Chapter 3

Writing Python Programs in Maya

CHAPTER OUTLINE

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

- Describe what functions are.
- Create Python functions using the **def** keyword.
- Describe and leverage Python's different function argument types.
- Use **return** statements to pass information between functions.

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00003-X

- Execute some common Maya commands.
- Exploit the **for** statement to create loops.
- Use the **range()** function to emulate **for** loops in MEL.
- Leverage conditional statements for branching and flow control.
- Mimic MEL ternary and **switch** statements using Python analogs.
- Create a loop using the **while** statement.
- Handle exceptions and errors in your code.
- Build the framework for a basic texture processing tool.

So far we have introduced you to some basic Maya commands and the essentials for understanding data and variables in Python. Nevertheless, you still need some more fundamentals to start creating Python programs.

In this chapter, we walk through a simple texture-processing framework for Maya as we introduce functions, the building blocks for complex programs. We begin by learning how to declare a function in Python and return data from it. We then explore different methods for controlling code execution using Python's sequence types and looping statements. Next, we demonstrate how to further control execution using Python's conditional statements and branching capabilities. We conclude our exploration of functions by learning techniques for handling exit cases, including error and exception handling and returns. Finally, we incorporate all the major topics of this chapter into a set of functions that work with Maya commands to process textures.

CREATING PYTHON FUNCTIONS

The term *function* carries specific connotations in different programming circles. *For our purposes, a function is a grouping of statements, expressions, and commands that can be accessed using a developer-defined name and that can optionally act on developer-supplied input parameters*. Functions can be thought of as the building blocks of large programs in that they can be reused in different contexts to act on different data sets. A Python function is analogous to MEL's procedures.

Anatomy of a Function Definition

Similar to how a MEL procedure is defined with the keyword **proc**, a Python function is declared using the **def** keyword, followed by a name, a set of parentheses that encloses definitions for optional input arguments, and a colon.

```
def function_name(optional, input, parameters):
    pass;
# This line is no longer part of the function
# optional, input, and parameters have no meaning here
```

As we noted in this book's introduction, Python is very particular about whitespace. The lines following the colon, inside the function block, must be indented. All lines at or below the indentation level following the **def** keyword are taken to be part of the function's body of executable statements, up until a line at the same indentation level as the **def** keyword.

In our hypothetical example, we included the **pass** keyword inside of the function's body to indicate where code would be. Normally, you would put a set of executable statements inside the function. Python offers the **pass** keyword as a placeholder where something is syntactically required (executable statements in this case). We could substitute in a line to print a message in our own function.

1. Execute the following lines in the Script Editor to define **process_all_textures()**. This function prints a simple string. (Note that we will be altering this function throughout this chapter, so it is advisable that you highlight it and execute it using **Ctrl + Enter** so it is not cleared from the Script Editor's Input Panel.)

   ```
   def process_all_textures():
       print('Process all textures');
   ```

Since the **print()** line is encapsulated in the function definition, executing this code in the Script Editor does nothing—or at least nothing immediately obvious.

Recall the definition we offered for the term *function*. The key phrase in our definition is that a function is "accessed by a developer-defined name." When you execute a function *definition*, the definition is registered with the Python interpreter, much like a name for a variable. Python is then aware of the function, the statements contained within it, and the name by which those statements should be called.

2. Execute the following line in the Script Editor to display the type associated with the name of the function you just defined.

   ```
   type(process_all_textures);
   ```

   As you can see, the name `process_all_textures` is like any other name you define, but is associated with an object of type function.

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

3. Likewise, you can define another name that points to the same object, just like you can with variables. Execute the following lines in the Script Editor to bind another name to your function, and then compare the identities associated with both names.

```
processAllTextures = process_all_textures;
print('Ref 1: %s'%id(process_all_textures));
print('Ref 2: %s'%id(processAllTextures));
```

   Your output should show that the identities for both references are the same.

Executing a function is referred to as *calling* a function. To instruct Python to call a function, the function name needs to be supplied to the interpreter, followed by parentheses containing values to pass to the function.

4. Execute the following lines in the Script Editor to call your function using both of the names you bound to it.

```
process_all_textures();
processAllTextures();
```

   While simply using the function's name returns a reference to it (as you saw in steps 2 and 3), entering a set of parentheses after the name instructs Python to execute its statements. Consequently, you should see the following lines of output.

```
Process all textures
Process all textures
```

While functions allow you to execute any number of statements inside their bodies, including Maya commands, their real value comes from including arguments.

Function Arguments

Recall the template we offered for function definitions in the previous section.

```
def function_name(optional, input, parameters):
    pass;
```

The names `optional`, `input`, and `parameters` enclosed in the parentheses are called *arguments*. Arguments are used to further customize a function by allowing the developer to provide data to the function from the calling context. While a function does not require arguments in its definition, arguments are one of the first steps required to make functions reusable. They are the primary mechanism for customizing the behavior of a function.

One of the shortcomings of the current implementation of **process_all_textures()** is the fact that every time this function is called from somewhere else, otherwise known as the calling context, it will produce the same result every time. Optimally, the function should be able to be called in different contexts and act upon specified textures.

As you saw in our template, the simplest type of argument is a named parameter, declared along with the function's name as part of the **def** statement. The function can then access this parameter by name in code and exists for the duration of the function's execution. In this case, we say that the scope of the parameter name is inside of the function. As we hinted a moment ago, scope in Python is determined by indent levels, which are analogous to curly braces in MEL.

```
def a_function():
    inside_func_scope = 0;
outside_func_scope = 1;
"""
The following line would produce a NameError since
inside_func_scope does not exist outside of a_function()
"""
outside_func_scope = inside_func_scope;
```

We can add an argument to the scope of **process_all_textures()** by changing the function definition.

5. Execute the following lines to alter the function definition for **process_all_textures()**.

   ```
   def process_all_textures(texture_node):
       print('Processed %s'%texture_node);
   ```

   At this point, you could now pass an argument to this function to change its behavior. Whatever argument is passed is bound to the name `texture_node` when executing statements inside the function's body.

6. Execute the following lines of code to create a new **file** node and pass it to the **process_all_textures()** function.

   ```
   import maya.cmds;
   texture = maya.cmds.shadingNode('file', asTexture=True);
   process_all_textures(texture);
   ```

   As you can see, the function has printed the name of the new node that you passed to it, instead of a static statement.

   ```
   Processed file1
   ```

   The important point is that Python maps the incoming reference to the internal name for the argument. Although the node name is assigned to

a variable called `texture` in the calling context, this variable maps to `texture_node` once execution enters the function body. However, the function now requires that an argument be passed.

7. Try to call the **process_all_textures()** function without specifying any arguments.

```
process_all_textures();
```

You can see from the output that Python has thrown an error and told you that you have specified the wrong number of arguments.

```
# Error: TypeError: file <maya console> line 1:
process_all_textures() takes exactly 1 argument
(0 given) #
```

The new declaration of **process_all_textures()** dictates that the function must now be called with an argument.

8. Note also that functions can be made to support multiple arguments by adding a comma-delimited list of names to the function definition. Execute the following lines to redefine the **process_all_textures()** function, requiring that it take two arguments.

```
def process_all_textures(texture_node, prefix):
    print('Processed %s%s'%(prefix, texture_node));
```

9. Execute the following lines to call **process_all_textures()** and pass its two arguments.

```
process_all_textures(texture, 'my_');
```

You should now see the following output.

```
Processed my_file1
```

As before, arguments defined in this manner are required to be passed with the function call. Otherwise, Python returns an error indicating the number of arguments required and the number found. However, Python provides many different approaches for declaring and passing arguments that help relax these restrictions somewhat. Arguments can be default arguments, keyword arguments, and variable-length argument lists.

Default Arguments

Default arguments are values defined along with the function's arguments, which are passed to the function if it is called without an explicit value for the argument in question.

10. Execute the following lines to assign a default value to the `prefix` argument in the **process_all_textures()** function definition.

```
def process_all_textures(texture_node, prefix='my_'):
    print('Processed %s%s'%(prefix, texture_node));
```

By assigning the value "my_" to the `prefix` argument in the function's declaration, you are ensuring that the prefix input parameter will always have a value, even if one is not supplied when the function is called.

11. Execute the following line to call the function again with its changes.

```
process_all_textures(texture);
```

This time, the output shows you that the `prefix` name used the default value "my_" inside the function body, even though you specified no prefix.

```
Processed my_file1
```

You may have noticed that it is not required that every argument carry a default value. As long as the function is called with a value for every argument that has no default value assigned, and as long as the function body can properly handle the type of incoming arguments, the function should execute properly.

As an aside, it is worth briefly noting that a default argument's value can also be None, which allows it to be called with no arguments without raising an error.

```
def a_function(arg1=None):
    pass;
a_function(); # this invocation works
```

Returning to **process_all_textures()**, it is worth mentioning that our definition now contains two different types of arguments. The first argument, `texture_node`, is called a positional argument. On the other hand, the definition you created in step 8 contained two positional arguments.

Positional input parameters are evaluated based on the order in which they are passed to the function and are absolutely required by the function to run. In the preceding definition, the first parameter passed into the function will be mapped to the `texure_nodes` name, while the second parameter will be mapped to `prefix`. As you can imagine, you could run into problems if you accidentally passed your arguments out of order. Fortunately, an input parameter can be mapped to a specific position if it is passed as a keyword argument.

Keyword Arguments

Keyword arguments are a form of input parameter that can be used when calling a function to specify the variable in the function to which the supplied data are bound.

12. Execute the following lines to redeclare the **process_all_textures()** function.

```
def process_all_textures(
    texture_node=None, prefix='my_'
):
    print('Processed %s%s'%(prefix, texture_node));
```

If the function were declared in this manner, any of the following calls would be *syntactically* valid, though the first one wouldn't produce an especially useful result.

```
# no arguments
process_all_textures();
# single positional argument
process_all_textures(texture);
# multiple positional arguments
process_all_textures(texture, 'grass_');
```

All of these examples rely on default values and positional arguments to pass data. Passing a keyword argument, on the other hand, allows you to specify a parameter in the form keyword=value. Passing an argument by keyword allows the caller to specify the name to which the argument will be bound in the function body.

13. Recall that when passing positional arguments, each argument is evaluated in the order it is passed. Execute the following call to **process_all_textures()**.

```
process_all_textures('grass_', texture);
```

Because the string intended to be used as `prefix` was passed as the argument in the first position (position 0) the function produces the wrong output.

```
Processed file1grass_
```

14. Passing each argument by keyword alleviates this problem. Execute the following line to pass keyword arguments (a syntax that should look familiar).

```
process_all_textures(
    prefix='grass_',
    texture_node=texture
);
```

While these techniques of argument passing do provide a fair bit of flexibility, there are two issues to be aware of, both of which are closely related. *The first issue is that positional arguments must come before keyword arguments, both in the function declaration and when calling the function. The second issue is that an argument can be passed by position or by keyword, but not both.*

In the current definition of **process_all_textures()**, `texture_node` is the argument at position 0 and `prefix` is the argument at position 1. Calling **process_all_textures()** with the following arguments would produce an error.

```
process_all_textures('grass_', texture_node=texture);
```

Because a positional argument is passed to **process_all_textures()** in position 0 in this situation, the variable at position 0 (`texture_node` in this case) is already bound by the time that Python attempts to map `texture` to it as a keyword argument.

Variable-Length Argument Lists with the * Operator

In addition to the cases we have examined so far, there are also situations in which you may need to implement functions that can take an unspecified number of variables. To address this situation, Python provides variable-length argument lists. Variable-length argument lists allow a developer to define a function with an arbitrary number of arguments.

15. Execute the following code to modify the **process_all_textures()** definition to support a variable-length argument list.

```
def process_all_textures(*args):
    print(args[0], args[1:]);
```

16. The function can now be called with a variety of different argument patterns. Execute the following lines to create new **file** nodes (textures) and then use different invocations for the function.

```
tx1 = maya.cmds.shadingNode('file', asTexture=True);
tx2 = maya.cmds.shadingNode('file', asTexture=True);
tx3 = maya.cmds.shadingNode('file', asTexture=True);
tx4 = maya.cmds.shadingNode('file', asTexture=True);
process_all_textures('grass_');
process_all_textures('grass_', tx1);
process_all_textures('grass_', tx2, tx3, tx4);
```

You should see the following output lines, indicating that each call was successful.

```
('grass_', ())
('grass_', (u'file2',))
('grass_', (u'file3', u'file4', u'file5'))
```

Although this particular syntax isn't terribly useful for our current case, the main issue to be aware of in this example is the use of the asterisk (*) operator in the function declaration.

```
def process_all_textures(*args):
```

This operator, when prefixed in front of an argument, instructs the Python interpreter to pack all of the arguments passed to the function into a tuple and to pass that tuple to the function, as opposed to passing each argument individually. Any name can be used to declare a variable-length argument list in a function declaration as long as the asterisk (*****) operator precedes the name, though convention is to use the name `args`.

In this implementation, the function assumes that the data at position 0 correspond to a prefix, and that all further arguments in the slice from 1 onward are texture names. Consequently, the current implementation requires at least one positional argument for the function to execute.

Recall that we passed each of the **file** nodes as individual arguments in step 16. The asterisk operator can also be used in the function call to pass the arguments differently. When used in this manner, the asterisk instructs the Python interpreter to unpack a sequence type and use the result as positional arguments.

17. Execute the following lines to pack the **file** nodes you created in step 16 into a list and pass the list to the function.

```
node_list = [tx1, tx2, tx3, tx4];
process_all_textures('grass_', node_list);
```

As you can see in the output, the list itself is contained in a tuple.

```
('grass_', ([u'file2', u'file3', u'file4', u'file5'],))
```

18. Passing `node_list` using the asterisk operator fixes this problem. Execute the following line to call your function and have it unpack the list of **file** node names.

```
process_all_textures('grass_', *node_list);
```

You should see output like that in step 16, confirming that the operation was successful.

```
('grass_', (u'file2', u'file3', u'file4', u'file5'))
```

*Variable-Length Keyword Argument Lists with the ** Operator*

Python also allows for variable-length keyword argument lists. The syntax is almost identical to a positional variable-length list. Variable-length keyword argument lists are declared using the double asterisk (**) operator. The double asterisk operator tells the interpreters to pack all key-value pairs passed to the function into a dictionary.

19. Execute the following changes to the **process_all_textures()** definition to add a variable-length keyword argument, `kwargs`.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix', 'my_');
    texture = kwargs.setdefault('texture_node');
    print('%s%s'%(pre, texture));
```

Recall from Chapter 2 that the **setdefault()** function is a special type of function associated with dictionary objects. This function searches the associated dictionary for the key specified in the first positional argument, returning a value of None if a second positional argument is not specified. In this case, the `pre` variable is initialized to "my_" if the `prefix` keyword is not set, while the `texture` variable is initialized to None if the `texture_node` keyword is not specified.

20. Execute the following lines to test different syntax patterns for the **process_all_textures()** function.

```
# calling with no keywords does nothing
process_all_textures();
# calling with an unhandled keyword produces no error
process_all_textures(file_name='default.jpg')
# specifying the 'texture_node' key will process file1
process_all_textures(texture_node=texture);
```

You should see the following output, confirming that each invocation worked as expected. The first two completions are from executions that did nothing, while the third prints a new name for `texture`. Keywords that you do not explicitly handle in your function are simply ignored.

```
my_None
my_None
my_file1
```

21. As with the single asterisk operator, the double asterisk can also be used to expand a dictionary and pass it to a function as a list of keyword arguments. Execute the following code to create an argument dictionary to pass to the function.

```
arg_dict = {
    'prefix':'grass_',
    'texture_node':tx1
};
process_all_textures(**arg_dict);
```

In addition to offering you a variety of options for how to construct your functions, pass arguments, and handle incoming parameters, Python's flexible syntax for working with functions also offers you many opportunities

to make your code much more readable, rather than relying on cryptic variable names being passed in an arbitrary order.

Return Values

While passing arguments to functions is a basic requirement for doing useful work, so, too, is returning data. Mutable objects, such as lists, can simply be passed as arguments to a function, and any mutations you happen to invoke inside your function are reflected in the calling context when the function has concluded. For instance, the following example creates a new empty list, mutates it inside a function, and then prints the list in the calling context to indicate that its mutations have affected the object in the calling context.

```
def mutate_list(list_arg):
    list_arg.append(1);
    list_arg.append(2);
    list_arg.append(3);
a_list = [];
mutate_list(a_list);
print(a_list);
# prints: [1, 2, 3]
```

On the other hand, immutable objects are not affected in the same way, as their values cannot be changed. Incrementing a number's value inside of a function, for example, has no result on the variable passed to the function in the calling context.

```
def increment_num(num):
    num += 1;
n = 1;
increment_num(n);
print(n);
# prints: 1
```

Moreover, in some cases, you actually want a function to perform some operation and yield a result that you wish to store in a new variable altogether. In such cases, you will want to use a **return** statement to send the function's results back to the calling context. Implementing the **return** statement in a function causes it to immediately and silently exit, sending the value specified to the right of the keyword back to the calling context.

22. Execute the following lines to modify the **process_all_textures()** function. It now renames the supplied **file** node using the `rename` command and returns the result of the operation to the calling context.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix', 'my_');
```

```
    texture = kwargs.setdefault('texture_node');
    return maya.cmds.rename(
        texture,
        '%s%s'%(pre, texture)
    );
```

23. Execute the following lines to rename the **file** node specified by `texture` using the **process_all_textures()** function and store the new value in the `texture` variable.

```
texture = process_all_textures(texture_node=texture);
print(texture);
# prints: my_file1
```

As you can see, the new version of the function in fact returns the value returned by the `rename` command, which is the new name of the object. Return values are an essential part of designing useful functions, particularly in cases where you are working with immutable objects. We'll keep working on this function throughout this chapter to introduce some further language features, but it is worth briefly returning to Maya commands for a moment before proceeding.

MAYA COMMANDS

At this point, you have probably noticed that functions resemble the Maya commands you have seen so far. In fact, Maya's commands are all functions! (You may have already used the built-in **type()** function with some commands' names and discovered this fact.)

Recall that we noted in Chapter 1 that MEL syntax requires command arguments to follow flag arguments, while Python requires the opposite order. At this point, it is hopefully clear why Python requires its syntax for working with commands: In Python, command arguments are simply a variable-length argument list, while flag arguments are variable-length keyword arguments. For instance, in the following example, we create a sphere and then execute the `polySphere` command in edit mode, passing it first the name of the object and then a series of keyword arguments to set values for the command flags.

```
import maya.cmds;
sphere = maya.cmds.polySphere();
maya.cmds.polySphere(
    sphere[1], edit=True,
    radius=5, sh=16, sa=16
);
```

Remember that we pointed out that commands like `polySphere` return a list containing a **transform** node and another node (a **polySphere** node in this case).

```
print(maya.cmds.polySphere());
# prints: [u'pSphere2', u'polySphere2']
```

In addition to capturing this return value in a list, you can use special Python syntax to unpack each individual list item in a separate variable, just as you can for ordinary Python functions that return sequences.

```
cube_xform, cube_shape = maya.cmds.polyCube();
print(cube_xform); # prints: pCube1
print(cube_shape); # prints: polyCube1
```

Although we have introduced a number of Maya commands already, such as those for creating basic objects and working with attributes, it is worth briefly pointing out a couple more common ones that we will use throughout this book. Developers accustomed to MEL will already recognize these commands and should be fairly comfortable with their syntax.

Listing and Selecting Nodes

The basic command for retrieving a list of nodes in the scene is the `ls` command.

1. Open a new Maya scene and execute the following lines to store a list of all the nodes in the scene and print it. You should see a number of default scene nodes, including time1, sequenceManager1, renderPartition, and so on.

   ```
   import maya.cmds;
   nodes = maya.cmds.ls();
   print(nodes);
   ```

2. The `ls` command also allows you to pass a string specifying the type of objects you would like in your list. Execute the following lines to store all of the **transform** nodes in the scene in the `nodes` variable and print the list.

   ```
   nodes = maya.cmds.ls(type='transform');
   print(nodes);
   ```

   You should see a list of camera **transform** nodes.

   ```
   [u'front', u'persp', u'side', u'top']
   ```

3. You can also pass strings as positional arguments containing an optional wildcard character (*) to store all objects of which the names match the pattern (as well as any other additional filters you specify using flag arguments). Execute the following lines to store and print the list of nodes of which the names begin with “persp”.

```
nodes = maya.cmds.ls('persp*');
print(nodes);
```

You should see a list containing the **transform** and **shape** nodes for the perspective camera.

```
[u'persp', u'perspShape']
```

4. Another handy, basic command is the `select` command, which allows you to populate the current global selection list. Like the `ls` command, the `select` command allows you to specify a wildcard character in its arguments. Execute the following lines to select the **transform** and **shape** nodes for the top and side cameras.

```
maya.cmds.select('side*', 'top*');
```

5. You can also use the `sl`/`selection` flag with the `ls` command to list the currently selected items. Execute the following line to print the current selection.

```
print(maya.cmds.ls(selection=True));
```

You should see that you have the **transform** and **shape** nodes for the top and side cameras currently selected.

```
[u'side', u'sideShape', u'top', u'topShape']
```

6. Both the `ls` and `select` commands allow you to pass either a list or any number of arguments. Execute the following line to create a list of items to select, and then select it and print the selection. You should see the same items in the `selection_list` variable in the output.

```
selection_list = ['front', 'persp', 'side', 'top'];
maya.cmds.select(selection_list);
print(maya.cmds.ls(sl=True));
```

7. You can use the `select` command in conjunction with the `ls` command to select all objects of a certain type. Execute the following lines to select all of the **shape** nodes in the scene and print the current selection.

```
maya.cmds.select(maya.cmds.ls(type='shape'));
print(maya.cmds.ls(sl=True));
```

You should see the following output, indicating you have selected all of the camera **shape** nodes.

```
[u'frontShape', u'perspShape', u'sideShape',
u'topShape']
```

The `file` Command

The `file` command provides a basic interface for working with Maya scene files.[1] It uses a number of flags to alter its behavior as needed.

1. Execute the following lines to open a new Maya scene.

```
import maya.cmds;
maya.cmds.file(new=True, force=True);
```

   The `force` flag allows you to bypass the dialog box that would ordinarily appear, asking if you wanted to save changes.

2. Execute the following lines to create a new cube in the scene and save it to your home folder (Documents folder in Windows) as cube.ma.

```
import os;
maya.cmds.polyCube();
maya.cmds.file(
    rename=os.path.join(
        os.getenv('HOME'),
        'cube.ma'
    )
);
maya.cmds.file(save=True);
```

   As you can see, the `rename` flag allows you to set the name for the file (as if performing a Save As operation), while the `save` flag allows you to save the file to the specified location. Note that the `rename` flag must be invoked by itself.

3. Execute the following line to open a new scene.

```
maya.cmds.file(new=True, force=True);
```

4. At this point, you can specify a command argument giving the location of a file you wish to open in conjunction with the `open` flag. Execute the following line to reopen the cube.ma file you saved in step 2.

```
maya.cmds.file(
    os.path.join(
        os.getenv('HOME'),
        'cube.ma'
    ),
    open=True,
    force=True
);
```

[1] Note that Python itself provides other tools for manipulation of arbitrary files. See Chapter 7 for an example.

Adding Attributes

Although getting, setting, connecting, and disconnecting attributes are all useful tasks, it is also possible to add custom attributes to objects in Maya. Such attributes are then compatible with all of the attribute manipulation commands we discussed in Chapter 2.

1. Execute the following lines to open a new scene, create a sphere named "Earth", and add a mass attribute to its **transform** node.

```
import maya.cmds;
maya.cmds.file(new=True, f=True);
sphere_xform, sphere_shape = maya.cmds.polySphere(
    n='Earth'
);
maya.cmds.addAttr(
    sphere_xform,
    attributeType='float',
    shortName='mass',
    longName='mass',
    defaultValue = 5.9742e24
);
```

2. You can now get and set this attribute like any other. Execute the following line to print the mass of Earth.

```
print(maya.cmds.getAttr('%s.mass'%sphere_xform));
```

   Note that some types of attributes do not use the `attributeType` flag, but instead use the `dataType` flag. Execute the following lines to add another attribute to Earth to store an alternate name on it.

```
maya.cmds.addAttr(
    sphere_xform,
    dataType='string',
    shortName='alt',
    longName='alternateName'
);
```

3. When setting a string attribute like this one, you must specify a `type` flag when invoking the `setAttr` command. There are some other types bound to the same requirements, and they are documented in Maya's Python Command Reference. Execute the following lines to assign a value to the new **alternateName** attribute.

```
maya.cmds.setAttr(
    '%s.alternateName'%sphere_xform,
    'Terra',
    type='string'
);
```

4. Note that you can specify and then subsequently access both long and short names for custom attributes. Execute the following line to print the alternate name for Earth using the short name for its custom attribute.

```
print(maya.cmds.getAttr('%s.alt'%sphere_xform));
# prints: Terra
```

ITERATION AND BRANCHING

Now that you understand the basics of functions and have familiarity with a range of common Maya commands, we can return to our texture-processing function to improve it. Let's look at it in its current state.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix', 'my_');
    texture = kwargs.setdefault('texture_node');
    return maya.cmds.rename(
        texture,
        '%s%s'%(pre, texture)
    );
```

In its present form, the function isn't especially remarkable—it can rename a single **file** node and return the new name. What happens if we fail to pass it a `texture_node` argument, or if we want to process multiple textures? Fortunately, you can use features such as iteration and branching to add sophisticated behavior to custom functions.

The for Statement

Any developer with experience writing Maya tools in MEL is undoubtedly familiar with a common development requirement: Iterate over a set of scene objects and process each object based on a condition or set of conditions. *Iteration is a technique that allows you to execute a set of statements repeatedly*.

Much like MEL, Python implements the **for** statement, which allows developers to loop through a collection of items, such as a sequence type. The **for** statement takes the following basic form.

```
for item in collection:
    perform_task(item);
```

This block of code instructs the Python interpreter to take the following steps:

- Iterate over every element in the `collection` object.
- Temporarily bind the element in the current iteration of `collection` to the name `item`.

- Pass the `item` name into the **for** loop.
- Execute statements in the **for** loop block.

The **in** keyword is a special operator that applies only to collection types. Recall that we mentioned in Chapter 2 that this operator returns True if the given item is found in a sequence. For example, the following snippet would print True.

```
print(5 in [1, 2, 5]);
```

When creating a **for** loop, the **in** operator has a slightly different meaning. It is the mechanism for binding the current item in the iteration of the collection (on the right of the operator) to the name for the item in the current step (on the left of the operator). Consider the following example.

```
sequence = [1, 2, 3];
for item in sequence:
    print(item);
```

This loop executes the **print()** function once for each item in the `sequence` list. The loop first prints 1, then 2, then 3.

Python's syntax differs significantly from the MEL **for** statement, which uses the following basic form.

```
for ($lower_bound; halting_condition; $step)
{
    perform_task();
}
```

For example, the following **for** loop in MEL would print numbers 1, 2, and 3.

```
for ($i=1; $i<=3; $i++)
{
    print($i+"\n");
}
```

range()

While Python's basic **for** statement is sufficient for most (in fact, almost all) cases, the MEL **for** syntax can be emulated in Python using the **range()** function. This simple and useful function can be used to generate a numeric progression in a list. It allows you to iterate over a list of numbers without having to define a literal list for iteration. Consider the following examples.

```
# following line prints [0, 1, 2, 3, 4]
print(range(5));
# following line prints [2, 3, 4, 5]
print(range(2,6));
# following line prints [4, 6, 8, 10, 12, 14, 16, 18]
```

```
print(range(4,20,2));
# following line prints [0, 1, 2]
print(range(len(['sphere','cube','plane'])));
```

As you can see, the **range()** function allows you to pass up to three arguments (much like the sequence slicing syntax we investigated in Chapter 2). It can take the following forms.

```
# following call returns numeric progression
# from 0 to upper_limit-1
range(upper_limit);
# following call returns numeric progression
# from lower_bound to upper_limit-1
range(lower_bound, upper_limit);
# following call returns numeric progression
# from lower_bound to upper_limit-1, using step value
range(lower_bound, upper_limit, step);
```

Because the **range()** function returns a list, you can also apply a slice to reverse it if you like.

```
# following line prints [4, 3, 2, 1, 0]
print(range(0,5)[::-1]);
```

The **range()** function can be used to generate an index for every element in a given sequence type, and thereby emulate a MEL-style **for** loop, as in the following examples.

```
"""
MEL equivalent:
string $a_list[] = {"spam","eggs","sausage","spam"};
for ($i=2; $i<size($a_list); $i++)
    print($a_list[$i]+"\n");
"""
a_list = ['spam', 'eggs', 'sausage', 'spam'];
for i in range(2, len(a_list)):
    print(a_list[i]);
"""
MEL equivalent:
int $nums[] = {1,2,3,4,5};
for ($i=0;$i<size($nums);$i+=2)
    print($nums[$i]+"\n");
"""
nums = [1,2,3,4,5];
for i in range(0,len(nums),2):
    print(nums[i]);
```

In addition to using the **range()** function to generate a numeric progression with an upper bound, you can also emulate a **for** loop with a halting condition

by slicing the iterated sequence up to the desired upper bound, as in the following example.

```
"""
MEL equivalent:
int $nums[] = {1,2,3,4,5,6,7};
for ($i=0; $i<5; $i++)
    print($nums[$i]+"\n");
"""
nums = [1,2,3,4,5,6,7];
for i in nums[:5]:
    print(i);
```

Let's modify our **process_all_textures()** function to be a bit more useful.

1. Execute the following lines to redefine the **process_all_textures()** function. As you can see, it now searches for a `texture_nodes` keyword. It will iterate through all of the items in the collection specified with this keyword, rename them all, and append them to a `new_texture_names` list to return.

```
import maya.cmds;
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix', 'my_');
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    for texture in textures:
        new_texture_names.append(
            maya.cmds.rename(
                texture,
                '%s%s'%(pre, texture)
            )
        );
    return new_texture_names;
```

2. Execute the following lines to create a new Maya scene with a list of three **file** nodes and print the list to see their names (which will be file1, file2, and file3).

```
maya.cmds.file(new=True, f=True);
textures = [];
for i in range(3):
    textures.append(
        maya.cmds.shadingNode(
            'file',
            asTexture=True
        )
    );
print(textures);
```

3. Execute the following lines to pass the new list of textures to the **process_all_textures()** function and print the result (which should be dirt_file1, dirt_file2, and dirt_file3).

```
new_textures = process_all_textures(
    texture_nodes=textures,
    prefix='dirt_'
);
print(new_textures);
```

As you can see, using a **for** statement is a powerful and concise way to repeat a set of operations. Because of Python's unique language features, including slicing and the **range()** function, you have a number of different options available when creating a **for** loop.

Branching

While you have made some important improvements to the **process_all_textures()** function, it still has some limitations. For example, passing no `texture_nodes` argument will simply cause an error.

4. Execute the followings lines to try to execute the **process_all_textures()** function without specifying the `texture_nodes` argument.

```
process_all_textures(
    prefix='mud_',
);
```

You should see the following error message print to the console.

```
# Error: TypeError: file <maya console> line 6: 'NoneType'
object is not iterable #
```

The problem is that our current implementation assumes that the argument passed with the `texture_nodes` keyword is a list or tuple containing object names. If the `texture_nodes` argument is not specified, then it initializes to a value of None.

5. Execute the following lines to make modifications to the **process_all_textures()** function.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix', 'my_');
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        for texture in textures:
            new_texture_names.append(
```

```
                maya.cmds.rename(
                    texture,
                    '%s%s'%(pre, texture)
                )
            );
        return new_texture_names;
    else:
        maya.cmds.error('No texture nodes specified');
```

6. Now attempt to execute the **process_all_textures()** function without supplying the `textures_nodes` argument.

```
new_textures[0] = process_all_textures(
    prefix='mud_'
);
```

As you can see, the function now displays a more descriptive error message.

```
# Error: RuntimeError: file <maya console> line 17: No
texture nodes specified #
```

Let's look back at the section we added.

```
if (isinstance(textures, list) or
    isinstance(textures, tuple)):
    # ...
else:
    maya.cmds.error('No texture nodes specified');
```

To inform the user there is a problem, we implemented an **if** clause. What happened in this case is that the `textures` object was initialized to a value of None, the type of which is **NoneType**. We test the type of `textures` using the **isinstance()** function, which lets us compare an item in argument position 0 to an item in argument position 1. Consequently, because the **NoneType** failed to meet our test conditions, execution jumped down to the **else** block, which executes a command to display an error message.

if, elif, and else

An **if** statement tells the Python interpreter to execute a specific code block only if specified conditions are met. This process is called *branching*. A Python **if** statement consists of three separate parts:

- An **if** clause
- An optional **elif** clause or clauses
- An optional **else** clause

An **if** statement takes the following basic form, whereby execution only enters the block if `condition` is True.

```
if condition:
    # do something
```

It can be extended to include **elif** and **else** clauses, which are evaluated sequentially. In the following case, the **elif** and **else** clauses would be ignored if `condition` is True. If `condition` were False, then `another_condition` would be tested. If no **elif** clauses evaluate True, then execution defaults to the **else** block.

```
if condition:
    # do something
elif another_condition:
    # do something else
# any number of other elif clauses
else:
    # do default action
```

Using Comparison Operations

Conditional statements are the primary situation where you will use the Boolean operations discussed in Chapter 2. For example, **do_something_else()** would be executed in this case if `node_name` were "sandwich".

```
if node_name == 'ham':
    do_something();
elif node_name == 'sandwich':
    do_something_else();
```

However, there are in fact a number of comparison operators usable by many built-in types, which return Boolean values that you can use to evaluate conditions. These operators, described in Section 5.3 of Python Standard Library, are summarized in Table 3.1.

Table 3.1 Comparison Operators

Operation	Returns
x < *y*	True if *x* is less than *y*
x <= *y*	True if *x* is less than or equal to *y*
x > *y*	True if *x* is greater than *y*
x >= *y*	True if *x* is greater than or equal to *y*
x == *y*	True if *x* and *y* have the same value
x != *y*	True if *x* and *y* have different values
x **is** *y*	True if *x* and *y* have the same identity
x **is not** *y*	True if *x* and *y* have different identities

For example, the following statements would each print True.

```
v1, v2, v3, v4 = [1,2,3,4];
print(v1 < v2);
print(v3 > v1);
print(v4 <= v4);
print(v2 == v2);
```

On the other hand, the following statements would each print False.

```
v1, v2, v3, v4 = [1,2,3,4];
print(v1 >= v2);
print(v3 <= v1);
print(v4 > v4);
print(v2 != v2);
```

Multiple conditions can be tested to compute a single condition, in which case each one is evaluated in order. Consider the following example.

```
xform_list = maya.cmds.ls(type='transform');
camera_name = 'persp';
if (isinstance(xform_list, list) and
    camera_name in xform_list):
    print('%s is in the list of transforms'%camera_name);
```

Because `xform_list` is, in fact, a list object and the value referenced by `camera_name` is a member of the list, this condition will evaluate to True and the statement in the **if** clause will be printed. However, if `xform_list` were anything other than a list (e.g., a string, tuple, or set), the condition would immediately be evaluated to False, irrespective of whether `camera_name` happened to be in the collection or not, and the **if** block would be skipped.

Moreover, the same principle applies when testing multiple statements using the **or** operator.

```
cameras = maya.cmds.ls(type='camera');
shapes = maya.cmds.ls(type='shape');
camera_shape = 'perspShape';
if camera_shape in shapes or camera_shape in cameras:
    print(
        '%s has been found in one of the lists'%
        camera_shape
    );
```

As soon as a condition is found that evaluates to True, the **if** block is entered, no matter what the other values happen to be. For instance, in this example, the node name referenced by `camera_shape` happens to be in both lists. As such, when it is found in the `shapes` list, it does not need to be further tested for membership in the `cameras` list. You can exploit this principle to minimize the number of tests you perform.

7. Make the following modifications to the **process_all_textures()** function to require that the supplied prefix contains a trailing underscore if it is specified.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix');
    if (isinstance(pre, str) or
        isinstance(pre, unicode)):
        if not pre[-1] == '_':
            pre += '_';
    else: pre = '';
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        for texture in textures:
            new_texture_names.append(
                maya.cmds.rename(
                    texture,
                    '%s%s'%(pre, texture)
                )
            );
        return new_texture_names;
    else:
        maya.cmds.error('No texture nodes specified');
```

If you examine the test we implemented, you can see that it comprises a few steps. First, we test whether `pre` is a string or Unicode object. If this test fails, we drop to the **else** block, where `pre` is assigned an empty value. Otherwise, if `pre` is a string or Unicode object, we jump into the first **if** block and proceed to test whether the final character in `pre` is an underscore. If it is, then we do nothing, but if it is not, then we add an underscore.

8. Execute the following lines to create a new **file** node and then process it using a prefix without an underscore.

```
tex = [maya.cmds.shadingNode('file', asTexture=True)];
print('Before: %s'%tex);
tex = process_all_textures(
    texture_nodes=tex,
    prefix='metal'
);
print('After: %s'%tex);
```

As you can see from the output, the function now appends an underscore as needed, which other tools in your pipeline may require as part of your naming convention.

```
Before: [u'file1']
After:  [u'metal_file1']
```

Note that the **not** keyword can be used to test if an expression is not true. Consider the following test.

```
if not b:
```

If the expression represented by `b` evaluates to False, the statement as a whole evaluates to True. Otherwise, the statement will evaluate to False. In short, the **not** operator negates whatever follows it. The **not** operator can and should also be used to compare Boolean values to False, as in the following example.

```
# this statement
if x == False:
# can be written as
if not x:
```

Moreover, the **not** keyword is most often used to test against what is called an *implicit false*. There are certain values in Python that can be evaluated as Booleans and will return a value of False. The most common of these include "empty" values, such as 0, None, [], and ".

```
# this statement
a_list = list();
if len(a_list)==0:
    print('This list is empty');
# can be written as
if not a_list:
    print('This list is empty');
```

Finally, note that the **if** statement can also be used as part of an expression in Python 2.5 and later (e.g., Maya 2008 and later). The following example demonstrates this concept by presenting a pattern for emulating MEL's ternary operator using an **if** statement.

```
// MEL
$result = $value_1 == $value_2 ? "equal" : "not equal";
# Python
result = 'equal' if value_1 == value_2 else 'not equal';
```

Emulating Switches

As we noted previously, a component of an **if** statement is an optional **elif** clause, which is a shortened name for the "else if" pattern found in other languages. The **elif** statement is very useful, as it can appear multiple times and create multiple branches. It can be used to emulate the **switch** statement found in MEL.

```
// MEL
switch($func_test):
{
    case "A":
    {
        exec_func_A();
        break;
    }
    case "B":
    {
        exec_func_B();
        break;
    }
    default:
    {
        exec_default();
        break;
    }
}
# Python
if func_test is 'A':
    exec_func_A()
elif func_test is 'B':
    exec_func_B()
else:
    exec_default()
```

Because function names are accessible like any other name in Python, you can also use a dictionary to implement a **switch** in some cases. For example, the following dictionary-based approach is equivalent to the previous example.

```
func_dict = {
    'A':exec_funcA,
    'B':exec_funcB,
};
func_dict.setdefault(func_test, exec_default)();
```

continue and break

Along with conditional statements, Python provides a few other tools for branching specifically for use inside of loops: the **break** and **continue** statements.

The **break** statement is used to exit a loop immediately.

9. Execute the following lines to include a test in the **for** loop inside of **process_all_textures()**. If the current item is not found or is not a texture, then the loop exits immediately.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix');
    if (isinstance(pre, str) or
        isinstance(pre, unicode)):
        if not pre[-1] == '_':
            pre += '_';
    else: pre = '';
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        for texture in textures:
            if not (maya.cmds.ls(texture) and
                maya.cmds.nodeType(texture)=='file'):
                break;
            new_texture_names.append(
                maya.cmds.rename(
                    texture,
                    '%s%s'%(pre, texture)
                )
            );
        return new_texture_names;
    else:
        maya.cmds.error('No texture nodes specified');
```

10. Execute the following lines to supply the **process_all_textures()** function with a list of three items, two of which are invalid, and print the result.

```
new_textures = [
    'nothing',
    'persp',
    maya.cmds.shadingNode('file', asTexture=True)
];
print('Before: %s'%new_textures);
new_textures = process_all_textures(
    texture_nodes=new_textures,
    prefix='concrete_'
);
print('After: %s'%new_textures);
```

As you can see from the results, the loop simply exited when it reached the first invalid object, causing it to return an empty list.

```
Before: ['nothing', 'persp', u'file1']
After:  []
```

In contrast, the **continue** statement is used to skip an element in a loop's sequence, and is a much more useful alternative in this case.

11. Execute the following lines to change the **break** statement you implemented in step 9 into a **continue** statement. This alternative approach will simply skip over invalid nodes, rather than exiting the loop entirely.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix');
    if (isinstance(pre, str) or
        isinstance(pre, unicode)):
        if not pre[-1] == '_':
            pre += '_';
    else: pre = '';
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        for texture in textures:
            if not (maya.cmds.ls(texture) and
                maya.cmds.nodeType(texture)=='file'):
                continue;
            new_texture_names.append(
                maya.cmds.rename(
                    texture,
                    '%s%s'%(pre, texture)
                )
            );
        return new_texture_names;
    else:
        maya.cmds.error('No texture nodes specified');
```

12. Execute the following lines to supply the **process_all_textures()** function with a new list similar to the one you created in step 10, and print the results.

```
new_textures = [
    'nothing',
    'persp',
    maya.cmds.shadingNode('file', asTexture=True)
];
print('Before: %s'%new_textures);
new_textures = process_all_textures(
    texture_nodes=new_textures,
    prefix='concrete_'
);
print('After: %s'%new_textures);
```

As you can see from the output, the item with no match in the scene was skipped, as was the perspective camera's **transform** node. The resulting list contains only a valid **file** node.

```
Before: ['nothing', 'persp', u'file2']
After: [u'concrete_file2']
```

List Comprehensions

One of Python's key features is simple, readable syntax. Often when dealing with loops and conditionals, code can become obfuscated. Consider the following block of code that walks through a theoretical directory of textures and builds a list of names by splitting the file name and extension.

```
textures = [];
for file_name in folder:
    if '_diff' in file_name:
        texture_list.append(name.split('.')[0]);
```

Rewriting these lines as a list comprehension both reduces the amount of code in the function and in many cases makes the code more readable. You could rewrite the preceding hypothetical block as a list comprehension as in the following line.

```
textures = [
    name.split('.')[0] for name in folder if '_diff' in name
];
```

List comprehensions take the following basic form.

```
[expression_or_element for element in sequence if condition];
```

Here are a few other examples of common Python iteration patterns that can be rewritten as list comprehensions.

```
# join a folder and a name to create a path
root_path = 'C:\\path\\';
name_list = ['name_1','name_2','name_3','name_4'];
# using a for loop
full_paths = [];
for name in name_list:
    full_paths.append(root_path+name);
# as a list comprehension
full_paths = [root_path+name for name in name_list];


# filter a list into a new list
texture_list = ['rock_diff',
                'base_diff',
                'base_spec',
                'grass_diff',
                'grass_bump'];
# using a for loop
diff_list = [];
```

```
for texture in texture_list:
    if '_diff' in texture:
        diff_list.append(texture);
# as a list comprehension
diff_list = [
    texture for texture in texture_list if '_diff' in texture
];

# mapping a function to a list
some_strings = ['one','two','three','four'];
# using a for loop
uppercased = [];
for s in some_strings:
    uppercased.append(s.upper());
# as a list comprehension
uppercase = [s.upper() for s in some_strings];
```

As you can see, list comprehensions are a unique feature that allow for clear and concise construction of new sequences using a combination of the **for** statement and conditional statements.

The while Statement

Now that you have mastered the nuances of conditional statements and understand the concepts of iteration, we can discuss another loop statement that Python offers: **while**. The **while** statement, also found in MEL, is a type of loop that evaluates as long as a particular condition is True. The following template shows the basic form of a **while** loop.

```
while (expression_is_true):
    perform_task();
    update_expression_parameter();
```

A **while** statement enacts the following steps:

- The expression to the right of the **while** keyword is evaluated.
- If the conditional expression is True, then execution enters the **while** block.
- Statements in the **while** block are executed.
- Conditional parameters are updated for the next test.
- If the condition is False, then execution moves to the next statement outside of the **while** block.

Unlike the **for** statement, the Python **while** statement is a bit closer in form to its MEL counterpart, as the following code snippet demonstrates.

```
"""
MEL equivalent:
int $ctr = 0;
while ($ctr < 4)
```

```
{
    print("counted "+$ctr+"\n");
    $ctr += 1;
}
"""
ctr = 0;
while (ctr < 4):
    print('counted %s'%ctr);
    ctr += 1;
```

Although a **for** loop is more sensible for our purposes, implementing **process_all_textures()** with a **while** statement could look like the following example.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix');
    if (isinstance(pre, str) or
        isinstance(pre, unicode)):
        if not pre[-1] == '_':
            pre += '_';
    else: pre = '';
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        ctr = len(textures) -1;
        while ctr > -1:
            texture = textures[ctr];
            if (maya.cmds.ls(texture) and
                maya.cmds.nodeType(texture)=='file'):
                new_texture_names.append(
                    maya.cmds.rename(
                        texture,
                        '%s%s'%(pre, texture)
                    )
                );
            ctr -= 1;
        return new_texture_names;
    else:
        maya.cmds.error('No texture nodes specified');
```

Just like **for** statements, **while** statements are also helpful tools for iteration. In contrast, however, they make use of conditional statements to determine whether they should continue. Consequently, although **for** statements are useful tools for executing linear, repeatable steps, **while** statements may be useful in nonlinear situations. Moreover, **while** loops are helpful in situations that need to monitor other objects that may be manipulated outside the **while**

loop, or that have their values altered as a cascading effect of something that does happen in the **while** loop.

ERROR TRAPPING

When dealing with large programs, you often would like to be able to handle how errors are processed and reported back to the developer calling your function. These situations all concern a function's exit path, and Python provides several different options. Although we have already covered the **return** statement, we explore here some basic strategies for handling errors.

try, except, raise, and finally

In a perfect world, functions would perform exactly as intended on every call and our need to worry about handling flawed execution would be minimal. As this ideal is absolutely not the case, developers must be prepared not only to handle errors in code cleanly, but also to provide meaningful messages in such cases. Fortunately, Python provides some built-in features to address these situations:

- Exception and Error classes
- A **try** statement that encompasses exception handling and cleanup
- A **raise** statement that allows the developer to specify an Error class and message

Detailing Python's Exception and Error classes is beyond the scope of this chapter. It is strongly recommended that you consult Section 8 of Python Tutorial online for more detailed information on these topics.

The concept behind Exception and Error classes is that they allow a developer to halt the flow of execution and handle an unexpected or incorrect result. Triggering an exception or error is referred to as raising an exception (or error) and is done by using Python's **raise** statement.

```
raise StandardError('An Error Occurred');
# Error: StandardError: file <maya console> line 1: An Error
Occurred #
```

Raising an error in this case could also be achieved with alternative syntax.

```
raise StandardError, 'An Error Occurred';
```

1. Execute the following lines to raise a **TypeError** in **process_all_textures()** when the supplied `texture_nodes` argument is not a tuple or a list.

```
def process_all_textures(**kwargs):
    pre = kwargs.setdefault('prefix');
    if (isinstance(pre, str) or
```

```
        isinstance(pre, unicode)):
        if not pre[-1] == '_':
            pre += '_';
    else: pre = '';
    textures = kwargs.setdefault('texture_nodes');
    new_texture_names = [];
    if (isinstance(textures, list) or
        isinstance(textures, tuple)):
        for texture in textures:
            if not (maya.cmds.ls(texture) and
                maya.cmds.nodeType(texture)=='file'):
                continue;
            new_texture_names.append(
                maya.cmds.rename(
                    texture,
                    '%s%s'%(pre, texture)
                )
            );
        return new_texture_names;
    else:
        raise TypeError(
            'Argument passed was not a tuple or list'
        );
```

2. Execute the following line to try to pass a string to the `texture_nodes` argument when calling **process_all_textures()**.

```
process_all_textures(texture_nodes='this is a string')
```

Now if **process_all_textures()** is called without an argument for `texture_nodes`, a **TypeError** with the developer-supplied message is raised.

```
# Error: TypeError: file <maya console> line 24: Argument
passed was not a tuple or list #
```

Python supports a higher-level pattern for handling errors that allows a developer to try a block of code, handle any errors, and clean up any results. This pattern is called a **try** statement, and it consists of the following parts:

- A **try** clause that contains the code to be tested
- An optional **except** clause or clauses that handle any errors by type
- An optional **else** clause following an **except** clause, which specifies a block of code that must run if no errors occur in the **try** block
- An optional **finally** clause that must be executed when leaving the **try** block to perform any cleanup required, and that is always executed whether or not there was an error

A **try** statement must further include either an **except** clause or a **finally** clause. Either of the following code snippets is valid.

```
# example 1
try:
    # code to be tested
except:
    # raise an exception
# example 2
try:
    # code to be tested
finally:
    # clean up
```

You can also combine the two, in which case the **finally** block will execute whether or not an error occurs in the **try** block.

```
try:
    # code to be tested
except:
    # raise an exception
finally:
    # called no matter what, upon completion
```

You are also allowed to include an optional **else** clause following an **except** clause. Such an **else** clause specifies code that must run if no error occurred in the **try** block.

```
try:
    # code to be tested
except:
    # raise an exception
else:
    # called if try block is successful
```

It is also valid to use all four components if you choose.

```
try:
    # code to be tested
except:
    # raise an exception
else:
    # called if try block is successful
finally:
    # called no matter what, upon completion
```

While you have a variety of options, any errors raised by code in the **try** clause will only be handled by the **except** clause.

3. Execute the following lines to wrap an invalid call to **process_all_textures()** in a **try**-**except** clause.

```
try:
    process_all_textures();
except TypeError as e:
    print(e);
```

You should see the custom **TypeError** message you defined inside the function print as a result, which is the specified action in the **except** block.

```
Argument passed was not a tuple or list
```

When an exception is raised from inside a **try** clause, the Python interpreter checks the **try** statement for a corresponding **except** clause that defines a handler for the raised error type and executes the block of code contained within the **except** clause. This process is called catching the exception.

Because **process_all_textures()** raises a **TypeError** if it is called with an incorrect argument, we needed to tell the **except** clause to handle a **TypeError** specifically. Once the exception is caught, information regarding the exception can be extracted from the variable `e` and manipulated in the **except** clause's expression. To better understand this concept, consider the **except** statement on its own.

```
except TypeError as e:
```

Given that a **TypeError** was raised in **process_all_textures()** with the message "Argument passed was not a tuple or list", the **except** statement is functionally equivalent to the following code.

```
e = TypeError('Argument passed was not a tuple or list');
```

In this case, `e` is referred to as a target, meaning that the **TypeError** was bound to it using the **as** keyword. The variable `e` is not a required keyword; in fact, any legal variable name can be used as a target. Targets are also optional, though excluding them prevents you from obtaining any specific information about the exception. Any of the following statements use valid **except** syntax.

```
except TypeError:
except TypeError as my_error:
except TypeError, improper_type:
```

Hopefully you can see the tremendous value in protecting the execution of your code by properly handling errors. It is strongly recommended you refer to the Python documentation for more information on working with errors, as it represents one of Python's most helpful features for tracking down problems.

DESIGNING PRACTICAL TOOLS

Now that you have a firm understanding of some core programming concepts in Python, as well as knowledge of some important Maya commands, we will walk through a set of functions that can be used to perform some

real production tasks. Any time you design a new tool, it is always useful to plan out each of the steps it will need to perform. We will be designing a simple texture-processing framework according to the following specifications:

- Query the current scene for all the **file** nodes (textures).
- Iterate over the resulting **file** nodes.
- If the given list is empty, the main function should exit with an appropriate error message.
- Design a function that validates each node and verifies that it is assigned to an object and named properly.
- Process each node's texture in a separate function based on a type determined by the naming convention:
  - Textures with the suffix "_diff" will be processed as diffuse textures.
  - Textures with the suffix "_spec" will be processed as specular textures.
  - Textures with the suffix "_bump" will be processed as bump textures.
- The individual texture processors can raise errors, but errors should be handled by the main function. Ideally, the texture processors should return a success or fail status.
- The main function should return a list of textures that processed successfully, a list of textures that produced errors, and a list of skipped textures.
- The main function should save out a new scene once the textures have been reassigned.

Up to this point, all the example functions have been for illustrative purposes. We will be implementing all the functions and patterns from scratch even though we use names that we used in prior examples.

The first step is to implement the main function, **process_all_textures**(). We would start by coding a simple skeleton for the function, as in the following example.

```
import maya.cmds;
import os;
def process_all_textures(out_dir = os.getenv('HOME')):
    """
    A function that gets a list of textures from the current scene
    and processes each texture according to name
    """
    texture_nodes = []; # to be replaced
    processed_textures = [];
    error_textures = [];
    skipped_textures = [];
```

```
if not texture_nodes:
    maya.cmds.warning('No textures found, exiting');
    return (
        processed_textures,
        error_textures,
        skipped_textures
    );
for name in texture_nodes:
    if is_valid_texture(name):
        print('Processing texture %s'%name);
        as_type = None;
        status = False;
        texture = None;
        if '_diff' in name:
            status, texture = process_diffuse(
                name, out_dir
            );
            if status:
                processed_textures.append(texture);
                as_type = 'diffuse';
            else:
                error_textures.append(texture);
        elif '_spec' in name:
            status, texture = process_spec(
                name, out_dir
            );
            if status:
                processed_textures.append(texture);
                as_type = 'specular';
            else:
                error_textures.append(texture);
        elif '_bump' in name:
            status, texture = process_bump(
                name, out_dir
            );
            if status:
                processed_textures.append(texture);
                as_type = 'bump';
            else:
                error_textures.append(texture);
        if status:
            print(
                'Processed %s as a %s texture'%(
                    texture, as_type
                )
            );
        else:
            print('Failed to process %'%name);
```

```
        else:
            print('%s is not a valid texture, skipping.'%name);
            skipped_textures.append(name);
return (
    processed_textures,
    error_textures,
    skipped_textures
);
```

As previously noted, this function is simply a skeleton, and in fact does no actual work. Calling **process_all_textures()** at this point would return three empty lists, as it would fail the first **if** test.

The next step would be to generate a list of texture nodes. To accomplish this task, we search the scene and create a list by querying the attributes on all the file texture nodes. This task is accomplished with a simple call of the `ls` command, specifying that we are looking for **file** nodes.

```
texture_nodes = maya.cmds.ls(type='file');
```

This list is really the only piece of data required by the preceding function to return a result, given properly named textures. One of the shortcomings of such an approach, however, is that it would process *every* **file** node in the scene. Recall from the original specifications that an **is_valid_texture()** function should be used to verify a **file** node for processing. A minimum specification for the **is_valid_texture()** function could be:

- Check the **file** node to determine that it is connected to an assigned shader.
- Verify that the file has one of the predefined substrings ("_diff", "_bump", or "_spec").

The following implementation of **is_valid_texture()** satisfies these conditions.

```
def is_valid_texture(file_node):
    shaders = maya.cmds.listConnections(
        '%s.outColor'%file_node,
        destination=True
    );
    if not shaders: return False;
    for shader in shaders:
        groups = maya.cmds.listConnections(
            '%s.outColor'%shader
        );
        if not groups: return False
        for group in groups:
            meshes = maya.cmds.listConnections(
                group,
```

```
                type='mesh'
            );
            if meshes:
                if '_diff' in file_node:
                    return True;
                elif '_spec' in file_node:
                    return True;
                elif '_bump' in file_node:
                    return True;
    return False;
```

This function employs a fairly simple pattern that most MEL developers are probably familiar with:

- A **file** node is passed to the function.
- The node is queried for its connections to a shader via the **outColor** attribute.
- If any connected nodes are found, they are further searched for connections to meshes.
- If meshes are found, the function assumes the shader is assigned and queries the given **file** node's **fileTextureName** attribute.
- The texture name is checked for one of the predefined tokens, and if found, returns True.
- All other cases fail and return False.

Now, rather than calling **is_valid_texture()** as part of the main **for** loop, it can instead be called as part of a list comprehension and used to generate the initial list of **file** nodes.

```
texture_nodes = [
    i for i in maya.cmds.ls(type='file')
    if is_valid_texture(i)
];
```

Using this technique to get our list of textures, we could revise **process_all_textures()** to the following form.

```
def process_all_textures(out_dir = os.getenv('HOME')):
    """
    A function that gets a list of textures from the current
    scene and processes each texture according to name
    """
    texture_nodes = [
        i for i in maya.cmds.ls(type='file')
        if is_valid_texture(i)
    ];
    processed_textures = [];
```

```
error_textures = [];
skipped_textures = [];
if not texture_nodes:
    maya.cmds.warning('No textures found, exiting');
    return (
        processed_textures,
        error_textures,
        skipped_textures
    );
for name in texture_nodes:
    print('Processing texture %s'%name);
    as_type = None;
    status = False;
    texture = None;
    if '_diff' in name:
        status, texture = process_diffuse(
            name, out_dir
        );
        if status:
            processed_textures.append(texture);
            as_type = 'diffuse';
        else:
            error_textures.append(texture);
    elif '_spec' in name:
        status, texture = process_spec(
            name, out_dir
        );
        if status:
            processed_textures.append(texture);
            as_type = 'specular';
        else:
            error_textures.append(texture);
    elif '_bump' in name:
        status, texture = process_bump(
            name, out_dir
        );
        if status:
            processed_textures.append(texture);
            as_type = 'bump';
        else:
            error_textures.append(texture);
    if status:
        print(
            'Processed %s as a %s texture'%(
                texture, as_type
            )
        );
    else:
        print('Failed to process %s'%name);
```

```
    return (
        processed_textures,
        error_textures,
        skipped_textures
    );
```

Note that the main **for** loop has been simplified by excluding the main **else** clause, since it is no longer needed—our texture list is inherently valid since it is created using the list comprehension.

Now we can move on to the implementation of the actual texture-processing functions. For the most part, the texture-processing functions follow a similar pattern. Consequently, we will only discuss one of these functions in detail, **process_diffuse()**. To reiterate, this function should:

- Take a verified **file** node and extract the texture.
- Process the texture.
- On success:
  - A new shading network is created.
  - The new shader is assigned to the original object.
  - A status of True is returned along with the new texture name.
- On failure:
  - A warning is printed.
  - The original texture name is returned.

Given this design specification, **process_diffuse()** could be implemented as follows. Note that this implementation makes use of the maya.mel module, which allows you to execute statements in MEL to simplify the creation and setup of the new file texture node, as well as a set of nested calls for retrieving the shader's assigned object. We will examine the maya.mel module in further detail in coming chapters.

```
import maya.mel;
def process_diffuse(file_node, out_dir):
    """
    Process a file node's texture, reassign the new texture
    and return a status and texture name
    """
    status = False;
    texture = None;
    file_name = maya.cmds.getAttr('%s.ftn'%file_node);
    meshes = [];
    shaders = maya.cmds.listConnections(
        '%s.outColor'%'file1',
        destination=True,
    );
```

```
if shaders:
    for s in shaders:
        groups = maya.cmds.listConnections(
            '%s.outColor'%s
        );
        if groups:
            for g in groups:
                m = maya.cmds.listConnections(
                    g, type='mesh'
                );
                if m: meshes += m;
try:
    # processing code would be here
    new_file_name = file_name
    # new_file_name is the processed texture
    shader = maya.cmds.shadingNode(
        'blinn',
        asShader=True
    );
    shading_group = maya.cmds.sets(
        renderable=True,
        noSurfaceShader=True,
        empty=True
    );
    maya.cmds.connectAttr(
        '%s.outColor'%shader,
        '%s.surfaceShader'%shading_group
    );
    texture = maya.mel.eval(
        'createRenderNodeCB -as2DTexture "" file "";'
    );
    maya.cmds.setAttr(
        '%s.ftn'%file_node,
        new_file_name,
        type='string'
    );
    maya.cmds.connectAttr(
        '%s.outColor'%texture, '%s.color'%shader
    );
    for mesh in meshes:
        maya.cmds.sets(
            mesh,
            edit=True,
            forceElement=shading_group
        );
    status = True;
except:
    texture = file_node;
```

```
        status = False;
    return (status, texture);
```

With this function in place, the texture framework is almost complete. The final step is to generate a new scene name and save the new file. To keep things simple, **process_all_textures()** will take the new file name and an output directory as arguments. Below is the listing for the final version of **process_all_textures()**.

```
def process_all_textures(
    out_dir = os.getenv('HOME'),
    new_file='processed.ma'
):
    """
    A function that gets a list of textures from the current scene
    and processes each texture according to name
    """
    texture_nodes = [
        i for i in maya.cmds.ls(type='file')
        if is_valid_texture(i)
    ];
    processed_textures = [];
    error_textures = [];
    skipped_textures = [];
    if not texture_nodes:
        maya.cmds.warning('No textures found, exiting');
        return (
            processed_textures,
            error_textures,
            skipped_textures
        );
    for name in texture_nodes:
        print('Processing texture %s'%name);
        as_type = None;
        status = False;
        texture = None;
        if '_diff' in name:
            status, texture = process_diffuse(
                name, out_dir
            );
            if status:
                processed_textures.append(texture);
                as_type = 'diffuse';
            else:
                error_textures.append(texture);
        elif '_spec' in name:
            status, texture = process_spec(
```

```
                    name, out_dir
                );
                if status:
                    processed_textures.append(texture);
                    as_type = 'specular';
                else:
                    error_textures.append(texture);
            elif '_bump' in name:
                status, texture = process_bump(
                    name, out_dir
                );
                if status:
                    processed_textures.append(texture);
                    as_type = 'bump';
                else:
                    error_textures.append(texture);
            if status:
                print(
                    'Processed %s as a %s texture'%(
                        texture, as_type
                    )
                );
            else:
                print('Failed to process %s'%name);
        try:
            maya.cmds.file(
                rename=os.path.join(
                    out_dir, new_file
                )
            );
            maya.cmds.file(save=True);
        except:
            print('Error saving file, %s not saved.'%new_file);
        finally:
            return (
                processed_textures,
                error_textures,
                skipped_textures
            );
```

While this framework is somewhat naïve in implementation, it does present working examples of everything covered in this chapter, including function declarations, arguments, iteration, conditionals, error handling, **return** statements, and the use of Maya commands. This texture-processing framework could be made production-ready, and we encourage you to take this code and rework it for your own use or study.

CONCLUDING REMARKS

You now have a firm grasp of many of the fundamentals of programming with Python, including the ins and outs of functions, loops, branching, and error handling. While we covered a number of examples in this chapter, you are still only scratching the surface of Python in Maya. The functions we implemented barely make use of any advanced Python Standard Library functionality or language-specific features.

This chapter has introduced enough basic concepts that you should be able to work through all of the examples in further chapters in this book. Nevertheless, we encourage any developers interested in furthering their knowledge of Python and its capabilities to browse the online documentation, or to consult any of the other excellent Python-specific books available. Because you are now capable of creating Python programs, the next step is to organize them into modules, which we examine in Chapter 4.