Chapter 1

Maya Command Engine and User Interface

CHAPTER OUTLINE

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00001-6

BY THE END OF THIS CHAPTER, YOU WILL BE ABLE TO:

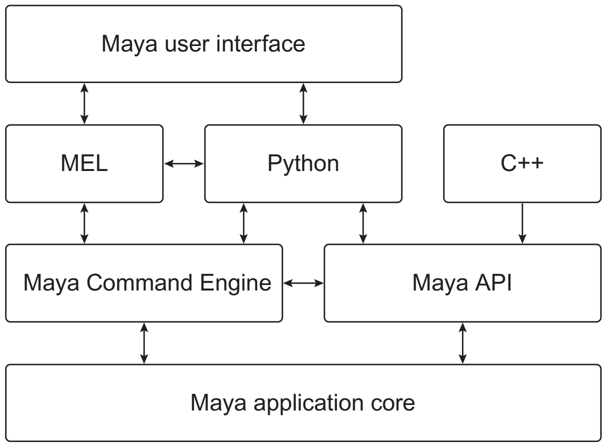
- Compare and contrast the four Maya programming interfaces.
- Use the Command Line and Script Editor to execute Python commands.
- Create a button in the Maya GUI to execute custom scripts.
- Describe how Python interacts with Maya commands.
- Define nodes and connections.
- Describe Maya's command architecture.
- Learn how to convert MEL commands into Python.
- Locate help for Python commands.
- Compare and contrast command arguments and flag arguments.
- Define the set of core Python data types that work with Maya commands.
- Compare and contrast the three modes for using commands.
- Identify the version of Python that Maya is using.
- Locate important Python resources online.

To fully understand what can be done with Python in Maya, we must first discuss how Maya has been designed. There are several ways that users can interact with or modify Maya. The standard method is to create content using Maya's graphical user interface (GUI). This interaction works like any other software application: Users press buttons or select menu items that create or modify their documents or workspaces. Despite how similar Maya is to other software, however, its underlying design paradigm is unique in many ways. Maya is an open product, built from the ground up to be capable of supporting new features designed by users. Any Maya user can modify or add new features, which can include a drastic redesign of the main interface or one line of code that prints the name of the selected object.

In this chapter, we will explore these topics as you begin programming in Python. First, we briefly describe Maya's different programming options and how they fit into Maya's user interface. Next, we jump into Python by exploring different means of executing Python code in Maya. Finally, we explore some basic Maya commands, the primary means of modifying the Maya scene.

INTERACTING WITH MAYA

Although the focus of this book is on using Python to interact with Maya, we should briefly examine all of Maya's programming interfaces to better understand why Python is so unique. Autodesk has created four different



■ **FIGURE 1.1** The architecture of Maya's programming interfaces.

programming interfaces to interact with Maya, using three different programming languages. Anything done in Maya will use some combination of these interfaces to create the result seen in the workspace. Figure 1.1 illustrates how these interfaces interact with Maya.

Maya Embedded Language

Maya Embedded Language (MEL) was developed for use with Maya and is used extensively throughout the program. MEL scripts fundamentally define and create the Maya GUI. Maya's GUI executes MEL instructions and Maya commands. Users can also write their own MEL scripts to perform most common tasks. MEL is relatively easy to create, edit, and execute, but it is also only used in Maya and has a variety of technical limitations. Namely, MEL has no support for object-oriented programming. MEL can only communicate with Maya through a defined set of interfaces in the Command Engine (or by calling Python). We will talk more about the Command Engine later in this chapter.

Python

Python is a scripting language that was formally introduced to Maya in version 8.5. Python can execute the same Maya commands as MEL using Maya's Command Engine. However, Python is also more robust than MEL because it is an object-oriented language. Moreover, Python has existed since 1980 and has an extensive library of built-in features as well as a large community outside of Maya users.

C++ Application Programming Interface

The Maya C++ application programming interface (API) is the most flexible way to add features to Maya. Users can add new Maya objects and features that can execute substantially faster than MEL alternatives. However, tools developed using the C++ API must be compiled for new versions of Maya and also for each different target platform. Because of its compilation requirements, the C++ API cannot be used interactively with the Maya user interface, so it can be tedious to test even small bits of code. C++ also has a much steeper learning curve than MEL or Python.

Python API

When Autodesk introduced Python into Maya, they also created wrappers for many of the classes in the Maya C++ API. As such, developers can use much of the API functionality from Python. The total scope of classes accessible to the Python API has grown and improved with each new version of Maya. This powerful feature allows users to manipulate Maya API objects in ordinary scripts, as well as to create plug-ins that add new features to Maya.

In this book, we focus on the different uses of Python in Maya, including commands, user interfaces, and the Python API. Before we begin our investigation, we will first look at the key tools that Maya Python programmers have at their disposal.

EXECUTING PYTHON IN MAYA

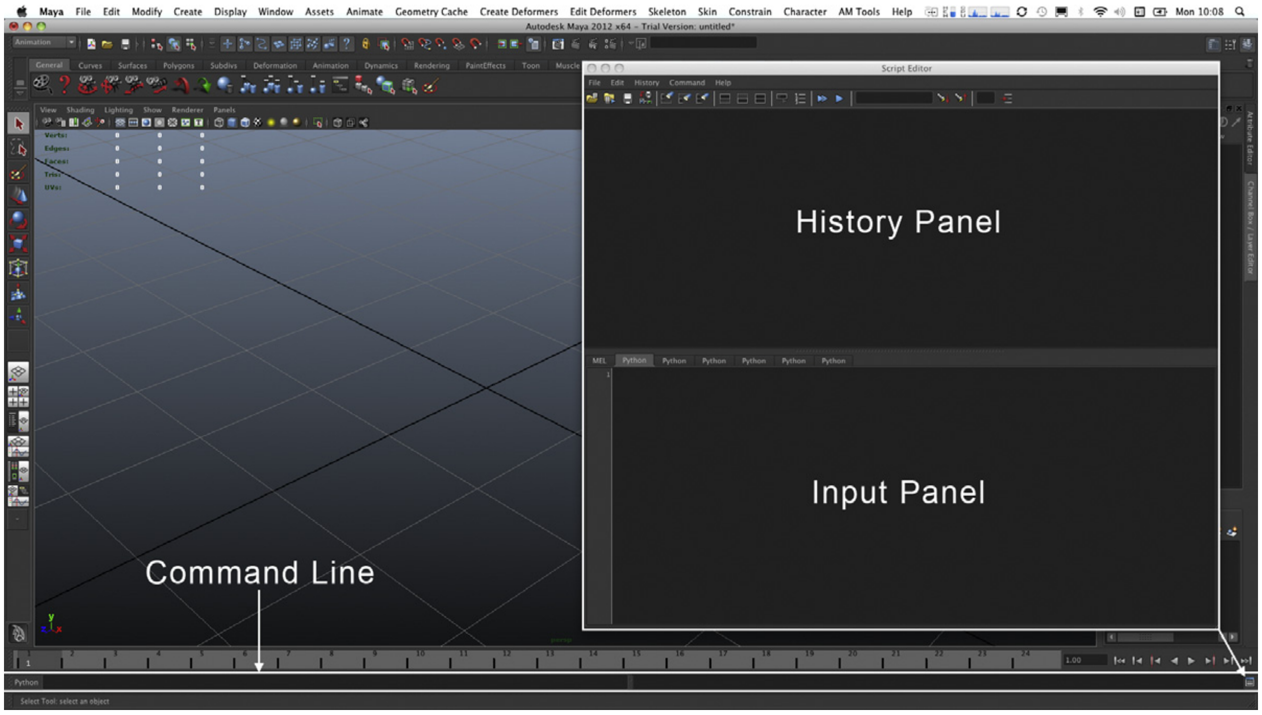
Maya has many tools built into its GUI that allow users to execute Python code. Before you begin programming Python code in Maya, you should familiarize yourself with these tools so that you know not only what tool is best for your current task, but also where to look for feedback from your scripts.

Command Line

The first tool of interest is the Command Line. It is located along the bottom of the Maya GUI. You can see the Command Line highlighted in Figure 1.2.

The Command Line should appear in the Maya GUI by default. If you cannot see the Command Line, you can enable it from the Maya main menu by selecting **Display → UI Elements → Command Line**.

The far left side of the Command Line has a toggle button, which says “MEL” by default. If you press this button it will display “Python.”



■ **FIGURE 1.2** Programming interfaces in the Maya GUI.

The language displayed on this toggle button tells Maya which scripting language to use when executing commands entered in the text field immediately to the right of the button. The right half of the Command Line, a gray bar, displays the results of the commands that were entered in the text field. Let's create a polygon sphere using the Command Line.

1. Switch the Command Line button to "Python." The button is located on the left side of the Command Line.
2. Click on the text field in the Command Line and enter the following line of text.

```
import maya.cmds;
```

3. Press **Enter**.
4. Next enter the following line of code in the text field.

```
maya.cmds.polySphere();
```

5. Press **Enter**. The above command will create a polygon sphere object in the viewport and will print the following results on the right side of the Command Line.

```
# Result: [u'pSphere1', u'polySphere1']
```

You can use the Command Line any time you need to quickly execute a command. The Command Line will only let you enter one line of code at a time though, which will not do you much good if you want to write a complicated script. To perform more complex operations, you need the Script Editor.

Script Editor

One of the most important tools for the Maya Python programmer is the Script Editor. The Script Editor is an interface for creating short scripts to interact with Maya. The Script Editor (shown on the right side in Figure 1.2) consists of two panels. The top panel is called the History Panel and the bottom panel is called the Input Panel. Let's open the Script Editor and execute a command to make a sphere.

1. Open a new scene by pressing **Ctrl + N**.
2. Open the Script Editor using either the button located near the bottom right corner of Maya's GUI, on the right side of the Command Line (highlighted in Figure 1.2), or by navigating to **Window → General Editors → Script Editor** in Maya's main menu. By default the Script Editor displays two tabs above the Input Panel. One tab says "MEL" and the other tab says "Python."
3. Select the Python tab in the Script Editor.
4. Click somewhere inside the Input Panel and type the following lines of code.

```
import maya.cmds;
maya.cmds.polySphere();
```

5. When you are finished press the **Enter** key on your numeric keypad. If you do not have a numeric keypad, press **Ctrl + Return**.

The **Enter** key on the numeric keypad and the **Ctrl + Return** shortcut are used only for executing code when working in the Script Editor. The regular **Return** key simply moves the input cursor to the next line in the Input Panel. This convention allows you to enter scripts that contain more than one line without executing them prematurely.

Just as in the Command Line example, the code you just executed created a generic polygon sphere. You can see the code you executed in the History Panel, but you do not see the same result line that you saw when using the Command Line. In the Script Editor, you will only see a result line printed when you execute a single line of code at a time.

6. Enter the same lines from step 4 into the Input Panel, but do not execute them.

7. Highlight the second line with your cursor by triple-clicking it and then press **Ctrl + Return**. The results from the last command entered should now be shown in the History Panel.

   ```
   # Result: [u'pSphere2', u'polySphere2']
   ```

Apart from printing results, there are two important things worth noting about the previous step. First, highlighting a portion of code and then pressing **Ctrl + Return** will execute only the highlighted code. Second, highlighting code in this way before executing it prevents the contents of the Input Panel from emptying out.

Another useful feature of the Script Editor is that it has support for marking menus. Marking menus are powerful, context-sensitive, gesture-based menus that appear throughout the Maya application. If you are unfamiliar with marking menus in general, we recommend consulting any basic Maya user's guide.

To access the Script Editor's marking menu, click and hold the right mouse button (**RMB**) anywhere in the Script Editor window. If you have nothing selected inside the Script Editor, the marking menu will allow you to quickly create new tabs (for either MEL or Python) as well as navigate between the tabs. As you can see, clicking the **RMB**, quickly flicking to the left or right, and releasing the **RMB** allows you to rapidly switch between your active tabs, no matter where your cursor is in the Script Editor window. However, the marking menu can also supply you with context-sensitive operations, as in the following brief example.

1. Type the following code into the Input Panel of the Script Editor, but do not execute it.

   ```
   maya.cmds.polySphere()
   ```

2. Use the left mouse button (**LMB**) to highlight the word `polySphere` in the Input Panel.
3. Click and hold the **RMB** to open the Script Editor's marking menu. You should see a new set of options in the bottom part of the marking menu.
4. Move your mouse over the **Command Documentation** option in the bottom of the marking menu and release the **RMB**. Maya should now open a web browser displaying the help documentation for the `polySphere` command.

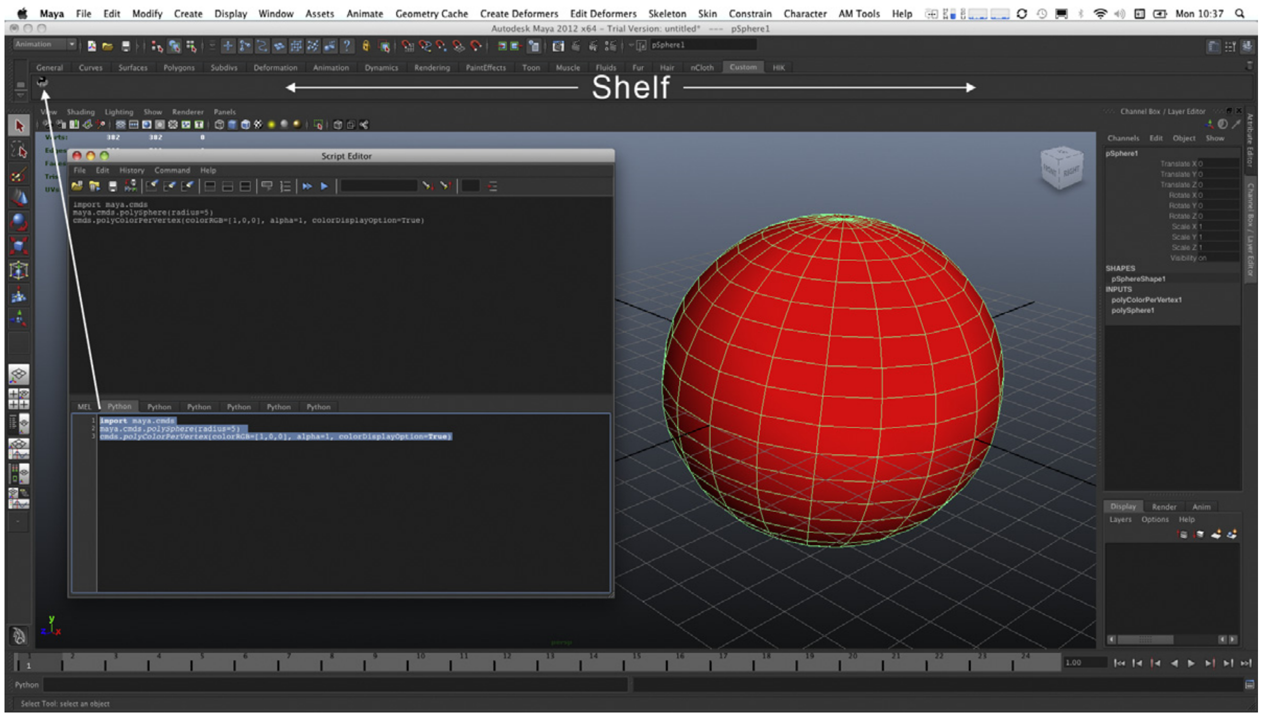
As you can see, the Script Editor is a very useful tool not only for creating and executing Python scripts in Maya, but also for quickly pulling up information about commands in your script. We will look at the command documentation later in this chapter.

At this point, it is worth mentioning that it can be very tedious to continually type common operations into the Script Editor. While the Script Editor does allow you to save and load scripts, you may want to make your script part of the Maya GUI. As we indicated earlier, clicking GUI controls in Maya simply calls commands or executes scripts that call commands. Another tool in the Maya GUI, the Shelf, allows you to quickly make a button out of any script.

Maya Shelf

Now that you understand how to use the Command Line and the Script Editor, it is worth examining one final tool in the Maya GUI that will be valuable to you. Let's say you write a few lines of code in the Script Editor and you want to use that series of commands later. Maya has a location for storing custom buttons at the top of the main interface, called the Shelf, which you can see in Figure 1.3. If you do not see the Shelf in your GUI layout, you can enable it from Maya's main menu using the **Display → UI Elements → Shelf** option.

You can highlight lines of code in the Script Editor or Command Line and drag them onto the Shelf for later use with the middle mouse button


Shelf
Script Editor
File Edit History Command Help
MEL Python Python Python Python Python Python

```
import maya.cmds
maya.cmds.polySphere(radius=5)
cmds.polyColorPerVertex(colorRGB=[1,0,0], alpha=1, colorDisplayOption=True)
```

```
import maya.cmds
maya.cmds.polySphere(radius=5)
cmds.polyColorPerVertex(colorRGB=[1,0,0], alpha=1, colorDisplayOption=True)
```

■ **FIGURE 1.3** The Shelf.

(**MMB**). In the following example, you will create a short script and save it to the Shelf.

1. Type in the following code into the Script Editor, but do not execute it (when executed, this script will create a polygon sphere and then change the sphere's vertex colors to red).

   ```
   import maya.cmds;
   maya.cmds.polySphere(radius=5);
   maya.cmds.polyColorPerVertex(
       colorRGB=[1,0,0],
       colorDisplayOption=True
   );
   ```

2. Click the Custom tab in the Shelf. You can add buttons to any shelf, but the Custom shelf is a convenient place for users to store their own group of buttons.
3. Click and drag the **LMB** over the script you typed into the Script Editor to highlight all of its lines.
4. With your cursor positioned over the highlighted text, click and hold the **MMB** to drag the contents of your script onto the Shelf.
5. If you are using Maya 2010 or an earlier version, a dialog box will appear. If you see this dialog box, select "Python" to tell Maya that the script you are pasting is written using Python rather than MEL.
6. You will now see a new button appear in your Custom tab. Left-click on your new button and you should see a red sphere appear in your viewport as in Figure 1.3. If you are in wireframe mode, make sure you enter shaded mode by clicking anywhere in your viewport and pressing the number **5** key.

You can edit your Shelf, including tabs and icons, by accessing the **Window** → **Settings/Preferences** → **Shelf Editor** option from the main Maya window. For more information on editing your Shelf, consult the Maya documentation or a basic Maya user's guide. Now that you have an understanding of the different tools available in the Maya GUI, we can start exploring Maya commands in greater detail.

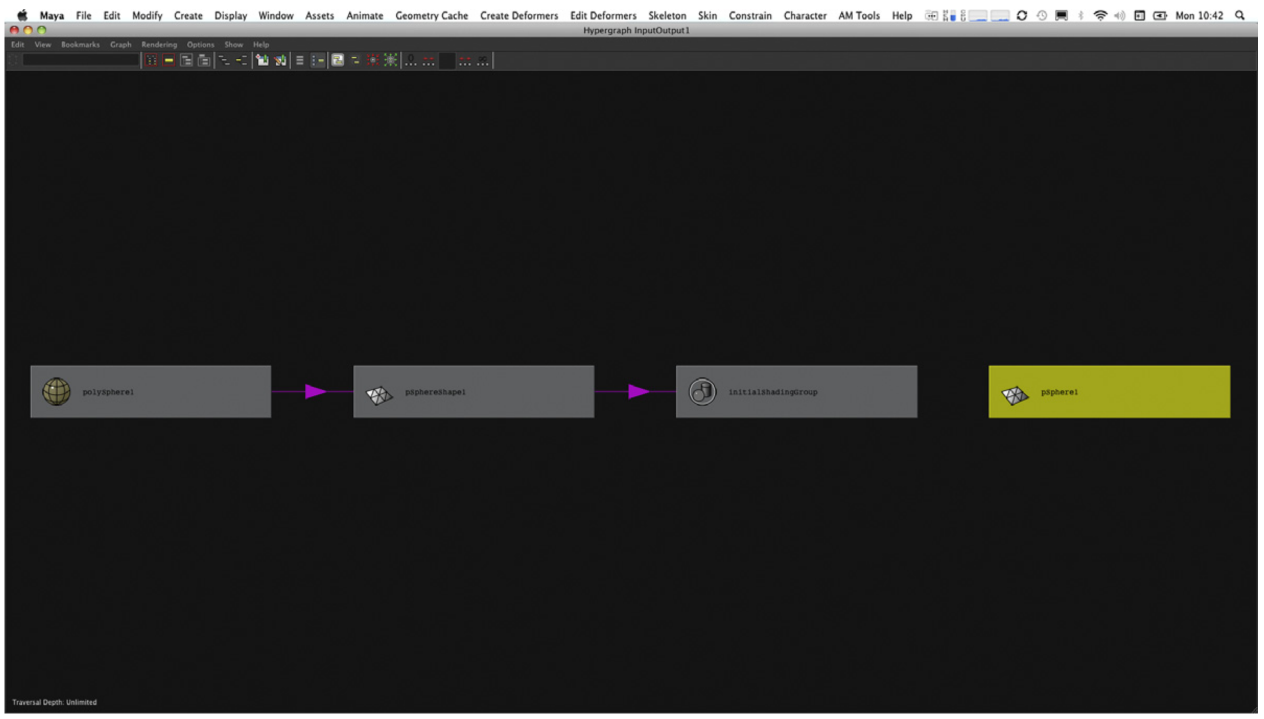
MAYA COMMANDS AND THE DEPENDENCY GRAPH

To create a polygonal sphere with Python, the `polySphere` command must be executed in some way or other. The `polySphere` command is part of the Maya Command Engine. As we noted previously, the Maya Command Engine includes a set of commands accessible to both MEL and Python.

As we briefly discussed previously, Maya is fundamentally composed of a core and a set of interfaces for communicating with that core (see Figure 1.1). The core contains all the data in a scene and regulates all operations on these data—creation, destruction, editing, and so on. All of the data in the core are represented by a set of objects called *nodes* and a series of *connections* that establish relationships among these nodes. Taken together, this set of relationships among nodes is called the *Dependency Graph* (DG).

For example, the polygon sphere object you created earlier returned the names of two nodes when you created it: a node that describes the geometry of the sphere and a **transform** node that determines the configuration of the sphere shape in space. You can see information on nodes in an object's network using the Attribute Editor (**Window → Attribute Editor** in the main menu) or as a visual representation in the Hypergraph (**Window → Hypergraph: Connections** in the main menu). Because this point is so important, it is worth looking at a brief example.

1. If you no longer have a polygon sphere in your scene, create one.
2. With your sphere object selected, open the Hypergraph displaying connections by using the **Window → Hypergraph: Connections** option from the main menu.
3. By default, the Hypergraph should display the connections for your currently selected sphere as in Figure 1.4. If you do not see anything,



■ **FIGURE 1.4** The Hypergraph.

then select the option **Graph → Input and Output Connections** from the Hypergraph window's menu.

As you can see, a default polygon sphere consists of four basic nodes connected by a sequence of arrows that show the flow of information. The first node in the network is a **polySphere** node, which contains the parameters and functionality for outputting spherical geometry (e.g., the radius, the number of subdivisions, and so on). In fact, if you highlight the arrow showing the connection to the next node, a **shape** node, you can see what data are being sent. In this case, the **polySphere** node's **output** attribute is piped into the **inMesh** attribute of the **shape** node.

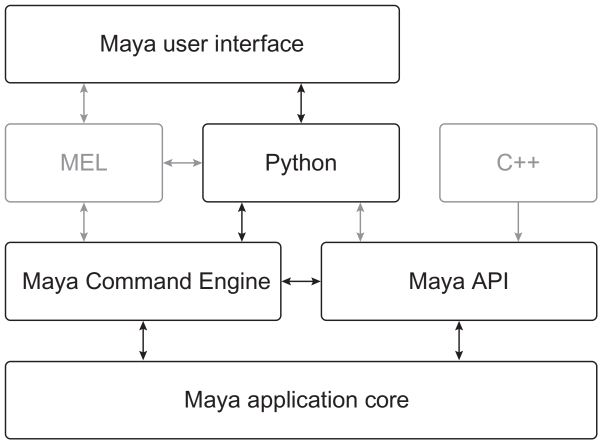
If you were to delete the construction history of this polygonal sphere (**Edit → Delete by Type → History** from the main menu), the **polySphere** node would disappear and the sphere's geometry would then be statically stored in the **shape** node (**pSphereShape1** in Figure 1.4). In short, if the **polySphere** node were destroyed, its mesh information would be copied into the **pSphereShape** node, and you would no longer be able to edit the radius or number of subdivisions parametrically; you would have to use modeling tools to do everything by hand.

While you can also see that information is piped from the **shape** node into a **shadingGroup** node (to actually render the shape), there is a node that appears to be floating on its own (**pSphere1** in Figure 1.4). This separate node is a special kind of object, a **transform** node, which describes the position, scale, and orientation of the polygonal sphere's geometry in space. The reason why this node is not connected is because it belongs to a special part of the DG, called the *Directed Acyclic Graph* (DAG). For right now, it suffices to say that the DAG essentially describes the hierarchical relationship of objects that have **transform** nodes, including what nodes are their parents and what transformations they inherit from their parents.

The Maya DG is discussed in greater detail in Chapter 11 in the context of the Maya API, yet this principle is critical for understanding how Maya works. We strongly recommend consulting a Maya user guide if you feel like you need further information in the meantime.

Although Maya is, as we pointed out, an open product, the data in the core are closed to users at all times. Autodesk engineers may make changes to the core from one version to another, but users may only communicate with the application core through a defined set of interfaces that Autodesk provides.

One such interface that can communicate with the core is the Command Engine. In the past, Maya commands have often been conflated with



■ **FIGURE 1.5** Python's interaction with the Maya Command Engine.

MEL. Indeed, commands in Maya may be issued using MEL in either scripts or GUI elements like buttons. However, with the inclusion of Python scripting in Maya, there are now two different ways to issue Maya commands, which more clearly illustrates the distinction.

Figure 1.5 highlights how Python interacts with the Maya Command Engine. While Python can use built-in commands to retrieve data from the core, it can also call custom, user-made commands that use API interfaces to manipulate and retrieve data in the core. These data can then be returned to a scripting interface via the Command Engine. This abstraction allows users to invoke basic commands (which have complex underlying interfaces to the core) via a scripting language.

MEL has access to over 1,000 commands that ship with Maya and has been used to create almost all of Maya's GUI. While Python has access to nearly all the same commands (and could certainly also be used to create Maya's GUI) there is a subset of commands unavailable to Python. The commands unavailable to Python include those specifically related to MEL or that deal with the operating system. Because Python has a large library of utilities that have grown over the years as the language has matured outside of Maya, this disparity is not a limitation.

Maya has documentation for all Python commands so it is easy to look up which commands are available. In addition to absent commands mentioned previously, there are some MEL scripts that appear in MEL command documentation as though they were commands. Because these are scripts rather than commands, they do not appear in the Python command documentation and are not directly available to Python. Again, this absence is also not a limitation, as it is possible to execute MEL scripts with Python when needed. Likewise, MEL can call Python commands and scripts when required.[1]

Another important feature of the Maya Command Engine is how easy it is to create commands that work for MEL and Python. Maya was designed so that any new command added will be automatically available to both MEL and Python. New commands can be created with the Maya C++ API or the Python API. Now that you have a firmer understanding of how Maya commands fit into the program's architecture, we can go back to using some commands.

INTRODUCTION TO PYTHON COMMANDS

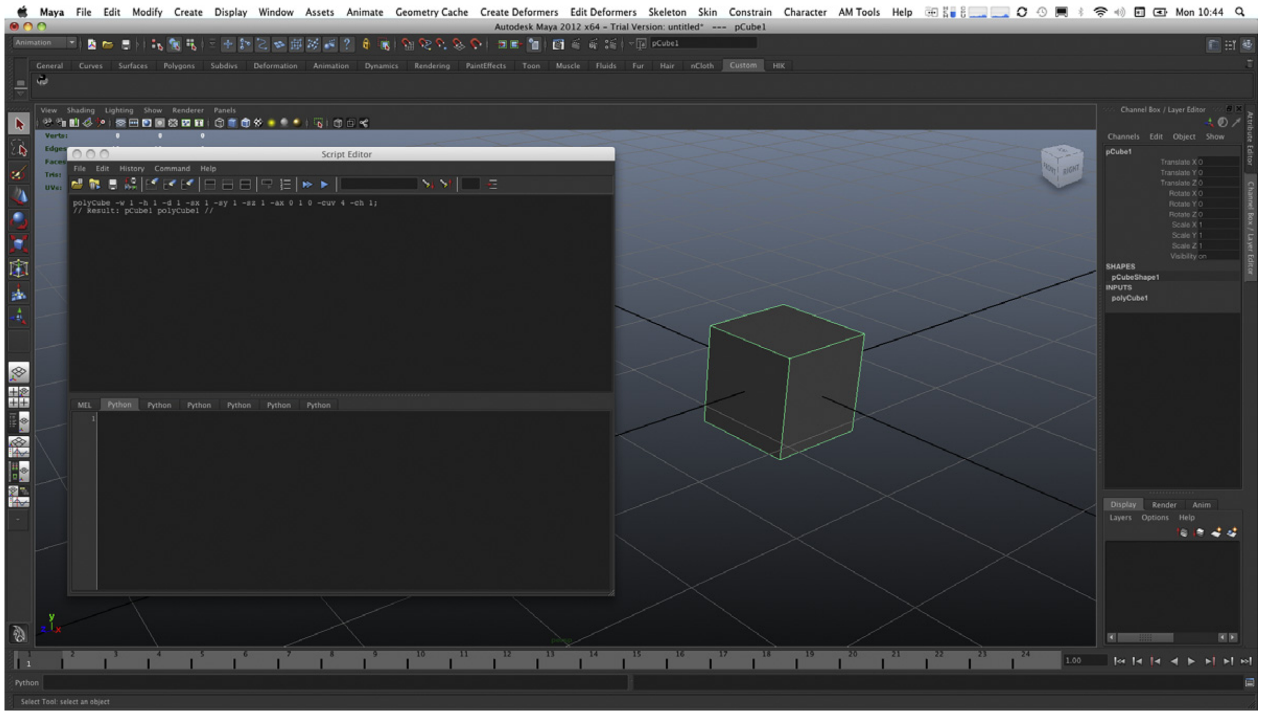
Let's return to Maya and open up the Script Editor. As discussed earlier in this chapter, the top panel of the Script Editor is called the History Panel. This panel can be very useful for those just learning how to script or even for advanced users who want to figure out what commands are being executed. By default, the History Panel will echo (print) most Maya commands being executed. You can also make the History Panel show all commands being executed, including commands called by the GUI when you press a button or open a menu. To see all commands being executed, select the **History → Echo All Commands** option from the Script Editor's menu. While this option can be helpful when learning, it is generally inadvisable to leave it enabled during normal work, as it can degrade Maya's performance. Right now, we will go through the process of creating a cube and look at the results in the History Panel (Figure 1.6).

1. In the menu for the Script Editor window, select **Edit → Clear History** to clear the History Panel's contents.
2. In the main Maya window, navigate to the menu option **Create → Polygon Primitives → Cube**.
3. Check the History Panel in the Script Editor and confirm that you see something like the following results.

```
polyCube -w 1 -h 1 -d 1 -sx 1 -sy 1 -sz 1 -ax 0 1 0 -cuv 4 -ch 1;
// Result: pCube1 polyCube1 //
```

The first line shown is the `polyCube` MEL command, which is very similar to the `polySphere` command we used earlier in this chapter. As you can see,

[1]MEL can call Python code using the `python` command. Python can call MEL code using the `eval` function in the maya.mel module. Note that using the `python` command in MEL executes statements in the namespace of the __main__ module. For more information on namespaces and modules, see Chapter 4.



```
polyCube -w 1 -h 1 -d 1 -sx 1 -sy 1 -sz 1 -ax 0 1 0 -cuv 4 -ch 1;
// Result: pCube1 polyCube1 //
```

■ **FIGURE 1.6** The results of creating a polygon cube.

a MEL command was called when you selected the **Cube** option in the **Polygon Primitives** menu. That MEL command was displayed in the Script Editor's History Panel.

Because Maya's entire interface is written with MEL, the History Panel always echoes MEL commands when using the default Maya interface. Custom user interfaces could call the Python version of a command, in which case the History Panel would display the Python command.

This problem is not terribly troublesome for Python users though. It does not take much effort to convert a MEL command into Python syntax, so this feature can still help you learn which commands to use. The following example shows what the `polyCube` command looks like with Python.

```
import maya.cmds;
maya.cmds.polyCube(
    w=1, h=1, d=1, sx=1, sy=1, sz=1,
    ax=(0, 1, 0), cuv=4, ch=1
);
```

If you execute these lines of Python code they will produce the same result as the MEL version. However, we need to break down the Python version of the command so we can understand what is happening. Consider the first line:

```
import maya.cmds;
```

This line of code imports a Python module that allows you to use any Maya command available to Python. There is only one module that holds all Maya commands and you only need to import it once per Maya session. Once it is in memory you don't need to import it again (we only have you reimport it for each example in case you're picking the book back up after a break from Maya). We will discuss modules in greater depth in Chapter 4. The next line of code is the Python command.

```
maya.cmds.polyCube(
    w=1, h=1, d=1, sx=1, sy=1, sz=1,
    ax=(0, 1, 0), cuv=4, ch=1
);
```

As you can see, the name of the command, `polyCube`, is prefixed by the name of the module, `maya.cmds`. The period between them represents that this command belongs to the Maya commands module. We then supply the command several flag arguments inside of parentheses. A key-value pair separated by the equals sign, such as `w=1`, represents the name and value for the flag argument, and each of these pairs is separated by a comma.

Each flag may be added using a shorthand abbreviation or long version of the flag name. Although many Maya programmers tend to use the shorthand flag names in their code, it can make the code more difficult to read later. In the previous example, the command is using the shorthand flags so it is hard to understand what they mean. Here is the same version of the command with long flag names.

```
maya.cmds.polyCube(
    width=1,
    height=1,
    depth=1,
    subdivisionsX=1,
    subdivisionsY=1,
    subdivisionsZ=1,
    axis=(0, 1, 0),
    createUVs=4,
    constructionHistory=1
);
```

The long names are easier to read and so it can be good practice to use them when scripting. Code that is easier to read can be much easier to work with—especially if you or a coworker has to make any changes several months later! You may now be wondering how to find the long flag names in the future.

1. Type the following lines into the Script Editor and press **Ctrl + Return** to execute them.

```
import maya.cmds;
print(maya.cmds.help('polyCube'));
```

2. Look for the results in the History Panel, which should look like the following lines.

```
Synopsis: polyCube [flags] [String...]
Flags:
   -e -edit
   -q -query
  -ax -axis                  Length Length Length
 -cch -caching               on|off
  -ch -constructionHistory   on|off
 -cuv -createUVs             Int
   -d -depth                 Length
   -h -height                Length
   -n -name                  String
 -nds -nodeState             Int
   -o -object                on|off
  -sd -subdivisionsDepth     Int
  -sh -subdivisionsHeight    Int
  -sw -subdivisionsWidth     Int
  -sx -subdivisionsX         Int
  -sy -subdivisionsY         Int
  -sz -subdivisionsZ         Int
  -tx -texture               Int
   -w -width                 Length

Command Type: Command
```

As you can see, the result first displays the command for which help was requested—`polyCube` in this case. The following items in brackets, `[flags]` and `[String...]`, show MEL syntax for executing the command. In MEL, the command is followed by any number of flag arguments and then any number of command arguments. We'll differentiate these two items momentarily.

Next, the output shows the list of flags for the command, displaying the short name on the left, followed by the long name in the middle column. Each flag is prefixed by a minus symbol, which is required to indicate a flag in MEL syntax, but which you can ignore in Python. To the very right of each flag name is the data type for each argument, which tells us what kind of value each flag requires.

We can see how flags work with the `polyCube` command. Consider the following example.

```
import maya.cmds;
maya.cmds.polyCube();
```

Executing this command causes Maya to create a polygon cube with default properties. The parentheses at the end of the command basically indicate that we want Maya to do something—execute a command in this case. Without them, the command will not execute. We will discuss this topic further in Chapter 3 when we introduce functions. For now, it suffices to say that any command arguments we wish to specify must be typed inside of the parentheses, as in the following alternative example.

```
import maya.cmds;
maya.cmds.polyCube(name='myCube', depth=12.5, height=5);
```

If you execute the previous lines, Maya will create a polygon cube named "myCube" with a depth of 12.5 units and a height of 5 units. The first flag we set, `name`, is a string, as indicated in the help results. A *string* is a sequence of letters and numbers inside of quotation marks, and is used to represent a word or words. Immediately afterward is a comma before the next flag, `depth`. We specify that the depth should be the decimal number 12.5. Such values are listed as type Length in the help results. Last, we provided the `height` flag and supplied a value of 5. In this case, we used the long names of the flags, but we could also have used the short ones to do the same thing.

```
import maya.cmds;
maya.cmds.polyCube(n='myCube', d=12.5, h=5);
```

Looking at the help results, you can see that the `axis` flag takes three decimal numbers. To specify this kind of argument in Python, we use what is called a tuple. A *tuple* is basically a sequence of objects inside of parentheses, separated by commas. The following lines show an example of the same command using a tuple to specify a different axis.

```
import maya.cmds;
maya.cmds.polyCube(
    name='myCube',
    depth=12.5,
    height=5,
    axis=(1,1,0)
);
```

FLAG ARGUMENTS AND PYTHON CORE OBJECT TYPES

As you have seen, most Maya Python commands have flags, which allow you to change the default settings of the command being executed. Each flag argument must be passed a value. A flag's value can be one of several different built-in Python types. Table 1.1 lists Python's core object types that are used by Maya commands.

Note that Table 1.1 is not a complete list of Python core object types—there are many others that you may use for other purposes in your scripts.

Table 1.1 Python Core Object Types Used by Maya Commands

Type	Examples

Numbers	`1`
	`−5`
	`3.14159`
	`9.67`

Strings	`"Maya"`
	`'ate'`
	`"my dog's"`
	`"""homework"""`

Lists	`[1, "horse", 'town']`
Tuples	`(1, "two", 'three')`

Booleans	`True`
	`False`
	`1`
	`0`

However, the core object types in this list are the only ones that Maya *commands* have been designed to use, so we may ignore the others for now. Other Python data types are discussed in Chapter 2. Let's focus for now on the five types in this list.

Numbers

Maya commands expecting Python numbers will accept any real number. Examples could include integer as well as decimal numbers, which correspond to int/long and float/double types, respectively, in other languages.

Strings

The string type is any sequence of letters or numbers enclosed in single quotation marks, double quotation marks, or a matching pair of triple quotation marks of either type. For instance, "boat", "house", and "car" are equivalent to 'boat', 'house', and 'car' as well as to """boat""", """house""", and """car""". However, the string "3" is different from the number object 3. Strings are typically used to name objects or parameters that are accessible from the Maya user interface.

Lists

A list is a sequence of any number of Python objects contained within the bracket characters **[** and **]**. A comma separates each object in the list. Any Python object may be in a list, including another list!

Tuples

The Python tuple is very similar to the list type except that it is not mutable, which means it cannot be changed. We discuss mutability in greater detail in Chapter 2. Tuples are contained inside of ordinary parentheses, (and).

Booleans

A Boolean value in Python can be the word True or False (which must have the first letter capitalized), or the numbers 1 and 0 (which correspond to the values True and False, respectively). These values are typically used to represent states or toggle certain command modes or flags.

Flag = Object Type

To find out what type of object a command flag requires, you can use the `help` command. As you saw earlier in this chapter it will give you a list of the command flags and what type of value they require. The argument type is not an option—you must pass a value of the required type or you will get an error. Using the `polyCube` command as an example, let's look at its `width` flag and pass it correct and incorrect argument types.

1. Create a new scene by pressing **Ctrl + N**.
2. Execute the Maya `help` command for the `polyCube` command in the Script Editor:

```
import maya.cmds;
print(maya.cmds.help('polyCube'));
```

3. Look for the `width` flag in the results displayed in the History Panel and find its argument type on the right side:

```
-w -width Length
```

As you can see, the `width` flag requires a Length type argument, as shown to the right of the flag name. This is technically not a Python type but we can deduce that `Length` means a number, so we should pass this flag some sort of number. If the number needed to be a whole number, the flag would specify `Int` to the right of the flag instead of `Length`. We can therefore also deduce that the flag may be passed a decimal number in this case. Let's first pass a correct argument.

4. Type the following command into the Script Editor and press **Ctrl + Return** to execute it.

```
maya.cmds.polyCube(width=10);
```

You should see the following result in the Script Editor's History Panel.

```
# Result: [u'pCube1', u'polyCube1'] #
```

The result lets us know that the command succeeded and also shows that the command returned a Python list containing the names of two new nodes that have been created to make our cube object: "pCube1" (a **transform** node) and "polyCube1" (a **shape** node). Now, let's see what happens when we intentionally supply the `width` flag with the wrong data type.

5. Type the following command into the Script Editor and press **Ctrl + Return** to execute it.

```
maya.cmds.polyCube(width='ten');
```

This time the command returns an error.

```
# Error: TypeError: file <maya console> line 1: Invalid
arguments for flag 'width'. Expected distance, got str #
```

The error tells you that the argument for the `width` flag was incorrect and it expected a distance value. Even though the `help` command showed the `width` flag needed a Length type, Maya is now calling it a distance type. This can be confusing at first but most of the time it is very clear what the flag argument requires simply by looking at the flag in context. The `help` command does not describe what each flag does, but you can get more detailed descriptions using the Python Command Reference, which we will examine shortly.

COMMAND MODES AND COMMAND ARGUMENTS

Maya commands often have more than one mode in which they can work. Some commands may be available to use in create mode, edit mode, and/or query mode, while certain flags may only be available in certain modes.

Create Mode

Most commands at least have a create mode. This mode allows users to create new objects in the scene and specify any optional parameters. By default, the `polyCube` command operates in create mode.

1. Create a new Maya scene.
2. Execute the following lines in the Script Editor to create a new cube.

```
import maya.cmds;
maya.cmds.polyCube();
```

Note that you do not have to do anything special to execute commands in create mode. Leave the cube in your scene for the next steps.

Edit Mode

Another mode that many commands support is edit mode. This mode allows users to edit something the command has created.

3. Execute the following line in the Script Editor to change the cube's width.

```
maya.cmds.polyCube('pCube1', edit=True, width=10);
```

As you can see, you specified that the command should operate in edit mode by setting the `edit` flag with a value of True. In edit mode, you were able to change the width of the object named "pCube1" to a value of 10. It is worth mentioning that some flags in MEL do not require an argument, such as the `edit` flag (see `help` output previously). Such flags, when invoked from Python, simply require that you set some value (True) to indicate their presence.

Another important point worth noting is the syntax for operating in edit and query modes. The first argument we pass to the command is called a *command argument*, and specifies the name of the node on which to operate. As we saw in the `help` output previously, MEL syntax expects command arguments to follow flag arguments, while Python requires the opposite order. The reason for Python's syntax requirement will be discussed in greater detail in Chapter 3. Leave the cube in the scene for the next step.

Query Mode

The last mode that many commands support is query mode. This mode permits users to request information about something the command has already created.

4. Execute the following line in the Script Editor to print the cube's width.

```
maya.cmds.polyCube('pCube1', query=True, width=True);
```

The result in the History Panel should display something like the following line.

```
# Result: 10.0 #
```

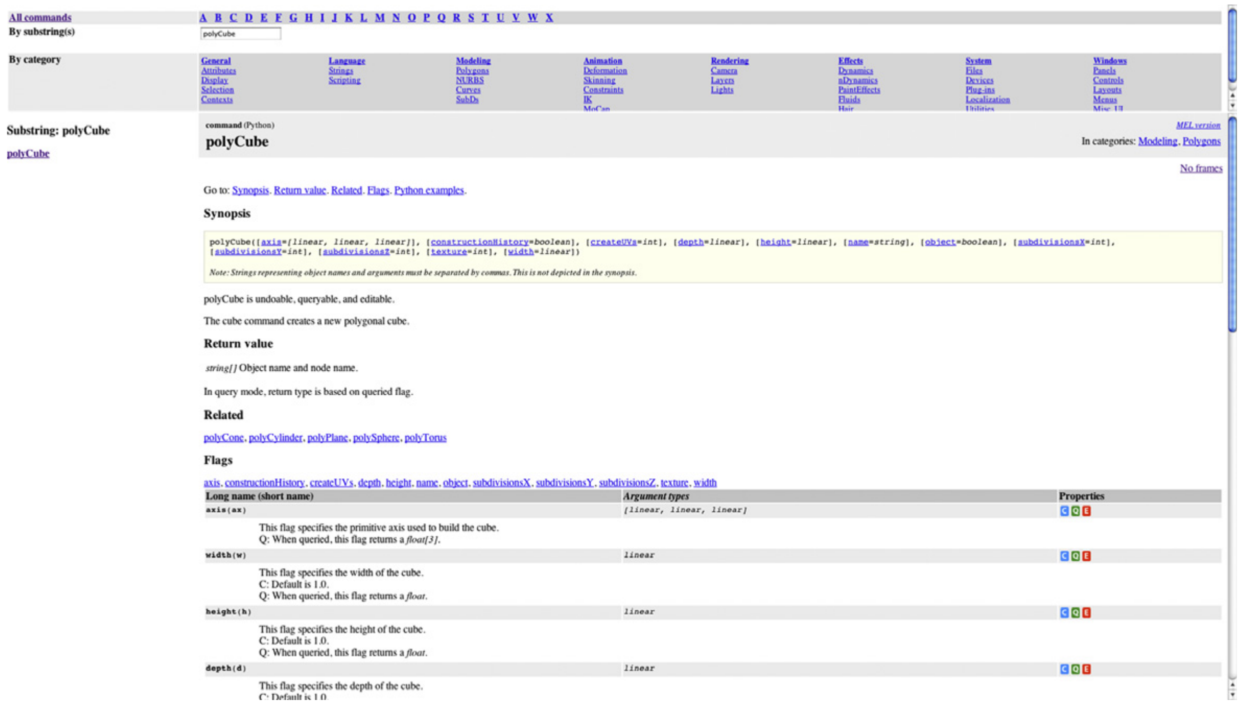
As with edit mode, query mode requires that you specify a command argument first and then set the `query` flag with a value of True. Another point worth noting is that, although the width flag normally requires a decimal number (when being invoked from create or edit mode), you simply pass it a value of True in query mode. The basic idea is that in this case, the Command Engine is only interested in whether or not the flag has been set, and so it will not validate the value you are passing it.

As you can see, these three modes allowed you to create an object, change it, and finally pull up information about its current state. We also noted at the outset of this section that some flags are only compatible with certain command modes. While the `help` command will not give you this information, the Command Reference documentation will.

PYTHON COMMAND REFERENCE

Another place you can get more help on a command is from the Maya help documents. These documents detail every Python command available. The Python Command Reference is shown in Figure 1.7. Let's browse the Python Command Reference to find more information on the `polyCube` command.

1. In the main Maya window select the menu item **Help → Python Command Reference**.
2. At the top of the page is a text field. Click in the search field and enter the word polyCube.
3. The page will update to show you only the `polyCube` command. Select the `polyCube` command from the list under the label "Substring: polyCube". Clicking this link will show you detailed information for the `polyCube`



command (Python)

polyCube

In categories: Modeling, Polygons

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

Synopsis

```
polyCube([axis=[linear, linear, linear]], [constructionHistory=boolean], [createUVs=int], [depth=linear], [height=linear], [name=string], [object=boolean], [subdivisionsX=int],
[subdivisionsY=int], [subdivisionsZ=int], [texture=int], [width=linear])
```

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

polyCube is undoable, queryable, and editable.

The cube command creates a new polygonal cube.

Return value

string[] Object name and node name.

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

Related

polyCone, polyCylinder, polyPlane, polySphere, polyTorus

Flags

axis, constructionHistory, createUVs, depth, height, name, object, subdivisionsX, subdivisionsY, subdivisionsZ, texture, width

Long name (short name)	*Argument types*	Properties
`axis(ax)`	*[linear, linear, linear]*	C Q E
This flag specifies the primitive axis used to build the cube. Q: When queried, this flag returns a *float[3]*.		
`width(w)`	*linear*	C Q E
This flag specifies the width of the cube. C: Default is 1.0. Q: When queried, this flag returns a *float*.		
`height(h)`	*linear*	C Q E
This flag specifies the height of the cube. C: Default is 1.0. Q: When queried, this flag returns a *float*.		
`depth(d)`	*linear*	C Q E
This flag specifies the depth of the cube. C: Default is 1.0.		

■ **FIGURE 1.7** Python Command Reference.

command. As you can see, the Command Reference documents break up information for all commands into different sections, which we will now look at in more detail.

Synopsis

The synopsis provides a short description of what the command does. In this case the synopsis states:

polyCube is undoable, queryable, and editable.
The cube command creates a new polygonal cube.

Return Value

The return value section describes what the command will return when it is executed. In this case the documentation states:

string[] *Object name and node name.*

This description shows us that it returns a list of string type objects, which will be the name of the object (**transform** node) and the (**polyCube**) node that were created.

Related

This section can help you with finding commands that are similar to the command at which you are looking. For the `polyCube` command, this section lists other commands for creating primitive polygon objects:

polyCone, polyCylinder, polyPlane, polySphere, polyTorus

Flags

For the `polyCube` command, the `axis` flag is listed first. It shows a short description of what the flag does and then it lists the argument type to pass to it. The documentation shows the following text:

[linear, linear, linear]

The command requires a Python list (or tuple) type and that list should hold three real numbers. If int is not specified for an argument type, then the argument may be a decimal number, though integer numbers are still valid. In this case, if we were to define the argument for the flag it could be something like [1.00, 0, 0].

As we noted earlier, the documentation also displays icons to represent the command mode(s) with which each flag is compatible.

- *C (Create)*: The flag can appear in the create mode.
- *E (Edit)*: The flag can appear in the edit mode.
- *Q (Query)*: The flag can appear in the query mode.
- *M (Multiuse)*: The flag can be used multiple times.

In MEL, multiuse flags are invoked by appending them multiple times after a command. In Python, however, you can only specify a flag once. As such, Python requires that you pass a tuple or list argument for a multiuse flag, where each element in the tuple or list represents a separate use of the flag.

Python Examples

This section can be very useful for those just learning how to script with Python or those learning how to work with Maya using Python. Here you can find working examples of the command in use. Some example sections can have several different samples to help you understand how the command works. The example for the `polyCube` command in the documents shows you how to create a polygon cube and also how to query an existing cube's width.

PYTHON VERSION

One final point that is worth discussing is how to locate which version of Python your copy of Maya is using. Python has been integrated into Maya since version 8.5, and each new version of Maya typically integrates the newest stable version of Python as well. Since newer versions of Python will have new features, you may want to investigate them. First, find out what version of Python your version of Maya is using.

1. Open up the Script Editor and execute the following lines.

```
import sys;
print(sys.version);
```

You should see a result print in the Script Editor's History Panel that looks something like the following lines.

```
2.6.4 (r264:75706, Nov 3 2009, 11:26:40)
[GCC 4.0.1 (Apple Inc. build 5493)]
```

In this example, our copy of Maya is running Python version 2.6.4.

PYTHON ONLINE DOCUMENTATION

Once you know what version of Python you are running, you can look up the Python documentation online. The Python web site is located at *http://www.python.org/*. If you navigate to the Documentation section, you should see documentation for multiple versions of Python.

As you learn to program using Python you might also be interested in downloading Python for your operating system if it does not already include it. You can find the latest versions of Python at the Python web site. If you plan to write Python scripts that interact with Maya, it is advisable that you install the same version that Maya is using. For the most part, many versions of Python that you will see in Maya are almost identical. However, a newer version of Python, 3.x, may break a few things that work in older versions.

If you choose to install Python for your operating system, you will be able to use a Python interpreter, such as the Python IDLE, which acts just like the Maya Script Editor but for your operating system. This can be useful for creating tools outside of Maya that communicate with Maya using Python. Moreover, you could write tools using Python that have nothing to do with Maya, yet may be helpful for your project's organization or interaction with other software like MotionBuilder.

CONCLUDING REMARKS

In this chapter you have learned how Maya commands and Python work within Maya's architecture. We have introduced a few methods of entering Maya commands with Python to modify Maya scenes. We have also explained how to look up help and read the Python Command Reference documentation and how to find information about your version of Python. In the chapters that immediately follow, we will further explain some of the underlying mechanics and syntax of Python and then start creating more complicated scripts to use in Maya.