Introduction to Python Scripting For Maya Artists

By Chad Vernon

Table of Contents

Introduction

This workshop is geared towards students with little to no scripting/programming experience. By the end of this workshop, you will have the knowledge to write and run Python scripts inside and outside of Maya. You will not learn everything about Python from this workshop. This workshop includes all the basic information you should know in order to be proficient in reading, writing, running, and modifying Python scripts. The purpose of this workshop is not to make you expert Python scripters, but to give you a solid foundation from which in further your Python studies.

Additional Resources

- Learning Python, 3rd Edition by Mark Lutz
- Dive Into Python: http://www.diveintopython.org/toc/index.html
- The python_inside_maya Google email list: http://groups.google.com/group/python_inside_maya
- http://www.pythonchallenge.com/

Python in the Computer Graphics Industry

Why should you learn Python?

- Blizzard
  - Senior Look Development Technical Director
    - Strong technical knowledge of the Python and MEL programming languages.
  - Cinematics Engineer
    - Significant Python and/or Perl experience
- Pixar
  - Technical Director, Rigging
    - Writing scripts to facilitate workflow improvements in Python or MEL, or working with a programmer to enable them to make such improvements effectively.
- Lucasfilm/ILM
  - Digital FX Artist (Mid Level)
    - Some experience writing scripts (MEL, Python) and/or programming is necessary.
  - Animation Assistant Technical Director
    - Experience with Mel & Python required.
  - Digital Artist Group - Creature TD
    - Experience with C++, mel, python, or other scripting languages is desirable.
  - Technical Assistant
    - Strong scripting language (i.e. Python, Perl) and programming language (i.e. C/C++, Java) is a must.
- ImageMovers Digital
  - Look Development TD
    - Some experience with scripting/programming languages (python, MEL, PERL, C, C++) is a plus.

    - Stereoscopic TD
        - Python -- solid intermediate-to-advanced skill level.
    - Character TD
        - Python scripting and/or API/programming experience preferred.
    - Lighting TD
        - Thorough knowledge of UNIX/Linux, procedural and object-oriented programming languages, shader/plug-in writing as well as shell/Perl/Mel/Python scripting.
- Weta Digital
    - Water TD
        - Experience scripting in Python or mel preferred.
    - Lighting TD
        - Experience scripting in Python or mel preferred.
    - FX TD
        - Experience scripting in Python or mel preferred.
    - Creature TD/Senior Creature Rigger
        - Python, Perl, and C++ knowledge a plus.
    - Facial Modelers
        - Experience with Mudbox, and ability to script in MEL and Python is an asset
    - Modelers
        - Experience with Mudbox, and ability to script in MEL and Python is an asset.
- Tippett
    - Character TD
        - Experience with scripting and programming: Strong Maya MEL scripting skills. Python scripting knowledge preferred.
    - FX Animator
        - Experience with scripting and programming: Strong Maya MEL scripting skills. Python scripting knowledge preferred.

Some Programs that support Python:

- Maya
- Modo
- Houdini
- XSI
- Massive
- Blender
- Photoshop (indirectly)
- 3ds max (indirectly)

What is Python used for?

Artists can

- Automate repetitive and/or tedious tasks.
- Reduce human error.

- Produce more creative iterations in the feedback loop.

Engineers can

- Create applications and tools to run studio pipelines.
- Customize existing applications to support studio specific workflows.
- Let artists be artists.

Introduction to Python

What is Python?

Python is a general purpose scripting language used in many different industries. It is a relatively easy to use and easy to learn language. Python is used in internet services, hardware testing, game development, animation production, interface development, database programming, and many other domains.

The Python Interpreter

How does the computer run a Python program? How can you get the computer to understand the Python commands you write in a text file? The Python Interpreter is the software package that takes your code and translates it into a form that the computer can understand in order to execute the commands. This translated form is called *byte code*.

The Python Interpreter can be downloaded and installed for free from the Python website (http://www.python.org/download/). Linux, Unix, and Mac platforms usually ship with a Python Interpreter already installed. Windows users will need to download and install the interpreter if they want to use Python outside of programs such as the ones listed above. Maya 8.5 and later has a Python Interpreter built in so you could learn to use Python inside the script editor of Maya.

There are different versions of the Python Interpreter. At the time of this writing, the latest version is 3.0. Maya 8.5 uses Python 2.4. Maya 2008 uses Python 2.5. Linux and Mac users could have older versions shipped with their machines so they'll need to upgrade if they want access to newer Python functionality.

Python is available in two download types: installer and source code. The installer (an .msi file or .dmg file) is what most people will download to get Python on their machine. Source code is used by Linux users and super dorks. Python is a scripting language that was created with another programming language and the source code contains the files necessary to build and install it on Linux machines. Ignore the source download and just get the installer for now.

What is a Python Script?

A Python script is simply a bunch of commands written in a file with a .py extension. This text file is also called a Python *module*. This .py file can be written in any text editor like Note Pad, Word Pad, vi, emacs, Scite, etc. However, you do not always have to write your code in a .py file. For quick tests and scripts, you can run code directly from the Maya script editor or from the interactive prompt.

The Interactive Prompt

Interactive prompts allow you to execute code on the fly and get an immediate result. This is a good way to experiment with chunks of code when you are just learning Python. When you download and install Python, you will see that Python ships with its own editor and interactive prompt called IDLE. IDLE is a convenient editor and prompt with syntax highlighting and is all you really need in order to learn Python when just starting out. The following 3 figures show different ways of accessing an interactive prompt.

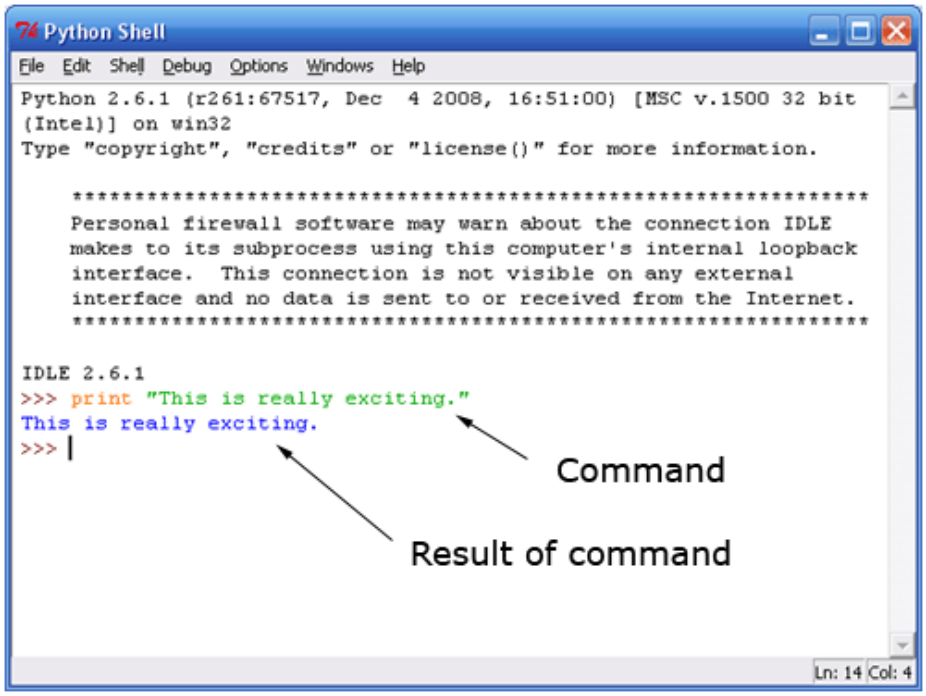


Figure 1 – IDLE

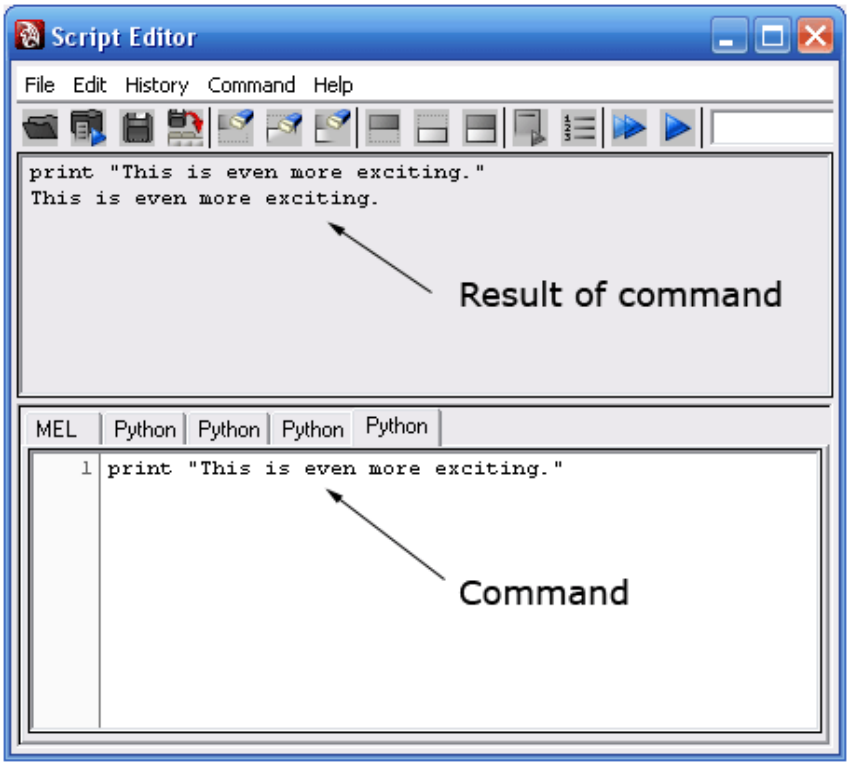


Figure 2 – Script Editor

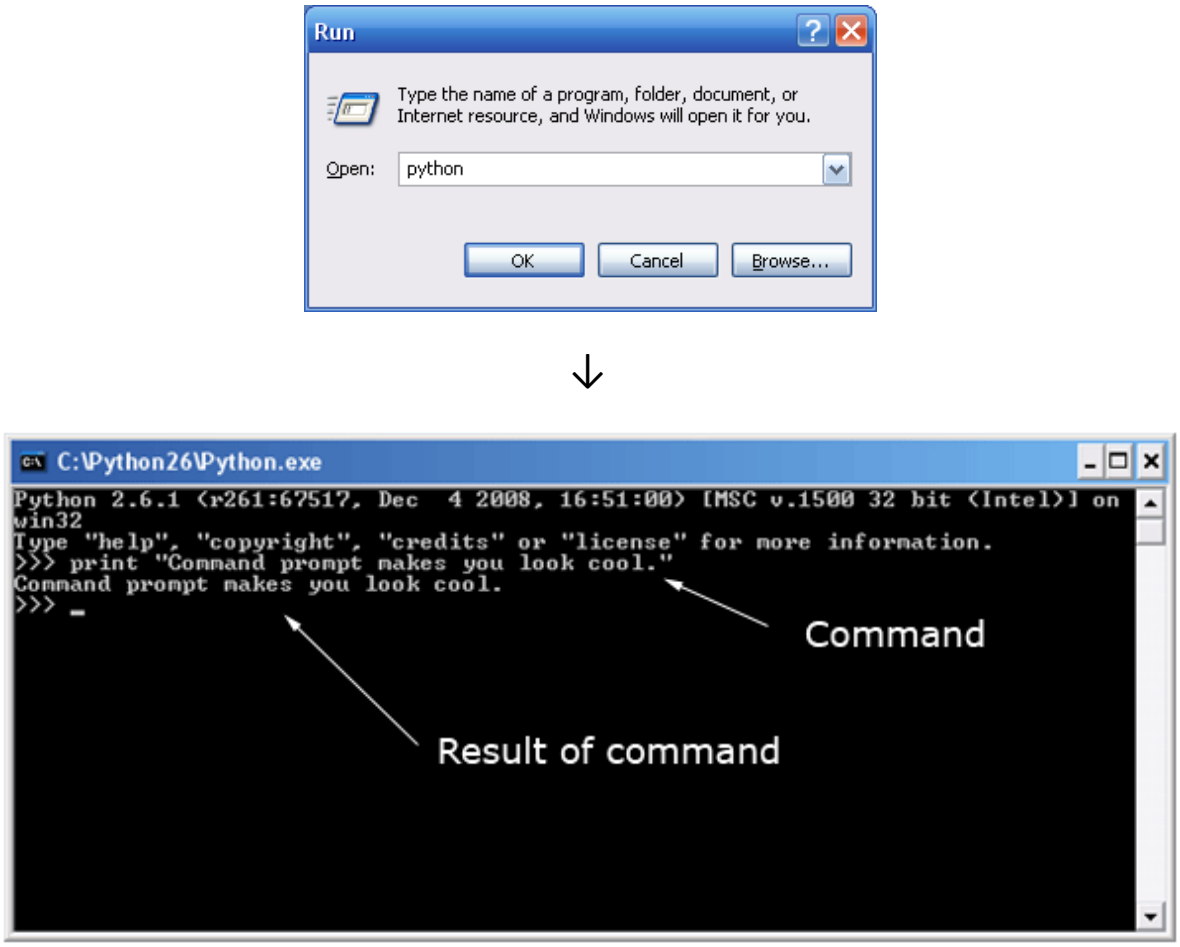


Figure 3 – Command Prompt

Running Python Scripts From a Command Prompt

Most of the code you write will be in external .py files like the one shown below.

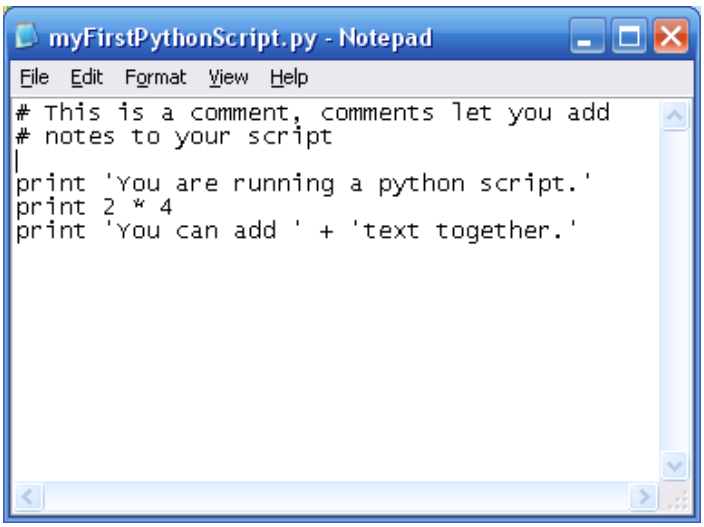


Figure 4 - A Simple Python Script

To run this script from a command prompt or terminal window, you call the script with the `python` command (which is on your system once you install Python).

```
D:\>C:\Python26\python myFirstPythonScript.py
You are running a python script.
8
You can add text together.
```

If you have the PATH environment variable (Google search “path environment variable”) set to include Python, you don’t need to specify the path to the Python executable.

```
D:\>python myFirstPythonScript.py
You are running a python script.
8
You can add text together.
```

You can also route the output of your script to a file to save it for later use:

```
D:\>python myFirstPythonScript.py > output.txt
```

Another way to run a Python script is to open it in IDLE and run it from within the editor as shown below. This is a great way to quickly experiment with Python code as you learn and write new scripts and applications.

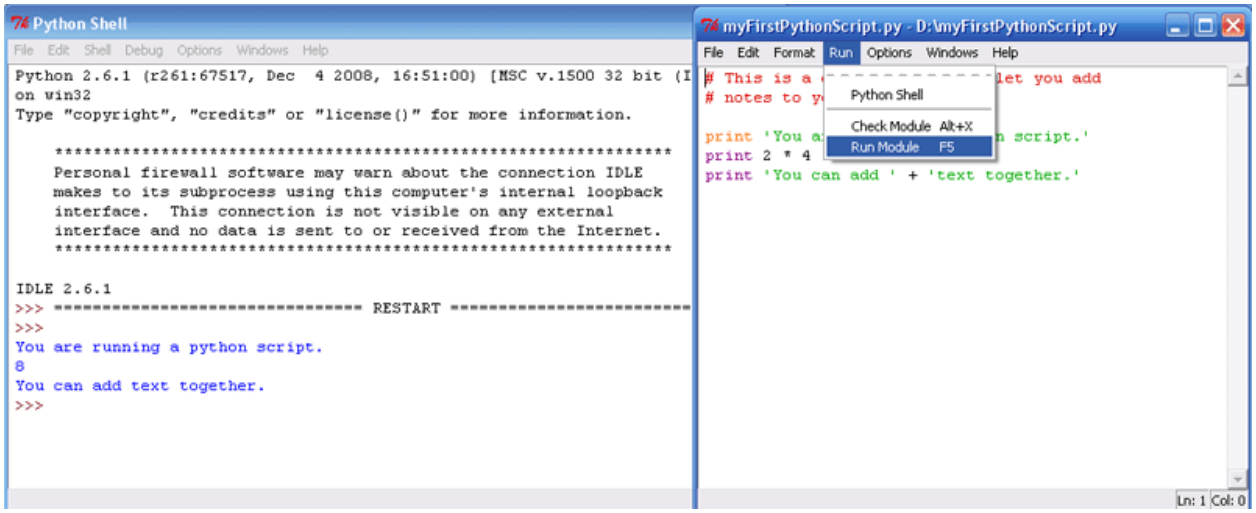


Figure 5 - Running a Script in IDLE

Running Scripts in Maya and within a Python Session

When we are in Maya, we cannot call a script with “`python myScript.py`”. The `python` command starts up the Python interpreter and runs the given script with the interpreter. When we are in Maya, the Python interpreter is already running. To call a script from inside Maya and other scripts, you need to `import` it.

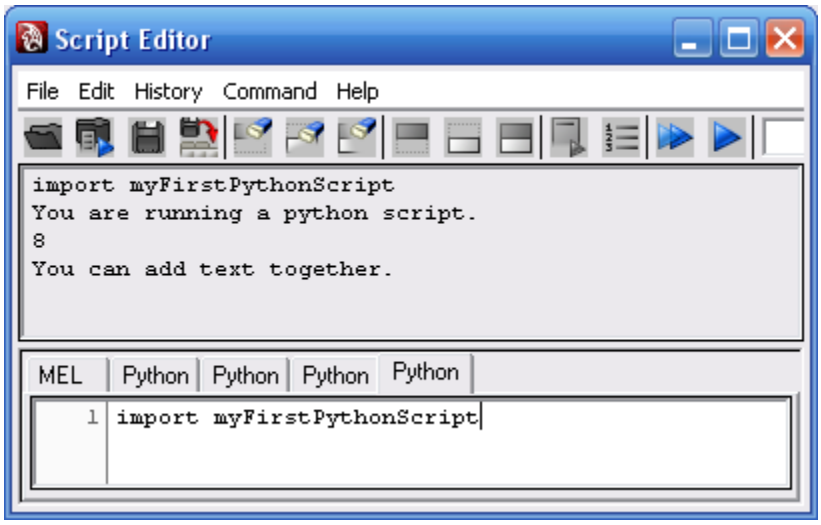


Figure 6 - Calling a Script from Maya

When you import a Python module, you leave off the .py extension. How does Python know where to find the script? Python uses an environment variable called PYTHONPATH to find scripts. The PYTHONPATH variable contains all the directories Python should search in order to find an imported module. By default, inside Maya your scripts directories are added to the PYTHONPATH. An easy way to add directories to your Maya scripts directories is to create a maya.env file. The maya.env file is a file that modifies your Maya environment each time you open Maya. Place the maya.env file in your "My Documents\maya" folder.

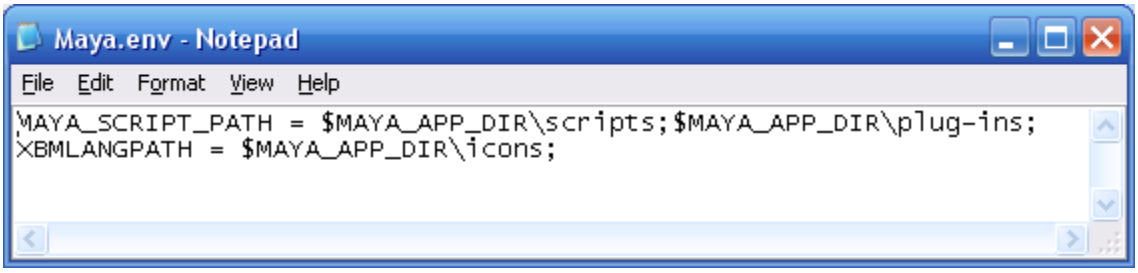


Figure 7 - maya.env File

Consult the Maya documentation for all the other variables you can set in the maya.env file. If you have multiple scripts with the same name in different directories, Python will use the first one it finds.

Notice when I import the module again, the script does not run:

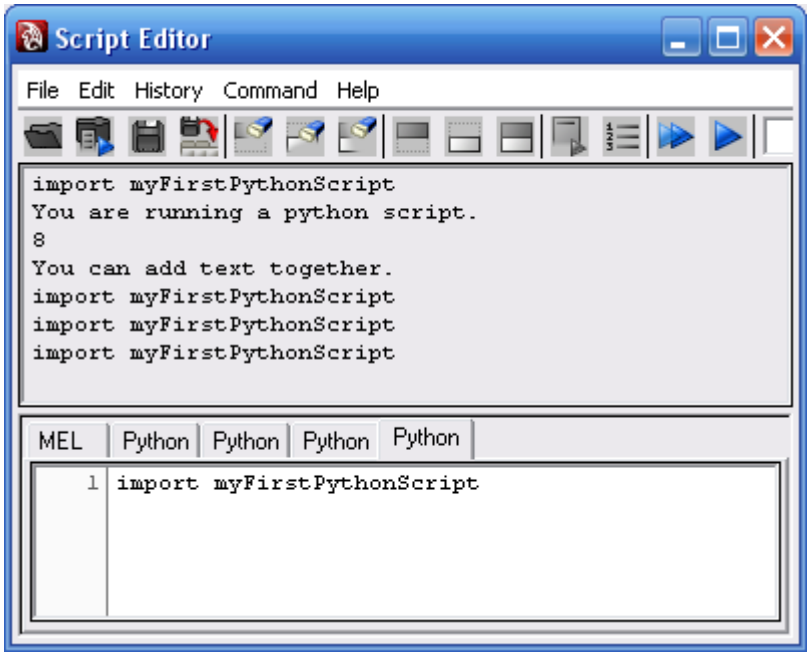
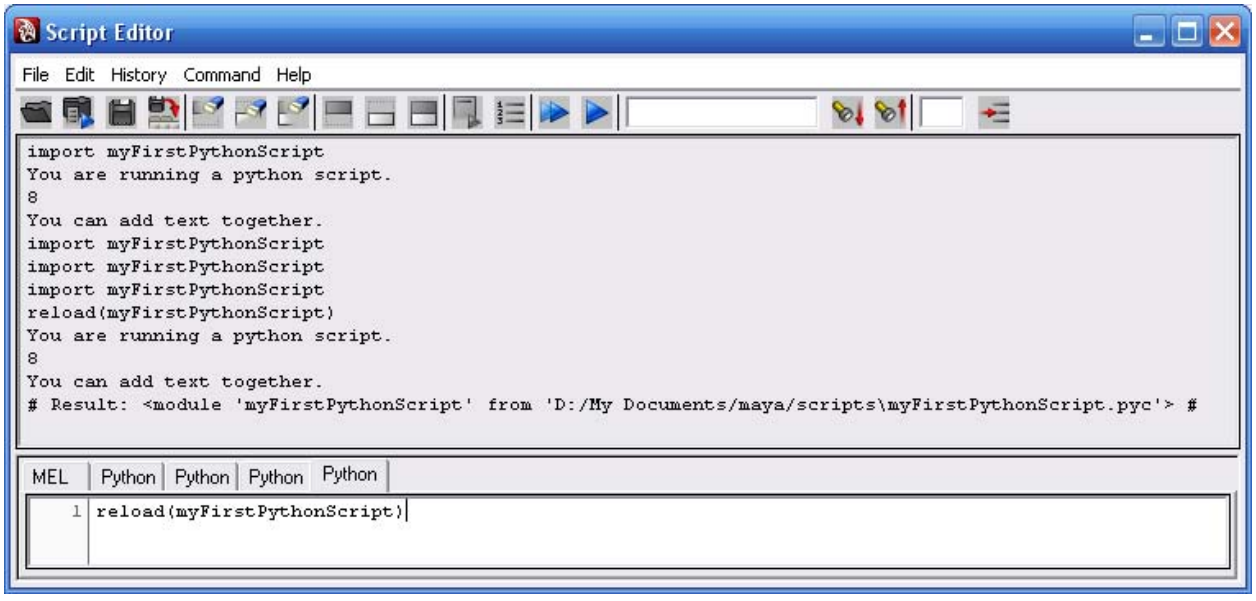


Figure 8 – Re-importing a Module

Importing a module only works once per Python session. This is because when you import a module, Python searches for the file, compiles it to byte code, then runs the code, which is an expensive process to execute multiple times. Once a module is imported, it is stored in memory. To run a script again or if you've updated a script and wish to have access to the updates, you need to `reload` it:



Notice that when I `reload` the Python module, the result states it read the module from a .pyc file. A .pyc file is a compiled Python file. When you `import` a Python module, Python compiles the code and generates a .pyc file. You could distribute these .pyc files if you do not want people looking at your code. Import statements will work with .pyc files.

Python Modules

As you can see, Python modules are simply Python scripts that contain specific functionality. Since Python is so widely used, you can find thousands of free Python modules on the internet that implement various tasks such as networking, image manipulation, file handling, scientific computing, etc. To interface with Maya, you import the maya.cmds module, which is the module that ships with Maya to implement all the Maya commands.

Data Types and Variables

Now that we know how to setup and run Python scripts, we can start learning how to write Python scripts.

Variables

The most common concept in almost all scripting and programming languages is the *variable*. A variable is a storage container for some type of data:

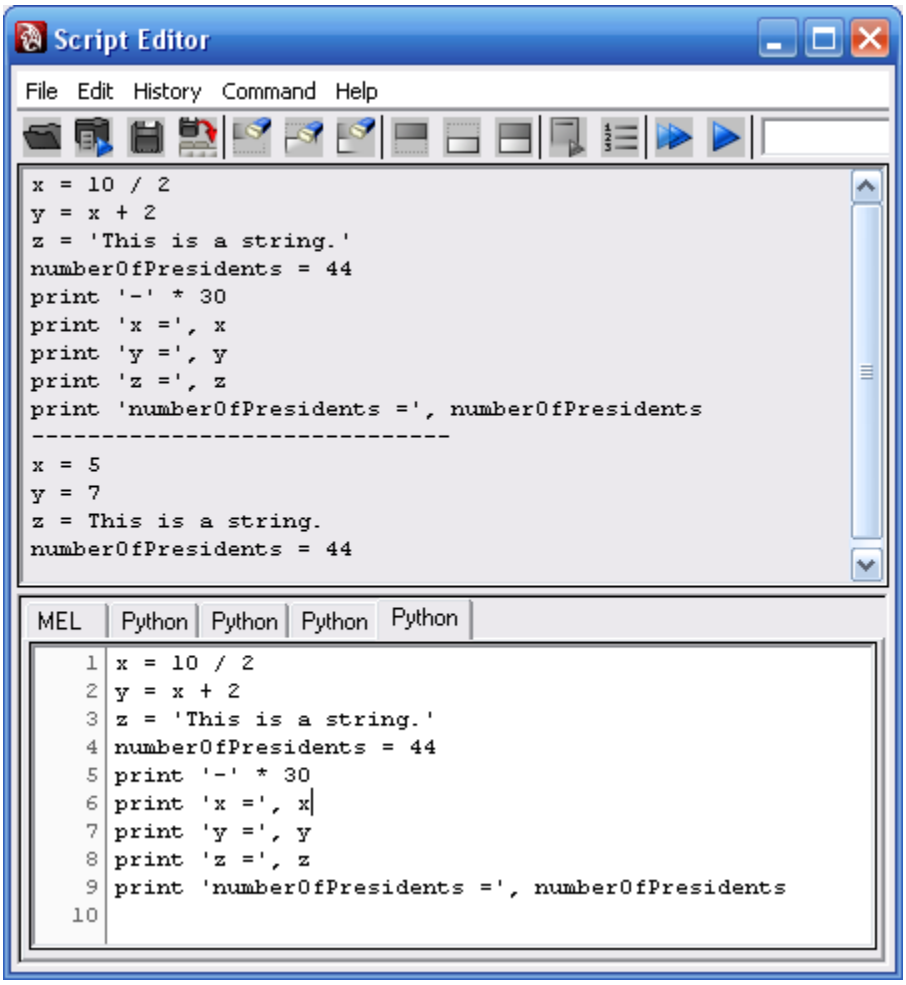


Figure 9 – Variables

Variables allow us to store data in order to use it later. Variables, sometimes called identifiers, must start with a non-numeric character or underscore (_), may contain letters, numbers, underscores (_). Identifiers are case sensitive. It is always a good idea to name your variables with descriptive names so your code is easy to read.

Legal variable names

- numberOfJoints
- button1
- teeth_geometry
- _particleEffect

Illegal variable names

- finger.nail
- 4vertexEdgeId
- cluster-handle

Python is a dynamically typed language. This means that a variable can hold one type of value for a while and then can hold another type later.

```
>>> x = 5
>>> x = 'Now x holds a string'
```

Not all languages allow you to do this. For example, in MEL, you have to declare a variable as an integer and then that variable can only hold an integer. This is what is known as a statically typed language.

Numbers and Math

There are 5 different types of number representions in Python: integers, long integers, floating-point, octal/hex, and complex numbers. The two representations that you will work with the most are integers and floating-point literals. Integers are numbers without a decimal point. Floating-point numbers are numeric values with a decimal point.

Integers

- 43
- -9234
- 6

Floating-point

- 1.324
- -23.5325
- 6.2

Python supports all the math operators that you would want to use on these numbers.

```
>>> x = 4 + 5          # Addition
>>> y = x – 8          # Subtraction
>>> y = y * 2.2        # Multiplication
>>> z = y / 1.2        # Division
>>> z = z ** 4         # Power
>>> z = -z             # Negation
>>> a = 10 % 3         # Modulus (division remainder)
>>> x += 2             # Addition and store the result back in x, same as x = x + 2
>>> x -= 2             # Subtraction and store the result back in x
>>> x *= 2             # Multiplication and store the result back in x
>>> x /= 2             # Division and store the result back in x
```

Notice in some of these statements, I use the same variable on both the right and left side of the assignment operator (=). In most programming languages, the right side is evaluated first and then the result is stored in the variable on the left. So in the statement `y = y * 2.2,` the expression `y * 2.2` is evaluated using the current value of `y,` in this case 1, and then the result (1 * 2.2) = 2.2 is stored in the variable `y.`

Math operators have a precedence of operation. That is, some operators always execute before others even when in the same expression. For example the following two lines give different results:

```
>>> print (5 + 3.2) * 2
16.4
>>> print 6 + 3.2 * 2
12.4
```

Multiplication and division always get evaluated before addition and subtraction. However, you can control which expressions get evaluated first by using parentheses. Inner most parentheses always get evaluated first.

```
3 * (54 / ((2 + 7) * 3)) – 4
3 * (54 / (9 * 3)) – 4
3 * (54 / 27) – 4
3 * 2 – 4
6 – 4
2
```

When you mix integers and floating-point values, the result will always turn into a floating-point:

```
>>> 4 * 3.1
# Result: 12.4 #
```

However, you can explicitly turn an int to a float or a float to an int.

```
>>> int(4 * 3.1)
# Result: 12 #
>>> float(12)
# Result: 12.0 #
```

Strings

Strings are text values. You can specify strings in single, double or triple quotes.

```
>>> ‘This is a string’                          # Single quotes
>>> “This is also a string”                     # Double quotes
>>> “””This string can span multiple lines”””    # Triple quotes
```

Single and double quoted strings are the same. The main thing to keep in mind when using strings are escape sequences. Escape sequences are characters with special meaning. To specify an escape code, you use the backslash character followed by a character code. For example a tab is specified as ‘\t’, a new line is specified as ‘\n’. You have to be mindful whenever escape sequences are involved because they can lead to a lot of errors and frustration. For example, in Windows, paths are written using the backslash: (e.x. C:\tools\new). If you save this path into a string, you get unexpected results:

```
>>> print "C:\tools\new"
C:    ools
Ew
```

Python reads the backslashes as an escape sequence. It thinks the '\t' is a tab and the '\n' is a new line. To get the expected results, you either use escaped backslashes or a raw string:

```
>>> print "C:\\tools\\new"
C:\tools\new
>>> print r"C:\tools\new"
C:\tools\new
```

You can concatenate strings together with the + operator.

```
>>> x = 'I am '
>>> y = '25 years old.'
>>> print x + y
I am 25 years old.
```

However, you cannot add a string and a number.

```
>>> x = 'I am '
>>> y = 25
>>> print 'I am ' + y
# Error: cannot concatenate 'str' and 'int' objects
# Traceback (most recent call last):
#   File "<maya console>", line 3, in <module>
# TypeError: cannot concatenate 'str' and 'int' objects #
```

Python will throw an error saying you cannot concatenate a string and an integer. You can fix this in a couple different ways. The easiest is to convert the integer to a string.

```
>>> x = 'I am '
>>> y = 25
>>> print 'I am ' + str(y)
I am 25
```

A better way is to use string formatting. String formatting allows you to code multiple string substitutions in a compact way. You use string substitution with the % operator.

```
>>> x = 'I am '
>>> y = '25 years old.'
>>> print '%s%s' % (x, y)
I am 25 years old.
>>> x = 'I am '
>>> y = 25
>>> print '%s%d' % (x, y)
I am 25
```

To use string formatting, you provide a format string on the left of the % operator and the values you want to print on the right of the % operator. Different types of values have different format codes. Common codes include:

- %s - string
- %d, %i - integer
- %f - floating-point

In the second example above, `'%s%d' % (x, y)` contains two format codes, a string followed by an integer. On the right side of the % operator, we need to specify a value for each of the format codes in the order they appear in the format string. String formatting not only lets us code in a more compact way, it also lets us format values for output:

```
>>> print '%03d' % 5
005
>>> print '%+03d' % 6
+06
>>> print '%+03d' % -4
-04
>>> print '%.5f' % 4.4242353534
4.42424
```

With string formatting we can specify decimal precision, how many spaces a number should be printed with, whether to include the sign, etc. This is especially useful when printing out large tables of data.

String Methods

Methods are chunks of code that perform some type of operation. We will learn more about methods, also known as functions, in more detail later on, but now is a good time to introduce you to the syntax of calling a method. We call a method using the dot operator (.):

```
object.method()
```

An object is an instance of a particular type. For example, we could have a string object.

```
>>> x = 'This is a string.'    # x is a string object
>>> y = 'Another string.'      # y is another string object
```

When we read the code, `someObject.someMethod()`, we say we are calling the method named `someMethod` from the object called `someObject`. Most objects have many callable methods. Below are some of the methods found in string objects:

```
>>> x = 'This is my example string.'
>>> print x.lower()
this is my example string.
>>> print x.upper()
THIS IS MY EXAMPLE STRING.
>>> print x.replace('my', 'your')
This is your example string.
>>> print x.title()
This Is My Example String.
>>> print x.endswith('ng.')
True
>>> print x.endswith('exam')
False
```

Some of the methods have *arguments* in the parentheses. Many methods allow you to pass in values to perform operations based on the arguments. For example `x.replace('my', 'your')` will return a copy of string `x` with all of the `'my'` instances replaced with `'your'`. There are many methods available for many different types of objects and there is no need to memorize them. You will begin to

memorize them after using them a lot. You can find help documentation for all objects with the `help` command.

`help(str)`

The help command will print out the documentation associated with an object, class, or function.

Lists

Lists are sequences or arrays of data. Lists allow us to use organized groups of data in our scripts.

```
>>> listOfLights = ['keyLight', 'rimLight', 'backLight', 'fillLight']
>>> print listOfLights[0]     # print the first element
keyLight
>>> print listOfLights[2]     # print the third element
backLight
>>> print listOfLights[3]     # print the fourth element
fillLight
>>> print listOfLights[-1]    # print the last element
fillLight
>>> print listOfLights[0:2]   # print the first through second elements
['keyLight', 'rimLight']
>>> print len(listOfLights)   # print the length of the list
4
```

We can access elements in a list with a numeric index. The indices start at index 0 and increment with each value in the list. We can also access subsets of lists with *slicing*

```
>>> print listOfLights[0:2], listOfLights[:3], listOfLights[:-1]
['keyLight', 'rimLight'] ['keyLight', 'rimLight', 'backLight'] ['keyLight',
'rimLight', 'backLight']

>>> listOfLights[0:2] = ['newLight', 'greenLight']
>>> print listOfLights
['newLight', 'greenLight', 'backLight', 'fillLight']
```

When an index is negative, it counts from the end of the list back. Lists can also contain mixed types of data including other lists.

```
>>> myList = [3, 'food', ['anotherList', 4 * 2, 'dog'], 80.3]
>>> print myList[2][1]
8
```

When you try to access an element that does not exist, Python will throw an error.

```
>>> print myList[34]
# Error: list index out of range
# Traceback (most recent call last):
#   File "<maya console>", line 1, in <module>
# IndexError: list index out of range #
```

Lists, like strings, have their own set of methods available.

```
>>> listOfLights = ['keyLight', 'rimLight', 'backLight', 'fillLight']
>>> listOfLights.sort()
>>> print listOfLights
['backLight', 'fillLight', 'keyLight', 'rimLight']
>>> listOfLights.reverse()
>>> print listOfLights
['rimLight', 'keyLight', 'fillLight', 'backLight']
>>> listOfLights.append('newLight')
>>> print listOfLights
['rimLight', 'keyLight', 'fillLight', 'backLight', 'newLight']
```

This concept of indexing is not unique to lists. We can actually access strings the same way.

```
>>> x = 'Eat your vegetables'
>>> print x[0:6]
Eat yo
>>> print x[:-9]
Eat your v
>>> print x[5:]
our vegetables
>>> print x.find('veg')
9
>>> print x.split('e')
['Eat your v', 'g', 'tabl', 's']
```

You can think of strings as lists of characters. The main difference is strings cannot be edited in place with indices. For example, the following is illegal.

```
>>> x[4] = 't'
# Error: 'str' object does not support item assignment
# Traceback (most recent call last):
#   File "<maya console>", line 1, in <module>
# TypeError: 'str' object does not support item assignment #
```

Strings are what are known as immutable objects. Once they are created, they cannot be changed. Lists on the other hand are mutable objects, meaning they can change internally.

Tuples

Tuples are the same as lists except they are immutable. They cannot be changed once created.

```
>>> myTuple = (5, 4.2, 'cat')
>>> myTuple[1] = 3
# Error: 'tuple' object does not support item assignment
# Traceback (most recent call last):
#   File "<maya console>", line 1, in <module>
# TypeError: 'tuple' object does not support item assignment #
```

What is the purpose of tuples? There are aspects of Python that use or return tuples. I'd say 99.9% of the time, you'll be using lists. Just be aware that you cannot change a tuple when one eventually pops up.

Dictionaries

Dictionaries are like lists except their elements do not have to be accessed with numeric indices. Dictionaries are a type of look-up table or hash map. They are useful in storing unordered types of data.

```
>>> characters = {'male' : ['Derrick', 'Chad', 'Ryan'], 'female' : ['Olivia', 'Sarah',
'Zoe']}
>>> print characters['male']
['Derrick', 'Chad', 'Ryan']
>>> print characters['female']
['Olivia', 'Sarah', 'Zoe']

>>> # Same functionality, different syntax:
>>> characters = dict(male=['Derrick', 'Chad', 'Ryan'], female=['Olivia', 'Sarah',
'Zoe'])
>>> print characters['male']
['Derrick', 'Chad', 'Ryan']
>>> print characters['female']
['Olivia', 'Sarah', 'Zoe']

# Some methods of dictionaries
>>> print characters.keys()
['male', 'female']
>>> print characters.values()
[['Derrick', 'Chad', 'Ryan'], ['Olivia', 'Sarah', 'Zoe']]
>>> print characters.has_key('male')
True
>>> print characters.has_key('shemale')
False
>>> print 'female' in characters
True

# Elements can be added on the fly
>>> characters['alien'] = ['ET', 'Alf']
>>> characters[6] = 4.5
```

Booleans and Comparing Values

In many of your programs, you'll need to determine the relationship between variables such as if values are equal or if a value is greater than or less than another. These comparisons return a *boolean* value. A boolean value is simply True or False. You have already seen these values returned in a few of the previous examples.

```
>>> # x == y tests is x is equal to y
>>> # x != y tests if x does not equal y
>>> # x > y tests if x is greater than y
>>> # x < y tests if x is less than y
>>> # x >= y tests if x is greater than or equal to y
>>> # x <= y tests if x is less than or equal to y
>>> x = 4.3
>>> y = 2
>>> 5 == 5, 5 == 8, x == y, x == 4.3
(True, False, False, True)
>>> x <= y, x > y, x == y + 2.3, x >= 93
(False, True, True, False)
>>> a = 'alpha'
>>> b = 'beta'
>>> a > b, a == 'alpha', a < b
(False, True, True)
```

The following values are also considered False:

```
>>> ''          # Empty strings
>>> []          # Empty lists
```

```
>>> {}           # Empty dictionaries
>>> 0.0          # 0 valued numbers
>>> None         # An empty value
```

The following values are considered True:

```
>>> 'text'       # Strings with any characters
>>> [2, 3]       # Lists with elements
>>> {3 : 4.5}    # Dictionaries with elements
>>> 2.3          # Non-zero numbers
```

Controlling the Flow of Execution

All the scripts so far have been executed line by line. In most of your scripts, you will need to be able to control the flow of execution. You will need to execute code only if certain conditions are met and you will need to execute code multiple times. This is called the logic of the script. Python, like most other languages, supports the basic constructs to accomplish this.

Conditionals

To execute code only if a condition is True or False, you use the `if`, `elif`, and `else` statements.

```
>>> x = 5
>>> if x == 5:
>>>     x += 3
>>> print x
8

>>> x = 5
>>> if x < 5:
>>>     x += 3
>>> else:                # Optional else
>>>     x *= 2
>>> print x
10

>>> x = 5
>>> if x > 5 and not x == 2:
>>>     x += 2
>>> elif x == 5:         # Optional elif
>>>     x += 4
>>> elif x == 9:         # Optional elif
>>>     x -= 3
>>> else:                # Optional else
>>>     x *= 2
>>> print x
9

>>> x = 5
>>> if x == 5:
>>>     x += 3
>>> if x == 8:
>>>     x *= 2
>>>     if x == 16
>>>         x -= 10
>>> print x
6

>>> x = [‘red’, ‘green’, ‘blue’]
>>> if ‘red’ in x:
>>>     print ‘Red hot!’
Red hot!
```

`if` statements let you select chunks of code to execute based on boolean values. In a sequence of `if/elif/else` statements, the first `if` statement is evaluated as True or False. If the condition is True, the code in its corresponding code block is executed. If the condition is False, the code block is skipped and execution continues to the next `else` or `elif (else if)` statement if one exists. `else` statements

must always be preceeded by an `if` or `elif` statement but can be left out if not needed. `elif` statements must always be preceeded by and `if` statement but can left out if not needed.

Code Blocks

Code blocks are chunks of code associated with another statement of code. Take the following code for example.

```
>>> x = 'big boy'
>>> y = 7
>>> if x == ‘big boy’ and y < 8:
>>>     print ‘Big Boy!’
>>>     y += 3
```

The last two lines are in the code block associated with the `if` statement. When the condition of the `if` statement is evaluated as True, execution enters the indented portion of the script. Code blocks in Python are specified by the use of whitespace and indentation. You can use any number of spaces or even tabs, you just have to be consistent throughout your whole script. However, even though you can use any amount of whitespace, **the Python standard is 4 spaces.** Code blocks can be nested in other code blocks, you just need to make sure your indentation is correct.

```
>>> x = 5
>>> y = 9
>>> if x == 5 or y > 3:
>>>     x += 3
>>>     if x == 8:
>>>         x *= 2
>>>     elif y == 7:
>>>         y -= 3
>>>         if x == 16 or y < 21:
>>>             x -= 10
>>>     else:
>>>         y *= 3
>>> else:
>>>     x += 2
>>> print x
16
>>> print y
9
```

`if/elif/else` statements that are chained together need to be on the same indentation level. Read though the previous example and work out the flow of execution in your head or on paper.

While Loops

While loops allow you to run code while a condition is True.

```
>>> x = 5
>>> while x > 1:
>>>     x -= 1
>>>     print x
4
3
2
1
```

In the above example, the condition is tested as True, so execution enters the code block of the `while` loop. When execution reaches the print statement, the value of x as been decremented and execution returns to the `while` statement where the condition is tested again. This loop continues until the condition is False. You have to take care that this condition is eventually gets a False value.

```
>>> x = 5
>>> while x > 1:
>>>     x += 1
```

In the above example, x will continue to increment and the condition will always be True. This is called an infinite loop. If you create one of these, you'll have to shut down your program or Ctrl-Alt-Delete out of Maya.

Sometimes you will want to exit out of a loop early or skip certain iterations in a loop. These can be done with the `break` and `continue` commands.

```
>>> x = 0
>>> while x < 10:
>>>     x += 1
>>>     if x % 2:
>>>         continue    # When x is an odd number, skip this loop iteration
>>>     if x == 8:
>>>         break       # When x == 8, break out of the loop
>>>     print x
>>> else:               # This optional else statement is run if the loop finished
>>>     x = 2           # without hitting a break
2
4
6

>>> # This code causes an infinite loop, try to find out why.
>>> x = 0
>>> while x < 10:
>>>     if x % 2:
>>>         continue
>>>     if x == 8:
>>>         break
>>>     print x
>>>     x += 1
```

For Loops

For loops iterate over sequences objects such as lists, tuples, and strings. Sequence objects are data types comprised of multiple elements. The elements are usually accessed by square brackets (e.g. `myList[3]`) as you've seen previously. However, it is often useful to be able to iterate through all of the elements in a sequence.

```
>>> someItems = ['truck', 'car', 'semi']
>>> for x in someItems:
>>>    print x
truck
car
semi
```

```
>>> print range(5)        # Built-in function range creates a list of integers
[0, 1, 2, 3, 4]
>>> for x in range(5):
>>>     # Create a spine joint in Maya
>>>     pass              # pass is used when you have no code to write

>>> for letter in 'sugar lumps':
>>>    print letter,
s u g a r   l u m p s

>>> someItems = ['truck', 'car', 'semi']
>>> someColors = ['red', 'green', 'blue']
>>> # the zip function pulls out pairs of items
>>> for x, y in zip(someItems, someColors):
>>>    print 'The %s is %s' % (x, y)
The truck is red
The car is green
The semi is blue

>>> for i in range(0, 10, 2):      # range(start, stop, step)
>>>     if i % 2 == 1:
>>>         continue
>>>     if i > 7:
>>>         break
>>>     print i,
>>> else:                          # Optional else
>>>     print 'Exited loop without break'
0 2 4 6
```

Like the `while` loop, `for` loops support the `continue`, `break`, and `else` statements.

Functions

Previously, we've seen functions and methods built in to Python (such as `range` and `zip`) and built in to different data types (string, list, and dictionary methods). Functions allow us to create reusable chunks of code that we can call throughout our scripts. Functions are written in the form

```
def functionName(optional, list, of, arguments):
    # statements
```

```
>>> def myPrintFunction():
>>>     print 'woohoo!'
>>>
>>> myPrintFunction()
woohoo!
>>> for x in range(3):
>>>     myPrintFunction()
woohoo!
woohoo!
woohoo!
```

Functions also accept arguments that get passed into your function.

```
>>> def printMyOwnRange(start, stop, step):
>>>     x = start
>>>     while x < stop:
>>>         print x
>>>         x += step

>>>
>>> printMyOwnRange(0, 5, 2)
0
2
4
```

Functions can return values.

```
>>> def getMyOwnRange(start, stop, step):
>>>     x = start
>>>     listToReturn = []
>>>     while x < stop:
>>>         list.append(x)
>>>         x += step
>>>     return listToReturn
>>>
>>> x = getMyOwnRange(0, 5, 2)
>>> print x
[0, 2,4]
```

Functions can also return multiple values.

```
>>> def getSurroundingNumbers(number):
>>>     return number + 1, number - 1
>>>
>>> x, y = getSurroundingNumbers(3)
>>> print x, y
2 4
```

Function Arguments

Function parameters (arguments) can be passed to functions a few different ways. The first is positional where the arguments are matched in order left to right:

```
>>> def func(x, y, z):
>>>     pass
>>>
>>> func(1, 2, 3)       # Uses x = 1, y = 2, z = 3
```

Functions can have a default value if a value isn't passed in:

```
>>> def func(x, y=3, z=10):
>>>     pass
>>>
>>> func(1)             # Uses x = 1, y = 3, z = 10
```

You can also specify the names of arguments you are passing if you only want to pass certain arguments. These are called keyword arguments. **This is the method used in the Maya commands.**

```
>>> def func(x=1, y=3, z=10):
>>>     pass
>>>
>>> func(y=5)           # Uses x = 1, y = 5, z = 10
```

You can also have an arbitrary number of arguments:

```
>>> def func(*args):
>>>     print args
>>>
>>> func(1, 2, 3, 4)      # Passes the arguments as a tuple
(1, 2, 3, 4)
```

And you can have an arbitrary number of keyword arguments:

```
>>> def func(**kwargs):
>>>     print kwargs
>>>
>>> func(joints=1, x=2, y=3, z=4)      # Passes the arguments as a dictionary
{'y': 3, 'joints': 1, 'z': 4, 'x': 2}
```

Scope

Scope is the place where variables and functions are valid. Depending on what scope you create a variable, it may or may not be valid in other areas of your code.

```
>>> def myFunction():
>>>     x = 1     # x is in the local scope of myFunction
```

Basic scope rules:

1. The enclosing module (the .py file you create the variable in) is a global scope.
2. Global scope spans a single file only.

3. Each call to a function is a new local scope.
4. Assigned names are local, unless declared global.

Examples:

```
>>> x = 10
>>> def func():
>>>     x = 20
>>>
>>> func()
>>> print x        # prints 10 because the function creates its own local scope

>>> x = 10
>>> def func():
>>>     global x
>>>     x = 20
>>>
>>> func()
>>> print x        # prints 20 because we explicitly state we want to use the global x

>>> x = 10
>>> def func():
>>>     print x
>>> func()         # prints 10 because there is no variable x declared in the local
                   # scope of the function so Python searches the next highest scope
```

Lambda Expressions

Lambda expressions are basically a way of writing short functions. Normal functions are usually of the form:

```
def name(arg1, arg2):
    statements...
```

Lambda expressions are of the form:

```
lambda arg1, arg2: expression
```

This is useful because we can embed functions straight into the code that uses it. For example, say we had the following code:

```
>>> def increment(x):
>>>     return x + 1
>>> def decrement(x):
>>>     return x – 1
>>> def crazy( x ):
>>>     return x / 2.2 ** 3.0
>>>
>>> D = {'f1' : increment, 'f2' : decrement, 'f3' : crazy}
>>> D['f1']( 2)
```

A drawback of this is that the function definitions are declared elsewhere in the file. Lambda expressions allow us to achieve the same effect as follows:

```
>>> D = {'f1' : (lambda x: x + 1), 'f2' : (lambda x: x - 1), ('f3' : lambda x: x / 2.2
** 3.0)}
>>> D['f1'](2)
```

Note that lambda bodies are a single expression, not a block of statements. It is similar to what you put in a def return statement. When you are just starting out using Python in Maya, you probably won't be using many lambda expressions, but just be aware that they exist.

Exceptions and Error Handling

Whenever an error is encountered in your program, Python will raise an exception stating the line number and what went wrong:

```
>>> x = [1,2,3]
>>> x[10]
Traceback (most recent call last):
  File "<pyshell#107>", line 1, in <module>
    x[10]
IndexError: list index out of range
```

If we want the code to continue running, we can use a try/except block:

```
>>> x = [1,2,3]
>>> try:
>>>     x[10]
>>> except IndexError:
>>>     print "What are you trying to pull?"
>>> print "Continuing program..."
```

Another variation is:

```
>>> x = [1,2,3]
>>> try:
>>>     x[10]
>>> except IndexError:
>>>     print "What are you trying to pull?"
>>> else:                  # Will only run if no exception was raised
>>>     print "No exception was encountered"
>>> print "Continuing program..."
```

You can raise your own exception if you want to prevent the script from running:

```
>>> if func() == "bad return value":
>>>     raise RuntimeError("Something bad happened")
```

Files

Python is used a lot in managing files and file systems.

```
>>> output = open(r'X:\data.dat', 'w')  # Open file for writing
>>> input = open('data.txt', 'r')       # Open file for reading
>>> x = input.read()                    # Read entire file
>>> x = input.read(5)                   # Read 5 bytes
>>> x = input.readline()                # Read next line
>>> L = input.readlines()               # Read entire file into a list of line strings
>>> output.write("Some text")           # Write text to file
>>> output.writelines(list)             # Write all strings in list L to file
>>> output.close()                      # Close manually
```

Here's a more practical example of opening a maya ascii file and renaming myBadSphere to myGoodSphere.

```
>>> mayaFile = open(r'C:\myfile.ma', 'r')
>>> fileContents = mayaFile.readlines()
>>> mayaFile.close()
>>> for i in range(len(fileContents)):
>>>     fileContents[i] = fileContents[i].replace('myBadSphere', 'myGoodSphere')
>>> mayaFile = open(r'C:\myfile2.ma', 'w')
>>> mayaFile.writelines(fileContents)
>>> mayaFile.close()
```

Modules

We learned in the beginning that Python modules are simply .py files full of Python code. These modules can be full of functions, variables, classes (more on classes later), and other statements. We also learned that to load a Python module from within Maya or another interactive prompt, we need to `import` the module. When we `import` a module, we gain access to all the functionality of that module.

myMathModule.py:

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x – y
```

Inside Maya, we can access this functionality as follows:

```
import myMathModule
myMathModule.add(1, 2)
```

Or

```
import myMathModule as mm
mm.add(1,2)
```

Or

```
from myMathModule import add
add(1, 2)
```

Or

```
from myMathModule import *
add(1, 2)
subtract(1, 2)
```

Remember, we can only import a module once per Python session. If we were to update the code in myMathModule.py, we wouldn't have access to the updates until we reload the module.

```
reload(myMathModule)
```

We can also import modules into other modules. If we have one module full of some really useful functions, we can import that module into other scripts we write (which are also modules) in order to gain access to those functions in our current module.

Module Packages

Packages allow us to organize our Python modules into organized directory structures. Instead of placing all of our modules into one flat directory, we can group our modules based on functionality.

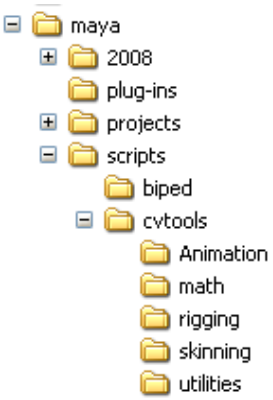


Figure 10 - A Sample Package Organization

To create a package, create a folder in one of you PYTHONPATH directories like your Maya script directory, then create a file called __init__.py inside of that new folder. The __init__.py can be empty. You can then place your modules into that package and import them as follows.

```
>>> import packageName.moduleName
```

You can have packages inside of packages.

```
>>> import cvtools.rigging.createIkLeg as createIkLeg
>>> createIkLeg('L_leg_joint')
```

Any code you put in the __init__.py file of a package gets executed with the package. For example, in the above sample package hierarchy, I could have code in scripts/cvtools/__init__.py to create a Maya menu when the package is imported.

```
>>> import cvtools     # creates a Maya menu
```

Built-In and Third Party Modules

Python ships with many built-in modules that give you access to really useful functionality. There is a module with many useful math functions, a module to copy files, a module to generate random numbers, etc.

```
import sys          # System commands
import os           # Generic module to the operating system
import shutil       # Contains file copying functions
import struct       # Deals with binary data
import xmllib       # XML parser module
import math         # Contains many useful math operations
import random       # Random number module
import re           # Regular expressions module
import optparse     # Command-line option parser
```

There are also hundreds of third-party modules available online. For example, the pil module contains many functions that deal with image manipulation. To find out what functions are in a module, run the help command or view online documentation.

Classes and Object Oriented Programming

Quick Notes

Python is an object-oriented language. It supports all the usual functionality of such languages such as classes, inheritance, and polymorphism. If you are not familiar with object-oriented programming (or scripting), it basically means it is easy to create modular, reusable bits of code, which are called objects. We have already been dealing with objects such as string objects and list objects. Below is an example of creating a string object and calling its methods.

```
x = "happy"
x.capitalize()              # returns Happy
x.endswith("ppy")           # returns True
x.replace("py", "hazard")   # returns "haphazard"
x.find( "y" )               # returns 4
```

You are free to use Python without using any of its object oriented functionality by just sticking with functions and groups of statements as we have been doing throughout these notes. However, if you would like to create larger scale applications and systems, I recommend learning more about object oriented programming. The Maya programming API and pymel, the popular Maya commands replacement module, are built upon the notions of object oriented programming so if you want to use API functionality or pymel in your scripts, you should understand the principles of OOP.

Classes

Classes are the basic building blocks of object oriented programming. With classes, we can create independent instances of a common object. It is kind of like duplicating a cube a few times. They are all cubes, but they have their own independent attributes.

```
class shape:
    def __init__(self, name):
        self.name = name

    def printMyself(self):
        print 'I am a shape named %s.' % self.name

>>> shape1 = shape(name='myFirstShape.')
>>> shape2 = shape(name='mySecondShape.')
>>> shape1.printMyself()
I am a shape named myFirstshape.
>>>shape2.printMyself ()
I am a shape named mySecondShape.
```

The above example shows a simple class that contains one data member (name), and two functions. Functions that begin with a double underscore usually have a special meaning in Python. The `__init__` function of a class is a special function called a constructor. It allows us to construct a new instance of an object. In the example, we create two independent *instances* of a shape object: shape1 and shape2. Each of these instances contains its own copy of the name attribute defined in the class definition. In the shape1 instance, the value of name is “myFirstShape”. In the shape2 instance, the value of name is “mySecondShape”. Notice we don’t pass in any value for the `self` argument. We don’t

pass in any value for the self argument because the self argument refers to the particular instance of a class.

The first argument in all class methods (functions) should be the `self` argument. The `self` argument is used to represent the current instance of that class. You can see in the above example when we call the `printMyself` method of each instance, it prints the name stored in each separate instance. So objects are containers that hold their own copies of data defined in their class definition.

Inheritance and Polymorphism

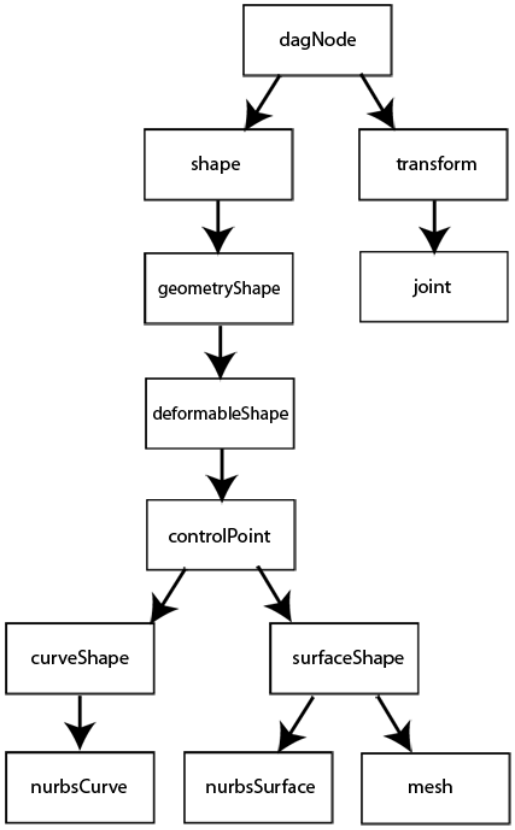
Inheritance and polymorphism are OOP constructs that let us build off of existing functionality. Say we wanted to add additional functionality to the previous shape class but we don't want to change it because many other scripts reference that original class. We can create a new class that *inherits* the functionality of that class and then we can use that inherited class to build additional functionality:

```
class polyCube(shape):
    def __init__(self, name, length, width, height):
        # Call the constructor of the inherited class
        shape.__init__(name)

        # Store the data associated with this inherited class
        self.length = length
        self.width = width
        self.height = height

    def printMyself(self):
        shape.printMyself(self)
        print 'I am also a cube with dimensions %.2f, %.2f, %.2f.' % (length, width,
height)


class polySphere(shape):
    def __init__(self, name, radius):
        # Call the constructor of the inherited class
        shape.__init__(name)

        # Store the data associated with this inherited class
        self.radius = radius

    def printMyself(self):
        shape.printMyself(self)
        print 'I am also a sphere with a radius of %.2f.' % radius

>>> cube1 = polyCube('firstCube', 2.0, 1.0, 3.0)
>>> cube2 = polyCube('secondCube', 3.0, 3.0, 3.0)
>>> sphere1 = polySphere('firstSphere', 2.2)
>>> sphere2 = polySphere('secondSphere', 2.5)
>>> shape1 = shape('myShape')
>>> cube1.printMyself()
I am a shape named firstCube.
I am also a cube with dimensions 2.00, 1.00, 2.00.
>>> cube2.printMyself()
I am a shape named secondCube.
I am also a cube with dimensions 3.00, 3.00, 3.00.
>>> sphere1.printMyself()
I am a shape named firstSphere.
I am also a sphere with a radius of 2.20.
>>> sphere2.printMyself()
```

```
I am a shape named secondSphere.
I am also a sphere with a radius of 2.50.
```

In the above example, we create two new classes, `polyCube` and `polySphere`, that inherit from the base class, shape. The two new classes will have all the data and methods associated with the shape `base` class. When we call the constructor method, __init__, of `polyCube` and `polySphere`, we still want to use the functionality of the constructor of its *super class*, `shape`. We can do this by calling the super class constructor explicitly with shape.__init__(self). This will set the name variable since the name variable is inherited from the `shape` class. Notice when we call the constructor of shape inside of the inherited classes, we pass in the self argument. We do this because we've already created the instance by calling the polyCube or polySphere constructors. By passing in the self argument to the shape constructor, we are telling Python that we do not want to create a new instance, rather we would like to use the current instance that self refers to. The second method contains the added functionality of our new class. The method name is the same as the method name in the super class, character. However, when we call the method with `cube1.printMyself()`, Python knows to use the method in `polyCube` and not the method from shape. This is called polymorphism. It allows us to replace, change, or add functionality to existing classes. By creating these hierarchies of objects, you can create pretty complex systems in neat, reusable classes that will keep your code clean and maintainable.

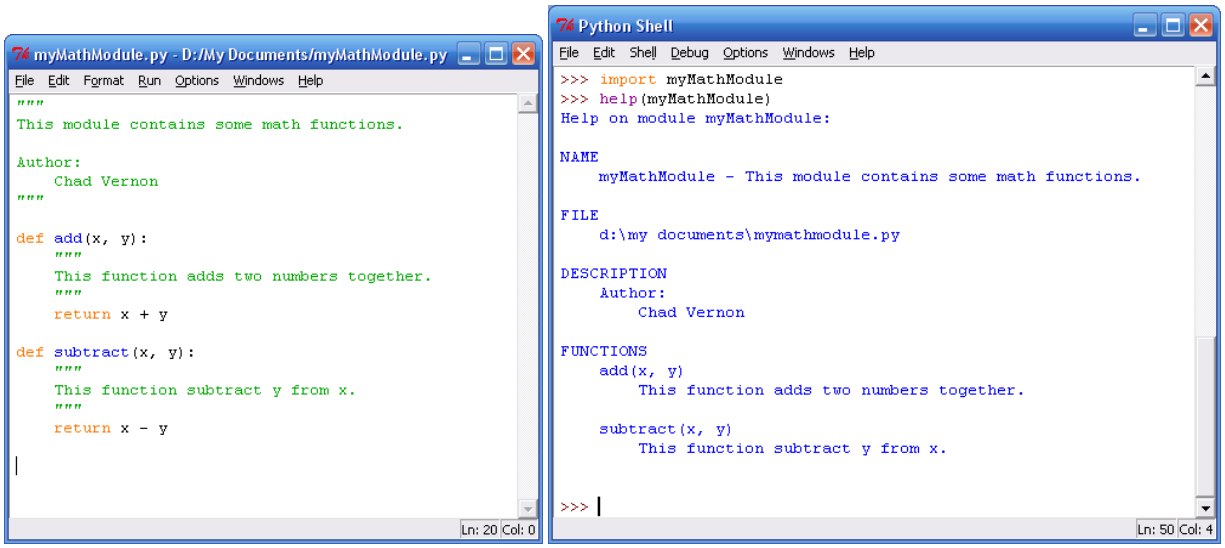
The previous example is an extremely simplified version of Maya's architecture. Nodes inherit off of other nodes to build a complex hierarchy. Below is part of Maya's object oriented node hierarchy:



Documenting Your Code

You have seen me putting in comments in most of my samples. These not only help other people understand your code but will also help you remember what you were thinking when you revisit your scripts months later to fix something. Comments not only explain your code, but also explain how to use your code or functions that you want others to use.

Python supports a method of creating documentation for your modules known as docstrings. Docstrings are strings written with triple quotes (“””) and can be placed at the top of modules and inside functions. When you type help(moduleName), Python will print some nice documentation using the docstrings that you specified in your module.



You should try to put docstrings in your code as much as possible. It will save you and your coworkers a lot of time down the road.

Sample Scripts

fileOrganizer.py:

```
import shutil
import os
import stat
import time

def run(directory):
    """
    Scans a directory of files and creates a folder for each day a file was last
modified.
    The script then copies the file into that directory.

    Parameters:
        directory - Directory to scan.
```

```
    Returns:
        Nothing.

    """
    # The os.walk function is used to traverse file directories
    for root, directories, files in os.walk(directory):
        # Loops through each file
        for image in files:
            # Create the full path to the file
            imageFilePath = os.path.join(root, image)
            # Get the date the file was last modified
            date = time.localtime(os.stat(imageFilePath)[stat.ST_MTIME])
            # Extract out the year, month and day from the date
            year = date[0]
            month = date[1]
            day = date[2]
            # Form the directory name as yyyymmdd
            dateDirectory = os.path.join(root, '%d%02d%02d' % (year, month, day))
            # Create the directory is it does not exist
            if not os.path.exists(dateDirectory):
                os.mkdir(dateDirectory)
                print 'Made directory: %s' % dateDirectory
            # Form the full path to where we want to copy the file
            newFilePath = os.path.join(root, '%s\\%s' % (dateDirectory, image))
            # Copy the file
            shutil.copy2(imageFilePath, newFilePath)
            print 'Copied %s to %s' % (imageFilePath, newFilePath)
```

Concepts used: functions, lists, for loops, conditional statements, string formatting, exceptions.

Python Outside of Maya Conclusion

We have now covered enough Python to begin scripting in Maya. Believe it or not, you have also learned about 75% of the syntax of most other scripting and programming languages. Most other languages have the same concepts like while and for loops, if/else statements, functions, lists, math operators, etc. There are however, many aspects of Python that we have not covered. If you really want to do some advanced scripts or design some advanced systems, I highly recommend studying more about Python in books and online.

Python in Maya

Maya's scripting commands come in the module package maya.cmds.

```
import maya.cmds
```

People usually use the shorthand

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

Or

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

You will notice that if you type in help(cmds), you do not get anything useful besides a list of function names. This is because Autodesk converted all of their MEL commands to Python procedurally. To get help with Maya's Python commands, you will need to refer to the Maya documentation.

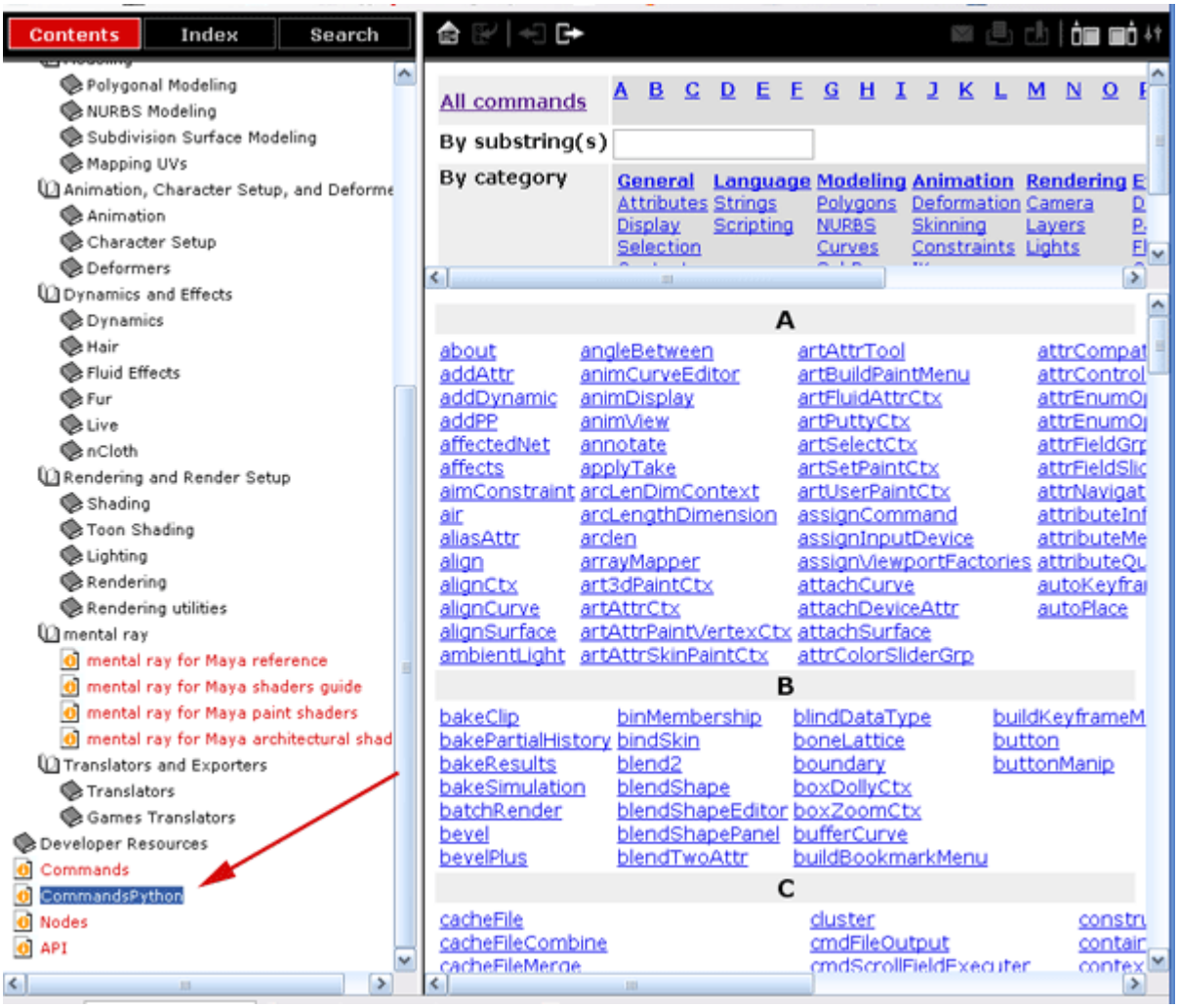


Figure 11 - Maya Python Command Documentation

There are so many commands that there is no reason to memorize them all. You will come to memorize many of the commands simply by using them a lot. How do you get started learning commands? Do what you are trying to accomplish with Maya's interface and look at the script editor. Most the actions you do with the Maya interface output what commands were called in the script editor. The only caveat is that the output is in MEL so we'll have to do a little bit of translating. Since all of the commands and arguments are the same between the MEL and Python commands, translating between the two is pretty easy.

The Maya Python Command Documentation

In this section, we will go over how to learn Maya's Python commands by studying the MEL output in the script editor when interacting with Maya. We will then decipher the MEL output and reconstruct the command using the Maya Python documentation.

Creating a polygon sphere outputs the following MEL command into the script editor:

```
polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -cuv 2 -ch 1;
```

MEL commands usually consist of the command name followed by several flags. In the above code, `polySphere` is the command, and each group of letters following a "`-`" is a flag. The numbers after each flag are arguments of their corresponding flag. For example, "`-r`" is a flag with an argument of 1, "`-sx`" is a flag with an argument of 20, "`-ax`" is a flag with 3 arguments: 0, 1, 0. Using this information, we can look up the command in the Maya Python command documentation and write its Python equivalent. Like I said earlier, commands in MEL and Python have the exact same name, look up "`polySphere`" in the Python documentation. It looks like this:

command (Python) *MEL version*

polySphere

No frames

Go to: Synopsis. Return value. Related. Flags. Python examples.

Synopsis

```
polySphere([axis=[float, float, float]], [constructionHistory=boolean],
[createUVs=int], [name=string], [object=boolean], [radius=linear],
[subdivisionsX=int], [subdivisionsY=int], [texture=boolean])
```

Note: Strings representing object names and arguments must be separated by commas. This is not depicted in the synopsis.

polySphere is undoable, queryable, and editable.

The sphere command creates a new polygonal sphere.

Return value

string[] Object name and node name.

In query mode, return type is based on queried flag.

Related

polyCone, polyCube, polyCube, polyCylinder, polyPlane, polyTorus

Flags

axis, constructionHistory, createUVs, name, object, radius, subdivisionsX, subdivisionsY, texture

Long name (short name)	*argument types*	Properties
axis(**ax**)	*[float, float, float]*	C Q E
This flag specifies the primitive axis used to build the sphere. Q: When queried, this flag returns a *float[3]*.		
radius(**r**)	*linear*	C Q E
This flag specifies the radius of the sphere. C: Default is 0.5. Q: When queried, this flag returns a *float*.		
subdivisionsX(**sx**)	*int*	C Q E
This specifies the number of subdivisions in the X direction for the sphere. C: Default is 20. Q: When queried, this flag returns an *int*.		
subdivisionsY(**sy**)	*int*	C Q E
This flag specifies the number of subdivisions in the Y direction for the sphere. C: Default is 20.		

Q: When queried, this flag returns an *int*.

createUVs(cuv) *int* C

This flag alows a specific UV mechanism to be selected, while creating the helix.
The valid values are 0, 1, or 2.
0 implies that no UVs will be generated (No texture to be applied).

1 implies UVs are created with pinched at poles

2 implies UVs are created with sawtooth at poles

For better understanding of these options, you may have to open the texture view window

C: Default is 2

texture(tx) *boolean* C

This flag is obsolete and will be removed in the next release. The -cuv/createUVs flag should be used instead.
Common flags

object(o) *boolean* C

Create the result, or just the dependency node (where applicable).

Common flags

name(n) *string* C

Give a name to the resulting node.

constructionHistory(ch) *boolean* C Q

Turn the construction history on or off (where applicable). If construction history is on then the corresponding node will be inserted into the history chain for the mesh. If construction history is off then the operation will be performed directly on the object.
Note: If the object already has construction history then this flag is ignored and the node will always be inserted into the history chain.

C Flag can appear in Create mode of command

E Flag can appear in Edit mode of command

Q Flag can appear in Query mode of command

M Flag can have multiple arguments, passed either as a tuple or a list.

Python examples

```
import maya.cmds as cmds

# Create a sphere, with 10 subdivisions in the X direction,
# and 15 subdivisions in the Y direction,
# the radius of the sphere is 20.
cmds.polySphere(sx=10, sy=15, r=20)

# Create a sphere, called "mySphere", on each direction there are 5 subdivisions.
cmds.polySphere( n='mySphere', sx=5, sy=5)
```

The documentation page contains all the information you need to work with the command. The Synopsis shows the function and all the possible arguments that can be passed in along with a description of what the command does. In this case, it creates a new polygonal sphere. The return value tells us what the function returns. The polySphere command returns a list containing two strings. The first element of the list will be the name of the transform of the new polygon sphere. The second string in the list will be the name of the polySphere node, which controls how the sphere is constructed.

```
>>> x = cmds.polySphere()
>>> print x
[u'pSphere1', u'polySphere1']
```

Notice that each string has a 'u' before it. This stands for Unicode string which is a type of string that you can ignore for now. Unicode strings help with international languages so just assume they are the same as normal strings.

Following the Return value section is a list of related Maya Commands. Following these links is a good way to learn about other commands. After the related commands is the Flags section. This section should really be called Arguments or Parameters; Flags are more of a MEL construct. The list of Flags (arguments) contains all the arguments that can be passed into the documented function. Each argument description contains the argument name, an abbreviated argument name, what kind of data you can pass into the argument, in what context the argument is valid, and a description of the argument. Take the radius argument for example. By passing this argument to the polySphere function, we can control the radius of the created sphere.

```
>>> x = cmds.polySphere(radius=2.5)
```

or the abbreviated form

```
>>> x = cmds.polySphere(r=2.5)
>>> print x
[u'pSphere2', u'polySphere2']
```

Personally, I tend to avoid the abbreviated form as I can never remember what all the abbreviations mean when I read my code. Using the full name is more typing and makes your code longer, but I find it easier to read.

The documentation is not always clear about what type of data is expected with an argument. For example, the documentation says that the radius argument expects some data of type `linear`. Usually by looking at the equivalent MEL command, you can figure out what to pass into the Python command. However, there are some cases where the documented format of the expected data is just really vague or cryptic. In these cases, if you can't figure out how to format the command, ask on a forum or mailing list.

After the list of arguments, there is usually an examples section that gives various usage examples of the given command.

Going back to our example MEL command:

```
polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -cuv 2 -ch 1;
```

we can see that each of the MEL flags corresponds to an argument in the Python function. By looking up the flags in the Python documentation, we can write the equivalent Python command:

cmds.polySphere(r=1, sx=20, sy=20, ax=(0, 1, 0), cuv=2, ch=1)

As a personal preference, I would write this command as

```
cmds.polySphere(radius=1, subdivisionsX=20, subdivisionsY=20, axis=(0, 1, 0),
createUVs=2, constructionHistory=True)
```

It's up to you on whether to use the abbreviated flags or not. Note that I also swapped the `ch=1` for `ch=True`. Refering to the documentation, the constructionHistory argument expects a Boolean value. Remember from the Booleans and Comparing Values section that all non-zero numbers are evaluated as `True`. I like to actually pass in the value `True` (or `False` if you want `False`) to these types of arguments just for my own preference.

You will also notice in the documentation the letters in the colored squares. The C, Q, and E stand for Create, Query, and Edit (You can ignore the M, I never pay attention to it). These letters tell you in what context an argument is valid. Many commands have different functionality depending on what context you are running the command in. In the previous example, we were creating a sphere, so all the arguments marked C were valid.

When you run a command in query mode, you can find information about an object created with the same command. You run a command in query mode by passing `query=True` as an argument.

```
>>> x = cmds.polySphere(radius=2.5)
>>> print cmds.polySphere(x[1], query=True, radius=True)
2.5
```

When you query a command, you pass in the object you want to query first, followed by your arguments. When in query mode, you pass a True or False to the argument you want to query regardless of what the expected type for that argument is documented as. You should only query one argument at a time. You'll notice that when you query a value, the value returned from the function may not be the same as what the documentation says is returned. In create mode, the polySphere command returns a list of 2 strings. In query mode, the return type depends on the value you are querying.

```
>>> x = cmds.polySphere()
>>> print cmds.polySphere(x[0], query=True, radius=True)
1.0
>>> print cmds.polySphere(x[0], query=True, axis=True)
[0.0, 1.0, 0.0]
```

Besides create and query modes, you can also run a command in edit mode. Edit mode lets you edit values of an existing node created with the command. You run a command in edit mode by passing in `edit=True` as an argument to the function.

```
>>> x = cmds.polySphere()
>>> cmds.polySphere(x[1], edit=True, radius=5)  # Change the radius to 5
```

You now know how to look up command syntax and decipher the documentation. You have just about all the knowledge you need now to write your own Maya scripts in Python. All you need now is to learn the various commands. A really good way to do that is to look at other people's scripts.

Sample Scripts

In most of the sample scripts, you will notice that I always put the code in functions. When you import a module, all of the code in the module gets executed. However code inside functions does not get run until the function is called. When writing scripts for Maya, it is good practice to structure your code as functions to be called by users. Otherwise, you may surprise your users by executing unwanted code when they import your modules.

lightIntensity.py

```
import maya.cmds as cmds

def changeLightIntensity(percentage=1.0):
    """
    Changes the intensity of each light in the scene by a percentange.

    Parameters:
        percentage - Percentange to change each light's intensity. Default value is 1.

    Returns:
        Nothing.

    """
    # The ls command is the list command.  It is used to list various nodes
    # in the current scene.  You can also use it to list selected nodes.
    lightsInScene = cmds.ls(type='light')

    # If there are no lights in the scene, there is no point running this script
    if not lightsInScene:
        raise RuntimeError, 'There are no lights in the scene!'

    # Loop through each light
    for light in lightsInScene:
        # The getAttr command is used to get attribute values of a node
        currentIntensity = cmds.getAttr('%s.intensity' % light)
        # Calculate a new intensity
        newIntensity = currentIntensity * percentage
        # Change the lights intensity to the new intensity
        cmds.setAttr('%s.intensity' % light, newIntensity)
        # Report to the user what we just did
        print 'Changed the intensity of light %s from %.3f to %.3f' % (light,
currentIntensity, newIntensity)
```

Concepts used: functions, lists, for loops, conditional statements, string formatting, exceptions.

To run this script, in the script editor type:

```
import samples.lightIntensity as lightIntensity
lightIntensity.changeLightIntensity(1.2)
```

renamer.py

```
import maya.cmds as cmds

def rename(name, nodes=None):
    """
    Renames a hierarchy of transforms using a naming string with '#' characters.
    If you select the root joint of a 3 joint chain and pass in 'C_spine##_JNT',
    the joints will be named C_spine01_JNT, C_spine02_JNT, and C_spine03_JNT.

    Parameters:
        name - A renaming format string.  The string must contain a consecutive
               sequence of '#' characters
        nodes - List of root nodes you want to rename.  If this argument is omitted,
               the script will use the currently selected nodes.

    Returns:
        Nothing.
    """

    # The variable "nodes" has a default value of None.  If we do not specify a value
    # for nodes, it will be None.  If this is the case, we will store a list of the
    # currently selected nodes in the variable nodes.
    if nodes == None:
        # The ls command is the list command. Get all selected nodes
        # of type transform.
        nodes = cmds.ls(sl=True, type='transform')

        # If nothing is selected, nodes will be None so we don't need
        # to continue with the script
        if nodes == None:
            raise RuntimeError, 'Select a root node to rename.'

    # Find out how many '#' characters are in the passed in name.
    numDigits = name.count('#')
    if numDigits == 0:
        raise RuntimeError, 'Name has no # sequence.'

    # We need to verify that all the '#' characters are in one sequence.
    substring = '#' * numDigits             # '#' * 3 is the same as '###'
    newsubstring = '0' * numDigits          # '0' * 3 is the same as '000'

    # The replace command of a string will replace all occurances of the first
    # argument with the second argument.  If the first argument is not found in
    # the string, the original string is returned.
    newname = name.replace(substring, newsubstring)

    # If the string returned after the replace command is the same as
    # the original string, it means that the sequence of '#' was not found in
    # our specified name.  This would happen if the '#' characters were not all
    # consecutive (e.g. 'my##New##Name').
    if newname == name:
        raise RuntimeError, 'Pound signs must be consecutive..'

    # Here we are creating a format string to use in our naming.  The number of digits
    # is determined by the number of consecutive '#' characters.
    # Example 'C_spine##_JNT' has 2 '#' characters.
    # In a chain of 3 joints, we would want to name the joints
    # C_spine01_JNT, C_spine02_JNT, C_spine03_JNT.
    # In a format string we would want to say 'C_spine%02d_JNT' % number.
    # We are creating the '%02d' part here.
    name = name.replace(substring, '%0' + str(numDigits) + 'd')
```

```
    # Start at number 1
    number = 1
    for node in nodes:
        # Loop through each selected node and rename its child hierarchy.
        number = renameChain(node, name, number)

def renameChain(node, name, number):
    """
    Recursive function that renames the passed in node to name % number.

    Parameters:
        node - The node to rename.
        name - A renaming format string.  The string must contain a
               consecutive sequence of '#' characters
        number - The number to use in the renaming.

    Returns:
        The next number to use in the renaming chain.
    """
    # Create the new name.  The variable name is a string like 'C_spine%02d_JNT'
    # so when we say name % number, it is the same as 'C_spine%02d_JNT' % number
    newName = (name % number)

    # The rename command renames a node.  Sometimes you have to be careful.
    # If you try to rename a node and there's already a node with the same name,
    # Maya will add a number to the end of your new name.  The returned string of
    # the rename command is the name that Maya actually assigns the node.
    node = cmds.rename(node, newName)

    # The listRelatives command is used to get a list of nodes in a dag
    # hierarchy.  You can get child nodes, parent nodes, shape nodes, or all
    # nodes in a hierarchy.  Here we are getting the child nodes.
    children = cmds.listRelatives(node, children=True, type='transform',
fullPath=True)

    # Since we renamed the current node, we increment the number for the next
    # node to be renamed.
    number += 1
    if children:
        for child in children:
            # We will call the renameChain function for each child of this node.
            number = renameChain(child, name, number)

    return number
```

Concepts used: functions, recursive functions, lists, for loops, conditionals, string formatting, exceptions.

To run this script, select the root joint of a joint chain and in the script editor type:

```
import samples.renamer as renamer
renamer.rename('C_tail##_JNT')
```

The renamer script uses a concept called recursive functions. A recursive function is a function that calls itself. Recursive function are useful when you are performing operations on data in a hierarchical graph, like Maya's DAG. In recursive functions, you must specify an ending condition or else the function will call itself in an infinite loop. In the above example, the function calls itself for each child node in the hierarchy. Since there are always a limited number of children in a Maya hierarchy, the recursive function is guaranteed to stop at some point.

blendShapes.py

```
import maya.cmds as cmds
import os

# The weights of these shapes are the product of the weights of the two listed shapes
COMBINATION_SHAPES = {
    'innerbrowraiser'                   : ('browsdown', 'browsup'),
    'outerbrowraiser'                   : ('browsup', 'nosewrinkler'),
    'nosewrinklesmile'                  : ('nosewrinkler', 'lipcornerpuller'),
    'eyesclosedsquint'                  : ('eyesclosed', 'squint'),
    'browsdowneyesclosedsquint'         : ('eyesclosedsquint', 'browsdown'),
    'nosewrinklerbrowsdown'             : ('nosewrinkler', 'browsdown'),
    'innerbrowraisernosewrinkler'       : ('innerbrowraiser', 'nosewrinkler'),
    'lipcornerdepressorpucker'          : ('lipcornerdepressor', 'lippucker'),
    'lipcornerpullerpucker'             : ('lipcornerpuller', 'lippucker'),
    'lipcornerpullerupperlipraiser'     : ('lipcornerpuller', 'upperlipraiser'),
    'eyeslookupleft'                    : ('eyeslookup', 'eyeslookleft'),
    'eyeslookupright'                   : ('eyeslookup', 'eyeslookright'),
    'eyeslookdownleft'                  : ('eyeslookdown', 'eyeslookleft'),
    'eyeslookdownright'                 : ('eyeslookdown', 'eyeslookright'),
}

def importShapes(directory):
    """
    Imports the shapes from directory.

    Parameters:
        directory - Directory that holds all the shape obj's.

    Returns:
        The created blendShape node, the neutral transform, and a list of all the
imported shape transforms.
    """
    # Get shape file paths
    shapes = []
    for root, dirs, files in os.walk(directory):
        for name in files:
            path = os.path.join(root, name)
            if name.startswith('neutral') and name.endswith('.obj'):
                # Put the neutral at the start of the list
                shapes.insert(0, path)
            elif name.endswith('.obj'):
                # Append the shape to the list
                shapes.append(path)

    # Create node to control all shape weights in 0-1 range.
    faceShapesTransform = cmds.createNode('transform', name='faceShapes')

    targets = []
    neutral = ''

    # Import shapes from blendshape folder and create blendShape node on neutral
    for shape in shapes:
        # Import the shape into a namespace called TEMP
        cmds.file(shape, i=True, namespace='TEMP', type='OBJ', options='mo=0')
        # Get the imported shape
        transform = cmds.ls('TEMP:*', type='transform')[0]
        # Rename the shape based off of the file name
        newName = os.path.basename(shape).split('.')[0]
        transform = cmds.rename(transform, newName)
        print 'Importing %s' % transform
```

```
        # Delete everything else in the namespace
        try:
            cmds.delete('TEMP:*')
        except:
            pass
        # Remove the temporary namespace since we don't need it anymore
        cmds.namespace(removeNamespace='TEMP')

        if transform.startswith('neutral'):
            # Create blendShape on neutral.
            neutral = transform
            # Unfreeze the normals, sometimes they get frozen in obj's
            cmds.polyNormalPerVertex(transform, unFreezeNormal=True)
            # Soften the normals on the neutral
            cmds.polySoftEdge(transform, angle=180)
            # Delete history on the neutral
            cmds.delete(transform, constructionHistory=True)
            # Create a blendshape on the neutral
            blendShape = cmds.blendShape(neutral, name='faceShapes_BLS')[0]
        else:
            # Store the target transform in the list
            targets.append(transform)

    # Sort the targets alphabetically
    targets.sort()

    # index will store the index of the target on the blendshape node
    index = 0

    # inbetweens will store all the inbetween targets and what index they belong to on
    # the blendshape node
    inbetweens = []

    for target in targets:
        if target.endswith('_100'):
            # Strip off the '_100' from the name
            target = cmds.rename(target, target[:-4])
            # Add the target to the blendshape
            addShapeToBlendShape(neutral, blendShape, target, index,
faceShapesTransform)
            index += 1
        else:
            # Add inbetweens later since the target needs to exist first
            inbetweens.append((target, index))

    # Add the inbetweens
    for node in inbetweens:
        # Called the weight at which the inbetween should turn on between 0-1
        onIndex = float('%.3f' % (int(node[0].split('_')[-1]) / 100.0))
        # Add the inbetween target to the blendshape
        addShapeToBlendShape(neutral, blendShape, node[0], node[1],
faceShapesTransform, onIndex)

    # Connect combination shapes
    for key in COMBINATION_SHAPES.keys():
        # Combination shapes are blendshapes on top of other blendshapes
        # When two or more shapes are turned on, they trigger another shape
        # to be turned on
        # Create a multiplyDivide node to multiply two weights together
        mdn = cmds.createNode('multiplyDivide', name='%s_MDN' % key)
        # Hook up the inputs to the multiplyDivide node
        cmds.connectAttr('%s.%s' % (faceShapesTransform, COMBINATION_SHAPES[key][0]),
'%s.input1X' % mdn)
```

```
        cmds.connectAttr('%s.%s' % (faceShapesTransform, COMBINATION_SHAPES[key][1]),
'%s.input2X' % mdn)
        print 'Connecting combination shape %s driven by the product of %s and %s' %
(key, COMBINATION_SHAPES[key][0], COMBINATION_SHAPES[key][1])
        # Connect the output of the multiplyDivide node to the blendShape control
        cmds.connectAttr('%s.outputX' % mdn, '%s.%s' % (faceShapesTransform, key))


def addShapeToBlendShape(neutral, blendShape, mesh, index, faceShapesTransform,
onIndex=1.0):
    """
    Adds a shape to the blendShape node.

    Parameters:
        neutral         - The neutral mesh
        blendShape      - The blendShape node to add shapes to.
        mesh            - Target mesh
        index           - BlendShape target index
        faceShapesTransform  - Node to add control attributes to.
        onIndex         - Index at which the target is fully on.  Normally this
                          is 1.0.  For inbetweens, this is between 0 and 1.

   Returns:
        Nothing
    """

    if onIndex == 1.0:
        # Add a new shape
        print 'Adding shape %s to %s at index %d' % (mesh, blendShape, index)
        cmds.blendShape(blendShape, edit=True, target=(neutral, index, mesh, onIndex))
    else:
        # Add an inbetween shape
        print 'Adding inbetween shape %s to %s at index %d with an onWeight of %.3f' %
(mesh, blendShape, index, onIndex)
        cmds.blendShape(blendShape, edit=True, target=(neutral, index, mesh, onIndex),
inBetween=True)

    if onIndex == 1.0:
        # Add attribute to the control node to control this blendshape target
        cmds.addAttr(faceShapesTransform, longName=mesh, shortName=mesh,
attributeType='float', keyable=True, min=0.0, max=1.0)
        # Connect the new control attribute to the blendshape weight
        cmds.connectAttr('%s.%s' % (faceShapesTransform, mesh), '%s.%s' % (blendShape,
mesh))
    cmds.delete(mesh)
    # Sometimes Maya crashes when it runs out of memory so free some up by clear the
    # undo queue.
    cmds.flushUndo()
```

Concepts used: functions, lists, for loops, conditionals, string formatting, dictionaries.

To run this script, in the script editor type:

```
import samples.blendShapes as blendShapes
blendShapes.importShapes(r"C:\pathToBlendShapesFolder")
```

Calling MEL from Python

Most Maya commands (MEL commands) have been implemented into the maya.cmds module. There are still some cases where MEL must be used because Maya does not fully incorporate Python is all aspects of its architecture.

MEL can be called from Python with the maya.mel module:

```
import maya.cmds as cmds
selection = cmds.ls( sl=True )

import maya.mel as mel
selection = mel.eval( "ls -sl" )
```

We can also invoke python from MEL

```
string $list[] = python( "['a', 'b', 'c']" );
size( $list );
// Result: 3 //
```

To source/execute existing MEL scripts in Python:

```
import maya.mel as mel
mel.eval('source "myScript.mel"')
mel.eval('source "myOtherScript.mel"')
mel.eval('mySourcedFunction(1)')
```

Maya Python Standalone Applications

Maya provides the maya.standalone module for creating command-line applications. These applications allow us to create and run operations without opening Maya's interface. Maya Python standalone applications are run with the *mayapy* interpreter.

Run "mayapy" from the Run… dialog in Windows, or run "mayapy" from a command line:

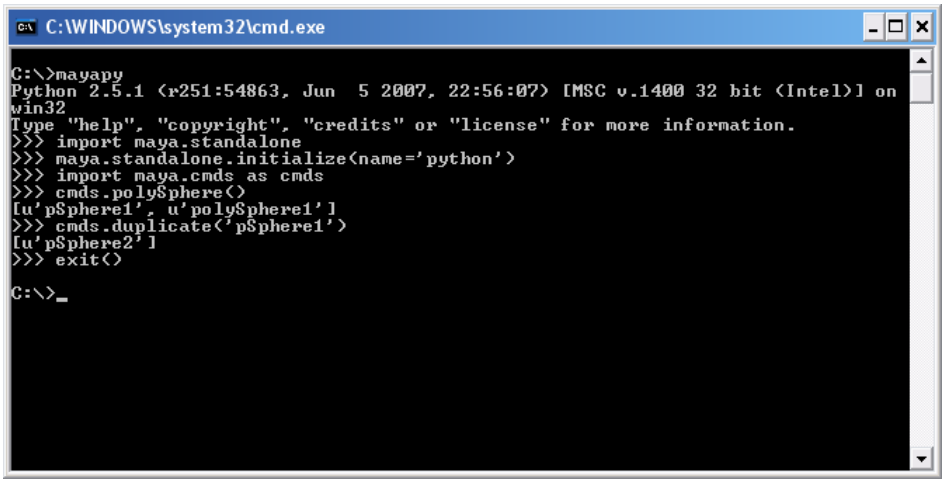


Figure 12 - Running Maya from the Command Line

To open a file in batch mode, you would run:

```
mayapy X:\file.py
```

You can also write scripts to run operations on Maya files in batch mode.

Example: Open a Maya file, assign the default shader to all meshes, and save the scene.

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

def assignDefaultShader(fileToOpen):
    # Start Maya in batch mode
    maya.standalone.initialize(name='python')

    # Open the file with the file command
    cmds.file(fileToOpen, force=True, open=True)

    # Get all meshes in the scene
    meshes = cmds.ls(type="mesh", long=True)
    for mesh in meshes:
        # Assign the default shader to the mesh by adding the mesh to the
        # default shader set.
        cmds.sets(mesh, edit=True, forceElement='initialShadingGroup')

    # Save the file
    cmds.file(save=True, force=True)
```

In order to run this script, you need to use the mayapy Python interpreter and not the normal Python interpreter. In addition to stand alone scripts, I recommend reading about the `optparse` module which would allow you to start the mayapy interpreter and call the script all in one line from the command line.

Conclusion

After reading these notes, you should have a good understanding of how to write and run Python scripts inside and outside of Maya. There is still plenty to learn though. There are hundreds of Maya commands and plenty of useful modules out there to experiment with. I recommend continuing your Python education by looking through the references included with these notes, reading other peoples' scripts, and experimenting with your own scripts. Good luck!