Chapter 9

Understanding C++ and the API Documentation

CHAPTER OUTLINE

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

- Define what the Maya API is.
- Explain how the Maya API uses virtual functions and polymorphism.
- Describe the organization of the Maya API.
- Understand the difference between data objects and function sets.
- Define what SWIG is.
- Navigate the Maya API documentation.

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00009-0

- Identify what API functionality is accessible to Python.
- Compare and contrast Python and C++ in the API.

Armed with an understanding of Python and familiarity with the cmds module, you have many new doors open to you. In addition to being able to access all of the functionality available in MEL, you can also architect completely new classes to create complex behaviors. However, these new tools only scratch the tip of a much larger iceberg. In addition to the basic modules examined so far, Maya also implements a set of modules that allow developers to utilize the Maya API via Python.[1] These modules include:

- OpenMaya.py
- OpenMayaAnim.py
- OpenMayaRender.py
- OpenMayaFX.py
- OpenMayaUI.py
- OpenMayaMPx.py
- OpenMayaCloth.py[2]

The specifics of some of these modules are discussed in greater detail throughout the following chapters. For now, let's take a step back and discuss what the Maya API is more generally before using it in Python. If you are relatively new to software and tools development, you might be wondering what exactly the Maya API is in the first place.

The Maya API (short for application programming interface) is an abstract layer that allows software developers to communicate with the core

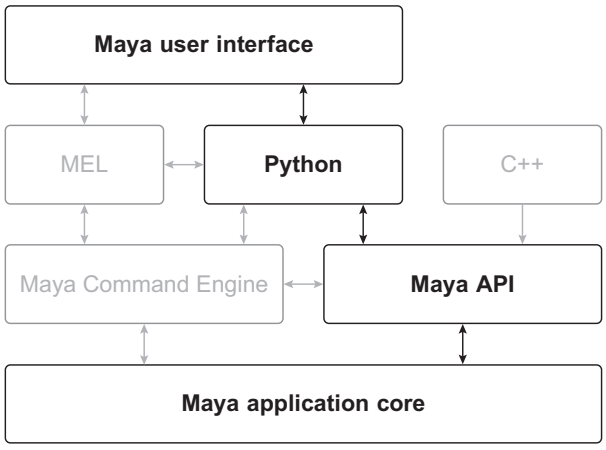
[1]Note that Maya 2012 Hotfix 1 introduced the Python API 2.0 in the maya.api module. This module contains a mostly parallel structure of extension modules (e.g., maya.api.OpenMaya, maya.api.OpenMayaAnim, and so on). The advantage of the Python API 2.0 is that it is often faster, eliminates some of the complexities of working with the API, and allows use of Pythonic features with API objects, such as slicing on API objects. While the new and old API can be used alongside each other, their objects cannot be intermixed (you cannot pass an old API object to a new API function). Nevertheless, they have the same classes and so learning with the old API is not disadvantageous. Because the new Python API is still in progress, and because it is only supported in the latest versions of Maya, all of our API examples in this book are written using the ordinary API modules. Refer to the Python API 2.0 documentation or the companion web site for more information on the Python API 2.0.

[2]The OpenMayaCloth.py module is for working with classic Maya cloth, and has become deprecated in favor of the newer nCloth system (introduced in Maya 8.5). The nCloth system is part of Maya Nucleus, which is accessed by means of the OpenMayaFX.py module. Consequently, you will generally not use the OpenMayaCloth.py module unless you are supporting legacy technology that has been built around the classic Maya cloth system.

functionality of Maya using a limited set of interfaces. In other words, the API is a communication pathway between a software developer and Maya's core. What exactly does this mean, though? How does the API differ from the scripting interfaces we have discussed so far?

Although Python scripts for Maya are very powerful, they still only interface with the program at a very superficial level when not using the API. If you recall the architecture we discussed in Chapter 1, the Command Engine is the user's principal gateway into Maya for most operations. Thus, Python scripts that only use the cmds module essentially allow us to replicate or automate user behavior that can otherwise be achieved by manually manipulating controls in Maya's GUI. However, programming with the API directly enables the creation of entirely new functionality that is either unavailable or too computationally intensive for ordinary scripts using only cmds. Using the API modules, you can create your own commands, DG nodes, deformers, manipulators, and a host of other useful features (Figure 9.1).

The trouble for Python developers, however, is that the API is written in C++. Although you don't have to be a C++ programmer to use the API, it is still important to have some understanding of the language to effectively read the API documentation. In this chapter, we will first revisit some concepts of object-oriented programming to examine the structure of the Maya API and understand how Python communicates with it. Thereafter, we will take a short tour of the API documentation. The chapter concludes with a brief section detailing some important differences between Python and C++ with respect to the Maya API.



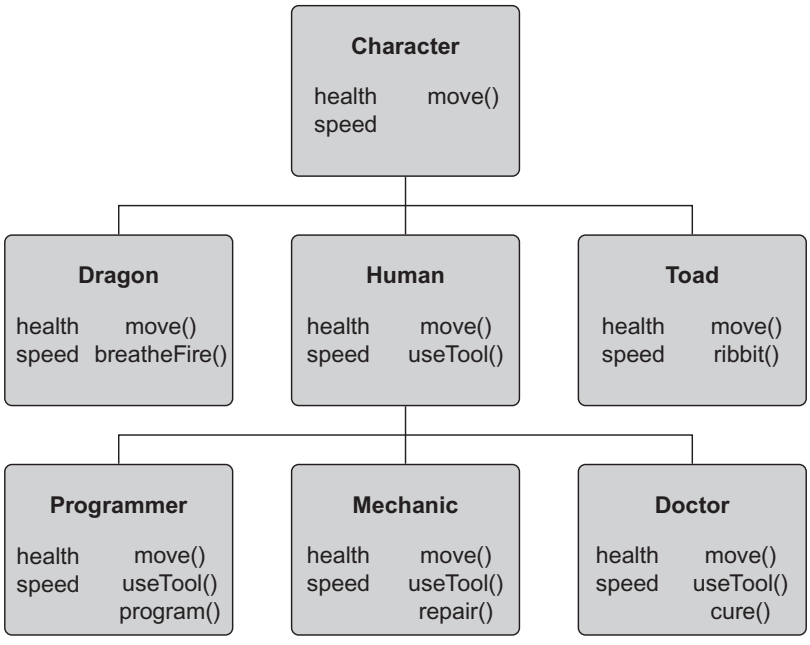
■ **FIGURE 9.1** Using Python to access the Maya API offers richer interfaces with Maya's application core.

ADVANCED TOPICS IN OBJECT-ORIENTED PROGRAMMING

Before delving too deeply into Maya's API structure, we must briefly discuss a couple of advanced topics in OOP that we did not detail in Chapter 5. Because the Maya API is written in C++, a statically typed language, it makes use of some features that may not be immediately obvious to programmers whose OOP experiences are limited to Python.

Inheritance

As we discussed in Chapter 5, OOP fundamentally consists of complex objects that contain both data and functionality, and that may inherit some or all of these properties from each other. For instance, a program may have a base class called **Character** with some common properties and functionality necessary for all characters (Figure 9.2). These properties may include **health** and **speed**, as well as a **move()** method. Because there may be a variety of different types of characters—**Human**, **Dragon**, and **Toad**—each of these characters will likely require its own further properties or its own special methods not shared with the others. The **Human** may have a **useTool()** method, the **Dragon** a **breatheFire()** method, and the **Toad** a **ribbit()** method. Moreover, each of these classes



■ **FIGURE 9.2** An example object hierarchy.

may have further descendant classes (e.g., **Human** may break down into **Programmer**, **Mechanic**, and **Doctor**).

Virtual Functions and Polymorphism

In a language like C++, the **Character** class in our present example contains what is referred to as a virtual function for **move()**. As far as Python developers need to be concerned, a *virtual function* is basically a default behavior for all objects of a certain basic type, which is intended to be overridden in child classes with more specific types. While a language like Python inherently incorporates this principle, C++ must explicitly specify that a method is virtual. Considering this concept in the present example, the **move()** method would be overridden in each descendent class, such that it may contain logic for walking for a **Human**, while a **Dragon** character's **move()** method may implement logic for flying.

Thus, using virtual functions in this fashion allows a programmer to operate with a single object as multiple different types based on the abstraction level at which it is being manipulated. The technique is referred to as *polymorphism*. Again, while Python permits this technique with relative ease, it is something that a language like C++ must manually specify. The ultimate consequence, however, is that a function can be made to work with different types of objects, and can be invoked at different abstraction levels. A hypothetical Python example can help illustrate this point.

```
# create some characters
kratos = Character();
nathan = Human();
smaug = Dragon();
kermit = Toad();
# add the characters to a list
characters = [kratos, nathan, smaug, kermit];
# make all of the characters move
for character in characters: character.move();
```

Although each of the objects in the `characters` list is constructed as a different specific type, we can treat them all as equals inside of the iterator when calling the **move()** method. As you will soon see, understanding this technique is especially important when using the Maya API.

STRUCTURE OF THE MAYA API

While the Maya API makes use of techniques like polymorphism, it also generally adheres to an important design philosophy. Recalling Maya's program architecture (see Figure 9.1), the Maya API fundamentally consists of a

set of classes that allow developers to *indirectly* interface with Maya's core data. It is important to emphasize that the API does *NOT* permit direct access to these data existing in the core framework. *Maya maintains control over its core data at all times*.[3] Therefore, Autodesk engineers developing Maya can theoretically make dramatic changes to the application's core from one version to the next without worrying too much about whether any custom tools will still comply.

Because developers interface only with the API, which changes comparatively little and communicates with the core on their behalf, they do not need to worry about the core's structure. Likewise, they are protected from errors that may cause critical dysfunction, as the API acts as a communication filter. However, these particular facts lend themselves to an API class hierarchy that may not be immediately intuitive. Specifically, developers are generally only given access to a primary API base class: **MObject**.

Introducing Maya's Core Object Class: MObject

While we could directly interact with objects in descendant classes in the previous hypothetical examples, Maya only allows developers to interact with an **MObject** when working with complex data.[4] The **MObject** class knows about the whole data hierarchy inside of Maya's core, and so it is able to locate the proper functionality when developers ask an object to do something.

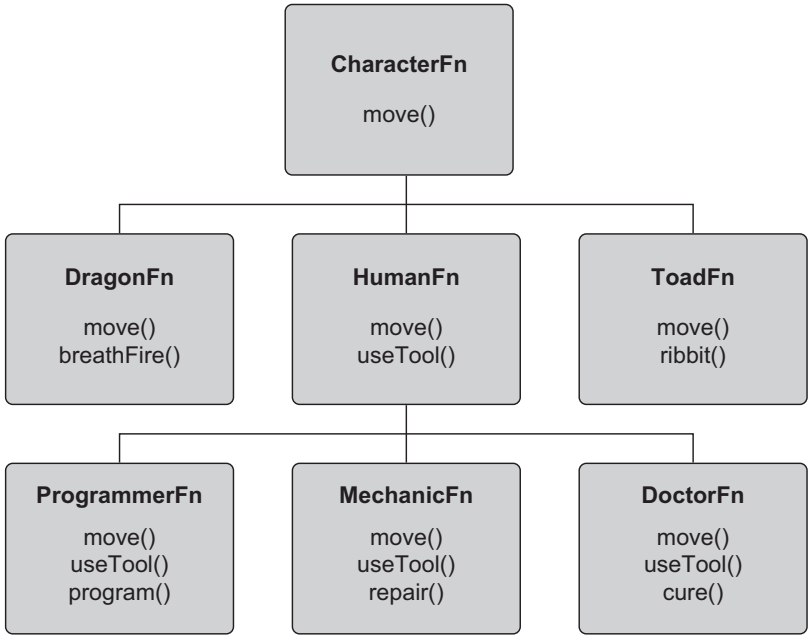
It is important to note, however, that **MObject** is not an actual data object within Maya's core. You are not allowed direct access to those objects! Rather, **MObject** is simply a pointer to some object in the core. Creating or deleting an **MObject**, then, leaves the actual data object inside the core untouched. Following this paradigm, internal data such as attributes, DG nodes, meshes, and so on are accessed by means of an **MObject**.

At this point in our discussion, a new question naturally arises. Namely, if developers only have access to a single base class, how can they access functionality in descendent classes?

---

[3]Technically speaking, there are some cases where the Maya API does in fact provide direct access to data in the core, typically only where needed to improve performance and only when there is no risk of adversely affecting them. However, these cases are not documented for developers and are not important to know as a practical matter. Consequently, the philosophy of the API still holds as a general rule as far as developers need to be concerned.

[4]Some basic types are directly accessible in the API, such as **MAngle**, **MDataHandle**, **MTime**, etc.



■ **FIGURE 9.3** An example hierarchy of function sets.

Manipulating MObjects Using Function Sets

Following its overarching design philosophy, the Maya API implements separate classes for data objects and their corresponding function sets. While a *data object* contains all of an object's properties, a *function set* is a special type of class that acts as an interface for calling methods associated with particular types of objects.

If we were to apply this paradigm to the previous hypothetical example, there would be not only a **Character** class, but also an associated class, **CharacterFn**, containing a **move()** method. This class would have further descendants implementing the children's functionality—**HumanFn**, **DragonFn**, and **ToadFn**. (See Figure 9.3.)

Because the Maya API makes use of polymorphism, these function sets can operate on different types of data. Consider the following hypothetical example using our character scenario.

```
# create a human
human = Human();
# instantiate the CharacterFn function set
charFn = CharacterFn();
# tell the instance of the function set to operate on the human
charFn.setObject(human);
```

```
# tell the human to move
charFn.move();
```

In this instance, polymorphism allows a **CharacterFn** object to interact with any objects in descendant classes, such as a **Human** in this case. *The key here is to remember that we are not directly accessing the data!* Because of the programming features that the Maya API has implemented, what we are really doing is asking the `human` object to perform its implementation of **move()** itself, rather than some implementation within **CharacterFn**. Because **move()** is a virtual function defined in **Character**, its behavior has been overridden in the **Human** class, which is the behavior that will ultimately be invoked in this case due to the type specified when the `human` object was created.

Thus, Maya implements classes such as **MFnAttribute**, **MFnDependencyNode**, and **MFnMesh** to operate on attributes, DG nodes, and meshes, respectively. Although each of these types of objects is opaque to developers, you can supply any of these function sets with an **MObject** and call functionality as needed at the appropriate abstraction level.

Now that we have discussed the basics of how the API is structured, we can move on to another important question. Namely, if the Maya API is written in C++, how can we use it with Python?

HOW PYTHON COMMUNICATES WITH THE MAYA API

As noted earlier in this chapter, the Maya API is architected in C++. Because of this, we need some means of interacting with the API using Python.

To solve this problem, Autodesk has used a technology called Simplified Wrapper and Interface Generator, or simply SWIG for short.[5] SWIG is able to automatically generate bindings directly to the C++ code if it is written in a certain way. In essence, when Autodesk introduced Python formally in Maya 8.5, a central task to create the OpenMaya modules was to simply restructure the C++ API header files to allow SWIG to work. The advantage of this approach is that the Python modules will automatically benefit from improvements in the C++ API with each new version and require minimal additional investment to produce. The disadvantage to this approach—at least at the time of this writing—is that not all of the C++

[5]Python API 2.0 is not generated via SWIG, and many of its methods have been reworked from the ground up to return data differently, which is why it eliminates the use of **MScriptUtil**. See note 1.

API functionality is accessible to Python. Fortunately, however, it is fairly clear what you can and cannot do in Python, as we will examine in further detail in the next section.

If you're in Maya, try entering the following example into your Script Editor window to see SWIG in action.

```
import maya.OpenMaya as om;
vec = om.MVector();
print(vec);
```

You can see from the output that the vector is a special type, and not simply a tuple or list.

```
"<maya.OpenMaya.MVector; proxy of <Swig Object of type
'MVector *' at [some memory address]> >"
```

It is important to emphasize here that using API functionality in Python is simply calling an equivalent C++ API method. As a general rule, this process tends to be relatively fast for most simple operations, but can induce a substantial performance penalty in situations that involve complex or numerous API calls. Each time an API call is made, Python's parameters must be converted into something that C++ can understand, and the C++ result must respectively be translated back into something that Python understands.

The fact that Python is interpreted also introduces a minimal speed penalty, but the communication between Python and C++ ultimately tends to be the largest contributor to performance problems in complex tools.[6] As such, there are some methods included in the API (with more being constantly added) that allow large amounts of data to be obtained from C++ or given back to C++ with a minimal function call overhead.[7] In such cases, especially when working on meshes, there is less overhead to pull all of the data at once, operate on it in Python, and then give it all back to C++ afterward. Consequently, C++ may still ultimately be the best option for extremely computationally intensive tasks or simulations, though Python will almost certainly always be the fastest means of prototyping in the API.

The fact that the API is written in C++ has one further consequence as far as Python developers are concerned: The API documentation is in C++.

[6]Edmonds, D. (2008). Advanced Maya Python. Presented at Autodesk Maya Developer's Conference, 2008, Scottsdale, AZ.

[7]This issue is why the example in this book's Introduction chapter used the **MItGeometry.setAllPositions()** method, for example.

As a result, you need to know a little bit of C++ to clearly understand how to use API functionality with Python.

HOW TO READ THE API DOCUMENTATION

When you need to know how to do something with the Maya API, the documentation is always the most important source of information. Being able to navigate the Maya API documents will allow you to work far more efficiently. Although this text does not attempt to teach C++ in great detail, we will identify and define some common terms that appear in the API documents to help reduce confusion.

To find API documentation, there are two basic options. The first option is to open Maya's HTML help documents, either from Maya's menu (**Help → Maya Help**) or on the Web.[8] The second option is to navigate to the actual C++ header files manually. Each file corresponds to a class in the API (e.g., MVector.h contains definitions for the **MVector** class). You can find these files in the following locations based on your operating system:

- **Windows:** Program Files\Autodesk\Maya<version>\include\maya\
- **OS X:** /Applications/Autodesk/maya<version>/devkit/include/maya/
- **Linux:** /usr/autodesk/maya<version>-x64/include/maya

In most cases, the HTML documentation will be the easiest method of finding help, as it is more thoroughly commented, easy to browse, and does not require the depth of C++ knowledge necessary to make sense of the header files. Unfortunately, however, depending on your version of Maya, the HTML help documents may not always be 100% accurate or complete. Many times, if you come across something that does not appear to be working as it should, a good first step in troubleshooting is to look at the actual header file in question and make sure it reflects what you see in the HTML documentation.[9] This test is useful because the header files are exactly what the C++ API uses. For now, however, we will focus on the HTML documentation, as it is generally the best source of information overall.

If you open the HTML help and scroll to the bottom of the table of contents in the left menu, depending on what version of Maya you are using, there is

[8]Visit the Autodesk web site at *http://www.autodesk.com* to find these documents.

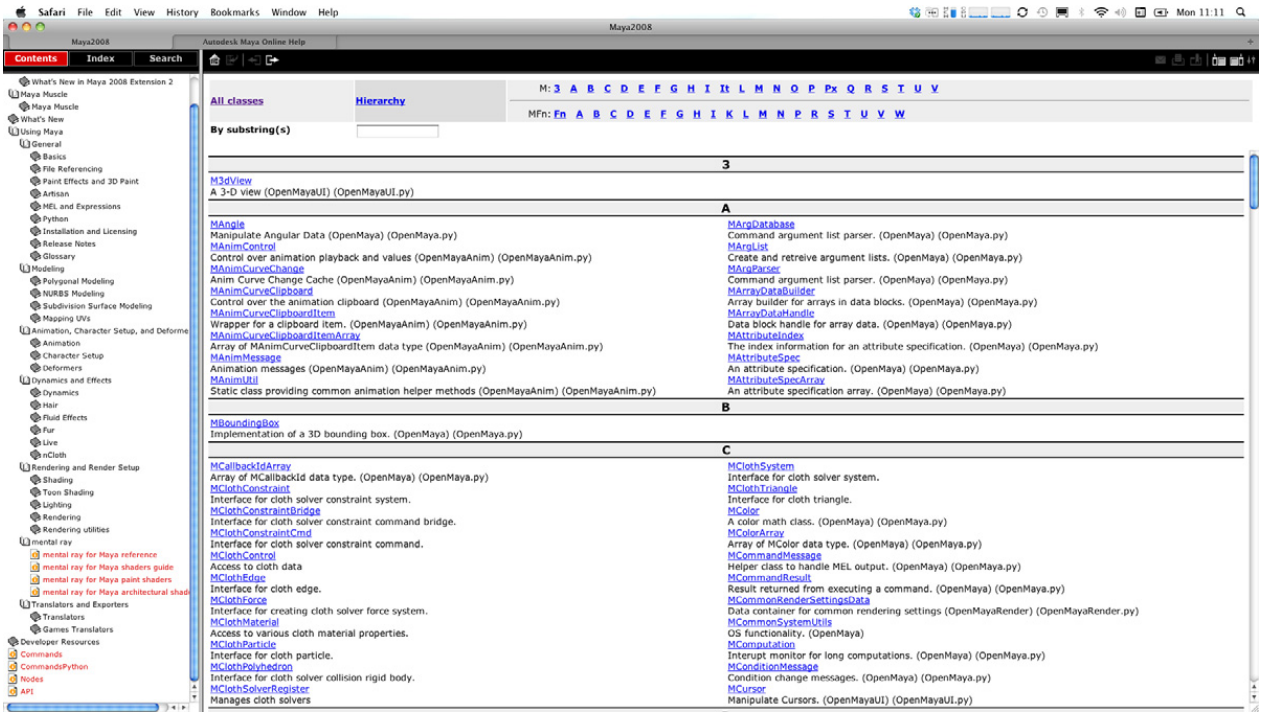
[9]Autodesk used a technology called doxygen to generate the newer versions of the documentation. Because doxygen requires that code be specially formatted to be correctly parsed, some API methods may seem to have disappeared in version 2009. Checking the actual header files confirms, however, that they still exist, and were simply not parsed out correctly. See, for instance, **MFnMesh::getUVShellsIds()**.

either an "API" link or a "Technical Documentation" section containing an "API reference" link. The pages for Maya 8.5 and 2008 share many commonalities, while 2009 was completely reformatted. The structures of each version are slightly different and worth briefly reviewing.

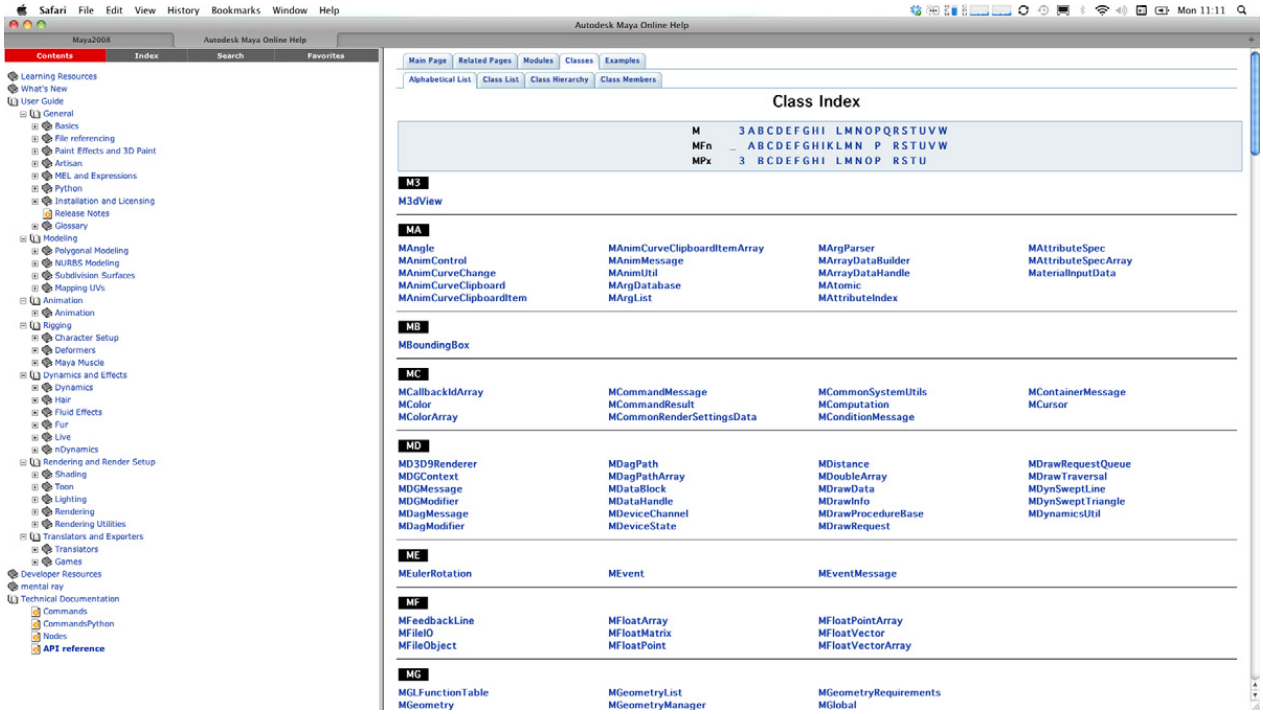
In Maya 8.5 and 2008, by default, there is a list of all of the API classes (Figure 9.4). At the top, you can click a link to browse by hierarchy or stay in the view containing all classes. In this view, everything is sorted alphabetically, with options in the header to search by first letter in the top right or by substring just underneath and to the left. The alphabetical search list is further split by category, with all M classes on top and MFn (function set) classes underneath.

In the main view of the page are links to each individual class. Each link also has a basic description of what the class is, as well as a parenthetical notation of which Python module allows access to the class. If you click on a link to a class, you are given a list of all of its members (data attributes) and methods, as well as descriptions of what they do.

The documentation for versions of Maya 2009 and later contains tabs in the header for navigating (Figure 9.5). As there is both a "Modules" tab



■ **FIGURE 9.4** The API documentation for Maya 2008.

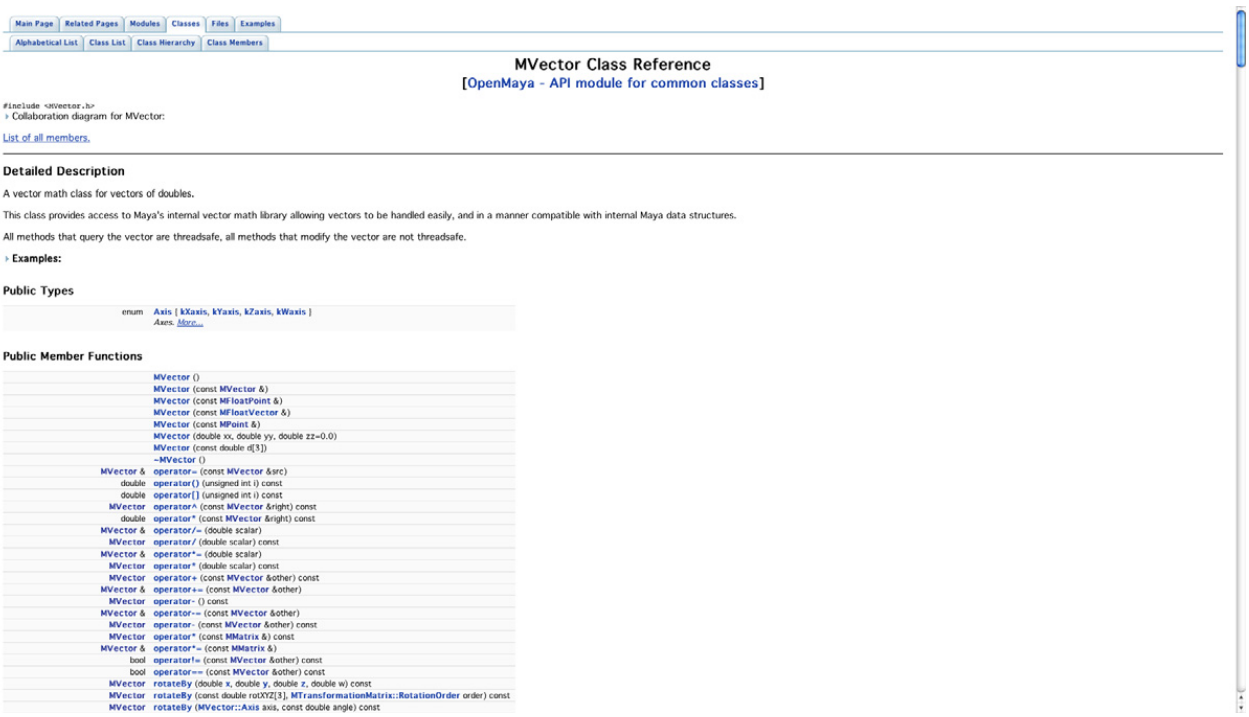


■ **FIGURE 9.5** The API documentation for Maya 2009.

and a "Classes" tab, you can navigate for information using either option. Clicking the "Modules" tab displays a list of all the Maya Python API modules. Clicking the name of one of these modules brings up a list of all API classes that are wrapped in it with a partial description of the class.

Clicking the "Classes" tab displays a browser that is more similar to the documentation in Maya 8.5 and 2008. At the top is a list of tabs that allow you to view the classes as an unannotated alphabetical list ("Alphabetical List"), an annotated alphabetical list ("Class List"), a hierarchically sorted alphabetical list ("Class Hierarchy"), or to see an alphabetical list of all class members with links to the classes in which they exist ("Class Members"). The "Class Members" section lets you further narrow your search by filtering on the basis of only variables, only functions, and so on. Now that we have discussed how to find the API classes, let's walk through an example to see some common terminology.

Whichever version of the Maya documentation you are using, navigate to the **MVector** class and click its link to open its page (Figure 9.6). At the top, you will see a brief description telling you what the **MVector** class



■ **FIGURE 9.6** The API documentation for the **MVector** class.

does: "A vector math class for vectors of doubles." Immediately below this description is a list of public types and public members.

The first item in this list is an enumerator called **Axis**. The **Axis** enumerator has four possible values: **kXAxis**, **kYAxis**, **kZAxis**, and **kWAxis**. Enumerators such as these are typically included in basic classes and are often required as arguments for some versions of methods (more on this later). Enumerated types preceded with the letter **k** in this fashion typically represent static constants (and are thus accessed similar to class attributes in Python). Execute the following code in a Python tab in the Script Editor.

```
import maya.OpenMaya as om;
print('X-Axis: %s'%om.MVector.kXaxis);
print('Y-Axis: %s'%om.MVector.kYaxis);
print('Z-Axis: %s'%om.MVector.kZaxis);
print('W-Axis: %s'%om.MVector.kWaxis);
```

As you can see, the value of each of these enumerators is an integer representing the axis in question, rather than an actual vector value.

```
X-Axis: 0
Y-Axis: 1
```

```
Z-Axis: 2
W-Axis: 3
```

Consequently, you are also able to test and set enumerator values in code using an integer rather than the name for the enumerator. Such shorthand is typically not recommended, however, as it makes the code harder to read later than does simply using logical names.

After this enumerator, the next several functions are constructors. These functions are all called **MVector()** and take a variety of arguments (see Figure 9.6). In C++, a function with a single name can be overloaded to work with different types of data or values as well as with different numbers of arguments. **MVector** has seven different constructor options, depending on what type of data you want to give it. The first constructor, **MVector()**, is the default constructor taking no arguments. This will initialize the members of the **MVector** object to their default values.

The second, third, fourth, and fifth constructors—**MVector (const MVector &)**, **MVector (const MFloatPoint &)**, **MVector (const MFloatVector &)**, and **MVector (const MPoint &)**—all follow a common pattern. Namely, they create a new **MVector** object and initialize its *x*, *y*, *z*, and *w* values from some other object's corresponding members. In our effort to understand C++, we should note two common features of these constructors: first, each argument is prefixed by the keyword **const**, and second, each argument is suffixed by an ampersand (**&**).

The **&** symbol means that the object is passed by reference; you can think of the **&** symbol as standing for "address of" something. This symbol is given because C++ programmers have the option to pass parameters by value or by reference. Passing an item by reference means that the caller and the method share the same copy of the data in memory. Passing by reference reduces runtime and memory overhead for complex types by eliminating the need to copy the data before passing it to the method. It also means that changes made to the data by the method will be visible to the caller (like mutable objects in Python).

The keyword **const** is short for "constant." It indicates that the method will only read the value and not modify it. If a parameter that is passed by reference is also marked as **const,** it is effectively the same as if the parameter were being passed by value, except that it is more efficient because the data are not copied.

Although Python does not use the same terminology, class types in Python are effectively passed by reference, while simple types are passed by value. Therefore, if a Maya API method specifies a parameter of a class type, such

as **MVector**, you can simply pass the value normally, as you would to any other Python method. It does not matter whether the method wants the parameter to be passed by reference or has it marked as **const**; the SWIG interface layer will take care of getting it to the API method correctly.

As we will soon see, however, the situation is a bit more complicated when working with simple, mutable data types. If an API method requests that a parameter of a simple type be passed by value (i.e., the parameter does not have the **&** qualifier), then you can just pass the value normally since Python natively passes simple types by value. Similarly, as noted earlier, a parameter that is both **const** and passed by reference is effectively the same as passing by value, so once again you can simply pass an item normally in such cases.

A problem arises, however, when an API method indicates that it wants a parameter of a simple type to be passed by reference but does not mark it as **const**. Such methods intend to change the value of that parameter and want the new value to also be reflected in the calling context. The SWIG interface layer unfortunately cannot work around this situation automatically because there is no mechanism built into Python for doing so. Instead, the Maya API provides a special class called **MScriptUtil** to work around these situations. We will soon see examples of this class in action.

The sixth constructor, **MVector (double xx, double yy, double zz = 0.0)**, creates a new **MVector** object with initial *x*, *y*, and *z* values corresponding to the three doubles passed in. The **double** data type in C++ is a decimal number with double precision. While a standard floating-point number (**float** type) requires 32 bits in memory, a double requires twice as much: 64 bits.[10] The equals sign (**=**) after the **zz** parameter indicates that this argument is optional and will assume the default value shown if nothing is supplied (just like a keyword argument in a Python definition). In this way, an **MVector** can be initialized with only two doubles using this constructor, which will be its initial *x* and *y* values, while its *z* will be the default 0.0.

The seventh and final constructor, **MVector (const double d[3])**, creates a new **MVector** object and initializes its *x*, *y*, and *z* values to the elements in

[10]Compare the **MFloatVector** class to the **MVector** class, for example. **MFloatVector** has essentially the same functionality, yet is composed of floats rather than doubles. Because floats allocate less memory, they are therefore suited to tasks where performance is more critical than precision. Floats are the most common decimal format used in real-time videogame applications.

the double array being passed in. The square brackets ([]) indicate that the data are an array with the length given inside of them. This parameter is not suffixed with the **&** symbol, which means that the array itself is not passed by reference. However, the *elements* of an array parameter in C++ are always effectively passed by reference. As we mentioned earlier, a parameter of a simple type that is passed by reference requires special handling, unless it is also marked as **const**. We have **const** in this case, so you may assume that should allow us to just pass in a Python list containing three decimal values.

```
import maya.OpenMaya as om;
doubleList = [1.0, 2.0, 3.0];
vec = om.MVector(doubleList);
```

Unfortunately, this approach results in an error message.

```
# Error: NotImplementedError: Wrong number of arguments for
overloaded function 'new_MVector'.
Possible C/C++ prototypes are:
MVector()
MVector(MVector const &)
MVector(MFloatPoint const &)
MVector(MFloatVector const &)
MVector(MPoint const &)
MVector(double,double,double)
MVector(double const [3]) #
```

Depending on what version of Maya you are using, you may see an error message like the following output instead.

```
# Error: TypeError: in method 'new_MVector', argument 1 of type
'double const [3]' #
```

Although our logic was sound in this case, the API's SWIG layer is not smart enough (as of Maya 2012) to make the conversion where arrays are concerned. This situation thus requires Maya's special **MScriptUtil** class, as in the following example.

```
import maya.OpenMaya as om;
# create a double array dummy object
doubleArray = om.MScriptUtil();
# populate the dummy with data from a list
doubleArray.createFromList([1.0, 2.0, 3.0], 3);
# create the vector by passing a pointer to the dummy
vec = om.MVector(doubleArray.asDoublePtr());
print('(%s, %s, %s)'%(vec.x, vec.y, vec.z));
```

You should see the value (1.0, 2.0, 3.0) printed to confirm that the vector was correctly created.

The **MScriptUtil** class is a handy, but somewhat confusing, way of interfacing with the API where it requires special representations of data that Python does not implement. Specifically, because Python always passes simple data types by value, a special tool is needed when the API requires arrays, pointers, or references of simple types like integers and decimals. The **MScriptUtil** class is briefly explained in more detail in the final section of this chapter, and explored in-depth on the companion web site.

At the end of the **MVector** class's list of constructors is also a destructor, prefixed with the tilde character (~). C++ uses functions like this one to deallocate memory when an object is deleted (just like the built-in **__del__()** method for class objects in Python). As mentioned in Chapter 2, however, Python handles memory management and garbage collection on its own. Consequently, manual deallocation is not strictly necessary, and may often only be beneficial in situations for which C++ is better suited than Python anyway.

Next, you should see a set of 18 items prefixed with **operator** followed by some symbol. These are operator overloads for the **MVector** class. Many of these symbols are normally reserved for working with basic data types like numbers and lists. However, they have been overridden for this class, meaning that they are available for shorthand use. Consider the following example.

```
import maya.OpenMaya as om;
vec1 = om.MVector(1.0, 0.0, 0.0);
vec2 = om.MVector(0.0, 1.0, 0.0);
# without operator overloading, you would have to add manually
vec3 = om.MVector(vec1.x + vec2.x, vec1.y + vec2.y, vec1.z +
vec2.z);
# with operator overloading, you can add the objects themselves
vec4 = vec1 + vec2;
# confirm that the results are the same
print('vec3: (%s, %s, %s)'%(vec3.x, vec3.y, vec3.z));
print('vec4: (%s, %s, %s)'%(vec4.x, vec4.y, vec4.z));
```

You should see from the output that both addition approaches produce the same result, but the overloaded operator is obviously preferable.

Operator overloading can also offer the advantage of bypassing the requirement of typing out method names for common operations. For instance, note that there are overloads for **operator^ (const MVector & right) const** and **operator* (const MVector & right) const**, which allow for quickly computing the cross-product and dot-product, respectively. A number of these overloads are also suffixed with the **const** keyword. When **const** appears at the end of a function declaration, it indicates that the function will not change any of the object's members. Note, in contrast, that the

in-place operators (e.g., **+=**, **-=**, ***=**, **/=**) do not have the trailing **const** keyword, as they will in fact alter the object's members. (Note that Python also allows a similar technique, as discussed in section 3.4.8 of the Python Language Reference.)

The next items in the **MVector** class are five different versions of the **rotateBy()** method, which rotates a vector by some angle and returns the rotated vector. Note that you can see the return type (**MVector**) to the left of the function's name. In the detailed description lower in the document, you can also see a double colon preceded by the class name (**MVector::rotateBy()**). This notation simply indicates that **rotateBy()** is an instance method. The following example illustrates an invocation of the first version, **rotateBy (double x, double y, double z, double w) const**, which rotates the vector by a quaternion given as four doubles.

```
import math;
import maya.OpenMaya as om;
vector = om.MVector(0.0, 0.0, 1.0);
# rotate the vector 90 degrees around the x-axis
halfSin = math.sin(math.radians(90*0.5));
halfCos = math.cos(math.radians(90*0.5));
rotatedVector = vector.rotateBy(halfSin, 0.0, 0.0, halfCos);
# confirm rotated vector is approximately (0.0, -1.0, 0.0)
print(
    '(%s, %s, %s)'%(
        rotatedVector.x,
        rotatedVector.y,
        rotatedVector.z
    )
);
```

As you can see, you simply need to call the **rotatedBy()** method on an instance of an **MVector** object, and the resulting rotated vector will be returned. The second version, **rotateBy (const double rotXYZ[3], MTransformationMatrix::RotationOrder order) const**, rotates the vector using three Euler angles with a given rotation order. The first parameter in this case is a double array, while the second is an enumerator defined in the **MTransformationMatrix** class. Clicking on the enumerator link, you can see that this enumerator can have eight different values: **kInvalid**, **kXYZ**, **kYZX**, **kZXY**, **kXZY**, **kYXZ**, **kZYX**, **kLast**. These enumerators are accessed just like the **Axis** enumerator within the **MVector** class.

```
import maya.OpenMaya as om;
print(om.MTransformationMatrix.kXYZ); # prints 1
```

The third version of the method, **rotateBy (MVector::Axis axis, const double angle) const**, rotates the vector using the axis angle technique.

The first parameter is one of the enumerated axes from the **MVector** class (**kXaxis**, **kYaxis**, or **kZaxis**), and the second parameter is the angle in radians by which to rotate around this axis.

The fourth and fifth versions of this method, **rotateBy (const MQuaternion &) const** and **rotateBy (const MEulerRotation &) const**, rotate the vector using Maya's special objects for storing rotation as either a quaternion or Euler angles, respectively.

The next function in the list is **rotateTo (const MVector &) const**. In contrast to the **rotateBy()** methods, **rotateTo()** returns a quaternion rather than a vector, as indicated by the return type of **MQuaternion**. This quaternion is the angle between the **MVector** calling the method and the **MVector** passed in as an argument.

The **get()** method transfers the *x*, *y*, and *z* components of the **MVector** into a double array. Again, because Python can only pass simple types by value, you must use the **MScriptUtil** class to utilize this method, as in the following example.

```
import maya.OpenMaya as om;
# create a double array big enough to hold at least 3 values
doubleArray = om.MScriptUtil();
doubleArray.createFromDouble(0.0, 0.0, 0.0);
# get a pointer to the array
doubleArrayPtr = doubleArray.asDoublePtr();
# create the vector
vec = om.MVector(1.0, 2.0, 3.0);
# use the get() method to transfer the components into the array
# there is no MStatus in Python, so try-except must be used
instead
try: vec.get(doubleArrayPtr);
except: raise;
# confirm that the values (1.0, 2.0, 3.0) were retrieved
print('(%s, %s, %s)'%(
    doubleArray.getDoubleArrayItem(doubleArrayPtr,0),
    doubleArray.getDoubleArrayItem(doubleArrayPtr,1),
    doubleArray.getDoubleArrayItem(doubleArrayPtr,2)
));
```

As you can see, this method acts on the double-array pointer in-place, rather than returning an array. Instead, the method returns a code from the **MStatus** class. This class, used throughout the API, allows C++ programmers to trap errors. For example, if this call created a problem, it would return the code **kFailure** defined in the **MStatus** class. However, because Python has strong error-trapping built in with **try-except** clauses, the Python API does not implement the **MStatus** class at all. Therefore,

Python programmers must use **try-except** clauses instead of **MStatus** to handle exceptions.

The next method, **length()**, simply returns a decimal number that is the magnitude of the **MVector** object.

The following two methods, **normal()** and **normalize()**, are similar, albeit slightly different. The first, **normal()**, returns an **MVector** that is a normalized copy of the vector on which it is called. The **normalize()** method, on the other hand, returns an **MStatus** code in C++ because it operates on the **MVector** object in-place. Again, nothing in Python actually returns an **MStatus** code since the class is not implemented.

The **angle()** method provides a shortcut for computing the angle (in radians) between the **MVector** from which it is called and another **MVector** supplied as an argument.

The next two methods, **isEquivalent()** and **isParallel()**, provide shortcuts for performing comparison operations between the **MVector** from which they are called and another supplied as an argument within a specified threshold. The former tests both the magnitude and direction of the vectors, while the second compares only the direction. Each method returns a Boolean value, making them useful in comparison operations.

The final method accessible to Python is **transformAsNormal()**, which returns an **MVector** object. While the details in the documentation provide an elegant technical explanation, it suffices to say that the **MVector** this method returns is equivalent to a vector transformed by the nontranslation components (e.g., only rotation, scale, and shear) of the **MMatrix** argument supplied.

The final two items listed are operator overloads for the parenthesis **()** and square brackets **[]** characters. You will notice that, in contrast to the similar operators listed earlier, these two operators are listed as having NO SCRIPT SUPPORT. The reason is because, as you can see, each of these two specific operator overloads returns a reference to a decimal number, **& double**, while the two corresponding operators that *are* supported in Python (listed with the math operators) return values. Consequently, the lack of script support does not mean that you cannot use these two symbol sequences at all, but rather that *you cannot use them to acquire a reference to a decimal number*. As discussed earlier, Python always passes simple types by value. Consequently, it has no support for references to simple types. For the purposes of Python programming, you can effectively ignore their lack of compatibility in this case.

KEY DIFFERENCES BETWEEN THE PYTHON AND C++ APIS

As our brief tour of the **MVector** class has shown, there are some various differences between the Python and C++ APIs. Although the Maya documentation provides a list of these items (**Developer Resources → API Guide → Maya Python API → Differences between the C++ Maya API and Maya Python API**), it is worth enumerating them here for the sake of convenience.

MString and MStringArray

The C++ API contains the **MString** and **MStringArray** classes for returning and working with strings. Python's far more robust string implementation and extensive string utilities render this class useless for Python programmers working with the Maya API. As such, anywhere in the API that indicates one of these types is necessary in fact accepts standard Python strings and lists of strings in their places. Because tuples are immutable, however, a tuple of strings may only be used where the **MStringArray** is passed by value or as a **const** reference.

MStatus

As indicated in our discussion of the **MVector** class, the **MStatus** class exists in the C++ API to confirm the success of many operations by returning a status code either as the return value of a function or as one of its parameters. Whereas Python natively incorporates robust exception handling via the **try-except** clause, the **MStatus** class has not been implemented in the Python API at all. Consequently, Python programmers must use **try-except** clauses instead of **MStatus**. Trying to use Python to extract a value from functions that return **MStatus** may actually crash some versions of Maya!

Void* Pointers

The C++ API makes use of void* pointers in various places including messages. Such pointers essentially point to raw memory of an unspecified type, making them ideal for handling different data types if a programmer knows precisely what to expect. Because these pointers are unspecified in C++, any Python object can be passed where a void* pointer is required.

Proxy Classes and Object Ownership

Some scripted objects like custom commands and nodes require the creation of proxy classes (with the MPx prefix). However, these classes must

give control to Maya when they are created to ensure that Python does not accidentally garbage collect an object to which Maya is actively pointing. This process is described in greater detail in Chapters 10 and 12.

Commands with Arguments

As we showed in the first part of the book, Maya commands can accept a variety of arguments. When creating custom commands that support flags and arguments, programmers have a variety of options in the API, such as **MArgList** objects. However, you must use an **MSyntax** object for parsing a command's arguments for its flags to work properly when invoked from the cmds module. *The same is true whether the command is programmed in C++ or Python; flags passed to the command will be lost when an **MArgList** object is used, if the command is invoked from Python*. A command containing no flags can still use an **MArgList** object, as can a command that will only be invoked from MEL. Therefore, it behooves both C++ and Python programmers to avoid **MArgList** if their command has flags and they want it to work from Python. The process of designing a command with custom syntax is described in greater detail in Chapter 10.

Undo

Because Python allows programmers to use the Maya API directly within a script, it is worth noting that doing so is not compatible with Maya's undo mechanism. As such, any scripts that modify the Dependency Graph in any way must be implemented within a scripted command to support undo functionality. Maya's undo and redo mechanisms are described in greater detail in Chapter 10.

MScriptUtil

As mentioned in the discussion of the **MVector** class, Python does not support references to simple types, such as integers and decimals. Imagined through the lens of a language like C++, Python technically passes all parameters by value. While the value of a class object in Python can be imagined as being equivalent to a reference, the same does not apply for simple types, of which the value corresponds to an actual number.

Because simple types are always passed by value as far as SWIG is concerned, it needs a way to extract a reference from these values. This process is accomplished by effectively collecting the data for the simple type(s) as an attribute (or several attributes) on a class object, which SWIG can then manipulate as a reference.

As such, functions that require numeric array parameters require the invocation of an **MScriptUtil** object to act as a pointer. The **MScriptUtil** class contains methods for converting Python objects into a variety of different pointer and array types required throughout the C++ API. Using the **MScriptUtil** class can be fairly cumbersome at times, but it is unfortunately the only means of accessing many functions in the C++ API. See the companion web site for more information on using **MScriptUtil**.

CONCLUDING REMARKS

Maya's C++ API contains many classes and function sets that are accessible through various Python modules. These modules use SWIG to automatically convert Python parameters into data for C++, as well as to bring the data back into Python. Because of this intermediate layer, as well as a lack of perfect correspondence between C++ and Python, programmers working with the Python API must have some basic understanding of C++ to read the documentation, understand some limitations they will face, and better grasp classes like **MScriptUtil**. Though it is somewhat different from ordinary Python scripting in Maya, the Maya Python API provides an accessible way to rapidly develop powerful tools for Maya. The remainder of the book focuses on various uses of the Maya Python API.