Maya Scripting for Noobs

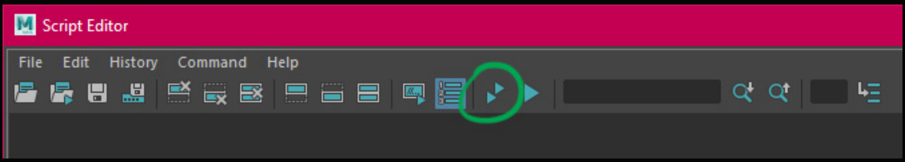
by Andy Latham, age 39

MEL vs Python

Pretty much every action in Maya has a MEL command. But MEL is nasty. So we use lovely Python code instead! Also if you learn Python, you can do fun things outside of Maya too!

Script Editor Basics

- Always make sure you type Python code in a Python tab rather than a MEL tab. Otherwise Maya will cry.
- You can run your code by pressing the little double play button on the toolbar. Don't use the single play button. It deletes your code and you will cry.



- You can highlight any bit of code and hit Ctrl + Enter to run just that bit. Ctrl + A (select all) followed by Ctrl + Enter is a common set of shortcuts, equivalent to pressing the double play button.

The Anatomy of a Maya Python Script

All Maya Python scripts need to start with this line:

```
import maya.cmds as cmds
```

This imports all of Maya's commands, allowing us to use them by typing things like:

```
cmds.polySphere(radius = 100)
```

Give it a go! Type both of those lines into the script editor and run them with Ctrl + A and Ctrl + Enter. Or use the double play button.

- We use a . to access anything that is a child of anything else.
- When we use commands, we pass arguments to specify details (such as the radius of the sphere we want to create). These must be placed in parentheses and separated with commas.

Finding Maya Commands

often we don't know what command we need to use. Luckily Maya will tell us the relevant command if we manually perform our task and look at what appears in the script editor. Try moving the sphere we just created and you should see something like this:

```
move -r 0 39.2 0 ;
```

Uh oh! That's MEL! Arrggghhhhh! Well no need to worry, we can translate this to Python.

Translating MEL to Python

We can see that the command we want to use is called "move". Now in Python we have to select the "move" command from the bank of Maya commands that we imported earlier:

```
cmds.move()
```

We follow the command with a set of parentheses into which we will place our relevant information. Each Maya command is something that in the coding world we call a function or method or subroutine or... well there are lots of names, but we'll stick with "function". This just means a packet of code that does something specific. Functions follow the same kind of form in most programming languages. You'll find that learning one language makes learning others MUCH easier!

However if we run that command, nothing happens! This is because we haven't told it what to move or where to move it to! This is where arguments come into play, and we choose those arguments based on the other information that the MEL script gives us. Let's look at it:

- The -r means "relative", and tells Maya to move something relative to where is currently is rather than using absolute coordinates.
- The three numbers at the end are the x, y and z values for the move.

How do I know this stuff? Well every command can be looked up by going to Help > Maya Scripting Reference > Python Command Reference. Or just Google Maya Python [command name].

Arguments with a - are called flags. To use these flags in Python, we give the same letter (or its full name), followed by = True to tell Maya that we want to use it. We also have to put the coordinates before any flags for this particular command. So our command becomes:

```
cmds.move(0, 39.2, 0, r = True)
```

Try it out! You may notice that it only works if you have the sphere selected. If we want the command to work without having the sphere selected, we need to give it the sphere's name in quotation marks:

```
cmds.move(0, 39.2, 0, “pSphere1”, r = True)
```

Phew! That was a lot of nonsense we had to go through to translate. You may think that the Python version looks more complicated than the MEL version. However a little complexity here will make everything else a lot easier later on, trust me!

Querying and Editing

Some commands can do more than perform operations on objects in your scene. Sometimes they can tell you useful information. The polySphere command we used earlier is one such example. Maybe instead of creating a new sphere, we want to find out the radius of an existing sphere. For this we use a special query flag (-q = True) to say "hey Maya, I want to know something", followed by the flag of the thing we want to know:

```
cmds.polySphere(“pSphere1”, q = True, r = True)
```

Notice that we provide the name of the object we want to know about too. Try it! You should see that the script editor tells you the radius of the sphere.

Why don't we try changing the radius? We need a different special flag for this - the edit flag (-e = True). Now this time, instead of just telling Maya that the radius is what we're interested in with r = True, we tell it the radius that we want:

```
cmds.polySphere(“pSphere1”, e = True, r = 50)
```

Give it a go! You should notice your sphere change size!

Variables

Programming/coding/scripting becomes useful when our code can remember things. For this we use variables. In some languages variables can get a bit complicated, but in Python they are nice and easy. Basically any time you want something to be remembered for later, you assign it to a variable with a = sign (called the assignment operator). You can choose any word you like for a variable, just as long as it doesn't start with a number. For example we could have a really boring variable that just remembers the number 2:

```
myBoringVariable = 2
```

Or we could have a more interesting one that remembers a word:

```
myFavouriteAnimator = “Andy”
```

A variable can hold absolutely anything though, not just numbers and words. It can hold lists of things, or objects in Maya, or things that you have invented yourself!

The thing you have to remember in all coding is that assignment happens from right to left. So whatever you have on the right of the = will get passed to whatever is on the left. Never the other way around.

From Commands to Proper Coding

We know that we can query details about an object, and we know we can change details about an object. Well how about we combine the two? Maybe we can create a sphere and a cube and we set the cube's size to match that of the sphere. Wouldn't that be exciting?!

First let's start from scratch. Remember to first import maya's commands or you're going to be scratching your head later on:

```
import maya.cmds as cmds
```

Now we can create a new sphere. However this time we will store our sphere as a variable.

```
mySweetSphere = cmds.polySphere(r = 20)
```

Storing it as a variable means that we can access the sphere nice and easily in the future. Okay so let's next create a cube and store it in another variable:

```
myCoolCube = cmds.polyCube(w = 10, h = 10, d = 10)
```

Now, let's forget that we just specified the size of the two shapes. Have you forgotten? Good.

So to match our cube size to our sphere size, we need to know the sphere size, which we currently don't know...right?...RIGHT?! So let's query the radius, and just for the hell of it let's store it as another variable. Since we have the sphere stored in our mySweetSphere variable, we will use that variable name instead of the name of the sphere itself:

```
size = cmds.polySphere(mySweetSphere, q = True, r = True)
```

Notice that we don't need to put the variable name in quotation marks. We use those any time we use a string (a silly name for a bit of text data such as an object name).

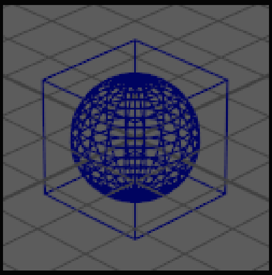
If we want our cube to be the same size as our sphere, we need to double the radius to get the diameter. We can simply multiply it by 2 and then then save the result into the original variable:

```
size = size * 2
```

So the size variable now holds a value that is double the sphere's radius...whatever that mystery number may be! Now let's use that variable to edit the size of our cube:

```
cmds.polyCube(myCoolCube, e = true, w = size, h = size, d =
size)
```

So you should have a script that's six lines long. Try running it and be blown away by what appears in the viewport!



Lists

A list is a super useful thing. In fact it's one of THE most useful things. It is a kind of variable that holds lots of things at the same time. We can define one like this:

```
shoppingList = [“cheese”, “mousetrap”, “disinfectant”]
```

Or we can create an empty list like this:

```
mySkateboardTricks = []
```

We can append things to a list with:

```
shoppingList.append(“bandaid”)
```

We can access whatever is in position 3 in the list (or index) with:

```
shoppingList[3]
```

One really useful Maya thing that we can do is automatically create a list of objects of a certain type by using the ls command:

```
objectsInMyScene = cmds.ls(type = “geometryShape”)
```

There are a huge amount of things that can be done with lists - too much to cover here.

If Conditions

We can make really cool things (nerd!!) when our code can change direction. One way we can do this is to use if and else. Take a look:

```
import maya.cmds as cmds
someVariable = 4
if someVariable < 10:
  cmds.polySphere(r = 20)
  print(“Sphere created! Whoop!”)
else:
  cmds.polyCube(w = 10, h = 10, d = 10)
  print(“Cube created! OMG Yes!”)
```

Here we create a variable which holds a value. If we give it a value that is less than 10, a sphere will be created. And if we instead give it a value of 10 or greater then a cube will be created. And in either case a message will be output to the script editor. You think that's cool too, right? Oh by the way, if you want to test if something is equal to something else use ==. Remember, a single = is the assignment operator, it does not mean "equals"!

There are a couple of important things to note about conditional things. First, you may have noticed that each condition line ends, like any good digestive system, with a colon. This just has to be there because reasons.

Second, each branching line of code needs to be indented. You can either do this with a tab or a couple of spaces. The script editor will also sometimes automatically take care of it for you. The important thing is that you are consistent in the size of your indentations. If you're not, then Python won't be happy, Maya won't work, and anyone looking at your code will laugh.

For Loops

Remember I said lists are super useful? well they are upgraded to mega useful when combined with a for loop. we can use such a thing to iterate through a list, performing actions on each element in it:

```
for item in shoppingList:
  cmds.polySphere(name = item)
```

This incredibly useful example creates a sphere for each item in shoppingList, with each sphere named after the item.

While Loops

Ready to crash Maya? You'll do that fairly regularly with while loops. Such a loop will iterate over and over again until its condition is not met, so you have to be careful that you don't give it a condition that will always be met, or your script will run forever!

```
myVariable = 0
while myVariable < 10:
  cmds.polySphere(name = item)
  myVariable = myVariable + 1
```

This piece of code sets a variable to hold the number 0. The while condition checks if the variable is less than 10. If it is, then it creates a sphere and adds 1 to the value of the variable before repeating the check. As soon as the variable has a value of 10, the loop ends and we find 10 spheres in our scene. Just be careful not to forget the last line!

Comments

You should leave helpful messages in your scripts for your future self or others to know what the hell you've done. You can do this with a # and Maya will ignore everything after it until the end of the line:

```
# This is a really interesting comment.
```

Functions

We have met functions before. The commands we use to control Maya are all functions. But we can create our own! Remember on the last page how we created 10 spheres? Well it takes a bit of head-scratching to decide what the code does at first. Wouldn't it be useful if we could just write MakeMeSomeSpheres instead? Well we can! First we have to wrap our existing code up in a function (which we call a definition) and give it that very name. And to spice things up, let's also involve a new variable called numberofSpheres. We'll make this an argument of the funtion.

```
def MakeMeSomeSpheres(numberOfSpheres):
  myVariable = 0
  while myVariable < numberOfSpheres:
    cmds.polySphere(name = item)
    myVariable = myVariable + 1
```

Notice how the number 10 has been replaced with numberofSpheres? We can use the argument to specify how many spheres to create. So now that we have our function defined, it will just sit there, dormant, until we call it, like Batman, at some future point in our script with just one line. We could create a hundred spheres:

```
MakeMeSomeSpheres(100)
```

Then maybe later on we need some more:

```
MakeMeSomeSpheres(24)
```

As well as passing information into functions, we can also get information out by using a return value. Let's write a function that squares a number - because maths is fun:

```
def SquareNumber(number):
  squaredNumber = number * number
  return squaredNumber
```

Notice the last line in which we return the squared number to wherever we call the function from. When we call the function, we then have the option of using that returned value (e.g. saving it as a variable):

```
squaredNumber = SquareNumber(8)   # Stores the number 64.
```

- A rule of thumb is if you're typing the same code twice, you should probably think about making a function.
- Although we now have the power to jump around our script like mad people, functions have to be defined before they are used, which means higher up in the file than the code that is calling them.

Variable Scope

You may notice that I used a variable called squaredNumber in the function above, and that I also used the same name outside the function. Does that mean these are the same variable? Well I wouldn't be drawing attention to it if that was the case! They are NOT the same!

Variables have something called scope. This is just a fancy way of saying they are only visible to the section of code in which they are used. So if we use a variable in a function, that variable is not visible outside of that function, and vice versa, which is why we use arguments and return values. It is possible to create global variables that are visible from anywhere in our code, but it is wise to use these sparingly because it becomes easy to accidentally change them. To use them, we type the global keyword before the variable name when we first define it:

```
global myVariable = 20
```

Then if we wish to use this variable in a function, before we use it we have to type (best at the top of the function):

```
global myVariable
```

If we don't do that, then when we use myVariable, it will be treated like a brand new local variable and we'll be all kinds of confused.

External Editing, Importing and Sharing

If you have made it this far then you are going to start getting pretty fed up with the script editor. It's not the nicest thing to use, it's easy to accidentally clear all of your code, and when Maya inevitably crashes, you'll go apeshit. So I highly recommend that you use an external editor if your code is more than a few lines long.

You can use a nice simple text editor like Notepad++ or a more fully-featured program like Visual Studio Code. These free programs do a lot of fancy things, but most importantly at this stage, they help you spot syntax errors, make code easier to look at, and they very rarely crash! But we need to know how to get code from them into Maya without just copying and pasting.

First, you want to ensure that you save your code files as .py files in your Documents\maya\scripts folder. This ensures that all versions of Maya on your computer can see the scripts you create.

Let's pretend we have created a useful animation script called AnimateMySceneForMe.py. To import this into Maya, we need to run the following commands in the script editor, or within a shelf button, or as a hotkey:

```
import AnimateMySceneForMe
reload(AnimateMySceneForMe)
```

Or if you're using Maya 2022 with its swanky Python version 3, you need these commands (just an extra line). Because "new" means more work:

```
import AnimateMySceneForMe
from importlib import reload
reload(AnimateMySceneForMe)
```

Once you have tested that your code works, you can share it with other people, just tell them where to save the script and how to import it.

Final Words

The stuff I've written here barely scratches the surface of what can be done with Python (or with MEL, if you're one of "those"), and I encourage you to learn more. As a starting point, I particularly recommend that you learn about classes as they are amazingly powerful.

Be patient though. You'll bash your head against your desk, you'll use language you didn't know you knew and you'll hate your lack of ability... until suddenly your script does what you set out to do and suddenly the clouds part and a sunbeam bathes you in glorious golden light as you proclaim yourself a GOD!!

And then you'll pass your script to someone else and they'll immediately find ten bugs.

I'm no master of the subject, but I'm happy to try to help with any questions anyone has.

Enjoy the world of coding!

Version 1.1 - January 2022