Chapter 2

Python Data Basics

CHAPTER OUTLINE

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00002-8

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

- Define and manipulate data in variables.
- Compare and contrast Python's and MEL's typing systems.
- Recast variables as different data types.
- Prevent naming conflicts with built-in keywords.
- Explain how Python manages memory.
- Compare and contrast mutable and immutable types.
- Use variables in conjunction with commands to manipulate attributes.
- Describe the different numeric types in Python.
- Use mathematical operators with numeric types.
- Use logical and bitwise operators with Boolean variables.
- Use common operators with sequence types.
- Manipulate nested lists.
- Create Unicode and raw strings.
- Format strings that contain variable values.
- Compare and contrast sets and dictionaries with sequence types.

In Chapter 1, we covered executing commands in Maya using Python. To do anything interesting, however, we need more tools to create programs. In this chapter we will explore some of the basics for working with variables in Python. Because much of this information is available in the online Python documentation, we will not belabor it a great deal. However, because we make use of certain techniques throughout the text, it is critical that you have an overview here.

We begin by discussing variables in the context of the data types you learned in Chapter 1, and show how you can use variables along with some basic Maya commands. Thereafter, we discuss some more interesting properties of the basic sequence types we discussed in Chapter 1. Finally, we discuss two additional, useful container types.

VARIABLES AND DATA

The basic unit of data storage in any programming language is a variable. A *variable* is a name that points to some specific data type. The mechanism for creating a variable in Python is the assignment operator (=). Because Python does not require (or allow) declaration without an assignment, creating a variable is as simple as separating a name and a value with the assignment operator.

1. Execute the following line in a Python tab in the Script Editor, or from the Command Line. This line simply creates a variable named `contents`, and assigns a string value to it.

```
contents = 'toys';
```

2. Once you create this variable, you can substitute in this name anywhere you would like this value to appear. Execute the following line of code. You should see the string "toys in the box" print on the next line.

```
print(contents+' in the box');
```

3. You can change the value assigned to this variable at any time, which will substitute in the new value anywhere the name appears. Execute the following lines of code, which should print "shoes in the box" in the History Panel.

```
contents = 'shoes';
print(contents+' in the box');
```

4. You can also assign completely different types of values to the variable. Execute the following line of code.

```
contents = 6;
```

Python is what is called a strong, dynamically typed language. The line of code in step 4 demonstrates dynamic typing. You are able to change the type of any Python variable on-the-fly, simply by changing its assignment value. However, because it is strongly typed, you cannot simply add this new value to a string.

5. Try to execute the following statement.

```
print('there are '+contents+' things in the box');
```

The console should supply an error message like the following one.

```
# Error: TypeError: file <maya console> line 1: cannot
concatenate 'str' and 'int' objects #
```

Because Python is strongly typed, you cannot so easily intermix different types of variables. However, you can now perform addition with another number.

6. Execute the following line of code, which should print the number 16.

```
print(contents+10);
```

In Python, to intermix types, you must explicitly recast variables as the expected type.

7. Execute the following line in Python to cast the number stored in `contents` to its string representation. You should see "there are 6 things in the box" in the History Panel.

```
print('there are '+str(contents)+' things in the box');
```

In this case, we called the **str()** function to recast `contents` as a string. We discuss functions in greater detail in Chapter 3. Table 2.1 lists some other built-in functions for recasting variables.

Python provides a built-in function, **type()**, which allows you to determine the type of a variable at any point.

8. Execute the following line to confirm the type of the `contents` variable.

```
type(contents);
```

You should see the following line in the History Panel, which indicates that the variable is currently pointing to an integer.

```
# Result: <type 'int'> #
```

As you can see, casting the variable to a string as you did in step 7 did not change its inherent type, but only converted the value we retrieved from the variable. As such, you would still be able to add and subtract to and from this variable as a number. You could also reassign a string representation to it by assigning the recast value.

9. Execute the following lines to convert the variable to a string.

```
contents = str(contents);
print(type(contents));
```

You should see the following output in the History Panel.

```
<type 'str'>
```

Table 2.1 Python Functions for Recasting Variable Types

Function	Casts To	Notes
float()	Decimal number	If argument is string, raises **ValueError** if not properly formatted
int()	Integer number	If argument is string, raises **ValueError** if not properly formatted If argument is decimal number, result is truncated toward 0 Allows optional argument to specify non-base 10
str()	String	
unicode()	Unicode string	Allows optional argument to specify encoding

Variables in MEL

Variables in Python are incredibly flexible. Compare the previous example with a comparable MEL snippet.

1. Enter the following lines in a MEL tab in the Script Editor. You should again see the output "toys in the box" in the History Panel.

```
$contents = "toys";
print($contents+" in the box");
```

While MEL allows—but does not require—that you explicitly provide the variable's type, the variable `$contents` is statically typed at this point. It is a string, and so cannot now become a number. Those unfamiliar with MEL should also note that it requires variable names to be prefixed with a dollar sign (**$**).

2. Execute the following lines in MEL.

```
$contents = 6;
print("there are "+$contents+" things in the box");
```

Because you get the output you expect ("there are 6 things in the box"), you may think that MEL has implicitly handled a conversion in the print call. However, the number 6 stored in `$contents` is not in fact a number, but is a string. You can confirm this fact by trying to perform addition with another number.

3. Execute the following line in MEL.

```
print($contents+10);
```

Whereas the Python example printed the number 16 in this case, MEL has printed 610! In MEL, because the type of the variable cannot change, MEL implicitly assumed that the number 10 following the addition operator (+) was to be converted to a string, and the two were to be concatenated. While seasoned MEL developers should be well aware of this phenomenon, readers who are new to Maya will benefit from understanding this difference between MEL and Python, as you may occasionally need or want to call statements in MEL.

On the other hand, all readers who are as yet unfamiliar with Python would do well to remember that variable types could change on-the-fly. This feature offers you a great deal of flexibility, but can also cause problems if you're frequently reusing vague, unimaginative variable names.

Keywords

Apart from issues that may arise from vague variable names, Python users have some further restrictions on names available for use. Like any other language, Python has a set of built-in keywords that have special meanings.

You saw one of these keywords—**import**—in Chapter 1. To see a list of reserved keywords, you can execute the following lines in a Python tab in the Script Editor.

```
import keyword;
for kw in keyword.kwlist: print(kw);
```

The list of reserved keywords printed out are used for various purposes, including defining new types of objects and controlling program flow. We will be covering a number of these keywords throughout the text, but it is always a good idea to refer to the Python documentation if you would like more information. The important point right now is that these words all have special meanings, and you cannot give any of these names to your variables.

Python's Data Model

It is now worth highlighting a few points about Python's data model, as it has some bearing on other topics in this chapter (as well as many topics we discuss later in this book). Although we must be brief, we recommend consulting Section 3.1 of Python Language Reference online if you are interested in more information.

To maximize efficiency, statically typed languages allocate a specific amount of memory for different types of variables. Because this amount of memory is specified in advance, you cannot simply assign a new type to a name at some later point. On the other hand, as you have seen in this chapter, Python lets you change the type of a variable at any point. However, the underlying mechanisms are actually somewhat subtler.

In Python, variables are just names that point to data. All data are objects, and each object has an identity, a type, and a value.

- An object's *identity* describes its address in memory.
- An object's *type* (which is itself an object) describes the data type it references.
- An object's *value* describes the actual contents of its data.

While we will discuss objects in greater detail in Chapter 5, programmers coming from other languages may find this principle novel.

When you create a new variable and assign it a value, your variable is simply a name pointing to an object with these three properties: an identity, a type, and a value.[1] Consider a situation where you cast an integer to a string, such as the following lines.

```
var = 5;
var = str(var);
```

In this case, you are not actually altering the data's underlying type, but are pointing to some different piece of data altogether. You can confirm this fact by using the built-in **id()** function, which provides the address to the data. For instance, printing the identity at different points in the following short sample will show you different addresses when the variable is an integer and when it is a string.

```
var = 5;
print('int id',id(var));
var = str(var);
print('str id',id(var));
```

Mutability

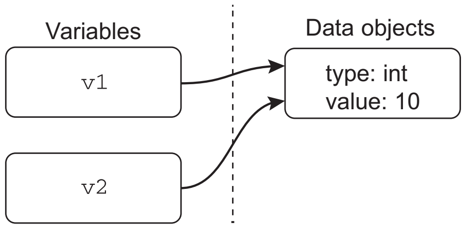
In Python, objects can be either mutable or immutable. Briefly construed, mutable objects can have their values changed (mutated), while immutable objects cannot.[2] We briefly mentioned this concept in Chapter 1 when comparing lists and tuples.

In fact, tuples, strings, and numbers are all immutable. As a consequence, when you assign a new integer value to a variable, instead of changing the underlying value of the object to which the variable is pointing, the variable instead points to another piece of data. You can see the effects of this concept in the following short code example, which will print the identity for the variable after different integer assignments.

```
var = 5;
print('5 id',id(var));
var = 6;
print('6 id',id(var));
```

[1]C++ programmers should note that although Python variables are references to data, they cannot simply be used like pointers. To shoehorn Python into the language of C++, it always passes parameters by value, but the value of a variable in Python is a reference. The consequence is that reassigning a variable inside of a function has no effect on its value outside the function. Chapter 9 discusses these consequences of Python's data model in Maya's API.

[2]Immutable containers that reference mutable objects can have their values changed if the value of one of the mutable objects in the container changes. These containers are still considered immutable, however, because identities to which they refer do not change. See the "Nested Lists" section in this chapter for more information.



■ **FIGURE 2.1** Two variables may point to the same object.

We examine the effects of mutability on sequence types later in this chapter.

Reference Counting

As part of its data model, Python uses a system known as reference counting to manage its memory. The basic idea is that rather than requiring developers to manually allocate and deallocate memory, data are garbage collected when there are no more names referencing them.

An interesting side effect of this paradigm is that two immutable variables with the same assignment (e.g., data with the same type and value) may in fact be pointing to the same data in memory (Figure 2.1). You can see this principle in action by assigning the same numeric value to two different variables and printing their identities.

```
v1 = 10;
v2 = 10;
print('v1 id', id(v1));
print('v2 id', id(v2));
```

In this case, both `v1` and `v2` are pointing to the same piece of data. If these two variables are assigned different data (e.g., some other number, a string, etc.), then the reference count for their previous data (the integer 10) drops to zero. When the reference count for objects drops to zero, Python normally automatically garbage collects the data to free up memory as needed. Although this concept has only a few consequences for our current discussion, it becomes more important in later chapters as we discuss modules, classes, and using the API.

It is important to note that while variables pointing to data with an immutable type may show this behavior, two *separate* assignments to mutable objects with the same type and value are always guaranteed to be different. For example, even though the following lists contain the same items, they will be guaranteed to have unique identities.

```
list1 = [1, 2, 3];
list2 = [1, 2, 3];
print('list1 id', id(list1));
print('list2 id', id(list2));
```

The assignment of one variable to another variable pointing to a mutable object, however, results in both pointing to the same data.

```
list1 = [1, 2, 3];
list2 = list1;
print('list1 id', id(list1));
print('list2 id', id(list2));
```

This concept has important consequences that we will cover later.

del()

Python has a built-in function, **del()**, which allows you to delete variables. Note that this process is not the same as deleting the data referenced by the variable: Python's garbage collector will still manage those data. Using this function with a variable name simply clears the name (and thus eliminates a reference to its data). The following example illustrates that, even though `v1` and `v2` will reference the same data, deleting `v1` has no effect on `v2`. Trying to access `v1` after this point would result in a **NameError**.

```
v1 = 5;
v2 = 5;
del(v1);
print(v2);
```

The None Type

Because we use it throughout the text in some cases, it is worth noting that, because of how Python's variables work, Python also implements a None type.

```
var = None;
```

One use of this type is to initialize a name without wastefully allocating memory if it is unnecessary. For instance, many of our API examples in this text use the None type to declare names for class objects whose values are initialized elsewhere. We will talk more about class objects starting in Chapter 5.

USING VARIABLES WITH MAYA COMMANDS

As we noted in Chapter 1, a Maya scene is fundamentally composed of nodes and connections. Each node has a number of different attributes, such as the radius of a **polySphere**, the height of a **polyCube**, or the maximum number of influences in a **skinCluster**. Although we will talk about attributes in much greater detail when we introduce the API in later chapters, it suffices for now to say that they describe the actual data housed in a node.

You have already seen some multimodal, built-in commands designed to work with different node types. For example, the `polySphere` command not only creates nodes for a polygon sphere, but also allows you to query and edit **polySphere** nodes. You can use these commands along with variables very easily. Let's work through a quick example.

1. Create a new Maya scene and execute the following lines in the Script Editor.

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

Recall that the results of the `polySphere` command return a list of object names (a **transform** node and a **shape** node). You can store this result in a variable when you create a cube. The `sphereNodes` list now contains the names of the two objects that were created. If you were to print this list, you would see something like the following line.

```
[u'pSphere1', u'polySphere1']
```

Note that this list does not contain the Maya nodes themselves, but simply contains their names. For example, using the **del()** function on this list would not actually delete Maya nodes, but would simply delete a list with two strings in it.

2. We will discuss this syntax later in this chapter, but you can use square bracket characters to access items in this list. For example, you can store the name of the **polySphere** node ("polySphere1") in another variable.

```
sphereShape = sphereNodes[1];
```

3. Now that you have stored the name of the shape in a variable, you can use the variable in conjunction with the `polySphere` command to query and edit values. For example, the following lines will store the sphere's radius in a variable, and then multiply the sphere's radius by 2. Remember that in each flag argument, the first name is the flag, and the second name is the value for the flag.

```
rad = maya.cmds.polySphere(
    sphereShape, q=True, radius=True
);
maya.cmds.polySphere(sphereShape, e=True, radius=rad*2);
```

4. You could now reread the **radius** attribute from the sphere and store it in a variable to create a cube the same size as your sphere.

```
rad = maya.cmds.polySphere(
    sphereShape, q=True, radius=True
);
maya.cmds.polyCube(
    width=rad*2,
    height=rad*2,
    depth=rad*2
);
```

Hopefully you see just how easy it is to use variables in conjunction with Maya commands.

Capturing Results

In practice, it is critical that you always capture the results of your commands in variables rather than making assumptions about the current state of your nodes' attributes. For example, Maya will sometimes perform validation on your data, such as automatically renaming nodes to avoid conflicts.

Some Maya users may insist that this behavior makes certain tools impossible without advanced, object-oriented programming techniques. In reality, you simply need to take care that you use variables to store your commands' results, and do not simply insert literal values—especially object names—into your code. The following example illustrates this point.

1. Create a new Maya scene and execute the following lines in the Script Editor. This code will create a sphere named "head."

```
import maya.cmds;
maya.cmds.polySphere(name='head');
```

2. Now execute the following line to try to make a cube with the same name.

```
maya.cmds.polyCube(name='head');
```

If you actually look at your cube in the scene, you can see that Maya has automatically renamed it to "head1" so it will not conflict with the name you gave the sphere. However, if you were writing a standalone tool that creates objects with specific names and then tries to use those specific names, your artists may run into problems if they already have objects in the scene with conflicting names.

3. Try to execute the following line, and you will get an error.

```
maya.cmds.polyCube('head', q=True, height=True);
```

Always remember that Maya commands work with nodes and attributes on the basis of their names, which are simply strings. When using ordinary Maya commands, you should always capture the results of your commands since there is no default mechanism for maintaining a reference to a specific node. In Chapter 5 we will see how the pymel module provides an alternative solution to this problem.

`getAttr` and `setAttr`

While there are many commands that pair with common node types, not all nodes have such commands (and sometimes not all attributes have flags). It may also become tedious to memorize and use commands with modes, flags, and so on. Fortunately, Maya provides two universal commands for getting and setting attribute values that will work with any nodes: `getAttr` and `setAttr`. We'll demonstrate a quick example.

1. Open a new Maya scene and execute the following lines of code. These lines will create a new locator and store the name of its **transform** node in a variable called `loc`.

   ```
   import maya.cmds;
   loc = maya.cmds.spaceLocator()[0];
   ```

2. Execute the following lines to store the locator's *x*-scale in a variable and print the result, which will be 1 by default.

   ```
   sx = maya.cmds.getAttr(loc+'.scaleX');
   print(sx);
   ```

The `getAttr` command allows you to get the value of any attribute on any node by simply passing in a string with the node's name, a period, and the attribute name. We'll talk more about working with strings shortly, but the complete string that we passed to the `getAttr` command was actually "locator1.scaleX" in this case.

3. Execute the following lines to double the `sx` value and assign the new value to the node's attribute.

   ```
   sx *= 2;
   maya.cmds.setAttr(loc+'.scaleX', sx);
   ```

The `setAttr` command works just like the `getAttr` command, and allows you to set any attribute value on any node by passing a string with the node's name, a period, and the attribute name.

Compound Attributes

Many attributes will simply be a string, a Boolean, or a number value. However, some attributes are called compound attributes, and may contain several values. It is important to note that these attribute types work differently with the `getAttr` and `setAttr` commands compared to other built-in commands.

The reason for this difference is that the `getAttr` and `setAttr` commands do not know what data type the particular attribute expects or contains until they look it up by name. Other commands only work with specific attributes on specific nodes, and so can work in a more straightforward way, as you will see in the remainder of this example.

4. In the scene you created in the previous steps, execute the following line to print the locator's translation using the `xform` command.

   ```
   print(maya.cmds.xform(loc, q=True, translation=True));
   ```

   As you would expect the result is simply a list: [0.0, 0.0, 0.0].
5. Likewise, when using the `xform` command, you can *set* a new translation value using a list, as in the following line.

   ```
   maya.cmds.xform(loc, translation=[0,1,0]);
   ```

The `xform` command can work in this way because it is designed to work exclusively with **transform** nodes, and the command internally knows about the data type for the appropriate attribute. On the other hand, `getAttr` and `setAttr` are not so straightforward.

6. Execute the following line to print the locator's translation using the `getAttr` command.

   ```
   print(maya.cmds.getAttr(loc+'.translate'));
   ```

   As you can see from the output, the command returns a list that contains a tuple: [(0.0, 1.0, 0.0)].
7. Using `setAttr` for a compound attribute also uses a different paradigm. Execute the following line to set a new translation value for the locator.

   ```
   maya.cmds.setAttr(loc+'.translate', 1, 2, 3);
   ```

As you can see, setting a compound attribute like translation using `setAttr` requires that you specify each value in order (*x*, *y*, *z* in this case).

`connectAttr` and `disconnectAttr`

The mechanism for transforming data in the Maya scene is to connect attributes: an output of one node connects to some input on another node. For instance, you saw in Chapter 1 how the **output** attribute of a **polySphere** node is connected to the **inMesh** attribute of a **shape** node when you execute the `polySphere` command. While Chapter 1 showed how you can delete the construction history of an object to consolidate its node network, the `connectAttr` and `disconnectAttr` commands allow you to reroute connections in altogether new ways.

The basic requirement for attributes to be connected is that they be of the same type. For example, you cannot connect a string attribute to a decimal number attribute. However, Maya does perform some built-in conversions for you automatically, such as converting angle values into decimal numbers. The finer distinctions between these types of values will be clearer when we discuss the API later in this book. In the following short example, you will create a basic connection to control one object's translation with another object's rotation.

1. Open a new Maya scene and execute the following lines to create a sphere and a cube and store the names of their **transform** nodes.

```
import maya.cmds;
sphere = maya.cmds.polySphere()[0];
cube = maya.cmds.polyCube()[0];
```

2. Execute the following lines to connect the cube's *y*-rotation to the sphere's *y*-translation.

```
maya.cmds.connectAttr(cube+'.ry', sphere+'.ty');
maya.cmds.select(cube);
```

Use the Rotate tool (**E**) to rotate the cube around its *y*-axis and observe the behavior. The result is very dramatic! Because you are mapping a rotation in degrees to a linear translation, small rotations of the cube result in large translations in the sphere.

It is also worth mentioning that the connection is only one way. You cannot translate the sphere to rotate the cube. In Maya, without some very complicated hacks, connections can only flow in one direction. Note, too, that an output attribute can be connected to as many inputs as you like, but an input attribute can only have a single incoming connection.

3. Execute the following line to disconnect the two attributes you just connected.

```
maya.cmds.disconnectAttr(cube+'.ry', sphere+'.ty');
```

The `disconnectAttr` command works very similarly to the `connectAttr` command. It simply expects two strings that name the nodes and their attributes.

4. Execute the following lines to create a **multiplyDivide** node between the two attributes to scale the effect.

```
mult = maya.cmds.createNode('multiplyDivide');
maya.cmds.connectAttr(cube+'.ry', mult+'.input1X');
maya.cmds.setAttr(mult+'.input2X', 1.0/90.0);
maya.cmds.connectAttr(mult+'.outputX', sphere+'.ty');
maya.cmds.select(cube);
```

Now if you rotate the cube, the sphere translates 1 unit for every 90 degrees of rotation.

WORKING WITH NUMBERS

In Chapter 1, we introduced some basic object types with which Maya commands have been designed to work. Although we lumped numbers together generally, they are actually decomposable into more specific types. Because you may want to take advantage of these specific types' features, and because you may encounter them when testing the types of your variables, it is worth briefly discussing them.

Number Types

Although you can often intermix different types of numbers in Python, there are four types of numbers: integers, long integers, floating-point numbers, and complex numbers. We briefly cover them here, but you can read more about these different types in Section 5.4 of Python Standard Library.

As you saw in the beginning of this chapter, an integer is simply a whole (nonfractional) number. Integers can be positive or negative. The type of an integer in Python is given as int, as the following code illustrates.

```
var = -5;
print(type(var));
```

A long integer differs from an integer only in that it occupies more space in memory. In many cases, ordinary integers suffice and are more efficient, but long integers may be useful for computation that deals with large numbers. In a language like C or C++, a long integer occupies twice as many bits in memory as an ordinary integer. In Python, a long integer can occupy as much memory as you require. To create a long integer, you can simply assign a really long value to your variable and it will become a long integer automatically, or you can suffix the value with the character **l** or **L**. The type in Python is given as long.

```
var = -5L;
print(type(var));
```

Floating-point numbers are any numbers, positive or negative, with digits after a decimal point. You can explicitly indicate that a whole number value is to be a floating-point number by adding a decimal point after it. The type of a floating-point number is float.

```
var = -5.0;
print(type(var));
```

Finally, Python allows you to use complex numbers, which consist of both a real and an imaginary component. It is highly unlikely that you will need to work with complex numbers regularly. You can create a complex number by suffixing the imaginary part with the character **j** or **J**. The type is given as complex.

```
var = -5+2j;
print(type(var));
```

You can also access the individual real and imaginary parts of complex numbers using the **real** and **imag** attributes.

```
print(var.real, var.imag);
```

Basic Operators

Once you start creating variables, you will want to do things with them. Operators are special symbols you can include in your code to perform special functions. (Note that we also include some related built-in functions and object methods in our tables.) Section 5 of Python Standard Library includes information on various operators, and Section 3.4 of Python Language Reference contains information on overloading operators, which allows you to assign special functionality to these symbols. Here, we briefly review some of the basic operators for working with numeric values.

Thus far, you've seen some of the basic math operators, such as **+** for addition and – for subtraction or negation. Section 5.4 in Python Standard Library contains a table of Python's built-in operators for working with numeric types. In Table 2.2 we have recreated the parts of this table containing the most common operators.

Note that many of these operators also allow in-place operations when used in conjunction with the assignment operator (**=**). For example, the following

Table 2.2 Important Numeric Operators

Operation	Result	Notes
x **+** *y*	Sum of *x* and *y*	
x **−** *y*	Difference of *x* and *y*	
x ***** *y*	Product of *x* and *y*	
x **/** *y*	Quotient of *x* and *y*	If *x* and *y* are integers, result is rounded down
x **//** *y*	Floored quotient of *x* and *y*	Use with floating-point numbers to return a decimal result identical to *x*/*y* if *x* and *y* were integers
x **%** *y*	Remainder of *x* / *y*	
divmod(*x*, *y*)	Tuple that is (*x* // *y*, *x* % *y*)	
pow(*x*, *y*)	*x* to the *y* power	
x ****** *y*	*x* to the *y* power	

lines create a variable, `v1`, with a value of 2, and then subtracts 4 from the same variable, resulting in −2.

```
v1 = 2;
v1 -= 4;
print(v1);
```

Note that Python does not support incremental operators such as **++** and **−−**.

For those readers who are new to programming, it is important to point out how division works with integer numbers. Remember that integers are whole numbers. Consequently, the result is always rounded down. The following floating-point division results in 0.5.

```
1.0/2.0;
```

The following integer division results in 0.

```
1/2;
```

On the other hand, the following integer division results in −1.

```
-1/2;
```

You can also mix an integer and a floating-point number, in which case both are treated as floats. The following division results in 0.5.

```
1.0/2;
```

It is also worth mentioning, for those unacquainted with the finer points of computer arithmetic, that floating-point operations often result in minor precision errors as a consequence of how they are represented internally. It is not uncommon to see an infinitesimally small number where you might actually expect zero. These numbers are given in scientific notation, such as the following example.

```
-1.20552649145e-10
```

WORKING WITH BOOLEANS

Another basic type that is used with Maya commands is the Boolean type (called bool in Python). Recall that in practice, Boolean values True and False are interchangeable with 1 and 0, respectively. This principle becomes more important when using conditional statements, as you will see in Chapter 3.

Boolean and Bitwise Operators

Some operators are especially important when you are working with Boolean values. You will use these operators widely in Chapter 3 as we discuss using conditional statements to control program flow. In Table 2.3

Table 2.3 Important Boolean and Bitwise Operators

Operation	Result	Notes
x **or** *y*	True if either *x* or *y* is True	Only evaluates *y* if *x* is False
x **and** *y*	True only if both *x* and *y* are True	Only evaluates *y* if *x* is True
not *x*	False if *x* is True; True if *x* is False	
x **|** *y*	Bitwise **or** of *x* and *y*	
x **&** *y*	Bitwise **and** of *x* and *y*	
x **^** *y*	Bitwise exclusive-or of *x* and *y*	

we have consolidated the most common operators from tables in Sections 5.2 and 5.4.1 of Python Standard Library.

When working with Boolean variables, the bitwise operators for and (&) and or (|) function just like their Boolean operator equivalents. The exclusive-or operator (^), however, will only return True if either *x* or *y* is True, but will return False if they are both True or both False.

There are more bitwise operators that we have not shown here only because we do not use them throughout this text. Programmers who are comfortable with binary number systems should be aware that bitwise operators also work with int and long types.[3]

WORKING WITH SEQUENCE TYPES

In Chapter 1, three of the variable types introduced can be called sequence types. Sequence types include strings, lists, and tuples. These types basically contain a group of data in a linear sequence. While each of these types is obviously unique, they also share some properties, which we briefly discuss in this section.

Operators

As with numbers and Booleans, sequence types have a set of operators that allow you to work with them conveniently. Section 5.6 of Python Standard Library has a table of operations usable with sequence types. We have selected the most common of these operations to display in Table 2.4. Because they are perhaps not as immediately obvious as math operators, some merit a bit of discussion.

[3]It is also worth noting that the bitwise inversion operator (~) is not equivalent to the Boolean **not** operator when working with Boolean variables. Booleans are not simply 1-bit values, but are built on top of integers. Consequently, ~False is −1 and ~True is −2.

Table 2.4 Important Sequence Operators

Operation	Result	Notes
x **in** *s*	True if *x* is in *s*	Searches for item in lists/tuples Searches for character sequence in strings
x **not in** *s*	True if *x* is not in *s*	(see **in** operator)
s + *t*	Concatenation of *s* and *t*	
s[*i*]	*i*th item in *s*	First index is 0 Negative value for *i* is relative to len(s)
s[*i*:*j*]	Slice of *s* from *i* to *j*	(see index operator) *i* is starting index, *j* is end of slice If *i* is omitted, *i* is 0 If *j* is omitted or greater than len(s), *j* is len(s)
s[*i*:*j*:*k*]	Slice of *s* from *i* to *j* with step *k*	(see index operator)
len(*s*)	Length of *s*	
min(*s*)	Smallest item in *s*	Corresponds to ASCII code for strings
max(*s*)	Largest item in *s*	Corresponds to ASCII code for strings
s.**index**(*x*)	Index of first *x* in *s*	
s.**count**(*x*)	Total occurrences of *x* in *s*	

Concatenation

The first operator worthy of a little discussion is the concatenation operator (**+**). As you saw in the examples at the outset of this chapter, MEL concatenated two strings that we expected to be numbers. Python allows sequence concatenation in the same way. Concatenation creates a new sequence composed of all of the elements of the first sequence, followed by all of the elements in the second sequence. You can concatenate any sequence types, but only with other sequences of the same type. You cannot concatenate a list with a tuple, for instance.

The following line produces the string "Words make sentences."

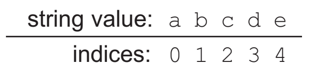
```
'Words' + ' ' + 'make' + ' ' + 'sentences.';
```

Likewise, the following line produces the list [1, 2, 3, 4, 5, 6].

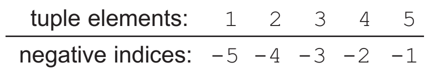
```
[1,2,3] + [4,5,6];
```

Indexing and Slicing

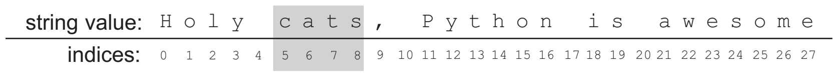
An important operator for sequence types is the index operator, represented by the square bracket characters (**[** and **]**). At its most basic, it corresponds to the index operator found in most languages. Note also that sequence types all use zero-based indices (Figure 2.2). For example, the following line results in just the "**c**" character, as it occupies index 2 in the string.



■ **FIGURE 2.2** Sequences use zero-based indices.



■ **FIGURE 2.3** Negative indices are relative to sequence length.


string value: H o l y c a t s , P y t h o n i s a w e s o m e
indices: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

■ **FIGURE 2.4** Slicing a string from index 5 to 9.

```
'abcde'[2];
```

Because lists are mutable, you can also use this operator to set individual values for a list. The following line would result in the list [0, 2, 3].

```
[1,2,3][0] = 0;
```

However, because they are immutable, you cannot use the index operator to change individual items in a tuple or characters in a string. The following two lines would fail.

```
'abcde'[2] = 'C';
(1,2,3)[0] = 0;
```

You can use a series of index operators to access elements from sequences embedded in sequences. For instance, the following example will extract just the "**x**" character.

```
('another', 'example')[1][1];
```

Another important feature of Python's index operator is that it allows you to supply a negative index. Supplying a negative index gives the result relative to the length of the sequence (which is one index beyond the final element; Figure 2.3). For example, the following line would print the number 4.

```
print((1,2,3,4,5)[-2]);
```

Python's index operator also offers powerful, concise syntax for generating slices from sequences. You can think of a slice as a chunk extracted from a sequence. The most basic syntax for a slice includes two numbers inside the square brackets, separated by a colon. The first number represents the start index, and the second number represents the end index of the slice (Figure 2.4). Consider the following example.

```
var = 'Holy cats, Python is awesome';
```

You could print just the word “cats” using the following slice. Note that the letter “s” itself is index 8.

```
print(var[5:9]);
```

As you can see, the end index for the slice is the index just beyond the last element you want to include.

Slicing syntax assumes that you want to start at index 0 if the first number is not specified. You could print “Holy cats” with the following slice.

```
print(var[:9]);
```

If you omit the second number, or if it is greater than the sequence’s length, it is assumed to be equal to the sequence’s length. You could print just the word “awesome” using the following snippet. The start index is specified using a negative number based on the length of the word “awesome” at the end of the string.

```
print(var[-len('awesome'):]);
```

Or you could print the string “Python is awesome” using the following line. The start index is specified using the **index()** method, which returns the index of the first occurrence in the sequence of the item you specify.

```
print(var[var.index('Python'):]);
```

Slicing syntax also offers the option of providing a third colon-delimited number inside the square brackets. This third number represents the step count for a slice, and defaults to 1 if it is left empty. Consider the following variable.

```
nums = (1, 2, 3, 4, 5, 6, 7, 8);
```

The following line would return a tuple containing the first four numbers, (1, 2, 3, 4).

```
nums[:4:];
```

Specifying a skip value would allow you to return a tuple containing all of the odd numbers, (1, 3, 5, 7).

```
nums[::2];
```

Specifying a skip value and a start index would return a tuple with all of the even numbers, (2, 4, 6, 8).

```
nums[1::2];
```

You can also specify a negative number to reverse the sequence! For instance, the following line would return the tuple (8, 7, 6, 5, 4, 3, 2, 1).

```
nums[::-1];
```

String Types

As with numbers, there are in fact multiple string types, which Maya commands allow you to use interchangeably. Recall that a string is a sequence of numbers or characters inside of quotation marks that is treated as though it were a word. However, Python also allows you to prefix strings with special characters to create raw and Unicode strings. Each of these requires that we first introduce you to escape sequences.

Escape Sequences and Multiline Strings

Ordinarily, you must use what are known as escape sequences to include certain special characters in your strings, such as Unicode characters (**\u**), a new line (**\n**), a tab (**\t**), or the same type of quotation marks you use to define your string (**\'** or **\"**). You create an escape sequence by including a backslash (\) before a certain ordinary ASCII character to indicate that it is special. See Section 2.4.1 of Python Language Reference for more information.

Suppose, for instance, you want to create a string that includes quotation marks in it. Either of the following lines would be equivalent.

```
var = '"I\'m tired of these examples."';
var = "\"I'm tired of these examples.\"";
```

Each of these examples would print the following result:

```
"I'm tired of these examples."
```

Similarly, if you wanted to include a line break, you would use one of the following three examples. (Note that we omit semicolons at line endings here for demonstration, since we can assume this sample is short enough to print correctly.)

```
var = 'line 1\nline 2'
var = "line 1\nline 2"
var = """line 1
line 2"""
```

If you were to print `var`, each of these three examples would produce the same result.

```
line 1
line 2
```

Note that using a pair of triple quotation marks allows you to span multiple lines, where your return carriage is contained in the literal value. Recall that we mentioned in the introduction that Python also lets you use a backslash character at the end of the line to carry long statements onto a new line.

However, this pattern does not translate into a literal return carriage. Consider the following lines.

```
var = 'line 1 \
line 2'
```

Contrary to the triple-quotation-mark example, this variable would print the following result.

```
line 1 line 2
```

Raw Strings

Python supports what are called raw strings. A raw string, though still a string according to its underlying type, allows you to include backslashes without escaping them. You can include the **r** or **R** character prefix for your value to indicate that the value is a raw string, and that backslashes should not be presumed to escape the following character. The following two examples produce the same result.

```
var = 'C:\\Users\\Adam\\Desktop';
var = r'C:\Users\Adam\Desktop';
```

Raw strings can be helpful when creating directory paths or regular expressions. Regular expressions are special patterns that allow you to search string data for particular sequences. While they are invaluable, they are also complex enough that they merit entire books on their own, so we do not cover them here. Consult the companion web site for some tips on using regular expressions in Maya.

Unicode Strings

A Unicode string allows you to handle special characters, irrespective of region encoding. For example, with ordinary strings, each character is represented internally with an 8-bit number, allowing for 256 different possibilities. Unfortunately, while 256 characters may suffice for any one language, they are insufficient to represent a set of characters across multiple languages (e.g., Roman characters, Korean, Japanese, Chinese, Arabic). You can create a Unicode string by prefixing the character **u** or **U** onto the value. Their type is Unicode.

```
var = u'-5';
print(type(var));
```

You can include escape sequences to print special characters. A table of their values is publicly available at *ftp.unicode.org*. The following line uses the Unicode escape sequence **\u** to print a greeting to our German readers: "Grüß dich!"

```
print(u'Gr\u00fc\u00df dich!');
```

Unicode strings benefit from most of the same operators and functions available to ordinary strings. Section 5.6.1 of Python Standard Library lists a variety of useful built-in methods for strings. We strongly recommend browsing these built-in methods, because there are a number of incredibly useful ones that will save you loads of time when working in Maya. You can also pull up a list of string methods directly in your interpreter by calling the built-in **help()** function.

```
help(str);
```

You may have also noticed that Maya returns object names as Unicode characters when you execute commands. For example, if you create a new cube and print the result, the History Panel shows you a list of Unicode strings. You can verify this by printing the result of the `polyCube` command.

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

This example produces something like the following line of output.

```
[u'pCube1', u'polyCube1']
```

Formatting Strings

While MEL users will feel right at home simply concatenating strings, this approach can quickly become verbose and inefficient. For instance, the following example would print the string "There are 6 sides to a cube, and about 6.283185308 radians in a circle."

```
cube = 6;
pi = 3.141592654;
print(
    'There are '+str(cube)+
    ' sides to a cube, and about '+
    str(2*pi)+' radians in a circle.'
);
```

Apart from the tedium of casting your variables as strings, the value of 2π is unnecessarily verbose for human-readable output.

Fortunately, strings and Unicode strings in Python allow you to use powerful, concise formatting operations to increase efficiency, reduce verbosity, and alter the appearance of output. The basic approach is that you insert a **%** character in the string to initiate a formatting sequence, and then follow the string's value with another **%** character and a list of the arguments in order. The previous example could be rewritten in the following way.

```
cube = 6;
pi = 3.141592654;
print(
    'There are %s sides to a cube, and about %.2f radians in a
    circle'%(
        cube, 2*pi
    )
);
```

Table 2.5 Important String Formatting Patterns

Code	Meaning	Notes
d	Signed integer	Truncates decimal numbers
i	Signed integer	Truncates decimal numbers
e	Scientific notation	
f	Floating point	
g	Mixed floating point	Uses scientific notation if number is very small or very large
s	String	Same as using **str()** function with object

Using this formatting process, the result is now "There are 6 sides to a cube, and about 6.28 radians in a circle."

Section 5.6.2 of Python Standard Library provides information on more detailed options for string formatting, though we only use this basic paradigm throughout the book. The documentation also contains a list of available string formatting codes. We have listed some common patterns in Table 2.5.

The Python documentation also lists other flags that can accompany some of these codes. Though you should refer to Python's documents for more information, it is worth describing the precision modifier, as we use it occasionally.

Note that you can modify the precision that a decimal number will use by following the **%** character in the string with a period and number, before the formatting character. For example, the following lines create a variable `pi` and print it to five decimal places.

```
pi = 3.141592654;
print('%.5f'%pi);
```

More on Lists

Because lists are mutable, they have some properties not shared with other sequence types. For instance, as you have already seen, lists alone allow for individual element assignment using the index operator. However, lists have a couple of other unique properties worth briefly discussing.

del()

Just as you can alter items in a list using the index operator, you can delete items in a list using the **del()** function. For instance, the following example produces the list [0, 2, 3, 4, 5, 6, 7].

```
nums = [0,1,2,3,4,5,6,7];
del(nums[1]);
```

You also delete a slice from a list in the same way. The following example results in the list [0, 1, 4, 5, 6, 7].

```
nums = [0,1,2,3,4,5,6,7];
del(nums[2:4]);
```

Note in both cases that the original list is being modified. A new list is not being created. You can execute the following lines to confirm that the identity is intact.

```
nums = [0,1,2,3];
print('id before', id(nums));
del(nums[1:2]);
print('id after', id(nums));
```

Nested Lists

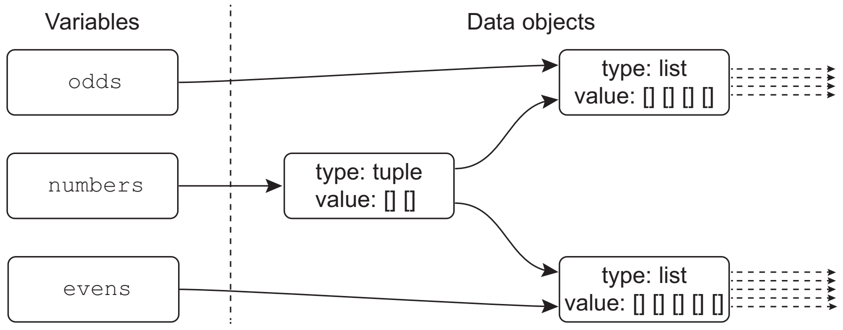
As you have seen up to this point, immutable objects, such as tuples, do not allow you to alter their values. Altering the value of a variable pointing to an object with an immutable type simply references new data altogether. However, as the index assignment and **del()** examples illustrated with lists, mutable objects can have their values modified. The consequence of this principle is that references to mutable types are retained when their values are mutated.

Consider the following example, which creates three immutable objects (numbers) and puts them inside a list.

```
a = 1;
b = 2;
c = 3;
abc = [a, b, c];
print(abc); # [1, 2, 3]
```

At this point, making further changes to the variables `a`, `b`, and `c` produces no change in the list `abc`.

```
a = 2;
b = 4;
c = 6;
print(abc); # [1, 2, 3]
```



■ **FIGURE 2.5** Variables pointing to mutable objects refer to the same data as collections containing the mutable objects.

You see no change in the list because, when you initially create the variables `a`, `b`, and `c`, they are all referencing immutable objects in memory. When you initialize the list using these variables, each element in the list is pointing to the same objects in memory. However, when you reassign the values of `a`, `b`, and `c`, they are referring to new data, while the elements in the list are still referring to the original data.

However, making changes to mutable objects, such as lists, will result in changes in other places the same objects are referenced (Figure 2.5). Consider the following example, which will print a tuple containing two lists: ([1, 3, 5, 7], [0, 2, 4, 6, 8]).

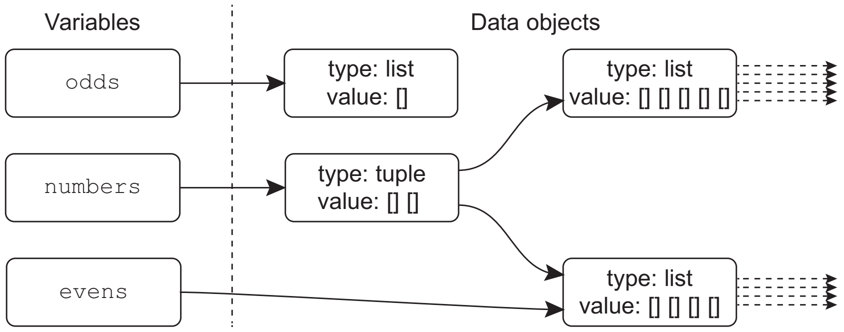
```
odds = [1, 3, 5, 7];
evens = [0, 2, 4, 6, 8];
numbers = (odds, evens);
print(numbers);
```

At this point, there are two lists, each pointing to a different, mutable location in memory. There is also a tuple, the elements of which are pointing to these same locations in memory. At this point, it is worth noting the identities of each of these variables.

```
print('odds', id(odds));
print('evens', id(evens));
print('numbers', id(numbers));
```

Now, you could make some adjustments to your lists, such as adding or removing elements. If you execute the following lines, you can see that the `numbers` tuple has new values when you modify the lists: ([1, 3, 5, 7, 9], [2, 4, 6, 8]).

```
odds.append(9);
del(evens[0]);
print (numbers);
```



■ **FIGURE 2.6** Reassigning a variable simply points it to a new object.

Because tuples are immutable, you may be wondering how exactly you have changed the values in the tuple. Remember that the tuple contains two elements, each referencing the same location in memory as the respective lists. Because the lists are mutable, their values can be altered, and the reference is retained. If you print the variables' identities now, they should be identical to what they were before altering the lists.

```
print('odds', id(odds));
print('evens', id(evens));
print('numbers', id(numbers));
```

Note that although we demonstrated this principle by nesting lists in a tuple, it also applies to lists nested in lists. *The key feature is the mutability of lists: All other mutable types will produce the same behavior.* It is important that you understand this concept, as it becomes more important when you begin working with classes and objects in Chapter 5.

Finally, note that only mutations of mutable data, not reassignments of variables pointing to the data, affect other variables' values (Figure 2.6). The reason for this behavior is that the list outside the tuple (or list) is a reference to a data object, not to another variable (which is itself a reference to some object). The following lines illustrate that reassigning the `odds` list does not affect the object referred to inside the `numbers` tuple.

```
odds = [];
print(numbers);
```

OTHER CONTAINER TYPES

In addition to the basic sequence types, Python includes a couple of important container types. Although these types are not used with Maya commands directly, you will make frequent use of them in your own programs that use Maya commands. In this section we briefly introduce two containers: sets and dictionaries.

Sets

Section 5.7 of Python Standard Library describes sets as "an unordered collection of distinct hashable objects." While sets have many applications, the most common uses include membership testing and removing duplicates from a sequence. Note that sets are a mutable type. Python also implements a comparable type, frozenset, which is immutable.

Creating a set is as easy as using the **set()** constructor with an iterable type, such as any of the sequences we have discussed. For instance, you can create a set from a string or from a list.

```
set1 = set('adam@adammechtley.com');
set2 = set([1, 2, 3, 3, 5, 6, 7, 7, 7, 8, 9]);
```

If you were to print `set2`,

```
print(set2);
```

you would see the following output:

```
set([1, 2, 3, 5, 6, 7, 8, 9]);
```

As you can see, the set appears to be a version of the list that has had its duplicate elements pruned.

Operators

While sets implement many of the same operations as the sequence types we already discussed (e.g., **in** and **len()**), the fact that they are unordered means that they cannot make use of others (e.g., indexing and slicing). Table 2.6 lists important set operators.

While they fundamentally serve different purposes, sets can test membership more efficiently than ordinary sequences. Consider the following case.

```
nums = [1, 2, 3, 3, 5, 6, 7, 7, 7, 8, 9];
```

Table 2.6 Important Set Operators

Operation	Result
x **in** set	True if *x* is in set
x **not in** set	True if *x* is not in set
len(set)	Length of set
set **|** other	New set with all elements from set and other
set **&** other	New set with only common elements from set and other
set **–** other	New set with elements in set that are absent in other
set **^** other	New set with elements in set or other, but not both

If you were writing a program that needed to test membership, you could simply use the **in** operator with the list.

```
8 in nums;
```

However, this operation effectively needs to test each of the items in the list in order. The fact that there are several duplicates in the list wastes computation time. Although this example is fairly simple, you can imagine how a very long list or a large number of membership tests might slow things down. On the other hand, you can put this list into a set.

```
numset = set(nums);
```

Testing membership on the set is much more efficient.

```
8 in numset;
```

While removing duplicates may seem like the source of the gain, the actual gain relates to the fact that the items in the set are hashable. Thinking back to our discussion of Python's data model, a hashable piece of data is one that will have a consistent identity for its lifetime—an immutable object. *While you can generate a set from a list, a list cannot be an item in a set!* The advantage to looking up items on the basis of a hash is that the cost of a lookup is independent of the size of the set.

Including an item in a set maintains an active reference to the data. Even if the object loses all of its other references, through reassignments or use of **del()**, the set maintains an active reference unless the item is specifically removed from the set using a built-in method, or the set itself is deleted.

Dictionaries

Another useful container type is the dictionary. Programmers coming from other languages will recognize the dictionary as a hash table. A dictionary is composed of a set of hashable (immutable) keys, each of which maps to an arbitrary object.

Like a set, because the keys in a dictionary are hashable, each key can only appear once, and therefore may only have one value associated with it. Moreover, the dictionary itself is a mutable object. Consequently, a dictionary cannot be a key in another dictionary, but it may be a value associated with a particular key in another dictionary.

Creating a dictionary is very simple, and is somewhat similar to lists and tuples. To create a dictionary, enclose a set of comma-delimited, colon-separated key-value pairs inside of curly braces (**{** and **}**). For example, the following assignment creates a dictionary with two keys—the strings “radius” and “height”—which have values of 2 and 5 associated with them, respectively.

```
cylinderAttributes = {'radius':2, 'height':5};
```

Remember that the keys may be any immutable type, and that the values may be anything. You could create a dictionary that maps numbers to their names (in German, no less).

```
numberNames = {1:'eins', 2:'zwei', 3:'drei'};
```

The keys also do not all need to be the same type. The following dictionary will map numbers to names, as well as names to numbers.

```
numberNames = {
    1:'one',2:'two',3:'three',
    'one':1,'two':2,'three':3
};
```

Operators

Section 5.8 of Python Standard Library includes operators and methods for dictionaries. We have included some of the most important ones in Table 2.7. While the first three operations should be pretty clear by this point, two of the remaining items in our table merit some discussion.

Because dictionaries are a table of hashable objects, the items in a dictionary have no order, much like a set. Consequently, there is no concept of slicing in a dictionary. However, unlike sets, dictionaries implement the square bracket operator. Instead of supplying a dictionary with an index, however, you supply it with a hashable object. For instance, the following snippet would print the name “two” associated with the number 2.

```
numberNames = {1:'one', 2:'two', 3:'three'};
print(numberNames[2]);
```

Table 2.7 Important Dictionary Operators

Operation	Result	Notes
key **in** *d*	True if key is in *d*	
key **not in** *d*	True if key is not in *d*	
len(*d*)	Length of dictionary	
d[key]	Value (object) corresponding to key	
d.**keys()**	List of all keys in the dictionary	
d.**setdefault**(key)	Value corresponding to key if it exists, otherwise None	Optional default value

Because dictionaries are mutable, you can use this operator not only to get an object but also to set it. The following lines assign Polish names to our number keys.

```
numberNames[1] = 'jeden';
numberNames[2] = 'dwa';
numberNames[3] = 'trzy';
```

In this respect, you can almost think of dictionaries as a table of mailing addresses: The occupant at each particular address can change, but the address will still be present. What happens then if you try to access an address that does not exist?

Conveniently, any time you set the value for a key that does not yet exist, it is simply added to the dictionary. You could execute the following line to add another item to your dictionary.

```
numberNames[4] = 'cztery';
```

However, if you try to access a key that does not yet exist, you will get a **KeyError**, as in the following line.

```
print(numberNames[5]);
```

Thus, querying a key that does not yet exist would throw an exception and stop execution of your program. Thankfully, dictionaries implement a **setdefault()** method, which, if a key exists, will return its value. If a key does not exist, the method will add the key and assign a value of None to it. Optionally, you can supply a parameter for this method that, if the key does not exist, will become its default value. The following line will add a new entry with the key 5 and value "pięć" that will be printed.

```
print(numberNames.setdefault(5, u'pi\u0119\u0107'));
```

Bardzo dobrze!

Dictionaries in Practice

Dictionaries, though perhaps initially confusing for newcomers, are powerful objects in practice, especially when working with Maya commands. To take one example, Python developers can use dictionaries to map rotation orders.

In Maya, some commands return or specify rotation order as a string, such as "xyz" or "zxy". The following example demonstrates how `xform` is one such command, and will print "xyz" when querying a locator's rotation order.

```
import maya.cmds;
loc = maya.cmds.spaceLocator()[0];
print(maya.cmds.xform(loc, q=True, rotateOrder=True));
```

Internally to Maya, however, rotation order is a special kind of type, called an enumerated type. While we will discuss enumerated types in greater detail when exploring the API, the basic idea is that each different rotation order sequence corresponds to an integer. Consequently, using the `getAttr` and `setAttr` commands with rotation order requires use of an integer in Python. The following example demonstrates this issue, and will print "0" when using `getAttr` to get a rotation order.

```
import maya.cmds;
loc = maya.cmds.spaceLocator()[0];
print(maya.cmds.getAttr('%s.rotateOrder'%loc));
```

As you can imagine, intermixing such commands can be an annoyance.

Fortunately, you can easily create dictionaries to help with this problem. One dictionary can map strings to integers, and the other can map integers to strings.[4]

```
roStrToInt = {
    'xyz':0,'yzx':1,'zxy':2,
    'xzy':3,'yxz':4,'zyx':5
};
roIntToStr = {
    0:'xyz',1:'yzx',2:'zxy',
    3:'xzy',4:'yxz',5:'zyx'
};
```

Armed with these two dictionaries, you could do something like the following example to set the rotation order of a cube and a joint to `zxy`.

```
import maya.cmds;
cube = maya.cmds.polyCube()[0];
maya.cmds.setAttr(
    '%s.rotateOrder'%cube,
    roStrToInt['zxy']
);
rotateOrder = maya.cmds.getAttr('%s.rotateOrder'%cube);
joint = maya.cmds.joint(
    rotationOrder=roIntToStr[rotateOrder]
);
```

[4]For a dictionary where all the values are known to be unique, a concise technique to automatically generate an inverse mapping, which we avoid here for the sake of clarity and also progress in the text, would be `inverseMap = dict((val, key) for key, val in originalMap.iteritems())`.

CONCLUDING REMARKS

Any useful programming task requires the use of variables. You have seen in this chapter how variables in Python differ from those in MEL and C++, and you have explored much of their underlying mechanics. You are now able to work effortlessly with many of Python's principal built-in objects, and also have a strong foundation for understanding a range of complex topics that we will explore in this text. The stage is now set to start developing programs with functions, conditional statements, loops, and more.