A Complete Reference for

Maya Python and the Maya Python API

MAYA PYTHON
for Games and Film

Adam Mechtley · Ryan Trowbridge

MORGAN KAUFMANN

Maya Python for Games and Film

Maya Python for Games and Film

A Complete Reference for Maya Python and the Maya Python API

Adam Mechtley

Ryan Trowbridge

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20140814

International Standard Book Number-13: 978-0-12-378579-4 (eBook - PDF)

Contents

Acknowledgments xiii
Introduction: Welcome to Maya Python xv

PART 1 BASICS OF PYTHON AND MAYA

CHAPTER 1 Maya Command Engine and User Interface 3
Interacting with Maya 4
Maya Embedded Language 5
Python 5
C++ Application Programming Interface 6
Python API 6
Executing Python in Maya 6
Command Line 6
Script Editor 8
Maya Shelf 10
Maya Commands and the Dependency Graph 11
Introduction to Python Commands 15
Flag Arguments and Python Core Object Types 19
Numbers 20
Strings 20
Lists 20
Tuples 21
Booleans 21
Flag = Object Type 21
Command Modes and Command Arguments 22
Create Mode 22
Edit Mode 23
Query Mode 23
Python Command Reference 24
Synopsis 25
Return Value 25
Related 25
Flags 25
Python Examples 26
Python Version 26
Python Online Documentation 26
Concluding Remarks 27

CHAPTER 2 Python Data Basics 29
Variables and Data 30
Variables in MEL 33
Keywords 33
Python's Data Model 34
Using Variables with Maya Commands 37
Capturing Results 39
`getAttr` and `setAttr` 40
`connectAttr` and `disconnectAttr` 41
Working with Numbers 43
Number Types 43
Basic Operators 44
Working with Booleans 45
Boolean and Bitwise Operators 45
Working with Sequence Types 46
Operators 46
String Types 50
Formatting Strings 52
More on Lists 53
Other Container Types 56
Sets 57
Dictionaries 58
Concluding Remarks 62

CHAPTER 3 Writing Python Programs in Maya 63
Creating Python Functions 64
Anatomy of a Function Definition 64
Function Arguments 66
Return Values 74
Maya Commands 75
Listing and Selecting Nodes 76
The `file` Command 78
Adding Attributes 79
Iteration and Branching 80
The for Statement 80
Branching 84
List Comprehensions 93
The while Statement 94
Error Trapping 96
try, except, raise, and finally 96

Designing Practical Tools 99
Concluding Remarks 109

CHAPTER 4 Modules 111
What Is a Module? 113
Modules and Scope 114
Module Encapsulation and Attributes 115
The __main__ Module 116
Creating a Module 118
The spike Module 118
Default Attributes and help() 120
Packages 121
Importing Modules 125
import versus reload() 125
The as Keyword 126
The from Keyword 126
Python Path 127
sys.path 128
Temporarily Adding a Path 129
userSetup Scripts 131
sitecustomize Module 133
Setting Up a PYTHONPATH Environment Variable 135
Using a Python IDE 140
Downloading an IDE 140
Basic IDE Configuration 141
Concluding Remarks 145

CHAPTER 5 Object-Oriented Programming in Maya 147
Object-Oriented versus Procedural Programming 148
Basics of Class Implementation in Python 150
Instantiation 150
Attributes 151
Data Attributes 153
Methods 155
Class Attributes 160
Human Class 163
Inheritance 164
Procedural versus Object-Oriented Programming in Maya 168
Installing PyMEL 168
Introduction to PyMEL 168
PyNodes 169

PyMEL Features 170
Advantages and Disadvantages 171
A PyMEL Example 172
Concluding Remarks 175

PART 2 DESIGNING MAYA TOOLS WITH PYTHON

CHAPTER 6 Principles of Maya Tool Design 179
Tips When Designing for Users 180
Communication and Observation 181
Ready, Set, Plan! 181
Simplify and Educate 183
Tools in Maya 183
Selection 184
Marking Menus 186
Options Windows 190
Concluding Remarks 192

CHAPTER 7 Basic Tools with Maya Commands 193
Maya Commands and the Maya GUI 194
Basic GUI Commands 196
Windows 196
Building a Base Window Class 198
Menus and Menu Items 199
Executing Commands with GUI Objects 201
Layouts and Controls 206
Complete AR_OptionsWindow Class 215
Extending GUI Classes 218
Radio Button Groups 219
Frame Layouts and Float Field Groups 220
Color Pickers 222
Creating More Advanced Tools 224
Pose Manager Window 224
Separating Form and Function 226
Serializing Data with the cPickle Module 226
Working with File Dialogs 229
Concluding Remarks 232

CHAPTER 8 Advanced Graphical User Interfaces with Qt 233
Qt and Maya 234
Docking Windows 236

Installing Qt Tools 237
The Qt SDK 237
Qt Designer 239
Widgets 240
Signals and Slots 241
Qt Designer Hands On 241
Loading a Qt GUI in Maya 246
The `loadUI` command 248
Accessing Values on Controls 250
Mapping Widgets with Signals and Slots 251
PyQt 254
Installing PyQt 254
Using PyQt in Maya 2011+ 255
Using PyQt in Earlier Versions of Maya 257
Concluding Remarks 257

PART 3 MAYA PYTHON API FUNDAMENTALS

CHAPTER 9 Understanding C++ and the API Documentation 261
Advanced Topics in Object-Oriented Programming 264
Inheritance 264
Virtual Functions and Polymorphism 265
Structure of the Maya API 265
Introducing Maya's Core Object Class: MObject 266
How Python Communicates with the Maya API 268
How to Read the API Documentation 270
Key Differences between the Python and C++ APIs 281
MString and MStringArray 281
MStatus 281
Void* Pointers 281
Proxy Classes and Object Ownership 281
Commands with Arguments 282
Undo 282
MScriptUtil 282
Concluding Remarks 283

CHAPTER 10 Programming a Command 285
Loading Scripted Plug-ins 287
Anatomy of a Scripted Command 289
OpenMayaMPx Module 290
Command Class Definition 290

doIt() 291
Command Creator 291
Initialization and Uninitialization 292
Adding Custom Syntax 293
Mapping Rotation Orders 296
Class Definition 297
Syntax Creator 298
Initialization of Syntax 301
doIt() 302
doItQuery() 305
Maya's Undo/Redo Mechanism 308
Supporting Multiple Command Modes and Undo/Redo 313
Undo and Redo 313
Command Modes 314
Syntax Creator 317
__init__() Method 318
doIt() 320
redoIt() 323
undoIt() 325
Concluding Remarks 326

CHAPTER 11 Data Flow in Maya 327
Dependency Graph 328
Dependency Nodes 330
Connections 333
Debugging the Dependency Graph 336
The `dgTimer` Command 338
Directed Acyclic Graph 339
DAG Paths and Instancing 344
The Underworld 347
Concluding Remarks 350

CHAPTER 12 Programming a Dependency Node 351
Anatomy of a Scripted Node 352
The ar_averageDoubles Node 352
Node Class Definition 354
Node Creator 356
Node Initializer 356
compute() 357
Initialization and Uninitialization 359

Attributes and Plugs 360
Attribute Properties 361
Readable, Writable, and Connectable 361
Storable Attributes and Default Values 361
Cached Attributes 363
Working with Arrays 363
Compound Attributes 370
Concluding Remarks 375

Acknowledgments

We would like to thank the many people who made this book possible. Foremost, we want to thank our wives, Laurel Klein and Brenda Trowbridge, without whose unflagging support we would have been hopeless in this endeavor. We would also like to thank the various contributors to this project for their individual efforts, including Seth Gibson of Riot Games for Chapters 3 and 5, Kristine Middlemiss of Autodesk for Chapter 8, and Dean Edmonds of Autodesk for his technical editing, a man whose power to explain is surpassed only by his knowledge of the topics covered herein.

We would also like to thank the team at Focal Press who has helped out on this project, including Sara Scott, Laura Lewin, and Anaïs Wheeler. Your support and patience has been a blessing. Thanks also to Steve Swink for putting us in touch with the wonderful people at Focal Press.

We want to thank all of the people in the Maya Python and API community who have been essential not only for our own growth, but also for the wonderful state of knowledge available to us all. Though we will undoubtedly leave out many important people, we want to especially thank Chad Dombrova, Chad Vernon, Paul Molodowitch, Ofer Koren, and everyone who helps maintain the tech-artists.org and PyMEL communities.

Last, but certainly not least, we want to thank our readers for supporting this project. Please get in touch with us if you have any feedback on the book!

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

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

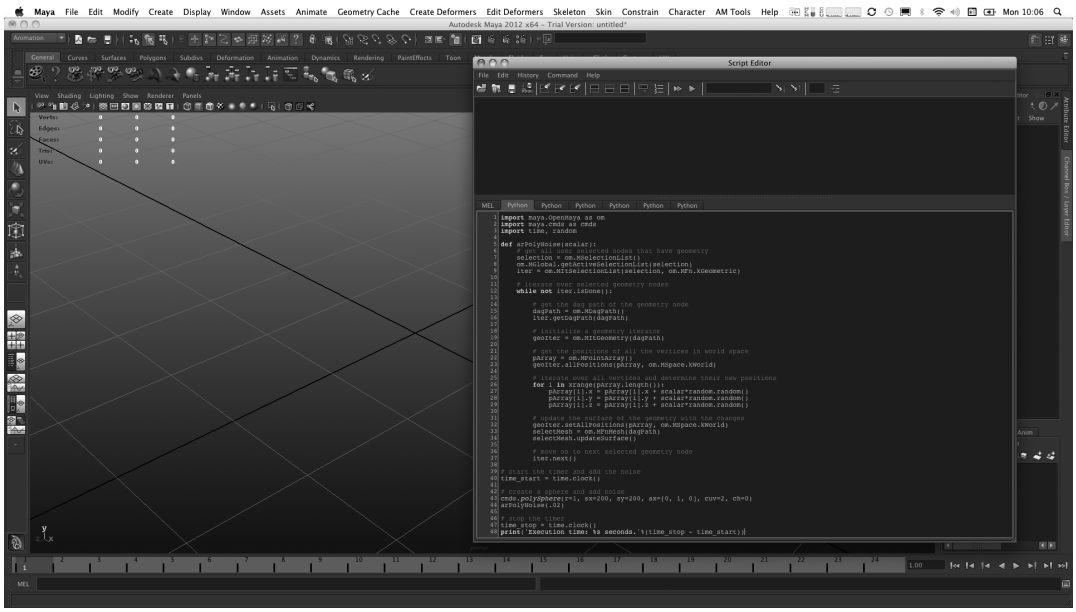
In our introduction to Python, we have told you how useful it will be to you in Maya. We'll now run through a brief example project to get a taste of things to come. Don't worry if you do not understand anything in the example scripts yet, as this project is for demonstration purposes only.

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.

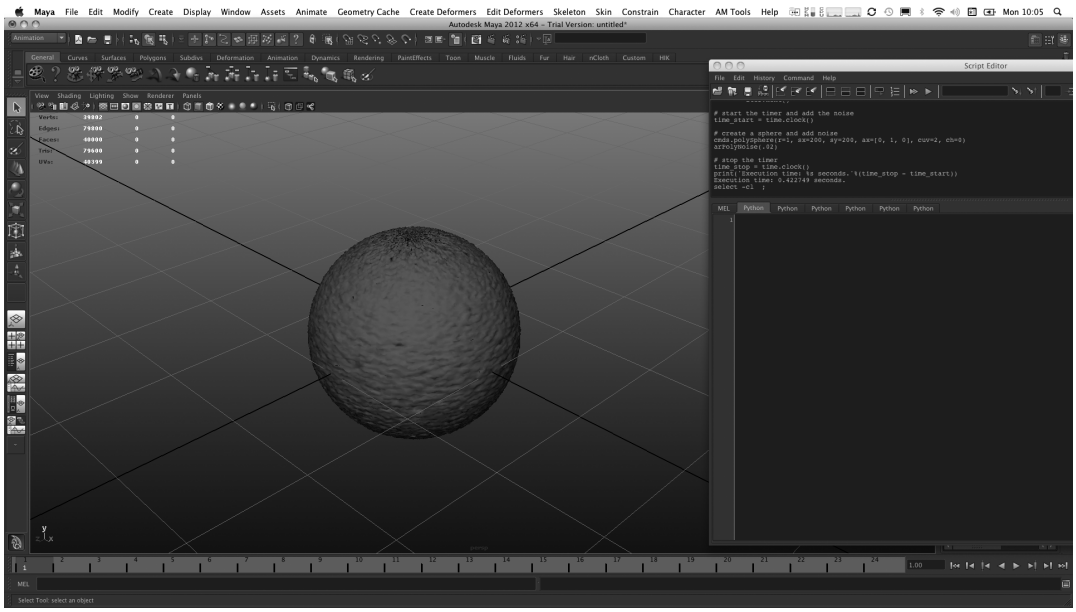
Once the script has executed, you will see a distorted sphere in your viewport, as in Figure 0.2. However, you will also see something interesting in the top half of the Script Editor window. In addition to the script that you just executed, you should see a line that says something like the following:

```
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!

Part 1

Basics of Python and Maya

Chapter 1

Maya Command Engine and User Interface

CHAPTER OUTLINE

Interacting with Maya 4
- Maya Embedded Language 5
- Python 5
- C++ Application Programming Interface 6
- Python API 6

Executing Python in Maya 6
- Command Line 6
- Script Editor 8
- Maya Shelf 10

Maya Commands and the Dependency Graph 11

Introduction to Python Commands 15

Flag Arguments and Python Core Object Types 19
- Numbers 20
- Strings 20
- Lists 20
- Tuples 21
- Booleans 21
- Flag = Object Type 21

Command Modes and Command Arguments 22
- Create Mode 22
- Edit Mode 23
- Query Mode 23

Python Command Reference 24
- Synopsis 25
- Return Value 25
- Related 25
- Flags 25
- Python Examples 26

Python Version 26

Python Online Documentation 26

Concluding Remarks 27

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

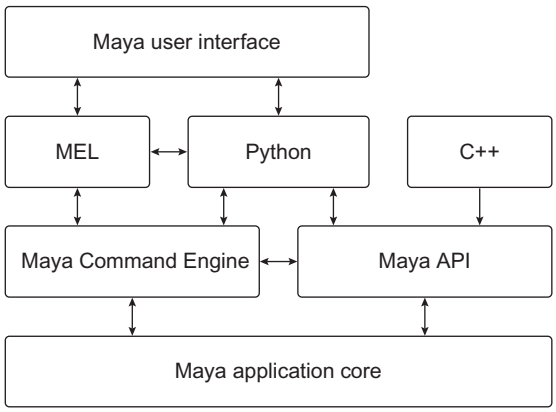
- Compare and contrast the four Maya programming interfaces.
- Use the Command Line and Script Editor to execute Python commands.
- Create a button in the Maya GUI to execute custom scripts.
- Describe how Python interacts with Maya commands.
- Define nodes and connections.
- Describe Maya's command 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

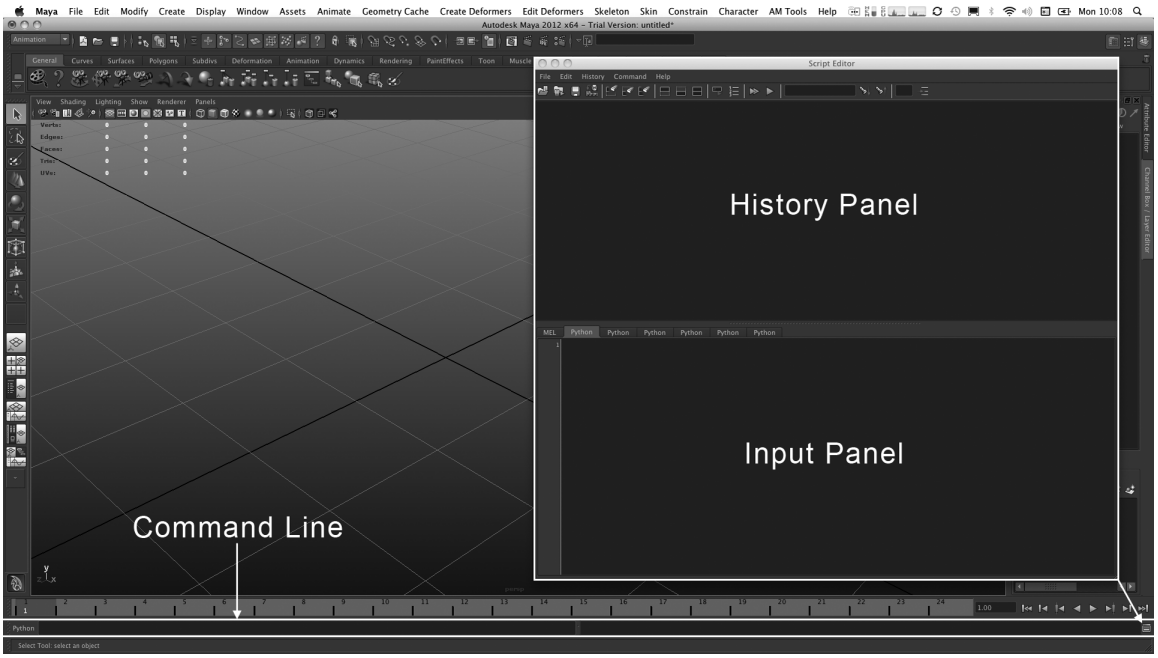
Maya has many tools built into its GUI that allow users to execute Python code. Before you begin programming Python code in Maya, you should familiarize yourself with these tools so that you know not only what tool is best for your current task, but also where to look for feedback from your scripts.

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.

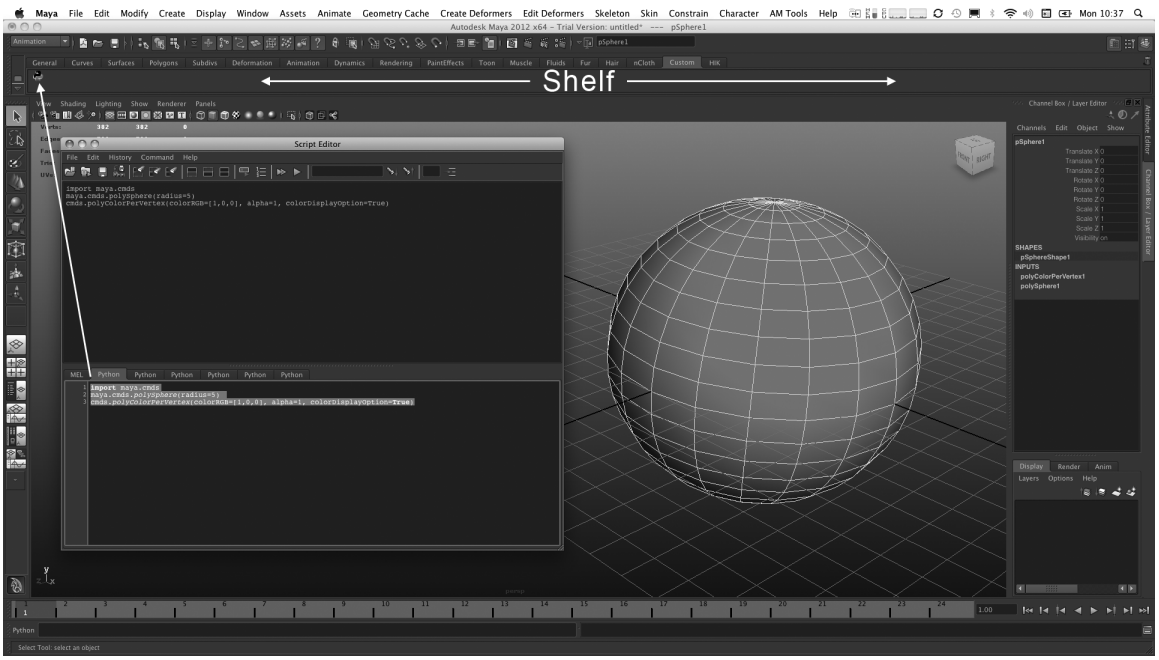
As you can see, the Script Editor is a very useful tool not only for creating and executing Python scripts in Maya, but also for quickly pulling up information about commands in your script. We will look at the command documentation later in this chapter.

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



■ **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

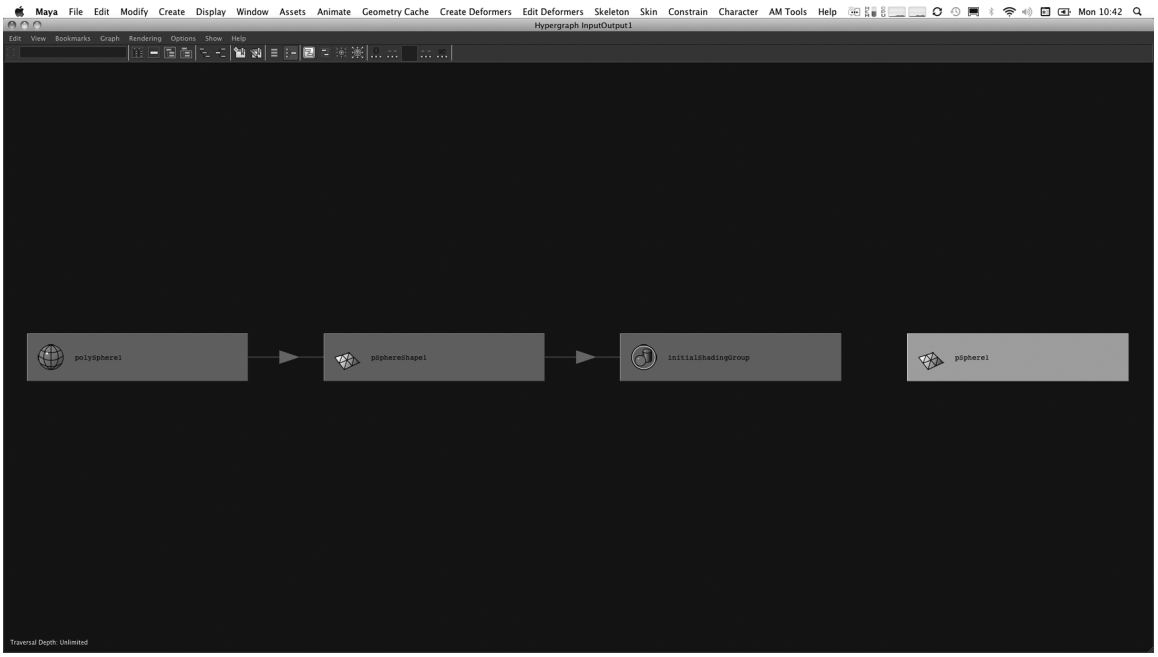
To create a polygonal sphere with Python, the `polySphere` command must be executed in some way or other. The `polySphere` command is part of the Maya Command Engine. As we noted previously, the Maya Command Engine includes a set of commands accessible to both MEL and Python.

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.

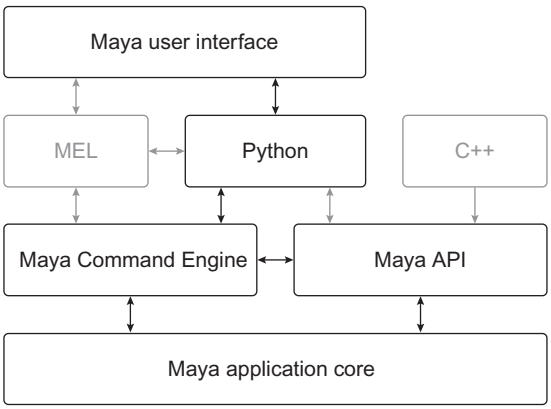
If you were to delete the construction history of this polygonal sphere (**Edit → Delete by Type → History** from the main menu), the **polySphere** node would disappear and the sphere's geometry would then be statically stored in the **shape** node (**pSphereShape1** in Figure 1.4). In short, if the **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

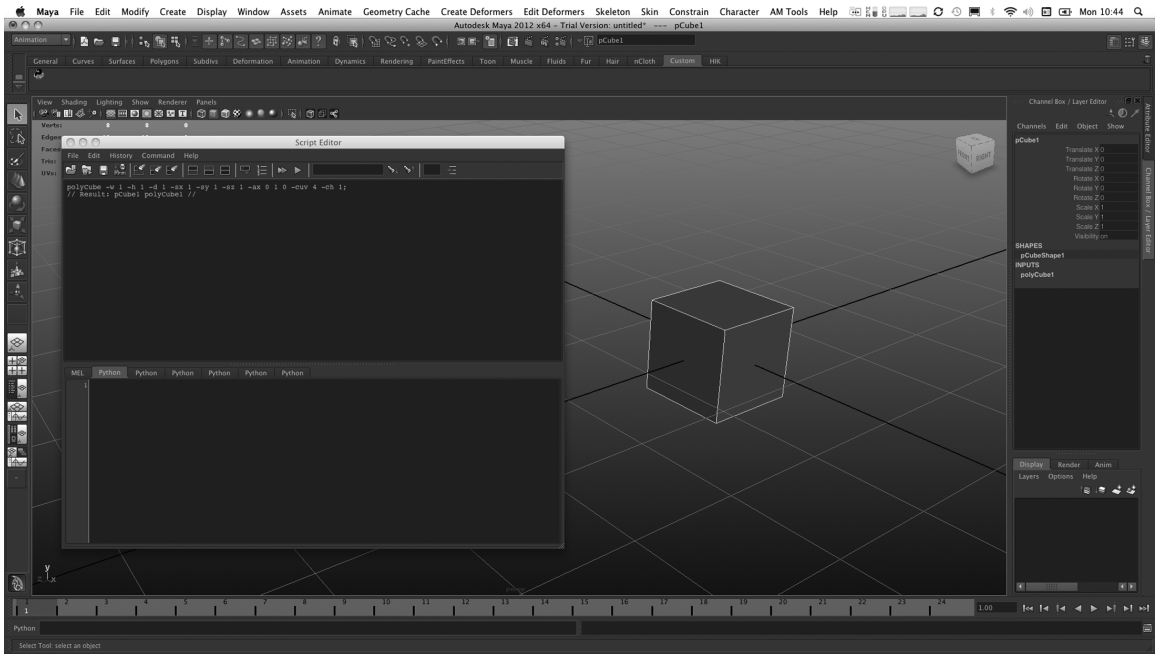
Let's return to Maya and open up the Script Editor. As discussed earlier in this chapter, the top panel of the Script Editor is called the History Panel. This panel can be very useful for those just learning how to script or even for advanced users who want to figure out what commands are being executed. 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.



■ **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;
```