Automate yourself with MEL

If you are really showing symptoms of being a rigger when you grow up – you will want to learn some scripting. The whole beginning of Rig it Right is a repeatable pattern that you will see over and over again throughout your career. Wouldn't you rather do that in the push of a button and leave the real rigging time to challenges not repetition? There are plenty of auto-rigging scripts out there. They don't always work for what you are doing though. Being able to rip apart a script to use or adapt what you require will speed up your pipeline.
To be honest, I couldn't teach myself scripting at first. I started reading lots of code and couldn't quite fathom out the logic. I could get bits and pieces but couldn't push through to full understanding. I had to take a few professor led classes to get it in my head. The funny thing about that is that the "professor led classes" were online, so I never *heard* anyone lecture on it. Nothing was different in the material; it was still a book and exercises. What I did have was a grade and the onus of having to pay for the class myself if I made under a "B –"[s1] so with a book, homework, quizzes and amazing feedback from my professors – I broke through the barrier I had placed in front of myself all those years and learned to script; learned to program; learned to code. There are plenty of places to get information on how to script and program. Let's talk about the levels of understanding one on the journey to learning code could go through.

1. **Cut and paste scripting**: one who can get a simple script to work by cutting and pasting sections from other people's code. (If you are in my class – properly credit who you honored to cut and paste from.) Obviously, there are different levels. There is the noob who can take a script and tweak a few lines to get it working for their needs to the experienced scripter who can create a recipe of their own by taking a pinch and a dash from many other scripts.
2. **Full Scripting**: one who can start with pseudo code and builds a script up from scratch. This person can even cut and paste snippets from others code or their own code. As one tops this level they make a script more flexible by eliminating hard coded elements. (We'll cover that.)
3. **User Interface**: adding a user interface to the script.
4. **Optimized Scripting**: a scripter who is cognizant of what slows a script down including error checking.
5. **Programming**: creator of large scale programs that go beyond thousands of lines of code. This requires a top level of understanding code methodologies, patterns and how to make the most efficient code for the platform on which it will run. Look to real-time game engine programmers for what I think is the highest level of that, at least in my experience of dealing with programmers. Oh heck, what about those who program medical devices or anything that life depends upon. They have to be efficient and perfect –eek.

In the purest sense of the word you should know that scripting is not programming. Scripts are a text file that you can read. Programs involve writing text files that get compiled into a binary

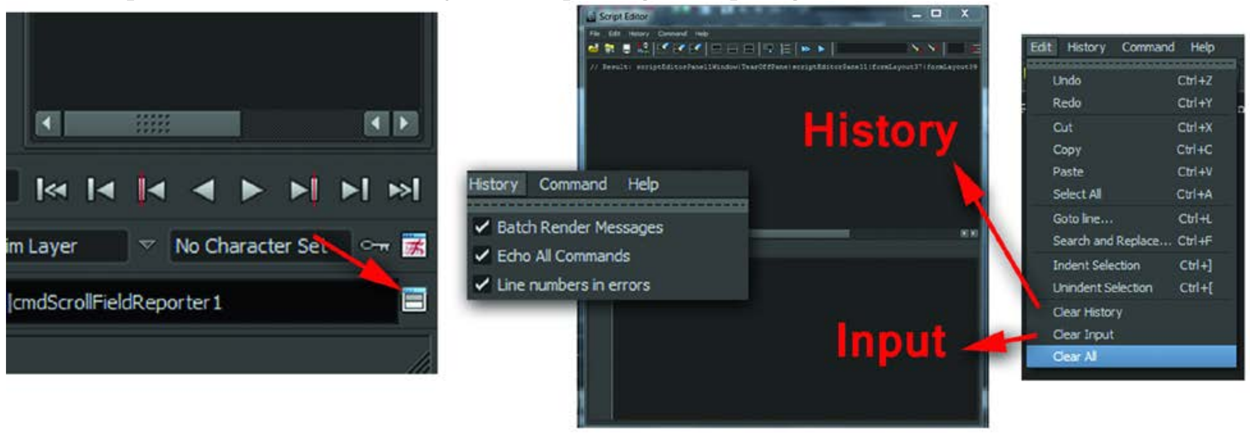
form which you can not read. Programs, or executables, run faster (if written properly) and "hook" directly into a program via API architecture. API stands for application programming interface. Maya actually has an API for C++, a programming language. In fact, remember in chapter one when we learned about nodes? Maya is built on nodes. If you were inclined and armed with knowledge of C++, you could build your own "node." What could you do with that? Anything. Photoshop has the same ability, by the way - an API for C++. Oh, that's super geeky fun there. I wish I had a clone who could play with all of that and report back. One of the strongest things many industry standard packages allow is for scripting and programming customization. They can't please everyone and every pipeline, so allowing the customer to customize and proprietary-ize their use of the software is excellent.
But, where do you get started? Let's jump in with a script we used in the beginning of the book. In chapter six we learned how to use an offset group node to allow us to offset a controller's rotational axis. It is useful to create offset group nodes quite often, so I decided to write a quick script to simplify this task. This script is utilized in the book after chapter nine. How would you go about writing such a thing? Let's learn.

First Step: Fake It. Before you can write a script, you have to know what you want to do. First I started with fake code to help outline the steps I would normally take to do this task. It usually looks like steps from a tutorial.
Pseudo Code:

```
    //Get controller and joint names (driver/driven)

    //Create empty group node

    //move group node to joint with point constraint
    //delete point constraint

    //get rotational numbers from joint with orient constraint
    //delete orient constraint

    //make controller child of group node

    //rename group node

    //freeze transformations on the controller

    //done
```

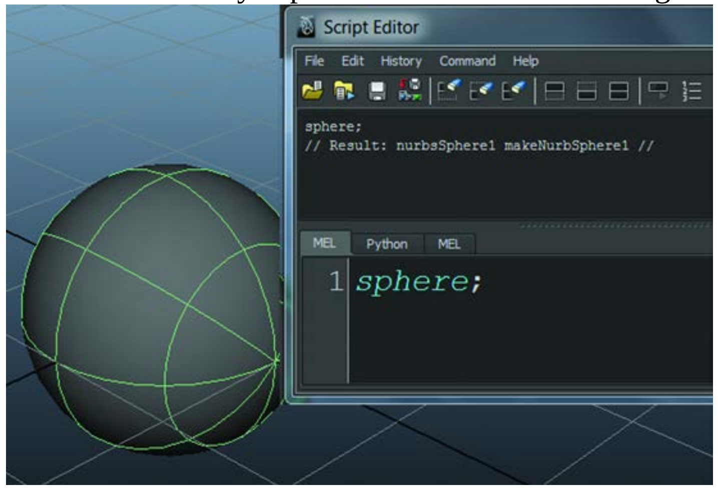
The use of the // is very helpful. In MEL, those two backslashes mean that the line written behind them is *not* code, so it will be ignored. It only works for one line of writing, if there is a carriage return, the next line will be assumed to be code. This is also called "commenting". A well commented script is a script anyone can read and learn from. I urge you to always comment in your scripts, over comment; it will help you remember months later what you were thinking. If someone takes your script and needs to adjust it they will thank you for adding in comments.

Second Step: cheat off of Maya. You can learn a lot by watching Maya call out the MEL commands when you are working in the interface. Here's a good way to start – open the script editor by clicking on the script editor button at the bottom right of the Maya interface. In that window select **History > Echo All Commands.** Click around in the interface and see that everything you do is a MEL command. Also turn on **History > Line Numbers** in errors; that will be super useful later on when you start putting a script together.



[INSERT FigAY-1.tif]FigAY-1 The script editor window.

The top of the window is the history area. This is where you see what Maya is calling out. The bottom of the window area is the input area. This is where you call the shots. Let's try it. Type in: *sphere* How do you hit enter? If you use the enter on the keyboard you'll get a carriage return. However, if you hit **enter** on the **numeric keypad** – the command will be entered. On a laptop, I do not have a numeric keypad. In this case, select the word with the mouse and hit **cntrl + enter**. I tend to do this a lot since I have many, many commands listed in my input window at any given time and only use a few of them at a time. If you select the line and use cntrl + enter, the command stays in the input window. If you don't, the command disappears.
Now you see that you can tell Maya to do things via command line. Take a look in FigAY-2; once the sphere is created, Maya "returns" information. Notice that it is the name of the transform and the input node making the sphere. You know from the beginning of this book that when delete history is preformed that second node gets removed.

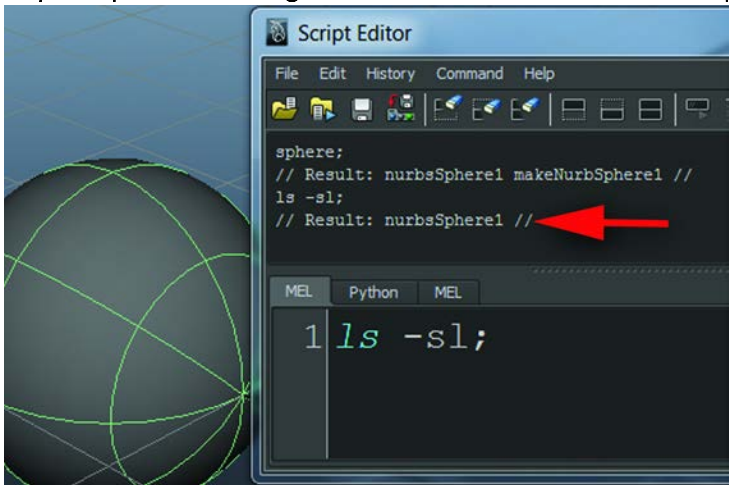


```
sphere;
// Result: nurbsSphere1 makeNurbSphere1 //
```

```
sphere;
```

[INSERT FigAY-2.tif]FigAY-2 Your first MEL command a NURBS sphere.

Step Three: asking a question. You can ask Maya questions too. This is called *querying*. For example, a very common question to ask is "What is selected?" In MEL this is asked by typing: *ls –sl* Go ahead, give it a try. The **command** is "ls", which is short for list and the flag that comes after it, "-sl", stands for selection. They are separated by a space. Think of the space like taking a breath. If the space wasn't there, Maya would think the command was ls-sl, and would be very confused. After typing in ls –sl (correctly) and hitting enter (correctly), Maya returns to you an answer. Different commands and flags return different answers. You can see the answer by looking in the script window for *"// Result:"* Whatever is listed after the colon is the answer to your question. In FigAY-3 it returns the name of the sphere we just creates: nurbsSphere1.



```
sphere;
// Result: nurbsSphere1 makeNurbSphere1 //
ls -sl;
// Result: nurbsSphere1 //
```

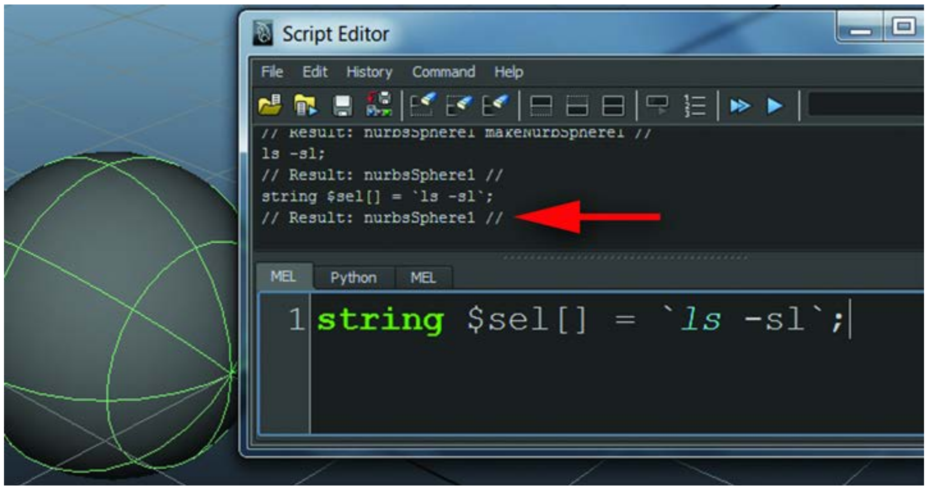
```
ls -sl;
```

[INSERT FigAY-3.tif]FigAY-3 Getting a straight answer from Maya

Step Four: storing the answer. You probably asked the question because you want to do something with the answer. You'll need to take that answer and hold on to it. We hold the answers in something called a **variable**. I always think of variables like buckets you put on the counter to hold screws, nails, milk, shoes whatever you want to put in it. There are different types of variables, called **data types**. The reason for this is when you are holding things in the variables – it is physically residing in the memory of the computer. You have to tell it how big the data is going to be, so it can set up the right sized bucket. What type of data type does your command return to you? What type of bucket do you need? Open the MEL command reference (found under **Help** > **MEL Command Reference**) and click on "ls". (*Did you see all of those commands?*) Let's focus on ls. Scroll down just a little to see the "return value". It states: return value = string[]. So, we need to set up a variable that can hold strings. Strings are text. The square brackets "[]" mean that it can be a whole list of strings. That list is called an array. So in tech speech it is a string array. My most used line of code:

```
string $sel[] = `ls -sl`;
```

The left side of this piece of code creates a string variable called sel. The $ is MEL for "this is the name of the variable", get your bucket ready and call it sel. The right side of the equals sign is the command you already know. The special little tick marks around the command mean "execute this first" in MEL speak. The equals sign takes the output of the right hands side and puts it in the bucket or the variable named on the left hand side. So in layman's term this code means: "Create a string array named sel and fill it with the answer from 'What is selected?'



```
// Result: nurbsSphere1 makeNurbSphere1 //
ls -sl;
// Result: nurbsSphere1 //
string $sel[] = `ls -sl`;
// Result: nurbsSphere1 //
```

```
string $sel[] = `ls -sl`;
```

[INSERT FigAY-4.tif]FigAY-4 Keeping Maya's answer in a variable

Let me save you an hour that I spent the first time I saw those tick marks. Those are not apostrophes. Those are back ticks, found under the tilde, to the left of the one key. See it? Back ticks. Any command surrounded by back-ticks will get executed.

Step Five: doing something with a variable.
Let's say you had two spheres selected and ran the above question: *string $sel[] = `ls -sl`;*
It would store the answer (the names of the spheres) in the variable $sel. Let's say Bob and Fred where the sphere's names. An array is a numbered list where the numbers start at 0. So, to get to the first item (Bob) in the array is *$sel[0]*. The second item (Fred) in the array is *$sel[1]*. What do you think this command will do?
parent $sel[0] $sel[1];

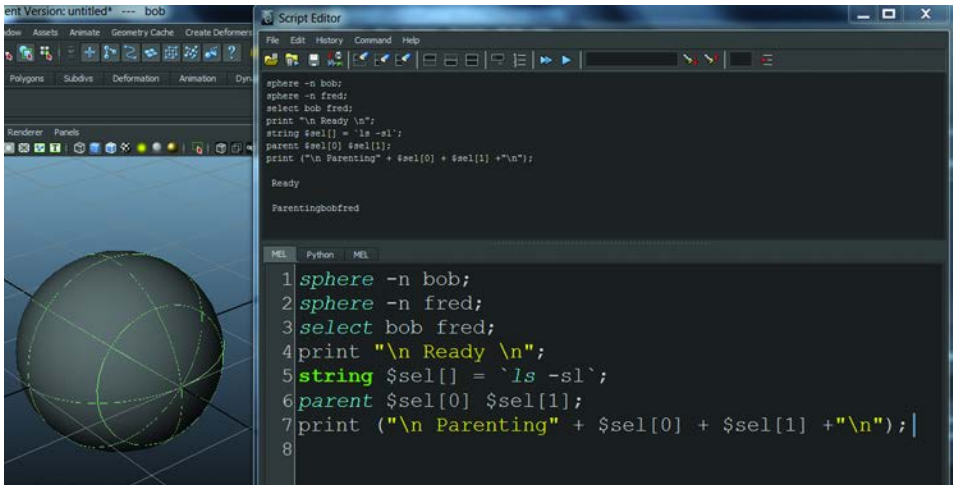
I hope you guessed make Fred a parent of Bob. There are no spaces between the variable name $sel and the square brackets, by the way. That's another 30 minutes of my life trying to figure out what I had typed wrong.

Step Six: print – the poor man debugger. I use the print command for everything. I'll print comments back to the user. When testing I'll put test print statements so I know what a variable was set to. I'll print smart aleck things just to be silly. Printing statements in the command line is very important. You can print pure text by typing: *print " my message goes here";* Or you can print the contents of a variable by typing: *print $sel* .That will print all of the contents of the array. To print just one you can type: *print $sel[0]* The tricky part is how to print text **<u>and</u>** a variable. *print ("myMessage" + $sel + "Moremessage");* To print a combination of strings you have to put it all in parentheses and use plus signs. I always have to look up that syntax to remember. Syntax is what you call the grammar rules of the scripting/programming language.

Step Seven: break it into itty bitty chunks
Try little things at a time until you get the hang of what you are doing. Do one line of code at a time. This will help you get through the dreaded syntax error phase. Inevitably you are going to type something incorrectly or incoherently for Maya and it will complain. It is best to try to find

those errors one line at a time instead of writing many lines and trying to figure out where the error is in all of that. Let's take what we know so far, I'll add in just a little bit – see if you can see what it does. Create a new scene and input the lines below into the script editor; run them. Note that each line is ended with a semicolon. This is scripting/programming langue for end of statement, kind of like a period in our own language.

```
sphere -n bob;
sphere -n fred;
select bob fred;
print "\n Ready \n";
string $sel[] = `ls -sl`;
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel[0] + $sel[1] +"\n");
```



```
sphere -n bob;
sphere -n fred;
select bob fred;
print "\n Ready \n";
string $sel[] = `ls -sl`;
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel[0] + $sel[1] +"\n");

 Ready

 Parentingbobfred
```

```
sphere -n bob;
sphere -n fred;
select bob fred;
print "\n Ready \n";
string $sel[] = `ls -sl`;
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel[0] + $sel[1] +"\n");
```

[INSERT FigAY-5.tif]FigAY-5 Script editor with example script executed.

Well, now – aren't you doing some fun stuff?

We created a sphere named bob; then, created a sphere named fred. We selected bob and fred. We printed that the script was ready. We asked what was selected and put that answer in a variable called $sel. We made bob (the first item on the list) a child of fred (the second item on the list). We printed that we parented the first item on the list and the second item on the list. The special characters I put in the print line "\n" means carriage return. That wasn't so bad at all, was it? Experiment with the print statements to add in spaces so that the printed "Parentingbobfred" line is more legible.

Obviously, there is way more to learn. But let's push through with those building blocks and make something happen.

From our pseudo code that we wrote at the beginning of this chapter – let's see how we can flesh that out into a script. I created an offset controller manually and took at the script editor as I did it, taking note of the MEL commands that occurred when I selected things, grouped things and parented things. Then, I opened up the MEL command reference documentation to see what I could understand about each of the commands that was getting called. I was able to piece together a workable bit of code using everything you know right now. Let’s take a look:

Working code:

```
//Goal: hook up a controller to match the joint's rotation

//Usage (what the user should do before running): Select the controller and then
the joint

//Get names of the selected controller and joint (0 driver / 1 driven)
string $sel[] = `ls -sl`;

//get controller name and create offset name by adding a _os_GRP at the end
string $grp = ($sel[0] + "_os_GRP " );

//Create empty group node, name it
select -cl;
group -em; xform -os -piv 0 0 0;
rename $grp;

//move group node to joint with point constraint
//delete point constraint
select $sel[1];
select -add $grp;
string $temp2Delete[] = `pointConstraint -offset 0 0 0 -weight 1`;
delete $temp2Delete;

//get rotational numbers from joint with orient constraint
//delete orient constraint
select $sel[1];
select -add $grp;
string $temp2Delete[] = `orientConstraint -offset 0 0 0 -weight 1`;
delete $temp2Delete;

//make controller child of group node
parent $sel[0] $grp;

//freeze transformations on the controller
select $sel[0];
makeIdentity -apply true -t 1 -r 1 -s 1 -n 0;

//done
```

Let’s go through line by line:

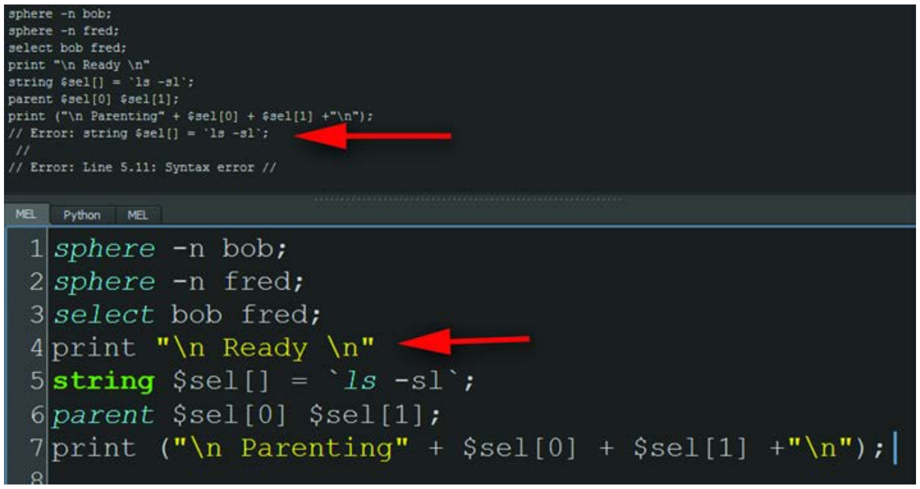
string $sel[] = `ls -sl`;	You know this one – take the answer to “What is selected” and put it in a variable called *sel*.
string $grp = ($sel[0] + "_os_GRP ");	Get the name from the first item selected, which should be the

	controller, add "_os_GRP" at the end and save that in a variable called *grp*.
select -cl;	Clear the selections.
group -em; xform -os -piv 0 0 0;	Create an empty group node using the default command.
rename $grp;	Give the current item selected (which is the newly made group) the name we created and stuffed in the variable called *grp*.
select $sel[1];	Select the second item on the list (the joint).
select -add $grp;	Also, select the newly made group.
string $temp2Delete[] = `pointConstraint -offset 0 0 0 -weight 1`;	Create a point constraint, with no offsets to move the group node to the joint. Keep the name of the point constraint in a temporary variable called *temp2Delete* so that we can delete it.
delete $temp2Delete;	The foretold deletion of the point constraint.
select $sel[1];	Select the joint again as the "driver".
select -add $grp;	Select the empty group node as the "driven".
string $temp2Delete[] = `orientConstraint -offset 0 0 0 -weight 1`;	Create an orient constraint with no offset from the joint to the group node. Store the name of this in our variable aptly named *temp2Delete*. (Always use logical variable names.)
delete $temp2Delete;	Deletion of the temporary orient constraint that was only needed to pump rotational numbers from the joint into the group node.
parent $sel[0] $grp;	Makes the group node a parent of the controller.
select $sel[0];	Selects only the controller.
makeIdentity -apply true -t 1 -r 1 -s 1 -n 0;	Freeze transformation on the controller.

OK, hold the phone. Don't go any further until you understand that. If you are a little wobbly on it and have not hit that "OOooooh, I get it" moment; don't keep going. Input those commands one at a time. Type them longhand, get those syntax errors out of your system now. You have to go through the "A semi-colon, space, capitol letter, dollar sign, parenthesis, quotation mark - will be the death of me" phase.

Step Eight: the joy of syntax errors. There really is no joy in syntax errors and the error messages that Maya throws are helpful most of the time, but sometimes they can be misleading. Let's take a look at some common things that Maya complains about.

1. A statement with a missing semi-colon will show up as an error in the next line because it is trying to read the both lines together as one command.

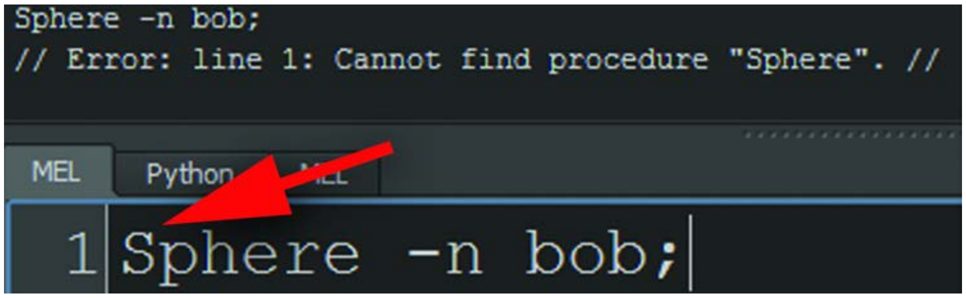


```
sphere -n bob;
sphere -n fred;
select bob fred;
print "\n Ready \n"
string $sel[] = `ls -sl`;
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel[0] + $sel[1] +"\n");
// Error: string $sel[] = `ls -sl`;
 //
// Error: Line 5.11: Syntax error //
```

```
sphere -n bob;
sphere -n fred;
select bob fred;
print "\n Ready \n"
string $sel[] = `ls -sl`;
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel[0] + $sel[1] +"\n");
```

[INSERT FigAY-6.tif]FigAY-6 Missing semicolon causes misleading error message.

2. Capitalization really matters. A command “Sphere” is very different from a command “sphere”.

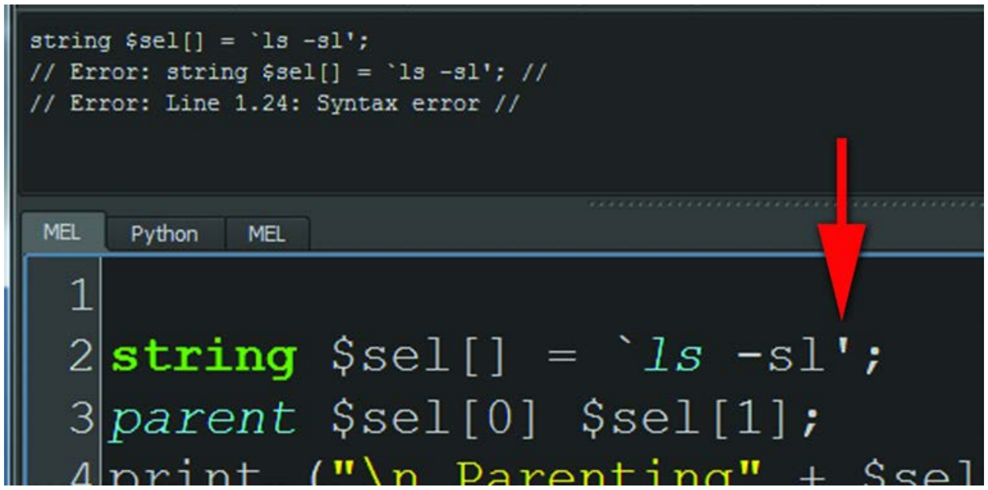


```
Sphere -n bob;
// Error: line 1: Cannot find procedure "Sphere". //
```

```
Sphere -n bob;
```

[INSERT FigAY-7.tif]FigAY-7 Capitols are important.

3. You typed something wrong, but you have to figure out what it was. For example, do you see the error in FigAY-8? There is an apostrophe instead of a back-tick.



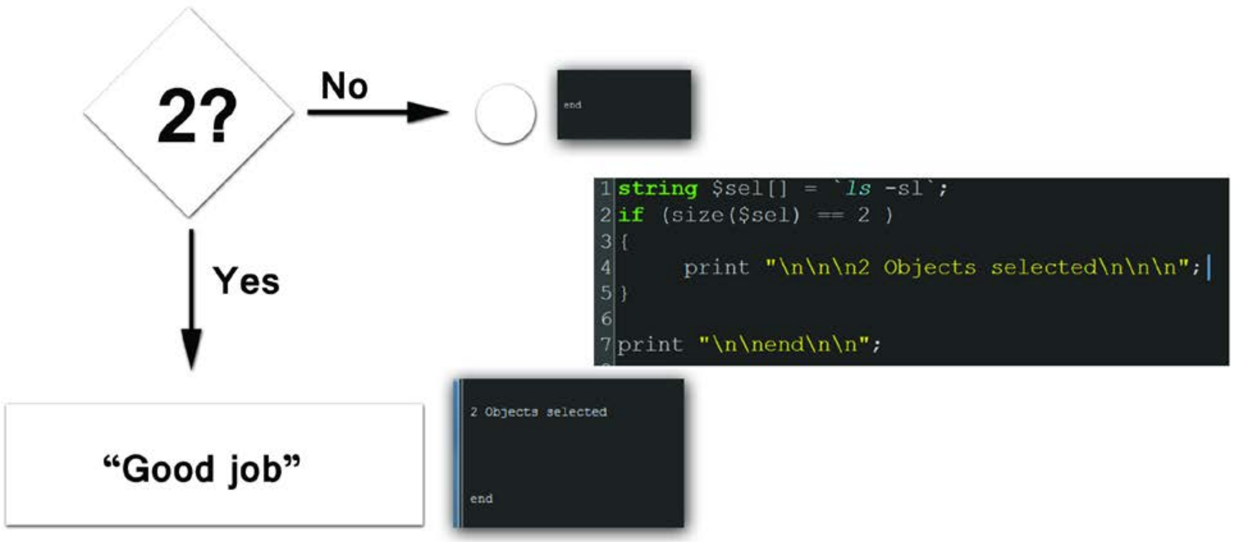
```
string $sel[] = `ls -sl';
// Error: string $sel[] = `ls -sl'; //
// Error: Line 1.24: Syntax error //
```

```

string $sel[] = `ls -sl';
parent $sel[0] $sel[1];
print ("\n Parenting" + $sel
```

[INSERT FigAY-8.tif]FigAY-8 Punctuation is important.

Step Nine: Flow control. To get the next step, you’ll need to learn a little bit about **flow control** of scripts. What is flow control? A script isn’t just an automated set of directions; there should be choices in there, some yes and no answers. Every scripting language has a different set of flow controls that you can use. Let’s look at MEL’s. I will use flowcharts; they teach them in Jr. High School now. They give me a visual look at how things work, as an artist, I find this extremely helpful.

If statement: this lets you ask a question. Does the user have two things selected? If they do continue with the script, print hooray; if they don’t, do nothing.



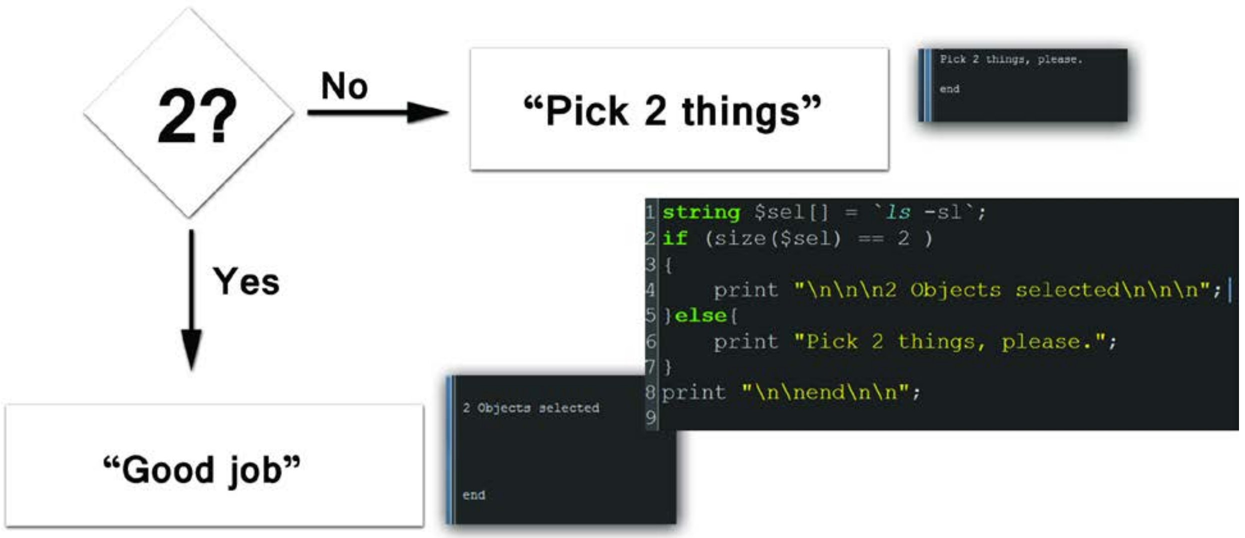
```
end
```

```
string $sel[] = `ls -sl`;
if (size($sel) == 2 )
{
      print "\n\n\n2 Objects selected\n\n\n";
}

print "\n\nend\n\n";
```

```
2 Objects selected

end
```

[INSERT FigAY-9.tif]FigAY-9 If statement

If else statement: this asks a question and gives a way to deal with the no. If they don’t have anything selected do something, instruct them to select something.



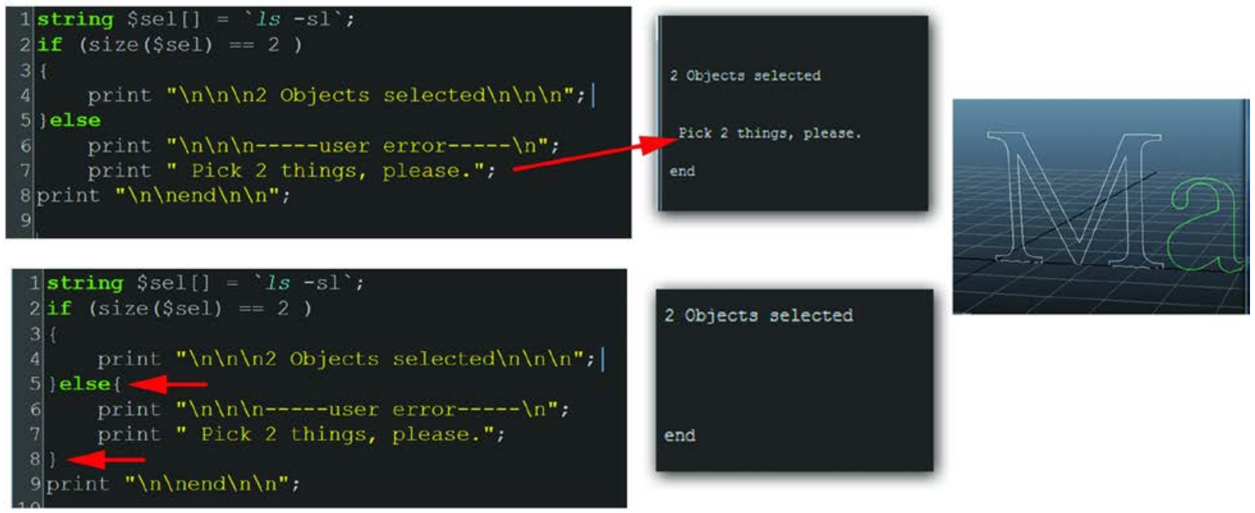
```
Pick 2 things, please.
end
```

```
string $sel[] = `ls -sl`;
if (size($sel) == 2 )
{
    print "\n\n\n2 Objects selected\n\n\n";
}else{
    print "Pick 2 things, please.";
}
print "\n\nend\n\n";
```

```
2 Objects selected

end
```

[INSERT FigAY-10.tif]FigAY-10 If else statement

You'll note the use of curly brackets and indentation to help the script be legible. Any innards to a flow control statement should be indented so that you can see it is different than the main script. The curly brackets actually aren't needed if you only have one thing to do after the if statement; for example, one print statement. In my purest opinion, that is sloppy and can work you right into an unintended error when you add a second statement after the if statement that is meant to only work if the if statements is true. However, if you don't put in curly brackets – it isn't treated as such. So, rule to follow – use the curly brackets. I tip my hat to my favorite programming professors who graded us on how beautiful the "white space" (indentations), legibility and commenting of our scripts.



```
string $sel[] = `ls -sl`;
if (size($sel) == 2 )
{
    print "\n\n\n2 Objects selected\n\n\n";
}else
    print "\n\n\n-----user error-----\n";
    print " Pick 2 things, please.";
print "\n\nend\n\n";
```

```
2 Objects selected

 Pick 2 things, please.
end
```

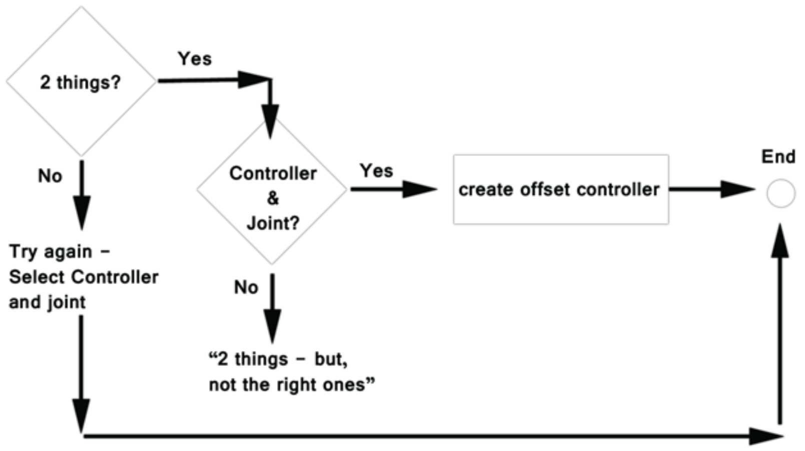
```
string $sel[] = `ls -sl`;
if (size($sel) == 2 )
{
    print "\n\n\n2 Objects selected\n\n\n";
}else{
    print "\n\n\n-----user error-----\n";
    print " Pick 2 things, please.";
}
print "\n\nend\n\n";
```

```
2 Objects selected


end
```

[INSERT FigAY-11.tif]FigAY-11 Curly brackets

There's more to logic control, as well. But that is enough to get into our next version of the script and leaves you just a little bit to figure out. Ahh, it is a sneaky professor that leads you into asking a question.

Step Ten: user error checking. We have a working script, but, take a step back and think about what could the user do wrong? The only thing they have to do is select a controller and a joint. We can check to see if they have done that and give them information on what to do if they did it wrong. Isn't it useful to get feedback when using a script? I'm going to guess that many people do not know that often times the user directions is written in the script text file; usually at the top. A well written script can be open and read by the user. What if the user doesn't know, or doesn't want to do that? Instead, we'll give them feedback.
So, I decided to put in a little bit of checking so that the user was prompted if they didn't have the correct amount or the correct items selected.


2 things?
Yes
No
Controller & Joint?
Yes
create offset controller
End
Try again – Select Controller and joint
No
"2 things – but, not the right ones"

[INSERT FigAY-12.tif]FigAY-12 Error checking

First, let's do the error checking piece of code and leave out the part of the code for doing the offset controller.

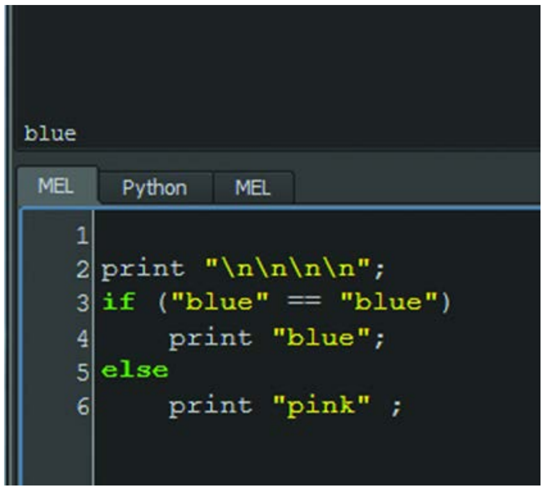
```
string $sel[] = `ls -sl`;

//are they using this correctly?
//did they select 2 things?

if (size($sel) == 2 )
{
        print "\n\n\n2 Objects selected\n\n\n";

        //ok - did they select 2 correct things?
        if (`nodeType $sel[1]` == "joint" && `nodeType $sel[0]` == "transform")
        {
                print "\n\n\nOK - these are the right things selected. Creating offset now \n\n\n ";

                //code for offset goes here

                }//end if they picked the correct 2 controllers

        else
                print "\n\n\n You have two objects selected, but not the right two objects:
                Controller and Joint\n\n\n";

        } //end if they picked 2 objects

else
        print "\n\n\n Gosh, you need to select two objects: Controller and Joint\n\n\n";

//done
```

print ("\n\n\nDONE\n\n\n");

Could you read it? Isn't it funny how some of that is starting to make sense to you? The line "*if (size($sel) == 2)*" means if two items are selected. The command *size()* returns how big something is. Adding in the variable name causes the little script called *size()* to look at the variable and give back a number stating how big it is. If it returns 2, then the user has selected two items. Everything else should read pretty easily except for that mouthful of a statement:

if (`nodeType $sel[1]` == "joint" && `nodeType $sel[0]` == "transform")

Look up the command nodeType in the MEL reference. It returns a string for what type of item is selected. OK, so that means the if statement is looking for a joint and a transform. That makes sense. But what is that "*&&*" in there? That's a **logic operator**. It means "and". This statement can be read as "If the second item selected is a joint *and* the first item is a transform node". The other question you might have is, what is "=="? There is a big difference between a "=" and a "==". The equal sign, "=", is what we use to define a variable, meaning, it puts the number, milk, screw, whatever in the memory bucket, as in the statement: *$sel = `ls –sl`* . A double equal sign "==" really does mean equals. Here's a simple if statement using the "==" in FigAY-13. You'll note the non use of curly brackets; call me lazy.



```
blue
```

```
print "\n\n\n\n";
if ("blue" == "blue")
    print "blue";
else
    print "pink" ;
```

[INSERT FigAY-13.tif]FigAY-13 The use of ==

Learning to scripting is an interesting path. The more code you read and test the less it will look like geek to you. Here is the completed script with the script placed inside:

```
//Goal: hook up a controller to match the joint's rotation

//Usage (what the user should do before running): Select the controller and then the joint

//Get names of the selected controller and joint (0 driver / 1 driven)

string $sel[] = `ls -sl`;

//are they using this correctly?
//did they select 2 things?
```

```
    if (size($sel) == 2 )
    {
            print "\n\n\n2 Objects selected\n\n\n";

            //ok - did they select 2 correct things?
            if (`nodeType $sel[1]` == "joint" && `nodeType $sel[0]` == "transform")
            {
                    print "\n\n\nOK - these are the right things selected. Creating offset now
                    \n\n\n ";

                    //get controller name and create offset name
                    string $grp = ($sel[0] + "_os_GRP " );

                    //Create empty group node, name it
                    select -cl;
                    group -em; xform -os -piv 0 0 0;
                    rename $grp;

                    //move group node to joint with point constraint
                    //delete point constraint
                    select $sel[1];
                    select -add $grp;
                    string $temp2Delete[] = `pointConstraint -offset 0 0 0 -weight 1`;
                    delete $temp2Delete;

                    //get rotational numbers from joint with orient constraint
                    //delete orient constraint
                    select $sel[1];
                    select -add $grp;
                    string $temp2Delete[] = `orientConstraint -offset 0 0 0 -weight 1`;
                    delete $temp2Delete;

                    //make controller child of group node
                    parent $sel[0] $grp;

                    //freeze transformations on the controler
                    select $sel[0];
                    makeIdentity -apply true -t 1 -r 1 -s 1 -n 0;
                    }//end if they picked the correct 2 controllers

            else
                    print "\n\n\n You have two objects selected, but not the right two objects:
                    Controller and Joint\n\n\n";

            } //end if they picked 2 objects
```

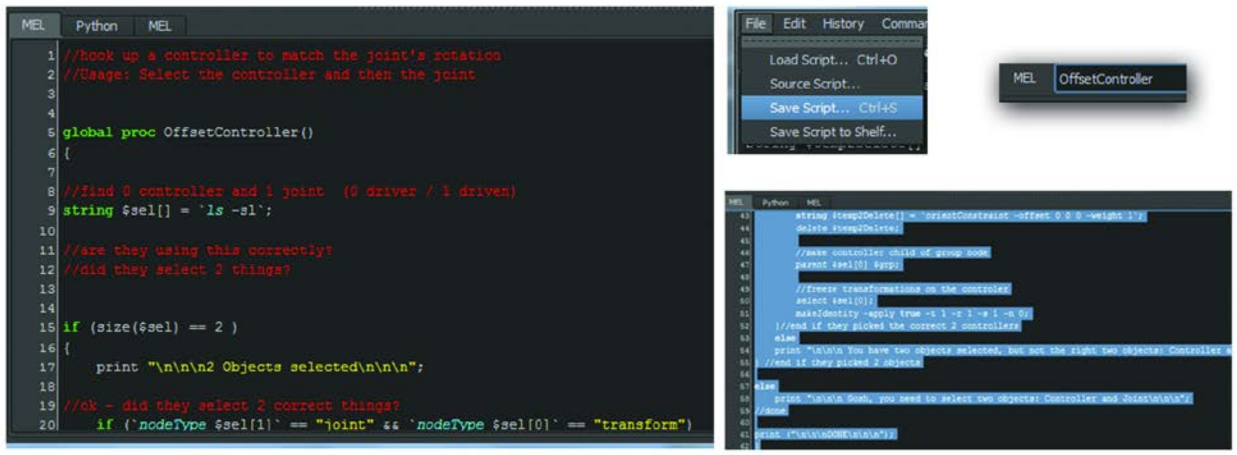
```
        else
                print "\n\n\n Gosh, you need to select two objects: Controller and Joint\n\n\n";

        //done
        print ("\n\n\nDONE\n\n\n");
```

To let this run in Maya, we need to package all of that into a procedure. This lets you give a name to your script and you can call it anytime inside of Maya via the MEL command line. To do this you add the following line at the top and bottom of the code:

```
global proc OffsetController()
{
        ///the script goes in here
}
```

Saving the script: Once the script is written out you can save it in an external mel file. There are text editors that you can write scripts in: nedit or cutter, among others. You can also write the script in the script editor. You can see in FigAY-14 that Maya provides contextual color clues to help you write the code. This can be very helpful to catch syntax errors. In Maya's script editor you can select all of the code and choose **File** > **Save Script… .** Give the file a name and make sure to put the suffix .mel at the end of the file name.

Running the script: To run the script, you can save the script in the scripts folder and restart Maya or directly source the script from the script editor by selecting the text and hitting cntrl + enter. Either way, once the script is sourced into Maya's brain you can call it as we have throughout this book by typing **OffsetController** into the command line. Now you know where that name comes from.



```
//hook up a controller to match the joint's rotation
//Usage: Select the controller and then the joint


global proc OffsetController()
{

//find 0 controller and 1 joint  (0 driver / 1 driven)
string $sel[] = `ls -sl`;

//are they using this correctly?
//did they select 2 things?


if (size($sel) == 2 )
{
    print "\n\n\n2 Objects selected\n\n\n";

//ok - did they select 2 correct things?
    if (`nodeType $sel[1]` == "joint" && `nodeType $sel[0]` == "transform")
```

```
            string $temp2Delete[] = `orientConstraint -offset 0 0 0 -weight 1`;
            delete $temp2Delete;

            //make controller child of group node
            parent $sel[0] $grp;

            //freeze transformations on the controler
            select $sel[0];
            makeIdentity -apply true -t 1 -r 1 -s 1 -n 0;
        }//end if they picked the correct 2 controllers
        else
        print "\n\n\n You have two objects selected, but not the right two objects: Controller a
} //end if they picked 2 objects

else
        print "\n\n\n Gosh, you need to select two objects: Controller and Joint\n\n\n";
//done

print ("\n\n\nDONE\n\n\n");
}
```

[INSERT FigAY-14.tif]FigAY-14 Saving and running the script

Where to go from here:

When you first start out into the land of scripting it is hard to know where to start. There is an informational smorgasbord for you, but there are no rules of how to fill your plate and what to eat first. Hopefully this chapter has taken you from concept to completion, and even to error checking in a short time so that you have a little confidence to start learning more. Let's look at some of the concepts you'll want to read up on:

1. How can I find help on commands? (Hint – look under the help menu in the script editor)
2. What other types of commenting are there in MEL?
3. What are the most useful commands in MEL?
4. We used querying – asking a question. Can you do other things like use a command to edit something?
5. What other data types are there that we can store variables in?
6. When you create a variable – does it matter where the variable is placed in the script? (Hint – look up "scope".)
7. What other methods of flow control are there in MEL?
8. What other logic operators are there?

That's a great start. MEL is just in Maya, but the concepts extend to all sorts of other scripting languages. Python, you might have noticed is also present in Maya. Python is a great scripting language. It extends into a full programming language and has been around for a good many years. It is prevalent in quite a few off the shelf software packages as well as some industry proprietary software packages. It is also a pretty easy language to learn. I say that, but we won't cover it here because I haven't used it yet in Maya. There are some great books, Digital Tutor tutorials and lots of avenues to start to learn that language as well. I've got by fine with MEL and not needed to do anything in Python, even though I've used Python as a stand-alone language for years. Adding that functionality to Maya has allowed it to extend its uses in the industry where those companies that use Python have a wealth of proprietary Python applications that can now be used with Maya.

Extra Effort: there is always something more to learn. While I was trolling through help documentation I noticed the mel command **scriptCtx**. I wonder if that could have been used for this script? Hmmmm.