Introduction: Welcome to Maya Python

Imagine you are creating animations for a character in Maya. As you are creating animations for this character, you find that you are repeating the following steps:

- Reference a character.
- Import a motion capture animation.
- Set the frame range.
- Reference the background scene.
- Configure the camera.

If you are working on a large production, with 10 to 50 animators, these simple steps can pose a few problems. If we break down this process, there are many good places for tools:

- First, animators need to look up the correct path to the character. This path may change at any given time, so having animators handpick this file could result in picking the wrong file. A simple tool with a list of characters can make this task quick and reliable.
- Next, you would want a tool to organize and manage importing the motion capture data correctly onto the character's controls. This tool could also set the frame range for the animation and move the camera to the correct location in the process.
- The last step, referencing the correct background scene, could easily take a minute each time an animator manually searches for the correct file. You could create another simple tool that shows a list of all the backgrounds from which to pick.

Hopefully you can see that such tools would save time, make file selection more accurate, and let the animators focus on their creative task. The best way to create such tools is by using one of Maya's built-in scripting languages—particularly Python.

Python is a scripting language that was originally developed outside of Maya, and so it offers a powerful set of features and a huge user base. In Maya 8.5, Autodesk added official support for Python scripting. The inclusion of this language built upon Maya's existing interfaces for programming (the MEL scripting language and the C++ API). Since Maya Embedded Language (MEL) has been around for years, you might wonder why Python even matters. A broader perspective quickly reveals many important advantages:

- *Community:* MEL has a very small user base compared to Python because only Maya developers use MEL. Python is used by all kinds of software developers and with many types of applications.
- *Power:* Python is a much more advanced scripting language and it allows you to do things that are not possible in MEL. Python is fully object-oriented, and it has the ability to communicate effortlessly with both the Maya Command Engine and the C++ API, allowing

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00019-3

you to write both scripts and plug-ins with a single language. Even if you write plug-ins in C++, Python lets you interactively test API code in the Maya Script Editor!

- *Cross-platform:* Python can execute on any operating system, which removes the need to recompile tools for different operating systems, processor architectures, or software versions. In fact, you do not need to compile at all!
- *Industry Standard:* Because of Python's advantages, it is rapidly being integrated into many other content-creation applications important for entertainment professionals. Libraries can be easily shared between Maya and other applications in your pipeline, such as MotionBuilder.

PYTHON VERSUS MEL

There are many reasons to use Python as opposed to MEL, but that doesn't mean you have to give up on MEL completely! If your studio already has several MEL tools there is no reason to convert them if they are already doing the job. You can seamlessly integrate Python scripts into your development pipeline with your current tools. Python can call MEL scripts and MEL can call Python scripts. Python is a deep language like C++ but its syntax is simple and very easy to pick up.

Nonetheless, Python handles complex data more gracefully than MEL. MEL programmers sometimes try to imitate complex data structures, but doing so often requires messy code and can be frustrating to extend. Since Python is object-oriented and it allows nested variables, it can handle these situations more easily. Moreover, Python can access files and system data much faster than MEL, making your tools more responsive for artists in production. Programmers also have many more options in Python than in MEL. Since Python has been around much longer, you might find another user already created a module to help Python perform the task you need to do.

If you don't understand some of the language used yet, don't worry. This book is here to help you understand all of these concepts and emerge production-ready!

COMPANION WEB SITE

This book is intended to be read alongside our companion web site, which you can access at *http://maya-python.com*.

The companion web site contains downloads for the example projects to which we refer throughout the book, as well as a host of useful links and supplemental materials.

NOTES ON CODE EXAMPLES AND SYNTAX

You will encounter many code examples throughout this book. Unfortunately, because this book will not only be printed, but is also available in a variety of e-book formats, we have no guarantees where line breaks will occur. While this problem is a nonissue for many programming languages, including MEL, Python is very particular about line breaks and whitespace. As such, it's worth briefly noting how Python handles these issues, and how we have chosen to address them throughout the text. Thereafter, we can move on to our first example!

Whitespace

Many programming languages are indifferent to leading whitespace, and instead use mechanisms like curly braces ({ and }) to indicate code blocks, as in the following hypothetical MEL example, which would print numbers 1 through 5.

```
for (int $i=0; $i<5; $i++)
{
    int $j = $i+1;
    print($j+"\n");
}
```

In this example, the indentation inside the block is optional. The example could be rewritten like the following lines, and would produce the same result.

```
for (int $i=0; $i<5; $i++)
{
int $j = $i+1;
print($j+"\n");
}
```

Python, on the other hand, uses leading whitespace to structure blocks. An example such as this one would look like the following lines in Python.

```
for i in range(5):
    j = i+1
    print(j)
```

While Python does not care exactly how you indent inside your blocks, they must be indented to be syntactically valid! The standard in the Python community is to either use one tab or four spaces per indentation level.

Because of how this book is formatted, you may not be able to simply copy and paste examples from an e-book format. We try to mitigate this problem by providing as many code examples as possible as downloads on the companion web site, which you can safely copy and paste, and leave only short examples for you to transcribe.

Line Breaks

Python also has some special rules governing line breaks, which is particularly important for us since we have no way to know exactly how text in this book will wrap on different e-book hardware.

Most programming languages allow you to insert line breaks however you like, as they require semicolons to indicate where line breaks occur in code. For example, the following hypothetical MEL example would construct the sentence "I am the very model of a modern major general." and then print it.

```
string $foo = "I am the very" +
"model of a modern" +
"major general");
print($foo);
```

If you tried to take the same approach in Python, you would get a syntax error. The following approach would not work.

```
foo = 'I am the very' +
'model of a modern' +
'major general'
print(foo)
```

In most cases in Python, you must add a special escape character (\) at the end of a line to make it continue onto the following line. You would have to rewrite the previous example in the following way.

```
foo = 'I am the very' + \
'model of a modern' + \
'major general'
print(foo)
```

There are some special scenarios where you can span onto further lines without an escape sequence, such as when inside of parentheses or brackets. You can read more about how Python handles line breaks online in Chapter 2 of Python Language Reference. We'll show you where you can find this document in Chapter 1.

Our Approach

Because of Python's lexical requirements, it is difficult to craft code examples that are guaranteed to be properly displayed for all of our readers. As such, our general approach is to indicate actual line endings in print with the optional semicolon character (**;**) as permitted by Python. We trust you to recognize them as line endings where there should be a return carriage and to adjust your indentation if necessary.

For instance, though we can be reasonably certain that the previous examples are short enough to have appeared properly for all our readers, we

would have rewritten them in the following way throughout the remainder of this book.

```
for i in range(5):
      j = i+1;
      print(j);
foo = 'I am the very model of a modern major general';
print(foo);
```

Since Python allows but does not require the semicolon, we avoid using it in our actual code examples on the companion web site, though we do show them in the text for readers who do not have the code examples on their computers in front of them.

Quotation Marks

It is also worth mentioning that some of our code examples throughout the book make use of double quotation marks ("), some make use of single quotation marks ('), and some make use of triple sequences of either type (""" or '''). While it should be clear which marks we are using when reading a print copy, double quotation marks may not copy and paste correctly. In short, make sure you double check any code you try to copy and paste from the book.

Comments

Finally, it is worth noting here that, like other programming languages, Python allows you to insert comments in your code. Comments are statements that only exist for the developer's reference, and they are not evaluated as part of a program. One way to create a comment is to simply write it in as a string literal. Though we will talk more about strings later, the basic idea is that you can wrap a standalone statement in quotation marks on its own line and it becomes a comment. In the following code snippet, the first line is a comment describing what the second line does.

```
'''Use the print() function to print the sum of 5 and 10'''
print(5+10);
```

Another way to insert comments is to prefix comment statements with a **#** symbol. You can use this technique to comment an entire line or to add a comment to the end of a line. The following two lines accomplish the same task as the previous code snippet.

```
# Use the print() function to print the sum of 5 and 10
print(5+10); # This statement should output 15
```

Many of our examples, especially transcriptions of code from example files on the companion web site, are devoid of comments. We opted to limit

comments to save printing space, but hope that you will not be so flippant as a developer! Most of our examples on the companion web site contain fairly thorough comments, and we tend to break up large sections of code in the text to discuss them one part at a time.

PYTHON HANDS ON

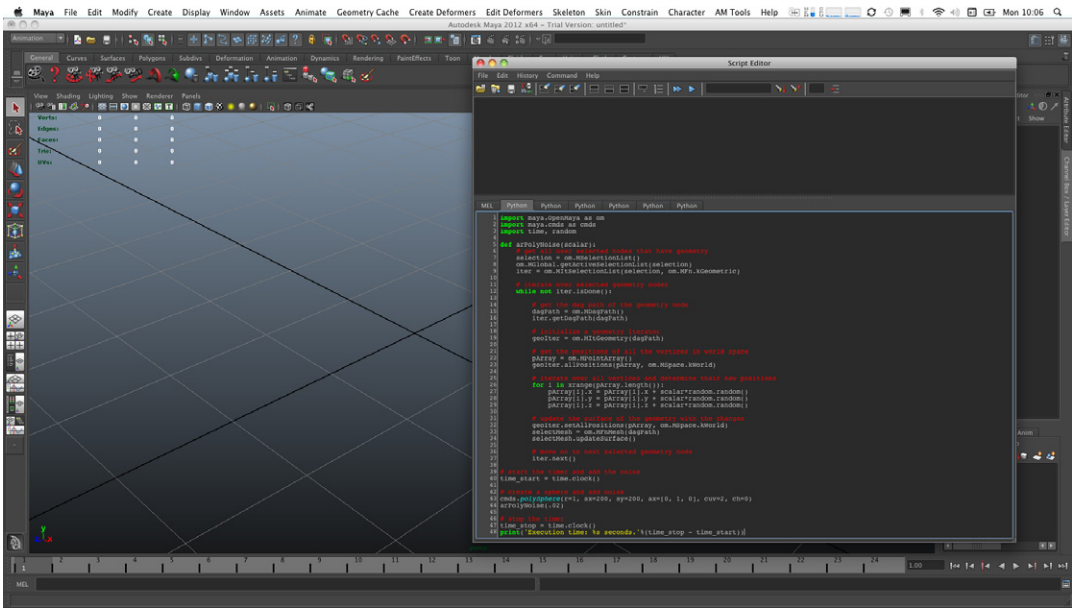
In our introduction to Python, we have told you how useful it will be to you in Maya. We'll now run through a brief example project to get a taste of things to come. Don't worry if you do not understand anything in the example scripts yet, as this project is for demonstration purposes only.

In this example, you are going to execute two versions of the same script, one written in MEL using only Maya commands, and the other written using the Maya Python API. These scripts highlight one example of the dramatically different results you can achieve when using Python in place of MEL.

The scripts each create a basic polygon sphere with 200 by 200 subdivisions (or 39,802 vertices) and apply noise deformation to the mesh. In essence, they iterate through all of the sphere's vertices and offset each one by a random value. This value is scaled by an amount that we provide to the script to adjust the amount of noise.

Please also note that the Python script in the following example makes use of functions that were added to the API in Maya 2009. As such, if you're using an earlier version of Maya, you can certainly examine the source code, but the Python script will not work for you.

1. Open the Maya application on your computer.
2. In Maya's main menu, open the Script Editor by navigating to **Window → General Editors → Script Editor**.
3. You should now see the Script Editor window appear, which is divided into two halves (Figure 0.1). Above the lower half you should see two tabs. Click on the Python tab to make Python the currently active language.
4. Download the polyNoise.py script from the companion web site and make note of its location.
5. Open the script you just downloaded by navigating to the menu option **File → Load Script** in the Script Editor window (rather than the main application window). After you browse to the script and load it, you will see the contents of the script appear in the lower half of the Script Editor window, where they may be highlighted.
6. Click anywhere in the bottom half of the Script Editor to place your cursor in it, and execute the script by pressing **Ctrl + Enter**.



■ **FIGURE 0.1** The polyNoise.py script in the Maya Script Editor.

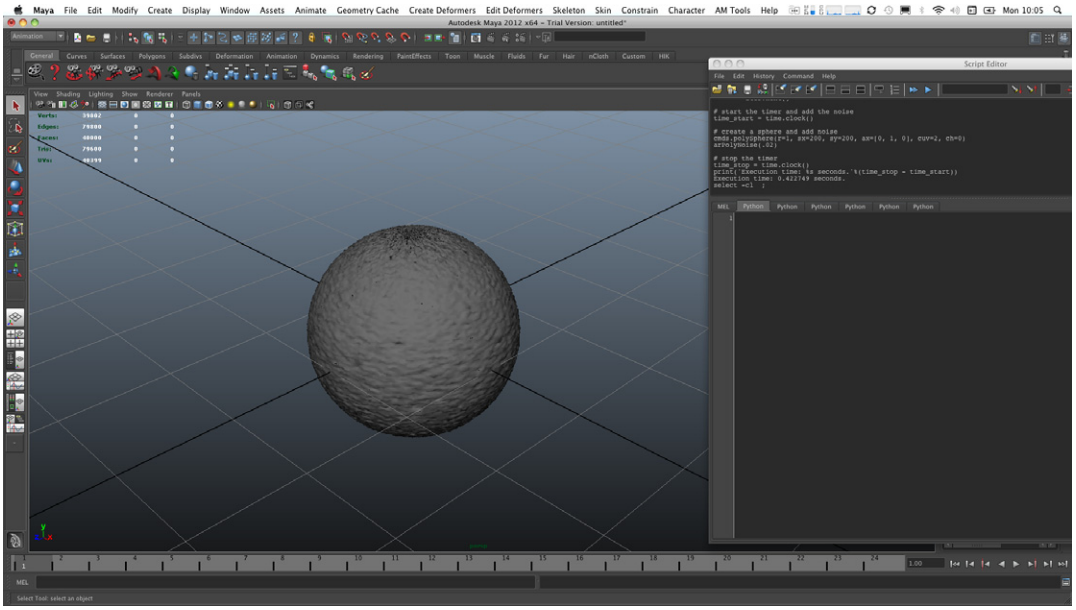
Once the script has executed, you will see a distorted sphere in your viewport, as in Figure 0.2. However, you will also see something interesting in the top half of the Script Editor window. In addition to the script that you just executed, you should see a line that says something like the following:

```
Execution time: 0.53 seconds.
```

This final line shows how long it took your computer to create the sphere, subdivide it, and apply noise to the mesh. In this particular case, it took our computer 0.53 seconds, though your result may be slightly different based on the speed of your computer.

Now, you will execute the MEL version of this script to see how long it takes to perform the same operation.

1. In Maya's Script Editor, click the MEL tab that appears above the bottom half of the window.
2. Download the polyNoise.mel script from the companion web site and make note of its location.
3. Open the script you just downloaded by navigating to the menu option **File → Load Script** in the Script Editor window. After loading the



■ **FIGURE 0.2** A polygon sphere with the noise script applied.

script, you should see its contents appear in the lower half of the Script Editor window, and they may be highlighted.

4. Click anywhere in the bottom half of the Script Editor to place your cursor in it, and execute the script by pressing **Ctrl + Enter**.

You should very quickly notice a problem—or should we say very slowly. Maya will stop responding after executing the MEL script. Mac users will see the infamous "beach ball" loading cursor. Don't worry though! Maya has not crashed. It will create the sphere … eventually. You might want to go get a cup of coffee or two before you come back to check on Maya. After what seems like an eternity, Maya will finally create the sphere with the noise and print something like the following line in the top half of the Script Editor window.

```
Execution time: 203.87 seconds.
```

Because the Python script and the MEL script both perform the exact same task, you could theoretically use either one to get the job done. Obviously, however, you would want to use the Python version. If we created a MEL script that did this task it would be useless in production. No artist would want to use a script that takes a few minutes to execute. It would make the tool very difficult for artists to experiment with and most likely it would be abandoned.

There is only one catch: The Python script was created using the Maya Python API. If you compare the two scripts side-by-side, you will see that the Python script is slightly more complex than the MEL script as a result. Using the API is more complicated than just using Maya commands, yet MEL cannot use the API directly. This disadvantage on MEL's part is the primary reason the performance difference is so dramatic in this case.

Another issue to point out about the Python example script is that it is not complete. Because it uses API calls to modify objects, you cannot use undo like you could with MEL if you didn't like the result. Although Python can certainly use Maya commands just like MEL (and automatically benefit from the same undo support), we used the API in this case because it was necessary to gain the substantial speed increase. If we used Python to call Maya commands, however, the Python script would have been just as slow as the MEL script.

Although we chose not to in this example for the sake of simplicity, we would need to turn this script into a Python API plug-in to maintain undo functionality and still use the API to modify objects. There may in fact be many times you want to do this very same thing to mock up a working version before adding the final bits of script to create a plug-in. Don't worry—it isn't hard, but it is another step you will need to take. Fortunately for you, one of the main topics in this book is to explain how to work with the API using Python, so you will have no trouble creating lightning-fast tools yourself. Once you understand more about how Python works in Maya, you could write other scripts that work with meshes. As you can see, MEL tends to run very slowly on objects with large vertex counts, so this opens the door for new tools in your production.

Another convenient advantage to using the Python API is it uses the same classes as the C++ API. If you really needed additional speed, you could easily convert the Python script into a C++ plug-in. If you don't understand C++, you could pass on the script as a template for a programmer at your studio, as the API classes are identical in both languages. C++ programmers can also find the Python API useful because they can access the Python API in Maya interactively to test out bits of code. If you are using C++, on the other hand, you absolutely must compile a plug-in to test even one line of code. You can even mix the Maya API with Python scripts that use Maya commands.

With all of these features in mind, we hope you are excited to get started learning how to use Python in Maya. By the end of this book, you should be able to create scripts that are more complicated than even the example from this chapter. So without further delay, let's get to it!