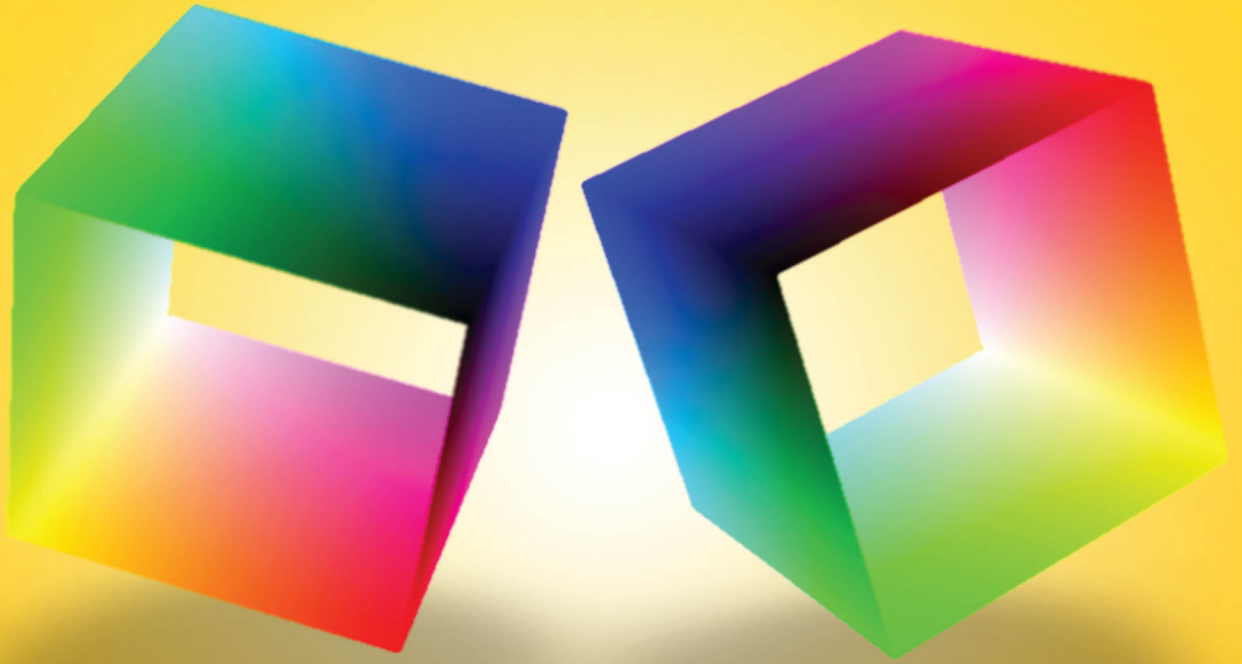
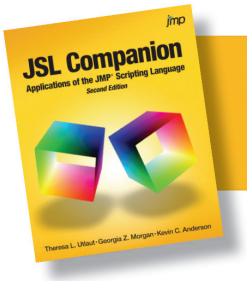
JSL Companion

Applications of the JMP® Scripting Language

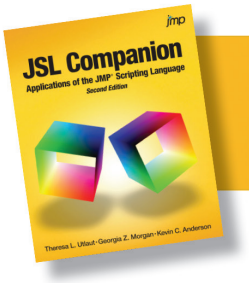
Second Edition



Theresa L. Utlaut · Georgia Z. Morgan · Kevin C. Anderson



Contents



1 Getting Started with JSL

Introduction

We don't want anyone to get hurt, so the first chapter warms up the reader with gentle stretching using the JMP Scripting Language (JSL). This chapter demonstrates a portion of the utility of scripting in JMP, using explanations and examples that detail the basics of the language. Then, we introduce more useful and advanced concepts. After a short demonstration showing the vast possibilities of JSL, we cover a few basic concepts, describing some of the windows, effective and efficient script writing from JMP, and preliminary scripting concepts, including punctuation, referencing objects, messages, naming, and lists. This chapter builds a foundation that supports your journey into JSL scripting.

The Power of JMP and JSL

Opportunities to transform data into information come at us every day like a fire hose aimed at a shot glass. Our experience is in industrial statistics, supporting the development and manufacturing fabrication facilities in the technology manufacturing group of a large semiconductor company. We consult with engineers to maximize their returns on investments of time and effort. We teach classes on statistics and experimental design. We try to do something wonderful by finding innovative ways to get valid, actionable information in front of management to better enable its decisions. And, for all of this and more, one of our most useful tools is JMP.

JMP is a powerful software application that was created by SAS almost 30 years ago "because graphical representations of data reveal context and insight impossible to see in tables of numbers." Its point-and-click interface, capabilities, and style enable analysts without much formal training to make defensible, data-supported recommendations in a short period of time with less effort. JMP is as advertised: visual, interactive, comprehensive, and extensible.

That extensibility comes from JSL. JSL is an interpreted language that can implement the data manipulation and analyses available in JMP in a flexible, concise, consistent, standardized, and schedulable way. It can perform routine and redundant tasks that are typically done using

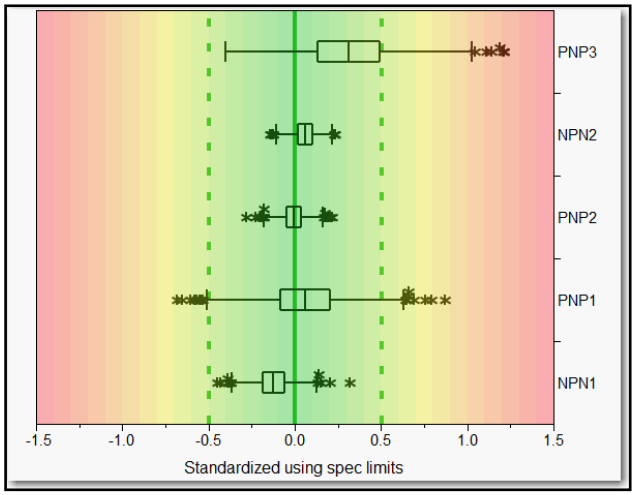
point-and-click, as well as extend current JMP capabilities. Indeed, a talented and motivated scripter can write new analyses, new procedures, or new visualizations that implement methods not available in the point-and-click interface of JMP. The scripter can deploy these methods across an entire enterprise. Through JSL, almost any data manipulation, analysis, or graphic can now be generated, provided enough knowledge, innovation, and perseverance are applied. We are often amazed at the scripts written by our coworkers that demonstrate not only the generation of information from data elegantly, but do so in a manner or sequence that we would not have considered ourselves. Of course, there are some holes in the innate capabilities of any software application, but we believe that the capability of a script is usually only limited by the skill, perseverance, and imagination of the scripter.

If you have some experience with JMP and JSL, you probably already feel this way. Or, you suspect that it's true at the least. We can hear the uninitiated saying, "Wow, the hyperbole meter has hit the peg!" Fair enough. We know the doubters need proof. Hang tight; we provide demonstrations within our JSL applications throughout the rest of this book. But, for right now, let's look at a few samples.

First and foremost, JMP is visual. You might have already peeked at the sample script named Teapot.jsl in the Scene3D folder. This script is an impressive display of visual power, even if only for artistic appreciation.

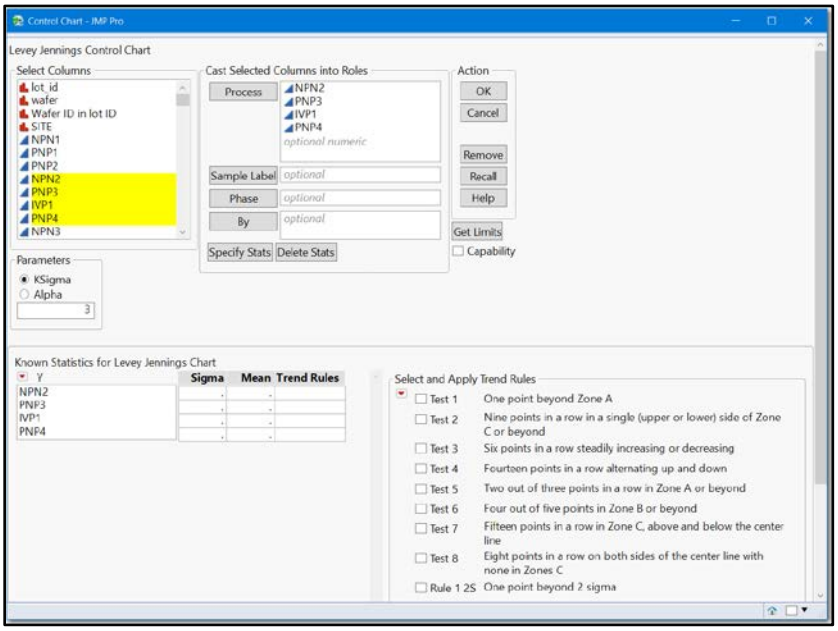
Let's say your manager wants a presentation-ready process-capability report in his inbox every Monday morning. You can take comfort in knowing that this report and the accompanying tabular reports are possible to generate, publish on a website, and mail on a scheduled basis using JSL.

Figure 1.1 Capability Report



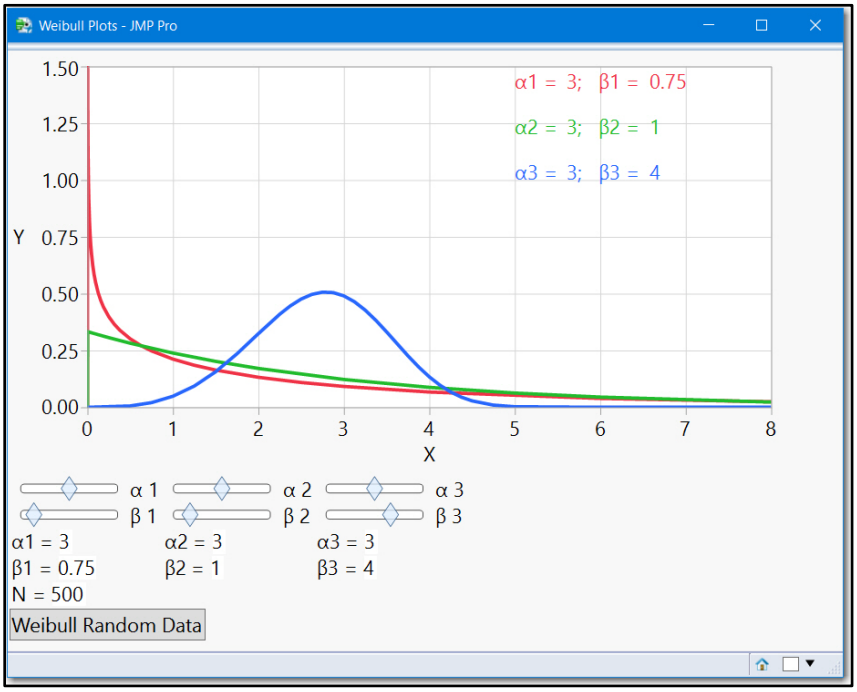
JMP is interactive. Using JSL, dialog boxes can gather salient information from users for deployment in analyses.

Figure 1.2 Custom Dialog Built with JSL



With some JSL, users can interact with graphics through text entry or sliders.

Figure 1.3 Visualizing the Weibull Distribution



JMP is comprehensive. JSL lets you control most of the innate capabilities of JMP. Even where there are holes in the capabilities of JMP, a wily scripter can use SAS, R, Python, MATLAB; run external programs like PERL with JSL data, functions, matrix-manipulation abilities; and use extensive graphic control to generate and manipulate data tables, perform innovative procedures and analyses, and return the results for display and reporting. Again, your script is limited not by the capabilities of JMP, but only by your skill and imagination.

The Basics

Have you ever had an instructor who started the class with a comment similar to, "You'll have no problem learning this. It's really quite easy"? Isn't that an annoying comment from someone who is an expert? Of course, it seems easy if you already know it. Learning something new can be intimidating and hard. Fortunately, many tasks in JSL are relatively easy. There is no sense in being disingenuous, saying that mastering JSL is simple. It's not. In fact, expert JSL programmers learn how to do something new or optimize a script on a regular basis. With the helpful scripting tools in JMP and a few instructions, useful JSL scripts can be written in a short time. We regularly see students write useful scripts that improve productivity after taking just a four-hour introductory course. We predict that, as you write more scripts, you will discover that you have developed a feel for JSL. You might start surprising yourself by writing scripts with commands that you have never used simply because you have an understanding of the structure of the language.

Create and Run a Script

Now that you have warmed up with some stretching, let's do a little exercise. You are going to create a script. It is a simple script, but it will give you a sense of the structure of JSL, and your confidence will build about learning a new language.

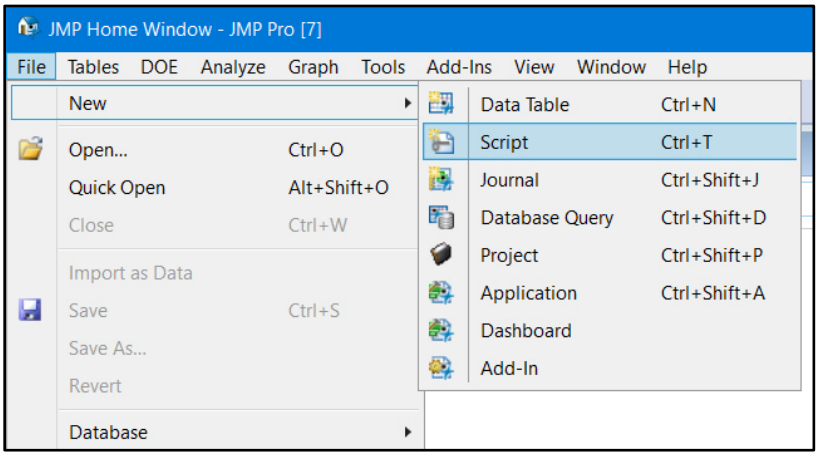
In JMP, open a new script window. The script window is discussed in more detail in the next section of this chapter.

There are several ways to open a new script window in JMP.

- From the menu bar**, select File ► New ► Script**. (See Figure 1.4.)
- From the **Home** toolbar, click the Script icon. 
- From the **JMP Starter** window, select **New** in the Script section.
- Hold down the CTRL key, and select T.

As noted in the introduction, this book is specific to Windows, so throughout the book Macintosh users will need to translate our instructions into instructions for the Macintosh. For example, the CTRL + key sequence in Windows will be the Command + key sequence on the Macintosh.

Figure 1.4 New Script Window Using the Menu Bar



For your first script, type the following code into the script window. Note: All scripts in this section are included in the 1_TheBasics.jsl script.

```
txt = "In teaching others we teach ourselves.";
Show( txt );
```

Now, run your script. There are several ways to run a script in JMP.

- Click the Run Script icon on the **Script Editor** toolbar.
- Select **Edit ► Run Script.**
- Right-click on the script, and select **Run Script.**
- Hold down the CTRL key, and select R.

You can run portions of a script by highlighting the lines of code to run, and then using one of the previous ways to run just the highlighted code.

After the script is run, it prints the variable name and text in the Log window. If the Log window is not open, select **View ► Log.**

```
txt = "In teaching others we teach ourselves.";
```

There are a few important things to note about this simple script:

- The text string is assigned to the variable txt using a single equal sign.
- The text string is enclosed within double quotation marks.
- An **expression** is a section of valid JSL code that, when run, accomplishes a task. We also refer to it as a "command" or "statement", both of which are familiar computer programming terms to describe some action to be carried out. This example has two expressions or two JSL statements.

- Semicolons follow each JSL statement and glue them together. Semicolons are the operator form of the **Glue( )** function. They tell JMP there is more to do. The semicolon in the last JSL statement is not required, but it does not cause an error if it is included.
- The text enclosed within double quotation marks is magenta in color, and the JSL function **Show( )** is blue. These are the default colors used in the script window to make the code more readable and easier to debug.
- There are spaces in the **Show( )** function. Extra spaces within or between JMP functions or within JMP words are okay, and they can make the code easier to read. The same is true for tabs, returns, and blank lines.
- The Log window is your friend.

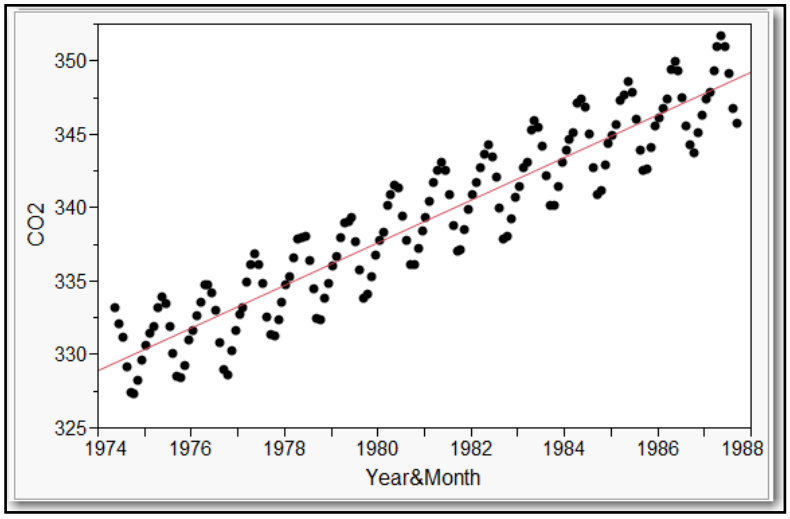
All of these points are covered in more detail throughout the book.

Open, Modify, and Save a Script

In the following example, the JMP Sample Data file CO2.jmp is used. A script opens the data file from the JMP Sample Data file directory. It creates a scatter plot of CO2 versus Year&Month, and then fits a line to the data.

```
CO2_dt = Open( "$SAMPLE_DATA/Time Series/CO2.jmp" );

CO2_dt << Bivariate( Y( :CO2 ),
     X( :Name( "Year&Month" ) ),
     Fit Line()
);
```

Figure 1.5 CO2 Versus Year&Month Fit Line



From the scatter plot, you can see that there is a linear structure to the data. Fitting a line does not tell the entire story. There is structure that remains unaccounted for in the data. To get a

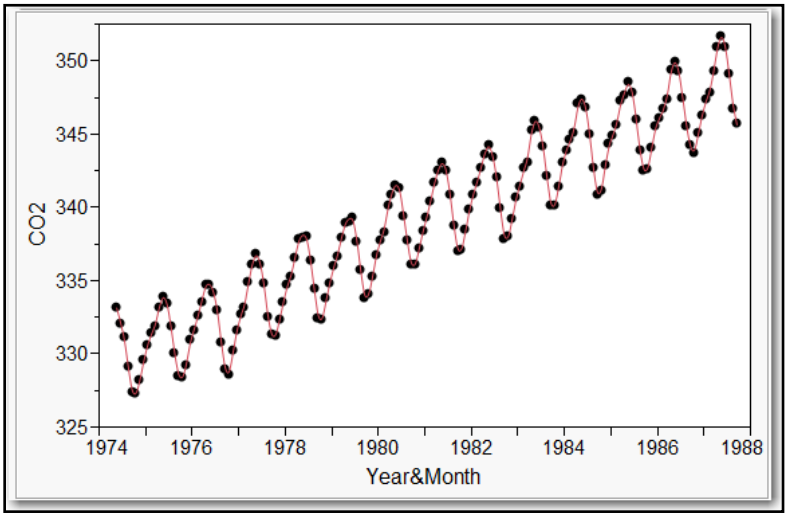
better understanding of the structure of the data, modify the script so that a flexible spline is fit to the data.

To modify the script and fit a spline, replace the **Fit Line( )** command with the **Fit Spline(** 0.0001 **)** command:

```
CO2_dt = Open( "$SAMPLE_DATA/Time Series/CO2.jmp" );

CO2_dt << Bivariate( Y( :CO2 ),
     X( :Name( "Year&Month" ) ),
     Fit Spline( 0.0001 )
);
```

The syntax for the **Fit Spline** command matches the menu option in the **Bivariate** platform. Because the smoothness of the spline is needed, additional information is included in the parentheses. As you learn JSL, you will find that many commands have the same syntax as they do in JMP menu options.

Figure 1.6 CO2 Versus Year&Month Fit Spline



This is an example of scripting, not a proper statistical analysis, so we feel that a brief comment on this example is necessary. The periodicity in the data is obvious—fitting a simple line to the data would usually be insufficient. For an analysis of this data table that better supports prediction, some other method, such as time series or trigonometric regression, is needed.

To save the script, select **File ► Save,** or select **Save As** and provide a filename such as CO2.jsl. The script is saved as a text file that can be opened by any text editor. If the .jsl extension is used, then JMP recognizes it as a type of JMP file, and it will open the file in JMP when it is double-clicked.

To open the script, select **File ► Open,** and navigate through the folders to find the script. You can also double-click on a script to open it, or drag and drop the script into another JMP window or into the JMP Home window.

Make It Stop!

As you become more familiar with JSL, and you learn about iterative looping, an important thing to know is how to stop a runaway script. It's not that hard to write a script that goes into an infinite loop that needs to be stopped.

The following script is one that you will certainly want to stop before it gets to the end. To stop a script, select **Edit ► Stop Script.** Or, if you are in Windows and the caption is in focus, press the ESC key. Many scripts execute faster than you can stop them. Not this one, however!

```
For( i = 99, i > 0, i--,
Caption( Wait( 2 ), {10, 30},
  Char( i )
  || " bottles of beer on the wall, "
  || Char( i )
  ||" bottles of beer; take one down  pass it around, "
  || Char( i - 1 )
  || " bottles of beer on the wall. "
  )
);
Wait( 3 );
Caption( Remove );
```

Stopping a script introduces the concept of handling the flow of a program. As more advanced topics are discussed, the concept of program flow (i.e., starting and stopping a script, error-checking, and capturing user input) are included.

The Script Window

A Bugatti Veyron is to a car what the JSL script window is to a text editor. A car gets you where you are going, but a Veyron can get you there much faster. Similarly, the JSL script window is not just a text editor; its features help you write and debug your script faster.

One of the more useful features of the script window is the ability to show line numbers to the left of the code. (See Figure 1.7.) This helps you keep track of progress and debug the script. If line numbers are not showing, then right-click in the script window, and select **Show Line Numbers**. Even though the default in JMP is that line numbers are not shown, we recommend that this feature be used.

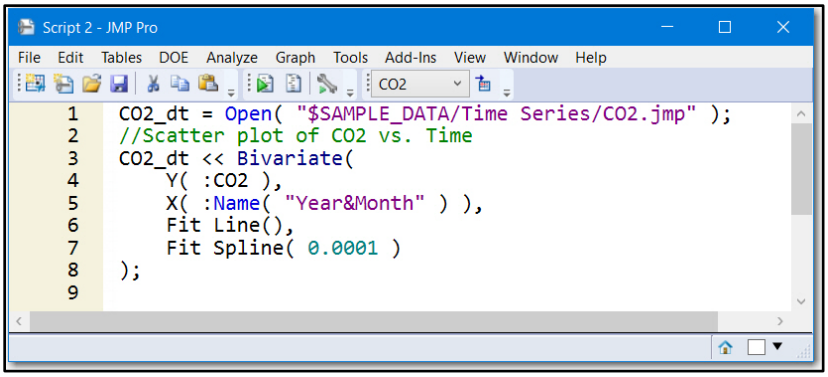
In the script window, several other features are useful and worth mentioning:

- Text for JMP keywords, strings, comments, and scalar values are color-coded.
- The script can be reformatted for readability.
- JSL functions can be auto-completed.

- If you hover over JSL functions or variables, tooltips or values are displayed.
- Fence matching is available.
- The script window can be split either horizontally or vertically, and the Log window can be embedded in the script window.

You can specify code folding markers that allow you to expand and collapse blocks of code, allowing for easier readability of the script.

Figure 1.7 Script Window



Understand the Features of the Script Window

The *JMP Scripting Guide*, included in the JMP installation and available by selecting **Help ► Books,** gives a complete description of the features of the script window. You are encouraged to refer to the guide often for additional details. A script named JMP Script Editor Tour.jsl is also included. It is available by selecting **Help ► Sample Data,** and clicking **Open the Sample Scripts Directory.**

Color of Code

When you create or open a script, you will notice that certain types of words or text are in different colors to make the script easier to read and debug. If you are familiar with SAS, you will notice that the coloring is similar to SAS code. The colors discussed in this section refer to JMP default colors, which are configurable in the preferences. In the script shown in Figure 1.7 and included in the 1_ScriptWindow.jsl script, the following conventions were used:

- JMP functions, such as **Open( )**, are blue.
- Strings, such as Year&Month, are purple.
- Comments are green.
- Scalar values are teal blue.
- Platform names are maroon.
- All other text is black.

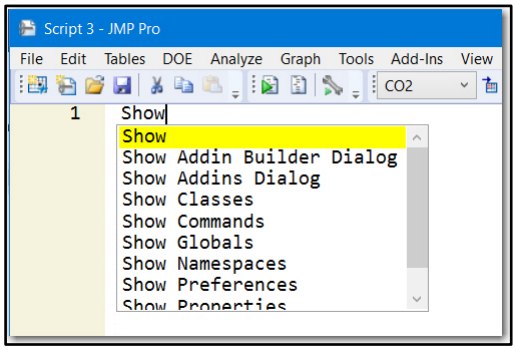
Reformat Script

Everyone has a preferred style of spacing and indenting when scripting. It might make perfect sense to the person scripting, but makes no sense to the people who are trying to interpret the code or debug it. The **Reformat Script** option uses JMP default spacing and indenting to make the script's format standardized and easier to read. When a script window is active, the **Reformat Script** option can be selected either from the **Edit** menu or by right-clicking on the script. A portion of a script can be reformatted by selecting the portion first. When this option is run, if there are syntax problems, such as unbalanced parentheses, missing commas, and so on, an error is produced. The script is not reformatted until the syntax errors are fixed, and the Reformat Script option is run again.

Auto-completion of JSL Functions

If you do not remember the exact name of a JSL function, or if you are just in a hurry, auto-completion helps you complete the correct syntax of the function. To use auto-completion, type the first few characters, hold down the CTRL key, and press the space bar, or hold down the CTRL key, and select the Enter key. For example, as shown in Figure 1.8, if you want to see a list of all JSL functions and messages that begin with the word "show," simply type show, hold down the CTRL key, and press the space bar. The selection box appears. Select **Show Properties**. Auto-completion can be used after a send operator (**<<**) if the variable to the left of the operator is a reference to an object that accepts messages.

Figure 1.8 Auto-completion



Hovering Over Functions and Variables

In the script window, when you hover over a JSL function, a tooltip pops up, and shows a brief summary of the syntax. This is extremely useful if you are new to JSL, and you are getting familiar with functions. Hovering over a variable shows a tooltip about the current value of the variable. The code needs to have been run before JSL assigns a value to a variable. If the code has not been run successfully, the variable name will show in the tooltip when you hover over a variable.

Fence Matching

When we talk about fence matching, we mean matching closing parentheses, brackets, and curly braces with opening ones. There are several facets of this feature.

- When an opening fence is typed, the closing fence is automatically added. If you type the closing fence, JMP recognizes that it has already automatically added the closing one, and does not add the extra one.
- To help check that fences are matched, when you place the cursor on the outside of a fence, its matching fence turns blue. If there is no matching fence, the unmatched fence turns red.
- To select the fences and the text within them, either double-click on a fence, or place your cursor inside the fence, hold down the CTRL key, and select the] key.

Split Window

The script window can be split into two vertical or two horizontal windows. This allows you to work on the same script in two different windows. You can scroll and edit in both windows, and when a change is made in one window it is updated in the second window. To split the script window, right-click in the window and select **Split,** and then choose either **Horizontal** or **Vertical**. To revert to a single window, right-click on the script window and select **Remove Split**.

The script window can also be split so that the Log window appears at the bottom. We find this useful for quick debugging of smaller scripts but prefer having the full Log window when doing a lot of debugging. To show the embedded Log window, right-click on the window and select **Show Embedded Log**.

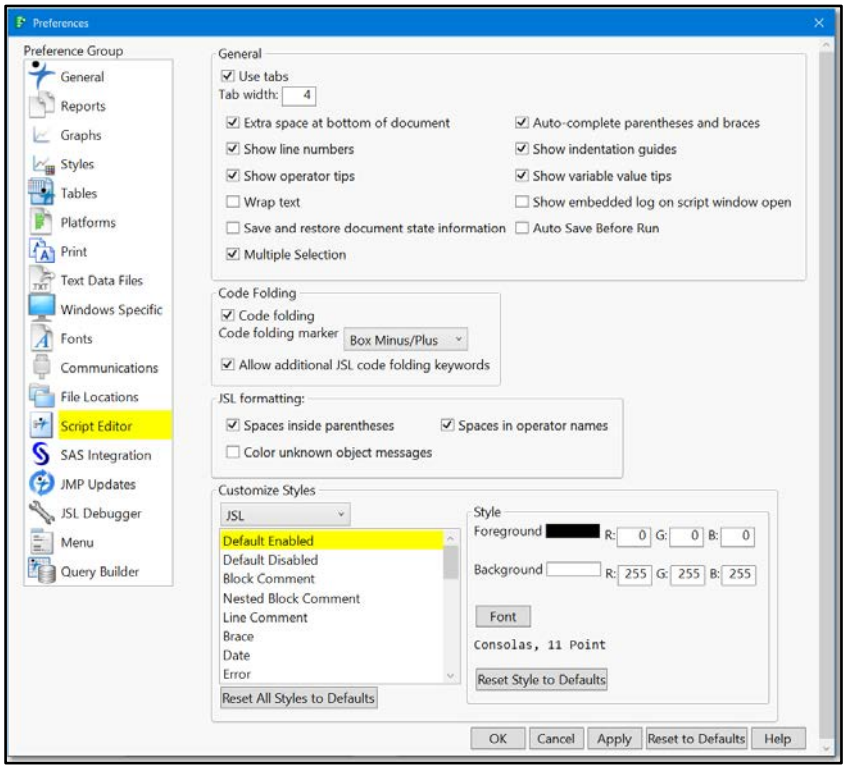
Code Folding Markers

Code folding markers allow the user to hide and display blocks of code, which is convenient when working with longer scripts. If code folding is turned on in the Script Editor preferences, as shown in Figure 1.9, JMP will recognize a list of key words and apply code folding. It is possible to add your own key words to this list. For details on how to customize code folding, see the *JMP Scripting Guide*. We have included an example script of key words that we use for code folding. It can be found in the script 1_jmpKeywords.jsl.

Change Script Window Preferences

When you select **File ► Preferences**, you can change the current preferences for the Script Editor. If the preferences have not been changed since installation, then the script window preferences will look the same as they do in Figure 1.9, but with two exceptions. One is the **Show line numbers** option. Even though the default in JMP is that line numbers are not shown, we recommend that this feature be used. This feature helps you debug code because the error message typically includes a line number. The second change to the default settings is to enable the **Code folding** feature. This is especially useful when writing and debugging longer scripts.

Figure 1.9 Script Window Preferences



Options can be deselected. However, we have found the default options to be useful, in addition to selecting **Show line numbers** and **Code folding**.

The font used in the script window can be changed. To change the font, select **Fonts** in the **Preference Group**. The **Mono** option controls the font for the script window.

There a few more items to note about the script window:

- From the **Edit** menu, the **Search** option includes a **Find** (**Replace**) function that supports the use of regular expressions. All of the features in the Search option are available for use in the script window.
- You can even script the script window, which is a more advanced topic that is not covered in this section. Briefly, information from one script can be captured and written to another script. You can read or write lines of code from one script and store them as a variable to be used later, or you can write them to another script.

The Log Window

When you are scripting, access to the Log window is essential. When a JSL script is run, the Log window captures messages from JMP about the code, errors, and JSL commands and syntax. This information is invaluable as you write scripts. You might want to arrange your windows so that the script window and the Log window are side-by-side. Alternatively, right-click on the script window and select **Show Embedded Log**. This way, you can run portions of the script or the whole script, and immediately check the Log window for errors. The Log window is basically a script window without line numbers. In fact, JSL code can be executed from the Log window. The Log window is unique in that it captures messages from JMP when the code is run, replicates the executed code, and allows the user to write messages to the Log window. It can also capture messages that will help you write your script.

View the Log Window

If the Log window is not available when JMP is opened, you can open it by selecting **View ► Log**, or by holding down the CTRL key, and selecting the Shift key and L. You can set your preferences so that the Log window appears only when explicitly opened, when text is written to the log, or when JMP is started. If you plan to do a lot of scripting, then setting the Log window preference to open when JMP is started is recommended.

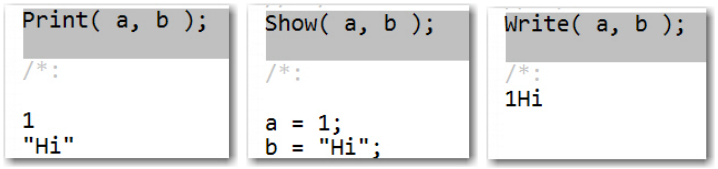
Send Messages to the Log Window

The three functions **Print( )**, **Show( )**, and **Write( )** send messages to the Log window. The **Print( )** function writes text or variable values to the Log window. Each variable value is on a new line, and text is enclosed within double quotation marks. The **Show( )** function is similar to **Print( ).** However, the Show function also includes the variable name, and sets the variable equal to the value. The **Write( )** function is similar to **Print( )**, but it does not enclose text within double quotation marks, and it writes everything on a single line unless a return sequence (\!N) is included. For more information on controlling line spacing as well as other escape sequences, see the *JMP Scripting Guide.* Also. there are a few comments in the 1_LogWindow.jsl script.

To show how each of these functions works, run the first two lines of the 1_LogWindow.jsl script, followed by the **Print( )** line, the **Show( )** line, and then the **Write( )** line functions. Figure 1.10 shows the results. Note the differences between the three functions. The **Show( )** function includes the variable names. The **Write( )** function does not enclose the text within quotation marks, and it writes all of the output on one line.

```
a = 1;
b = "Hi";
Print( a, b );
Show( a, b );
Write( a, b );
```

Figure 1.10 Log Window Output



Clear and Save

You will often want to clear the contents of the Log window so that you can see new messages sent to the window. To clear the Log window, right-click in the Log window, and select **Clear Log**. Or, you can select **Select All,** and then select the Delete key**.** A keyboard shortcut is to hold down the CTRL and the A keys, and select the Delete key.

If you want to save the contents of a Log window, click on the Log window, and select **File ►Save As.** The default file type is .jsl, and a text file option is available.

Review Error Messages

If there are errors in a script, the messages sent to the Log window will help you debug the code. (There is an entire section in Chapter 11, "Debug Your Scripts," devoted to debugging code. The section here focuses on the output sent to the Log window.) If you run a JSL script with errors, there are three different types of error messages that JMP might produce in the Log window.

1. A JMP Alert. This pop-up window gives a brief message about the type of error encountered, and specifies the line number where it occurred. This type of error halts the execution of the code, and requires the user to click **OK**. The error message in the pop-up window is written to the Log window. In the script window, the cursor moves to the place where the error occurred.

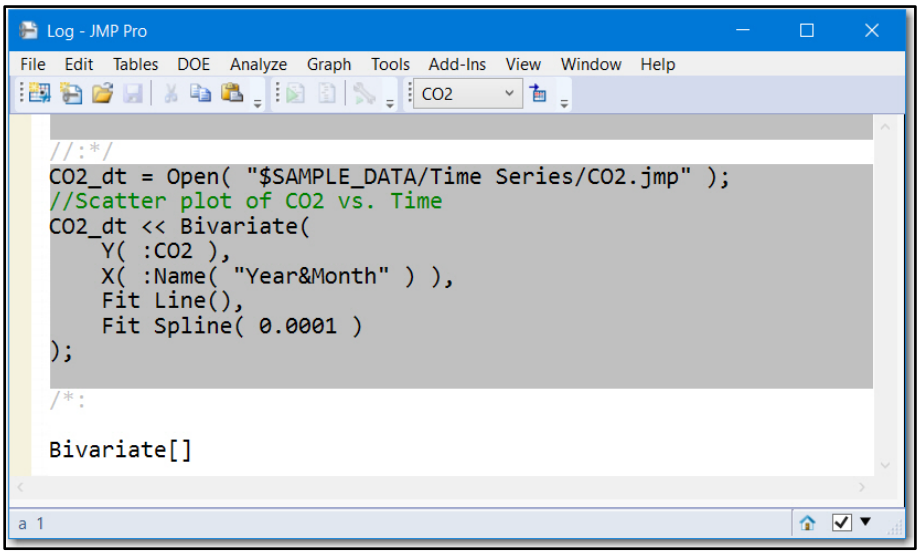
2. The special symbol **/*###*/**. This symbol is embedded in the code that is written to the Log window. The symbol is placed where JMP encounters the error, and an error message precedes the code. We call this "getting pounded."

3. The message **Scriptable[ ]**. This message doesn't always indicate an error, but it is a message that JMP writes to the Log window if there are no syntax errors and no other output produced by the script. This message indicates that the script was executed. It can also indicate that there might be a problem with the code if output was expected.

Example

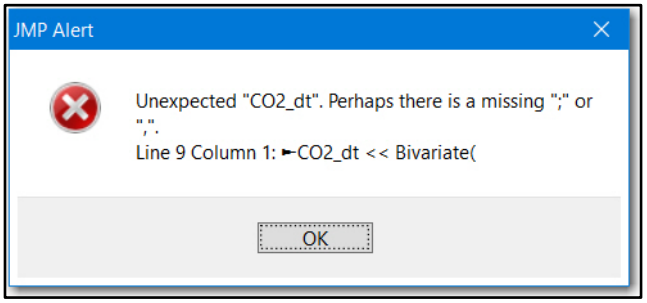
Figure 1.11 shows the Log window after running the CO2.jsl script. Note how the code is written to the Log window with a gray background. The command **Bivariate[ ]** is printed at the end because it is the result of the executed code.

Figure 1.11 Log Window for the CO2.jsl Script



If the semicolon is omitted from the first line of code, the following error occurs. It suggests what the issue might be, and provides the line number.

Figure 1.12 JMP Alert: Missing Semicolon



The following description of the error is written in the Log window:

```
Unexpected "CO2_dt". Perhaps there is a missing ";" or ",".
Line 3 Column 1: ►CO2_dt << Bivariate(

The remaining text that was ignored was
CO2_dt<<Bivariate(Y(:CO2),X(:Name"("Year&Month)),Fit Line(),Fit
Spline(0.0001));
```

Suppose that, in this script, the keyword **Open** is spelled incorrectly as Ope. The following error message is sent to the Log window. The error message is not the JMP Alert type—instead, you have been pounded. Note the placement of the special symbol at the end of the line where the misspelled keyword exists, and the error message before the code is replicated.

```
Name Unresolved: Ope in access or evaluation of 'Ope' , Ope(
"$SAMPLE_DATA/Time Series/CO2.jmp" )

In the following script, error marked by /*###*/
CO2_dt = Ope( "$SAMPLE_DATA/Time Series/CO2.jmp" ) /*###*/;
CO2_dt << Bivariate(
Y( :CO2 ),
X( :Name( "Year&Month" ) ),
Fit Line(),
Fit Spline( 0.0001 )
);
```

Get Help with Your Script

This tip might be leaping ahead a bit, but the **Get Script** command is so useful that we can't resist mentioning it. JMP provides commands that help you write your script by sending the syntax to the Log window. After running the CO2.jsl script, if you run the following command, it produces the code to generate the data file CO2.jmp:

```
Current Data Table() << Get Script;
```

If you run the following code, it lists all of the messages that are available for the data table:

```
Show Properties( Current Data Table() );
```

A Few Items to Note

- When you send the **Get Script** command to a data table, the Log window captures the syntax of the data table. This will help you write your code. Or select the red triangle menu near the table's name, then select **Copy Table Script**, open a new script window, and paste.

Let JMP Write Your Script

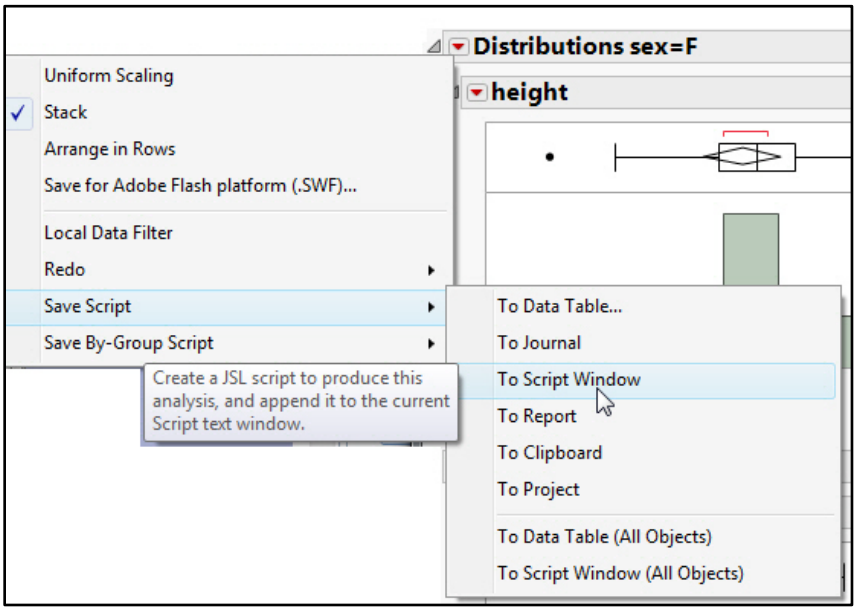
The most efficient scripter ever on this planet is JMP itself. JMP writes scripts from generated reports or table manipulations. This feature enables a novice scripter to write scripts in a matter of minutes. While teaching an introductory four-hour JSL class, we have seen novice scripters write fairly complex scripts by combining different pieces of code produced by JMP in a script window. Even advanced scripters take advantage of JMP writing their code. It saves them time, ensures that there are no typos, and eliminates the need to search for forgotten syntax.

Capture a JSL Script from a Report

There are numerous ways to capture a script from a JMP report. In addition to capturing the script, you can capture enhancements to the report such as reference lines, changes to the axis

scales, inclusions and exclusions of options, and much more. If you click on the top left inverted red triangle in a report window, there is a Save Script option. If the report produces an analysis using a By Group, then there is also a Save By-Group Script option. Figure 1.13 shows the options available under Save Script. Only the options directly related to scripting are discussed in this section.

Figure 1.13 Capturing Scripts



To Data Table...—This option saves the script as a table script to the table panel of the data table that generated the report.

To Journal—This option creates a link on the current journal, or opens a new journal if one is not open. The link runs a script that reproduces the report.

To Script Window—This option saves the script for the object to a new script window (if one is not open), or appends it to an open script window.

To Report—This option writes the script to the top of the report window.

To Clipboard—This option copies the script so that it can be pasted into a script window, text file, or any other program that handles text.

To Project— This option saves the script for the object to a new project window (if one is not open), or appends it to an open project.

To Data Table (All Objects)—This option saves the script for all objects in a report as a table script to the table panel of the data table that generated the report. When you save a script for

all objects, the **Where** clause defines what is included in a report. It combines all objects in a single window using the **New Window( )** function.

To Script Window (All Objects)—This option saves the script for all objects in a report to a new script window (if one is not open), or appends it to an open script window. When you save a script for all objects, the **Where** clause defines what is included in a report. It combines all objects in a single window using the **New Window( )** function.

Capture By-Groups Analysis

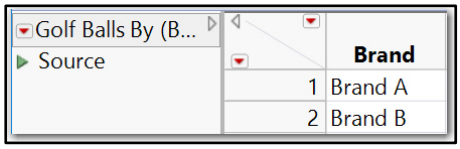
In addition to the Save Script option, there might be a Save By-Group Script option. The Save By-Group Script option appears if the report produces a By-group analysis. The options available in **SaveBy-Group Script** are a subset of the options available in **Save Script**. The difference between **Save Script** and **SaveBy-Group Script** is that SaveBy-Group Script saves the script using the JSL command **By**, and reproduces the analysis as if you used the **By** command in a dialog box. With Save Script, you save the script for all objects, a new window is created, and each object is added to the window.

Capture Table Manipulations

At this point, you know how to save a script from a report that JMP generates. Now, you are going to find out about one of the most powerful and essential features in JMP—its ability to easily manipulate data tables.

When a new data table is generated from a **Tables** menu command, the new data table has a table script called **Source**. The JSL code that generated the new data table from an original table is included in the Source table script (also called a table property). However, there are a few exceptions in which the code is either not captured or not that useful.

Figure 1.14 Source Table Script



- If you replace a table as a result of selecting **Tables ► Sort**, the code is not saved in the Source table script. If you want the code to be saved, do not replace the table when you do the sort. If you do not replace the table, the code is added to the new data table. You can copy and paste the code to another location, and add the option **Replace Table**.
- If you select **Tables ► Subset**, the row numbers of the selected rows are included in the code. Having the row numbers is not very useful unless you want the same row numbers every time the code is run. Keep in mind that, if you are writing a flexible script, you must select rows and columns before selecting **Tables ► Subset**. The commands that select portions of a data table are discussed in detail in Chapter 3, "Modifying and Combining Data Tables."

Example

In this example, the JMP Sample Data file Golf Balls.jmp is used to demonstrate the ability to capture a JSL script from a report and to create a summary table. These two elements are combined in a script window, and they work together to produce the needed output.

Suppose you are asked to examine the distance and durability of different brands of golf balls. You have collected information about three brands. You analyze these brands, but you know that additional brands will be added later, so you want to script a generalized analysis. Here are the three operations required of the script:

1. Create a scatter plot of the relationship between distance and durability. You want to use different colors for each brand to highlight differences in the relationships by brand.
2. Create side-by-side box plots to compare the brands for each response.
3. Create a data table that summarizes the mean and range of distance and durability by brand.

The scripting of these tasks can be accomplished by letting JMP do the work for you! Follow these easy steps:

1. Open the JMP Sample Data file Golf Balls.jmp.
2. In JMP, create a scatter plot of Distance versus Durability. Use the **Fit Y by X** platform, and add a legend that identifies colors and marks by Brand.
3. Click on the inverted red triangle in the scatter plot, and select **Save Script ► To Script Window.**
4. Create multiple box plots in the **Fit Y by X** platform using Distance and Durability as your Y value, and Brand as your X value. After the box plots are created, click on the inverted red triangle again, and select **Save Script ► To Script Windows**. The script is saved to the same script window used in the previous step.
5. Create a summary table with the mean and range of Distance and Durability, with Brand as the group variable. Click the table script **Source,** and select **Edit**. Right-click on the script, select it, and copy it. Paste the script in the script window used in the previous steps.

This script is now complete. Because this is a simple script that was captured directly from JMP, there are no variable references to tables. As a result, before you run the script, close the summary table that was created. Otherwise, the script can become confused about which data table to use. For your convenience, the script used in the previous example is included for downloading. It is named 1_LetJMPWrite.jsl.

As you script more, you will want to enhance your script. For example, you will likely want to open the data table directly in the script, reference the data table so that the correct one is always used, and format and save the output. The previous example demonstrated how to write a simple script, but remember you can do so much more!

Get More Help with Your Script

By now, you know that JMP sends valuable information to the Log window. This includes information about a data table generated by the **Get Script** command, or the **Copy Table Script** that can be pasted into a script window or the Log window. Both options provide the JSL code to re-create the table: commands for adding rows, table variables, columns, column values, formulas, and so on. The output will be very long for large tables. A helpful tip when using a large table is to subset the data table to include only the first row of the table. The resulting script in the Log window (or script window) shows the structure of the table, but the length of the output is now shorter and easier to read.

A Few Items to Note

- JMP captures the code required to run analyses or to perform data table manipulations. However, putting the code together in a logical flow, and then adding appropriate references to data tables and reports are both critical changes that need to be made to the script for it to run correctly and efficiently.
- JMP captures many items, but it does not do everything. For example, it does not select rows, open tables, reference tables, reformat output, save reports, or save output.

Objects and Messages

In the theater, a script or screenplay is a set of instructions for directors, actors, and stage hands. In JMP, a script is a set of instructions for creating and manipulating JMP objects. JMP objects include tables, columns, reports, windows, displays, dialog boxes, and much more.

Reference Objects

Like in a screenplay, an instruction needs to have a target or a reference. The instruction, "Enter stage right" needs to be targeted for an actor or an object (for example, "Mariachi Band: enter stage right"). Similarly, JSL instructions need a target or reference. Suppose you have the following simple instruction:

```
Distribution();    //instruction to open the Distribution platform dialog
```

Note: This JSL code has the same effect as selecting **Analyze ► Distribution** from the JMP main menu.

If you do not have a table open, an **Open Data File** dialog box appears. After you open the data file, the **Distribution** dialog box appears.

Open several JMP tables, and run this command. This time, only the **Distribution** dialog box appears, and **Select Columns** lists the columns of the current data table. To direct this command to a specific table, a table reference is required.

```
BC_dt    = Open( "$SAMPLE_DATA/Big Class.jmp" );
Candy_dt = Open( "$SAMPLE_DATA/Candy Bars.jmp" );

BC_dt << Distribution(); //open Distribution dialog for Big Class
```

Now, let's look at the general syntax of a command:

```
result_reference = object_reference << message(arguments);
```

The result_reference is a variable that is referenced later in the script or JMP session. The object_reference is an object in JMP that can be acted upon, such as a data column, data table, window, or graph. The message(arguments) is a named task with precise syntax that is associated with the object. Messages are object specific. For example, **Sort( )** is a valid table message, but it is not a valid graph message. The **<<** is a send operator that sends the message to the object. JMP objects that have associated messages and properties are described as *scriptable*.

The code above shows how to define a reference to a data table and how to send a message to a data table. Data tables and columns are probably the most common objects that are referenced. There are various ways of referencing a column. It is recommended that the data table is explicitly defined when a column is referenced. This is not required, and JMP will use the current data table if no data table is defined. However, this is poor practice and will likely cause issues for you at some point.

Below are examples of the different ways of referencing columns. Note that the first column in the data table, dt, is ID so that each of the four lines of code references the same column. The two lines of code that use the data table reference, dt, are explicit in defining the data table. The last two lines of code refer to the ID column by the column number, which can cause issues if the script is run again and the order of the columns has changed. The recommended syntax to define a reference to a column is the first line, where the column is referenced by name and the data table is explicitly defined.

```
ID_col1 = Column( dt, "ID" );
ID_col2 = Column( "ID" );
ID_col3 = Column( dt, 1 );
ID_col4 = Column( 1 );
```

The script 1_ReferencingColumns.jsl has more information on referencing columns. There are parts of this script that are more advanced, so you might want to come back to it as you learn more about JSL.

Send a Message

As you script, there are two methods to quickly determine what messages are appropriate for scriptable objects.

The first method is to use JMP Help. In JMP, select **Help**. In the Help menu, you will find the options **Statistics Index** and **Scripting Index**.

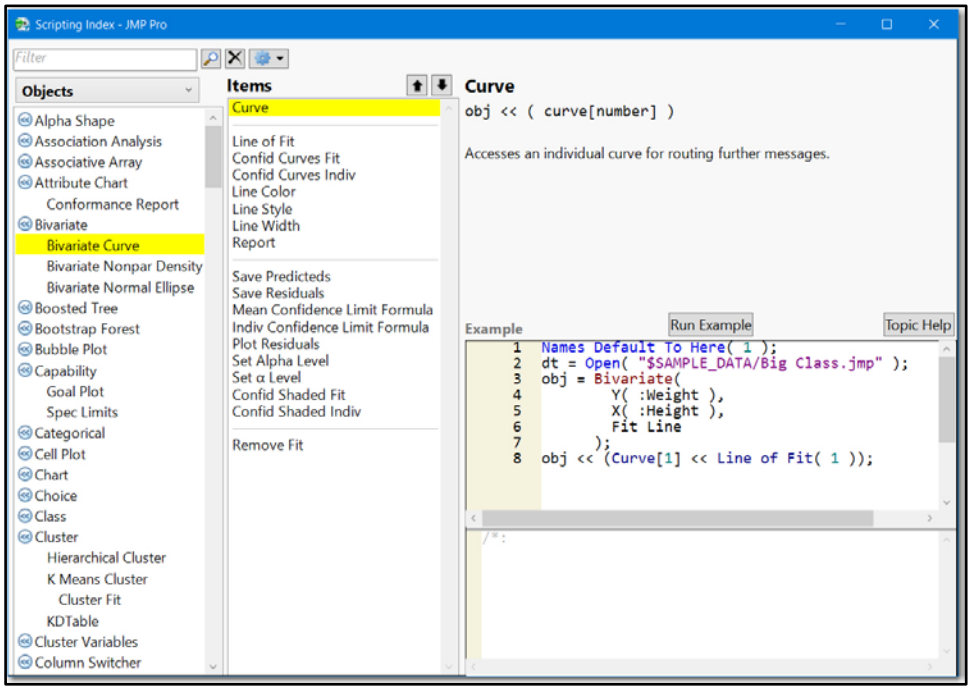
These indexes provide topic help, syntax help, and example scripts that are ready to run. Figure 1.15 displays the **Scripting Index** for **Objects** for the **Bivariate Curve** As noted above in the general syntax for sending messages to objects, the **Scripting Index** shows that the **<<**

operator is used to add a curve to the bivariate object. The JMP Scripting Index has saved us countless hours of looking for the correct syntax in the *Scripting Guide*, or searching through numerous project folders for an existing script where a specific command was deployed successfully. Regardless of your experience and knowledge, you should explore the index.

A Few Items to Note

- The **Scripting Index** includes the options **All Categories**, **Functions**, **Objects** and **Display Box**, and a filter field. Select **All Categories** and type "curve" in the filter field. The display now shows all items in the index that include curve.
- If you are not sure which filter to select, select **All Categories**. If you are looking for help with options for a JMP table (Data Table), graph, or analysis report, select **Objects**. Otherwise, use **Functions**. **Display Box** will be important when building custom dialogs and displays.

Figure 1.15 Object Scripting Index for the Bivariate Curve



When sending a message to a column, it is recommended that the data table is used explicitly in the Send statement. The following lines of code come from the script 1_ReferencingColumns.jsl. Note the two different choices in syntax that can be used for sending a message to a column. Both achieve the same result, so it is a matter of choice.

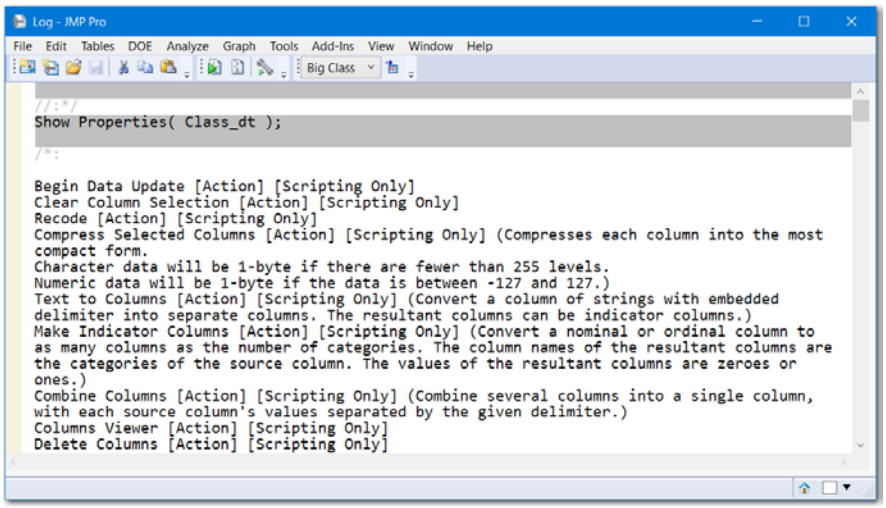
```
//Examples of sending messages to columns
Name_col = Column( dt, "Name" );

//change the name of the Name column to First Name
Column( dt, "Name" ) << Set Name( "First Name" );
```

```
//set the column name back to Name
dt:Name_col << Set Name( "Name" );

//an alternative syntax for setting the column name back to Name
dt:First Name << Set Name( "Name" );
```

The second method is to use the **Show Properties(**reference**)** command. This command can be typed in the Log window or in a script window and run. If you type it into the Log window, all messages are listed.

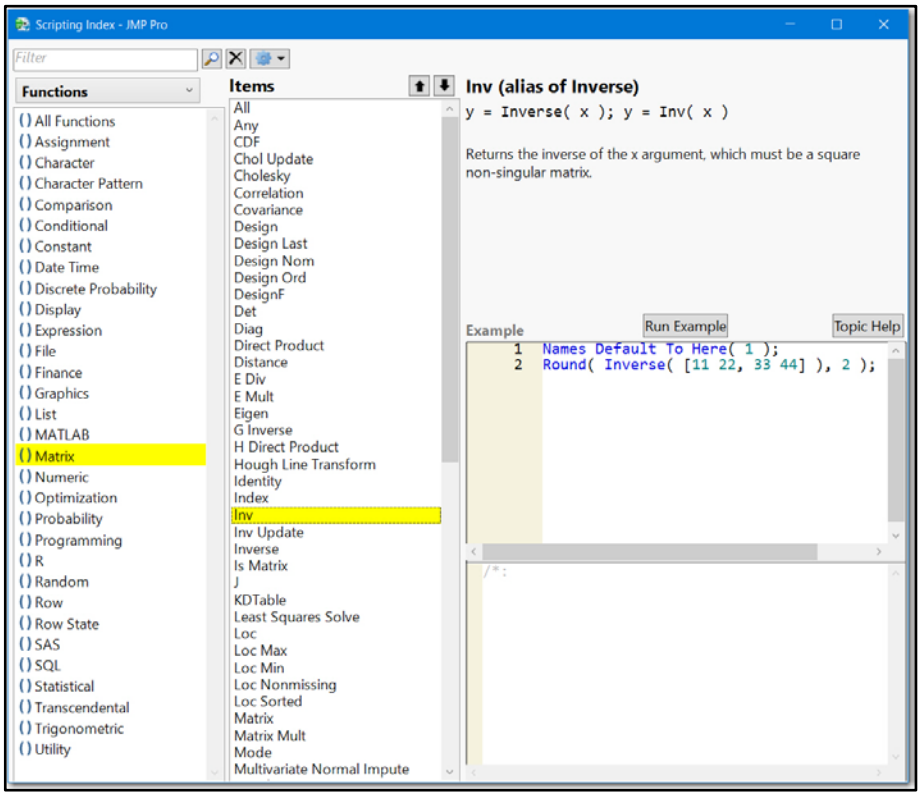
```
Class_dt = Open( "$Sample_Data/Big Class.jmp" );  //table reference
ageCol   = Column( class_dt, "age" );  //column reference

//--a1 is the value in the first row of age
a1 = ageCol[1];

//--ageVal is a vector of all values in ageCol
ageVal = ageCol << get values;

//---table is a scriptable object with numerous messages
//--includes Table/Analyze/Graph commands
Show Properties( Class_dt );

//---column is scriptable with many messages
Show Properties( ageCol );

//---a global variable is not scriptable, no messages
Show Properties( a1 );

//---a vector [or a list] is not scriptable,
//---no messages
Show Properties( ageVal );
```

Figure 1.16 Show Properties Output in Log Window



Variables representing numbers (such as a1), strings, vectors (such as ageVal), matrices, lists, and expressions are JMP objects. However, they are not scriptable objects because they do not have inherent messages. They have functions and lexical rules (methods) to propagate new objects, and get information. Notice the long scrollable list of available functions for the matrix data structure in Figure 1.17.

Figure 1.17 JSL Functions Index for Matrix



The **Scripting Index/Functions** is a superset of categories from the **Formula Editor**. There are no **R**, **SAS**, or **Utility** categories in the **Formula Editor**.

You should browse the **Utility**, **File**, and **Programming** categories, which include definitions and example scripts for getting information about objects in a script or for communicating with script users.

The 1_ObjectProperties.jsl script includes a list and a vector constant. Both of these are important JMP data structures.

Punctuation and Spacing

In most languages, punctuation can be defined as the use of standard marks in writing to separate words into sentences, clauses, and phrases in order to clarify meaning. Similarly, words in the JMP Scripting Language are separated by commas, quotation marks, parentheses, semicolons, various operators (such as {}, /, +, –), and so on. It is important to use punctuation properly to clearly express your scripting intentions. In most situations, the existence of a space, tab, or return, inside or between operators or within words, is treated by JMP as if it doesn't exist. However, there are a few situations where one of these *does* matter. This section shows some good and bad examples of punctuation and spacing. The examples are included in the 1_PunctuationSpacing.jsl script.

Use Punctuation

In JSL, commas separate items, such as elements in a list, rows in a matrix, or arguments in a function. Semicolons glue functions together. Curly braces and brackets define lists, subscripts, and matrices. Strings are enclosed in double quotation marks. Parentheses delimit function arguments, and group arguments in an expression.

Consider the following lines of a script:

```
thislist = {3, 7, 31};
thatlist = {"Oregon", "Arizona", "New Mexico"};
thismatrix = [1 2 3, 4 5 6, 7 8 9];
For( i = 1, i < 10, i++,
  Show( i, Factorial( i ) )
);
```

Note how the commas separate the items in the script, whether the items are elements in a list, rows in a matrix, or arguments in a function. Each statement in the script has a semicolon at the end that glues it to the subsequent statement. Lists can be defined by curly braces or with the **List( )** function. Matrices can be defined by square brackets or with the **Matrix** function. Strings are enclosed within double quotation marks. Parentheses tell functions what arguments to evaluate. It is important to follow an opening brace, bracket, double quotation mark, and parenthesis with a closing one to avoid errors or unintended consequences. As mentioned previously, the JSL Script Editor can automatically complete incomplete parentheses and braces by selecting the Auto-complete parentheses and braces check box in **Preferences**. This makes it more difficult (although not impossible!) to make this error. We strongly recommend using this feature.

What if you need to use double quotation marks in a string? JSL provides several escape sequences (search the *JMP Scripting Guide* for "escape sequences" for a full list). A backslash bang (\!) is the escape sequence to use when you need double quotation marks. For example, if you want the quoted string "Rescue me!" in a **Caption( )**, then your script would look like the following:

```
Caption( "\!"Rescue me!\!"" );
```

If a string requires multiple quoted text, begin the text with \[and end with]\,

This eliminates the need for the escape sequence \! before every quotation mark.

```
Caption( "\[Specify a list of columns as {"weight", "height", "age"}]\" );
```

Use Spacing

In most situations, JSL doesn't care about a space or tab, or a line or page delimiter in a name. In fact, JMP acts like these don't exist. Although there are valid justifications for this behavior, it can be important for a scripter to know.

Consider these lines of the previous script:

```
thatlist = {"Oregon", "Arizona", "New Mexico"};
thismatrix = [1 2 3, 4 5 6, 7 8 9];
```

The Log window displays the same list when you run show(thatlist); or show(th at list);, which is not a problem. However, the space between "New" and "Mexico" in "New Mexico" is important because it is part of the string. For instance, a search for "NewMexico" would not find the desired state name in that list. The columns of the matrix are defined by spaces, and thismatrix has three rows (separated by commas) and three columns (separated by spaces). You get very different matrices with and without spaces and commas!

The two-character operator, such as ||, >=, <=, :*, ++, or /*, cannot have a space between the characters to be understood correctly.

You can disable **Spaces inside parentheses** and **Spaces in operator names** in the **Script Editor Preferences.** They are enabled by default. We esthetically prefer the formatted scripts that the default settings generate.

Above all, we encourage you to develop a consistent style in your scripting. Since spaces are important only in a few situations, spacing can be a style element. Spaces can make your scripts more descriptive and easier to read and understand.

Rules for Naming Variables

Simply put, everything you plan to use later in a script needs a name. Relative to some other languages (like SAS, for example), scripts written in JSL have fewer rules for variable naming. However, knowing them and following them can save you a lot of time and effort, not to mention sanity.

As stated in the *JMP Scripting Guide*, any valid JMP object can be assigned to variable name:

- columns and table variables in a data table
- non-scriptable objects, such as, numbers, strings, lists, matrices, expressions, and references to objects
- types of scriptable objects
- parameters and local variables inside formulas

A variable name in a script is resolved the first time it is used. This typically happens when getting or setting a value. The variable value persists forever, or at least until it is deleted or changed, or until the JMP session is ended.

All variable naming rules are listed in the *JMP Scripting Guide*. It is a good idea to understand them before getting too far down the road on your JSL journey. This section highlights what we think are the more important naming rules.

First off, scoping is an important concept. Scoping syntax tells JMP how to interpret a variable name in a case that could be ambiguous, such as when you have a data table column and a JSL global variable with the same name. Scoping is described in more detail in Chapter 4, "Essentials: Variables, Formats, and Expressions," and Chapter 9, "Writing Flexible Code."

Open the 1_Naming.jsl script, which uses the Sample Data table Body Measurements.jmp. Run the following code:

```
Clear Symbols();  //erases the values set for variables

//Open data table and define Body Measurements.jmp dataset as BMI_dt.
BMI_dt = Open( "$SAMPLE_DATA\Body Measurements.jmp" );

<80Waist_col = New Column( "<80 Waist" );
```

The data table name BMI_dt resolves without error. However, there is going to be a message, "Unexpected <", in the Log window. And, the column naming error in the last line is so egregious that JMP displays an Error Alert window.

To fix these problems, either of the following two lines can be used:

```
lt80Waist_col = BMI_dt << New Column( "<80 Waist" );
Name( "<80Waist_col" ) = BMI_dt << New Column( "<80 Waist" );
```

The first line works because the variable name starts with an alphabetic character. The second line works because of the special parser directive, the **Name("…")** convention. If the second line of the script is run before deleting the column created in the first line, then the column name will be <80 Waist 2 because a column in the table is already named <80 Waist. It is not recommended to use variable names that require the use of the **Name("…")** convention.

The general rule of variable naming in JSL is to start with an alphabetic character or an underscore. After the letter or underscore, numbers, spaces, and some special characters can be used with abandon. Using the **Name("…")** convention allows the use of all special characters. You can even use double quotation marks with the backslash bang (\!) operator. Keep in mind that unconventional names can cause problems when exporting data to other formats, particularly when using ODBC.

JSL ignores spaces and tabs in variable names, unless they are enclosed within quotation marks. For example, Var1 2 is equivalent to var12.

There are some reserved words that might cause problems with variable naming. These reserved words are mostly functions. Our advice is to avoid using JMP keywords/functions as variable names. Here is an example:

```
beta = 0.05;
```

In this case, there is a function for a distribution named beta. The variable name can be explicitly resolved by using the special parser directive:

```
Name("beta")=0.05;
```

In the section "Rules for Name Resolution" in the *JMP Scripting Guide*, JMP has eight possible resolutions for a name in a JSL script with some exceptions to these. It is important that you have a basic understanding of these rules, so spend some time reading that section of the *JMP Scripting Guide*. Here are some of the key rules:

- Look it up as a function if it is followed by parentheses.
- Look it up as a table column or table variable if it is preceded by a colon.
- Look it up as a global variable if it is preceded by a double colon.

A colon can be used as a scoping operator. Here is an example:

```
::var;  // Var is a Global Variable

:var;   // Var is a Table Column

var;    // Depends on when first used
```

Capitalization is ignored by JSL. But, that doesn't mean that you should ignore capitalization when you write a script. The use of mixed case to name columns can make scripts easier to read and understand. Capitalizing platform names (such as **Distribution**, **Bivariate,** etc.) can make them more obvious and easier to find.

Develop your own consistent style when naming data tables, columns, global variables, etc. Using a standardized style can help a team of script writers generate consistent and understandable scripts that can be seamlessly integrated. More often than not, a consistent naming style can save many hours of frustrating work. Whatever style you choose, you might be typing the name over and over, so keep the names simple and descriptive. Always use comments liberally in your scripts to help others understand your intentions and logic. You might find that these conventions help you when you revisit scripts that you wrote weeks, months, or years ago.

Operators

Without operators, scripting might read like a novel. Operators get things done! JSL has different types of operators: arithmetic, logical, matrix, comparison, and more. Operators are one- and two-character symbols for common arithmetic actions. Table 1.1 is a table of common

JSL operators and their functions. In earlier versions of JMP, operators and functions were not distinguished from one another. All operators have associated functions but not all functions have operators.

There are three basic categories of operators: prefix, infix, and postfix. As you might guess from the names, a prefix operator comes before the operand (the object being acted upon). An example is the negative sign (-). An infix operator comes between the operands. An example is the subtraction sign (–). And, a postfix operator comes after the operand. An example is the decrement sign (--). Some operators can be of several types, depending on their use. Operators can be substituted with JSL functions. For example, c = a - b can also be performed with c = Subtract(a, b).

All operators are documented in the *JMP Scripting Guide*. Here are several common operators with script examples and descriptions. These examples are in the 1_Operators.jsl script.

Table 1.1 Common JSL Operators

Operator	Function	Script Example	Description
Arithmetic Operators			
Prefix: - (unary)	**Minus**	Result = -a	Result returns the negative of a.
Infix: **+, -, *, /, ^**	**Add, Subtract, Multiply, Divide, Power**	Result = a*b/c–d^e	Result returns a times b divided by c, that quantity minus d raised to the power of e.
Postfix: **++, --**	**Post Increment, Post Decrement**	Result = 0; For(a=0, a<=100, a++, Result=Result+a);	Result returns the sum of the numbers from 0 to 100.
Assignment Operators			
=	**Assign**	Result=a;	Assigns the current value of a to Result; replaces Result with a.
Comparison/Logical Operators			
==	**Equal**	Result==a;	Boolean logical value for comparisons. Returns 1 if true, 0 if false. Missing values in either Result or a causes a return value of missing. This case evaluates as neither true nor false.
<, <=	**Less, Less or Equal**	a<b	Returns a 1 if a is less than b, a 0 if not; missing values in either a or b return missing.
>, >=	**Greater, Greater or Equal**	a>=b	Returns a 1 if a is greater than or equal to b, a 0 if not; missing values in either a or b return missing.

Operator	Function	Script Example	Description
&	**And**	Result=a & b;	Boolean logical **And( ).** Returns true if both are true. See the paragraph below about behavior for missing values.
|	**Or**	Result=a | b;	Boolean logical **Or( ).** Returns true if either or both are true. See the paragraph below about behavior for missing values.
Other Operators			
{ }	**List**	Result = List(a,b); Result = {a,b};	Lists are containers in which to store different objects. Lists are powerful. They are discussed briefly in this chapter, and are covered more completely in Chapter 5, "Lists, Matrices, and Associative Arrays."
||	**Concat**	Result=a || b;	Appends b to a.

This table is not a comprehensive list of all of the operations in JMP. On the contrary, there are an infinite number because you have the power to create your own function. For example, you could create a function named Mag that finds the magnitude of a column vector: Mag = function({a},sqrt(a`*a));. The possibilities are endless!

Missing values require special attention. For most logical and comparison operators, any missing values in the calculation return a missing value in the result. In other words, almost all calculations involving a missing value return a missing value. There are two notable exceptions to this. If one value is missing and another is true, then **Or( )** returns true. If one value is missing and another is false, **And( )** returns false. Only numeric values are considered missing. A missing character value is considered a string of zero length, not a missing value. For a character variable named Result, checking for a missing Result could be accomplished by returning Result == "". The function IsMissing( ) treats spaces as missing, so **IsMissing(**Result**)** returns a 1 (indicating it is missing) and **IsMissing(" ")** also returns a 1. Care needs to be used when using **IsMissing( )** with strings.

Often, successful script writing depends on evaluating operations in a specific order. JSL has a specific precedence of operator evaluation. For example, in the formula d = c * a - b, the expression c * a is evaluated before b is subtracted from the product. It might behoove you to familiarize yourself with precedents in the *JMP Scripting Guide*. The inside-out order of operation can be controlled with an apparent overuse of parentheses. However, some scripting aficionados might consider a plethora of parentheses as gauche and amateurish. The order of evaluation can be surprising, so make sure the operators you use return the results that you intend. Don't be afraid to overuse parentheses!

Lists: A Bridge to Next-Tier Scripting

At this point, you might think that we believe too many characteristics of JSL are *essential*. But, understanding the JMP list object and several list functions is *really essential* for creating scripts that interact with the user and for customizing output. Because lists are so indispensable when writing scripts, this section gives a preview. Additional sections in Chapter 5 provide more in-depth information and examples. As you learn about lists, it might be helpful to read this section first, and then skip to the sections on lists in Chapter 5.

Lists are pervasive in JMP. Here are examples:

- **New Window( )** using <<**Return Results** and Column Dialog boxes save user responses in a list.
- Report windows are lists of display boxes.
- The **Summarize( )** function, which produces results similar to Tables ▶ Summary, stores results in lists and vectors.

Lists are compound data structures for numbers, text, functions, expressions, matrices, and even other lists. The data structure is described as *compound* because it can be a container for other data structures.

Curly braces are the lexical representation for the List function **List( )**. Items in a list are separated by commas.

```
myList = {1, 2, 3};
myFormalList = List( 1, 2, 3 );
Show( myList, myFormalList );  //same result
```

JSL provides many functions to extract information and manipulate lists. These functions are provided in the *JMP Scripting Guide* or in the **Scripting Index** under the Functions menu. A few of the more frequently used functions are in the following table. The examples in this section are included in the 1_Lists.jsl script. For the functions in Table 1.2, the lists are defined as follows:

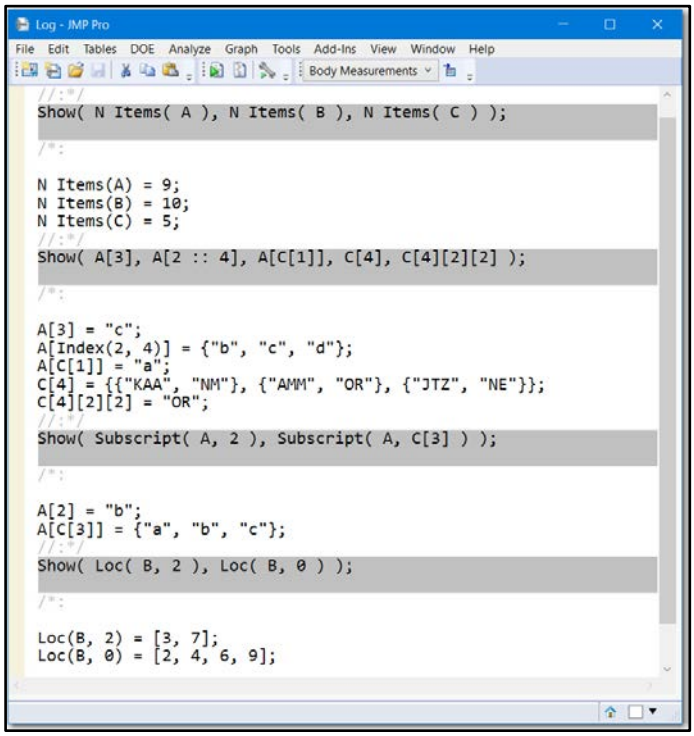
```
A = { "a", "b", "c", "d", "e", "f", "g", "H", "I" };
B = { 1, 0, 2, 0, 3, 0, 2, 3, 0, 3 };
C = { 1, "a", {1, 2, 3}, { {"KAA", "NM"}, {"AMM", "OR"},
   {"JTZ", "NE"} }, [1, 2, 3] };
```

Table 1.2 List Functions for Referencing and Finding Items

Function	Definition
N Items(list **)**	Returns the number of items in the list.
[ ] **Subscript(** list, values **)**	References elements in a list. Brackets are also used to define matrices. For lists, brackets are used to reference elements in a list.
Loc(list, value **)**	Returns a matrix (column vector) of locations in the list where the value is found.

```
Show( N Items( A ), N Items( B ), N Items( C ) );
Show( A[3], A[2 :: 4], A[C[1]], C[4], C[4][2][2] );
Show( Subscript( A, 2 ), Subscript( A, C[3] ) );
Show( Loc( B, 2 ), Loc( B, 0 ) );
```

Figure 1.18 Log Window—Lists



A Few Items to Note

- Index(2,4) is the function format for 2::4, which is equivalent to [2,3,4]. The **Index** function allows a third element for increment, which can be negative. Index(6,1,-2) is equivalent to [6,4,2].
- C[4][2][2] is interpreted left to right. C[4] is equivalent to {{"KAA","NM"}, {"AMM", "OR"}, {"JTZ", "NE"}}. C[4][2] is equivalent to {"AMM", "OR"} and C[4][2][2] is "OR". A sequence of {list}[index1][index2]..[indexN] can be indefinitely long.
- **Contains(**list, value**)** returns the first location of the value in the list, or zero if the list does not contain the value. We think the function **Loc(**list, value**)** is more useful, except in the special case where all values in the list are unique. **Contains( )** returns a scalar, nonnegative integer. **Loc( )** returns a matrix. Loc(A,"c") returns [3] in a 1x1 matrix. **Contains(**A,"c"**)** returns 3.
- Below for Insert Into( ) and Remove From( ), the double colon (::) before a variable makes the variable a global variable. Global variables are discussed in Chapter 4.

Table 1.3 List Functions for Inserting and Removing Items in a List

Function	Definition
Insert(list, value, ,<i> **)**	Returns a copy of the list with a value inserted at the end if a position <i> is not specified. This function can be used to join lists.
Remove(list, i, <n=1> **)** **Remove(** list, {item #s} **)**	Returns a copy of the list with items removed. The starting position is i. By default, one item is removed. In other words, n=1 if it is not specified.
Insert Into(::x, value, <i> **)**	Inserts value into ::x at position i, or at the end if i is not specified. ::x must be a variable. value can be a single item, a list, or a variable.
Remove From(::x, i, <n=1> **)**	Deletes n items in ::x, starting with position i. If n is not specified, only one item is removed. ::x must be a variable.

```
myAList = {1, 2, 3};
myBList = Insert( myAList, 10 ); //{1, 2, 3, 10}
myCList = Insert( myAList, 10, 2 ); //{1, 10, 2, 3}
myDList = Insert( myAList, {10, 11, 12}, 2 ); //{1, 10, 11, 12, 2, 3}

myEList = Remove( myDList, 2, 2 ); //{1, 12, 2, 3}
myFList = Remove( myDList, {1, 3, 5} ); //{10, 12, 3}

myAList = {1, 2, 3};
Insert Into( myAList, 10, 2 );   //{1, 10, 2, 3}
Insert Into( myAList, {15, 22} ); //{1, 10, 2, 3, 15, 22}
xx = {-2, -1};
Insert Into( myAList, xx, 1 ); //{-2, -1, 1, 10, 2, 3, 15, 22 }

myAList = {-2, -1, 1, 10, 2, 3, 15, 22};

Remove From( myAList, 2, 2 ); //{-2, 10, 2, 3, 15, 22}

myAList = {-2, -1, 1, 10, 2, 3, 15, 22};
Remove From( myAList, {2, 4, 6, 8} ); //{-2, 1, 2 ,15}
```

Insert( ) and **Remove( )** do not modify the original list. myList=Insert({1,2,3},{4,5,6}) is valid. However, **Insert Into( )**, and **Remove From( )** have no assignment. They are considered in-place commands.

```
myList = Insert( {1, 2, 3}, {4, 5, 6} );  //{1, 2, 3, 4, 5, 6}
Insert Into( {1, 2, 3}, {4, 5, 6} );      //does nothing

ex = {1, 2, 3};
Insert Into( ex, {4, 5, 6} );  //1st argument must be a variable
Show( ex );
```

Insert Into( ) and **Remove From( )** modify the starting list. The first argument must act on a variable (a place to store the results).

Lists in this section have contained numbers and strings, and lists of numbers and strings. Lists can contain expressions, functions, and matrices. (Expressions are covered in future chapters.)

Assignment lists and function lists are special cases. In Table 1.4, two functions are listed that are especially useful when working with assignment lists and function lists.

Table 1.4 Functions for Assignment Lists and Function Lists

Function	Definition
Eval List(list **)**	Returns a list where every item is evaluated.
Eval(::x **)**	Eval replaces ::x with its values. Often, this function is applied to a list of column names or references to be used in a command. Here is an example: Bivariate**(**Y**(** eval**(**yList**))),** X(eval(xList)))

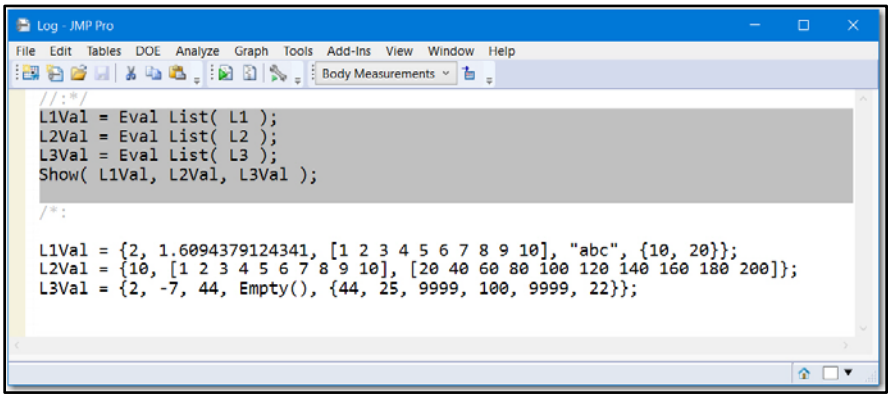
In the following examples, L2 is an assignment list, and L3 is a function list. JMP enables you to reference items in these lists by their "names". For example, L2["x"] is 10. L2[1] is the expression x=10. If you are saying to yourself, "that's not something I'd likely use," put on the brakes. Keep in mind that a **New Window( )** using <<Return Results returns a list of user responses in an assignment list. The script in this section includes simple examples using **New Window( )**. We are not quite done with this topic. There's more territory to cover regarding lists, expressions, and getting user input. But, these few functions should provide you with enough to get started.

For the examples, let:

```
L1 = { 1 + 1, Log( 5 ), 1 :: 10, "abc", {10, 20} };  // general list
L2 = { x = 10, y = 1 :: 10, z = 20 * y };            //assignment list

//h function returns value with largest magnitude ignoring sign
h = Function( {x, y}, If( Maximum( x, y ) < 0, Minimum( x, y ), Maximum( x, y ) ) );

//g function returns an Empty() if value is +/-9999,missing value code
g = Function( {x}, If( Abs( Abs( x ) - 9999 ) < .1, Empty(), x ) );

L3 = {h( 2, -3 ), h( -7, -3 ), g( 44 ), g( -9999 ),
   Abs( {44, 25, 9999, -100, -9999, 22} )}; //L3 is a function list

L1Val = Eval List( L1 );
L2Val = Eval List( L2 );
L3Val = Eval List( L3 );
```

Figure 1.19 is an excerpt from the Log window after evaluating each **Eval List** statement.

Figure 1.19 Log Window Results—Eval List



A Few Items to Note

- Almost every computer program or language includes data structures like lists, vectors, and matrices. Data structures enable the efficient organization of information, and they are key components for managing large data files and complex computations. The script in this section introduces the basic syntax for data structures.
- Vectors and matrices store numeric data only. Lists can store numbers, expressions, strings, other lists, and more.

About This Book

Purpose

The title that we selected, *JSL Companion: Applications of the JMP Scripting Language*, is intended to be representative of what we want to achieve. Learning is a journey, and a good travel companion can make a journey more engaging and less arduous. A good travel companion helps with navigating a route, pointing out not only features of interest, but also how to avoid wrong turns into treacherous terrain. Our purpose is to provide support and resources for people interested in applying JSL.

Is This Book for You?

There are great resources for JMP scripting. This book is intended to supplement them and reach a broad spectrum of JMP and JSL users. If you are a novice just learning JSL, if you are somewhat familiar with scripting, or if you want to advance your scripting skills, this book might prove useful. There are many example scripts and applications included with the text that can be used as reference or as a building block for your own applications. This second edition tries to retain usefulness for novice scripters while extending some of the more advanced topics.

Prerequisites

This book assumes that you have access to JMP and that you are comfortable using JMP interactively, including facility with data tables and the menu structure, and at least some basic familiarity with table manipulation and analysis platforms.

Scope of This Book

This book has characteristics of learning a language by immersion: we use JSL scripts to teach JSL scripting. Topics in each chapter include example scripts. There are over 200 example scripts that detail, highlight, and extend the topics of each chapter. The example scripts might prove useful in themselves, and if you copy portions of these example scripts and use them immediately and productively, even before understanding them completely, then this immersion approach has been successful. Please try to understand the example scripts, but our experience suggests that after JSL novices produce their first useful script, the added confidence is accompanied by an eagerness to read and learn more about JSL.

JMP extends and improves their software's capability regularly. The first edition of this book was written to illustrate JMP 9; this second edition uses JMP 13. There are big differences between four versions of any software, and since we don't have the burden of being comprehensive in our documentation, we can and did focus only on the techniques and capabilities we feel are most important. **New SQL Query**, enhancements to **Graph Builder**, new and enhanced **Display Box** features, the **Application Builder,** and numerous examples of using **Xpath** and **Find** for display navigation are just a few of the new items in JMP 13 that we include in this second edition.

How This Book Is Organized

The book is divided into 11 chapters with multiple sections in each. The first chapter is introductory and intended for a JSL novice. It covers some of the basics and assists with getting started. Chapters 2 through 5 are what we consider the building blocks of JSL: input and output, working with data tables, script-writing essentials, and JMP data structures. Chapters 6 through 8 are the core for building an application: creating reports, communicating with users, and custom displays. Chapter 9 focuses on flexible scripting, Chapter 10 details building and deploying applications, and Chapter 11 provides some helpful hints for planning your scripts, debugging, and improving performance.

Throughout the book, there are snippets of the example scripts included in the text. Most of the scripts include much more information than what is included in the text. In this edition, we have added a list of the scripts at the end of the book just before the Index. We believe that the scripts contain useful notes, tips, and examples and want to make them easy to find. We tried to give them meaningful names so that a quick glance through the list might direct a scripter to an example that will help accomplish a desired task.

Typographical Conventions Used in This Book

Text is used for text.

Bold is used for JMP functions, commands, and any JMP interface item used in an example or figure directly associated with the text around it.

Not Bold is used for variables, filenames, and any JMP interface item in general.

Software Used for This Book

JMP 13 for Windows was used exclusively for the scripts and as the target of all the screen captures in this book. Specifically, the scripts were written and tested in JMP Pro 13.2.1.

How to Use This Book

Navigating

If you are "brand new" to scripting, then Chapter 1, "Getting Started with JSL," is the place to begin. If you already have some familiarity with scripting, then that chapter can probably be skimmed without missing too much. However, if you have not worked with lists in JSL, then reviewing the section about lists in the first chapter might be beneficial.

The chapters in this book, and even the sections within a chapter, are not really intended to be strictly read sequentially. From personal experience, we realize that few people will read a book like this from cover to cover. For the majority of readers, it will function as a reference book, and it will be used when a new project is started or when an existing script needs extending or debugging. Our advice is to read through salient sections as you need help on a topic, and browse through the others when you can spare some time. However, please be aware that some advanced topics later in the book build on a foundation formed earlier in the book. JSL is a rich, extensive language, and we find we learn something new almost every time we script. Hopefully, you will too.

Running the Example Scripts

We don't think we can emphasize this strongly enough: Download, run, and understand the Example Scripts! We think that significantly less than 50% of the value of this book lies in these pages. The example scripts contain good and bad examples, many comments, suggestions, bons mots, and much more information than we could pack into this book.

To easily run the sample scripts downloaded from the *JSL Companion* website, extract the zipped file of scripts and data to the directory of your choosing.

Log in to JMP, and run the script 0_CreatePathVariables.jsl. The script prompts you to browse to the path where you extracted the scripts and data.

This script creates a JMP path variable named JSL_Companion. The script includes two commands to test that the path has been set correctly and to provide the syntax for using it. Many of our example scripts include Open and Write statements using this path variable.

```
//once this is run you specify this path variable with a leading $
Open("$JSL_Companion/2_ReadData.jsl");
Open("$JSL_Companion/Deli Items.jmp");
```

Ensure the preference JSL Scripts should be run only, not opened… is disabled. Go to **File ► Preferences ► Windows Specific** and ensure that this preference is unchecked. Most of our scripts assume the default platform preferences are enabled (for example, Show Points). If your results do not match the results displayed in this book, first check your Platform Preferences.

PC Versus Mac

Because this book was written using JMP on a Windows operating system, when shortcut keys are used or referenced, we assume that you are using JMP for Windows. JMP provides a handy *Quick Reference* guide, available by selecting **Help ► Books ► Quick Reference**. It includes a long list of shortcut keys for Windows and Macintosh. It is worth a look, regardless of your operating system.

References

Dickens, Charles. 1997. *Our Mutual Friend*. London: Penguin Books..

Morgan, Joseph. (2015, April 14). "Expression Handling Functions: Part I - Unraveling the Expr( ), NameExpr( ), Eval( ), … Conundrum." JMPer Cable [Blog post]. Retrieved from https://community.jmp.com/t5/JMPer-Cable/Expression-Handling-Functions-Part-I-Unraveling-the-Expr/ba-p/28963

Murphrey, Wendy., and Rosemary. Lucas. 2009. *Jump into JMP Scripting*. Cary, NC: SAS Institute Inc.

NASA Langley Research Center, Atmospheric Science Data Center, ISCCP. Percent Cloud Cover Data. Available at http://isccp.giss.nasa.gov/products/products.html.

SAS Institute Inc. 2017. *JMP® 13 Documentation Library*. Cary, NC: SAS Institute Inc. [SAS Library Note: there isn't a formal document titled *JMP® 13 Documentation Library.* However, there is a website of JMP documentation that includes JMP 13 documents: https://www.jmp.com/en_us/support/jmp-documentation.html]

Trigg, D.W., and A.G. Leach 1967. "Exponential Smoothing with an Adaptive Response Rate." *Operational Research Quarterly*; Vol. 18 No. 1 (March)., pp. 53-59.

Weiss, Edmond. H. 1985. *How to Write a Usable User Manual*, Philadelphia: ISI Press.

About These Authors

Theresa L. Utlaut is a statistician at Intel Corporation, where she is responsible for providing statistical training, consulting, and support in the development and effective use of integrated silicon technologies. A user and teacher of JMP since the late 1990s, she received her B.S. in mathematics from the University of Portland and her M.S. and Ph.D., both in statistics, from Oregon State University. She is a member of the American Statistical Association and a senior member of the American Society for Quality.

Georgia Z. Morgan is a statistician who retired from Intel Corporation after 29 years of support for semiconductor manufacturing and research. For seven years prior, she worked in the nuclear industry and university. Georgia has used and taught JMP since 1996 and JSL since 2000. A member of the American Statistical Association, she received a B.A. and an M.S., both in mathematics, from the University of Nebraska at Omaha and an M.S. in statistics from Montana State University.

Kevin C. Anderson is the president, owner, and principal statistician of KCA Consulting, LLC. He worked for Intel Corporation, Motorola, Sperry, and Texas Instruments as a statistician and scientist in the semiconductor manufacturing industry since 1977. A SAS and JMP user for more than 27 years, he received a B.S. in chemistry from the University of Tulsa.

Learn more about these authors by visiting their author pages, where you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more:
http://support.sas.com/utlaut
http://support.sas.com/morgan
http://support.sas.com/anderson

Index

E

F

G

H

I

J

L

M

N

O

P

T