following example writes DOUBLE values to an array.

```sas
data _null_;
dcl package logger lgr( 'App.tk.D2PKG.JSON' );
dcl package json j();
dcl nvarchar(256) matrixA;
dcl int rc;

method init();
rc = j.createWriter();
rc = j.writeArrayOpen();
rc = j.writeArrayOpen();
rc = j.writeDouble(1.1);
rc = j.writeDouble(1.2);
rc = j.writeClose();
rc = j.writeArrayOpen();
rc = j.writeDouble(2.1);
rc = j.writeDouble(2.2);
rc = j.writeClose();
rc = j.writeClose();
j.writerGetText(rc, matrixA);
lgr.log( 4, 'matrix A = $s', matrixA );
rc = j.destroyWriter();
end;
enddata;
run;
```

The following lines are written to the SAS log:

```
NOTE: matrix A = [[1.1,1.2],[2.1,2.2]]
```

See Also

“Using the JSON Package” on page 161
WRITENULL Method

Writes a null value to the text.

Syntax

package.WRITENULL();

Arguments

package

specifies an instance of the JSON package.

See Also

•  “Using the JSON Package” on page 161

Methods:

•  “ISNULL Method” on page 1109

WRITEOBJOPEN Method

Writes the open brace ( { ) signifying the beginning of an object.

Syntax

package.WRITEOBJOPEN();

Arguments

package

specifies an instance of the JSON package.

Details

The WRITEOBJOPEN method explicitly opens an object container, which you must explicitly close with the WRITECLOSE method.

See Also

•  “Using the JSON Package” on page 161

Methods:

•  “WRITECLOSE Method” on page 1116
WRITERGETTEXT Method

Obtains the JSON text that is produced by the writer.

Syntax

```
package.WRITERGETTEXT (rc, json-text);
```

Arguments

- `package` specifies an instance of the JSON package.
- `rc` specifies the variable to hold the return code value. Possible return code values are as follows:
  - 0  Success
  - 100  The output token argument's maximum length was not large enough and truncation occurred.
  - 101  Done. Depending on the use case, this might or might not be expected.
  - 300  End of text. Depending on the use case, this might or might not be expected.
  - 301  A status condition was returned.
- `json-text` specifies a variable that receives the JSON text.

Details

The DS2 JSON package's writer currently does not support streaming. Instead, the Write instances are gathered in memory until retrieved by calling the WRITERGETTEXT method.

Example

For an example, see “WRITEARRAYOPEN Method” on page 1114.

See Also

- “Using the JSON Package” on page 161

Methods:

- “CREATEWRITER Method” on page 1100

WRITESTRING Method

Writes a string to the JSON text.
Note: Braces in the syntax convention indicate a syntax grouping.

Syntax

```
package WRITESTRING ({{'| '}}string{{'| '}}, flags);
```

Arguments

- `package` specifies an instance of the JSON package.
- `string` specifies the string to write.

Tip: The string can be a string literal in single quotation marks, a normal identifier without quotation marks, or a delimited identifier in double quotation marks.

- `flags` specifies options for special handling of the string. The following values are possible:
  - 0 indicates no flags.
  - 16 (JSN_SkipScan) indicates that the normal scanning and JSON encoding of the input string should be skipped. This means that either the string is known to contain no invalid or JSON escape characters, or the caller has already performed JSON scanning or encoding on the string. In the latter case, only quotation marks and separators would be inserted with the given string.
    - For normal scanning, omit the flag so that normal scanning and JSON encoding can proceed.
  - 32 (JSN_TrimBlanks) causes the writer to trim trailing blanks from the string.
  - 48 causes both the writer to skip scanning and trim blanks.

Example

The following example illustrates different types of string values.

```
data _null_;  
dcl package logger lgr( 'App.tk.D2PKG.JSON' );  
dcl package json j();  
dcl nvarchar(2000) txt;  
dcl varchar(20) "a**b";  
dcl varchar(20) myStr;  
dcl int rc;  

method init();  
rc = j.createWriter();  
r = j.writeObjectOpen();  
r = j.writeString( 'aaaAAA', 0 );  
"a**b" = 'bbbBBB';  
r = j.writeString( "a**b", 0 );
```
myStr = 'cccCCC';
rc = j.writeString( myStr, 0 );
rc = j.writeClose();
j.writerGetText( rc, txt );
lgr.log( 3, 'txt = $s', txt );
rc = j.destroyWriter();
end;
enddata;
run;

See Also

“Using the JSON Package” on page 161
Dictionary

DECLARE PACKAGE Statement, Logger Package

Creates a package variable and gives you the option of creating an instance of the logger package.

**Category:** Local

**Tip:** The PACKAGE statement is not required for a logger package.

**Syntax**

```
DECLARE PACKAGE LOGGER variable ([logger-name]);
```

**Arguments**

- `variable`
  - specifies a name that can reference an instance of the logger package.

- `logger-name`
  - specifies the name of the logger that is defined in the SAS logging facility.

**Default**

SAS root logger

**Details**

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.
You use a logger package to interface with the SAS logging facility. The logger package is predefined for DS2 programs. For more information about the logging facility, see *SAS Logging: Configuration and Programming Reference*.

You declare a logger package by using the DECLARE PACKAGE statement. This associates a logger package with a logger name. After you declare the new logger package, you can send messages to the logger at a specified logging level.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a logger package.

- Use the DECLARE PACKAGE statement along with the NEW operator:
  ```
  declare package logger logpkg;
  logpkg = _new_ logger();
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:
  ```
  declare package logger logpkg();
  ```

For more information about the logger package, see “Using the Logger Package” on page 164.

**Example**

This example creates an instance of a logger package.

```
data _null_;  
dcl package logger l();  
method init();  
  l.log('i', 'Hello World!');  
end;  
enddata;
```

**See Also**

- “Using the Logger Package” on page 164
- “Package Constructors and Destructors” on page 142

**Operators:**

- “_NEW_ Operator, Logger Package” on page 1130

**DELETE Method, Logger Package**

Deletes a logger package.

**Note:** The DELETE method is not required. When a logger package goes out of scope, the package is deleted.
Syntax

```pascal
package.DELETE( );
```

**Arguments**

`package`

specifies an instance of the logger package variable.

**Details**

When you no longer need the logger package, delete it by using the DELETE method. If you attempt to use a logger package after you delete it, an error will be written to the log.

**See Also**

“Package Constructors and Destructors” on page 142

---

### ISLEVELACTIVE Method

Returns a value that indicates whether the logger package that is associated with the logger name suppresses a message at the logger level.

**Syntax**

```pascal
package.ISLEVELACTIVE(['level']);
```

**Arguments**

`package`

specifies an instance of the logger package variable.

`['level']`

a numeric value that specifies the level at which a logging request is applied for the specified logger package.

**Requirements**

`level` must be a string that contains either the severity value or a one-character abbreviation for the severity value. Valid values are listed as follows.

- 2 or 'T' for TRACE
- 3 or 'D' for DEBUG
- 4 or 'I' for INFO
- 5 or 'W' for WARN
- 6 or 'E' for ERROR
- 7 or 'F' for FATAL

If a character is used for `level`, the character must be enclosed in single quotation marks. Numeric values can be quoted but do not need to be.

**Example**

These are examples of the ISLEVELACTIVE method.
LOG Method, Logger Package

Send the specified message to the logger at the specified level.

**Note:**

**Syntax**

Form 1:  
```
package.LOG(['level'], 'raw-message');
```

Form 2:  
```
package.LOG(['level'], [message-format] 'argument-1' [..., argument-9]);
```

**Arguments**

`package`

specifies an instance of the logger package variable.

`['level']`

a numeric value that specifies the level at which a logging request is applied for the specified logger package.

**Requirements**  
`level` must be a string that contains either the severity value or a one-character abbreviation for the severity value. Valid values are listed as follows. Note that any part of the word that indicates the level is valid. For example, `i`, `in`, `inf`, or `info` is valid.

- 2 or `'T'` for TRACE
- 3 or `'D'` for DEBUG
- 4 or `'I'` for INFO
- 5 or `'W'` for WARN
- 6 or `'E'` for ERROR
- 7 or `'F'` for FATAL

If a character is used for `level`, the character must be enclosed in single quotation marks. Numeric values can be quoted but do not need to be.

`'raw-message'`

specifies the message to write at the level.

**Range**  
1–65535 characters

**Requirement**  
The message is a string and must be enclosed in single quotation marks.

See Also

- *SAS Logging: Configuration and Programming Reference*
- “Using the Logger Package” on page 164
Tip
The message can be any character type expression. Here is an example.
\texttt{x.log(5, 'Error while processing function' || trimn(FNAME))};. However, using a character expression causes a conversion from a CHAR data type to an NCHAR data type. It is faster to use a character string.

$message-format$

specifies a message format that is used to produce the log message. The message format contains at least one $s$ format marker.

Requirement
The number of $s$ format markers must be less than or equal to the number of arguments. Otherwise, an error occurs.

Interaction
If the LOG method has more than two parameters and the level is valid, each $s$ format marker is replaced by the content of the corresponding argument.

Tip
To display a dollar sign ($$) in your message, use $$ in the $s$ format marker.

$argument$

specifies a value that replaces the $s$ format marker in the $message-format$.

Interaction
If the LOG method has more than two parameters and the level is valid, each $s$ format marker in the $message-format$ is replaced by the content of the corresponding argument.

Tip
Extra arguments are ignored when using formatted output.

Details

Unformatted Messages (Form 1)
A LOG method call with exactly two parameters, an active logger, and a valid level, sends the specified message to the associated logger in its raw format.

Formatted Messages (Form 2)
A LOG method call with more than two parameters, an active logger, and a valid level, sends a formatted message to the associated logger. Each $s$ format marker in the $message-format$ is replaced by the content of the corresponding argument.

Examples

Example 1: Unformatted Messages
These are examples of unformatted messages.

\texttt{mylog.log('T', 'The output was written to the log');}

\texttt{testlog.log(7, 'The output could not be written');}

Example 2: Formatted Messages
The following example shows several combinations of $s$ format markers.

\texttt{data _NULL_;}
\texttt{  dcl package logger root();}
method init();
/* $$ is not evaluated here - one parameter */
root.log(n'note', 'one-parm dollar pair: $$');
/* $s is not evaluated here - one parameter */
root.log(n'note', 'one-parm dollar-s: $s');
/* $$ is evaluated here */
root.log(n'note', 'two-parm dollar pair: $$', n'');
/* $$ is evaluated here and "mine" is substituted for $s */
root.log(n'note', 'dollar pair: $$; me:$s', n'mine');
/* "mine" is substituted for the first $s */
/* "thine" is substituted for the second $s */
root.log(n'note', 'me:$s', n'mine', n'thine');
/* there are five arguments */
root.log(n'note', 'one:$s two:$s three:$s four:$s five:$s', 1N, 2, 3.0, n'4', '5');
end;
enddata;
run;
quit;

The following lines are written to the SAS log.

| NOTE: hello world |
| NOTE: one-parm dollar pair: $$ |
| NOTE: one-parm dollar-s: $s |
| NOTE: two-parm dollar pair: $ |
| NOTE: dollar pair: $; me:mine |
| NOTE: me:mine |
| NOTE: me:mine thee:thine |
| NOTE: one:1 two:2 three:3 four:4 five:5 |

See Also

- SAS Logging: Configuration and Programming Reference
- “Using the Logger Package” on page 164

__NEW__ Operator, Logger Package

Constructs an instance of a logger package.

**Note:** The escape character (\) before the bracket indicates that the bracket is required in the syntax.

**Syntax**

```plaintext
package-variable = __NEW__ ([THIS] | [package-instance]) LOGGER([logger-name]);
```

**Arguments**

- `package-variable` specifies a name that can reference an instance of the package.
- `[THIS]` specifies that the package instance has global scope.
See “Packages and Scope” on page 137

```bash
[package-instance]
```
specifies that the new package instance has the same scope as package-instance. package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

```bash
logger-name
```
specifies the name of the logger that is defined in the SAS logging facility.

Default SAS root logger

Details
A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a logger package is declared, the variable representing the package can be considered an instance of the package. This means that two different package variables represent two completely separate copies of a package.

You declare a logger package using the DECLARE PACKAGE statement. After you declare the new logger package, use the _NEW_ operator to instantiate the package.

```bash
declare package logger mylog;
mylog = _new_ logger( );
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a logger package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```bash
declare package logger mylog( );
```

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also
- “Using the Logger Package” on page 164
- “Package Constructors and Destructors” on page 142

Statements:
- “DECLARE PACKAGE Statement, Logger Package” on page 1125
Chapter 35
DS2 Matrix Package Methods, Operators, and Statements

Dictionary

ABS Method ......................................................... 1134
ADD Method, Matrix Package .......................... 1134
ALL_AND Method ........................................ 1137
ALL_EQ Method .......................................... 1138
ALL_GE Method .................................... 1139
ALL_GT Method ..................................... 1140
ALL_LE Method .................................... 1141
ALL_LT Method ..................................... 1142
ALL_NE Method .................................... 1143
ALL_OR Method ................................... 1144
AND Method ...................................... 1145
ANY_AND Method .................................. 1145
ANY_EQ Method .................................. 1146
ANY_GE Method .................................. 1147
ANY_GT Method .................................. 1148
ANY_LE Method .................................. 1149
ANY_LT Method .................................. 1150
ANY_NE Method .................................. 1151
ANY_OR Method .................................. 1152
COLS Method ..................................... 1153
COPY Method ...................................... 1154
DECLARE PACKAGE Statement, Matrix Package ........ 1155
DELETE Method, Matrix Package ..................... 1156
DET Method ...................................... 1157
EDIV Method ...................................... 1158
EMAX Method ..................................... 1159
EMIN Method ..................................... 1161
EMOD Method ..................................... 1162
EMULT Method ..................................... 1164
EPOW Method ..................................... 1165
EQ Method ....................................... 1167
EXP Method ....................................... 1168
FLOOR Method ..................................... 1169
GE Method ....................................... 1170
GT Method ....................................... 1171
IN Method ....................................... 1172
INVERSE Method ................................ 1175
LE Method ....................................... 1176
LOG Method, Matrix Package ......................... 1178
LT Method ....................................... 1179
Dictionary

**ABS Method**

Returns a matrix that contains the absolute value of each value in the input matrix.

**Syntax**

\[ r = \text{package}.\text{ABS}(); \]

**Arguments**

\( r \)

specifies a matrix that contains the absolute value of each value in the input matrix.

\( \text{package} \)

specifies an instance of the matrix package variable.

**See Also**

“Matrix Operations” on page 169

**ADD Method, Matrix Package**

Adds one matrix to another.

**Syntax**

\[ r = \text{package-1}.\text{ADD}(\text{package-2}); \]

**Arguments**

\( r \)

specifies the matrix that is automatically created by the ADD method.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the addition operation.
package-2

specifies an instance of the second matrix package variable that is used in the addition operation.

Details

The matrix dimensions for the ADD method must be the same size in order for the matrix addition to take place. Each [i, j] element in the first matrix is added to its corresponding [i, j] element in the second matrix.

If you add matrices that have missing values, you do not receive an error.

It is also possible to perform scalar addition by using a 1x1 matrix.

Examples

Example 1: Adding Two Matrices
The following is an example of using the ADD method to add two 3x3 matrices. Each [i, j] element in the first matrix is added to its corresponding [i, j] element in the second matrix.

data _null_;  
dcl double a[3,3];  
dcl double b[3,3];  
dcl double c[3,3];  
method run();  
dcl package matrix m;  
dcl package matrix m2;  
dcl package matrix r;  
dcl double i j;  
a := (1,2,3,4,5,6,7,8,9);  
b := (1,5,9,2,6,10,3,7,1);  
m = _new_ matrix(a, 3, 3);  
m2 = _new_ matrix(b, 3, 3);  
r = m.add(m2);  
r.toarray(c);  
do i = 1 to 3;  
do j = 1 to 3;  
   put c[i,j];  
end;  
end;  
enddata;  
run;

The following lines are written to the SAS log.
Example 2: Scalar Addition of Two Matrices
The following is an example of how to perform scalar addition by using a 1x1 matrix.

```plaintext
data _null_
   ;
ccl double c[3,3];
dcl double d[1,1];
dcl double f[3,3];
method run()
   ;
dcl package matrix m3;
dcl package matrix m4;
dcl package matrix r;
dcl double i j;

c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);
d := (1);

m3 = _new_ matrix(c, 3, 3);
m4 = _new_ matrix(d, 1, 1);

r = m3.add(m4);
r.toarray(f);

do i = 1 to 3;
do j = 1 to 3;
   put f[i,j];
end;
end;
enddata;
run;
```

In this example, 1 was added to each entry in matrix `m3`. The scalar addition produces a 3x3 matrix that has the following values:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-1.2</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>-2.3</td>
<td>5.4</td>
<td>6.5</td>
<td></td>
</tr>
</tbody>
</table>

See Also
- “Matrix Operations” on page 169

Methods:
- “SUB Method” on page 1191
ALL_AND Method

Produces a scalar result in an ALL_AND comparison between all elements in one matrix and all elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL_AND}(\text{package-2}); \]

Arguments

\( x \)

specifies the scalar result.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the ALL_AND comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the ALL_AND comparison.

Details

The ALL_AND logical operation produces a result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. If all of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can also use a scalar comparison.

Comparisons

The ANY_AND operation is similar to the ALL_AND operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = \text{m1.all_and(m2)}; \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_OR Method” on page 1144
- “AND Method” on page 1145
- “ANY_AND Method” on page 1145
ALL_EQ Method

Produces a scalar result in an ALL_EQ (ALL equal-to) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1} . \text{ALL_EQ}(\text{package-2}); \]

Arguments

\( x \)

specifies the scalar result.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the ALL equal-to comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the ALL equal-to comparison.

Details

The ALL_EQ relational operation produces a scalar result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. All of the \([i, j]\) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_EQ operation is similar to the ALL_EQ operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = \text{m1} . \text{all_eq(m2)}; \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_NE Method” on page 1143
- “ANY_EQ Method” on page 1146
ALL_GE Method

Produces a scalar result in an ALL_GE (ALL greater-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL}_{\text{GE}}(\text{package-2}); \]

Arguments

\( x \)

specifies the scalar result.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the ALL greater-than-or-equal-to comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the ALL greater-than-or-equal-to comparison.

Details

The ALL_GE relational operation produces a scalar result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. All of the \([i, j]\) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_GE operation is similar to the ALL_GE operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = m1.\text{all}_{\text{ge}}(m2); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_LE Method” on page 1141
- “ALL_NE Method” on page 1143
- “ANY_GE Method” on page 1147
ALL_GT Method

Produces a scalar result in an ALL_GT (ALL greater-than) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL_GT}(\text{package-2}); \]

Arguments

\( x \)

specifies the scalar result.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the ALL greater-than comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the ALL greater-than comparison.

Details

The ALL_GT relational operation produces a scalar result that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The scalar result is 0 or 1. All of the \([i, j]\) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_GT operation is similar to the ALL_GT operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = m1.\text{all_gt}(m2); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_GE Method” on page 1139
- “ALL_LT Method” on page 1142
- “ANY_GT Method” on page 1148
ALL_LE Method

Produces a scalar result in an ALL_LE (ALL less-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL_LE}(\text{package-2}); \]

Arguments

- **x**: specifies the scalar result.
- **package-1**: specifies an instance of the first matrix package variable that is used in the ALL less-than comparison.
- **package-2**: specifies an instance of the second matrix package variable that is used in the ALL less-than comparison.

Details

The ALL_LE relational operation produces a scalar result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. All of the \([i, j]\) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_LE operation is similar to the ALL_LE operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = \text{m1.all_le(m2)}; \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_GE Method” on page 1139
- “ALL_LT Method” on page 1142
- “ANY_LE Method” on page 1149
ALL_LT Method

Produces a scalar result in an ALL_LT (ALL less-than) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL_LT}(\text{package-2}); \]

Arguments

- **x** specifies the scalar result.
- **package-1** specifies an instance of the first matrix package variable that is used in the ALL less-than comparison.
- **package-2** specifies an instance of the second matrix package variable that is used in the ALL less-than comparison.

Details

The ALL_LT relational operation produces a scalar result that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The scalar result is 0 or 1. All of the \([i, j]\) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_LT operation is similar to the ALL_LT operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = \text{m1}.\text{all_lt}(\text{m2}); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_GT Method” on page 1140
- “ALL_LE Method” on page 1141
- “ANY_LT Method” on page 1150
ALL_NE Method

Produces a scalar result in an ALL_NE (ALL not-equal-to) comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL\_NE}(\text{package-2}); \]

Arguments

- \( x \) specifies the scalar result.
- \( \text{package-1} \) specifies an instance of the first matrix package variable that is used in the ALL not-equal-to comparison.
- \( \text{package-2} \) specifies an instance of the second matrix package variable that is used in the ALL not-equal-to comparison.

Details

The ALL_NE relational operation produces a scalar result that indicates whether an \( [i, j] \)th element of the first matrix satisfies the comparison with the \( [i, j] \)th element of the second matrix. The scalar result is 0 or 1. All of the \( [i, j] \) element comparisons must be true for the result to be 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_NE operation is similar to the ALL_NE operation except that if any of the logical \( [i, j] \) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = m1.\text{all\_ne}(m2); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_EQ Method” on page 1138
- “ANY_NE Method” on page 1151
ALL_OR Method

Produces a scalar result in an ALL_OR comparison between elements in one matrix and elements in another matrix.

Syntax

\[ x = \text{package-1}.\text{ALL_OR}(	ext{package-2}); \]

Arguments

\( x \)

specifies the scalar result.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the ALL OR comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the ALL OR comparison.

Details

The ALL_OR logical operation produces a result that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The scalar result is 0 or 1. If all of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_OR operation is similar to the ALL_OR operation except that if any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

Example

\[ x = \text{m1.all_or(m2)}; \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_AND Method” on page 1137
- “ANY_OR Method” on page 1152
- “OR Method” on page 1187
AND Method

Compares two matrices based on the AND logical operation, and returns the resulting matrix.

Syntax

\[ r = \text{package-1}.\text{AND}(\text{package-2}); \]

Arguments

- \( r \): specifies a matrix that contains the results of an AND comparison between the values of two matrices.
- \( \text{package-1} \): specifies an instance of the first matrix package variable that is used in the AND comparison.
- \( \text{package-2} \): specifies an instance of the second matrix package variable that is used in the AND comparison.

Details

The AND logical operator behaves similarly to the binary relational operations (LT, LE, GE, GT, NE, and EQ). In each case, the AND logical operation will be applied to the \([i, j]\)th elements of two matrices and placed in the result matrix \( r \).

The matrix sizes must match or you can use a scalar comparison.

Example

\[ r = \text{m1}.\text{AND}(\text{m2}); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_AND Method” on page 1137
- “ANY_AND Method” on page 1145
- “OR Method” on page 1187

ANY_AND Method

Produces a scalar result in an ANY_AND comparison between elements in one matrix and elements in another matrix.
Syntax

\[ x = \text{package-1}.\text{ANY\_AND}(\text{package-2}); \]

**Arguments**

- \( x \)
  - specifies the scalar result.
- \( \text{package-1} \)
  - specifies an instance of the first matrix package variable that is used in the ANY AND comparison.
- \( \text{package-2} \)
  - specifies an instance of the second matrix package variable that is used in the ANY AND comparison.

**Details**

The ANY_AND logical operation produces a scalar result that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The scalar result is 0 or 1. If any of the logical \([i, j]\) operations is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Comparisons**

The ALL_AND operation is similar to the ANY_AND operation except that all of the logical \([i, j]\) operations have to be true for the result to be 1. Otherwise, the result is 0.

**Example**

\[ x = m1.\text{any\_and}(m2); \]

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “ALL\_OR Method” on page 1144
- “AND Method” on page 1145
- “ANY\_OR Method” on page 1152

---

**ANY\_EQ Method**

Produces a scalar result in an ANY_EQ (ANY equal-to) comparison between elements in one matrix and elements in another matrix.

**Syntax**

\[ x = \text{package-1}.\text{ANY\_EQ}(\text{package-2}); \]
**Arguments**

\( x \)

specifies the scalar result.

**package-1**

specifies an instance of the first matrix package variable that is used in the ANY equal-to comparison.

**package-2**

specifies an instance of the second matrix package variable that is used in the ANY equal-to comparison.

**Details**

The ANY_EQ relational operation produces a scalar result that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The scalar result is 0 or 1. If any of the \([i, j]\) element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Comparisons**

The ALL_EQ operation is similar to the ANY_EQ operation except that all of the logical \([i, j]\) operations have to be true for the result to be 1. Otherwise, the result is 0.

**Example**

\[ x = m1.\text{any}_{-}\text{eq}(m2); \]

**See Also**

- “Matrix Operations” on page 169
- “ALL_EQ Method” on page 1138
- “ANY_NE Method” on page 1151
- “EQ Method” on page 1167

---

**ANY_GE Method**

 Produces a scalar result in an ANY_GE (ANY greater-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

**Syntax**

\[ x = \text{package-1}.\text{ANY}_{-}\text{GE} (\text{package-2}); \]

**Arguments**

\( x \)

specifies the scalar result.
package-1
  specifies an instance of the first matrix package variable that is used in the ANY greater-than-or-equal-to comparison.

package-2
  specifies an instance of the second matrix package variable that is used in the ANY greater-than-or-equal-to comparison.

Details
The ANY_GE relational operation produces a scalar result that indicates whether an
\([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the
second matrix. The scalar result is 0 or 1. If any of the \([i, j]\) element comparisons is true,
then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons
The ALL_GE operation is similar to the ANY_GE operation except that all of the
logical \([i, j]\) operations have to be true for the result to be 1. Otherwise, the result is 0.

Example
  \(x = m1.\text{any_ge}(m2)\);

See Also
  • “Matrix Operations” on page 169

Methods:
  • “ALL_GE Method” on page 1139
  • “ANY_LE Method” on page 1149
  • “ANY_NE Method” on page 1151

ANY_GT Method
Produces a scalar result in an ANY_GT (ANY greater-than) comparison between elements in one matrix
and elements in another matrix.

Syntax
  \(x = \text{package-1.ANY_GT(package-2)};\)

Arguments
  \(x\)
    specifies the scalar result.

  \(\text{package-1}\)
    specifies an instance of the first matrix package variable that is used in the ANY
greater-than comparison.
package-2 specifies an instance of the second matrix package variable that is used in the ANY greater-than comparison.

Details
The ANY_GT relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons
The ALL_GT operation is similar to the ANY_GT operation except that all of the logical [i, j] operations has to be true for the result to be 1. Otherwise, the result is 0.

Example
```plaintext
x=m1.any_gt(m2);
```

See Also
• “Matrix Operations” on page 169

Methods:
• “ALL_GT Method” on page 1140
• “ANY_GE Method” on page 1147
• “ANY_LT Method” on page 1150

ANY_LE Method
Produces a scalar result in an ANY_LE (any less-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Syntax
```plaintext
x=package-1.ANY_LE(package-2);
```

Arguments
x specifies the scalar result.

package-1 specifies an instance of the first matrix package variable that is used in the ANY less-than comparison.

package-2 specifies an instance of the second matrix package variable that is used in the ANY less-than comparison.
Details
The ANY_LE relational operation produces a scalar result that indicates whether an 
\[i, j\]th element of the first matrix satisfies the comparison with the \[i, j\]th element of the 
second matrix. The scalar result is 0 or 1. If any of the \[i, j\] element comparisons is true, 
then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons
The ALL_LE operation is similar to the ANY_LE operation except that all of the logical 
\[i, j\] operations has to be true for the result to be 1. Otherwise, the result is 0.

Example
\[x=m1.any_le(m2);\]

See Also
• “Matrix Operations” on page 169

Methods:
• “ALL_LE Method” on page 1141
• “ANY_GE Method” on page 1147
• “ANY_LT Method” on page 1150

ANY_LT Method
Produces a scalar result in an ANY_LT (ANY less-than) comparison between elements in one matrix and 
elements in another matrix.

Syntax
\[x=package-1.ANY_LT(package-2);\]

Arguments
\[x\]
  specifies the scalar result.

\[package-1\]
  specifies an instance of the first matrix package variable that is used in the ANY 
  less-than comparison.

\[package-2\]
  specifies an instance of the second matrix package variable that is used in the ANY 
  less-than comparison.

Details
The ANY_LT relational operation produces a scalar result that indicates whether an 
\[i, j\]th element of the first matrix satisfies the comparison with the \[i, j\]th element of the
second matrix. The scalar result is 0 or 1. If any of the \([i, j]\) element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Comparisons**

The ALL_LT operation is similar to the ANY_LT operation except that all of the logical \([i, j]\) operations has to be true for the result to be 1. Otherwise, the result is 0.

**Example**

The following example produces a result of 1 because the \([1, 2]\) element of matrix \(m\) (= 2) is less than the \([1, 2]\) element of matrix \(m2\) (= 5). All of the other elements do not satisfy the comparison. If the \([1, 2]\) element of matrix \(m\) was changed to 6, for example, the result would be 0.

```plaintext
data _null_;  
dcl double a[3,3];  
dcl double b[3,3];  
dcl double r;  
method run();  
dcl package matrix m;  
dcl package matrix m2;  
  a := (1,2,10,4,7,11,15,9,12);  
  b := (1,5,9,2,6,10,3,7,11);  
  m = _new_matrix(a, 3, 3);  
  m2 = _new_matrix(b, 3, 3);  
  r = m.any_lt(m2);  
  put r=;  
end;  
enddata;  
run;
```

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “ALL_LT Method” on page 1142
- “ANY_GT Method” on page 1148
- “ANY_LE Method” on page 1149

---

**ANY_NE Method**

 Produces a scalar result in an ANY_NE (ANY not-equal-to) comparison between elements in one matrix and elements in another matrix.
Syntax

\[ x = \text{package-1}.\text{ANY\_NE}(\text{package-2}); \]

**Arguments**

\[ x \]

specifies the scalar result.

\[ \text{package-1} \]

specifies an instance of the first matrix package variable that is used in the ANY not-equal-to comparison.

\[ \text{package-2} \]

specifies an instance of the second matrix package variable that is used in the ANY not-equal-to comparison.

**Details**

The ANY\_NE relational operation produces a scalar result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. If any of the \([i, j]\) element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Comparisons**

The ALL\_NE operation is similar to the ANY\_NE operation except that all of the logical \([i, j]\) operations have to be true for the result to be 1. Otherwise, the result is 0.

**Example**

\[ x = \text{m1}.\text{any\_ne}(\text{m2}); \]

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “ALL\_NE Method” on page 1143
- “ANY\_EQ Method” on page 1146

**ANY\_OR Method**

Produces a scalar result in an ANY\_OR comparison between elements in one matrix and elements in another matrix.

**Syntax**

\[ x = \text{package-1}.\text{ANY\_OR}(\text{package-2}); \]
Arguments

\( x \)

specifies the scalar result.


package-1

specifies an instance of the first matrix package variable that is used in the ANY OR comparison.


package-2

specifies an instance of the second matrix package variable that is used in the ANY OR comparison.

Details

The ANY OR logical operation produces a scalar result that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The scalar result is 0 or 1. If any of the \([i, j]\) element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL OR operation is similar to the ANY OR operation except that all of the logical \([i, j]\) operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

\[ x = m1.\text{any}\_\text{or}(m2); \]

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL OR Method” on page 1144
- “ANY AND Method” on page 1145
- “OR Method” on page 1187

COLS Method

Returns the number of columns in the specified matrix.

Syntax

\[ \text{variable-name}=\text{package.COLS}(); \]

Arguments


variable-name

specifies the name of a variable that contains the number of columns after the method is complete.
package specifies an instance of the matrix package variable.

**Example**
See the example in the ROWS method on page 1190.

**See Also**
- “Using the MATRIX Package” on page 167

**Methods:**
- “ROWS Method” on page 1190

### COPY Method
Copies one matrix to another.

**Syntax**

\[ r = \text{package.COPY}(); \]

**Arguments**

\[ r \]

specifies the matrix that is automatically created by the COPY method.

\[ \text{package} \]

specifies an instance of the matrix package variable.

**Details**

The COPY method copies a matrix into a new matrix.

**Example**

This example creates a new copy of a 3x4 matrix.

```plaintext
data _null_
method run()
    dcl package matrix m;
    dcl package matrix r;
    dcl double i j;
    a := (1,2,3,4,5,6,7,8,9,10,11,12);
    m = _new_ matrix(a, 3, 4);
    r = m.copy();
    r.toarray(b);
```

1154 Chapter 35 • DS2 Matrix Package Methods, Operators, and Statements
do i = 1 to 3;
   do j = 1 to 4;
      put b[i,j];
   end;
end;
enddata;
run;

The following lines are written to the SAS log.

1
2
3
4
5
6
7
8
9
10
11
12

See Also

“Using the MATRIX Package” on page 167

DECLARE PACKAGE Statement, Matrix Package

Creates an instance of a MATRIX package.

Category: Local

Syntax

DECLARE PACKAGE MATRIX variable ([row-dimension, column-dimension]);

Arguments

variable
   specifies a name that can reference an instance of the matrix package.

row-dimension
   specifies the number of rows in the matrix instance.

column-dimension
   specifies the number of columns in the matrix instance.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

The matrix package provides a DS2-level implementation of SAS/IML functionality. The matrix package is predefined for DS2 programs.
When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

A matrix package is created by declaring and instantiating the package using the DECLARE PACKAGE statement.

This example creates an empty 2x2 matrix, and stores the instance in the variable \( m \).

```plaintext
declare package matrix m(2, 2);
```

A matrix must be initialized before it can be used, and initialization is done in the code stream, not in the declarations. You can use the following actions to load data into a matrix instance.

- \_NEW\_ operator to load an array
- \IN\ method to load an array
- \SET\ statement to load external data

**Note:** Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

**See Also**

- “Declaring and Instantiating a MATRIX Package” on page 167
- “Package Constructors and Destructors” on page 142

**Methods:**

- “\IN\ Method” on page 1172

**Operators:**

- \_NEW\_ Operator, Matrix Package” on page 1181

**Statements:**

- “SET Statement” on page 937

### DELETE Method, Matrix Package

Deletes a matrix package.

**Note:** The DELETE method is not required. When a matrix package goes out of scope, the package is deleted.

**Syntax**

```plaintext
package.DELETE();
```
**Arguments**

*package*

- specifies an instance of the matrix package variable.

**Details**

When you no longer need the matrix package, delete it by using the DELETE method. If you attempt to use a matrix package after you delete it, an error will be written to the log.

**See Also**

“Package Constructors and Destructors” on page 142

---

**DET Method**

Computes the determinant of a square matrix.

**Syntax**

\[ d = \text{package}.\text{DET}(); \]

**Arguments**

*d*

- specifies the matrix that is automatically created by the DET method.

*package*

- specifies an instance of the matrix package variable.

**Details**

The input matrix for a determinant must be square. Otherwise, you receive a run-time error. The output from the DET method is a real or complex number that is called the determinant.

**Example**

The following example computes a determinant for a 3x3 matrix.

```plaintext
data _null_;  
dcl double a[3,3];

method run();  
dcl package matrix m;  
dcl double d;

    a := (1,3,2,5,4,6,9,8,9);  
m = _new_ matrix(a, 3, 3);  
d = m.det();  
put d=;  
end;
enddata;
run;
```
The following line is written to the SAS log.

\texttt{d=23}

**See Also**

“Matrix Operations” on page 169

---

**EDIV Method**

Performs an element-wise scalar division.

**Syntax**

\[ x = \text{package-1}.\text{EDIV}(\text{package-2}); \]

**Arguments**

\[ x \]

- specifies the matrix that is automatically created by the EDIV method.

\[ \text{package-1} \]

- specifies an instance of the first matrix package variable that is used in the element-wise division operation.

\[ \text{package-2} \]

- specifies an instance of the second matrix package variable that is used in the element-wise division operation.

**Details**

The EDIV method enables you to apply the element-wise scalar division of one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each \([i,j]\) element

The EDIV method produces a result matrix from the element-by-element operations on the two argument matrices.

**Example**

\[ x = \text{m1}.\text{ediv}(\text{m2}); \]

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “EMULT Method” on page 1164
EMAX Method
Performs an elementwise comparison of two matrices and returns the largest elements.

Syntax

\[ x = \text{package-1}.\text{EMAX}(\text{package-2}); \]

Arguments

- **x** specifies the matrix that is automatically created by the EMAX method.
- **package-1** specifies an instance of the first matrix package variable that is used in the elementwise maximum operation.
- **package-2** specifies an instance of the second matrix package variable that is used in the elementwise maximum operation.

Details

The EMAX method enables you to apply an elementwise maximum value comparison to one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each \([i,j]\) element

The EMAX method produces a result matrix from the element-by-element operations on the two argument matrices.

Examples

**Example 1: Comparing Maximum Values Using a 1x1 Matrix**
The following example creates a matrix that contains the maximum value from two 2x2 matrices.

```plaintext
data _null_
method run()
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   data _null_
   method run()
      dcl package matrix m;
      dcl package matrix m1;
      dcl package matrix r;
      dcl double i j;
```
a := (2, 2, 3, 4);
b := (4, 5, 1, 0);

m = _new_ matrix(a, 2, 2);
ml = _new_ matrix(b, 2, 2);

r = m.emax(ml);
r.toarray(f);

do i = 1 to 2;
do j = 1 to 2;
put f[i,j];
end;
end;
enddata;
run;

The resulting matrix has the following values.
4  5
3  4

Example 2: Vector Operation on a Matrix
In this example, the maximum operator is applied to all the rows of matrix m by using the matrix ml as a row.

data _null_;  
method init();
dcl package matrix m;
dcl package matrix ml;
dcl package matrix r;
dcl double i j;
dcl double a[4];
dcl double b[2];
dcl double f[2,2];

a := (2, 2, 3, 4);
b := (1, 5);
m = _new_ matrix(a, 2, 2);
ml = _new_ matrix(b, 1, 2);
r = m.emax(ml);
r.toarray(f);
do i = 1 to 2;
do j = 1 to 2;
put f[i,j];
end;
end;
enddata;
run;

The resulting matrix has the following values.
2 5
3 5
See Also

- “Matrix Operations” on page 169

Methods:

- “EMIN Method” on page 1161

EMIN Method

Performs an elementwise comparison of two matrices and returns the smallest elements.

Syntax

\[ x = \text{package-1.EMIN}(\text{package-2}); \]

Arguments

- \( x \) specifies the matrix that is automatically created by the EMIN method.
- \( \text{package-1} \) specifies an instance of the first matrix package variable that is used in the elementwise minimum operation.
- \( \text{package-2} \) specifies an instance of the second matrix package variable that is used in the elementwise minimum operation.

Details

The EMIN method enables you to apply an elementwise minimum value comparison of one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each \([i,j]\) element

The EMIN method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example creates a matrix that contains the maximum value from two 2x2 matrices.

```plaintext
data _null_
  dcl double a[2,2];
  dcl double b[2,2];
  dcl double f[2,2];

method run();
```
dcl package matrix m;
dcl package matrix m1;
dcl package matrix r;
dcl double i j;

a := (2, 2, 3, 4);
b := (4, 5, 1, 0);

m = _new_ matrix(a, 2, 2);
m1 = _new_ matrix(b, 2, 2);

r = m.emin(m1);
r.toarray(f);

do i = 1 to 2;
do j = 1 to 2;
   put f[i, j];
end;
end;
enddata;
run;

The resulting matrix has the following values.

2  2
1  0

See Also

• “Matrix Operations” on page 169

Methods:

• “EMAX Method” on page 1159

EMOD Method

Returns the remainder of the division of elements of the first matrix by elements of the second matrix in an
elementwise scalar operation.

Syntax

\[ x = \text{package-1.EMOD(package-2)}; \]

Arguments

\( x \)

specifies the matrix that is automatically created by the EMOD method.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the
elementwise MOD comparison.
specifies an instance of the second matrix package variable that is used in the elementwise MOD comparison.

Details

The EMOD elementwise operation enables you to find the remainder of a division operation on one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMOD method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example divides the elements in matrix, \( m \), by the elements in matrix, \( m2 \). The EMOD method is used to return the matrix of remainders, \( f \).

```plaintext
data _null_;  
dcl double a[2,2];  
dcl double b[2,2];  
dcl double f[2,2];  

method run();  
dcl package matrix m;  
dcl package matrix m1;  
   dcl package matrix r;  
   dcl double i j;  

   a := (125, 17, 39, 40);  
   b := (40, 5, 12, 8);  

   m = _new_ matrix(a, 2, 2);  
   m1 = _new_ matrix(b, 2, 2);  
   
   r = m.emod(m1);  
   r.toarray(f);  

   do i = 1 to 2;  
      do j = 1 to 2;  
         put f[i,j];  
      end;  
   end;  
end;  
enddata;  
run;  
```

The resulting matrix has the following values.

\[
\begin{bmatrix} 5 & 2 \\ 3 & 0 \end{bmatrix}
\]
EMULT Method

Performs an elementwise scalar multiplication.

Syntax

\[ r = \text{package-1.EMULT(package-2)}; \]

Arguments

- \( r \) specifies the matrix that is automatically created by the EMULT method.
- \( \text{package-1} \) specifies an instance of the first matrix package variable that is used in the elementwise multiplication operation.
- \( \text{package-2} \) specifies an instance of the second matrix package variable that is used in the elementwise multiplication operation.

Details

The EMULT method enables you to apply the elementwise scalar multiplication on one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMULT method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example shows how to perform an elementwise scalar multiplication. Each element of matrix \( c \) is multiplied by 2.

```plaintext
data _null_
  dcl double c[3,3]
  dcl double d[1,1]
  dcl double f[3,3]

method run();
  dcl package matrix m3
  dcl package matrix m4
  dcl package matrix r
  dcl double i j
```

See Also

“Matrix Operations” on page 169
c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);
d := (2);

m3 = _new_ matrix(c, 3, 3);
m4 = _new_ matrix(d, 1, 1);

r = m3.emult(m4);
r.toarray(f);

do i = 1 to 3;
do j = 1 to 3;
    put f[i,j];
end;
end;
enddata;
enddata;
run;

The resulting matrix has the following values.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>2</td>
<td>-4.4</td>
<td>4.4</td>
</tr>
<tr>
<td>-6.6</td>
<td>8.8</td>
<td>11</td>
</tr>
</tbody>
</table>

See Also

• “Matrix Operations” on page 169

Methods:

• “EDIV Method” on page 1158

---

**EPOW Method**

Raises a number to a specified power in an elementwise operation.

**Syntax**

\[ x = \text{package-1}.EPOW(\text{package-2}); \]

**Arguments**

- \( x \) specifies the matrix that is automatically created by the EPOW method.

- \( \text{package-1} \) specifies an instance of the first matrix package variable that is used in the elementwise operation.

- \( \text{package-2} \) specifies an instance of the second matrix package variable that is used in the elementwise operation.
Details

The EPOW elementwise operation enables you to raise a value exponentially in one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EPOW method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example raises each element of matrix $c$ to a power of 2.

```r
data _null_;
    dcl double c[3,3];
    dcl double d[1,1];
    dcl double f[3,3];
    method run();
        dcl package matrix m3;
        dcl package matrix m4;
        dcl package matrix r;
        dcl double i j;

        c := (0, 15, 1.7, 13, -2.2, 10, -3.3, 7, 2);
        d := (2);

        m3 = _new_ matrix(c, 3, 3);
        m4 = _new_ matrix(d, 1, 1);

        r = m3.epow(m4);
        r.toarray(f);
        do i = 1 to 3;
            do j = 1 to 3;
                put f[i,j];
            end;
        end;
        end;
enddata;
run;
```

The resulting matrix has the following values.

0     225    2.89
169   4.84   100
10.89 49    4

See Also

“Matrix Operations” on page 169
EQ Method

Produces a scalar result in an equal-to comparison between elements in one matrix and elements in another matrix.

Syntax

\[ r = \text{package-1}.\text{EQ}(\text{package-2}); \]

Arguments

\( r \)

specifies a matrix that contains the results of an equal-to comparison between the values of two matrices.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the equal-to comparison.

\( \text{package-2} \)

specifies an instance of the second matrix package variable that is used in the equal-to comparison.

Details

The EQ relational operation produces a matrix that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The result is 0 or 1. If the \([i, j]\) elements are equal, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

The following example compares the elements in two matrices for equality. Note that missing values are compared.

```plaintext
data _null_
  dcl double a[2,2];
  dcl double b[2,2];
  dcl double f[2,2];
method run();
  dcl package matrix m;
  dcl package matrix m1;
  dcl package matrix r;
  dcl double i j;
  a := (2, 2, 3, .);
  b := (4, 5, 1, .);
  m = _new_ matrix(a, 2, 2);
  m1 = _new_ matrix(b, 2, 2);
```

\[
\begin{align*}
    r &= \text{m.eq(m1)}; \\
    r &= \text{toarray(f)}; \\
    \text{do i = 1 to 2;} \\
    \text{do j = 1 to 2;} \\
    \text{put f[i,j];} \\
    \text{end;} \\
    \text{end;} \\
    \text{enddata;} \\
    \text{run;} \\
\end{align*}
\]

The resulting matrix has the following values.

\[
\begin{pmatrix}
    0 & 0 \\
    0 & 1 \\
\end{pmatrix}
\]

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “GE Method” on page 1170
- “LE Method” on page 1176
- “NE Method” on page 1186

---

**EXP Method**

Returns a matrix that contains an exponential value for each value in the input matrix.

**Syntax**

\[
r = \text{package.EXP()};
\]

**Arguments**

- \(r\)
  - specifies the matrix that is automatically created by the EXP method.

- \(\text{package}\)
  - specifies an instance of matrix package variable.

**Details**

The EXP method creates a matrix that contains each element of the input matrix raised to the \(e^\text{th}\) power.

**Example**

The following example raises each element of a matrix to the \(e^\text{th}\) power.

```plaintext
data _null_;  
dcl double a[3, 3];
```

---

```plaintext
r = m.eq(m1);  
r = toarray(f);  
\text{do i = 1 to 2;}  
\text{do j = 1 to 2;}  
\text{put f[i,j];}  
\text{end;}  
\text{end;}  
\text{enddata;}  
\text{run;}  
```

The resulting matrix has the following values.

\[
\begin{pmatrix}
    0 & 0 \\
    0 & 1 \\
\end{pmatrix}
\]
dcl double c[3, 3];

method run();
   dcl package matrix m;
   dcl package matrix r;
   dcl double i j;

   a := (1, 2, 3, 1, 2, 3, 1, 2, 3);

   m=_new_ matrix(a, 3, 3);
   r=m.exp();
   r.toarray(c);

   do i=1 to 3;
      do j=1 to 3;
         put c[i, j];
      end;
   end;
enddata;
run;

The following lines are written to the SAS log.

2.71828182845904 7.38905609893065 20.0855369231876
2.71828182845904 7.38905609893065 20.0855369231876
2.71828182845904 7.38905609893065 20.0855369231876
2.71828182845904 7.38905609893065 20.0855369231876

See Also

“Matrix Operations” on page 169

FLOOR Method

Returns a matrix that contains the integer part of each value in the input matrix.

Syntax

$r = \text{matrix-package.FLOOR}( );$

Arguments

$r$
   specifies a matrix that contains the integer part of each value in the input matrix.

matrix-package
   specifies an instance of matrix package variable.
Example

The following example creates a matrix that contains the integer part of input matrix, \( m \).

data _null_;  
dcl double a[2, 2];  
dcl double c[2, 2];

method run();  
dcl package matrix m;  
dcl package matrix r;  
dcl double i j;

a := (1323.43, -.72, 3.38, 45);

m = _new_ matrix(a, 2, 2);  
r = m.floor();  
r.toarray(c);

do i=1 to 2;  
  do j=1 to 2;  
    put c[i, j];  
  end;
end;
enddata;
run;

The resulting matrix has the following values.

1323  0
3   45

See Also

“Matrix Operations” on page 169

GE Method

Produces a scalar result in a greater-than-or-equal-to comparison between elements in one matrix and elements in another matrix.

Syntax

\[ r = \text{package-1.} \text{GE} (\text{package-2}) ; \]

Arguments

\( r \)

specifies a matrix that contains the results of a greater-than-or-equal-to comparison between the values of two matrices.

\( \text{package-1} \)

specifies an instance of the first matrix package variable that is used in the greater-than-or-equal-to comparison.
**Details**

The GE relational operation produces a matrix that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The result is 0 or 1. If the \([i, j]\) element greater-than-or-equal-to comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Example**

```r=m1.ge(m2);```

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “GT Method” on page 1171
- “LE Method” on page 1176

---

### GT Method

**GT Method**

Produces a scalar result in a greater-than comparison between elements in one matrix and elements in another matrix.

**Syntax**

```r=package-1.GT(package-2);```

**Arguments**

- **r** specifies a matrix that contains the results of a greater-than comparison between the values in two matrices.
- **package-1** specifies an instance of the first matrix package variable that is used in the greater-than comparison.
- **package-2** specifies an instance of the first matrix package variable that is used in the greater-than comparison.

**Details**

The GT relational operation produces a matrix that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix.
The result is 0 or 1. If the \([i, j]\) element greater-than comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

**Example**

\[
r = m1.gt(m2);
\]

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “GE Method” on page 1170
- “LT Method” on page 1179

---

### IN Method

Loads an array into a matrix.

**Alias:** LOAD

**Syntax**

```plaintext
package.IN(array-name);
```

**Arguments**

- **package**
  specifies an instance of the matrix package variable.

- **array-name**
  specifies an array that is used in loading the matrix.

**Restriction**

The array dimensions must match the matrix dimensions. Otherwise, an error occurs.

**Tip**

You can use variable arrays.

**See**

Chapter 15, “DS2 Arrays,” on page 121

**Details**

The IN method loads an array into a matrix. This process might be useful if an array \(a\) can change as the program executes and you want to repeatedly reset the values of matrix \(m\). Here is an example:

```plaintext
dcl double a[3, 3];
dcl package matrix m;

m = _new_ matrix(3, 3);
m.in(a);
```
You can use variable arrays to load data into a matrix. Here is an example:

```plaintext
vararray double va[3,3];
   dcl package matrix m;

m.in(va);
```

You can also use variable arrays to input and output data using the SET and OUT statements.

**Examples**

**Example 1: Loading and Writing Data**

This example reads data from a data set in matrix form, finds the matrix inverse, and writes the result matrix to an output table. The example uses the IN and OUT methods. The IN method loads data from a variable array into a matrix. The OUT method writes the data in the matrix to a variable array. The SET and OUTPUT statements are used in the program to read data from an array and write the results to an output array.

The IN and OUT methods are overloaded to accept an integer argument that tells which row of the matrix to load. For the IN method, the variable array row data is read into the matrix. For the OUT method, the matrix row data is written to the variable array.

```plaintext
/* DATA step to create an array of data */
data x;
   array a[4];

   /* Create a 4x4 matrix. */
   a1 = 1; a2 = 5; a3 = 2; a4 = 3;
   output;

   a1 = 3; a2 = 3; a3 = 1; a4 = 7;
   output;

   a1 = 2; a2 = 3; a3 = 8; a4 = 9;
   output;

   a1 = 3; a2 = 6; a3 = 7; a4 = 4;
   output;
run;

proc ds2;
  data inv/overwrite=yes;

   /* global declarations */
   vararray double v[4];
   vararray double a[4];
   keep v1 v2 v3;

   dcl package matrix m;
   dcl package matrix r;
   dcl package matrix inv;
   dcl double c[4, 4];
   dcl double i j;

   /* Create an empty matrix to hold the input values. */
```
method init();
    m=_new_[this] matrix(4, 4);
    i=1;
end;

/* Read and initialize each row of the matrix from VARARRAY a. */
method run();
    set x;
    m.in(a, i);
    i + 1;
end;

method term();
    /* Find the inverse of the matrix. */
    inv=m.inverse();
    /* Check whether it gives an identity matrix. */
    r=m.mult(inv);
    /* Write each row of inverse to the output table */
    /* using VARARRAY v. */
    do i=1 to 4;
        inv.out(v, i);
        output;
    end;
    /* Print the result to see if it’s the identity. */
    r.toarray(c);
    do i=1 to 4;
        do j=1 to 4;
            put c[i, j];
        end;
    end;
enddata;
run;
quit;

The following lines are written to the SAS log.

1
-2.7755575615628E-17
-1.1102230246251E-16
0
1.1102230246251E-16
1
0
0.1.1102230246251E-16
5.5511151231257E-17
1
0
1.6653345369377E-16
1.6653345369377E-16
-1.6653345369377E-16
1

These are the key calls for the program:

/* Input */
method run();
    set x;
    m.in(a, i);
    i + 1;
end;

/* Output */
do i=1 to 4;
    inv.out(v, i);
    output;
end;

Example 2: Using the IN and OUT Methods
For another example of using the IN and OUT methods, see “Example 2: Multiply Two Matrices That Are Read from External Data” on page 1184.

See Also
- “Matrix Data Input” on page 167

Methods:
- “OUT Method” on page 1188

Statements:
- “OUTPUT Statement” on page 916
- “SET Statement” on page 937

INVERSE Method
Computes the inverse of a matrix.

Syntax

im=matrix-package.INVERSE();

Arguments

im
    specifies the matrix that is automatically created by the INVERSE method.

matrix-package
    specifies an instance of matrix package variable.

Details
It is possible to perform basic matrix operations on a single matrix. The INVERSE matrix operation computes the inverse of a matrix. If the matrix is not square or is singular (not invertible), you receive a run-time error.
Example

The following example computes the inverse of a 3x3 matrix, and checks, using matrix multiplication, whether the resulting inverse produces the identity matrix.

```sas
data _null_;  
dcl double a[3,3];  
dcl double b[3,3];

method run();  
dcl package matrix m im r;  
dcl double i j;

a := (1, 2, -1, 2, 1, 0, -1, 1, 2);  
m = _new_ matrix(a, 3, 3);

im = m.inverse();  
r = m.mult(im);  
r.toarray(b);

do i = 1 to 3;  
  do j = 1 to 3;  
    put b[i,j];  
  end;  
end;  
enddata;
run;
```

Some values of the resulting matrix are not exactly zero. The values might be very small numbers, such as 1.1102230246251E-16. These small numbers are the exact results that the SAS/IML subsystem provides.

```
  1                           5.5511151231257E-17   0
  0                           1                    0
  1.1102230246251E-16   -1.1102230246251E-16   1
```

See Also

“Matrix Operations” on page 169

---

LE Method

Produces a scalar result in a less-than-or-equal-to comparison between elements in one matrix and elements in another matrix.

**Syntax**

```
r=package-1.LE(package-2);
```

**Arguments**

- `r`
  - specifies a matrix that contains the results of a less-than-or-equal-to comparison between the values of two matrices.
package-1
specifies an instance of the first matrix package variable that is used in the less-than-or-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the less-than-or-equal-to comparison.

Details
The LE relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] element less-than-or-equal-to comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Examples

Example 1: Comparing Two Matrices Using the LE Method
The following example uses the LE method. The [i, j]th element of matrix m is compared with the [i, j] element of matrix m2, using 0 or 1 for the result entries.

data _null_;  
dcl double a[3,3];  
dcl double b[3,3];  
dcl double c[3,3];  
method run();  
dcl package matrix m;  
dcl package matrix m2;  
dcl package matrix r;  
dcl double i j;  
   a := (1,2,3,4,5,6,7,8,9);  
   b := (1,5,9,2,6,10,3,7,11);  
   m = _new_ matrix(a, 3, 3);  
   m2 = _new_ matrix(b, 3, 3);  
   r = m.le(m2);  
   r.toarray(c);  
   do i = 1 to 3;  
      do j = 1 to 3;  
         put c[i,j];  
      end;  
   end;  
end;  
enddata;  
run;  
The resulting matrix has the following values.
1 1 1
0 1 1
0 0 1
Example 2: Using a Scalar Matrix
The following example uses a scalar matrix with the LE method.

```plaintext
data _null_;  
dcl double a[3,3];  
dcl double b[1,1];  
dcl double c[3,3];  

method run();  
dcl package matrix m;  
dcl package matrix m2;  
dcl package matrix r;  
dcl double i j;  

a := (1,3,3,4,5,6,7,8,9);  
b := (4);  

m = _new_ matrix(a, 3, 3);  
m2 = _new_ matrix(b, 1, 1);  

r = m.le(m2);  
r.toarray(c);  

do i = 1 to 3;  
do j = 1 to 3;  
   put c[i,j];  
end;  
end;  
enddata;  
run;
```

The resulting matrix has the following values.

```
1 1 1
1 0 0
0 0 0
```

See Also

- “Matrix Operations” on page 169

Methods:

- “GE Method” on page 1170
- “LT Method” on page 1179

LOG Method, Matrix Package
Returns a matrix that contains the natural logarithm for each value in the input matrix.

Syntax

```
r=package.LOG();
```
Arguments

\( r \)

specifies a matrix that is automatically created by the LOG method.

package

specifies an instance of matrix package variable.

See Also

“Matrix Operations” on page 169

LT Method

Produces a scalar result in a less-than comparison between elements in one matrix and elements in another matrix.

Syntax

\[ r = \text{package-1}.\text{LT}(\text{package-2}); \]

Arguments

\( r \)

specifies a matrix that contains the results of a less-than comparison between the values in two matrices.

package-1

specifies an instance of the first matrix package variable that is used in the less-than comparison.

package-2

specifies an instance of the second matrix package variable that is used in the less-than comparison.

Details

The LT relational operation produces a matrix that indicates whether an \([i, j]\)th element of the first matrix satisfies the comparison with the \([i, j]\)th element of the second matrix. The result is 0 or 1. If the \([i, j]\) element less-than comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Examples

**Example 1: Comparing Two Matrices Using the LT Method**

The following example uses the LT method. The \([i, j]\)th element of matrix \(m\) is compared with the \([i, j]\) element of matrix \(m2\), using 0 or 1 for the result entries.

data _null_
   
dcl double a[3,3];
   dcl double b[3,3];
   dcl double c[3,3];
method run();
   dcl package matrix m;
   dcl package matrix m2;
   dcl package matrix r;
   dcl double i j;

   a := (1,2,3,4,5,6,7,8,9);
   b := (1,5,9,2,6,10,3,7,11);

   m = _new_ matrix(a, 3, 3);
   m2 = _new_ matrix(b, 3, 3);

   r = m.lt(m2);
   r.toarray(c);

   do i = 1 to 3;
      do j = 1 to 3;
         put c[i,j];
      end;
   end;
end;
enddata;
run;

The resulting matrix has the following values.
0 1 1
0 1 1
0 0 1

Example 2: Using a Scalar Matrix
The following example uses a scalar matrix with the LT method.

data _null_;  
dcl double a[3,3];  
dcl double b[1,1];  
dcl double c[3,3];

method run();
   dcl package matrix m;
   dcl package matrix m2;
   dcl package matrix r;
   dcl double i j;

   a := (1,3,3,4,5,6,7,8,9);
   b := (4);

   m = _new_ matrix(a, 3, 3);
   m2 = _new_ matrix(b, 1, 1);

   r = m.lt(m2);
   r.toarray(c);

   do i = 1 to 3;
      do j = 1 to 3;
         put c[i,j];
      end;
   end;
The resulting matrix has the following values.

```
1 1 1
0 0 0
0 0 0
```

**See Also**

- “Matrix Operations” on page 169

**Methods:**

- “GT Method” on page 1171
- “LE Method” on page 1176

---

**NEW Operator, Matrix Package**

Constructs an instance of a matrix package.

**Note:** The escape character (\) before the bracket indicates that the bracket is required in the syntax.

**Syntax**

```
package-variable = NEW [THIS | package-instance] MATRIX(array-name, rows, columns)
```

**Arguments**

- **package-variable**
  - specifies a name that can reference an instance of the matrix package.

- **[THIS]**
  - specifies that the package instance has global scope.

  See “Packages and Scope” on page 137

- **[package-instance]**
  - specifies that the new package instance has the same scope as package-instance.
  - package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

  See “Package-Specific Scope” on page 139

- **array-name**
  - specifies the name of an array to load into the matrix package.

- **rows**
  - specifies the number of rows in the matrix.
columns
specifies the number of columns in the matrix.

Details
A DS2 package is a collection of variables and methods of which particular instances
can be constructed and used in other DS2 programs.

When a matrix package is declared, the variable representing the package can be
considered an instance of the package. This means that two different matrix package
variables represent two completely separate copies of a package.

A matrix must be initialized before it can be used, and initialization is done in the code
stream, not in the declarations. To initialize a matrix, you must use the _NEW_ operator.

A matrix can be initialized with values from a DS2 array. Here is an example:

```
method init();
   dcl double a[3, 3];
   dcl package matrix m;

   a :=(1, 2, -1, 2, 1, 0, -1, 1, 2);
   m=_new_ matrix(a, 3, 3);
end;
```

In this example, a 3x3 array is initialized with values that you specify, and is used to set
up the matrix m. The values are read in row-major order, and the matrix that is produced
has the following values:

```
 1 2 -1
 2 1 0
-1 1 2
```

You can also initialize a matrix by using a variable array, as the following example
shows:

```
vararray double a [3, 3];
dcl package matrix m;

method init();
   a := {1, 2, -1, 2, 1, 0, -1, 1, 2};
   m=_new_ matrix(a, 3, 3);
end;
```

Note: Package variables are subject to all variable scoping rules. For more information,
see “Packages and Scope” on page 137.

See Also
- “Matrix Data Input” on page 167
- “Package Constructors and Destructors” on page 142

Statements:
- “DECLARE PACKAGE Statement, Matrix Package” on page 1155
MULT Method

Multiplies one matrix by another matrix.

Syntax

\[ r = \text{matrix-package-1}.\text{MULT}(\text{matrix-package-2}); \]

Arguments

- \( r \): specifies the matrix that is automatically created by the MULT method.
- \( \text{matrix-package-1} \): specifies the name of the first matrix package that is used in the multiplication operation.
- \( \text{matrix-package-2} \): specifies the name of the second matrix package that is used in the multiplication operation.

Details

The MULT method automatically creates a matrix that contains the result of matrix multiplication. The result matrix has the same number of rows as the first matrix and the same number of columns as the second matrix.

The following considerations apply when performing matrix multiplication:

- Array dimensions for the matrices that are used in multiplication operations must be compatible. Multiplication requires that the number of columns in the first matrix be equal to the number of rows in the second matrix. Otherwise, a run-time error is generated.
- If you multiply matrices that have missing values, you will receive a run-time error.

Examples

Example 1: Simple Matrix Multiplication

The following example uses the MULT method to perform a simple matrix multiplication. A 3x4 matrix, \( m \), is multiplied by a 4x3 matrix, \( m2 \), to obtain a 3x3 result, which is stored in matrix, \( r \). The values in matrix \( r \) can be placed into a 3x3 array, \( c \), and written to the SAS log.

```sas
data _null_
  dcl double a[3,4];
  dcl double b[4,3];
  dcl double c[3,3];
  method run()
    dcl package matrix m;
    dcl package matrix m2;
    dcl package matrix r;
    dcl double i j;
```

a := (1,2,3,4,5,6,7,8,9,10,11,12);
b := (1,5,9,2,6,10,3,7,11,4,8,12);

m = _new_ matrix(a, 3, 4);
m2 = _new_ matrix(b, 4, 3);

r = m.mult(m2);
r.toarray(c);

do i = 1 to 3;
do j = 1 to 3;
put c[i,j];
end;
end;
enddata;
run;

The following lines are written to the SAS log.

| 30 |
| 70 |
| 110 |
| 70 |
| 174 |
| 278 |
| 110 |
| 278 |
| 446 |

---

**Example 2: Multiply Two Matrices That Are Read from External Data**

This example multiplies two matrices that are read from external data. The IN method, which is an alias for the LOAD method, reads the matrices and the OUT method writes the output. For more information, see the “IN Method” on page 1172 and the “OUT Method” on page 1188.

proc ds2;
data x(keep = (a1 a2 a3)) y(keep = (b1 b2 b3 b4))/overwrite=yes;
vararray double a[3];
vararray double b[4];

method init();
dcl double i j;

/* Create output for matrix a */
do i = 1 to 4;
do j = 1 to 3;
a[j] = 2 * j + i;
end;
output x;
end;

/* Create output for matrix b */
do i = 1 to 3;
do j = 1 to 4;
b[j] = 3 * j - 2 * i;

end;
output y;
end;
end;
enddata;
run;
quit;

proc ds2;
data z(keep=(v1 v2 v3 v4))/overwrite=yes;
  drop i;
  vararray double v[4];
  vararray double a[3];
  vararray double b[4];
dcl package matrix ma mb;
dcl package matrix r;

  method init();
    dcl double i;
    ma = _new_ [this] matrix(4,3);
    mb = _new_ [this] matrix(3,4);

    /* Read matrix a (row-by-row) */
    do i = 1 to 4;
      set x;
      ma.in(a, i);
    end;

    /* Read matrix b (row-by-row) */
    do i = 1 to 3;
      set y;
      mb.in(b, i);
    end;
  end;

  method term();
    dcl double i;
    /* Multiply matrices */
    r = ma.mult(mb);

    /* Output result */
    do i = 1 to 4;
      r.out(v, i);
      output;
    end;
  end;
enddata;
run;
quit;

See Also
• “Matrix Operations” on page 169

Methods:
NE Method

Produces a scalar result in a not-equal-to comparison between elements in one matrix and elements in another matrix.

Syntax

\[ r = \text{matrix-package-1}.\text{NE(matrix-package-2)}; \]

Arguments

- \( r \)
  - Specifies a matrix that contains the results of a not-equal-to comparison between the values in two matrices.
- \( \text{matrix-package-1} \)
  - Specifies the name of the first matrix package that is used in the not-equal-to comparison.
- \( \text{matrix-package-2} \)
  - Specifies the name of the second matrix package that is used in the not-equal-to comparison.

Details

The NE relational operation produces a matrix that indicates whether an \([i, j]^{th}\) element of the first matrix satisfies the comparison with the \([i, j]^{th}\) element of the second matrix. The result is 0 or 1. If the \([i, j]\) elements are not equal, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

The following example compares the elements in two matrices for inequality. Note that missing values are compared.

data _null_;
  dcl double a[2,2];
  dcl double b[2,2];
  dcl double f[2,2];

  method run();
    dcl package matrix m;
    dcl package matrix m1;
    dcl package matrix r;
    dcl double i j;

    a := (2, 2, 3, .);
    b := (4, 5, 1, .);

    m = _new_matrix(a, 2, 2);
    m1 = _new_matrix(b, 2, 2);
r = m.ne(m1);
r.toarray(f);

do i = 1 to 2;
do j = 1 to 2;
   put f[i,j];
end;
end;
end;
enddata;
run;

The resulting matrix has the following values.

1 1
1 0

See Also

- “Matrix Operations” on page 169

Methods:

- “EQ Method” on page 1167
- “GE Method” on page 1170
- “LE Method” on page 1176

OR Method

Compares two matrices based on the OR logical operation, and returns the resulting matrix.

Syntax

\[ r = \text{matrix-package-1}.\text{OR(matrix-package-2)}; \]

Arguments

\( r \)
\hspace{1cm} \text{specifies a matrix that contains the results of an OR comparison between the values of two matrices.}

\( \text{matrix-package-1} \)
\hspace{1cm} \text{specifies the name of the first matrix package that is used in the OR comparison.}

\( \text{matrix-package-2} \)
\hspace{1cm} \text{specifies the name of the second matrix package that is used in the OR comparison.}

Details

The OR logical operator behaves similarly to the binary relational operations (LT, LE, GE, GT, NE, and EQ). In each case, the OR logical operation is applied to the \([i, j]^{th}\) elements of two matrices and placed in the result matrix \(r\).
Example

```plaintext
r=r1.or(m2);
```

See Also

- “Matrix Operations” on page 169

Methods:

- “ALL_OR Method” on page 1144
- “AND Method” on page 1145
- “ANY_OR Method” on page 1152

## OUT Method

Writes matrix row data to a variable array.

### Syntax

```plaintext
matrix-package.OUT(array-name);
```

### Arguments

- **matrix-package**
  - Specifies a matrix package to be used with the OUT method.
- **array-name**
  - Specifies a matrix that is used in writing output.

#### Restriction

The array dimensions must match the matrix dimensions. Otherwise, an error occurs.

#### Tip

You can use variable arrays.

#### See

Chapter 15, “DS2 Arrays,” on page 121

### Details

The OUT method writes matrix row data to a variable array. Variable arrays can be used to input and output data using an existing DS2 table and output statements. For example, you can read data from a table in matrix form, find the matrix inverse, and write the result matrix to an output table. See the example below.

The OUT method can be overloaded to accept an integer argument that tells which row of a matrix to write to a variable array.

The synonym for the OUT method is the TOVARARRAY method. The following two statements are equivalent:

```plaintext
r.tovararray(va, i);
```

```plaintext
r.out(va, i);
```

The OUT method, which writes data, is often used with the IN method. The IN and OUT methods are overloaded to accept an integer argument that tells which row of a matrix to
load. For the IN method, the variable array row data is read into the matrix. For the OUT method, the matrix row data is written to the variable array. In this way, matrices can be loaded and unloaded a row at a time from and to external data storage using variable arrays.

Examples

**Example 1**
This example writes each row of an inverse to an output table using the variable array v.

```plaintext
do i=1 to 4;
   inv.out(v, i);
   output;
end;
```

**Example 2**
Similar to the standard IN method, a complete matrix can also be written to a variable array:

```plaintext
vararray double va(3, 3);
dcl package matrix r;

r=new_matrix(3, 3);
out(va);
```

In this example, a matrix does not need to be written row-by-row. The entire matrix can be written to the variable array. You can use this method in the case where the DS2 output statement (which is row-based) is not being used.

**Example 3: Loading and Writing Data**
For another example of using the IN and OUT methods, see “Example 2: Multiply Two Matrices That Are Read from External Data” on page 1184.

**Example 4: Writing the Entire Matrix to the Array**
Similar to the standard LOAD method, a complete matrix can also be written to a variable array:

```plaintext
vararray double va[3, 3];
dcl package matrix r;

r=new_matrix(3, 3);
out(va);
```

This example shows that matrix data does not need to be written row-by-row. You can write the entire matrix to the array. You could use this technique in a case where the DS2 OUTPUT statement (which is row-based) is not used.

See Also

- “Matrix Data Output” on page 169

Methods:

- “IN Method” on page 1172
- “OUT Method” on page 1188
ROWS Method

Returns the number of rows in the specified matrix.

Syntax

\[
\text{variable-name} = \text{matrix-package}.\text{ROWS}();
\]

Arguments

- **variable-name**
  - specifies the name of a variable that contains the number of rows after the method is complete.
- **matrix-package**
  - specifies a matrix package to use with the ROWS method.

Example

The following example returns the number of rows and columns in the matrix, m.

```sas
data _null_
   dcl double a[2, 3];
   method run();
      dcl package matrix m;
      dcl double mr mc;
      a := (1,2,3,4,5,6);
      m=_new_ matrix(a, 2, 3);
      mr=m.rows();
      mc=m.cols();
      put mr=;
      put mc=;
   end;
run;
```

The following lines are written to the SAS log.

```
mr=2
mc=3
```

See Also

- “Using the MATRIX Package” on page 167
### SQRT Method

Returns a matrix that contains the square root of each value in the input matrix.

**Syntax**

```plaintext
r = matrix-package.SQR( );
```

**Arguments**

- `r` specifies a matrix that contains the square root of each value in the input matrix.
- `matrix-package` specifies a matrix package to use with the SQRT method.

**See Also**

“Matrix Operations” on page 169

---

### SUB Method

Subtracts one matrix from another.

**Syntax**

```plaintext
r = matrix-package-1.SUB(matrix-package-2);
```

**Arguments**

- `r` specifies the matrix that is automatically created by the SUB method.
- `matrix-package-1` specifies the name of the first matrix package that is used in the subtraction operation.
- `matrix-package-2` specifies the name of the second matrix package that is used in the subtraction operation.

**Details**

The matrix dimensions for the SUB method must be the same size in order for the matrix subtraction to take place. Each `[i, j]` element in the first matrix is subtracted from its corresponding `[i, j]` element in the second matrix.

If you subtract matrices that have missing values, you do not receive an error.

It is also possible to perform scalar subtraction by using a 1x1 matrix.
Example

The following is an example of using the SUB method to subtract two matrices.

data _null_;  
dcl double c[3,3];  
dcl double d[1,1];  
dcl double f[3,3];  
method run();  
dcl package matrix m3;  
dcl package matrix m4;  
dcl package matrix r;  
dcl double i j;  

c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);  
d := (1);  

m3 = _new_ matrix(c, 3, 3);  
m4 = _new_ matrix(d, 1, 1);  

r = m3.sub(m4);  
r.toarray(f);  

do i = 1 to 3;  
  do j = 1 to 3;  
    put f[i,j];  
  end;  
end;
enddata;  
run;

The following lines are written to the SAS log.

```
-1
-1
-2
0
-3.2
1.2
-4.3
3.4
4.5
```

See Also

- “Matrix Operations” on page 169

Methods:

- “ADD Method, Matrix Package” on page 1134

TOARRAY Method

Moves the values from a matrix package into a DS2 array.
Syntax

\textit{array-name}.\texttt{TOARRAY(matrix-package)};

Arguments

\textit{array-name}

specifies the name of an array to which matrix values are moved.

\textit{matrix-package}

specifies the matrix package from which values are moved into an array.

Details

You use the \texttt{TOARRAY} method to move values from a matrix to a DS2 array. The array can then be used directly in a DS2 program.

\textit{Note}: You can also move values to a variable array by using the \texttt{TOVARARRAY} method.

Example

This example moves the values from a matrix into a DS2 array.

\begin{verbatim}
data _null_;  dcl double a[3,3];  dcl double c[3,3];
  method init();  dcl double i j;  dcl package matrix m;
    a := (1,2,3,4,5,6,7,8,9);  m=_new_ matrix(a,3,3);  m.toarray(c);  do i=1 to 3;
    do j=1 to 3;
      put c[i,j];
    end;
  end;
enddata;
run;
\end{verbatim}

The resulting array has the following values.

\begin{verbatim}
  1  2  3
  4  5  6
  7  8  9
\end{verbatim}

See Also

\begin{itemize}
  \item “Matrix Data Output” on page 169
  \item Chapter 15, “DS2 Arrays,” on page 121
\end{itemize}

Methods:
TOVARARRAY Method

Moves the values from a matrix package into a variable array.

Syntax

\[ \text{variable-array-name}.\text{TOVARARRAY}(\text{matrix-package}); \]

Arguments

- **variable-array-name**
  - specifies the name of a variable array to which matrix values are moved.

- **matrix-package**
  - specifies the matrix package from which values are moved into a variable array.

Details

You use the TOVARARRAY method to move values from a matrix to a DS2 variable array. The variable array can then be used directly in a DS2 program.

*Note:* You can also move values to an array by using the TOARRAY method.

Example

This example shows how to move the values from a matrix into a variable array.

```plaintext
data _null_;  
dcl double a[3, 3];  
vararray double c[3, 3];  
method run();  
dcl package matrix m;  
dcl double i j;  
a := (1,2,3,4,5,6,7,8,9);  
m=_new_ matrix{a, 3, 3};  
m.tovararray(c);  
do i=1 to 3;  
do j=1 to 3;  
  put c[i, j];  
end;  
end;  
run;  
The resulting array has the following values.  
1  2  3  
4  5  6  
7  8  9  
```
See Also

- “Matrix Data Output” on page 169
- “Variable Arrays” on page 123

Methods:

- “TOARRAY Method” on page 1192

TRANS Method

Returns a matrix that transposes the rows and columns of the input matrix.

Syntax

\[ r = \text{matrix-package}.\text{TRANS}(); \]

Arguments

\[ r \]

specifies the matrix that contains the transposition of the input matrix.

\[ \text{matrix-package} \]

specifies a matrix package to use with the TRANS method.

Details

The TRANS method exchanges the rows and columns of a given matrix producing the transpose of matrix. If \( v \) is the value in the \( i^{th} \) row and \( j^{th} \) column of matrix, then the transpose of matrix contains \( v \) in the \( j^{th} \) row and \( i^{th} \) column. If matrix contains \( n \) rows and \( p \) columns, the transpose has \( p \) rows and \( n \) columns.

Example

The following example transposes a 3x4 matrix to produce a 4x3 result matrix.

```
data _null_; 
  dcl double a[3, 4];
  dcl double b[4, 3];
  method run();
  dcl package matrix m;
  dcl package matrix r;
  dcl double i j;
  a := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
  m=new_matrix(a, 3, 4);
  r=m.trans();
  r.toarray(b);
  do i=1 to 4;
      do j=1 to 3;
```
put b[i, j];
end;
end;
end;
enddata;
run;

The input matrix is shown here.

1   2  3  
4   5  6  
7   8  9  
10  11 12

The transposed, result matrix is shown here.

1   5   9  
2   6  10  
3   7   11 
4   8  12

See Also

“Matrix Operations” on page 169
Chapter 36
DS2 SQLSTMT Package
Methods, Operators, and
Statements

Dictionary

BINDPARAMETERS Method .............................................. 1198
BINDRESULTS Method .................................................. 1199
CLOSERESULTS Method .................................................. 1200
DECLARE PACKAGE Statement, SQLSTMT Package .................. 1200
DELETE Method, SQLSTMT Package .................................. 1202
EXECUTE Method ....................................................... 1203
FETCH Method ........................................................... 1204
GETBIGINT Method ....................................................... 1204
GETBINARY Method ..................................................... 1206
GETCHAR Method ........................................................ 1207
GETCOLUMNCOUNT Method ............................................. 1208
GETCOLUMNNAME Method .............................................. 1209
GETCOLUMNTYPENAME Method ...................................... 1209
GETDATE Method ........................................................ 1210
GETDECIMAL Method .................................................... 1211
GETDOUBLE Method ..................................................... 1212
GETINTEGER Method ...................................................... 1213
GETNCHAR Method ....................................................... 1214
GETNUMERIC Method .................................................... 1215
GETNVARCHAR Method .................................................. 1216
GETREAL Method ......................................................... 1217
GETSMALLINT Method ................................................... 1218
GETTIME Method ......................................................... 1219
GETTIMESTAMP Method ............................................... 1220
GETTINYINT Method ..................................................... 1221
GETVARBINARY Method .................................................. 1222
GETVARCHAR Method .................................................... 1223
ISPREPARED Method ..................................................... 1224
.NEW._ Operator, SQLSTMT Package ................................. 1225
PREPARE Method ......................................................... 1226
SETPARAMETERS Method ............................................. 1227
SETBINARY Method ...................................................... 1228
SETCHAR Method ......................................................... 1229
SETDATE Method ........................................................ 1230
SETDECIMAL Method ..................................................... 1231
SETDOUBLE Method ..................................................... 1232
SETINTEGER Method ..................................................... 1233
SETNVARCHAR Method .................................................. 1234
SETNCHAR Method ....................................................... 1235
SETNUMERIC Method ..................................................... 1236
SETVARCHAR Method ..................................................... 1237

Dictionary End

1197
Dictionary

**BINDPARAMETERS Method**

Binds a list of variables to the parameters in the FedSQL statement.

**Syntax**

```
package.BINDPARAMETERS ([parameter-variable-list]);
```

**Arguments**

- **package**
  
  specifies an instance of the SQLSTMT package.

- **[parameter-variable-list]**
  
  specifies a variable list or named variable list that contains the variables to bind to the FedSQL statement’s parameters.

**Requirement**

Variables must be in the form of a variable list, which must be enclosed in brackets ([ ]) or a named variable list.

**Tip**

The number of variables in the variable list must match the number of parameters in the FedSQL statement.

**See**

“Specifying FedSQL Statement Parameter Values” on page 174

**Details**

If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package’s SETtype methods.

The BINDPARAMETERS method binds the variables in the specified variable list to the parameters in the FedSQL statement.

Parameter values must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and a SETtype method is invoked.

If the type of a bound variable differs from the corresponding parameter’s type, the bound variable’s value is converted to the parameter’s type.

The BINDPARAMETERS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.
A run-time error also results if the BINDPARAMETERS method is called after the FedSQL statement is executed.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “BINDRESULTS Method” on page 1199
• SET “type” methods in this chapter

BINDRESULTS Method

Binds a list of variables to the columns of the result set of the FedSQL statement.

Syntax

package.BINDRESULTS (parameter-variable-list);

Arguments

package
specifies an instance of the SQLSTMT package.

parameter-variable-list
specifies a variable list or named variable list that contains the variables to bind to the columns of the result set.

Requirement
Variables must be in the form of a variable list, which must be enclosed in brackets ([ ]) or a named variable list.

Tip
The number of variables in the variable list must match the number of columns in the result set.

See
“Specifying FedSQL Statement Parameter Values” on page 174

Details

The FETCH method returns the next row of data from the result set. If variables are bound to the result set columns with the BINDRESULTS method, then the fetched data for each result set column is placed in the variable bound to that column. If the type of a variable differs from the corresponding column’s type, the column data value is converted to the variable’s type.

The result data must be accessed exclusively with bound variables or exclusively with the GET[type] methods. A run-time error results if variables are bound to result set columns and a GET[type] method is invoked.

A run-time error results if the BINDRESULTS method is called after the result data is fetched.

The BINDRESULTS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.
CLOSERESULT Method

Releases the result set from the last execution of the statement.

Syntax

\[
\text{package.CLOSERESULTS}();
\]

Arguments

\[\text{package}\]

specifies an instance of the SQLSTMT package.

Details

An SQLSTMT instance maintains only one result set. The result set is automatically released when the FedSQL statement is executed or deleted.

The CLOSERESULTS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

See Also

“Using the SQLSTMT Package” on page 172

DECLARE PACKAGE Statement, SQLSTMT Package

Creates a package variable and enables you to create an instance of the SQLSTMT package.

Category: Local

Note: Braces in the syntax convention indicate a syntax grouping. The escape character ( \ ) before a brace indicates that the brace is required in the syntax.

Syntax

Form 1: DECLARE PACKAGE SQLSTMT variable ['sql-text [,[parameter-variable-list]])];
Form 2: DECLARE PACKAGE SQLSTMT variable ['sql-text [, connection-string]]);
Form 3: DECLARE PACKAGE SQLSTMT variable ( );
Form 4: DECLARE PACKAGE SQLSTMT variable ;

See Also

“Using the SQLSTMT Package” on page 172
**Arguments**

**variable**

specifies a name that can reference an instance of the SQLSTMT package.

**'sql-text'**

is a valid FedSQL statement or string variable that contains a FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.

**Requirement**

The FedSQL statement must be enclosed in single quotation marks (') unless the statement is stored in a string variable.

**Notes**

The statement is a string literal.

The rules for identifiers for the FedSQL language apply to variables used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

**[parameter-variable-list]**

specifies variables that are bound to the parameters contained in the FedSQL statement.

**Requirements**

Variables must be in the form of a variable list that must be enclosed in brackets ([ ]) or a named variable list.

Parameter data must be specified exclusively with either bound variables or exclusively with the SQLSTMT SET type methods.

**See**

“Specifying FedSQL Statement Parameter Values” on page 174

**connection-string**

contains the fully specified connection string.

**Default**

If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

**Note**

The connection string is a string literal.

**Tip**

A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

**See**

This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see the SAS Federation Server: Administrator's Guide.

**Details**

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

The SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution. The FedSQL statements could create, modify, or delete tables. If the FedSQL statements selects rows from a table, the SQLSTMT package provides methods for interrogating the rows returned in a result set. The SQLSTMT package is predefined for DS2 programs.
You declare an SQLSTMT package by using the DECLARE PACKAGE statement. This associates an SQLSTMT package with an SQLSTM name.

There are two ways to construct an instance of an SQLSTMT package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  ```sql
  declare package sqlstmt sqlpkg;
  sqlpkg = _NEW_ sqlstmt('update db2.dataset2 set y=? where x=?', [y x]);
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:
  ```sql
  declare package sqlstmt sqlpkg('update db2.dataset2 set y=? where x=?', [y x]);
  ```

If the DECLARE statement includes arguments for construction within its parentheses (and no arguments is valid for the SQLSTMT package), then the package instance is allocated. If no parentheses are included, then a variable is created but the package instance is not allocated.

When an SQLSTMT instance is created with SQL text (Forms 1 and 2), the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package’s SET type methods. The DECLARE PACKAGE statement binds the variables in the optional variable list to the parameters in the FedSQL statement.

When an SQLSTMT instance is created without FedSQL text (Form 3), the SQLSMT instance is allocated and left in an unprepared state. Use the PREPARE method to prepare the FedSQL statement at a later time.

*Note:* Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

For more information about SQLSTMT packages, see “Using the SQLSTMT Package” on page 172.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods:**

- “BINDPARAMETERS Method” on page 1198
- “PREPARE Method” on page 1230
- SET “type” methods in this chapter

**Operators:**

- “_NEW_ Operator, SQLSTMT Package” on page 1228

---

**DELETE Method, SQLSTMT Package**

Deletes an instance of the SQLSTMT package.

*Note:* The DELETE method is not required. When an SQLSTMT package goes out of scope, the package is deleted.
Syntax

```c
package.DELETE();
```

**Arguments**

*package*

specifies an instance of the SQLSTMT package variable.

**Details**

When you no longer need the SQLSTMT package, delete it by using the DELETE method. If you attempt to use an SQLSTMT package after you delete it, an error will be written to the log.

**See Also**

- “Using the SQLSTMT Package” on page 172
- “Package Constructors and Destructors” on page 142

---

**EXECUTE Method**

Executes the FedSQL statement.

Syntax

```c
package.EXECUTE();
```

**Arguments**

*package*

specifies an instance of the SQLSTMT package variable.

**Details**

The EXECUTE method executes the FedSQL statement and returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists when a FedSQL UPDATE or DELETE statement does not affect any rows.

An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed.

The FedSQL statement executes dynamically at run time. Because the statement is prepared at run time, it can be built and customized dynamically during the execution of the DS2 program.

**See Also**

“Using the SQLSTMT Package” on page 172
FETCH Method

Fetches the next row of data from the result set of the FedSQL statement.

Syntax

```
package.FETCH ([result-variable-list]);
```

Arguments

- **package**
  - specifies an instance of the SQLSTMT package.

- **result-variable-list**
  - specifies a variable list that contains the variables to bind to the columns of the result set.

Details

The FETCH method returns the next row of data from the result set. A status indicator is returned. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists if the next row to be fetched is located after the end of the result set.

If variables are bound to the result set columns with the BINDRESULTS method or by the FETCH method, then the fetched data for each result set column is placed in the variable bound to that column. If the variables are not bound to the result set columns, the fetched data can be returned by the GET type methods.

A run-time error results if FETCH is called before the statement is executed.

An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed. You can also use the CLOSERESULTS method to release the result set at any time.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “BINDRESULTS Method” on page 1199
- GET “type” methods in this chapter

GETBIGINT Method

Returns the value of the designated result set column as type BIGINT.

**Note:** You can call the GETBIGINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.
Syntax

```plaintext
variable = package.GETBIGINT(index);
package.GETBIGINT(index, variable, rc);
```

**Arguments**

- **variable**
  - Specifies the variable that will hold the value of the designated result set column.
  - **Note**: If the designated result set column’s type is not type BIGINT, the column value is converted to type BIGINT and then returned.

- **package**
  - Specifies an instance of the SQLSTMT package.

- **index**
  - Specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  - Specifies the variable in which to place the return code. The following values are possible:
    - 0 (SUCCESS) the result set column data is retrieved.
    - 1 (ERROR) an error occurred during data retrieval.
    - 2 (NODATA) there is no more data to be retrieved.

**Details**

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETBIGINT method returns the value of the designated result set column as type BIGINT. If the designated result set column’s type is not type BIGINT, the column value is converted to type BIGINT and then returned.

A run-time error results if the GETBIGINT method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods:**

- “GETINTEGER Method” on page 1214
- “GETSMALLINT Method” on page 1220
- “GETTINYINT Method” on page 1224
GETBINARY Method

Returns the designated result set column as type BINARY.

**Note:** You can call the GETBINARY method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

**Syntax**

\[
\text{variable} = \text{package.GETBINARY}(\text{index}); \\
\text{package.GETBINARY}(\text{index}, \text{variable}, \text{rc});
\]

**Arguments**

- **variable**
  - specifies the variable that will hold the value of the designated result set column.
  - **Note:** If the designated result set column’s type is not type BINARY, the column value is converted to type BINARY and then returned.

- **package**
  - specifies an instance of the SQLSTMT package.

- **index**
  - specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  - specifies the variable in which to place the return code. The following values are possible:
    - 0
      - (SUCCESS) the result set column data is retrieved.
    - 1
      - (ERROR) an error occurred during data retrieval.
    - 2
      - (NODATA) there is no more data to be retrieved.

**Details**

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETBINARY method returns the value of the designated result set column as type BINARY. If the designated result set column’s type is not type BINARY, the column value is converted to type BINARY and then returned.
A run-time error results if the GETBINARY method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods:**

- “GETVARBINARY Method” on page 1225
- “SETBINARY Method” on page 1232
- “SETVARBINARY Method” on page 1247

---

**GETCHAR Method**

Returns the designated result set column as type CHAR.

**Note:** You can call the GETCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

**Syntax**

```plaintext
variable = package.GETCHAR(index);

package.GETCHAR(index, variable, rc);
```

**Arguments**

- **variable**
  - specifies the variable that will hold the value of the designated result set column.

  **Note** If the designated result set column’s type is not type CHAR, the column value is converted to type CHAR and then returned.

- **package**
  - specifies an instance of the SQLSTMT package.

- **index**
  - specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  - specifies the variable in which to place the return code. The following values are possible:

  0 (SUCCESS) the result set column data is retrieved.

  1 (ERROR) an error occurred during data retrieval.
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET\textit{type} methods.

The GETCHAR method returns the value of the designated result set column as type CHAR. If the designated result set column’s type is not type BIGINT, the column value is converted to type CHAR and then returned.

A run-time error results if the GETCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the \textit{GETtype} methods. A run-time error results if variables are bound to result set columns and a \textit{GETtype} method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETNCHAR Method” on page 1215
- “GETNVARCHAR Method” on page 1217
- “GETVARCHAR Method” on page 1226
- “SETCHAR Method” on page 1234
- “SETNCHAR Method” on page 1239
- “SETNVARCHAR Method” on page 1241
- “SETVARCHAR Method” on page 1248

\textbf{GETCOLUMNCOUNT Method}

Returns the number of columns in the result set.

\textbf{Syntax}

\begin{verbatim}
variable=package.GETCOLUMNCOUNT();
\end{verbatim}

\textbf{Arguments}

\begin{description}
\item[variable] specifies the variable that will hold the number of columns in the result set.
\item[package] specifies an instance of the SQLSTMT package.
GETCOLUMNNAME Method

Returns the column name of the result set column with the designated index.

Syntax

variable = package.GETCOLUMNNAME(index);

getColumnName(index, variable, rc);

Arguments

variable
specifies the variable that will hold the name of the designated result set column.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0 (SUCCESS) the result set column data is retrieved.

1 (ERROR) an error occurred during data retrieval.

See Also

Methods:

• “GETCOLUMNCOUNT Method” on page 1208
• “GETCOLUMNMYPENAME Method” on page 1209

GETCOLUMNTYPENAME Method

Returns the data type of the result set column with the designated index.

Syntax

variable = package.GETCOLUMNTYPENAME(index);

GETCOLUMNTYPENAME Method
package.getColumnTypeName(index, variable, rc);

**Arguments**

**variable**
- specifies the variable that will hold the data type of the result set column.

**package**
- specifies an instance of the SQLSTMT package.

**index**
- specifies the result set column index ordered sequentially, starting at 1.

**rc**
- specifies the variable in which to place the return code. The following values are possible:
  - 0 (SUCCESS) the result set column data is retrieved.
  - 1 (ERROR) an error occurred during data retrieval.

**Details**

GETCOLUMNTYPENAME returns only the data types that are supported by DS2. For more information, see Chapter 9, “DS2 Data Types,” on page 71.

**See Also**

**Methods:**
- “GETCOLUMNCOUNT Method” on page 1208
- “GETCOLUMNTYPENAME Method” on page 1209

---

**GETDATE Method**

Returns the designated result set column as type DATE.

**Note:** You can call the GETDATE method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

**Syntax**

```
variable = package.GETDATE(index);
package.GETDATE(index, variable, rc);
```

**Arguments**

**variable**
- specifies the variable that will hold the value of the designated result set column.

**Note** If the designated result set column’s type is not type DATE, the column value is converted to type DATE and then returned.
package specifies an instance of the SQLSTMT package.

index specifies the result set column index ordered sequentially, starting at 1.

rc specifies the variable in which to place the return code. The following values are possible:

0 (SUCCESS) the result set column data is retrieved.

1 (ERROR) an error occurred during data retrieval.

2 (NODATA) there is no more data to be retrieved.

Details
The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETDATE method returns the value of the designated result set column as type DATE. If the designated result set column’s type is not type DATE, the column value is converted to type DATE and then returned.

A run-time error results if the GETDATE method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also
• “Using the SQLSTMT Package” on page 172

Methods:
• “GETTIME Method” on page 1221
• “GETTIMESTAMP Method” on page 1222
• “SETDATE Method” on page 1235
• “SETTIME Method” on page 1244
• “SETTIMESTAMP Method” on page 1245

GETDECIMAL Method
Returns the designated result set column as type DECIMAL.

Note: You can call the GETDECIMAL method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.
Syntax

\[
\text{variable} = \text{package.GETDECIMAL (index)}; \\
\text{package.GETDECIMAL (index, variable, rc)};
\]

Arguments

\text{variable}

specifies the variable that will hold the value of the designated result set column.

\text{Note}

If the designated result set column’s type is not type DECIMAL, the column value is converted to type DECIMAL and then returned.

\text{package}

specifies an instance of the SQLSTMT package.

\text{index}

specifies the result set column index ordered sequentially, starting at 1.

\text{rc}

specifies the variable in which to place the return code. The following values are possible:

\begin{itemize}
  \item 0 (SUCCESS) the result set column data is retrieved.
  \item 1 (ERROR) an error occurred during data retrieval.
  \item 2 (NODATA) there is no more data to be retrieved.
\end{itemize}

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETDECIMAL method returns the value of the designated result set column as type DECIMAL. If the designated result set column’s type is not type DECIMAL, the column value is converted to type DECIMAL and then returned.

A run-time error results if the GETDECIMAL method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “SETDECIMAL Method” on page 1236
GETDOUBLE Method

Returns the designated result set column as type DOUBLE.

Note: You can call the GETDOUBLE method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

Syntax

```
variable=package.GETDOUBLE(index);
package.GETDOUBLE(index, variable, rc);
```

Arguments

**variable**

specifies the variable that will hold the value of the designated result set column.

Note

If the designated result set column’s type is not type DOUBLE, the column value is converted to type DOUBLE and then returned.

**package**

specifies an instance of the SQLSTMT package.

**index**

specifies the result set column index ordered sequentially, starting at 1.

**rc**

specifies the variable in which to place the return code. The following values are possible:

- 0
  - (SUCCESS) the result set column data is retrieved.
- 1
  - (ERROR) an error occurred during data retrieval.
- 2
  - (NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETDOUBLE method returns the value of the designated result set column as type DOUBLE. If the designated result set column’s type is not type DOUBLE, the column value is converted to type DOUBLE and then returned.

A run-time error results if the GETDOUBLE method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.
GETINTEGER Method

Gets the designated result set column as type INTEGER.

**Note:** You can call the GETINTEGER method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

**Syntax**

```plaintext
variable = package.GETINTEGER(index);
package.GETINTEGER(index, variable, rc);
```

**Arguments**

- `variable` specifies the variable that will hold the value of the designated result set column.
  
  **Note** If the designated result set column’s type is not type INTEGER, the column value is converted to type INTEGER and then returned.

- `package` specifies an instance of the SQLSTMT package.

- `index` specifies the result set column index ordered sequentially, starting at 1.

- `rc` specifies the variable in which to place the return code. The following values are possible:
  
  0  
  (SUCCESS) the result set column data is retrieved.

  1  
  (ERROR) an error occurred during data retrieval.

  2  
  (NODATA) there is no more data to be retrieved.

**Details**

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETINTEGER method returns the value of the designated result set column as type INTEGER. If the designated result set column’s type is not type INTEGER, the column value is converted to type INTEGER and then returned.
A run-time error results if the GETINTEGER method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “GETBIGINT Method” on page 1204
• “GETSMALLINT Method” on page 1220
• “GETTINYINT Method” on page 1224
• “SETBIGINT Method” on page 1231
• “SETINTEGER Method” on page 1238
• “SETSMLLINT Method” on page 1243
• “SETTINYINT Method” on page 1246

GETNCHAR Method

Gets the designated result set column as type NCHAR.

Note: You can call the GETCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

Syntax

\[
\text{variable} = \text{package} \cdot \text{GETNCHAR} (\text{index}) ; \\
\text{package} . \text{GETNCHAR} (\text{index}, \text{variable}, \text{rc}) ;
\]

Arguments

\text{variable}

specifies the variable that will hold the value of the designated result set column.

Note

If the designated result set column’s type is not type NCHAR, the column value is converted to type NCHAR and then returned.

\text{package}

specifies an instance of the SQLSTMT package.

\text{index}

specifies the result set column index ordered sequentially, starting at 1.

\text{rc}

specifies the variable in which to place the return code. The following values are possible:
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details
The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET\textit{type} methods.

The GETNCHAR method returns the value of the designated result set column as type NCHAR. If the designated result set column’s type is not type NCHAR, the column value is converted to type NCHAR and then returned.

A run-time error results if the GETNCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET\textit{type} methods. A run-time error results if variables are bound to result set columns and a GET\textit{type} method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also
• “Using the SQLSTMT Package” on page 172

Methods:
• “GETCHAR Method” on page 1207
• “GETNVARCHAR Method” on page 1217
• “GETVARCHAR Method” on page 1226
• “SETCHAR Method” on page 1234
• “SETNCHAR Method” on page 1239
• “SETNVARCHAR Method” on page 1241
• “SETVARCHAR Method” on page 1248

GETNUMERIC Method
Returns the designated result set column as type NUMERIC.

Note: You can call the GETNUMERIC method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the \textit{rc} status indicator.

Syntax
\texttt{variable} = \texttt{package.GETNUMERIC (index)};
\texttt{package.GETNUMERIC (index, variable, rc)};
Arguments

variable
    specifies the variable that will hold the value of the designated result set column.

Note
    If the designated result set column’s type is not type NUMERIC, the column value is converted to type NUMERIC and then returned.

package
    specifies an instance of the SQLSTMT package.

index
    specifies the result set column index ordered sequentially, starting at 1.

rc
    specifies the variable in which to place the return code. The following values are possible:
    0
        (SUCCESS) the result set column data is retrieved.
    1
        (ERROR) an error occurred during data retrieval.
    2
        (NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETNUMERIC method returns the value of the designated result set column as type NUMERIC. If the designated result set column’s type is not type NUMERIC, the column value is converted to type NUMERIC and then returned.

A run-time error results if the GETNUMERIC method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

-  “Using the SQLSTMT Package” on page 172

Methods:

-  “SETNUMERIC Method” on page 1240

GETNVARCHAR Method

Returns the designated result set column as type NVARCHAR.
You can call the GETVARCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

Syntax

```plaintext
variable = package.GETVARCHAR (index);
package.GETVARCHAR (index, variable, rc);
```

Arguments

- **variable**
  - specifies the variable that will hold the value of the designated result set column.
  - **Note** If the designated result set column’s type is not type NVARCHAR, the column value is converted to type NVARCHAR and then returned.

- **package**
  - specifies an instance of the SQLSTMT package.

- **index**
  - specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  - specifies the variable in which to place the return code. The following values are possible:
    - **0** (SUCCESS) the result set column data is retrieved.
    - **1** (ERROR) an error occurred during data retrieval.
    - **2** (NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETVARCHAR method returns the value of the designated result set column as type NVARCHAR. If the designated result set column’s type is not type NVARCHAR, the column value is converted to type NVARCHAR and then returned.

A run-time error results if the GETVARCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:
GETREAL Method

Returns the designated result set column as type REAL.

**Note:** You can call the GETREAL method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

**Syntax**

```
variable=package.GETREAL (index);
package.GETREAL (index, variable, rc);
```

**Arguments**

- **variable**
  - specifies the variable that will hold the value of the designated result set column.
  - **Note:** If the designated result set column’s type is not type REAL, the column value is converted to type REAL and then returned.

- **package**
  - specifies an instance of the SQLSTMT package.

- **index**
  - specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  - specifies the variable in which to place the return code. The following values are possible:
    - 0
      - (SUCCESS) the result set column data is retrieved.
    - 1
      - (ERROR) an error occurred during data retrieval.
    - 2
      - (NODATA) there is no more data to be retrieved.

**Details**

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.
The GETREAL method returns the value of the designated result set column as type REAL. If the designated result set column’s type is not type REAL, the column value is converted to type REAL and then returned.

A run-time error results if the GETREAL method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “SETREAL Method” on page 1242

GETSMALLINT Method

Returns the designated result set column as type SMALLINT.

Note: You can call the GETSMALLINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

Syntax

\[
\text{variable} = \text{package.GETSMALLINT} (\text{index});
\]

\[
\text{package.GETSMALLINT} (\text{index, variable, rc});
\]

Arguments

variable

specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column’s type is not type SMALLINT, the column value is converted to type SMALLINT and then returned.

package

specifies an instance of the SQLSTMT package.

index

specifies the result set column index ordered sequentially, starting at 1.

rc

specifies the variable in which to place the return code. The following values are possible:

0 (SUCCESS) the result set column data is retrieved.

1 (ERROR) an error occurred during data retrieval.
(NODATA) there is no more data to be retrieved.

Details
The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET type methods.

The GETSMALLINT method returns the value of the designated result set column as type SMALLINT. If the designated result set column’s type is not type SMALLINT, the column value is converted to type SMALLINT and then returned.

A run-time error results if the GETSMALLINT method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET type methods. A run-time error results if variables are bound to result set columns and a GET type method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also
• “Using the SQLSTM Package” on page 172

Methods:
• “GETBIGINT Method” on page 1204
• “GETINTEGER Method” on page 1214
• “GETTINYINT Method” on page 1224
• “SETBIGINT Method” on page 1231
• “SETINTEGER Method” on page 1238
• “SETSMAINT Method” on page 1243
• “SETTINYINT Method” on page 1246

---

GETTIME Method
Returns the designated result set column as type TIME.

Note: You can call the GETTIME method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.

Syntax
variable=package.GETTIME (index);
package.GETTIME (index, variable, rc);

Arguments
variable
specifies the variable that will hold the value of the designated result set column.
If the designated result set column’s type is not type TIME, the column value is converted to type TIME and then returned.

**package**

specifies an instance of the SQLSTMT package.

**index**

specifies the result set column index ordered sequentially, starting at 1.

**rc**

specifies the variable in which to place the return code. The following values are possible:

0  
(SUCCESS) the result set column data is retrieved.

1  
(ERROR) an error occurred during data retrieval.

2  
(NODATA) there is no more data to be retrieved.

**Details**

The GETTIME method returns the value of the designated result set column as type TIME. If the designated result set column’s type is not type TIME, the column value is converted to type TIME and then returned.

A run-time error results if the GETTIME method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods:**

- “GETDATE Method” on page 1210
- “GETTIMESTAMP Method” on page 1222
- “SETDATE Method” on page 1235
- “SETTIME Method” on page 1244
- “SETTIMESTAMP Method” on page 1245

**GETTIMESTAMP Method**

Returns the designated result set column as type TIMESTAMP.

**Note:** You can call the GETTIMESTAMP method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the rc status indicator.
Syntax

\[ \text{variable} = \text{package.GETTIMESTAMP (index);} \]
\[ \text{package.GETTIMESTAMP (index, variable, rc);} \]

Arguments

\text{variable}

specifies the variable that will hold the value of the designated result set column.

\text{Note}

If the designated result set column’s type is not type TIMESTAMP, the column value is converted to type TIMESTAMP and then returned.

\text{package}

specifies an instance of the SQLSTMT package.

\text{index}

specifies the result set column index ordered sequentially, starting at 1.

\text{rc}

specifies the variable in which to place the return code. The following values are possible:

\begin{align*}
0 & \quad \text{(SUCCESS) the result set column data is retrieved.} \\
1 & \quad \text{(ERROR) an error occurred during data retrieval.} \\
2 & \quad \text{(NODATA) there is no more data to be retrieved.}
\end{align*}

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET\textit{type} methods.

The GETTIMESTAMP method returns the value of the designated result set column as type TIMESTAMP. If the designated result set column’s type is not type TIMESTAMP, the column value is converted to type TIMESTAMP and then returned.

A run-time error results if the GETTIMESTAMP method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET\textit{type} methods. A run-time error results if variables are bound to result set columns and a GET\textit{type} method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETDATE Method” on page 1210
- “GETTIME Method” on page 1221
- “SETDATE Method” on page 1235
GETTINYINT Method

Returns the designated result set column as type TINYINT.

**Note:** You can call the GETTINYINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the \( rc \) status indicator.

### Syntax

\[
\text{variable} = \text{package.GETTINYINT} (\text{index}); \\
\text{package.GETTINYINT} (\text{index, variable, rc});
\]

### Arguments

- **variable**
  Specifies the variable that will hold the value of the designated result set column.

  **Note** If the designated result set column’s type is not type TINYINT, the column value is converted to type TINYINT and then returned.

- **package**
  Specifies an instance of the SQLSTMT package.

- **index**
  Specifies the result set column index ordered sequentially, starting at 1.

- **rc**
  Specifies the variable in which to place the return code. The following values are possible:

  0  
  (SUCCESS) the result set column data is retrieved.

  1  
  (ERROR) an error occurred during data retrieval.

  2  
  (NODATA) there is no more data to be retrieved.

### Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET\_type methods.

The GETTINYINT method returns the value of the designated result set column as type TINYINT. If the designated result set column’s type is not type TINYINT, the column value is converted to type TINYINT and then returned.

A run-time error results if the GETTINYINT method is called before the result set is fetched.
The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also

• “Using the SQLSTM Package” on page 172

Methods:

• “GETBIGINT Method” on page 1204
• “GETINTEGER Method” on page 1214
• “GETSMALLINT Method” on page 1220
• “SETBIGINT Method” on page 1231
• “SETINTEGER Method” on page 1238
• “SETSMALLINT Method” on page 1243
• “SETTINYINT Method” on page 1246

GETVARBINARY Method

Returns the designated result set column as type VARBINARY.

Note: You can call the GETVARBINARY method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

Syntax

variable=package.GETVARBINARY (index);
package.GETVARBINARY (index, variable, rc);

Arguments

variable

specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column’s type is not type VARBINARY, the column value is converted to type VARBINARY and then returned.

package

specifies an instance of the SQLSTM package.

index

specifies the result set column index ordered sequentially, starting at 1.

rc

specifies the variable in which to place the return code. The following values are possible:

0 (SUCCESS) the result set column data is retrieved.
Details
The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETVARBINARY method returns the value of the designated result set column as type VARBINARY. If the designated result set column’s type is not type VARBINARY, the column value is converted to type VARBINARY and then returned.

A run-time error results if the GETVARBINARY method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also
• “Using the SQLSTMN Package” on page 172

Methods:
• “GETBINARY Method” on page 1206
• “SETBINARY Method” on page 1232
• “SETVARBINARY Method” on page 1247

GETVARCHAR Method
Returns the designated result set column as type VARCHAR.

Note: You can call the GETVARCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the rc status indicator.

Syntax

\[
\text{variable=package.GETVARCHAR (index);} \\
\text{package.GETVARCHAR (index, variable, rc);} \\
\]

Arguments

\textbf{variable}

specifies the variable that will hold the value of the designated result set column.

\textbf{Note} If the designated result set column’s type is not type VARCHAR, the column value is converted to type VARCHAR and then returned.
package
   specifies an instance of the SQLSTMT package.

index
   specifies the result set column index ordered sequentially, starting at 1.

rc
   specifies the variable in which to place the return code. The following values are possible:
   0
      (SUCCESS) the result set column data is retrieved.
   1
      (ERROR) an error occurred during data retrieval.
   2
      (NODATA) there is no more data to be retrieved.

Details
The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GETtype methods.

The GETVARCHAR method returns the value of the designated result set column as type VARCHAR. If the designated result set column’s type is not type VARCHAR, the column value is converted to type VARCHAR and then returned.

A run-time error results if the GETVARCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GETtype methods. A run-time error results if variables are bound to result set columns and a GETtype method is invoked.

For more information, see “Accessing Result Set Data” on page 175.

See Also
•  “Using the SQLSTMT Package” on page 172

Methods:
•  “GETCHAR Method” on page 1207
•  “GETNCHAR Method” on page 1215
•  “GETNVARCHAR Method” on page 1217
•  “SETCHAR Method” on page 1234
•  “SETNCHAR Method” on page 1239
•  “SETNVARCHAR Method” on page 1241
•  “GETVARCHAR Method” on page 1248

ISPREPARED Method
Returns a value that indicates whether the SQLSTMT package instance is prepared.
Syntax

`package.ISPREPARED();`

Arguments

`package`

specifies an instance of the SQLSTMT package.

Details

The ISPREPARED method returns a value of 0 (false) if the SQLSTMT package instance is not prepared. A nonzero value (true) is returned if the SQLSTMT package instance is prepared.

See Also

Methods:

- “PREPARE Method” on page 1230

Statements:

- “DECLARE PACKAGE Statement, SQLSTMT Package” on page 1200

_NEW_ Operator, SQLSTMT Package

Constructs an instance of an SQLSTMT package.

Note: The escape character (\) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1:

`package-variable=_NEW_\[\{THIS\}\ | \{package-instance\}\] SQLSTMT
\(\text{\textbackslash sql-text} \ [\{\text{\textbackslash parameter-variable-list}\}]\);`

Form 2:

`package-variable=_NEW_\[\{THIS\}\ | \{package-instance\}\] SQLSTMT
\(\text{\textbackslash sql-text} \ [, \text{connection-string}]\);`

Form 3:

`package-variable=_NEW_\[\{THIS\}\ | \{package-instance\}\] SQLSTMT
\(d);`

Arguments

`package-variable`

specifies a name that can reference an instance of the SQLSTMT package.

`\{THIS\}`

specifies that the package instance has global scope.

See “Packages and Scope” on page 137
specifies that the new package instance has the same scope as `package-instance`. `package-instance` must be an existing package instance, and the type of `package-instance` can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

'sql-text'

is a valid FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.

**Requirement**  
The FedSQL statement must be enclosed in single quotation marks (').

**Notes**  
The statement can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

The rules for identifiers for the FedSQL language apply to variables used in the SQLSTM package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

[parameter-variable-list]

specifies variables that are bound to the parameters contained in the FedSQL statement.

**Requirements**  
Variables must be in the form of a variable list that must be enclosed in brackets ([[]]) or a named variable list.

Parameter values must be specified exclusively with either bound variables or exclusively with the SQLSTM SETtype methods.

See “Specifying FedSQL Statement Parameter Values” on page 174

connection-string

contains the fully specified connection string.

**Default**  
If a connection string is not provided, the SQLSTM package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

**Note**  
The connection string can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

**Tip**  
A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

See This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see the SAS Federation Server: Administrator's Guide.

**Details**

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.
You use an SQLSTMT package to create and delete tables, select rows from a table, and access the returned result set. The SQLSTMT package is predefined for DS2 programs.

You can declare an SQLSTMT package by using the DECLARE PACKAGE statement. This associates an SQLSTMT package with an SQLSTMT name.

There are two ways to construct an instance of an SQLSTMT package.

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:
  
  ```
  declare package sqlstmt sqlpkg;
  sqlpkg = _NEW_ sqlstmt('update db2.dataset2 set y=? where x=?', [y x]);
  ```

- Use the DECLARE PACKAGE statement along with its constructor syntax:
  
  ```
  declare package sqlstmt sqlpkg('update db2.dataset2 set y=? where x=?', [y x]);
  ```

When an SQLSTMT instance is created with SQL text, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package’s SET type methods. The DECLARE PACKAGE statement binds the variables in the optional variable list to the parameters in the FedSQL statement.

**Note:** Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

**See Also**

- “Using the SQLSTMT Package” on page 172
- “Package Constructors and Destructors” on page 142

**Methods:**

- “BINDPARAMETERS Method” on page 1198

**Statements:**

- “DECLARE PACKAGE Statement, SQLSTMT Package” on page 1200

---

**PREPARE Method**

Prepares a FedSQL statement.

**Syntax**

Form 1:  
PREPARE ('sql-text');

Form 2:  
PREPARE ('sql-text', connection-string);

**Arguments**

'sql-text' 

is a valid FedSQL statement or string variable that contains a FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.
Requirement  The FedSQL statement must be enclosed in single quotation marks ('') unless the statement is stored in a string variable.

Notes  The statement is a string literal.

The rules for identifiers for the FedSQL language apply to variables that are used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL (not DS2) parses the string containing the FedSQL statement.

connection-string  contains the fully specified connection string.

Default  If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

Note  The connection string is a string literal.

Tip  A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

See  This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see *SAS Federation Server: Administrator's Guide*.

Details  The PREPARE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

A run-time error occurs if you call the PREPARE method and the FedSQL statement is already prepared.

See Also  

Methods:  
• “ISPREPARED Method” on page 1227

Statements:  
• “DECLARE PACKAGE Statement, SQLSTMT Package” on page 1200

**SETBIGINT Method**  
Sets the designated parameter to the specified value of type BIGINT.

**Syntax**  

```plaintext
package.SETBIGINT (index, value);
```
Arguments

package
  specifies an instance of the SQLSTMT package.

index
  specifies the parameter index ordered sequentially, starting at 1.

value
  specifies the value to which to set the designated parameter. The designated
  parameter is specified by index.

Tip  value can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and
prepared. If the FedSQL statement contains parameters, values to substitute for the
parameters must be obtained to execute the FedSQL statement. The substitution values

If the designated parameter’s type is not type BIGINT, the BIGINT value is converted to
the designated parameter's type. For example, if you use setbigint(1, 3) to set
parameter 1 to BIGINT value 3, and parameter 1 is type CHAR, the BIGINT value 3 is
converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with
the SETtype methods. A run-time error results if variables are bound to parameters and
the SETBIGINT method is invoked.

The SETBIGINT method returns a status indicator. Zero is returned for successful
execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETBIGINT Method” on page 1204
- “GETINTEGER Method” on page 1214
- “GETSMALLINT Method” on page 1220
- “GETTINYINT Method” on page 1224
- “SETBIGINT Method” on page 1238
- “SETSMALLINT Method” on page 1243
- “SETTINYINT Method” on page 1246

SETBINARY Method

Sets the designated parameter to the specified value of type BINARY.
Syntax

```
package.SETBINARY (index, value);
```

**Arguments**

*package*

  specifies an instance of the SQLSTMT package.

*index*

  specifies the parameter index ordered sequentially, starting at 1.

*value*

  specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

  **Tip** *value* can be a literal, variable, or expression.

**Details**

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type BINARY, the BINARY value is converted to the designated parameter's type. For example, if you use `setbinary(1, 0110)` to set parameter 1 to BINARY value 0110, and parameter 1 is type CHAR, the BINARY value 0110 is converted to the CHAR value 0110 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETBINARY method is invoked.

The SETBINARY method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods:**

- “GETBINARY Method” on page 1206
- “GETVARBINARY Method” on page 1225
- “SETVARBINARY Method” on page 1247
SETCHAR Method

Sets the designated parameter to the specified value of type CHAR.

Syntax

```plaintext
package.SETCHAR(index, value);
```

Arguments

- `package` specifies an instance of the SQLSTMT package.
- `index` specifies the parameter index ordered sequentially, starting at 1.
- `value` specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

Tip  `value` can be a literal, variable, or an expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type CHAR, the CHAR value is converted to the designated parameter's type. For example, if you use `setchar(1, 3)` to set parameter 1 to CHAR value 3, and parameter 1 is type INTEGER, the CHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETCHAR method is invoked.

The SETCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETCHAR Method” on page 1207
- “GETNCHAR Method” on page 1215
- “GETNVARCHAR Method” on page 1217
SETDATE Method

Sets the designated parameter to the specified value of type DATE.

Syntax

```
package.SETDATE (index, value);
```

Arguments

```
package
    specifies an instance of the SQLSTMT package.

index
    specifies the parameter index ordered sequentially, starting at 1.

value
    specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.
```

Tip  `value` can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s `SETtype` methods.

If the designated parameter's type is not type DATE, the DATE value is converted to the designated parameter's type.

Parameter data must be specified exclusively with bound variables or exclusively with the `SETtype` methods. A run-time error results if variables are bound to parameters and the SETDATE method is invoked.

The SETDATE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETDATE Method” on page 1210
SETDECIMAL Method

Sets the designated parameter to the specified value of type DECIMAL.

Syntax

```
package.SETDECIMAL (index, value);
```

Arguments

- `package` specifies an instance of the SQLSTMT package.
- `index` specifies the parameter index ordered sequentially, starting at 1.
- `value` specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

Tip: `value` can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SET type methods.

If the designated parameter's type is not type DECIMAL, the DECIMAL value is converted to the designated parameter's type. For example, if you use `setdecimal(1, 13.4)` to set parameter 1 to DECIMAL value 13.4, and parameter 1 is type CHAR, the DECIMAL value 13.4 is converted to the CHAR value 13.4 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET type methods. A run-time error results if variables are bound to parameters and the SETDECIMAL method is invoked.

The SETDECIMAL method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172
Methods:

• “GETDECIMAL Method” on page 1211

SETDOUBLE Method

Sets the designated parameter to the specified value of type DOUBLE.

Syntax

\[ \text{package.SETDOUBLE (index, value);} \]

Arguments

\- \text{package} specifies an instance of the SQLSTMT package.
\- \text{index} specifies the parameter index ordered sequentially, starting at 1.
\- \text{value} specifies the value to which to set the designated parameter. The designated parameter is specified by \text{index}.

Tip \text{value} can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type DOUBLE, the DOUBLE value is converted to the designated parameter's type. For example, if you use \text{setdouble(1, 33443452)} to set parameter 1 to DOUBLE value 33443452, and parameter 1 is type CHAR, the DOUBLE value 33443452 is converted to the CHAR value 33443452 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETDOUBLE method is invoked.

The SETDOUBLE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “GETDOUBLE Method” on page 1213
SETINTEGER Method
Sets the designated parameter to the specified value of type INTEGER.

Syntax

package.SETINTEGER (index, value);

Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by index.

Tip value can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type INTEGER, the INTEGER value is converted to the designated parameter's type. For example, if you use setinteger(1, 3) to set parameter 1 to INTEGER value 3, and parameter 1 is type CHAR, the INTEGER value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETINTEGER method is invoked.

The SETINTEGER method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “GETBIGINT Method” on page 1204
• “GETINTEGER Method” on page 1214
• “GETSMALLINT Method” on page 1220
SETNCHAR Method

Sets the designated parameter to the specified value of type NCHAR.

Syntax

\[
\text{package}.\text{SETNCHAR} (index, value);
\]

Arguments

- **package**
  - specifies an instance of the SQLSTMT package.
- **index**
  - specifies the parameter index ordered sequentially, starting at 1.
- **value**
  - specifies the value to which to set the designated parameter. The designated parameter is specified by \( \text{index} \).

  **Tip** \( \text{value} \) can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter’s type is not type NCHAR, the NCHAR value is converted to the designated parameter type. For example, if you use \( \text{setnchar} (1, 3) \) to set parameter 1 to NCHAR value 3, and parameter 1 is type INTEGER, the NCHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETNCHAR method is invoked.

The SETNCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172
**Methods:**

- “GETCHAR Method” on page 1207
- “GETNCHAR Method” on page 1215
- “GETNVARCHAR Method” on page 1217
- “GETVARCHAR Method” on page 1226
- “SETCHAR Method” on page 1234
- “SETNVARCHAR Method” on page 1241
- “SETVARCHAR Method” on page 1248

**SETNUMERIC Method**

Sets the designated parameter to the specified value of type NUMERIC.

**Syntax**

```sql
package.SETNUMERIC (index, value);
```

**Arguments**

- `package` specifies an instance of the SQLSTMT package.
- `index` specifies the parameter index ordered sequentially, starting at 1.
- `value` specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

**Tip** `value` can be a literal, variable, or expression.

**Details**

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type NUMERIC, the NUMERIC value is converted to the designated parameter's type. For example, if you use `setnumeric(1, 3)` to set parameter 1 to NUMERIC value 3, and parameter 1 is type CHAR, the NUMERIC value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETNUMERIC method is invoked.

The SETNUMERIC method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.
For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

**See Also**

- “Using the SQLSTMT Package” on page 172

**Methods**

- “GETNUMERIC Method” on page 1216

---

**SETNVARCHAR Method**

Sets the designated parameter to the specified value of type NVARCHAR.

**Syntax**

```
package.SETNVARCHAR (index, value);
```

**Arguments**

- `package`
  - specifies an instance of the SQLSTMT package.
- `index`
  - specifies the parameter index ordered sequentially, starting at 1.
- `value`
  - specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

**Tip** `value` can be a literal, variable, or expression.

**Details**

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type NVARCHAR, the NVARCHAR value is converted to the designated parameter's type. For example, if you use `setnvarchar(1, 3)` to set parameter 1 to NVARCHAR value 3, and parameter 1 is type INTEGER, the NVARCHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETNVARCHAR method is invoked.

The SETNVARCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.
SETREAL Method

Sets the designated parameter to the specified value of type REAL.

Syntax

```sql
package.SETREAL (index, value);
```

Arguments

- `package` specifies an instance of the SQLSTMT package.
- `index` specifies the parameter index ordered sequentially, starting at 1.
- `value` specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

Tip: `value` can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type REAL, the REAL value is converted to the designated parameter's type. For example, if you use `setreal(1, 3)` to set parameter 1 to REAL value 3, and parameter 1 is type CHAR, the REAL value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETREAL method is invoked.
The SETREAL method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “GETREAL Method” on page 1219

SETSMALLINT Method

Sets the designated parameter to the specified value of type SMALLINT.

Syntax

package.SETSMALLINT(index, value);

Arguments

package

specifies an instance of the SQLSTMT package.

index

specifies the parameter index ordered sequentially, starting at 1.

value

specifies the value to which to set the designated parameter. The designated parameter is specified by index.

Tip value can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type SMALLINT, the SMALLINT value is converted to the designated parameter's type. For example, if you use setsmallint(1, 3) to set parameter 1 to SMALLINT value 3, and parameter 1 is type CHAR, the SMALLINT value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETSMALLINT method is invoked.

The SETSMALLINT method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.
For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also
- “Using the SQLSTMT Package” on page 172

Methods:
- “GETBIGINT Method” on page 1204
- “GETINTEGER Method” on page 1214
- “GETSMALLINT Method” on page 1220
- “GETTINYINT Method” on page 1224
- “SETBIGINT Method” on page 1231
- “SETINTEGER Method” on page 1238
- “SETTINYINT Method” on page 1246

SETTIME Method

Sets the designated parameter to the specified value of type TIME.

Syntax

package.SETTIME (index, value);

Arguments

package
specifies an instance of the SQLSTMT package.
index
specifies the parameter index ordered sequentially, starting at 1.
value
specifies the value to which to set the designated parameter. The designated parameter is specified by index.

Tip  value can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type TIME, the TIME value is converted to the designated parameter's type.
Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETTIME method is invoked.

The SETTIME method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETDATE Method” on page 1210
- “GETTIME Method” on page 1221
- “GETTIMESTAMP Method” on page 1222
- “SETDATE Method” on page 1235
- “SETTIMESTAMP Method” on page 1245

SETTIMESTAMP Method

Sets the designated parameter to the specified value of type TIMESTAMP.

Syntax

package.SETTIMESTAMP (index, value);

Arguments

package

specifies an instance of the SQLSTMT package.

index

specifies the parameter index ordered sequentially, starting at 1.

value

specifies the value to which to set the designated parameter. The designated parameter is specified by index.

Tip  value can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type TIMESTAMP, the TIMESTAMP value is converted to the designated parameter's type.
Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETTIMESTAMP method is invoked.

The SETTIMESTAMP method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

• “Using the SQLSTMT Package” on page 172

Methods:

• “GETDATE Method” on page 1210
• “GETTIME Method” on page 1221
• “GETTIMESTAMP Method” on page 1222
• “SETDATE Method” on page 1235
• “SETTIME Method” on page 1244

SETTINYINT Method

Sets the designated parameter to the specified value of type TINYINT.

Syntax

\[ \text{package}}\text{.SETTINYINT (index, value);} \]

Arguments

\[ \text{package} \]

specifies an instance of the SQLSTMT package.

\[ \text{index} \]

specifies the parameter index ordered sequentially, starting at 1.

\[ \text{value} \]

specifies the value to which to set the designated parameter. The designated parameter is specified by \text{index}.

Tip\[ \text{value} \] can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type TINYINT, the TINYINT value is converted to the designated parameter's type. For example, if you use \text{settinyint}(1, 3) to set
Parameter 1 to TINYINT value 3, and parameter 1 is type CHAR, the TINYINT value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETTINYINT method is invoked.

The SETTINYINT method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

**See Also**
- “Using the SQLSTMT Package” on page 172

**Methods:**
- “GETBIGINT Method” on page 1204
- “GETINTEGER Method” on page 1214
- “GETSMALLINT Method” on page 1220
- “GETTINYINT Method” on page 1224
- “SETBIGINT Method” on page 1231
- “SETINTEGER Method” on page 1238
- “SETSMALLINT Method” on page 1243

---

**SETVARBINARY Method**

Sets the designated parameter to the specified value of type VARBINARY.

### Syntax

```
package.SETVARBINARY(index, value);
```

### Arguments

- `package`
  - specifies an instance of the SQLSTMT package.

- `index`
  - specifies the parameter index ordered sequentially, starting at 1.

- `value`
  - specifies the value to which to set the designated parameter. The designated parameter is specified by `index`.

  **Tip** `value` can be a literal, variable, or expression.

### Details

When an SQLSTMT instance is created, the FedEx SQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the
parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter's type is not type VARBINARY, the VARBINARY value is converted to the designated parameter's type. For example, if you use setvarbinary(1, 0110) to set parameter 1 to VARBINARY value 0110, and parameter 1 is type CHAR, the VARBINARY value 0110 is converted to the CHAR value 0110 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETVARBINARY method is invoked.

The SETVARBINARY method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTMT Package” on page 172

Methods:

- “GETBINARY Method” on page 1206
- “GETVARBINARY Method” on page 1225
- “SETBINARY Method” on page 1232

SETVARCHAR Method

Sets the designated parameter to the specified value of type VARCHAR.

Syntax

```
package.SETVARCHAR(index, value);
```

Arguments

- **package**
  - specifies an instance of the SQLSTMT package.
- **index**
  - specifies the parameter index ordered sequentially, starting at 1.
- **value**
  - specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

  Tip  value can be a literal, variable, or expression.
Details

When an SQLSTM instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package’s SETtype methods.

If the designated parameter’s type is not type VARCHAR, the VARCHAR value is converted to the designated parameter's type. For example, if you use `setvarchar(1, pass)` to set parameter 1 to VARCHAR value `pass`, and parameter 1 is type CHAR, the VARCHAR value `pass` is converted to the CHAR value `pass` and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SETtype methods. A run-time error results if variables are bound to parameters and the SETVARCHAR method is invoked.

The SETVARCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see “Specifying FedSQL Statement Parameter Values” on page 174.

See Also

- “Using the SQLSTM Package” on page 172

Methods:

- “GETCHAR Method” on page 1207
- “GETNCHAR Method” on page 1215
- “GETNVARCHAR Method” on page 1217
- “GETVARCHAR Method” on page 1226
- “SETCHAR Method” on page 1234
- “SETNCHAR Method” on page 1239
- “SETNVARCHAR Method” on page 1241
**Chapter 37**

**DS2 TZ Package Methods, Operators, and Statements**

---

**Dictionary**

---

**DECLARE PACKAGE Statement, TZ Package**

Creates a package variable and enables you to create an instance of the TZ package.

**Category:** Local

**Syntax**

```
DECLARE PACKAGE TZ variable ([time-zone-id]);
```

**Arguments**

- **variable**
  - specifies a name that can reference an instance of the TZ package.

- **time-zone-id**
  - specifies a time zone ID.

**Default**

The value specified in the TIMEZONE= system option.
Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a TZ package for time zone processing. The TZ package is predefined for DS2 programs. For more information about time zones in SAS, see SAS National Language Support (NLS): Reference Guide.

You declare a TZ package by using the DECLARE PACKAGE statement. This associates a TZ package with a time zone name. After you declare the new TZ package, you can format your date and time data accordingly.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a TZ package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:
  ```
declare package tz tzpkg;
tzpkg = _new_ tz();
  ```
- Use the DECLARE PACKAGE statement along with its constructor syntax:
  ```
declare package tz tzpkg();
  ```

See Also

- “Using the TZ Package” on page 176
- “Package Constructors and Destructors” on page 142

Operators:

- “_NEW_ Operator, TZ Package” on page 1259

System Options:

- “TIMEZONE= System Option” in SAS System Options: Reference

GETLOCALTIME Method

Returns current local time.

Syntax

```variable=package.GETLOCALTIME ([time-zone-ID]);
```

Arguments

- `variable` specifies the variable that will hold the value of the time zone ID of required local time.
**package**  
specifies an instance of the TZ package.

**time-zone-ID**  
specifies the time zone ID of the required local time.

**Example**

The following example uses multiple time zones.

```sas
data _null_;  
method init();

    declare package tz tokyo('Asia/Tokyo')
        london('Europe/London')
        new_york('America/New_York');
dcl double tokyo_time london_time new_york_time utc_time;
dcl integer tokyo_off london_off new_york_off;

tokyo_time = tokyo.getLocalTime();
tokyo_off = tokyo.getOffset();
london_time = london.getLocalTime();
london_off = london.getOffset();
new_york_time = new_york.getLocalTime();
new_york_off = new_york.getOffset();

utc_time = tokyo.getUTCTime(); /* can use any timezone */

    put utc_time = datetime. ;
    put tokyo_time = datetime. tokyo_off time5.;
    put london_time = datetime. london_off time5. ;
    put new_york_time = datetime. new_york_off time5. ;

end;
enddata;
run;
```

The following lines are written to the SAS log.

```
utc_time=08APR15:12:48:41
tokyo_time=08APR15:21:48:41  9:00
london_time=08APR15:23:48:41  1:00
new_york_time=08APR15:08:48:41 -4:00
```

**See Also**

- “Using the TZ Package” on page 176

**Methods:**

- “GETUTCTIME Method” on page 1259
GETOFFSET Method

Returns the time zone offset of the time zone from Universal Coordinated Time (UTC) at the specified local time. If local time is not specified, current local time is used.

Syntax

Form 1:  \texttt{variable=package.GETOFFSET ( );}
Form 2:  \texttt{variable=package.GETOFFSET (local-time);}
Form 3:  \texttt{variable=package.GETOFFSET (time-zone-ID);}
Form 4:  \texttt{variable=package.GETOFFSET (local-time, time-zone-ID);}

Arguments

\texttt{variable}  
specifies the variable that will hold the value of the time zone offset.

\texttt{package}  
specifies an instance of the TZ package.

\texttt{local-time}  
specifies the local time used to get the time zone offset.

\texttt{time-zone-ID}  
specifies the time zone ID of the required time zone offset.

\texttt{localTime}  
is a SAS date time value. It is used as the number of seconds since January 1, 1960 00:00:00 local time.

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form \texttt{yyyyymmddThhmnnss+\textminus hhmm} or the ISO 8601 extended form \texttt{yyyy-mm-ddThh:mm:ss+\textminus hh:mm}.

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form \texttt{+\textminus hhmm} or \texttt{+\textminus hh:mm}.

Example

The following example returns the offset from the 'Asia/Tokyo' time zone to the 'America/New_York' time zone. The example also illustrates the different ways in which the time zone ID can be expressed.

data _null_ ;
  method init();
  declare package tz tzone('asia/tokyo') ;
dcl double new_york;
dcl char(40) cstr;

new_york = tzone.getOffset('America/New_York');
put new_york time.;

new_york = tzone.getOffset(n'America/New_York');
put new_york time.;

cstr = 'America/New_York';
new_york = tzone.getOffset(cstr);
put new_york time.;

end;
enddata;
run;

The following lines are written to the SAS log.

-4:00:00
-4:00:00
-4:00:00

See Also

- “Using the TZ Package” on page 176

Methods:

- “GETOFFSETUTC Method” on page 1255

---

**GETOFFSETUTC Method**

Returns the time zone offset of the time zone from UTC at the specified UTC time.

**Syntax**

```
variable=package.GETOFFSETUTC (UTC-time, time-zone-ID);
```

**Without Arguments**

If no arguments are specified, the GETOFFSETUTC method returns the time zone offset for the specified TIMEZONE= system option.

**Arguments**

- `variable`
  - specifies the variable that will hold the value of the time zone offset.
- `package`
  - specifies an instance of the TZ package.
- `UTC-time`
  - specifies the UTC time used to get the time zone offset.
Tip  
UTC-time is a SASdatetime value at UTC. It is stored as the number of seconds since January 1, 1960 00:00:00 at UTC.

time-zone-ID
specifies the time zone ID of the required time zone offset.


Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form \( yyyyymmddThhmmss+|-hhmm \) or the ISO 8601 extended form \( yyyy-mm-ddThh:mm:ss+|-hh:mm \).

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form \( +|-hhmm \) or \( +|-hh:mm \).

See Also

• “Using the TZ Package” on page 176

Methods:

• “GETOFFSET Method” on page 1254

GETTIMEZONEID Method

Returns the current time zone ID.

Syntax

\[
\text{variable} = \text{package.GETTIMEZONEID ( );}
\]

Arguments

package  
specifies an instance of the TZ package.

variable  
specifies the variable that will hold the value of the time zone ID of the TZ package instance.

Details

The time zone ID specifies a region or area value that is defined by SAS. For more information about time zone IDs, see SAS National Language Support (NLS): Reference Guide.

Example

The following example uses the TZ package to calculate time durations.

\[
\text{options timezone='asia/tokyo'; /* TIMEZONE ID of origin */}
\]
proc ds2 ;
data _null_ ;
method route(package tz origin,
  package tz dest,
  timestamp departure,
  time duration) ;

dcl nvarchar(50) tzid dest_tzid;
dcl nvarchar(8)  tzname dest_tzname home_tzn ;
dcl double dept dur arrival utc ;
dcl double home_dept home_arr ;

declare package tz home() ;

utc = origin.toUTCTime(departure) ;
dur = TO_DOUBLE(duration) ;
arrival = dest.toLocalTime(utc+dur) ;
tzid = origin.getTimezoneID() ;
tzname = origin.getTimezoneName() ;
dest_tzid = dest.getTimezoneID() ;
dest_tzname = dest.getTimezoneName() ;

home_dept = home.toLocalTime(utc) ;
home_arr = home.toLocalTime(utc+dur);
home_tzn = home.getTimezoneName() ;

put 'Time Zone: ' tzid 'to' dest_tzid ;
put 'Departure Time: ' departure datetime.  tzname  '/'
  home_dept datetime. home_tzn;
put '   Arrial Time: ' arrival datetime.  dest_tzname  '/'
  home_arr datetime. home_tzn ;
put;

end ;
method init();

/* print itinerary */
declare package tz NRT('Asia/Tokyo') ;
declare package tz ORD('America/Chicago') ;
declare package tz RDU('America/New_York');
route(NRT,ORD,timestamp '2014-10-19 10:45:00',time '11:35:00') ;
route(ORD,RDU,timestamp '2014-10-19 11:03:00',time '01:56:00') ;
route(RDU,ORD,timestamp '2014-10-25 07:45:00',time '02:02:00') ;
route(ORD,NRT,timestamp '2014-10-25 10:50:00',time '12:55:00') ;

end;
enddata;
run;
quit;
The following lines are written to the SAS log.

| Time Zone: Asia/Tokyo to America/Chicago | Departure Time: 19OCT14:10:45:00JST / 19OCT14:10:45:00JST | Arrival Time: 19OCT14:08:20:00CDT / 19OCT14:22:20:00JST |
| Time Zone: America/Chicago to America/New York | Departure Time: 19OCT14:11:03:00CDT / 20OCT14:01:03:00JST | Arrival Time: 19OCT14:13:59:00EDT / 20OCT14:02:59:00JST |
| Time Zone: America/New_York to America/Chicago | Departure Time: 25OCT14:07:45:00EDT / 25OCT14:20:45:00JST | Arrival Time: 25OCT14:08:47:00CDT / 25OCT14:22:47:00JST |
| Time Zone: America/Chicago to Asia/Tokyo | Departure Time: 25OCT14:10:50:00CDT / 26OCT14:00:50:00JST | Arrival Time: 26OCT14:13:45:00JST / 26OCT14:13:45:00JST |

See Also

- “Using the TZ Package” on page 176

Methods:

- “GETTIMEZONENAME Method” on page 1258

---

**GETTIMEZONENAME Method**

Returns the current time zone name.

**Syntax**

```
variable=package.GETTIMEZONENAME();
```

**Arguments**

- `package`
  - specifies an instance of the TZ package.
- `variable`
  - specifies the variable that will hold the value of the time zone name of the TZ package instance.

**Details**

The time zone name specifies a region or area value that is defined by SAS. For more information about time zone names, see SAS National Language Support (NLS): Reference Guide.

**Example**

See the example in “GETTIMEZONEID Method” on page 1256.
GETUTCTIME Method

Returns the current UTC time.

Syntax

\[
\text{variable} = \text{package}.\text{GETUTCTIME}() ;
\]

Arguments

\( \text{variable} \)

specifies the variable that will hold the value of the current UTC time.

\( \text{package} \)

specifies an instance of the TZ package.

See Also

• “Using the TZ Package” on page 176

Methods:

• “GETTIMEZONEID Method” on page 1256

_NEW_ Operator, TZ Package

Constructs an instance of a TZ package.

Note: The escape character ( \ ) before the bracket indicates that the bracket is required in
the syntax.

Syntax

\[
\text{package-variable} = \_\text{NEW}_\ [\{\text{THIS}\} | \{\text{package-instance}\}] \text{TZ}([\text{time-zone-id}]) ;
\]

Arguments

\( \text{package-variable} \)

specifies a name that can reference an instance of the package.

\( \{\text{THIS}\} \)

specifies that the package instance has global scope.
See “Packages and Scope” on page 137

[package-instance]
specifies that the new package instance has the same scope as package-instance.
package-instance must be an existing package instance, and the type of package-instance can differ from the type of the new package instance.

See “Package-Specific Scope” on page 139

time-zone-id
specifies a time zone ID.

Default The value specified in the TIMEZONE= system option.

Details
A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a TZ package is declared, the variable representing the package can be considered an instance of the package. This means that two different package variables represent two completely separate copies of a package.

You declare a TZ package using the DECLARE PACKAGE statement. After you declare the new TZ package, use the _NEW_ operator to instantiate the package.

```
declare package tz localtz;
localtz = _new_ tz( );
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a TZ package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package tz localtz( );
```

Note: Package variables are subject to all variable scoping rules. For more information, see “Packages and Scope” on page 137.

See Also

- “Using the TZ Package” on page 176
- “Package Constructors and Destructors” on page 142

Statements:

- “DECLARE PACKAGE Statement, TZ Package” on page 1251

TOISO8601 Method

Converts local time to an ISO8601 string with time zone offset.

Syntax

Form 1: variable=package.TOISO8601 (local-time);
Form 2: \[ \text{variable}=\text{package}.\text{TOISO8601} \left( \text{local-time, time-zone-ID} \right); \]

**Arguments**

\textit{variable}

specifies the variable that will hold the value of an ISO8601 string such as '2014-10-10T00:01:02.00+09:00'.

\textit{package}

specifies an instance of the TZ package.

\textit{local-time}

specifies the local time to convert. \textit{local-time} can be DOUBLE or TIMESTAMP format.

\textit{time-zone-ID}

specifies the time zone ID of the required local time. Time zone ID 'UTC' can be used to specify UTC time.

**See Also**

- “Using the TZ Package” on page 176

**Methods:**

- “TOLOCALTIME Method” on page 1261
- “TOTIMESTAMPZ Method” on page 1262
- “TOUTCTIME Method” on page 1263

---

**TOLOCALTIME Method**

Converts UTC time to local time.

**Syntax**

Form 1: \[ \text{variable}=\text{package}.\text{TOLOCALTIME} \left( \text{UTC-time} \right); \]
Form 2: \[ \text{variable}=\text{package}.\text{TOLOCALTIME} \left( \text{UTC-time, time-zone-ID} \right); \]

**Arguments**

\textit{variable}

specifies the variable that will hold the value of the local time that is converted from the specified UTC time.

\textit{package}

specifies an instance of the TZ package.

\textit{UTC-time}

specifies the current UTC time in DOUBLE or TIMESTAMP format.

\textit{time-zone-ID}

specifies the time zone ID of the required local time.
Details
UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a
datetime value that uses the ISO 8601 basic form yyyymmddThhmmss+|–hhmm or the
ISO 8601 extended form yyyy-mm-ddThh:mm:ss+|–hh:mm.

Example
See the example in “GETTIMEZONEID Method” on page 1256.

See Also
• “Using the TZ Package” on page 176
• “Time Zone IDs and Time Zone Names” in SAS National Language Support (NLS):
  Reference Guide

Methods:
• “TOISO8601 Method” on page 1260
• “TOTIMESTAMPZ Method” on page 1262
• “TOUTCTIME Method” on page 1263

TOTIMESTAMPZ Method
Converts local time to a TIMESTAMP string with time zone.

Syntax
variable=package.TOTIMESTAMPZ (local-time[, time-zone-ID]);

Arguments
variable
  specifies the variable that will hold the value of a string such as '2014-10-14
  00:01:20 Asia/Tokyo'.

package
  specifies an instance of the TZ package.

local-time
  specifies the local time to convert. local-time can be DOUBLE or TIMESTAMP
  format.

time-zone-ID
  specifies the time zone ID of the required local time. Time zone ID 'UTC' can be
  specified to use UTC time.

See Also
• “Using the TZ Package” on page 176
• “Time Zone IDs and Time Zone Names” in SAS National Language Support (NLS):
  Reference Guide
TOUTCTIME Method

Converts local time to UTC time.

Syntax

Form 1:  \( \text{variable} = \text{package}.\text{TOUTCTIME} (\text{local-time}); \)

Form 2:  \( \text{variable} = \text{package}.\text{TOUTCTIME} (\text{local-time}, \text{time-zone-ID}); \)

Arguments

\( \text{variable} \)

specifies the variable that will hold the value of the current UTC time in DOUBLE or TIMESTAMP format.

\( \text{package} \)

specifies an instance of the TZ package.

\( \text{local-time} \)

specifies the local time to convert. \( \text{local-time} \) can be DOUBLE or TIMESTAMP format.

\( \text{time-zone-ID} \)

specifies the time zone ID of the required local time.

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form \( yyyy-mm-ddThh:mm:ssZ \) or the ISO 8601 extended form \( yyyy-mm-ddThh:mm:ssZ\pm hh:mm \).

Example

See the example in “GETTIMEZONEID Method” on page 1256.

See Also

- “Using the TZ Package” on page 176
• “TOUTCTIME Method” on page 1263
Part 5

Appendixes

Appendix 1
Data Type Reference .................................................. 1267

Appendix 2
DS2 Type Conversions for Expression Operands ............... 1299

Appendix 3
DS2 Loggers ................................................................. 1301

Appendix 4
DS2 Example Programs .................................................. 1305
# Data Type Reference

## Data Types for SAS Data Sets

The following table lists the data type support for a SAS data set.

The BINARY and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data

<table>
<thead>
<tr>
<th>Data Types for SAS Data Sets</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Types for SPD Engine Data Sets</td>
<td>1269</td>
</tr>
<tr>
<td>Data Types for Aster</td>
<td>1271</td>
</tr>
<tr>
<td>Data Types for DB2 under UNIX and PC Hosts</td>
<td>1272</td>
</tr>
<tr>
<td>Data Types for Greenplum</td>
<td>1273</td>
</tr>
<tr>
<td>Data Types for HAWQ</td>
<td>1274</td>
</tr>
<tr>
<td>Data Types for HDMD</td>
<td>1276</td>
</tr>
<tr>
<td>Data Types for Hive</td>
<td>1277</td>
</tr>
<tr>
<td>Data Types for Impala</td>
<td>1279</td>
</tr>
<tr>
<td>Data Types for MDS</td>
<td>1280</td>
</tr>
<tr>
<td>Data Types for MySQL</td>
<td>1282</td>
</tr>
<tr>
<td>Data Types for Netezza</td>
<td>1283</td>
</tr>
<tr>
<td>Data Types for ODBC</td>
<td>1285</td>
</tr>
<tr>
<td>Data Types for Oracle</td>
<td>1286</td>
</tr>
<tr>
<td>Data Types for PostgreSQL</td>
<td>1288</td>
</tr>
<tr>
<td>Data Types for SAP</td>
<td>1290</td>
</tr>
<tr>
<td>Data Types for SAP HANA</td>
<td>1292</td>
</tr>
<tr>
<td>Data Types for SASHDAT</td>
<td>1293</td>
</tr>
<tr>
<td>Data Types for Sybase IQ</td>
<td>1294</td>
</tr>
<tr>
<td>Data Types for Teradata</td>
<td>1296</td>
</tr>
</tbody>
</table>
source-specific information about the SAS numeric and SAS character data types, see
\textit{SAS Language Reference: Concepts}.  

\textbf{Table A1.1} \ Data Types for SAS Data Sets

<table>
<thead>
<tr>
<th>Data Type Definition Keyword\footnote*</th>
<th>SAS Data Set Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. \textit{Note:} There is potential for loss of precision.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string. \textit{Note:} Cannot contain ANSI SQL null values.</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>DATE ***</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the \textit{DATE9} SAS format.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>\textbackslash{}</td>
<td>NUMERIC((p,s))**</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>NCHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string. By default, sets the encoding to Unicode UTF-8. \textsuperscript{†}</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>NVARCHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string. By default, sets the encoding to Unicode UTF-8. \textsuperscript{†}</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>REAL**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>SMALLINT**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>TIME((p))***</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the \textit{TIME8} SAS format.</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
### Data Types for SPD Engine Data Sets

The following table lists the data type support for an SPD Engine data set.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

#### Table A1.2 Data Types for SPD Engine Data Sets

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>SPD Data Set Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT*</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> There is potential for loss of precision.</td>
<td></td>
</tr>
</tbody>
</table>

---

**Data Type Definition Keyword**

The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

**Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.**

***Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.***

† UTF-8 is an MBCS encoding. Depending on the operating environment, UTF-8 characters are of varying width, from one to four bytes. The value for $n$, which is the maximum number of multibyte characters to store, is multiplied by the maximum length for the operating environment. Note that when you are transcoding, such as from UTF-8 to WLatin2, the variable lengths (in bytes) might not be sufficient to hold the values, and the result is character data truncation.

---

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>SAS Data Set Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP($p$)***</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>TINYINT**</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>VARCHAR($n$)</td>
<td>CHAR($n$)</td>
<td>Fixed-length character string.</td>
<td>CHAR($n$)</td>
</tr>
</tbody>
</table>

**Note:** Cannot contain ANSI SQL null values.
<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>SPD Data Set Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
</table>
| CHAR(\(n\))                | CHAR(\(n\))           | Fixed-length character string.  
  \textit{Note:} Cannot contain ANSI SQL null values. | CHAR(\(n\)) |
| DATE **                     | DOUBLE                 | 64-bit double precision, floating-point number. By default, applies the DATE9 SAS format. | DOUBLE |
| DOUBLE*                     | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| FLOAT*                      | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| INTEGER*                    | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| REAL*                       | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| SMALLINT*                   | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| TIME(\(p\)) **             | DOUBLE                 | 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format. | DOUBLE |
| TIMESTAMP(\(p\)) **        | DOUBLE                 | 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format. | DOUBLE |
| TINYINT*                    | DOUBLE                 | 64-bit double precision, floating-point number. | DOUBLE |
| VARCHAR(\(n\))            | CHAR(\(n\))           | Fixed-length character string.  
  \textit{Note:} Cannot contain ANSI SQL null values. | CHAR(\(n\)) |

* Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

** Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.
The following table lists the data type support for an Aster database.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about Aster database data types, see the Aster database documentation.

### Table A1.3  Data Types for Aster

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Aster Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY((n))</td>
<td>BYTEA</td>
<td>Varying-length binary string.</td>
<td>BINARY((n))</td>
</tr>
<tr>
<td>*</td>
<td>BOOL</td>
<td>One byte integral data type that can contain values 0, 1, or NULL.</td>
<td>*</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string.</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC((p,s))</td>
<td>NUMERIC((p,s))</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT((p))</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT((p))</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Signed, single precision, floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME((p))</td>
<td>TIME((p))</td>
<td>Time value.</td>
<td>TIME((p))</td>
</tr>
<tr>
<td>TIMESTAMP((p))</td>
<td>TIMESTAMP((p))</td>
<td>Date and time value.</td>
<td>TIMESTAMP((p))</td>
</tr>
</tbody>
</table>
### Data Type Definition

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Aster Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARBINARY(n)</td>
<td>BYTEA</td>
<td>Varying-length binary string.</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>*</td>
<td>TEXT</td>
<td>Varying-length large character string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* The Aster data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

### Data Types for DB2 under UNIX and PC Hosts

The following table lists the data type support for a DB2 database under UNIX and PC hosts.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about the DB2 database data types, see the DB2 database documentation.

#### Table A1.4 Data Types for DB2 under UNIX and PC Hosts

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>DB2 Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>CHAR(n) FOR BIT DATA</td>
<td>Fixed-length binary string.</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>**</td>
<td>BLOB(n[K</td>
<td>M</td>
<td>G])</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>**</td>
<td>CLOB(n[K</td>
<td>M</td>
<td>G])</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>**</td>
<td>DBCLOB(n[K</td>
<td>M</td>
<td>G])</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
</tr>
<tr>
<td>Data Type Definition Keyword*</td>
<td>DB2 Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT(p)</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>**</td>
<td>GRAPHIC(n)</td>
<td>Fixed-length graphic string.</td>
<td>**</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>**</td>
<td>LONG VARCHAR [FOR BIT DATA]</td>
<td>Varying-length character or binary string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>LONG VARGRAPHIC(n)</td>
<td>Varying-length graphic string.</td>
<td>**</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Signed, single precision, floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>Varying-length binary string.</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>**</td>
<td>VARGRAPHIC(n)</td>
<td>Varying-length graphic string</td>
<td>**</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The DB2 data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

---

**Data Types for Greenplum**

The following table lists the data type support for a Greenplum database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.
For data source specific information about Greenplum data types, see the Greenplum database documentation.

**Table A1.5  Data Types for Greenplum**

<table>
<thead>
<tr>
<th>Data Type Definition</th>
<th>Greenplum Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>INT8</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL[NUMERIC(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>Signed, fixed-point decimal number.</td>
<td>DECIMAL[NUMERIC(p,s)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT(p)</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT8</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value in hours, minutes, and seconds.</td>
<td>TIME(p)**</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>BYTEA</td>
<td>Varying-length binary string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** Due to the ODBC-style interface that is used to communicate with the Greenplum server, fractional seconds are lost in the data transfer from server to client.

---

**Data Types for HAWQ**

The following table lists the data type support for a HAWQ database.
The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about HAWQ data types, see the HAWQ database documentation.

Table A1.6 Data Types for HAWQ

<table>
<thead>
<tr>
<th>Data Type Definition Keyword *</th>
<th>HAWQ Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>INT8</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string.</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC((p,s))</td>
<td>DECIMAL((p,s))</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT((p))</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT8</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME((p))</td>
<td>TIME((p))</td>
<td>Time value in hours, minutes, and seconds.</td>
<td>TIME((p))**</td>
</tr>
<tr>
<td>TIMESTAMP((p))</td>
<td>TIMESTAMP((p))</td>
<td>Date and time value.</td>
<td>TIMESTAMP((p))</td>
</tr>
<tr>
<td>VARBINARY((n))</td>
<td>BYTEA</td>
<td>Varying-length binary string.</td>
<td>VARBINARY((n))</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>VARCHAR((n))</td>
<td>Varying-length character string.</td>
<td>VARCHAR((n))</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** Due to the ODBC-style interface that is used to communicate with the HAWQ server, fractional seconds are lost in the data transfer from server to client.
The following table lists the data type support for HDMD.

The BINARY, NCHAR, and NVARCHAR data types are not supported for data type definition.

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>HDMD Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>A large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>A fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date value.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL/NUMERIC(p,s)</td>
<td>DECIMAL</td>
<td>A fixed-point decimal number, with 38 digits precision.</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
| DOUBLE                       | DOUBLE         | A signed, approximate, double-precision, floating-point number.  
*Note*: Supports SAS missing values | DOUBLE |
| FLOAT                        | DOUBLE         | A signed, approximate, double-precision, floating-point number.  
*Note*: Supports SAS missing values | DOUBLE |
<p>| INTEGER                      | INTEGER        | A regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,648 to 2,147,483,647. | INTEGER |</p>
<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>HDMD Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>A signed, approximate, single-precision, floating-point number. *Note: Supports SAS missing values.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>A small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME[(p)]</td>
<td>Time value with optional precision.</td>
<td>TIME[(p)]</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP[(p)]</td>
<td>Date and time value with optional precision.</td>
<td>TIMESTAMP[(p)]</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>A very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string data.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* Decimals processed by SAS are processed using a DOUBLE, which can alter the precision.

---

**Data Types for Hive**

The following table lists the data type support for Hive. Hive versions 0.10 and later are supported.

The NCHAR and NVARCHAR data types are not available for data definition. Nor are the Hive complex types ARRAY, MAP, STRUCT, and UNION.

The Hive complex data types can be read, beginning in the third maintenance release of SAS 9.4. The VARBINARY data type is also available for data definition beginning with the third maintenance release of SAS 9.4.

For data-source specific information about Hive data types, see the Hive database documentation.
<table>
<thead>
<tr>
<th>Data Type Definition</th>
<th>Hive Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>ARRAY&lt;$\text{data-type}$&gt;</td>
<td>An array of integers (indexable lists).</td>
<td>STRING‡</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY($n$)</td>
<td>BINARY**</td>
<td>A varying length binary string up to 32K.</td>
<td>BINARY</td>
</tr>
<tr>
<td>*</td>
<td>BOOLEAN</td>
<td>A textual true or false value.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>CHAR($n$)</td>
<td>CHAR($n$)**</td>
<td>A character string up to 255 characters. ***</td>
<td>CHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE†††</td>
<td>An ANSI SQL date type.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL/NUMERIC($p,s$)</td>
<td>DECIMAL***†††</td>
<td>A fixed-point decimal number, with 38 digits precision.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>An eight-byte, double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
<td>An eight-byte, double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>A signed four-byte integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>*</td>
<td>MAP&lt;$\text{primitive-type}, \text{data-type}&gt;$</td>
<td>An associative array of key-value pairs.</td>
<td>STRING‡</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE</td>
<td>A 64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>A signed two-byte integer, from -32,768 to 32,767.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>*</td>
<td>STRING</td>
<td>A variable-length character string.</td>
<td>VARCHAR($n$)††</td>
</tr>
<tr>
<td>*</td>
<td>STRUCT&lt;$\text{col-name: data-type}&gt;$</td>
<td>A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.</td>
<td>STRING‡</td>
</tr>
</tbody>
</table>
### Data Type Definition

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Hive Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME(p)</td>
<td>***</td>
<td>A time value.</td>
<td>STRING</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP</td>
<td>A UNIX timestamp with optional nanosecond precision.</td>
<td>TIMESTAMP[p]</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>A signed one-byte integer, from -128 to 127.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>*</td>
<td>UNION&lt;data-type, data-type-n&gt;</td>
<td>A type that can hold one of several specified data types.</td>
<td>STRING‡</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>A varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>BINARY</td>
<td>A varying length binary string up to 32K.</td>
<td>BINARY**</td>
</tr>
</tbody>
</table>

* The Hive data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

‡ Full support for this data type is available in Hive 0.12 and later. In Hadoop environments that use earlier Hive versions (which do not support the DATE type), any SASFMT TableProperties that are defined on STRING columns are applied when reading Hive, effectively allowing the STRING columns to be treated as DATE columns. When the DATE data type is used for data definition in earlier Hive versions, the DATE type is mapped to a STRING column with SASFMT TableProperties. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in SAS/ACCESS for Relational Databases: Reference.

† Full support for this data type is available in Hive 0.13 and later. In Hadoop environments that use earlier Hive versions (which do not support the CHAR and DECIMAL types), columns defined as CHAR are mapped to VARCHAR. Columns that are defined as DECIMAL are not supported. In Hadoop environments that use Hive versions earlier than Hive 0.13, BINARY columns can be created but not retrieved.

†† If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(n) instead.

††† Decimals processed by SAS are processed using a DOUBLE, which can alter the precision.

‡‡‡ The maximum length of VARCHAR(n) and the Hive complex types is determined by the DBMAX_TEXT= data source connection option.

SASFMT Table Properties are applied when reading STRING columns.

Hive does not support the TIME(p) data type. When data is read from Hive, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(p) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in SAS/ACCESS for Relational Databases: Reference.

The complex types ARRAY, MAP, STRUCT, and UNION are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.

## Data Types for Impala

The following table lists the data type support for Impala. Impala version 2.0 and later are supported, running on CDH 5.1 and later.

The BINARY, DECIMAL(p,s)/NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not available for data definition.

For data-source specific information about Impala data types, see the vendor documentation.
## Data Type Definition

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Impala Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR</td>
<td>A fixed-length character string up to 255 characters.</td>
<td>CHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>**</td>
<td>An ANSI SQL date type.</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>An eight-byte, double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
<td>An eight-byte, double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
<td>A signed four-byte integer, from -2,147,483,648 to 2,147,483,647.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE</td>
<td>An eight-byte, double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>A signed two-byte integer, from -32,768 to 32,767.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME</td>
<td>**</td>
<td>A time value.</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>A UNIX timestamp with optional nanosecond precision.</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>A signed one-byte integer, from -128 to 127.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>VARCHAR*</td>
<td>A varying-length character string.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

* Support for this data type is available in CDH 5.2 and later. In environments that use earlier CDH versions, which do not support the CHAR and VARCHAR types, columns defined as CHAR or VARCHAR are mapped to STRING.

** Impala does not support this data type. When a DATE or TIME column is defined, it is created as a column of type TIMESTAMP.

## Data Types for MDS

The following table lists the data type support for the in-memory database.

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>MDS Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>64-bit, signed integer.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Data Type Definition</td>
<td>MDS Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------</td>
<td>-------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>BINARY($n$)</td>
<td>BINARY($n$)</td>
<td>Fixed-length binary data.</td>
<td>BINARY($n$)</td>
</tr>
<tr>
<td>CHAR($n$)</td>
<td>CHAR($n$)</td>
<td>Fixed-length character string data.</td>
<td>CHAR($n$)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date value.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC($p,s$)</td>
<td>NUMERIC($p,s$)</td>
<td>Precision scale numeric.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>8-byte IEEE floating-point value.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
<td>8-byte IEEE floating-point value.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Note:</em> Supports ANSI SQL null values</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>32-bit, signed integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NCHAR($n$)</td>
<td>NCHAR($n$)</td>
<td>Fixed-length Unicode character string.</td>
<td>NCHAR($n$)</td>
</tr>
<tr>
<td>NVARCHAR($n$)</td>
<td>NVARCHAR($n$)</td>
<td>Varying-length Unicode character string.</td>
<td>NVARCHAR($n$)</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE</td>
<td>8-byte IEEE floating-point value.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Note:</em> Supports ANSI SQL null values</td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INTEGER</td>
<td>32-bit, signed integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIME($p$)</td>
<td>TIME($p$)</td>
<td>Time value.</td>
<td>TIME($p$)</td>
</tr>
<tr>
<td>TIMESTAMP($p$)</td>
<td>TIMESTAMP($p$)</td>
<td>Date and time value.</td>
<td>TIMESTAMP($p$)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INTEGER</td>
<td>32-bit, signed integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>*</td>
<td>UBIGINT</td>
<td>64-bit, unsigned integer.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>*</td>
<td>UINTINTEGER</td>
<td>32-bit, unsigned integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>VARCHAR($n$)</td>
<td>VARCHAR($n$)</td>
<td>Varying-length character string data.</td>
<td>VARCHAR($n$)</td>
</tr>
</tbody>
</table>
### Data Types for MySQL

The following table lists the data type support for a MySQL database.

The NCHAR, NVARCHAR, REAL, and VARBINARY data types are not supported for data type definition.

For data source specific information about MySQL data types, see the MySQL database documentation.

**Table A1.10** Data Types for MySQL

<table>
<thead>
<tr>
<th>Data Type Definition Keyword **</th>
<th>MySQL Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
<td>Fixed-length binary string.</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>**</td>
<td>BLOB</td>
<td>Varying-length binary large object string.</td>
<td>**</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>**</td>
<td>DATETIME</td>
<td>Date and time value.</td>
<td>**</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>**</td>
<td>ENUM(values)</td>
<td>Character values from a list of allowed values.</td>
<td>**</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT(p)</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>Data Type Definition Keyword</td>
<td>MySQL Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>**</td>
<td>LONGBLOB</td>
<td>Varying-length binary data.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>LONGTEXT</td>
<td>Varying-length character string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>MEDIUMBLOB</td>
<td>Varying-length binary data.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>MEDIUMINT</td>
<td>Regular signed, exact whole number.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>MEDIUMTEXT</td>
<td>Varying-length character string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>SET(values)</td>
<td>Character values from a list of allowed values.</td>
<td>**</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>**</td>
<td>TEXT</td>
<td>Varying-length text data.</td>
<td>**</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>**</td>
<td>TINYBLOB</td>
<td>Varying-length binary large object string.</td>
<td>**</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>Very small signed, exact whole number.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>**</td>
<td>TINYTEXT</td>
<td>Varying-length text data.</td>
<td>**</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The MySQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

---

### Data Types for Netezza

The following table lists the data type support for a Netezza database.
The BINARY and VARBINARY data types are not supported for data type definition.

For data source specific information about Netezza data types, see the Netezza database documentation.

**Table A1.11  Data Types for Netezza**

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Netezza Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Fixed-length character string data.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>Fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT(p)</td>
<td>64-bit double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Large integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>NCHAR(n)</td>
<td>Fixed-length Unicode character string.</td>
<td>NCHAR(n)</td>
</tr>
<tr>
<td>NVARCHAR(n)</td>
<td>NVARCHAR(n)</td>
<td>Varying-length Unicode character string.</td>
<td>NVARCHAR(n)</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small integer.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value.</td>
<td>TIME(p)**</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>BYTEINT</td>
<td>Tiny integer.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string data.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** Due to the ODBC-style interface that is used to communicate with the Netezza server, fractional seconds are lost in the data transfer from server to client.
Data Types for ODBC

The following table lists the data type support for an ODBC-compliant data source. For data source specific information about ODBC SQL data types, see the specific ODBC data source documentation.

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>ODBC SQL Identifier</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>SQL_BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>SQL_BINARY</td>
<td>Fixed-length binary string.</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>**</td>
<td>SQL_BIT</td>
<td>Single bit binary data.</td>
<td>**</td>
</tr>
<tr>
<td>CHAR(n)**</td>
<td>SQL_CHAR</td>
<td>Fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_TYPE_DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
<td>SQL_DECIMAL</td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>SQL_DOUBLE</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>SQL_FLOAT</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>**</td>
<td>SQL_GUID</td>
<td>Globally unique identifier.</td>
<td>**</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>**</td>
<td>SQL_INTERVAL</td>
<td>Intervals between two years, months, days, dates or times.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>SQL_LONGVARBINARY</td>
<td>Varying-length binary string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>SQL_LONGVARCHAR</td>
<td>Varying-length Unicode character string.</td>
<td>**</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>SQL_WCHAR</td>
<td>Fixed-length Unicode character string.</td>
<td>NCHAR(n)</td>
</tr>
</tbody>
</table>
### Data Type Definition

<table>
<thead>
<tr>
<th>Keyword</th>
<th>ODBC SQL Identifier</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVARCHAR(n)</td>
<td>SQL_WVARCHAR</td>
<td>Varying-length Unicode character string.</td>
<td>NVARCHAR(n)</td>
</tr>
<tr>
<td>REAL</td>
<td>SQL_REAL</td>
<td>Signed, single precision, floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SQL_SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>SQL_TYPE_TIME</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>SQL_TYPE_TIMESTAMP</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SQL_TINYINT</td>
<td>Very small signed, exact whole number.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>SQL_VARBINARY</td>
<td>Varying-length binary string.</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARCHAR(n)**</td>
<td>SQL_VARCHAR</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>**</td>
<td>SQL_WLONGVARCHAR</td>
<td>Varying-length Unicode character string.</td>
<td>**</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The ODBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** When you use the CHAR(n) or VARCHAR(n) data type to store multibyte data in a DB2, Greenplum, or Oracle database, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, for Oracle only, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.

### Data Types for Oracle

The following table lists the data type support for an Oracle database.

For data source specific information about Oracle data types, see the Oracle database documentation.

#### Table A1.13  Data Types for Oracle

<table>
<thead>
<tr>
<th>Data Type Definition Keyword*</th>
<th>Oracle Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>NUMBER</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>
### Data Type Definition

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Oracle Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY($n$)</td>
<td>RAW($n$)</td>
<td>Fixed or varying length binary string.</td>
<td>BINARY($n$)</td>
</tr>
<tr>
<td>CHAR($n$)</td>
<td>CHAR($n$)</td>
<td>Fixed-length character string.</td>
<td>CHAR($n$)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>TIMESTAMP($p$)***</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC($p$,$s$)</td>
<td>NUMBER($p$,$s$)</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC($p$,$s$)</td>
<td>NUMBER($p$,$s$)</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>BINARY_DOUBLE</td>
<td>Signed, double precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>BINARY_DOUBLE</td>
<td>Signed, double precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT($p$)</td>
<td>Signed, double precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>NUMBER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>**</td>
<td>LONG</td>
<td>Varying-length character string data.</td>
<td>**</td>
</tr>
<tr>
<td>NCHAR($n$)</td>
<td>NCHAR($n$)</td>
<td>Fixed-length Unicode character string.</td>
<td>NCHAR($n$)</td>
</tr>
<tr>
<td>NVARCHAR($n$)</td>
<td>NVARCHAR($n$)</td>
<td>Varying-length Unicode character string.</td>
<td>NVARCHAR($n$)</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT</td>
<td>Signed, single precision floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>NUMBER</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME($p$)</td>
<td>TIME($p$)</td>
<td>Time value.</td>
<td>TIMESTAMP($p$)***</td>
</tr>
<tr>
<td>TIMESTAMP($p$)††</td>
<td>TIMESTAMP($p$)</td>
<td>Date and time value.</td>
<td>TIMESTAMP($p$)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>NUMBER</td>
<td>Very small signed, exact whole number.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY($n$)</td>
<td>LONG RAW($n$)</td>
<td>Varying-length binary string.</td>
<td>VARBINARY($n$)</td>
</tr>
<tr>
<td>VARCHAR($n$)</td>
<td>VARCHAR2($n$)†††</td>
<td>Varying-length character string.</td>
<td>VARCHAR($n$)</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE
can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Oracle data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** The timestamp returned by the DATE and TIME data types can be changed to date and time values by using the DATEPART function with the PUT function.
†† The ORNUMERIC= connection argument and table option determine how numbers read from or inserted into the Oracle NUMBER column are treated. ORNUMERIC=YES, which is the default, indicates that non-integer values with explicit precision are treated as NUMERIC values.
††† The TIMESTAMP(\(p\)) data type is not available on Z/OS.
†††† The VARCHAR2(\(n\)) type is supported for up to 32,767 bytes if the Oracle version is 12c and the Oracle MAX\_STRING\_SIZE= parameter is set to EXTENDED.

### Data Types for PostgreSQL

The following table lists the data type support for a PostgreSQL database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source specific information about PostgreSQL data types, see the PostgreSQL database documentation.

**Table A1.14 Data Types for PostgreSQL**

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>PostgreSQL Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number. OR Signed eight-byte integer.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>**</td>
<td>BIGSERIAL</td>
<td>Autoincrementing eight-byte integer.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>BIT((n))</td>
<td>Fixed-length bit string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>BIT VARYING((n))</td>
<td>Variable-length bit string.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>BOOLEAN</td>
<td>Logical Boolean (true/false).</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>BOX</td>
<td>Rectangular box on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>BYTEA</td>
<td>Binary data (byte array).</td>
<td>**</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>Fixed-length character string.</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>**</td>
<td>CIDR</td>
<td>IPv4 or IPv6 network address.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>CIRCLE</td>
<td>Circle on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>Data Type Definition</td>
<td>PostgreSQL Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date value.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC($p$,$s$)</td>
<td>NUMERIC($p$,$s$)</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE PRECISION</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT($p$)</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>**</td>
<td>INET</td>
<td>IPv4 or IPv6 host address.</td>
<td>**</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>**</td>
<td>INTERVAL</td>
<td>Time span.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>LINE</td>
<td>Infinite line on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>LSEG</td>
<td>Line segment on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>MACADDR</td>
<td>Media Access Control address.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>MONEY</td>
<td>Currency amount.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>PATH</td>
<td>Geometric path on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>POINT</td>
<td>Geometric point on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>POLYGON</td>
<td>Closed geometric path on a plane.</td>
<td>**</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Signed, single precision floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>**</td>
<td>SERIAL</td>
<td>Autoincrementing four-byte integer.</td>
<td>**</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>**</td>
<td>SMALL SERIAL</td>
<td>Autoincrementing two-byte integer.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>TEXT</td>
<td>Variable-length character string.</td>
<td>**</td>
</tr>
<tr>
<td>TIME($p$)</td>
<td>TIME($p$)</td>
<td>Time value.</td>
<td>TIME($p$)</td>
</tr>
</tbody>
</table>
## Data Types for SAP

The following table lists the data type support for an SAP system.

For an SAP system, no data types are supported for column definition. Native ABAP SAP data types are mapped to similar data types for data retrieval only.

For data source specific information about the ABAP SAP data types, see the SAP system database documentation.

### Table A1.15 Data Types for SAP

<table>
<thead>
<tr>
<th>ABAP SAP Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCP</td>
<td>Posting period.</td>
<td>CHAR($n$) for non-Unicode SAP system; NCHAR($n$) for Unicode SAP system</td>
</tr>
<tr>
<td>CHAR</td>
<td>Fixed-length character string.</td>
<td>CHAR($n$) for non-Unicode SAP system; NCHAR($n$) for Unicode SAP system</td>
</tr>
<tr>
<td>CLNT</td>
<td>Client field.</td>
<td>CHAR($n$) for non-Unicode SAP system; NCHAR($n$) for Unicode SAP system</td>
</tr>
<tr>
<td>CUKY</td>
<td>Currency key. Fields of this type are referenced by fields of type CURR.</td>
<td>CHAR($n$) for non-Unicode SAP system; NCHAR($n$) for Unicode SAP system</td>
</tr>
<tr>
<td>ABAP SAP Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>CURR</td>
<td>Currency field. Corresponds to the DEC field. Field refers to a field of type CUKY.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATS</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DEC</td>
<td>Signed, fixed-point decimal number.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>FLTP</td>
<td>Floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INT1</td>
<td>Very small signed, exact whole number.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>INT2</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INT4</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LANG</td>
<td>Language key, which has its own field format for special functions. The conversion exit ISOLA converts the value to be displayed to that of the database and the opposite is true.</td>
<td>CHAR(n) for non-Unicode SAP system; NCHAR(n) for Unicode SAP system</td>
</tr>
<tr>
<td>LCHR</td>
<td>Fixed-length character string.</td>
<td>VARCHAR(n) for non-Unicode SAP system; NVARCHAR(n) for Unicode SAP system</td>
</tr>
<tr>
<td>LRAW</td>
<td>Uninterpreted varying-length byte string.</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>NUMC</td>
<td>Text string.</td>
<td>CHAR(n) for non-Unicode SAP system; NCHAR(n) for Unicode SAP system</td>
</tr>
<tr>
<td>PREC</td>
<td>The precision of a QUAN field.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>QUAN</td>
<td>A quantity that corresponds to the DEC field.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>RAW</td>
<td>An uninterpreted byte string.</td>
<td>BINARY</td>
</tr>
<tr>
<td>TIMS</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>UNIT</td>
<td>Units key and referenced by a QUAN data type.</td>
<td>CHAR(n) for non-Unicode SAP system; NCHAR(n) for Unicode SAP system</td>
</tr>
<tr>
<td>VARC</td>
<td>Varying-length character string data. As of SAP release 3.0, creating fields of this data type is no longer supported. Existing fields with this data type can be used, except in a WHERE condition in the SELECT statement.</td>
<td>VARCHAR(n) for non-Unicode SAP system; NVARCHAR(n) for Unicode SAP system</td>
</tr>
</tbody>
</table>
The following table lists the data type support for an SAP HANA database.
For data source specific information about the SAP HANA data types, see the SAP HANA database documentation.

### Table A1.16  Data Types for SAP HANA

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>SAP HANA Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td>ALPHANUM((n))</td>
<td>Varying-length character string.</td>
<td>WINAPICHAR((n))</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>64-bit integer.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY((n))</td>
<td>BINARY((n))</td>
<td>Fixed-length binary data.</td>
<td>BINARY((n))</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>BLOB</td>
<td>Varying-length binary large object string.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>Varying-length character string.</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>CLOB</td>
<td>Varying-length character large object string.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Year, month, and day values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL|NUMERIC((p,s))</td>
<td>DECIMAL((p,s))</td>
<td>Signed, exact, fixed-point decimal number.</td>
<td>DECIMAL((p,s))</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>Double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT((p))</td>
<td>Double-precision floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>32-bit integer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NCHAR((n))</td>
<td>NCHAR((n))</td>
<td>Fixed-length Unicode character string.</td>
<td>NCHAR((n))</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>NCLOB</td>
<td>Fixed-length character large object string.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>NVARCHAR((n))</td>
<td>NVARCHAR((n))</td>
<td>Varying-length Unicode character large object string.</td>
<td>NVARCHAR((n))</td>
</tr>
</tbody>
</table>
### Data Type Definition

<table>
<thead>
<tr>
<th>Keyword</th>
<th>SAP HANA Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>Floating-point number.</td>
<td>REAL</td>
</tr>
<tr>
<td>**</td>
<td>SECONDATE</td>
<td>Date and time value.</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>SHORTTEXT(n)</td>
<td>Varying-length character string.</td>
<td>NVARCHAR(n)</td>
</tr>
<tr>
<td>**</td>
<td>SMALLDECIMAL(p,s)</td>
<td>Floating-point decimal number.</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>16-bit integer.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>**</td>
<td>TEXT</td>
<td>Varying-length Unicode character large object string.</td>
<td>**</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>Unsigned 8-bit integer.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>VARBINARY(n)</td>
<td>Varying-length binary string.</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The SAP HANA data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

---

**Data Types for SASHDAT**

The following table lists the data type support for a SASHDAT table. A SASHDAT table contains data that is added to the Hadoop Distributed File System (HDFS) by SAS.

The BIGINT, BINARY, DECIMAL, FLOAT, INTEGER, NCHAR, NVARCHAR, NUMERIC, REAL, SMALLINT, TINYINT, VARBINARY, and VARCHAR data types are not supported for data type definition.

A SASHDAT table is Write only for data and Read-Write for metadata information such as column attributes.
### Table A1.17  Data Types for SASDAT

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>SASDAT Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)**</td>
<td>CHAR(n)</td>
<td>Fixed-length character string.</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>64-bit double-precision, floating-point number.</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>DOUBLE</td>
<td>64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** When the (n) is omitted from the CHAR definition, a column of CHAR(1) is created.

---

### Data Types for Sybase IQ

The following table lists the data type support for a Sybase IQ database.

For data source specific information about the Sybase IQ database data types, see the Sybase IQ database documentation.

### Table A1.18  Data Types for Sybase IQ

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Sybase IQ Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>64-bit integer.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
<td>Fixed-length binary string.</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>***</td>
<td>BIT</td>
<td>Integer that stores only the values 0 or 1.</td>
<td>***</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Varying-length character string.</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>Data Type Definition Keyword</td>
<td>Sybase IQ Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------</td>
<td>-------------</td>
<td>-------------------</td>
</tr>
</tbody>
</table>
| DECIMAL|NUMERIC($p,s$)      | DECIMAL($p,s$) | Signed, exact, fixed-point decimal number.  
| DOUBLE | DOUBLE             | Double-precision floating-point number.  
| FLOAT | FLOAT($p$)         | Double-precision floating-point number.  
| INTEGER | INTEGER | 32-bit integer. | INTEGER  
| ** | LONG BINARY | Varying-length binary string. | VARBINARY($n$)  
| ** | LONG VARBIT | Arbitrary length bit arrays. | VARCHAR($n$)  
| NCHAR($n$) | NCHAR($n$)  | Fixed-length Unicode character string. | NCHAR($n$)  
| ** | LONG NVARCHAR($n$) | Varying length Unicode character string. | NVARCHAR($n$)  
| ** | MONEY | Fixed-point decimal number that stores monetary data. | DOUBLE  
| NVARCHAR($n$) | NVARCHAR($n$)  | Varying-length Unicode character string. | NVARCHAR($n$)  
| REAL | REAL | Floating-point number. | REAL  
| SMALLINT | SMALLINT  | 16-bit integer. | SMALLINT  
| TIME($p$) | TIME($p$) | Time value. | TIME($p$)  
| TIMESTAMP($p$) | TIMESTAMP($p$) | Date and time value. | TIMESTAMP($p$)  
| TINYINT | TINYINT | Unsigned 8-bit integer. | TINYINT  
| VARCHAR($n$) | VARCHAR($n$) | Varying-length binary string. | VARCHAR($n$)  
| VARCHAR($n$) | VARCHAR($n$) | Varying-length character string. | VARCHAR($n$)  

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Sybase IQ data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** The Sybase IQ data type cannot be defined or retrieved.
Data Types for Teradata

The following table lists the data type support for a Teradata database.

The NCHAR, NVARCHAR, and REAL data types are not supported for data type definition.

For data source specific information about the Teradata data types, see the Teradata database documentation.

Table A1.19  Data Types for Teradata

<table>
<thead>
<tr>
<th>Data Type Definition Keyword</th>
<th>Teradata Data Type</th>
<th>Description</th>
<th>Data Type Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
<td>Large signed, exact whole number.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>BYTE(n)</td>
<td>Fixed-length binary string</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>BLOB</td>
<td>Large Binary Object.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>Fixed-length character string.</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>CLOB</td>
<td>Large Character Object.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Date values.</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>Signed, fixed-point decimal number.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT(p)</td>
<td>Signed, double precision, floating-point number.</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>Regular signed, exact whole number.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>LONG VARCHAR</td>
<td>Varying-length character string.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>Small signed, exact whole number.</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME(p)</td>
<td>TIME(p)</td>
<td>Time value.</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>TIMESTAMP(p)</td>
<td>Date and time value.</td>
<td>TIMESTAMP(p)</td>
</tr>
<tr>
<td>Data Type Definition Keyword</td>
<td>Teradata Data Type</td>
<td>Description</td>
<td>Data Type Returned</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>TINYINT</td>
<td>BYTEINT</td>
<td>Very small signed, exact whole number.</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY((n))</td>
<td>VARBYTE((n))</td>
<td>Varying-length binary string.</td>
<td>VARBINARY((n))</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>VARCHAR((n))</td>
<td>Varying-length character string.</td>
<td>VARCHAR((n))</td>
</tr>
</tbody>
</table>

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Teradata data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.
The following table lists the automatic type conversions for expression operands. The first column is the “from” type. The first row is the “to” type.
<table>
<thead>
<tr>
<th>From/To</th>
<th>TinyInt</th>
<th>SmallInt</th>
<th>Integer</th>
<th>BigInt</th>
<th>Decimal/</th>
<th>Real</th>
<th>Double</th>
<th>Date</th>
<th>Time</th>
<th>Timestamp</th>
<th>Char</th>
<th>Varchar</th>
<th>NChar</th>
<th>NVarchar</th>
<th>Binary</th>
<th>Varbinary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyInt</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SmallInt</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Integer</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BigInt</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Decimal/</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Real</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Double</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Date</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Time</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Timestamp</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Char</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Varchar</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>NChar</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>NVarchar</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Binary</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Varbinary</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Overview of DS2 Loggers

The SAS logging facility is a framework that categorizes and filters log messages in SAS server and SAS programming environments, and writes log messages to various output devices. In the server environment, the logging facility logs messages based on predefined message categories, such as Admin for administrative messages, App for application messages, and Perf for performance messages. Messages for a category can be written to files, consoles, and other system destinations simultaneously. The logging facility also enables messages to be filtered based on the following thresholds: TRACE, DEBUG, INFO, WARN, ERROR, and FATAL.

DS2 provides several loggers to report both configuration and run-time information. In addition, DS2 provides a logger for the HTTP package. In general, INFO provides the minimum amount of information and DEBUG provides most (perhaps all) of the information that is needed to debug a field problem. The TRACE level provides anything and everything that might be of interest.

For more information about loggers and the SAS logging facility, see *SAS Logging: Configuration and Programming Reference*.

Configuration Loggers

Configuration loggers track information as the DS2 compiler starts up and provide context for the actual execution of the user's code.

The following configuration loggers are available.

App.TableServices.DS2.Config.Options shows the options supplied to the DS2 compiler.
App.TableServices.DS2.Config.Source
shows the DS2 source code processed by the DS2 compiler.

App.TableServices.DS2.Config.Version
shows version information for all threaded kernel extensions loaded by the DS2 compiler.

Run-Time Loggers

Run-time loggers track actual execution. Some of the tracked information is output for each row processed. Therefore, a large input table can produce very large amounts of data.

The following run-time loggers are available.

App.TableServices.DS2.Runtime.Calls
shows a trace of all method calls during execution.

App.TableServices.DS2.Runtime.SQL
shows all SQL statements either prepared by the DS2 compiler, executed by the DS2 compiler, or both.

App.TableServices.DS2.Runtime.Timing
shows the time that is spent during code compilation and execution. Depending on the level of information that is requested (INFO, DEBUG, TRACE), timing information is provided at various points throughout DS2 execution. Examples are parse time, compilation time, various audit and transformation pass times, as well as the INIT, RUN, and TERM method execution times.

App.TableServices.DS2.Runtime.Put
records all PUT statement output.

App.TableServices.DS2.Runtime.Log
records all messages that are sent to the SAS log in a SAS session. On SAS Federation Server, these messages are also appended to the ODBC statement handle as diagnostic records by default.

HTTP Package Logger

The HTTP client supports logging of HTTP traffic through the SAS logging facility.

The name of the logger is App.TableServices.d2pkg.HTTP. The logger supports these logging levels:

INFO
shows general traffic information such as connections and disconnections from the web server, request information, and status information.

DEBUG
shows the headers from all requests and responses plus the first 64 bytes of body data. This enables you to see what the client and server are doing without having to see all of the data that is transmitted to the server.

TRACE
shows all of the data sent to and received from the web server.
Example: Logging All SQL Operations

The following example creates a logging configuration file to log all SQL commands. After the configuration file is created, the LOGCONFIGLOC= system option is set to specify the name of the configuration file that is used to initialize the SAS logging facility.

The following code creates the logging configuration file.

```xml
<?xml version="1.0"?>
<logging:configuration xmlns:logging="http://www.sas.com/xml/logging/1.0/"
    logversion="1.0">
    <logger name="App.TableServices.DS2.Runtime.SQL">
        <level value="trace"/>
    </logger>
    <logger name="App.TableServices.DS2* additivity=false">
        <appender-ref ref="DetailedOutput"/>
    </logger>
    <root>
        <appender-ref ref="RootLogger"/>
        <level value="info"/>
    </root>
    <appender name="DetailedOutput" class="FileAppender">
        <param name="append" value="false"/>
        <param name="FileNamePattern" value="sql.%S{App.Log}"/>
        <layout>
            <param name="ConversionPattern" value="%-5p:%sn:[%c{3}]:%m"/>
        </layout>
    </appender>
    <appender name="RootLogger" class="FileAppender">
        <param name="Append" value="false"/>
        <param name="ImmediateFlush" value="true"/>
        <param name="FileNamePattern" value="%S{App.Log}"/>
        <layout>
            <param name="ConversionPattern" value="%m"/>
        </layout>
    </appender>
</logging:configuration>
```

The LOGCONFIGLOC= system option is set to reference the configuration file as `config.14s`.

`sas -log test.log -logconfigloc config.14s`

For more information about this system option, see *SAS Logging: Configuration and Programming Reference*.

This produces a file, `sql.test.log`, which contains a series of messages similar to these.

```
DEBUG:00000084:[DS2.Runtime.SQL]:Found 0 NOCHANGE columns
INFO :00000085:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:CREATE TABLE WORK.outp
     ("i" DOUBLE, "j" DOUBLE )
DEBUG:00000086:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:passed:rc=0x00000000
DEBUG:00000087:[DS2.Runtime.SQL]:Found 0 NOCHANGE columns
INFO :00000088:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:SELECT * FROM WORK.outp
DEBUG:00000089:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:passed:rc=0x00000000
```
The exact content of the output file is defined by the `ConversionPattern` parameter of the layout within the DetailedOutput appender. The DetailedOutput appender is associated with the definition of the App.TableServices.DS2 logger. This causes every logger in the hierarchy that is rooted at App.TableServices.DS2 to be logged to the same file. The `additivity="false"` modifier prevents the log messages from moving upward.
Appendix 4
DS2 Example Programs

Example: Find Minimums ........................................... 1307
   Example Overview: Find Minimums .......................... 1307
   Example Code: Find Minimums ................................. 1307
   Example Output: Find Minimums .............................. 1308

Example: SQL in a DS2 Program ................................. 1308
   Example Overview: SQL in a DS2 Program ................. 1308
   Example Code: SQL in a DS2 Program ....................... 1308
   Example Output: SQL in a DS2 Program .................... 1310

Example: Make Two New Tables Based on a Condition .......... 1310
   Example Overview: Make Two New Tables Based on a Condition ................. 1310
   Example Code: Make Two New Tables Based on a Condition ....................... 1311
   Example Output: Make Two New Tables Based on a Condition .................... 1312

Example: Change Case of Text Output ......................... 1312
   Example Overview: Change Case of Text Output ........... 1312
   Example Code: Change Case of Text Output .............. 1312
   Example Output: Change Case of Text Output ........... 1313

Example: Scope .................................................. 1313
   Example Overview: Scope ..................................... 1313
   Example Code: Scope ......................................... 1313
   Example Output: Scope ..................................... 1314

Example: Functions ............................................. 1314
   Example Overview: Functions ................................. 1314
   Example Code: Functions ..................................... 1314
   Example Output: Functions ................................... 1315

Example: Arrays ................................................ 1315
   Example Overview: Arrays ................................... 1315
   Example Code: Arrays ....................................... 1315
   Example Output: Arrays ..................................... 1318

Example: SELECT Statement ...................................... 1319
   Example Overview: SELECT Statement ...................... 1319
   Example Code: SELECT Statement ............................ 1319
   Example Output: SELECT Statement ......................... 1320

Example: GOTO and LEAVE Statements with Labels ............ 1320
   Example Overview: GOTO and LEAVE Statements with Labels .................. 1320
   Example Code: GOTO and LEAVE Statements with Labels ......................... 1320
   Example Output: GOTO and LEAVE Statements with Labels ..................... 1322
Example: Overloaded Methods ........................................... 1323
  Example Overview: Overloaded Methods ................................ 1323
  Example Code: Overloaded Methods ..................................... 1323
  Example Output: Overloaded Methods .................................... 1324

Example: Analyze a Table Using Multiple Threads ..................... 1324
  Example Overview: Analyze a Table Using Multiple Threads .......... 1324
  Example Code: Analyze a Table Using Multiple Threads ............. 1324
  Example Output: Analyze a Table Using Multiple Threads .......... 1325

Example: Using Four Threads to Compute Summary Statistics ........... 1326
  Example Overview: Using Four Threads to Compute Summary Statistics ... 1326
  Example Code: Using Four Threads to Compute Summary Statistics .... 1326
  Example Output: Using Four Threads to Compute Summary Statistics ... 1327

Example: Data Cleaning .................................................. 1328
  Example Overview: Data Cleaning ....................................... 1328
  Example Code: Data Cleaning .......................................... 1328
  Example Output: Data Cleaning ........................................ 1329

Example: SUBSTR Function ............................................... 1329
  Example Overview: SUBSTR Function ................................... 1329
  Example Code: SUBSTR Function ....................................... 1329
  Example Output: SUBSTR Function ..................................... 1330

Example: PUT with SAS Formats ......................................... 1330
  Example Overview: PUT with SAS Formats .............................. 1330
  Example Code: PUT with SAS Formats .................................. 1330
  Example Output: PUT with SAS Formats ................................ 1331

Example: Generate Statistics from Table Data .......................... 1331
  Example Overview: Generate Statistics from Table Data .............. 1331
  Example Code: Generate Statistics from Table Data .................. 1331
  Example Output: Generate Statistics from Table Data ................. 1332

Example: Matrices and Non-linear Equations ............................ 1333
  Example Overview: Matrices and Non-linear Equations ............... 1333
  Example Code: Matrices and Non-linear Equations .................... 1333
  Example Output: Matrices and Non-linear Equations .................. 1336

Example: DS2 Matrices and Regression Analysis ........................ 1336
  Example Overview: DS2 Matrices and Regression Analysis ............. 1336
  Example Code: DS2 Matrices and Regression Analysis ................ 1336
  Example Output: DS2 Matrices and Regression Analysis ............... 1338

Example: Use the SQLSTMT Package in Another Package ............... 1339
  Example Overview: Use the SQLSTMT Package in Another Package .... 1339
  Example Code: Use the SQLSTMT Package in Another Package .......... 1339

Example: Update the Values of a Table By Using Two Databases ....... 1340
  Example Overview: Update the Values of a Table By Using Two Databases ... 1340
  Example Code: Update the Values of a Table By Using Two Databases .... 1340

Example: Store FedSQL Statements in a Hash Package ................. 1341
  Example Overview: Store FedSQL Statements in a Hash Package ....... 1341
  Example Code: Store FedSQL Statements in a Hash Package ............ 1341

Example: Run a DS2 Program from z/OS Batch .......................... 1346
  Example Overview: Run a DS2 Program from z/OS Batch ............... 1346
  Example Code: Run a DS2 Program from z/OS Batch .................... 1346
Example: Find Minimums

Example Overview: Find Minimums

This example demonstrates how to use the three system-defined methods, INIT, RUN, and TERM.

A DS2 program executes in the following sequence:

1. Any global variables are declared.
2. The INIT method is called. INIT is typically used for variable initialization.
3. The RUN method is called. The RUN method is where the implicit loop exists. RUN executes until all input tables are completely read.
4. The TERM method is called. Final processing is performed.

The following program demonstrates this flow of control by finding the minimum values in a table. In this program, the INIT method initializes the variables used to find the current minimum, the RUN method compares input values with the current minimum, and the TERM method outputs the minimums to an output table.

Example Code: Find Minimums

```ds2
proc ds2;
  /* Create table to work with in this example */
data xy_data;
    dcl double x y;
    method init();
      do x = 1 to 5;
        y = 2*x;
        output;
      end;
    end;
enddata;
run;
/* Find the minimum value for x and y */
data xy_mins;
  dcl double min_x min_y;
  retain min_x min_y;
  keep min_x min_y;

  method init();
    min_x = 999999;
    min_y = 999999;
  end;

  method run();
    set xy_data;
    if x < min_x then min_x = x;
    if y < min_y then min_y = y;
  end;
```
Example Output: Find Minimums

The SAS System

<table>
<thead>
<tr>
<th>MIN_X</th>
<th>MIN_Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Example: SQL in a DS2 Program

Example Overview: SQL in a DS2 Program

This example illustrates how to use an SQL statement in a DS2 program. To access the output from an SQL query, put the query in a SET statement. When the SET statement runs, it sequentially reads the rows that are returned by the query.

Example Code: SQL in a DS2 Program

In this example, the first DS2 program calculates annual balances for an account into which contributions of $2000 are made every year from 2004 to 2014. The account carries a 7% interest rate, compounded annually. The second program outputs the table. The third program outputs the results of an SQL query. The query selects all rows from INVESTMENT where the value of INVESTMENT_YEAR is greater than 2010.

```sas
proc ds2;
data investment;
dcl integer investment_year;
dcl double capital;
method init();
capital = 0;
do investment_year = 2004 to 2014;
capital = capital + (2000 + .07 * (capital+2000));
output;
end;
```
Example: SQL in a DS2 Program

```plaintext
end;
enddata;
run;
data;
    method run();
        set investment;
    end;
enddata;
run;
data;
    method run();
        set {select * from investment where investment_year > 2010};
    end;
enddata;
run;
quit;
```
Example Output: SQL in a DS2 Program

The SAS System

<table>
<thead>
<tr>
<th>INVESTMENT_YEAR</th>
<th>CAPITAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>2140</td>
</tr>
<tr>
<td>2005</td>
<td>4429.8</td>
</tr>
<tr>
<td>2006</td>
<td>6379.886</td>
</tr>
<tr>
<td>2007</td>
<td>9501.478</td>
</tr>
<tr>
<td>2008</td>
<td>12306.58</td>
</tr>
<tr>
<td>2009</td>
<td>15308.04</td>
</tr>
<tr>
<td>2010</td>
<td>18519.61</td>
</tr>
<tr>
<td>2011</td>
<td>21955.98</td>
</tr>
<tr>
<td>2012</td>
<td>25632.9</td>
</tr>
<tr>
<td>2013</td>
<td>29567.2</td>
</tr>
<tr>
<td>2014</td>
<td>33776.9</td>
</tr>
</tbody>
</table>

Example: Make Two New Tables Based on a Condition

Example Overview: Make Two New Tables Based on a Condition

This example illustrates how to create tables based on a condition. Programs 1 and 2 create two tables, DEPT1_ITEMS and DEPT2_ITEMS, that hold costs for items used by two departments. The third program creates two tables, HIGHCOSTS and LOWCOSTS,
Example Code: Make Two New Tables Based on a Condition

proc ds2;
/* Program 1 */
data dept1_items (overwrite=yes);
dcl varchar(20) item;
dcl double cost;
method init();
   item = 'staples';   cost =  1.59; output;
   item = 'pens';      cost =  3.26; output;
   item = 'envelopes'; cost = 11.42; output;
end;
enddata;
run;
/* Program 2 */
data dept2_items (overwrite=yes);
dcl varchar(20) item;
dcl double cost;
method init();
   item = 'erasers'; cost =  5.43; output;
   item = 'paper';   cost = 26.92; output;
   item = 'toner';   cost = 62.29; output;
end;
enddata;
run;
/* Program 3 */
data lowCosts (overwrite=yes) highCosts (overwrite=yes);
method run();
   set dept1_items dept2_items;
   if cost <= 10.00 then
      output lowCosts;
   else
      output highCosts;
end;
enddata;
run;
/* Program 4 */
data;
method run();
   set lowCosts;
end;
enddata;
run;
/* Program 5 */
data;
method run();
   set highCosts;
end;
enddata;
run;
quit;
**Example Output: Make Two New Tables Based on a Condition**

<table>
<thead>
<tr>
<th>ITEM</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>staples</td>
<td>1.59</td>
</tr>
<tr>
<td>pens</td>
<td>3.26</td>
</tr>
<tr>
<td>erasers</td>
<td>5.43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ITEM</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>envelopes</td>
<td>11.42</td>
</tr>
<tr>
<td>paper</td>
<td>26.92</td>
</tr>
<tr>
<td>toner</td>
<td>62.29</td>
</tr>
</tbody>
</table>

**Example: Change Case of Text Output**

**Example Overview: Change Case of Text Output**

The code in this example reads the two tables created in the previous example, DEPT1_COSTS and DEPT2_COSTS, and outputs rows with values in the COST column of less than or equal to $10 in lower case, and values greater than $10 in uppercase.

**Example Code: Change Case of Text Output**

```sas
proc ds2;
data;
   method run();
      set dept1_items dept2_items;
      if cost <= 10.00 then
         item = lowcase(item);
      else
         item = upcase(item);
   end;
enddata;
run;
quit;
```
**Example Output: Change Case of Text Output**

![The SAS System](image)

<table>
<thead>
<tr>
<th>ITEM</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>staples</td>
<td>1.59</td>
</tr>
<tr>
<td>pens</td>
<td>3.26</td>
</tr>
<tr>
<td>ENVELOPES</td>
<td>11.42</td>
</tr>
<tr>
<td>erasers</td>
<td>5.43</td>
</tr>
<tr>
<td>PAPER</td>
<td>26.92</td>
</tr>
<tr>
<td>TONER</td>
<td>62.29</td>
</tr>
</tbody>
</table>

---

**Example: Scope**

**Example Overview: Scope**

This example shows the use of CHAR and VARCHAR types and their operators and functions. Also, in this example, the variables A, B, and C are locally scoped to the INIT method. That is, their value is not seen outside of the INIT method and is not output to the result table.

**Example Code: Scope**

```sas
proc ds2;
data;
   dcl char(24) abc abc2;
   method init();
      dcl char(8) a b c;
      a = repeat('a',5);
      b = repeat('b',6);
      c = repeat('c',7);
      abc = a || b || c;
      abc2 = trim(a) || trim(b) || c;
   end;
enddata;
run;
data;
   dcl char(24) abc abc2;
   method init();
      dcl varchar(8) a b c;
```

---

**Example: Scope**

**Example Overview: Scope**

This example shows the use of CHAR and VARCHAR types and their operators and functions. Also, in this example, the variables A, B, and C are locally scoped to the INIT method. That is, their value is not seen outside of the INIT method and is not output to the result table.

**Example Code: Scope**

```sas
proc ds2;
data;
   dcl char(24) abc abc2;
   method init();
      dcl char(8) a b c;
      a = repeat('a',5);
      b = repeat('b',6);
      c = repeat('c',7);
      abc = a || b || c;
      abc2 = trim(a) || trim(b) || c;
   end;
enddata;
run;
data;
   dcl char(24) abc abc2;
   method init();
      dcl varchar(8) a b c;
```

---

**Example: Scope**

**Example Overview: Scope**

This example shows the use of CHAR and VARCHAR types and their operators and functions. Also, in this example, the variables A, B, and C are locally scoped to the INIT method. That is, their value is not seen outside of the INIT method and is not output to the result table.

**Example Code: Scope**

```sas
proc ds2;
data;
   dcl char(24) abc abc2;
   method init();
      dcl char(8) a b c;
      a = repeat('a',5);
      b = repeat('b',6);
      c = repeat('c',7);
      abc = a || b || c;
      abc2 = trim(a) || trim(b) || c;
   end;
enddata;
run;
data;
   dcl char(24) abc abc2;
   method init();
      dcl varchar(8) a b c;
```
Example Output: Scope

```
The SAS System

<table>
<thead>
<tr>
<th>abc</th>
<th>abc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaaaa bbbbbbb cccccc</td>
<td>aaaaaaabbhbccccc</td>
</tr>
<tr>
<td>cccc</td>
<td>cccc</td>
</tr>
</tbody>
</table>
```

Example: Functions

Example Overview: Functions

This example shows how to use a function in DS2. The code computes the number of bits required to store an integer.

Example Code: Functions

```
proc ds2;
data;
dcl integer i bits;
method init();
do i = 1 to 1000 by 100;
   bits = ceil(log(i) / log(2));
   output;
end;
end;
enddata;
```
Example Output: Functions

Example: Arrays

Example Overview: Arrays

The first program illustrates basic array procedures. In the final section, elements of `dblNegSubArray` are specified by using numeric expressions inside the array brackets. Expressions that resolve to a number that falls out of the bounds of the declared size of the array give an error message.

The second program gives several examples of array assignments. When an array is assigned, data types that do not match the type of the target array are converted to the target array type.

Note that if you add an equal sign after a variable or array element in a PUT statement, then the output is preceded by the variable or array element name and an equal sign.

Example Code: Arrays

```sas
proc ds2;
/* Basic Arrays */
data _null_;
dcl char(10) strArray[4] str;
dcl double dblArray[3];
```
dcl int intArray[10];
dcl double dblNegSubArray[-4:-1];
dcl int x;
method init();
  put 'BASIC ARRAYS';
  strArray[1] = 'abc';
  strArray[2] = 'def';
  strArray[3] = 'ghi';
  strArray[4] = 'jkl';
  put strArray[1]= ;
  put strArray[2]= ;
  put strArray[3]= ;
  put strArray[4]= ;
  put;

  str = strArray[1];
  put str=;
  put;

  dblArray[1] = 3;
  dblArray[2] = 99;
  dblArray[3] = dblArray[2];
  put dblArray[1]= ;
  put dblArray[2]= ;
  put dblArray[3]= ;
  put;

  do j = 1 to 10;
    intArray[j] = j;
    put intArray[j]=;
  end;
  put;

  dblArray[3] = intArray[5];
  put dblArray[3]=;
  put;

  dblNegSubArray[-4] = 102;
  dblNegSubArray[-3] = 101;
  dblNegSubArray[-2] = 100;
  dblNegSubArray[-1] = 99;

  x = 5;
  y = 7;
  a = dblNegSubArray[x-y];
  b = dblNegSubArray[x-y-1];
  c = dblNegSubArray[x-y-2];
  put a= b= c=;
  put;

  /* These will produce out-of-bounds messages */
  a = dblNegSubArray[x-y-3];
  e = dblNegSubArray[0];
end;
enddata;
run;
/* Array Assignment */
data _null_
  dcl int x;
  dcl char(10) s1[4] s2[4];
  dcl double d1[10] d2[10];
  dcl double arr2x3[2,3] arr3x2[3:5,-1:0];
method init();
  put 'ARRAY ASSIGNMENT';

/* Assign array of constants to array s1 */
s1 := ('abc', 'def', 'ghi', 'jkl');

/* Assign array s to array s2 */
s2 := s1;
  put s2[1]= ;
  put s2[2]= ;
  put s2[3]= ;
  put s2[4]= ;
  put;

/* Assign array of constants to array d1. Use * iterators for repeated values. Mismatched * types will be converted automatically. */
d1 := (3*3.14159, 2*'5', 2*(1,2), 99);

/* Assign array d to array e */
d2 := d1;
  put d2[1]= ;
  put d2[2]= ;
  put d2[3]= ;
  put d2[4]= ;
  put d2[5]= ;
  put d2[6]= ;
  put d2[7]= ;
  put d2[8]= ;
  put d2[9]= ;
  put d2[10]=;
  put;

/* Assign array of constants to array arr3x2 */
arr2x3 := (2*(1,2,3));

/* Assign arr2x3 to arr3x2 */
arr3x2 := arr2x3;
  put arr2x3[1,1]= arr3x2[3,-1]= ;
  put arr2x3[1,2]= arr3x2[3,0]= ;
  put arr2x3[1,3]= arr3x2[4,-1]= ;
  put arr2x3[2,1]= arr3x2[4,0]= ;
  put arr2x3[2,2]= arr3x2[5,-1]= ;
  put arr2x3[2,3]= arr3x2[5,0]= ;
end;
enddata;
Example Output: Arrays

The following lines are written to the SAS log.

```
BASIC ARRAYS
strarray[1]=abc
strarray[2]=def
strarray[3]=ghi
strarray[4]=jkl

str=abc
dblarray[1]=3

intarray[1]=1
intarray[2]=2
intarray[3]=3
intarray[4]=4
intarray[5]=5
intarray[6]=6
intarray[7]=7
intarray[8]=8
intarray[9]=9
intarray[10]=10

dblarray[3]=5

a=100 b=101 c=102

NOTE: Execution succeeded. No rows affected.

ERROR: [HY000] Index 0 out of bounds for array dblnegsubarray. (0x817ff04c)
ERROR: [HY000] Index -5 out of bounds for array dblnegsubarray. (0x817ff04c)

ARRAY ASSIGNMENT
s2[1]=abc
s2[2]=def
s2[3]=ghi
s2[4]=jkl

d2[1]=3.14159
d2[2]=3.14159
d2[3]=3.14159
d2[4]=5
d2[5]=5
d2[6]=1
d2[7]=2
d2[8]=1
d2[9]=2
d2[10]=99

arr2x3[1,1]=1 arr3x2[3,-1]=1
arr2x3[1,2]=2 arr3x2[3,0]=2
arr2x3[1,3]=3 arr3x2[4,-1]=3
arr2x3[2,1]=1 arr3x2[4,0]=1
arr2x3[2,2]=2 arr3x2[5,-1]=2
arr2x3[2,3]=3 arr3x2[5,0]=3
```
Example: SELECT Statement

Example Overview: SELECT Statement

This example illustrates the SELECT statement. In this example, a DO loop encloses a
SELECT statement. The SELECT statement reads the current value of the loop counter I
and outputs a character when the WHEN statement is true. The REPEAT statement
repeatedly prints the character based on the value of the loop counter.

Example Code: SELECT Statement

proc ds2;
data;
   dcl char(10) s;
   method run();
      dcl char(1) x;
      dcl int i;
      do i=1 to 10;
         select(i);
            when(1) x='A';
            when(2) x='B';
            when(3) x='C';
            when(4) x='D';
            when(5) x='E';
            when(6) x='F';
            when(7) x='G';
            when(8) x='H';
            when(9) x='I';
            otherwise x='J';
         end;
      s=repeat(x,i);
      output;
   end;
end;
enddata;
run;
quit;
Example Overview: GOTO and LEAVE Statements with Labels

This example presents three programs that show branching. The first uses GOTO to branch to a label. The second uses LEAVE without a label, and the third uses LEAVE with a label. For more information, see “GOTO Statement” on page 899 and “LEAVE Statement” on page 904.

Example Code: GOTO and LEAVE Statements with Labels

```sas
proc ds2;
data;
dcl double i j;
method init();
i = 1;
head:
j = 2*i;
i = i+1;
if i < 10 then do;
   output;
goto head;
```

Example: GOTO and LEAVE Statements with Labels

```plaintext
end;
end;
enddata;
run;
data _null_;
dcl int x y;
method init();
    put 'loop test 1';
    x = 1;
    y = 2;
    if (x ~= -5) then
        do i = 1 to 10;
            put i;
            if i > 4 then leave;
        end;
    else
        put 'else';
    end;
end;
enddata;
run;
data _null_;  
dcl int g;
method init();
    put 'label test 1';
    x = 1;
    y = 2;
    if (x > -5) then
        lab:
            do i = 1 to 10;
                do j = 1 to 5;
                    put i j;
                    if i > 4 then leave lab;
                end;
            end;
    else
        do;
            put 'else';
        end;
    end;
end;
enddata;
run;
quit;
```
Example Output: GOTO and LEAVE Statements with Labels

The following output is the result of using GOTO to branch to a label.

The SAS System

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

These lines from the loop test are written to the SAS log.

```
loop test 1
1
2
3
4
5
```

These lines from the label test are written to the SAS log.

```
label test 1
1
1
1
1
1
2
1
2
1
2
2
2
3
2
4
2
5
3
1
3
2
3
3
4
3
5
4
1
4
2
4
3
4
4
4
4
5
5
1
```
Example: Overloaded Methods

Example Overview: Overloaded Methods

This example illustrates how to set up overloaded methods. For more information about methods, see the “METHOD Statement” on page 910.

Example Code: Overloaded Methods

```plaintext
proc ds2;
data _null_;  
method concat (nvarchar(200) x, nvarchar(200) y) returns nvarchar(400);
  return x || y;
end;

method concat (nvarchar(200) x, nvarchar(200) y, nvarchar(200) z)
  returns nvarchar(600);
  return x || y || z;
end;

method run();
  y = concat(n'abc', n'def');
  put 'y= ' y;
  y = concat(n'abc', n'def', n'ghi');
  put 'y= ' y;
end;
enddata;
run;
data _null_;  
method d() returns double;
  return 99;
end;

method d(double x, double y, double z) returns double;
  return x + y + z;
end;

method d(int x, int y, int z) returns int;
  return x + y + z + 500;
end;

method run();
  dcl double d;
  d = d(1,2,3);
  put 'd= ' d;
  d = d(100.1, 100.2, 100.3);
  put 'd= ' d;
end;
enddata;
run;
quit;
```
Example Output: Overloaded Methods

The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>abcded</td>
</tr>
<tr>
<td>y</td>
<td>abcdedfgh</td>
</tr>
<tr>
<td>d</td>
<td>506</td>
</tr>
<tr>
<td>d</td>
<td>300.6</td>
</tr>
</tbody>
</table>

Example: Analyze a Table Using Multiple Threads

Example Overview: Analyze a Table Using Multiple Threads

This example shows the creation of a table that is then analyzed using multiple threads.

Example Code: Analyze a Table Using Multiple Threads

```sas
libname spde spde 'c:\temp\spde';
proc delete data=spde.incomes; run;
proc delete data=spde.results1; run;
proc delete data=spde.results2; run;
proc ds2;
data spde.incomes;
dcl double income citycode;
dcl char(8) name;
method run();
do j = 1 to 1E6;
  name = 'John'; income = 23234; citycode=1; output;
  name = 'Jane'; income = 62348; citycode=1; output;
  name = 'Joe'; income = 32932; citycode=2; output;
  name = 'Jan'; income = 58239; citycode=2; output;
  name = 'Josh'; income = 6523; citycode=3; output;
  name = 'Jill'; income = 80392; citycode=3; output;
/* The three people to find during mining */
  if j = 5E5 then do;
    name = 'James'; income = 103243; citycode=1; output;
  end;
  if j = 7E5 then do;
    name = 'Joan'; income = 233923; citycode=2; output;
  end;
  if j = 8E5 then do;
    name = 'Joyce'; income = 132443; citycode=3; output;
  end;
end;
enddata;
runc;
```
thread score;
  method run();
  set spde.incomes;

  accept = 0;
  if citycode = 1 and income > 100000 then accept = 1;
  else if citycode = 2 and income > 200000 then accept = 1;
  else if citycode = 3 and income > 120000 then accept = 1;

  /* faux work */
  do i = 1 to 50; end;

  if accept then output;
end;
endthread;
run;

data spde.results1;
  dcl thread score score;
  method run();
  set from score threads=1;
end;
enddata;
run;

data spde.results2;
  dcl thread score score;
  method run();
  set from score threads=2;
end;
enddata;
run;
quit;

Example Output: Analyze a Table Using Multiple Threads

The following lines are written to the SAS log.

```
NOTE: Execution succeeded. 6000003 rows affected.
NOTE: Execution succeeded. No rows affected.
NOTE: Execution succeeded. 3 rows affected.
NOTE: Execution succeeded. 3 rows affected.
```
Example: Using Four Threads to Compute Summary Statistics

Example Overview: Using Four Threads to Compute Summary Statistics

The following example creates a thread program that computes some summary statistics. The thread program is run in four threads. The program looks at which thread outputs which values.

Example Code: Using Four Threads to Compute Summary Statistics

```plaintext
/* Expand sashelp.class */
data class;
  do i = 1 to 1E5;
    do j = 1 to nobs;
      set sashelp.class nobs=nobs point=j;
      output;
    end;
  end;
stop;
run;

proc ds2;
/* Create the thread program
   * When the thread program executes in N threads, each thread receives
   * a unique set of rows from the table work.class.
   * This thread program sums all the student ages it sees along
   * with counting the number of men and women it sees.
   * Each thread's partial sum is output to a data program for
   * final summing.
   */
thread sum_student_measures / overwrite=yes;
  dcl double id cnt sum_age cnt_male cnt_female;
  keep id cnt sum_age cnt_male cnt_female;
  method run();
    set class;
    sum_age + age;
    cnt_male + (if sex = 'M' then 1 else 0);
    cnt_female + (if sex = 'F' then 1 else 0);
  end;
  method term();
    /* _threadid_ is the thread's "number" from 0 to N-1, when using N threads. */
    id = _threadid_;  
    cnt = _N_; 
    output;
  end;
endthread;
```
/* Start the thread program in 4 threads, sum the values
* received and output averages and total counts.
* The variable ID is the thread number for the thread that
* produced a particular set of counts. This is useful
* in looking at which thread output which values.
* Note in the output how the PUT statement output isn't ordered
* by ID. This is because some threads finish sooner than others.
* Also notice the variable CNT which indicates that different
* threads operate on different numbers of rows.
*/
data class_counts / overwrite=yes;
dcl thread sum_student_measures t();
dcl double tot_cnt avg_age tot_male tot_female tot_age;
keep tot_count avg_age tot_male tot_female;
method run();
   set from t threads=4;
   put id= cnt= sum_age= cnt_male= cnt_female=;
   tot_age + sum_age;
   tot_male + cnt_male;
   tot_female + cnt_female;
   tot_cnt + cnt;
end;
method term();
   avg_age = tot_age / tot_cnt;
   output;
end;
enddata;
run;
quit;
proc print data=class_counts; run;

Example Output: Using Four Threads to Compute Summary Statistics
The following lines are written to the SAS log. Note that every time you run the
program, the log output is different.

| id=2 | cnt=1 | sum_age=0 | cnt_male=0 | cnt_female=0 |
| id=0 | cnt=1404506 | sum_age=18702097 | cnt_male=739215 | cnt_female=665290 |
| id=3 | cnt=1 | sum_age=0 | cnt_male=0 | cnt_female=0 |
| id=1 | cnt=495496 | sum_age=6597903 | cnt_male=260785 | cnt_female=234710 |

The following table is output.

<table>
<thead>
<tr>
<th>Obs</th>
<th>avg_age</th>
<th>tot_male</th>
<th>tot_female</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.3158</td>
<td>1000000</td>
<td>900000</td>
</tr>
</tbody>
</table>
Example: Data Cleaning

Example Overview: Data Cleaning

Sometimes, due to input error, data values might have leading and trailing blanks. Unless you specifically filter for this possibility, joins and similar database operations that depend on identically formatted data values will not perform correctly. Another problem that can occur is unintentional duplication of records (that is, multiple records (rows) with the same key). This example shows programs that remove blanks from data values, and list duplicate rows for potential deletion.

The first DS2 program creates a simple data table, EMPLOYEES1, that contains duplicate rows and string values, some of which contain blanks.

The second program uses the STRIP function to remove leading and trailing blanks from values in the EMP column. The table is written to EMPLOYEES2.

The third program uses an embedded SQL SELECT statement to display the duplicates. You can use this output to determine which records should be deleted. Note that this program would have not generated correct output if you did not first remove the blanks from the data values, because the SELECT statement would not have grouped the data accurately.

Example Code: Data Cleaning

```plaintext
proc ds2;
data employees1 (overwrite=yes);
dcl double id;
dcl char emp;
method init();
id = 60918 ; emp = 'user1'; output;
id = 60919 ; emp = 'user2'; output;
id = 60920 ; emp = 'user3'; output;
id = 60918 ; emp = 'user1'; output;
id = 60922 ; emp = 'user4'; output;
id = 60925 ; emp = 'user5 '; output;
id = 60926 ; emp = 'user6 '; output;
id = 60919 ; emp = 'user2'; output;
id = 60928 ; emp = 'user7'; output;
id = 60918 ; emp = 'user1'; output;
end;
enddata;
run;
data employees2 (overwrite=yes);
method run();
set employees1;
emp = strip(emp);
end;
enddata;
run;
data ;
```
Example Output: Data Cleaning

The SAS System

<table>
<thead>
<tr>
<th>DUPID</th>
<th>DUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>60918</td>
<td>3</td>
</tr>
<tr>
<td>60919</td>
<td>2</td>
</tr>
</tbody>
</table>

Example: SUBSTR Function

Example Overview: SUBSTR Function

The example shows how to use the SUBSTR function. SUBSTR is used to convert the word CAT to DOG by changing one character at a time.

Example Code: SUBSTR Function

```sas
proc ds2;
  data _null_;  
    method init();
      a = 'cat';
      put 'a=' a;

      substr(a,2,1) = 'o';
      put 'a=' a;

      substr(a,1,1) = 'd';
      put 'a=' a;

      substr(a,3,1) = 'g';
      put 'a=' a;

      b = a;
    end;
  enddata;
run;
```
Example Output: SUBSTR Function

The following lines are written to the SAS log.

a= cat
a= cot
a= dot
b= dog

Example: PUT with SAS Formats

Example Overview: PUT with SAS Formats

This example illustrates how to use SAS formats with the PUT statement.

Example Code: PUT with SAS Formats

```sas
proc ds2;
data _null_
method init();
dcl double d;
dcl int i;
dcl char(32) x;
d = 99;
x = put(d, best12.6);
put 'x=' x;
d = 100;
x = put(d, best.);
put 'x=' x;
i = 10847;
x = put(i, mmdy8.);
put 'x=' x;
c = 'abc';
x = put(c, $char8.);
put 'x=' x;
end;
enddata;
run;
quit;
```
**Example Output: PUT with SAS Formats**

The following lines are written to the SAS log.

<table>
<thead>
<tr>
<th>x=           99</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=          100</td>
</tr>
<tr>
<td>x=  09/12/89</td>
</tr>
<tr>
<td>x=    abc</td>
</tr>
</tbody>
</table>

---

**Example: Generate Statistics from Table Data**

**Example Overview: Generate Statistics from Table Data**

This example analyzes the price movement of a simulated stock table. The first DS2 program creates the stock price table. There are two weeks of the open, high, low, and close prices for the stock.

The second program performs the analysis. A 2–day moving average of closing prices is created by assigning the previous two days' closing prices to elements of the CY array, averaging them by using the MEAN function, and then sending them to output as C_MA. The INIT method initializes the array CY and C_MA variables to the first price in the STOCK table. The RUN method assigns values to the array as it loops through the table.

The CTR variable is incremented each time through the RUN method. The moving average is not valid until two days of prices have been averaged, so the output begins with the third record (that is, when CTR > 2). After a row has been output, the CY array is updated.

The following statistics are also calculated:

- the daily change in closing price, Chng, calculated by subtracting yesterday's close, CY[1], from today's close
- the change from open to the closing price, O_C
- the range of the day's prices, H_L, calculated by subtracting the low price from the high price

**Example Code: Generate Statistics from Table Data**

```sas
proc ds2;
data stock (overwrite=yes);
dcl date d;
dcl double o h l c;
method init();
  d = date '2010-09-18' ; o =  20; h = 22.25; l = 18; c = 21.5; output;
  d = date '2010-09-19' ; o =  21; h = 23.5; l = 19.25; c = 22; output;
  d = date '2010-09-20' ; o =  22.25; h = 24.75; l = 20; c = 21; output;
  d = date '2010-09-21' ; o =  21; h = 21.5; l = 18.75; c = 19; output;
  d = date '2010-09-22' ; o =  18; h = 19; l = 17.25; c = 17; output;
```
Example Output: Generate Statistics from Table Data

The SAS System

<table>
<thead>
<tr>
<th>C_MA</th>
<th>H_L</th>
<th>O_C</th>
<th>Chng</th>
<th>C</th>
<th>L</th>
<th>H</th>
<th>O</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.75</td>
<td>4.75</td>
<td>1.25</td>
<td>-1</td>
<td>21</td>
<td>20</td>
<td>24.75</td>
<td>22.25</td>
<td>20SEP2010</td>
</tr>
<tr>
<td>21.5</td>
<td>2.75</td>
<td>2</td>
<td>-2</td>
<td>19</td>
<td>18.75</td>
<td>21.5</td>
<td>21</td>
<td>21SEP2010</td>
</tr>
<tr>
<td>20</td>
<td>1.75</td>
<td>1</td>
<td>-2</td>
<td>17</td>
<td>17.25</td>
<td>19</td>
<td>18</td>
<td>22SEP2010</td>
</tr>
<tr>
<td>18</td>
<td>2.5</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>15.5</td>
<td>18</td>
<td>17</td>
<td>25SEP2010</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>-0.5</td>
<td>1</td>
<td>18</td>
<td>16</td>
<td>20</td>
<td>17.5</td>
<td>26SEP2010</td>
</tr>
<tr>
<td>17.5</td>
<td>3</td>
<td>-1.5</td>
<td>2</td>
<td>20</td>
<td>18</td>
<td>21</td>
<td>18.5</td>
<td>27SEP2010</td>
</tr>
<tr>
<td>19</td>
<td>2.75</td>
<td>-1</td>
<td>1</td>
<td>21</td>
<td>19.5</td>
<td>22.25</td>
<td>20</td>
<td>28SEP2010</td>
</tr>
<tr>
<td>20.5</td>
<td>4.25</td>
<td>-0.5</td>
<td>0.5</td>
<td>21.5</td>
<td>18.75</td>
<td>23</td>
<td>21</td>
<td>29SEP2010</td>
</tr>
</tbody>
</table>
Example: Matrices and Non-linear Equations

Example Overview: Matrices and Non-linear Equations

This example solves a system of non-linear equations using Newton’s iterative process. In this example, the root (to within a given epsilon) is found after five iterations of the loop, and is equal to (.5, 0, -.5236). Zero appears as an extremely small number.

One interesting feature to note in Newton’s example is how the result matrix can be reused. In a simple method calculation, \( r = \text{m.mult}(\text{m2}) \); the result matrix instance is automatically created. However, if the result instance already exists, and its size is correct, the method will reuse the result matrix for efficiency purposes. Note how this is done in Newton’s computation loop. When the matrix is created the first time, the left side result is reused in computations such as \( \text{ji} = \text{jm.inverse()} \). Otherwise, you would have to free the old matrix and allocate a new one each time.

Example Code: Matrices and Non-linear Equations

```plaintext
proc ds2;
/*
 * Solve the system of equations
 * 3*x1 + cos(x2*x3) - .5 = 0
 * x1^2 - 81*(x2 + .1)^2 + sin(x3) + 1.06 = 0
 * e^(-x1*x2) + 20*x3 + (10pi-3)/3 = 0
 * using Newton’s method:
 * X_m = X_m0 - J^-1(X_m0) * F(X_m0)
 */
data _null_;
  /* Infinity norm */
  method norm(double x[3]) returns double;
    return max(abs(x[3]), max(abs(x[1]), abs(x[2])));
  end;
  /* Pi computation */
  method pi() returns double;
    return atan(1)*4;
  end;
/*
 * f1(x1, x2, x3)
 * f2(x1, x2, x3)
 * f3(x1, x2, x3)
*/
```
method compute_f(double f[3,1], double x[3]);
    f[1,1] = 3*x[1] - cos(x[2]*x[3]) - .5;
    f[2,1] = x[1]**2 - 81*(x[2]+.1)**2 + sin(x[3]) + 1.06;
    f[3,1] = exp(-x[1]*x[2]) + 20*x[3] + (10*pi() - 3)/3.0;
end;

/*
 *      Jacobian array
 *     @f1() @f1() @f1()
 *     ---   ---   ---
 *     @x1   @x2   @x3
 *     @f2() @f2() @f2()
 *     ---   ---   ---
 *     @x1   @x2   @x3
 *     @f3() @f3() @f3()
 *     ---   ---   ---
 *     @x1   @x2   @x3
 */

method compute_j(double j[3,3], double x[3]);
    j[1,1] = 3;
    j[1,2] = x[3]*sin(x[2]*x[3]);
    j[1,3] = x[2]*sin(x[2]*x[3]);
    j[2,1] = 2*x[1];
    j[2,2] = -162*(x[2]+.1);
    j[2,3] = cos(x[3]);
    j[3,1] = -x[2]*exp(-x[1]*x[2]);
    j[3,2] = -x[1]*exp(-x[1]*x[2]);
    j[3,3] = 20;
end;

method init();
dcl double j[3,3];
dcl double f[3,1];
dcl double y[3,1];
dcl double x[3];
dcl double x0[3];
dcl double d[3];
dcl package matrix jm;
dcl package matrix ji;
dcl package matrix fm;
dcl package matrix ym;
dcl package matrix xm0;
dcl package matrix xm;
dcl package matrix diff;
dcl int niter;
dcl double eps;

/* Instantiate matrices */

jm = _new_ matrix(3, 3);
fm = _new_ matrix(3, 1);
xm0 = _new_ matrix(3, 1);

/* Initial approximation */

x0[1] = .1;

x0[2] = .1;

x0[3] = -.1;

/* Start loop */

eps = 1;

niter = 0;

put '(', x0[1], ',', x0[2], ',', x0[3], ')';

do while(eps > 10**-6 and niter < 10);

/* Compute functions with current approximation : j(x_0), f(x_0) */

compute_j(j, x0);

compute_f(f, x0);

/* Load arrays into matrices */

jm.load(j);

fm.load(f);

xm0.load(x0);

/* Find inverse of Jacobian matrix */

ji = jm.inverse();

/* Multiply by function vector */

ym = ji.mult(fm);

/* Compute next approximation */

xm = xm0.sub(ym);

/* Compute error term */

xm.toarray(x);

diff = xm.sub(xm0);

diff.toarray(d);

eps = norm(d);

x0 := x;

put '(', x[1], ',', x[2], ',', x[3], ')';

put eps=;

niter + 1;

end;

put niter=;


end;
Example Output: Matrices and Non-linear Equations

The following results appear in the log:

```
{ 0.1 , 0.1 , -0.1 }
{ 0.49986967292642 , 0.01946684853741 , -0.52152047193583 }
eps=0.42152047193583
{ 0.500000000000707 , 7.7578571058927E-10 , -0.52359877559829 }
eps=7.7578571271436E-10
niter=5
root = ( 0.5 , -2.1250842759061E-18 , -0.52359877559829 )
```

Example: DS2 Matrices and Regression Analysis

Example Overview: DS2 Matrices and Regression Analysis

This example uses DS2 matrices to find the parameter estimate for a regression analysis.

Example Code: DS2 Matrices and Regression Analysis

```sas
libname z 'c:\mylib';

data _null_;
  dsid = open('z.hmeq4');
  nobs = attrn(dsid, 'nobs');
  call symput('nobs', nobs);
  hmnv = attrn(dsid, 'nvars');
  call symput ('hmnv', hmnv);
run;

/*
* This does a regression approximation of the hmeq model variable
* MORTDUE using the 'independent' variables CLAGE, CLNO, DELINQ,
* DEROG, NINQ, VALUE and YOJ.
*
* We set up a matrix X with the data for the independent variables,
* and a matrix Y with the dependent data, and then solve for the linear
* coefficients b in y = X * b:
*     X'*y = X'*X*b
*     (X'*X)^-1 * X'*y = b
*/
proc ds2;
data y(overwrite=yes);
  vararray double s[&hmnv] i clage clno delinq derog ninq value yoj;
  vararray double m[1] mortdue;

method init();
  dcl package matrix x ym xty xtx ti xt bm;
  dcl double b[&hmnv];
  /* Create the hmeq and mortdue matrices */
  x = _new_ matrix(&nobs, &hmnv);
  y = _new_ matrix(&nobs, 1);
  /* Read the data from the hmeq table */
  do j = 1 to &nobs
    set z.hmeq4;
    x.in(s, j);
  end;
  /* x now contains the observations for the variables in hmeq -
  * plus a column of 1's for the constant term in the approximation:
  *  i   clage     clno    delinq   derog     ninq     value      yoj
  *  1   94.367   9.0000   0.0000  0.00000   1.0000   39025.00  10.5000
  *  1  121.833  14.0000   2.0000  0.00000   0.0000   68400.00   7.0000
  *  1  149.467  10.0000   0.0000  0.00000   1.0000   16700.00   4.0000
  *  1  179.766  21.2961   0.4494  0.25457   1.1861  101776.05   8.9223
  *  
  * etc.
  */
  /* Read the data from the mortdue table */
  do j = 1 to &nobs
    set z.mortdue;
    y.in(m, j);
  end;
  /* y is now a column vector containing the known values of MORTDUE */
  /* Make sure the row count matches */
  xr = x.rows();
  er = y.rows();
  if (xr ne er) then do;
    put 'invalid data';
    stop;
  end;
  /* Compute ti = (X'*X)^-1 */
  xt = x.trans();
  xtx = xt.mult(x);
\[ \mathbf{t} = \mathbf{x} \mathbf{t}^{-1}; \]

\[ \begin{align*} /* \text{ Compute } y_{m} &= \mathbf{X}' \mathbf{y} */ \quad \mathbf{y}_{m} &= \mathbf{x}t \cdot \mathbf{y}; \\
/* \text{ Compute } \mathbf{b} &= (\mathbf{X}' \mathbf{X})^{-1} \cdot \mathbf{X}' \mathbf{y} */ \quad \mathbf{b}_{m} &= \mathbf{t} \cdot \mathbf{y}_{m}; \end{align*} \]

\[ /* \text{ Thus } */ \\
/* \text{ MORTDUE} = b_{0} + b_{1} \cdot \text{CLAGE} + b_{2} \cdot \text{CLNO} + \\
\quad b_{3} \cdot \text{DELINQ} + b_{4} \cdot \text{DEROG} + b_{5} \cdot \text{NINQ} + \\
\quad b_{6} \cdot \text{VALUE} + b_{7} \cdot \text{YOJ} + \varepsilon; */ \\
/* \text{ The } \mathbf{b} \text{ vector is equivalent to the parameter estimate from } */ \\
/* \text{ proc reg; model mortdue=} \text{ clage clno delinq derog ninq value yoj;} \text{run; } */ \\
/* \text{ Variable} \quad \text{DF} \quad \text{Parameter Estimate} \quad \text{F Value} 1434.92 */ \\
/* \text{ Intercept} \quad 1 \quad 11473 \\
\text{ clage} \quad 1 \quad -1.64128 \\
\text{ clno} \quad 1 \quad 466.74925 \\
\text{ delinq} \quad 1 \quad -46.87948 \\
\text{ derog} \quad 1 \quad -1371.68909 \\
\text{ ninq} \quad 1 \quad 544.05978 \\
\text{ value} \quad 1 \quad 0.56118 \\
\text{ yoj} \quad 1 \quad -530.88585 */ \\
\]

\[ \begin{align*} &\mathbf{b}_{m} \cdot \text{toarray} (\mathbf{b}); \\
&\text{do } i = 1 \text{ to } &\text{&hmnv} \\
&\quad \text{put } \mathbf{b}[i]; \\
&\quad \text{end;} \\
&\text{end;} \\
&\text{enddata;} \\
&\text{run;} \\
&\text{quit;} \\
\]

**Example Output: DS2 Matrices and Regression Analysis**

The following results appear in the log.

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11472.5599166895</td>
</tr>
<tr>
<td>-1.64128448706739</td>
</tr>
<tr>
<td>466.749252255557</td>
</tr>
<tr>
<td>-46.879474759862</td>
</tr>
<tr>
<td>-1371.68908534735</td>
</tr>
<tr>
<td>544.05979616903</td>
</tr>
<tr>
<td>0.561175479455582</td>
</tr>
<tr>
<td>-530.88585193411</td>
</tr>
</tbody>
</table>
Example: Use the SQLSTMT Package in Another Package

Example Overview: Use the SQLSTMT Package in Another Package

This example shows how to use an instance of a package SQLSTMT in another package. It also shows the use of the _NEW_ operator to delay construction of the SQLSTMT instance until after the creation of the table that is referenced by the FedSQL statement. If the SQLSTMT instance was constructed before the referenced table was created, then the prepare of the FedSQL statement fails in the constructor.

Example Code: Use the SQLSTMT Package in Another Package

```plaintext
class fibonacci {
  int nMax;

  method fibonacci(int n) {
    nMax = n;
  }

  method output(char(100) tablename) {
    int n;
    double fib;

    sqlstmt stmt;

    sqlexec('create table ' || tablename || ' (n int, fib double)');

    stmt = _new_ sqlstmt('insert into ' || tablename || ' values (?, ?)').

    stmt.setInteger(1, 0);   // column n
    stmt.setDouble(2, 0);    // column fib
    stmt.execute();

    stmt.setInteger(1, 1);   // column n
    stmt.setDouble(2, 2);    // column fib
    stmt.execute();

    first = 0;
    second = 1;

    do n = 2 to nMax;
      fib = first + second;
  }
}
```
The following code block creates an instance of package FIBONACCI, and the Fibonacci instance is used to generate an output table of a Fibonacci series.

```plaintext
data _null_; method init(); declare package fibonnaci fibseries(20); fibseries.output('fibdata'); end; enddata; run;
```

---

**Example: Update the Values of a Table By Using Two Databases**

**Example Overview: Update the Values of a Table By Using Two Databases**

The following example illustrates how the SQLSTM package can facilitate updating a table in one database based on the values in another table in a second database. In the example program, stmt1 queries for all the x and y columns from the ORACLE table db1.dataset1. Then for each rowset in the result set from db1.dataset1, stmt2 finds the rows in BASE table db2.dataset2 that have the same x column values as the x value read from db1.dataset1. Stmt2 then updates the BASE table db2.dataset2 y column values of the found rows to be the same as the y value read from db1.dataset1.

**Example Code: Update the Values of a Table By Using Two Databases**

```plaintext
libname db1 odbc user=XXXX pw=XXXX dsn=exadat;
libname db2 './base';
proc ds2;
data _null_; method run();
declare double x y;
dcl package sqlstmt stmt1('select x,y from db1.dataset1');
dcl package sqlstmt stmt2('update db2.dataset2 set y=? where x=?', [y x]);
stmt1.execute();
stmt1.bindResults([x y]);
do while (stmt1.fetch() = 0);
```
Example: Store FedSQL Statements in a Hash Package

Example Overview: Store FedSQL Statements in a Hash Instance

The following example shows how to use a hash package to manage a set of FedSQL statements. A set of FedSQL statements is dynamically allocated and stored in a hash package. The hash package is indexed by an integer key which is used to retrieve the appropriate FedSQL statement from the hash package.

Example Code: Store FedSQL Statements in a Hash Package

```plaintext
proc ds2;
/* Create 5 tables testdata1...testdata5.
* Each table has 3 double columns (x,y,z) and no observations. */
data _null_;  
method init();
declare int i;
declare int rc;

do i = 1 to 5;
    rc = sqlexec('create table testdata' || i || '
'(x double, y double, z double)');
    if (rc ne 0) then put 'TEST FAILED';
end;
end;
enddata;
run;
quit;

proc ds2;
data _null_;  
/* Create an sqlstmt reference. 
* Note: does NOT create an sqlstmt instance. */
declare package sqlstmt s;
/* Create a hash instance. */
declare package hash h();

/* Hash key: sqlstmts are accessed by index 1..5. */
declare int i;
/* Variables to bind to sqlstmt parameters. */
declare double u v w;
```
/* Create 5 sqlstmts to insert data in tables * testdata1...testdata5. Store the sqlstmts in hash * table h. */

method init();
declare int rc;

h.definekey('i');    /* Key is index 1..5. */
h.definedata('s');   /* Data is an sqlstmt reference. */
h.definedone();

/* Dynamically create 5 sqlstmt instances and add * each sqlstmt to hash table. */
do i = 1 to 5;

    /* Dynamcically create an sqlstmt in global ([this]) scope. * Variables [u v w] are bound to the parameters of * the sqlstmt. */
    s = _new_ [this] sqlstmt('insert into testdata' || i || ' values (?,?,?)', [u v w]);

    /* Add the sqlstmt to hash h with key i. */
    rc = h.add();
    if (rc ne 0) then put 'TEST FAILED';
end;
end;

/* Retrieve the sqlstmts from the hash table and execute * the sqlstmts to insert observations in the tables. */

method run();
declare int j;
declare int rc;

/* For each of 5 data tables testdata1...testdata5, */
/* insert 9 observations. */
do j = 1 to 9;
do i = 1 to 5;

    /* Find the sqlstmt in hash by index i. */
    rc = h.find();
    if (rc ne 0) then put 'TEST FAILED';

    /* Set values to insert into the table testdata i. */
    u = i;        /* data for 1st column (x) */
    v = -j;       /* data for 2nd column (y) */
    w = i*10+j;   /* data for 3rd column (z) */

    /* Execute the sqlstmt to insert the observation. */
    rc = s.execute();
    if (rc ne 0) then put 'TEST FAILED';

    /* Explicitly close the result set to free resources. * If not explicitly closed, result set would be * automatically closed at next execute of the sqlstmt. */
rc = s.closeResults();
if (rc ne 0) then put 'TEST FAILED';
end;
end;
end;

/* Explicitly delete each sqlstmt instance. Note if not
* explicitly deleted, the sqlstmt instances would automatically
* be deleted when their enclosing scope (global) was destroyed. */
method term();
declare int rc;
do i = 1 to 5;
   rc = h.find();
   if (rc ne 0) then put 'TEST FAILED';
   s.delete();
   end;
end;
enddata;
run;
quit;

proc print data=testdata1; quit;
proc print data=testdata2; quit;
proc print data=testdata3; quit;
proc print data=testdata4; quit;
proc print data=testdata5; quit;

The following tables are output.
Output A4.1  Testdata Tables

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>-5</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>-6</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>-7</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>-8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>-9</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>-2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-4</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>-5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>-6</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>-7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>-8</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>-9</td>
<td>29</td>
</tr>
</tbody>
</table>
### Example: Store FedSQL Statements in a Hash Package

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-2</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>-4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-5</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>-6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>-7</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>-8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>-9</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>-1</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>-2</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-3</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-4</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>-5</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>-6</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>-7</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>-8</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>-9</td>
<td>49</td>
</tr>
</tbody>
</table>
Example: Run a DS2 Program from z/OS Batch

Example Overview: Run a DS2 Program from z/OS Batch

The following example illustrates how to run a basic DS2 program from z/OS batch.

Example Code: Run a DS2 Program from z/OS Batch

```
//YOUR JOB BATCH INFO HERE
//TIME=1, NOTIFY=
//*JOBPARM FETCH
//VALID EXEC SAS94
//WORK DD PATH='/u/myuserid/mywork/'
//SYSIN DD *
proc ds2;
  data departure (overwrite=yes);
    dcl char(9) airline;
    dcl char(5) arrtime;
    method init();
      airline='Southwest'; arrtime='10:30'; output;
      airline='Delta'; arrtime='12:45'; output;
      airline='American'; arrtime='11:15'; output;
    end;
  enddata;
  run;
  quit;
  proc print data=departure;
  run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>-1</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>-2</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-3</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-4</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>-5</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>-6</td>
<td>56</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>-7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>-8</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>-9</td>
<td>59</td>
</tr>
</tbody>
</table>
Recommended Reading

- *Base SAS Procedures Guide*
- *Encryption in SAS*
- *SAS FedSQL Language Reference*
- *SAS Formats and Informats: Reference*
- *SAS In-Database Products: Administrator's Guide*
- *SAS Language Reference: Concepts*
- *SAS System Options: Reference*
- SAS offers instructor-led training and self-paced e-learning courses to help you get started with the DS2 programming language, and learn how the DS2 language works with the other SAS products. For more information about the courses available, see [sas.com/training](http://sas.com/training).

For a complete list of SAS publications, go to [sas.com/store/books](http://sas.com/store/books). If you have questions about which titles you need, please contact a SAS Representative:

SAS Books  
SAS Campus Drive  
Cary, NC 27513-2414  
Phone: 1-800-727-0025  
Fax: 1-919-677-4444  
Email: sasbook@sas.com  
Web address: [sas.com/store/books](http://sas.com/store/books)
### Special Characters

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>–</code></td>
<td>106</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1001</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1048</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1044</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1092</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1113</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1130</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1181</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1228</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>1259</td>
</tr>
<tr>
<td><code>NEW</code></td>
<td>857</td>
</tr>
<tr>
<td><code>:=</code></td>
<td>106</td>
</tr>
<tr>
<td><code>!</code></td>
<td>106</td>
</tr>
<tr>
<td><code>!!</code></td>
<td>106</td>
</tr>
<tr>
<td><code>/</code></td>
<td>106</td>
</tr>
<tr>
<td><code>..</code></td>
<td>106</td>
</tr>
<tr>
<td><code>^</code></td>
<td>106</td>
</tr>
<tr>
<td><code>^=</code></td>
<td>106</td>
</tr>
<tr>
<td><code>~</code></td>
<td>106</td>
</tr>
<tr>
<td><code>~=</code></td>
<td>106</td>
</tr>
<tr>
<td><code>( )</code></td>
<td>106</td>
</tr>
<tr>
<td><code>SBASE64Xw. format</code></td>
<td>234</td>
</tr>
<tr>
<td><code>SBINARYw. format</code></td>
<td>235</td>
</tr>
<tr>
<td><code>SCHARw. format</code></td>
<td>236, 243</td>
</tr>
<tr>
<td><code>SFw. format</code></td>
<td>236, 243</td>
</tr>
<tr>
<td><code>SHEXw. format</code></td>
<td>237</td>
</tr>
<tr>
<td><code>SOCTALw. format</code></td>
<td>238</td>
</tr>
<tr>
<td><code>SQUOTEw. format</code></td>
<td>239</td>
</tr>
<tr>
<td><code>SREVERJw. format</code></td>
<td>240</td>
</tr>
<tr>
<td><code>SREVERSw. format</code></td>
<td>241</td>
</tr>
<tr>
<td><code>SUPCASEw. format</code></td>
<td>242</td>
</tr>
<tr>
<td><code>Sw. format</code></td>
<td>236, 243</td>
</tr>
<tr>
<td><code>*</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>**</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>&amp;</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>&lt;&gt;</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>&lt;</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>&lt;=</code> operator</td>
<td>106</td>
</tr>
<tr>
<td><code>%TSLIT</code> macro function</td>
<td>80</td>
</tr>
<tr>
<td><code>+</code> operator</td>
<td>106</td>
</tr>
</tbody>
</table>

### A

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ABORT</code></td>
<td>http package</td>
</tr>
<tr>
<td><code>ABS</code> function</td>
<td>358</td>
</tr>
<tr>
<td><code>ABS</code> method</td>
<td>1134</td>
</tr>
<tr>
<td><code>absolute value sum of, for nonmissing arguments</code></td>
<td>799</td>
</tr>
<tr>
<td><code>ADD</code> method, hash package</td>
<td>1004</td>
</tr>
<tr>
<td><code>ADD</code> method, matrix package</td>
<td>1134</td>
</tr>
<tr>
<td><code>ADDREQUESTHEADER</code> method, http package</td>
<td>1076</td>
</tr>
<tr>
<td><code>ALL_AND</code> method</td>
<td>1137</td>
</tr>
<tr>
<td><code>ALL_EQ</code> method</td>
<td>1138</td>
</tr>
<tr>
<td><code>ALL_GE</code> method</td>
<td>1139</td>
</tr>
<tr>
<td><code>ALL_GT</code> method</td>
<td>1140</td>
</tr>
<tr>
<td><code>ALL_LE</code> method</td>
<td>1141</td>
</tr>
<tr>
<td><code>ALL_LT</code> method</td>
<td>1142</td>
</tr>
<tr>
<td><code>ALL_NE</code> method</td>
<td>1143</td>
</tr>
<tr>
<td><code>ALL_OR</code> method</td>
<td>1144</td>
</tr>
<tr>
<td><code>ANY_LT</code> method</td>
<td>1150</td>
</tr>
<tr>
<td><code>ANY_AND</code> method</td>
<td>1145</td>
</tr>
<tr>
<td><code>ANY_EQ</code> method</td>
<td>1146</td>
</tr>
<tr>
<td><code>ANY_GE</code> method</td>
<td>1147</td>
</tr>
<tr>
<td><code>ANY_GT</code> method</td>
<td>1148</td>
</tr>
<tr>
<td><code>ANY_LE</code> method</td>
<td>1149</td>
</tr>
<tr>
<td><code>ANY_NE</code> method</td>
<td>1151</td>
</tr>
<tr>
<td><code>ANY_OR</code> method</td>
<td>1152</td>
</tr>
<tr>
<td><code>ANYALNUM</code> function</td>
<td>359</td>
</tr>
<tr>
<td><code>ANYALPHA</code> function</td>
<td>362</td>
</tr>
<tr>
<td><code>ANYCNTRL</code> function</td>
<td>364</td>
</tr>
<tr>
<td><code>ANYDIGIT</code> function</td>
<td>366</td>
</tr>
<tr>
<td><code>ANYFIRST</code> function</td>
<td>368</td>
</tr>
<tr>
<td><code>ANYGRAPH</code> function</td>
<td>370</td>
</tr>
<tr>
<td><code>ANYLOWER</code> function</td>
<td>372</td>
</tr>
</tbody>
</table>
ANYNAME function 374
ANYPRINT function 376
ANYPUNCT function 378
ANYSPACE function 381
ANYUPPER function 383
ANYXDIGIT function 385
ARCOS function 386
ARCOSH function 387
arithmetic expressions type conversion 89
array
assignment 128
ARRAY Assignment statement 866
array expressions 96
arrays 121
arguments 131
bounded parameters 132
declaring with a HAVING clause 126
example 1315
loading a matrix package 167
output content 134
querying dimensions 133
temporary 122
unbounded parameters 132
variable 123
ARSIN function 388
ARSINH function 389
ARTAN function 391
ARTANH function 390
Assignment statement 866
ATAN function 391
ATAN2 function 392
audience 5

B
BAND function 393
Bernoulli distributions 473
BESTDOTXw. format 245
BESTDw.p format 246
BESTw. format 244
beta distributions probabilities from 683
BETA function 394
BETAINV function 395
binary constants 69
binary data type 87
BINARYw. format 248
BINDPARAMETERS method 1198
BINDRESULTS method 1199
binomial distributions 473
probabilities from 684
random numbers 731
bit strings 69
bivariate normal distribution probability computed from 685
Black model
call prices for European options on futures 396
put prices for European options on futures 399
Black-Scholes model
call prices for European options on stocks 401
BLACKCLPRC function 396
BLACKPTPRC function 399
blank spaces missing and null values 83
BLKSHCLPRC function 401
BLKSHPTPRC function 403
block
package program 52
block statements 862
blocks 52
BLSHIFT function 405
BNOT function 406
BOR function 406
BRSHIFT function 407
BUFNO table option 972
BUFSIZE table option 973
BULKLOAD table option 974
BXOR function 408
BY statement 937
BY groups identifying beginning and end of 869
processing 870
BY processing 870
BY statement 868
details 869
in DS2 869
BY values
interleaving tables and 207
BY variables
interleaving tables and 205
match-merge with duplicate values of 214
BY-group processing
FIRST.variable 202
IN= table option 986
LAST.variable 202
SET statement 202
BYTE function 408

C
call prices
European options on futures, Black model 396
European options on stocks, Black-Scholes model 401
for European options, based on Margrabe model 616
case changes 1312
cashflow stream, periodic
    convexity for 445
    modified duration for 483
    present value for 727
cashflow, enumerated
    convexity for 444
    modified duration for 481
CAT function 409
CATS function 411
CATT function 413
CATX function 415
CEIL function 419
CEILZ function 420
character constants 68
character data type 87
character functions
    missing and null values in 342
character set 74, 879, 913, 953
character strings
    character position of a word in 499
    counting words in 454
    number of a word in 499
    searching 499
CHECK method 1007
chi-squared distributions
    probabilities 686
CHOOSEC function 421
CHOOSE function 423
cleaning data 1328
CLEAR method 1008
CLOSERESULTS method 1200
CMP function 424
CMPT function 425
COALESCE function 426
COALESCEC function 427
coercible data type 87
CQLS method 1153
column order 190
columns
    data types 71
COMB function 428
combining data 191
combining tables 191
concatenating 195, 200
data relationships 192
interleaving 195, 202
match-merging 196, 212
methods for 194, 200
one-to-one reading 196, 208
preparing tables 197
testing programs 200
troubleshooting 197
COMM Aw.d format 249
COMMAXw.d format 250
COMPARE function 429
COMPBL function 433
COMPFUZZ function 434
complex expressions 95
COMPOUND function 436
COMPRESS function 437
COMPRESS table option 975
concatenating tables 195, 200
    DS2 processing during 200
    SQL for 202
    syntax 200
    with data program 201
concatenation expressions
    type conversion 91
configuration loggers 1301
connection string
    SQLSTMT package 174
CONSTANT function 439
constant lists 70
constants 67
    binary 69
    character 68
    date and time 69
    definition 67
    numeric 67
constants, calculating
    double-precision numbers, largest 441
    double-precision numbers, smallest 442
    Euler constant 441
    exact integer 441
    machine precision 443
    natural base 439
    overview 439
constructors, hash package 146
CONTINUE statement 873
convexity, for enumerated cashflow 444
convexity, for periodic cashflow stream 445
CONVX function 444
CONVXP function 445
COPY method 1154
COS function 447
COSH function 448
cosine
    inverse hyperbolic 387
COUNT function 449
COUNTC function 451
COUNTC function 451
counting
    words in a character string 454
COUNTW function 454
CREATEGETMETHOD method, http
    package 1077
CREATEHEADMETHOD method, http
    package 1081
CREATEPARSER method 1099
CREATEPOSTMETHOD method, http
    package 1082
CREATEWRITER method 1100
CSS function 458
CUMIPMT function 459
CUMPRINC function 460
cumulative interest
in CUMIPMT function 459
CV function 462
cycle index 542

D
data
combining 191
data cleaning 1328
data input
hash package 189
overview 185
SET statement 186
SQLSTMT package 189
data intervals
cycle index 542
DATA method 1009
data output
column order 190
OUTPUT statement 189
SQLEXEC function 189, 786
SQLSTMT package 175, 189
data program 52
concatenating tables 201
data relationships 192
many-to-many 194
many-to-one 193
one-to-many 193
one-to-one 192
data sources
supported 4
DATA statement 862, 874
data type conversion
in functions 342
data types 71
type conversions 87, 1299
data variables
hash package 147
DATASET method 1011
DATDIF function 463
date and time constants 69
date calculations
years between dates 847
DATE function 466
date intervals
seasonal cycle 572
seasonal index 560
date values
aligning output 567
holidays 529
date/time data type 87
DATEAMPMw.d format 254
DATEJUL function 466
DATEPART function 467
dates and times 113
converting DS2 to SAS 114
converting SAS to DS2 113
date, time, and datetime formats 116
date, time, and datetime functions 115
DS2 dates, times, and timestamps 111
local time 177
operations on DS2 dates and times 113
SAS date, time, and datetime values
113
time zone information 176
time zone offset 177
UTC 176
DATETIME function 468
datetime intervals
seasonal cycle 572
datetime values
See dates and times
DATETIMEw.d format 256
DATEw. format 253
DAY function 469
DAYw. format 258
DBCREATE_TABLE_OPTS= table
option 976
DBKEY table option 978
DBNULL table option 979
DDMMYYw. format 258
DDMMYYxw. format 260
DECLARE PACKAGE statement, FCMP
package 997
DECLARE PACKAGE statement, hash
iterator 1020
DECLARE PACKAGE statement, hash
package 1012
DECLARE PACKAGE statement, HTTP
package 1083
DECLARE PACKAGE statement, JSON
package 1101
DECLARE PACKAGE statement, logger
1125
DECLARE PACKAGE statement, matrix
package 1155
DECLARE PACKAGE statement, sqlstmt
1200
DECLARE PACKAGE statement, TZ
package 1251
DECLARE PACKAGE statement, user-
defined packages 884
DECLARE statement 877
scalar variable declaration 881
temporary array variable declaration 881
DECLARE THREAD statement 886
DEFINEDATA method 1021
DEFINEDONE method 1023
DEFINEKEY method 1024
defining key and data variables for a hash package 147
degrees
goingedetic distance input in 519
DELETE method, FCMP package 1001
DELETE method, hash and hash iterator package 1025
DELETE method, http package 1084
DELETE method, logger package 1126
DELETE method, matrix package 1156
DELETE method, SQLSTMT package 1202
delimited identifiers 78
%TSLIT macro function 80
non-Latin characters 80
overview 77
DEQUOTE function 470
descriptive statistic functions 343
DESTROYPARSER method 1102
DESTROYWRITER method 1103
DET method 1157
DEVIANANCE function 472
deviance, computing
Bernoulli distribution 473
binomial distribution 473
Gamma distribution 474
inverse Gaussian (Wald) distribution 474
normal distribution 475
overview 472
Poisson distribution 475
DHMS function 476
DIGAMMA function 477
digital signature 622
DIM function 478
DIVIDE function 480
DO programming block 52
DO statement 862, 887
DOLLARw.d format 262
DOLLARXw.d format 263
double-precision number constants 441, 442
DOWNAMEw. format 264
DROP PACKAGE statement 893
DROP statement 891
DROP table option 979
DROP THREAD statement 893
DS2
BY statement in 869
overview 3
DS2 constructs 94
DS2 processing
during concatenation 200
during interleaving 203
during match-merging 213
during one-to-one reading 209
DS2 variables
predefined 64
DS2_OPTIONS statement 894
MISSING_NOTE option 895
TRACE option 895
TYPEWARN option 896
DS2ACCEL= system option 965
DS2SCOND= system option 966
DTDATEw. format 265
DTMONNYw. format 266
DWTWDATXw. format 268
DTYEARw. format 269
DTYYQw. format 270
Dunnett's one-sided test 700
Dunnett's two-sided test 701
DUPLICATE method 1026
duplicate parameter
hash package with 1017
DUR function 481
DURP function 483
Dw.s format 251
E
EDIV method 1158
effective annual interest rate 485
EFFRATE function 485
EMAX method 1159
EMIN method 1161
EMOD method 1162
EMULT method 1164
ENCRYPT table option 981
ENCRYPTKEY table option 983
ENDDATA statement 896
ENDPACKAGE statement 896
ENDTABLE statement 896
ENDTHREAD statement 896
ENCRYPT table option 981
ENCRIPTKEY table option 983
ENDDATA statement 896
ENDPACKAGE statement 896
ENDTABLE statement 896
ENDTHREAD statement 897
enumerated cashflow
convexity for 444
modified duration for 481
EPOW method 1165
EQ method 1167
ERF function 486
ERFC function 487
Euler constants 441
European options on futures
call prices, based on Black model 396
put prices, based on Black model 399
European options on stocks
call prices, based on Black-Scholes model 401
call prices, based on Margrabe model 616

Index 1353
put prices based on Margrabe model
618
EUROW.d format 272
EUROXw.d format 273
Ew. format 271
exact integer constants 441
EXECUTE method 1203
EXECUTEMETHOD method, http package 1084
EXECUTEMETHODSTREAM method, http package 1085
EXP function 488
EXP method 1168
exponential distributions random numbers 744
expressions 93
arithmetic 89
array 96
complex 95
concatenation 91
executing functions with system expressions 342
expression values by operator function 107
IF 102
IN 97
kinds of 94
LIKE 97
logical 89
method 99
operators in 106
order of precedence 95
package method 100
primary 94
relational 90
SELECT 104
SYSTEM 101
THIS 101
unary 88

F distributions
probabilities from 693
FACT function 488
FCMP package 144
FedSQL
hash package 155
SET statement 188
SQLSTMT package 172
using DS2 package methods as functions 135
using package methods as functions 100
FETCH method 1204
filtering data
missing and null values 83
FIND function 489
FIND method 1027
FIND_NEXT method 1028
FIND_PREV method 1031
FINDC function 492
FINDW function 499
FIRST method 1032
hash iterator package with 1033
FIRST. variable 869
FIRST.variable 202
FLOATw.d format 275
FLOOR function 505
FLOOR method 1169
FLOORZ function 506
FMTINFO function 507
formats 227
categories of 229
date, time, and datetime 116
examples 229
outputting formatted data 228
PUT statement with 1330
specifying 228
syntax 227
user-defined 228
validation of 229
FORWARD statement 898
FRACTw. format 275
function expressions 96
function signature 342
functions 340
categories of 343
CONSTANT 439
data type conversion in 342
date, time, and datetime 115
descriptive statistic functions 343
examples 343, 1314
executing with system expressions 342
missing and null values in character functions 342
missing and null values in numeric functions 342
restrictions on arguments 341
syntax 341
YRDIF 847
futures
call prices for European options on, Black model 396
put prices for European options on, Black model 399
FUZZ function 509
Fw.d format 311

G
GAMINV function 510
<table>
<thead>
<tr>
<th>Gamma distributions</th>
<th>474</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilities from</td>
<td>694</td>
</tr>
<tr>
<td>Gamma function</td>
<td>511</td>
</tr>
<tr>
<td>Garkhclprc function</td>
<td>512</td>
</tr>
<tr>
<td>Garkhptprc function</td>
<td>514</td>
</tr>
<tr>
<td>GCD function</td>
<td>517</td>
</tr>
<tr>
<td>GDE method</td>
<td>1170</td>
</tr>
<tr>
<td>Geodetic distance</td>
<td>517</td>
</tr>
<tr>
<td>in kilometers</td>
<td>519</td>
</tr>
<tr>
<td>in miles</td>
<td>519</td>
</tr>
<tr>
<td>input measured in degrees</td>
<td>519</td>
</tr>
<tr>
<td>input measured in radians</td>
<td>520</td>
</tr>
<tr>
<td>Geodist function</td>
<td>517</td>
</tr>
<tr>
<td>Geomean function</td>
<td>520</td>
</tr>
<tr>
<td>Geomeanz function</td>
<td>522</td>
</tr>
<tr>
<td>GET method</td>
<td>159</td>
</tr>
<tr>
<td>Getbigint method</td>
<td>1204</td>
</tr>
<tr>
<td>Getbinary method</td>
<td>1206</td>
</tr>
<tr>
<td>Getchar method</td>
<td>1207</td>
</tr>
<tr>
<td>Getcolumnncount</td>
<td>1208</td>
</tr>
<tr>
<td>Getcolumnname</td>
<td>1209</td>
</tr>
<tr>
<td>Getcolumntypename</td>
<td>1209</td>
</tr>
<tr>
<td>Getaexponent</td>
<td>1036</td>
</tr>
<tr>
<td>Hash iterator package</td>
<td>157</td>
</tr>
<tr>
<td>First, Next, Prev, and Last</td>
<td>1033</td>
</tr>
<tr>
<td>ORDER parameter</td>
<td>1019</td>
</tr>
<tr>
<td>Hash package</td>
<td>145</td>
</tr>
<tr>
<td>Add and Find methods with 1006</td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>157, 1051</td>
</tr>
<tr>
<td>Constructor</td>
<td>146</td>
</tr>
<tr>
<td>Declaring and instantiating</td>
<td>146</td>
</tr>
<tr>
<td>Defining key and data variables</td>
<td>147</td>
</tr>
<tr>
<td>Duplicate parameter with 1017</td>
<td></td>
</tr>
<tr>
<td>FeSQL query</td>
<td>155</td>
</tr>
<tr>
<td>Implicit variable method</td>
<td>149</td>
</tr>
<tr>
<td>Initialization data</td>
<td>148</td>
</tr>
<tr>
<td>Key summaries</td>
<td>151</td>
</tr>
<tr>
<td>Keys-only</td>
<td>146</td>
</tr>
<tr>
<td>Loading tables into</td>
<td>1016</td>
</tr>
<tr>
<td>Non-unique key and data pairs</td>
<td>150</td>
</tr>
<tr>
<td>Reading data</td>
<td>189</td>
</tr>
<tr>
<td>Replacing and removing data</td>
<td>152</td>
</tr>
<tr>
<td>Saving data to a table</td>
<td>152</td>
</tr>
<tr>
<td>Storing and retrieving data</td>
<td>151, 1015</td>
</tr>
<tr>
<td>Using method calls</td>
<td>147</td>
</tr>
<tr>
<td>Using the <em>NEW</em> operator</td>
<td>149</td>
</tr>
<tr>
<td>Variable list method</td>
<td>149</td>
</tr>
</tbody>
</table>

**H**

| Harmean function    | 523 |
| Harmeanz function   | 525 |
| Has_next method     | 1037|
| Has_prev method     | 1039|
| Hasexp method       | 1036|
| Hash iterator package | 157 |

**Index**

| Examples in output | 64 |
| Goto statement     | 899 |
| with labels        | 1320|
| GT method          | 1171|

---

index
creating an HTTP GET method 159
creating an HTTP HEAD method 159
creating an HTTP POST method 159
declaring and instantiating loggers 161, 1302
overview 158
hyperbolic cosine inverse 387
hyperbolic sine inverse 389
hyperbolic tangent inverse 390
hypergeometric distributions probabilities from 695

I
identifiers 77
blocks 52
delimited 77, 78
overview 77
programs 52
regular 77
scope of 52
support for non-Latin characters 80
variable lifetime 55
variable lookup 53
IEEEw.d format 280
IF expression 102
IF statement, subsetting 900
IF-THEN/ELSE statement 900
IN expressions 97
IN method 1172
IN_OUT parameter 910
IN= table option 986
INDEX function 533
INDEXC function 535
indexes
cycle index 542
seasonal 560
INDEXW function 536
informats 853
DS2 support of 854
examples 855
specifying 854
specifying at run time 539
supplied by SAS 854
syntax 853
user-defined 853, 854
validation of 855
INIT method 958
finding minimums 1307
initialization
non-retained variables 186
retained variables 186, 931, 945
initialization data for hash packages 148
INPUTC function 538
compared to INPUTN function 540
INPUTN function 539
INT function 541
INTCINDEX function 542
INTCK function 545
INTCYCLE function 552
INTDT function 555
interleaving tables 195, 202
comments and comparisons 208
different BY values in each table 207
DS2 processing during 203
duplicate values of BY variable 205
simple interleaving 204
sort requirements 203
syntax 203
internal rate of return 571
as fraction 571
INTFIT function 556
INTGET function 557
INTINDEX function 560
INTNX function 564
aligning date output 567
INTRR function 571
compared to IRR function 586
INTSEAS function 572
INTSHIFT function 576
INTTEST function 578
INTTS function 580
INTZ function 581
inverse Gaussian (Wald) distributions 474
inverse hyperbolic cosine 387
inverse hyperbolic sine 389
inverse hyperbolic tangent 390
INVERSE method 1175
IPMT function 582
IQR function 584
IRR function 585
ISBOOLEANFALSE method 1105
ISBOOLEANTRUE method 1106
ISINTEGER method 1107
ISLABEL method 1108
ISLEVELACTIVE method 1127
ISNULL method 1109
ISNUMERIC method 1110
ISPARTIAL method 1110
ISSTRING method 1112
ITEM_SIZE attribute 1040
hash package with 1051
J
JSON package
- parsing text 163
JSON package 161
  - declaring and instantiating 162
JULDATE function 586
JULDATE7 function 587
JULIANw. format 282

K
KCOUNT function 588
KEEP statement 902
KEEP table option 987
key summaries for hash package 151
key variables
  - hash package 147
keys
  - hash package 150
KEYS method 1041
keys-only hash package 146
kilometers
  - geodetic distance in 519
KSTRCAT function 589
KSTRIP function 590
KUPDATE function 592
KUPDATES function 595
KURTOSIS function 597

L
LABEL table option 988
labels
  - GOTO and LEAVE statements with 1320
Labels, statement 903
LARGEST function 598
LAST method 1042
  - hash iterator package with 1033
LAST. variable 869
LAST.variable 202
latitude
  - geodetic distance between latitude and longitude coordinates 517
LBOUND function 600
LCM function 601
LE method 1176
LEAVE statement 904
  - with labels 1320
LEFT function 602
LENGTH function 603
LENGTHC function 605
LENGTHM function 606
LENGTHN function 603, 607
LGAMMA function 608
LIKE expression 97
LOAD method 1172
local scope 53
local statements 865
local time
  - TZ package 177
local variables 53, 63
  - examples 64
in output 64
LOCKTABLE table option 989
LOG function 609
log message 165
LOG method, logger package 1128
LOG method, matrix package 1178
LOG10 function 611
LOG1PX function 612
LOG2 function 613
LOGBETA function 610
logger package 164
  - declaring and instantiating 165
  - formatted messages 165
  - logging messages to a table 165
  - unformatted messages 165
loggers
  - configuration 1301
  - HTTP package 161, 1302
  - overview 1301
  - run-time 1302
logical expressions
  - type conversion 89
longitude
  - geodetic distance between latitude and longitude coordinates 517
LOWCASE function 614
LOWER function 614
LT method 1179

M
machine precision constants 443
macros 12
MAD function 615
many-one t-statistics, Dunnett’s one-sided test 700
many-one t-statistics, Dunnett’s two-sided test 701
many-to-many relationships 194
many-to-one relationships 193
Margrabe model
  - call prices for European options on stocks 616
  - put prices for European options on stocks 618
MARGRCLPRC function 616
MARGRPTPRC function 618
match-merge 908
match-merging 196, 212
combining rows based on a criterion 213
DS2 processing during 213
duplicate values of BY variable 214
nonmatched rows 216
syntax 212
matrix operations 169
matrix package 167
considerations when using 171
data input 167
data output 169
declaring and instantiating 167
matrix operations 169
MAX function 621
MD5 function 622
MDY function 624
MDYAMPMw.d format 283
MEAN function 625
means
multiple comparisons of 705
MEDIAN function 626
merge rows from one or more tables 188, 939
MERGE statement 906
match-merging 212
merging
match-merging 196, 212
merging rows 906
message digest 622
method expressions 99
method programming block 52
METHOD statement 862, 910
methods 51, 957
dictionary of 957
implicit variable for hash package 149
IN_OUT parameter 910
overloaded 1323
package used as functions in FedSQL 100, 135
system 51
user-defined 51
variable list for hash package 149
miles
geodetic distance in 519
MIN function 627
minimums 1307
MINUTE function 628
MISSING function 629
missing values 81
compared with null values 83
in character functions 342
in numeric functions 342
testing for 86
MISSING_NOTE option 895
MMDDYYw. format 284
MMDDYYx.w format 286
MMSSw.d format 288
MMYYw. format 289
MMYYxw. format 290
MOD function 631
MODZ function 633
MONNAMEw. format 292
MONTH function 635
MONTHw. format 293
MONYYw. format 294
MORT function 635
MULT method 1183
MULTIDATA method 1043

N
N function 637
named variable list 62, 955
NDIMS function 638
NE method 1186
negative binomial distributions
probabilities from 709
NEGPARENw.d format 295
NENGOw. format 296
net present value 640, 669
as fraction 640
as percentage 669
NETPV function 640
NEXT method 1049
hash iterator package with 1033
NLS
transcoding failures 190
NMISS function 641
NOMRATE function 642
non-coercible data type 87
non-Latin character support 80
non-Latin characters 80
non-retained variables
SET statement 186
nonexistent data
modes for 81
reading and writing in ANSI mode 84
reading and writing in SAS mode 85
normal distributions
deviance from 475
NOTALNUM function 644
NOTALPHA function 645
NOTCNTRL function 648
NOTDIGIT function 649
NOTFIRST function 651
NOTGRAPH function 653
NOTLOWER function 655
NOTNAME function 657
NOTPRINT function 659
NOTPUNCT function 661
NOTSPACE function 663
NOTUPPER function 666
NOTXDIGIT function 668
NPV function 669
NULL function 670
Null statement 915
null values 81
  ANSI SQL null values 81, 84
  compared with missing values 83
  in character functions 342
  in numeric functions 342
  testing for 86
NUM_ITEMS attribute 1050
  hash package with 1051
numeric constants 67
numeric data type 87
numeric functions
  missing and null values in 342
  numeric values
  choice from a list of arguments 423
NWKDOM function 673

O
OCTALw. format 297
one-to-many relationships 193
one-to-one reading 196, 208
  DS2 processing during 209
  equal number of rows 209
  syntax 209
  uneven number of rows 211
one-to-one relationships 192
operators
  expression values by operator 107
  in expressions 106
  order of precedence 106
options on futures
  call prices for European, based on Black
    model 396
  put prices for European, based on Black
    model 399
options on stocks
  call prices for European, based on
    Black-Scholes model 401
OR method 1187
  order of precedence
    expressions 95
    operators 106
ORDER parameter
  hash iterator package with 1019
ORDERED method 1052
ORDINAL function 675
OUT method 1188
  outer joins
  missing and null values 83
output
  changing case of text output 1312
  formatted data 228
  global and local variables in 64
OUTPUT method 1053
OUTPUT statement 916
overloaded methods 51, 1323
OVERWRITE table option 989

P
package block 52
package global scope 138
package method expressions 100
PACKAGE statement 862, 921
packages 135
See also hash iterator package
See also hash package
FCMP package 144
HTTP package 158
  instance 135
JSON package 161
  logger package 164
  matrix 167
  predefined 135, 143
  SQLSTMT 172
  TZ package 176
  user-defined 135, 143
  variables 135
PCTL function 676
PDV 64
PERCENTNW.d format 299
PERCENTw.d format 298
  periodic cashflow stream
    convexity for 445
    modified duration for 483
    present value for 727
Perl
  compiling regular expressions 716
  PERM function 678
  PMT function 679
  Poisson distributions 475
  POISSON function 680
  POST method 159
  POWER function 681
  PPMT function 682
  predefined DS2 variables 64
  predefined packages 135, 143
  PREPARE method 1230
  PREV method 1055
    hash iterator package with 1033
  primary expressions 94
  probabilities
    beta distributions 683
    binomial distributions 684
    chi-squared distributions 686
    F distribution 693
    gamma distribution 694
hypergeometric distributions 695
negative binomial distributions 709
probabilities, computing examples 705
for multiple comparisons of means, example 705
many-one t-statistics, Dunnett's one-sided test 700
many-one t-statistics, Dunnett's two-sided test 701
studentized maximum modulus 702
Williams' test 703
Williams' test, example 706
probability 685
PROBBETA function 683
PROBBNML function 684
PROBBNRM function 685
PROBCHI function 686
PROBDF function 687
PROBF function 693
PROBGAM function 694
PROBHYPR function 695
PROBIT function 696
PROBMC function 697
PROBMED function 708
PROBNEGB function 709
PROBNORM function 710
PROBT function 711
program block statements 863
program global scope 138
program semantics 50
methods 51
variable declaration statements 50
program syntax 49
PRXCHANGE function 712
PRXMATCH function 715
PRXPAREN function 718
PRXPARSE function 719
PRXPOSN function 721
PUT function 725
outputting formatted data 228
put prices
European options on futures, Black model 399
for European options, based on Margrabe model 618
PUT statement 925
formats with 1330
PVP function 727
PW table option 991

R
radians
geodetic distance input in 520
RANBIN function 731
RANCAU function 732
RAND function 733
random number streams
in STREAMINIT function 789
random numbers
binomial distribution 731
exponential distribution 744
uniform distribution 752
RANEXP function 744
RANGAM function 745
RANGE function 746
RANK function 747
RANNOR function 748
RANPOI function 749
RANTBL function 750
RANTRI function 751
RANUNI function 752
READ table option 992
reading data
hash package 189
overview 185
SET statement 186
SQLSTMT package 189
REF method 1056
regular identifiers 77
related data 192
relational expressions
type conversion 90
REMOVE method 1057
REMOVEALL method 1060
REMOVEDUP method 1061
RENAME statement 929
RENAME table option 993
REPEAT function 753
REPLACE method 1062
REPLACEDUP method 1064
reserved words 219
retail calendar intervals 562
RETAIN statement 930
retained variables 186, 931, 945
RETURN statement 932
REVERSE function 754
RIGHT function 754
RMS function 755
ROMANw. format 302
ROUND function 756
ROUNDE function 765
ROUNDZ function 767
rows

Q
QTR function 729
QTRRw. format 301
QTRw. format 301
QUOTE function 730
merging 906
ROWS method 1190
RUN method 959
  finding minimums 1307
run-time loggers 1302

S
  sample program 17
    block scope 32
    intro to methods 42
    intro to packages 44
    thread program 34
  “Hello World!” , implicit loop 20
  “Hello World!” , multiple package instances 21
  “Hello World!” , system method 17
    “Hello World!” , threaded 23
  “Hello World!” , user-defined method 18
  “Hello World!” , user-defined package 18
  SAS macros 12
  SAVINGS function 770
  SCAN function 772
    scope 54, 1313
      global 53
      local 53
    of identifiers 52
    of variables 63
  searching
    character strings 499
  seasonal cycle 572
  seasonal indexes 560
  SEC function 774
  SECOND function 775
  SELECT expression 104
  SELECT statement 933, 1319
  SET FROM statement 941
  SET statement 868, 937
    By-group processing 202
    compared to MERGE statement 908
    concatenating tables 200
    FedSQL 188
    interleaving tables 203
    merge rows from one or more tables 188, 939
    one-to-one reading 209
    reading data 186
  thread program 186
  SETBIGINT method 1231
  SETBIGINARY method 1232
  SETCHAR method 1234
  SETCUR method 1066
  SETDATE method 1235
  SETDECIMAL method 1236
  SETDOUBLE method 1237
  SETINTEGER method 1238
  SETNCHAR method 1239
  SETNUMERIC method 1240
  SETNVARCHAR method 1241
  SETPARMS method 961
  SETPARSERINPUT method 1114
  SETREAL method 1242
  SETREQUESTBODYASBINARY method, http package 1093
  SETREQUESTBODYASSTRING method, http package 1094
  SETREQUESTCONTENTTYPE method, http package 1095
  SETSMALLINT method 1243
  SETSOCKETTIMEOUT method, http package 1096
  SETTIME method 1244
  SETTIMESTAMP method 1245
  SETTINYINT method 1246
  SETVARBINARY method 1247
  SETVARCHAR method 1248
  SHA256HEX function 776
  SHA256MACHEX function 778
  SIGN function 779
    signature 51, 96
  SIN function 780
    sine
    inverse hyperbolic 389
  SINH function 781
  SIZEKMGw.d format 304
  SIZEKw.d format 303
  SKEWNESS function 782
  SLEEP function 783
  SMALLEST function 784
  sorting
    for interleaving tables 203
  SQL
    concatenating tables 202
    SQL, in DS2 programs 1308
  SQLEXEC function 786
    comparison with the SQLSTMT package 175
  SQLSTMT package
    comparison with SQLEXEC function 175
    connection string 174
    declaring and instantiating 173
    executing the FedSQL statement 174
    overview 172
    reading data 189
    result set data 175
    specifying parameter values 174
  SQRT function 787
  SQRT method 1191
  standard character conversion 87
standard numeric conversion 87
statements 862
  block 862
  DATA statement 862
dictionary of 866
  DO statement 862
global declaration 864
local statements 865
  METHOD statement 862
  PACKAGE statement 862
program block 863
  THREAD statement 862
statistics
descriptive statistic functions 343
generating from table data 1331
STD function 787
STDERR function 788
stocks
call prices for European options on,  
  Black-Scholes model 401
call prices for European options, based  
  on Margrabe model 616
put prices for European options, based  
  on Margrabe model 618
STOP statement 945
STREAMINIT function 789
STREAMRESPONSEBODYASBINARY  
  method, http package 1096
STREAMRESPONSEBODYASSTRING  
  method, http package 1097
strings
message digest of 622
STRIP function 790
studentized maximum modulus 702
SUB method 1191
SUBSTR function 792, 1329
SUBSTRN function 794
sum
  of absolute values, for nonmissing arguments 799
SUM function 798
SUM method 1068
Sum statement 945
SUMABS function 799
SUMDUP method 1069
SUMINC method 1072
SYSTEM expression 101
system expressions
  executing functions with 342
system methods 51, 957
dictionary of 957
system options 965
dictionary of 965
T
table options 969
  examples 971
  specifying 971
  usage 970
TABLE statement 874
  tables
    analyzing, using multiple threads 1324
    combining 191
    concatenating 195, 200
    creating, based on a condition 1310
    generating statistics from table data 1331
    interleaving 195, 202
    loading into hash package 1016
    match-merging 196, 212
    one-to-one reading 196, 208
    structure and contents of 197
TAN function 800
tangent
  inverse hyperbolic 390
TANH function 801
temporary arrays 122
TERM method 963
  finding minimums 1307
testing programs 200
text output
  changing case of 1312
THIS expression 101
thread global scope 138
thread program 52
THREAD statement 862, 947
threaded processing 179
  SAS In-Database Code Accelerator 180
threads 179
  analyzing a table with multiple threads 1324
time constants 69
TIME function 802
time intervals
  cycle index 542
  seasonal cycle 572
  seasonal index 560
  validity checking 578
time zone
  TZ package 176
time zone offset
  TZ package 177
TIMEAMPMw.d format 307
TIMEPART function 803
times
  See dates and times
timestamp intervals
  cycle index 542
  seasonal index 560
timestamps
See dates and times
TIMEVALUE function 803
TIMEw.d format 305
TINV function 805
TO_DATE function 807
TO_DOUBLE function 808
TO_TIME function 811
TO_TIMESTAMP function 812
TOARRAY method 1192
TODAY function 466, 813
TODw.d format 308
TOISO8601 method 1260
TOLOCALTIME method 1261
TOTIMESTAMPZ method 1262
TOVARRAY method 1194
TRACE option 895
TRANS method 1195
transcoding
failures 190
TRANSLATE function 814
TRANSTRN function 815
TRANWRD function 818
TRIGAMMA function 820
TRIM function 821
compared to TRIMN function 823
TRIMN function 821, 822
TRUNC function 823
type conversions 87
arithmetic expressions 89
Assignment statement 1299
concatenation expressions 91
definitions 87
in functions 342
logical expressions 89
overview 88
relational expressions 90
unary expressions 88
type signature 51
TYPE table option 995
TYPEWARN option 996
TZ package
declaring and instantiating 176
local time 177
overview 176
returning time information 176
returning time zone information 176
time zone offset 177
UTC time 177

U
unary expressions
type conversion 88
uniform distributions
random numbers 752
UNIFORM function 824
UPCASE function 825
UPPER function 825
user-defined formats 228
user-defined informats 853, 854
user-defined methods 51
user-defined packages 135, 143
UUID function 826
UTC time
TZ package 177
UUID (Universally unique identifier) 826
UUIDGEN function 826

V
validation of formats 229
validation of informats 855
VAR function 827
VARARRAY statement 124, 951
variable arrays 123
variable declaration statements 50
variable declaration strict mode 58
variable lifetime 55
variable lookup 53
variables 57
declaration of 50, 58
FIRST. 869
global 53, 63
LAST. 869
local 53, 63
non-retained 186
predefined for DS2 64
retained 186, 931, 945
scope of 63
VARLIST statement 955
VAXRBw.d format 310
VERIFY function 828
VFORMAT function 829
VINARRAY function 831
VINFORMAT function 832
VLABEL function 833
VLENGTH function 835
VNAME function 836
VTYPE function 837

W
w.d format 311
WEEK function 839
WEEKDATEw. format 312
WEEKDATXw. format 314
WEEKDAY function 842
WEEKDAYw. format 315
WHICH function 842
WHICHN function 844
Williams' test 706
words
  character position in a string 499
  counting, in character strings 454
  number of a word in a string 499
WRITE table option 995
WRITEARRAYOPEN method 1114
WRITEBOOLEANFALSE method 1115
WRITEBOOLEANTRUE method 1116
WRITECLOSE method 1116
WRITEDOUBLE method 1117
WRITENULL method 1120
WRITEOBJOPEN method 1120
WRITERGETTEXT method 1121
WRITESTRING method 1121
writing data
  OUTPUT statement 189
  SQLEXEC function 189, 786
  SQLSTMT package 175, 189

Y
  YEAR function 845

YEARw. format 316
YENw.d format 317
YIELDP function 846
YRDIF function 847
YYMDDw. format 321
YYMDDxw. format 323
YYMMw. format 318
YYMMxw. format 319
YYMONw. format 325
YYQ function 850
YYQRw. format 329
YYQRxw. format 330
YYQw. format 326
YYQxw. format 327
YYQZw. format 332

Z
  Zw.d format 333
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore for additional books and resources.