Paper 3387-2015

Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS®

Troy Martin Hughes

ABSTRACT

A familiar adage in firefighting—*If you can predict it, you can prevent it*—rings true in many circles of risk management and accident prevention, including software development. If you can predict that a fire, however unlikely, someday might rage through a structure, it's prudent to install smoke detectors to facilitate its rapid discovery. Moreover, the combination of smoke detectors, fire alarms, sprinklers, fire retardant building materials, and rapid intervention may not prevent a fire from starting, but it can prevent the fire from spreading and facilitate its immediate and sometimes automatic extinguishment. Thus, as fire codes have grown to incorporate increasingly more restrictions and regulations and as fire suppression gear, tools, and tactics have continued to advance, even the harrowing business of firefighting has become more reliable, efficient, and predictable. As operational SAS® data processes mature over time, they too should evolve to detect, respond to, and overcome dynamic environmental challenges. Erroneous data, invalid user input, disparate operating systems, network failures, memory errors, and other challenges can surprise users and cripple critical infrastructure. Exception handling describes both the identification of and response to adverse, unexpected, or untimely events that can cause process or program failure, as well as anticipated events or environmental attributes that must be handled dynamically through prescribed, predetermined channels. Rapid suppression and automatic return to functioning is the hopeful end state but, when catastrophic events do occur, exception handling routines can terminate a process or program gracefully while providing meaningful execution and environmental metrics to developers both for remediation and future model refinement. This text introduces fault-tolerant Base SAS® exception handling routines that facilitate robust, reliable, and responsible software design.

INTRODUCTION

Exception handling is so ubiquitous within software applications that it often may be overlooked and, in many cases, seamless programmatic adaptation without user awareness or intervention is the objective. In other instances, a user may be alerted to the exception but functionality is not impacted. For example, when a user scrolls through the "Recent Documents" tab in Microsoft Word and selects a document to open, under normal conditions, the document opens without hiccup. The application has responded to an event—the user's selection—and opens the selected document. But under *exceptional* conditions—for example, if the document has been moved or deleted—the application cannot locate the ghost document, displays an error message, but continues otherwise undaunted. The exception—the unavailability of the document—is "caught" (i.e., detected) and subsequently "handled" (i.e., processed) and the user is alerted without significant interference or detriment. Exceptions attributed to user input should be anticipated and articulated through business rules and exception handling routines. Thus, in robust applications, a user's actions—however menacing, malicious, or aberrant—never should result in abrupt termination of the application.

Events, however, describe not only user actions and inputs but also hardware, systems, network, and environmental states and attributes. While the occurrence of adverse events inherently may be unpredictable, their existence typically should not elude or surprise developers. As the complexity and dynamism of a software's objective and operational environment increase, so too must that software's ability to respond flexibly to environmental and other factors. For example, in an organization in which SAS programs run on both Windows and UNIX machines, and in both development and production environments, the identification and handling of SAS automatic macro variables can ensure that code flexibly adapts to these respective environments without user awareness. Moreover, if warnings or errors are detected during program execution, reliable software will attempt to circumnavigate incidents to return to normal functioning. And, when process or program functioning still cannot be restored, exception handling routines can facilitate graceful termination to ensure dependent processes and data are not corrupted by preceding failures.

Data analytic development often differs from traditional software development because it relies less on user action and input and more on the ingestion, transformation, and analysis of diverse data sets. Because of this disparity, exception handling routines often are utilized in complex extract transform load (ETL) infrastructures to provide quality assurance methods for data processes and quality control mechanisms that validate data products and solutions. Exceptional data—falling outside the expected type, scope, completeness, quantity, quality, or frequency—just as easily as user or environmental factors can cause data flows to fail when quality assurance and quality control methods are not emplaced.

Because the Base SAS language does not support inherent exception handling functions, a disparity exists between SAS and many object-oriented programming (OOP) and third generation languages (3GLs) that adroitly and natively handle exceptions. Java and Python, for example, seamlessly integrate exception handling routines that dynamically alter program flow and which are more readable and intuitive than similar routines that must be hacked in SAS. Given these SAS functional limitations, concepts such as "asking forgiveness not permission"—commonplace in and advantaging OOP languages—simply do not exist in SAS. Moreover, because SAS exception handling often convolutes already complex code, SAS literature and examples often omit this critical conduit to quality because of the confusion it introduces. Notwithstanding these limitations, critical SAS infrastructures warrant exception handling routines that deliver responsible, reliable, robust solutions which are flexible and fault tolerant.

To be clear, SAS programs that are simple, straightforward, exist in static environments, and are intended for a limited distribution and duration may not require reliability or robustness. In these instances, exception handling might unnecessarily delay project completion or increase costs, while providing neither immediate nor enduring business value to the customer. However, SAS software supporting critical components, enduring projects, diverse environments, dependent processes, or a large user base should adopt commensurately high quality standards that can benefit from exception handling routines that not only can detect smoke, but often put out the fire.

TOWARD QUALITY

Software quality often is assessed in regard to the combination of functional requirements and performance requirements. Functional requirements specify capabilities that software must demonstrate and describe what software does, such as ingesting data into an ETL process to support data analytics. Performance requirements—also known as non-functional requirements—demonstrate not *what* software does but rather *how* and *how well* it does it. For example, is the ETL process reliable? Is it robust enough to recover from memory errors? Can the application be ported from a Windows to a UNIX environment? Is the software easily maintained and modified when defects are discovered or changes are necessary? Thus, software may be functionally sound, but if it cannot perform reliably or is easily thwarted by environmental and other factors, it may require timely and costly upgrades or be abandoned for higher quality software.

One of the most basic performance requirements is that software be reliable. In other words, does it function accurately without failure and, if it fails, with what frequency? Because software failure often can be caused by user, data, system, or environmental injects or attributes, robustness assesses the ability of software to navigate through dynamic or unpredictable environments. A third common performance requirement is efficiency and, especially in the era of big data, customers are eager to implement solutions that rapidly process voluminous data to deliver tactical business intelligence and data-driven decisions. Portability, a fourth requirement, depicts the degree to which software can be used on different systems or in different environments. While these four attributes represent only a fraction of all performance requirements, each can benefit from exception handling routines discussed and demonstrated in this text.

Quality software does not, however, imply rogue inclusion of all performance attributes. Rather, the inclusion or exclusion of specific performance attributes should be defined in project scope in relation to specific software objectives, the intended user base, anticipated risks, and the expected functional environment. Thus, SAS developers need to understand whether they are building a Miata or a Mercedes, and all stakeholders should maintain a unified vision of software quality and performance. And, after all, quality is not free in the zero sum game of software development. The addition of performance requirements can add complexity, increase cost, delay development, or even cause a team

to forfeit or deprioritize the delivery of certain functional requirements. Notwithstanding, only through the combination of both functional and performance requirements can quality software be delivered.

EXCEPTION HANDLING – FORGIVENESS VERSUS PERMISSION

Many OOP languages include inherent exception handling functions that facilitate code functioning, flow, and readability. An extremely abridged introduction to the Python Try-Except handling follows and demonstrates this functionality. A single line of Python code, for example, will produce an error because the variable X has not been defined.

```
print(X)
```

The generated error reads: "NameError: name 'X' is not defined." This is undesirable because it causes the program to terminate with an error. The error can be avoided through three mechanisms, including: business rules to ensure that X always is defined, asking permission to print *before* execution is attempted, or asking forgiveness *after* execution is attempted. Thus, permission equates to error prevention through tests that determine whether all requirements have been met *a priori*, while forgiveness provides post hoc notification that an error has occurred. Asking permission yields the following Python statement, which prints the value of X if the variable exists.

```
if 'X' in locals():
    print(X)
else:
    print('Variable not defined')
```

The code is straightforward in this example, but in actual software that encompasses scores of variables and complex business logic, convoluted if-then-else decision trees can obfuscate readability and make code maintenance difficult. Moreover, in asking permission, the developer often must test for specific rather than general faults. Imagine, for example, the additional overhead incurred by having successive conditional logic statements that additionally must test the type or structure of X or additional fields. Thus, a preferred method in software development often is to ask for forgiveness rather than permission, as demonstrated in the following example.

```
try:
    print(X)
except NameError:
    print('Variable not defined')
```

Asking for forgiveness through the Try-Except structure instructs Python to attempt to print X but, if that statement fails, process flow switches to the Except statement and displays a warning message rather than producing an error. If the print statement succeeds, however, the Except block is skipped and process flow continues. In actual code, rather than simply displaying a message, the Except block could contain instructions that request the user to define the value of X, transfer process control elsewhere, write to an error log, or any number of other options. And, because a single Try-Except block can contain limitless statements that are tested and executed in sequence, at the first sign of trouble, process flow is switched safely to the Except block, improving both functionality and readability of code.

In Base SAS, exception handling typically occurs by asking for permission. The equivalent request to evaluate the existence of X before printing is demonstrated below in SAS macro statements.

```
%macro test;
    %if %symexist(X) %then %put &X;
    %else %put Variable not defined;
%mend;
%test;
```

In SAS, just as in Python, asking for permission is simple in straightforward conditional logic statements. However, as business rules, anticipated risks, and the number of other attributes that must be tested increase, the complexity quickly can produce unwieldy, inscrutable code. In nested conditional logic, SAS developers have the option of utilizing either the much disparaged %GOTO statement to transfer process control or the nested %IF-%THEN-%ELSE statements. SAS unfortunately has no equivalent functionality to ask for forgiveness, but this can be approximated by testing the SAS automatic macro variable &SYSCC ("system current condition") after module or program execution. The macro variable &SYSCC

will be “0” after successful execution, “4” after a warning code, or a higher number if runtime errors were encountered. The following code represents an attempt to ask for forgiveness in SAS.

```
%put SYSCC BEFORE: &SYSCC;
%put &X;
%put SYSCC AFTER: &SYSCC;
```

This code produces the following output to the SAS log:

```
%put SYSCC BEFORE: &SYSCC;
SYSCC BEFORE: 0
%put &X;
WARNING: Apparent symbolic reference X not resolved.
&X
%put SYSCC AFTER: &SYSCC;
SYSCC AFTER: 4
```

By testing the value of &SYSCC at the end of the code or after every boundary step (i.e., RUN or QUIT statement) or macro statement, code can respond dynamically if a runtime warning or error was encountered. Because &SYSCC is a read-write variable, its value can be reset manually after encountering a warning or error with the %LET SYSCC=0 statement. This test is extremely useful to test final disposition—successful or failed—of a macro or code module, but will not indicate where the warning or error occurred. Unlike the versatility of Try-Except blocks, multiple tests of &SYSCC are required to redirect SAS process flow immediately after an error occurs because the SAS default system option NOERRORABEND causes subsequent SAS processes to execute even after an error is encountered. Although the ERRORABEND system option will cause execution to terminate immediately, the &SYSCC code in this example is "4" and represents a warning that would not be affected by activating the ERRORABEND option.

The following example highlights the repetitive post hoc testing required after every boundary step in SAS code that asks for forgiveness, demonstrating the functional limitations and redundancies when implemented in SAS. Moreover, because a macro wrapper is required to conditionally execute SAS Data steps and procedures and because individual %GOTO statements are required to redirect process flow, readability of code is quickly diminished. And in this example, unlike the suave exception handling functionality found in OOP languages, an actual error still is produced which really should be avoided in all production-grade code.

```
%macro test;
data x (keep=id var1);
    set temp1;  /* data set does not exist */
run;
%if &SYSCC>0 %then %goto err;
data y (keep=id var2);
    set temp2;  /* data set does not exist */
run;
%if &SYSCC>0 %then %goto err;
data merged;
    merge x y;
by id;
    run;
%if &SYSCC>0 %then %goto err;
%err: %put An error occurred;
%mend;

%test
```

SAS RETURN CODE IDIOSYNCRASIES

The SAS automatic macro variable &SYSERR also has utility for exception handling, as it returns the numeric warning or error code from the most recent procedure or Data step. But, because the value resets after each boundary step, &SYSERR must be implemented immediately after a RUN or QUIT

statement. Moreover, in part because macro code does not trigger boundaries, macro errors are not reported in &SYSERR. Consider the following code that highlights an initial erroneous Data step, a subsequent successful Data step, and a final erroneous macro function.

```
%put SYSERR &SYSERR;
data x;
    set y;   /* does not exist */
run;
%put SYSERR &SYSERR;
data x;
run;
%put SYSERR &SYSERR;
%gobble;   /* macro gobble does not exist */
%put SYSERR &SYSERR;
```

This code produces the following results that demonstrate &SYSERR correctly identifies the error caused by the missing Y data set. However, &SYSERR fails to identify that the macro %GOBBLE does not exist, instead printing "0"—the code for error-free execution.

```
%put SYSERR &SYSERR;
SYSERR 0
data x;
    set y;   /* does not exist */
ERROR: File WORK.Y.DATA does not exist.
run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.X may be incomplete.  When this step was stopped
there were 0 observations and 0 variables.
WARNING: Data set WORK.X was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
       real time           0.03 seconds
       cpu time            0.05 seconds

%put SYSERR &SYSERR;
SYSERR 1012
data x;
run;

NOTE: The data set WORK.X has 1 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.04 seconds

%put SYSERR &SYSERR;
SYSERR 0
%gobble;

_
180
WARNING: Apparent invocation of macro GOBBLE not resolved.
ERROR 180-322: Statement is not valid or it is used out of proper order.

%put SYSERR &SYSERR;
SYSERR 0
```

To add to the above confusion, the automatic macro variables &SYSERRORTEXT and &SYSWARNINGTEXT are introduced. SAS v9.4 literature describes &SYSERRORTEXT simply as "the text of the last error message generated in the SAS log."[i] More accurate descriptions of these variables, however, merit many caveats and their use much caution. One might assume that to print the text of an

error message, a corresponding *error* must first exist, but in SAS this is not always the case. Consider the amendment of the above code with two additional lines that follow the final line, which demonstrate the addition of &SYSERRORTEXT and &SYSWARNINGTEXT return codes.

```
%put SYSERRORTEXT &SYSERRORTEXT;
%put SYSWARNINGTEXT &SYSWARNINGTEXT;
```

The amended code now produces the following truncated output.

```
%gobble;

_
180
WARNING: Apparent invocation of macro GOBBLE not resolved.
ERROR 180-322: Statement is not valid or it is used out of proper order.

%put SYSERR &SYSERR;
SYSERR 0
%put SYSERRORTEXT &SYSERRORTEXT;
SYSERRORTEXT 180-322: Statement is not valid or it is used out of proper
order.
%put SYSWARNINGTEXT &SYSWARNINGTEXT;
SYSWARNINGTEXT Apparent invocation of macro GOBBLE not resolved.
```

The results are confounding at best, because &SYSERR fails to identify the missing %GOBBLE as an error, yet its corresponding error message (&SYSERRORTEXT) does describe that an error has occurred. Only by testing the status of &SYSCC after the above code (which is 3000, representing a "Syntax error" in SAS v9.4 documentation[ii]) can the macro error be detected programmatically.

Another possible solution to the above conundrum, and given the accuracy of the error value in &SYSERRORTEXT, would be to test the length of its value. Any length greater than 0 would seem to indicate that an error had just occurred. To test this hypothesis, the above call to the ghost macro %GOBBLE is wrapped in its own test macro so that conditional logic can be applied.

```
%macro test;
%gobble;   /* macro gobble does not exist */
%put SYSERR &SYSERR    SYSCC &SYSCC;
%put SYSERRORTEXT &SYSERRORTEXT;
%put SYSWARNINGTEXT &SYSWARNINGTEXT;
%if %length(&SYSERRORTEXT)>0 %then %put Something aint right!;
%mend;

%test;
```

The output, demonstrated below, seems to indicate that this method also might be a candidate to programmatically detect warnings and errors and reroute process flow based on the length of &SYSERRORTEXT and &SYSWARNINGTEXT. The new code does correctly identify that *Something ain't right!*

```
%test;
NOTE: Line generated by the invoked macro "TEST".
%gobble;

_
180
WARNING: Apparent invocation of macro GOBBLE not resolved.
ERROR 180-322: Statement is not valid or it is used out of proper order.

SYSERR 0    SYSCC 3000
SYSERRORTEXT 180-322: Statement is not valid or it is used out of proper
order.
SYSWARNINGTEXT Apparent invocation of macro GOBBLE not resolved.
Something aint right!
```

The downfall of this method, however, is that unlike most automatic macro variables that reset after program termination (successful or unsuccessful), the values of &SYSERRORTEXT and &SYSWARNINGTEXT persist across these boundaries. This is patently clear when the above code is fixed by adding the macro %GOBBLE and the macro %TEST is subsequently rerun.

```
%macro gobble;
%put Gobble Gobble Gobble!;
%mend;

%test;
```

The inclusion of the macro %GOBBLE now causes the code to execute without error, as demonstrated by the 0 values of &SYSERR and &SYSCC. However, the values of &SYSERRORTEXT and &SYSWARNINGTEXT both have persisted from the first erroneous execution. Thus, testing the length of these values programmatically causes the code erroneously to assess that both a warning and an error have occurred, as demonstrated in the following output.

```
%test;
Gobble Gobble Gobble!
SYSERR 0    SYSCC 0
SYSERRORTEXT 180-322: Statement is not valid or it is used out of proper
order.
SYSWARNINGTEXT Apparent invocation of macro GOBBLE not resolved.
Something aint right!
```

And, because both &SYSERRORTEXT and &SYSWARNINGTEXT are read-only automatic macro variables, once they have committed non-null values, they can only be to reset to null values by terminating the entire SAS session! Due to this unresolvable idiosyncrasy, these automatic values obviously cannot be used in exception handling in production-grade code because of the false results they produce.

The automatic macro variable &SQLRC represents the process return code for the SQL procedure, similar to the &SYSERR macro variable in Data steps and SAS procedures. Because the SQL procedure can contain numerous statements before the QUIT statement is encountered, a successful statement that completes without errors will change the value of %SQLRC to 0 (i.e., no error) even if the prior statement contained runtime errors within the same procedure. Due to this functionality, an evaluation of &SQLRC would need to occur after each statement within the SQL procedure if multiple statements exist. A more efficient yet less specific solution is to use the automatic macro variable &SQLEXITCODE that contains the highest error value encountered during the entire SQL procedure.

While the above idiosyncrasies merely have described *limitations* in Base SAS software that hinder the implementation of exception handling routines, from time to time actual *errors* also exist in Base SAS that cause even robust exception handling routines to return completely spurious results that indicate process success when in fact a process has failed. In a separate text, the author demonstrates flaws in the GINSIDE procedure that cause it to produce completely erroneous data while yielding no log messages or return codes indicating these failures.[iii] Other authors too demonstrate examples where "the value of the *syserr* macro variables may be unreliable."[iv] Thus, it is critical for SAS developers to understand and anticipate expected output and results. Moreover, in production-grade systems that demand reliability and cannot tolerate dysfunctional Base SAS error-checking algorithms, a defense in depth approach is warranted in which other metrics are utilized to determine if processes *truly* completed without error.

Regardless of which method is attempted to ask for forgiveness in SAS, functional limitations are encountered. Whereas Python catches an error and immediately redirects process flow to an Except block, SAS produces an actual error. This triggers the error to be recorded in the SAS log and, when executed inside Enterprise Guide, an undesirable red X appears on the code node icon. Moreover, if the ERRORABEND system option is selected, the program will terminate even before the code has had a chance to detect and handle errors. For all these reasons, while forgiveness remains the preferred method in most languages to accomplish exception handling, in Base SAS, permission also often will need to be asked to facilitate reliability. And, in the most robust systems, post hoc testing of &SYSCC or &SQLEXITCODE automatic macro variables should be implemented to further validate process success.

EXCEPTION HANDLING – IS BIGGER ALWAYS BETTER?

Exception handling should be utilized commensurate to the intended degree of quality as well as function, intent, and scope of software, but is bigger handling always better? The primary goal of exception handling is to improve the responsiveness of software by making it more flexible, adaptable, resilient, reliable, and robust. Examples of exception handling utilizing SAS, however, often do little to alter program flow or add functionality or capabilities, but rather provide user feedback on runtime status via the log. In many cases, while this additional information may assist the developers during development to build more fault-tolerant code, it doesn't actually add quality to the software because the user gains no benefit from these unseen messages. Especially in end-user development environments in which developers represent the sole users of their programs, gratuitous exception handling unfortunately flourishes. Consider the following example, which highlights the error received when a data set that does not exist (test.states) is referenced.

```
data x;
    set test.states;
ERROR: File TEST.STATES.DATA does not exist.
run;
```

A simple macro wrapper can test for library and data set existence before attempted access in the Data step. While both tests in this example could be completed in a single statement, they are separated to represent the kind of nested logic common in actual data processes.

```
%macro testdata;
    %if %sysfunc(libref(test))=0 %then %do;
          %if %sysfunc(exist(test.states)) %then %do;
                data x;
                      set test.states;
                run;
                %end;
          %else %put Data Set does not exist;
          %end;
    %else %put Library does not exist;
%mend;
```

While the addition of this macro code does prevent the above error, additional functionality is not gained by the user because the end result remains consistent: the data set X was not generated. Therefore, in the above example, a simple post hoc check of the macro variable &SYSCC would have been sufficient and would have eliminated the gratuitous *a priori* testing. Or, better yet, simply let the native Base SAS error suffice and add no exception handling routines whatsoever. The revised example below demonstrates a more dynamic attempt to test if the data set is valid.

```
%macro testdata(lib=, dsn=);
    %global err;
    %let err=library or data set does not exist;
    %if (%sysfunc(libref(&lib))=0 and %sysfunc(exist(&lib..&dsn))^=0) %then
%let err=;
%mend;

%macro wrapper;
%testdata (lib=test, dsn=states);
%if %length(&err)=0 %then %do;
    data x;
          set test.states;
    run;
    %end;
%mend;

%wrapper;
```

Notwithstanding this increased flexibility, the code still provides no additional value to the user because the data set X is not created. Thus, as omission of test.states represents a catastrophic exception that must be handled and, at this point, exception handling only can signal to terminate either the process or the program. The following code modifies the macro %WRAPPER and, by demonstrating the process flow more contextually, finally highlights the first true performance improvements—robustness and resilience. The code now skips data sets that do not exist, thus allowing subsequent data sets in the &FILELIST macro variable to be processed in a fault-tolerant fashion. For example, if the data sets test.states and test.counties exist, but test.cities does not exist, the following code will execute creating output data sets xstates and xcounties correctly, but will skip over creating xcities because test.cities does not exist.

```
%macro wrapper;
%let filelist=states cities counties;
%let i=1;
%do %while(%length(%scan(&filelist,&i))>1);
    %let fil=%scan(&filelist,&i);
    %testdata (lib=test, dsn=&fil);
    %if %length(&err)=0 %then %do;
          data x&fil;
                set test.&fil;
          run;
          %end;
    %let i=%eval(&i+1);
    %end;
%mend;

%wrapper;
```

SAS developers should be cautioned that while exception handling can be utilized to relay warning and error messages to the SAS log, the true advantage of exception handling is its ability to dynamically alter program flow during execution. Thus, before embarking on an exception handling crusade, developers should determine for each project the specific added functionality or performance that proposed exception handling routines will provide. Those benefits, once enumerated, should be compared against the anticipated added cost, time, or complexity required for exception handling implementation to determine if their inclusion is warranted. Thus, the increased use of exception handling can—but does not always—denote increased business value or higher quality software.

EXCEPTION HANDLING TO MONITOR ENVIRONMENTAL STATE

Portability describes software's ability to function across diverse environments, such as on both Windows and UNIX operating systems. Although the majority of Base SAS language is portable between these operating environments, some functional or language components differ. For example, the SLEEP function operates in Windows, but must be replaced with the CALL SLEEP function in UNIX environments. The following macro code excerpt demonstrates the ability to interpret the automatic macro variable &SYSSCP that denotes the type of operating system, and to respond by conditionally executing the appropriate SLEEP function.

```
%macro test;
%if &SYSSCP=WIN %then %let sleeping=%sysfunc(sleep(10));
%else %if %sysfunc(substr(&SYSSCP,1,3))=LIN %then %do;
    data _null_;
          call sleep(10,1);
    run;
    %end;
%mend;

%test;
```

Another environmental difference that can exist is the distinction between development, testing, and production environments. SAS practitioners may design and develop code on one system, test code on

another, and finally deploy the validated code to a third system for production. Although the operating systems may be functionally identical, differences may exist between disparate systems such as directory structures, file names, or the type and nature of runtime log and error reporting. The &SYSSITE automatic macro variable reflects the SAS site number for each SAS license and, by testing this number, code conditionally can execute in development, testing, and production environments. By implementing flexible conditional logic, developers can maintain single-version source code applicable to all environments, thus obviating the error-prone method of maintaining disparate development and production code bases. The following example demonstrates a conditionally defined SAS library, thus facilitating one version of code that can be maintained and executed across diverse development and production environments.

```
%macro test;
%if &SYSSITE=1234512345 %then %do; /* development environment */
    libname final '/folders/dev/';
    %end;
%else %if &SYSSITE=5555599999 %then %do; /* production environment */
    libname final '/folders/prod/';
    %end;
%mend;

%test;
```

Some environmental errors—such as memory errors—can be detected through quality assurance routines and resolved through a variety of mechanisms. A SORT procedure that causes an "out of memory" error might be resolved by stopping other SAS sessions that are hogging resources and subsequently restarting the SORT procedure. While appropriate by laissez faire development standards, this manual approach would be neither pragmatic nor possible in a production environment. A responsible solution should automatically terminate processes dependent on the failed SORT procedure. A more creative solution might, after detecting the memory error on the SORT procedure, immediately initiate a user-created macro %SAFESORT that iteratively performs sorts of subsets and later joins the results, thus providing the same sorted results but through an incremental and less memory intensive approach. Such a macro might take significantly longer to execute, given its additional complexity, but nevertheless would run without memory errors unlike the out-of-the-box SORT procedure. And, despite its longer runtime, the automated detection and immediate response would create a much more efficient, end-to-end process flow.

Other environmental states or errors, however, require an even broader perspective. One comprehensive solution implements the SYSTASK statement to execute exception handling within a high quality, production-grade environment[v]. This facilitates the implementation of post hoc testing for process success based on return codes that are generated from separate batch jobs. Even environmental errors such as a process timeout can be captured by enclosing processes within the SYSTASK wrapper for execution. Notwithstanding, some SAS or system errors will defy even identification utilizing SAS. For example, when the SAS server becomes comatose and crashes, no amount of exception handling routines can identify or resolve this issue because the scripts are written on a system that has stopped functioning. In these extreme but hopefully rare circumstances, two possible solutions exist for catastrophic exception detection. Some systems implement a ping that executes a SAS script to test system health at regular intervals, the results of which can be ported to a BI interface, dashboard, or other dynamic report. In this type of system, absence of ping results will demonstrate that the system has stopped functioning. A second automated method employs scripts external to the SAS environment that regularly interrogate health of the SAS server or its data sets. Each method can provide automated assurance that server failure will be detected as an exception through routinized methods rather than through ad hoc discovery as analysts email or bellow to their administrator "Is the server down again?!"

EXCEPTION HANDLING TO MONITOR FILE STATE

Testing to determine library and data set existence already has been demonstrated and, as discussed, should be implemented only when performance is improved, for example, by creating fault-tolerant process flows. Thus, in those instances in which data set absence signals certain defeat, native SAS

error reporting may be a better solution than complex exception handling routines. Data availability, however, implies not only that a data set exists but also that it can be accessed. If a data set is being created by one process, other processes or users attempting to access that data set will fail because an exclusive lock is held by the first process. A single locked data set can cripple a complex data infrastructure but, by installing processes that test for data set availability before attempted use, process continuity can be maintained. The macro %LOCKITDOWN[vi] was created by the author and tests data set availability by identifying file locks and, after encountering a lock, repeatedly testing until access can be gained or the process times out. The %LOCKITDOWN macro is not described in detail in this text but its use in modifying process flows through exception handling is demonstrated below in which the lock status is tested every 5 seconds until access is gained or until the process times out after 300 seconds.

```
%include '/folders/myfolders/lockitdown.sas';
%macro test;
    %LOCKITDOWN(lockfile=test.states, sec=5, max=300);
    %if %length(&lockerr)=0 %then %do;  /* if data set is available */
          data x;
                set test.states;
          run;
          %end;
    %else %put ERROR;
%mend;

%test;
```

SAS literature is full of examples that depict the benefits of combining related input/output (I/O) tests into modular, reusable macros that simultaneously can test the existence, availability, or appropriateness (i.e., naming conventions) of SAS libraries, data sets, or external files. One example, the Validator[vii], demonstrates exception handling routines that support a number of dynamic situations, including validating SAS data sets. By combining similar functionality, macro reuse is maximized because the macro is generalizable to diverse purposes and projects. As long as each macro contains a return code that demonstrates its success, failure, and possibly other process metrics, these modular pieces easily can be linked together for process validation efforts. An automated dashboard developed by the author dynamically and comprehensively monitors both file states and data structures of all permanent data sets within a SAS server. By iteratively pinging data sets in a continuous manner, this smoke detector signals to users and developers if a data set is locked for an unexpected amount of time or if structural components of the data set have been unexpectedly modified.[viii] Integrated systems such as this that maintain a pulse on the SAS environment can provide invaluable feedback, including immediate recognition of process or data product failure.

EXCEPTION HANDLING TO MONITOR DATA STRUCTURE

The preceding examples of exception handling have demonstrated quality assurance routines that flexibly respond to errors, environmental attributes, and other injects to facilitate robust program execution. Quality control measures conversely monitor and validate products to ensure that they meet stated requirements, including functional objectives and quality standards. In data analytic development environments, common products include data products such as data sets or analytic reports that are generated by ETL processes or derivative analysis. Validation of a data product can demonstrate to an analyst that a data set is accurate and ready to be incorporated into analyses. It also can signal to dependent processes that the data set is ready to be parsed, transformed, or integrated into a data store or infrastructure. Because data often are ingested from unreliable third-party sources, data validation methods can be executed not only on data sets that are created by SAS process flows, but also on those that are ingested. Thus, in production-grade SAS environments, data quality validation can occur at any point throughout the process flow, on ingested, intermediate, and output data sets.

Data structure validation describes the validation of data set metadata, including everything that can be known about an empty data set. This includes the name, number, type, and format of fields, as well as whether indices, sorting algorithms, data constraints, descriptions, labels, or other attributes are present. Transactional data sources are a common candidate for data structure validation because transactional

updates typically are received with standardized frequency and format. Thus, if a data set historically received with eight fields suddenly has ten, some alert to stakeholders is warranted to investigate and potentially expunge the disparate transactional update.

One method to identify or test metadata is through the CONTENTS procedure, which produces a 41-attribute description of each field in a data set. The following example exports this metadata into a temporary data set (Temp) for manual viewing or validation through automated means.

```
proc contents data=test.states details out=temp;
run;
```

A second method is to utilize the SQL procedure to extract metadata from the dictionary.columns dictionary table, which produces an 18-attribute data set that describes each field in one or multiple data sets. The advantage of the SQL procedure is it can be optimized for faster performance with the WHERE clause.

```
proc sql;
    create table first as select * from dictionary.columns
    where libname="TEST" and memname="STATES";
quit;
run;
```

A third method to access metadata is through SAS I/O functions, such as VARNAME or VARTYPE. The primary advantage of this method is that access can be achieved entirely through macro coding with use of the %SYSFUNC macro function. For example, the following code prints the field name and type (e.g, character or numeric) for the first field in the data set test.states, all of which could be imbedded within a Data step if necessary.

```
%macro test;
    %let dsid=%sysfunc(open(test.states));
    %let varname=%sysfunc(varname(&dsid,1));
    %let vartype=%sysfunc(vartype(&dsid,1));
    %put &varname &vartype;
%mend;

%test;
```

Regardless of which method is utilized to extract metadata, the common objective is to compare all attributes of a newly ingested or newly created data set against its baseline—i.e., the metadata expected and known to be correct. One way to accomplish this is to maintain a library (Baseline) of empty data sets, for example, containing the data set baseline.states having zero observations but all other file attributes. To validate future data sets that are either created or received, a developer only would need to compare each new data set against its respective baseline.

The following exception handling routine extracts metadata from baseline.states, extracts metadata from a newly created data set test.states, and compares these using the COMPARE procedure. Note that the fields representing library name, member name, number of observations, create date, and modify date must be removed because these inherently will differ between baseline and comparison data sets. Because the OUTNOEQUAL option only outputs observations when discrepancies exist, if the data set structures do not match, the temporary data set Validate will have one or more observations. Thus, by testing the number of observations in Validate, users can determine if the two comparison data set structures are identical. Utilizing this logic, the macro %TEST attempts to validate the data set test.states and, if successful, prints a notification. In an actual production environment, this log entry would be replaced with further validation methods, business rules, or process flows that could proceed with freshly validated confidence that the data set structure is correct.

```
libname baseline '/folders/myfolders/baseline/';
libname test '/folders/myfolders/test/';

data baseline.states;
    length state $40 capital $40;
run;
```

```
data test.states;
    length state $40 capital $30; /* incorrect format for capital */
run;

%macro validate (dsn1=, dsn2=);
    %global err;
    %let err=;
    proc contents data=&dsn1 details out=temp1 (drop=libname memname nobs
           crdate modate) noprint;
    run;
    proc contents data=&dsn2 details out=temp2 (drop=libname memname nobs
           crdate modate) noprint;
    run;
    proc compare data=temp1 compare=temp2 noprint out=validate outnoequal;
    run;
    proc sql noprint;
          select count(*) into :nobs
          from work.validate;
    quit;
    run;
    %if &nobs>0 %then %let err=does not validate;
%mend;

%macro test;
    %validate (dsn1=baseline.states, dsn2=test.states);
    %if &SYSERR>0 or &SYSCC>0 %then %let err=process failure;
    %if %length(&err)=0 %then %put Validation Complete;
    %else %put &err;
%mend;

%test;
```

EXCEPTION HANDLING TO MONITOR DATA QUALITY

Exception handling and other methods that suffuse quality into software culminate with increased data quality. Because data, data products, data solutions, and data-driven decisions lie at the heart of data analytic development, data quality is discussed thoroughly throughout SAS literature. Whereas environmental, system, software, user, and other errors or attributes that must be handled in robust software can be done so through largely standardized methods that are generalizable across projects and even organizations, data quality must be enforced through sometimes complex or enigmatic business rules that are largely endemic to a field, organization, project, or data set. Thus, the additional complexity and need to understand both the industry-specific business rules as well as their technical implementation can make data quality exception handling an exceptionally taxing experience.

Not only are the rules tougher to enforce, but the crime may be much more difficult to recognize. In process validation or data structure validation, often a runtime error is produced when exception handling attempts fail. Once the error is discovered, the exception handling routines typically can be modified (i.e., improved) and the process can be run with the assurance that at least *that* error will not occur again. With data quality validation, however, logic errors rather than runtime errors often are produced, thus test data represent one method to validate known injects through a process that ensures output conforms to expectations. Data constraints, control charts, and statistical tests also can help validate output against expected norms, and can elucidate faulty business rules and logic that produce invalid results.

Because SAS literature is so replete with methodologies to clean and validate data, only one example is provided here. It highlights, however, the common requirement to validate values of a categorical variable against a known set of discrete values. Where data constraints have not been enforced during data entry or collection, a common first task is the standardization of values that bins acronyms, abbreviations, and other spelling variations into cleaned, congruous fields. Thus, when values are encountered that lie

outside the set of expected categorical values, exception handling routines can alert stakeholders and either delete or modify a value, observation, or data set to preserve the integrity of the system.

In the following example, the data set chem_data includes seizures of homemade explosive (HME) precursor material found in Afghanistan and recorded in an unstructured character field. Because of the lack of data constraints during data input, widely disparate spelling variations exist in the Chem field, some of which have been accommodated by the standardization model depicted in the conditional logic. But, because the value “aluminumnumnum” is not included in the binning algorithms, a quality control report should identify this outlier rather than ingesting it into the data set. Later, an analyst might decide that this value should in fact be binned as "aluminum" but, for now, it remains excluded from the data until the chemical data model has been updated.

```
data chem_data;
    infile datalines delimiter=',';
    length RecNo $20 Province $ 40 Chem $50;
    input RecNo $ Province $ Chem $;
    datalines;
1,Zabul,ammonium nitrate
2,Kandahar,ammonium nitrate
3,Helmand,calcium ammonium nitrate
4,Ghazni,calcium ammonium nitrate
4,Ghazni,ammmonium nitrate
4,Ghazni,potassium chlorate
5,Zabul,ammonium nitrate crystals
6,Kandahar,aluminumnumnum
;
run;
data chem_data_cleaned;
    set chem_data;
    length Chem_cleaned $50;
    if Chem in ('ammonium nitrate','ammmonium nitrate','ammonium nitrate
          crystals')
    then Chem_cleaned='ammonium nitrate';
    else if Chem in ('calcium ammonium nitrate')
          then Chem_cleaned='calcium ammonium nitrate';
    else if Chem in ('potassium chlorate')
          then Chem_cleaned='potassium chlorate';
    else Chem_cleaned='UNKNOWN!!!';
run;
```

In this example, the exceptional event that is being handled is the incidence of the new value “aluminumnumnum” that does not appear in the current data model and which must be evaluated. A separate text by the author demonstrates a dynamic, macro-based implementation of categorical variable binning and standardization, followed by quality control validation and exception reporting for data that are not found within a data model.[ix]

Data products lend themselves to automated validation much more readily when they constitute databases or data sets that follow prescribed, evaluable structures. Analyses, which can assume the form of reports, graphs, charts, and other output, can be less than straightforward to validate when a manual review is required to ensure that values accurately portray their underlying data sets. Nevertheless, several SAS procedures—by way of the OUT statement—encourage data validation by writing analytic or other results to data sets. For example, the REPORT procedure can generate output data sets that include all fields, including those newly created in COMPUTE blocks. Because one of the advantages of the REPORT procedure is its ability to perform mathematical calculations to create both new fields and summary statistics across observations, business rules can be integrated directly into the procedure. While highly efficient, this integration introduces a liability if these new statistics are not validated through quality control mechanisms that test their output.

The following code builds a report that reflects the author's order from Pizzeria Uno—famed progenitor of Chicago deep dish pizza. The data set includes observations for one large pizza and one order of bruschetta, as well as their respective prices and taxes. The total cost of each item is computed in the REPORT procedure for display and output to the Pizzaout data set.

```
data pizza;
    infile datalines delimiter=',';
    length item $50 price 8 tax 8;
    input item $ price tax;
    datalines;
large pepperoni and sausage pizza,28.51,3.07
bruschetta,9.99,1.07
;

proc report data=pizza out=pizzaout;
    columns item price tax itemtot;
    define itemtot / computed;
    compute itemtot;
          itemtot=price.sum+tax.sum;
    endcomp;
run;
```

In this example, the final data product is the report itself which is demonstrated in Table 1, and which includes the computed field Itemtot.

Item	price	tax	Itemtot
large pepperoni and sausage pizza	28.51	3.07	31.58
Bruschetta	9.99	1.07	11.06

Table 1. Pizzeria Uno Order

Because the customer has requested only a report, no output data set is required; however, the output data set Pizzaout optionally is created because it can be used to validate the REPORT procedure and its output. Data validation techniques presented above—which can validate either the data structure or data values, or both—can be utilized as quality control mechanisms in this and other procedures that can generate output data sets. It is much more reliable to validate SAS reporting through automated methods that validate underlying data sets against business rules rather than relying on analysts to manually review ODS output.

THE DREADED LOG – WHAT IS IT GOOD FOR?

The SAS log, while critical to the development environment, is the least useful process metric in a production-grade environment. During software development, the log is useful for design feedback, debugging, optimization, and validation of business rules and other logic. However, once software is operationally deployed to a user base, these benefits cease and reliance on the SAS log in a production environment only will act to reinforce poor coding practices. Consider that when a user opens Google Chrome and navigates to a Gmail account, no log is created or, if it is, the log is internal and unavailable to the end user. SAS software developers should code with the same fierce confidence and technical prowess when developing high-end data analytic programs. In a production environment, exception handling should be utilized both to prevent and detect errors, rather than relying upon too-little-too-late post hoc log analysis. There is no magical warning or error that appears solely in the SAS log that cannot be detected during program execution through competent exception handling.

In an ad hoc environment in which end-user developers are tactically coding short-term solutions that don't require significant quality or performance attributes, the log is the perfect place to validate process completion. But, as software becomes more dynamic, distributed, depended upon, and enduring, reliance

on the log for process validation should be avoided. Thus, in a SAS production environment, the log is good for one and only one thing: continuous quality improvement (CQI.) No one gets it right the first try. Just when a developer believes he's conceptualized everything possible that could cause code failure, something will fail and a new fault or defect will have been discovered. In these instances, hopefully an error is generated to the log, the log has been saved, the log file is parsed quickly through automated processes, and the developer is able to utilize this information to recreate the conditions under which the failure occurred. In most situations, similar errors can be prevented or detected through additional exception handling routines, thus making the code more reliable, robust, and enduring than before the fault was uncovered.

A best practice is to save log results for all production processes. Numerous SAS white papers detail extensive methods that automatically parse SAS logs, immediately identifying salient information such as unexpected warnings or errors. The vast majority of production logs should contain no unexpected warnings or errors and these clean logs can be deleted immediately by the log parser. But, on those rare occasions in which a production process fails unexpectedly, the existence of a log that has captured the event can ensure that developers engineer future processes that proactively handle and thus prevent recurrences of that specific failure.

CONCLUSION

Exception handling represents a common quality assurance method used in software development to facilitate the smooth execution of software despite environmental, system, user, and other adverse or unexpected events. Moreover, exception handling provides developers the means to build higher quality software that espouses performance requirements such as reliability, robustness, efficiency, portability, and modularity. Although Base SAS unfortunately has no inherent exception handling functions, much of this functionality nevertheless can be implemented through savvy design and creative solutions that maximize the use of SAS macro language, SAS automatic macro variables, and I/O functions. While not all SAS code necessitates performance enhancements like exception handling, all SAS developers should understand its benefits and functionality, and insist on its use in complex software projects that support critical infrastructure in diverse environments for distributed user bases.

REFERENCES

[i] SAS® 9.4 Macro Language: Reference, Third Edition. SYSERRORTEXT Automatic Macro Variable. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/68140/HTML/default/viewer.htm#n0f6lmit5jr1xen1dl1owpkufvfj.htm

[ii] SAS® 9.4 Macro Language: Reference, Third Edition. SYSERR Automatic Macro Variable. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/68140/HTML/default/viewer.htm#n1wrevo4roqsnxn1fbd9yezxvv9k.htm

[iii] Hughes, Troy Martin. 2013. Winning the War on Terror with Waffles: Maximizing GINSIDE Efficiency for Blue Force Tracking Big Data. Southeast SAS Users Group (SESUG).

[iv] Gregory, Martin and Serono, Merck. 2009. Techniques for writing robust SAS macros. Pharmaceutical Users Software Exchange.

[v] Cogswell, Denis. 2005. More than Batch – A Production SAS® Framework. SUGI 30.

[vi] Hughes, Troy Martin. 2014. From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. Western Users of SAS Software (WUSS).

[vii] Wilson, Steven A. 2011. The Validator: A Macro to Validate Parameters. SAS Global Forum.

[viii] Hughes, Troy Martin. 2014. Will You Smell Smoke When Your Data Are on Fire? The SAS Smoke Detector: Installing a Scalable Quality Control Dashboard for Transactional and Persistent Data. Midwest SAS Users Group (MWSUG).

[ix] Hughes, Troy Martin. 2013. Binning Bombs When You're Not a Bomb Maker: A Code-Free Methodology to Standardize, Categorize, and Denormalize Categorical Data Through Taxonomical Control Tables. Southeast SAS Users Group (SESUG).

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com