Chapter 2
Beginning Python

Abstract Before you can create GIS Python scripts, you need to know where to write and run the code and you need a familiarity with basic programming concepts. If you're unfamiliar with Python, it will be worthwhile to take some time to go over the basics presented in this chapter before commencing the next chapter. This chapter discusses Python development software for Windows® operating systems, interactive mode and scripting, running scripts with arguments, and some fundamental characteristics of Python, including comments, keywords, indentation, variable usage and naming, traceback messages, dynamic typing and built-in modules, functions, constants, and exceptions.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Test individual lines of code interactively in a code editor.
- Run Python scripts in a code editor.
- Differentiate between scripting and interactive code editor windows.
- Pass input to a script.
- Explain the advantages of using an integrated development environment, over a general purpose text editor.
- Match code text color with code components.
- Define eight fundamental components of Python code.

2.1 Where to Write Code

Python scripts can be written and saved in any text editor; when a file is saved with a '.py' extension, the computer interprets it as a Python script. However, an *integrated development environment* (IDE), a software application designed for computer programming, is a better choice than a general purpose text editor, because it is tailored for programming. The *syntax* of a programming language is the set of rules that define how to form code statements that the computer will be able to interpret in that language. An IDE can check code syntax, highlight special code

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_2

statements, suggest ways to complete a code statement, and provide special tools, called debuggers, to investigate errors in the code.

The introductory example in Chapter 1 used the ArcMap Python window. The Python window embedded in ArcGIS desktop applications has some IDE functionality, such as Help pages and automatic code completion. It allows the user to save code (right-click>Save as) or load a saved script (right-click>Load), but it is missing some of the functionality of a stand-alone IDE. For example, it provides no means to pass input into a script and it doesn't provide a debugger. Stand-alone IDE's are also lightweight and allow scripts to be run and tested outside of ArcGIS software. For these reasons, we will mainly use a popular stand-alone IDE called PythonWin.

The PythonWin IDE provides two windows for two modes of executing code: an interactive environment and a window for scripts (Figure 2.1). The interactive environment works like this:

1. The user types a line of code in the interactive window (for example, `print 'Hello'`).
2. The user presses 'Enter' to indicate that the line of code is complete.
3. The single line of code is run.

The interactive mode is a convenient way to try out small pieces of code and see the results immediately. The code written in the interactive window is not saved

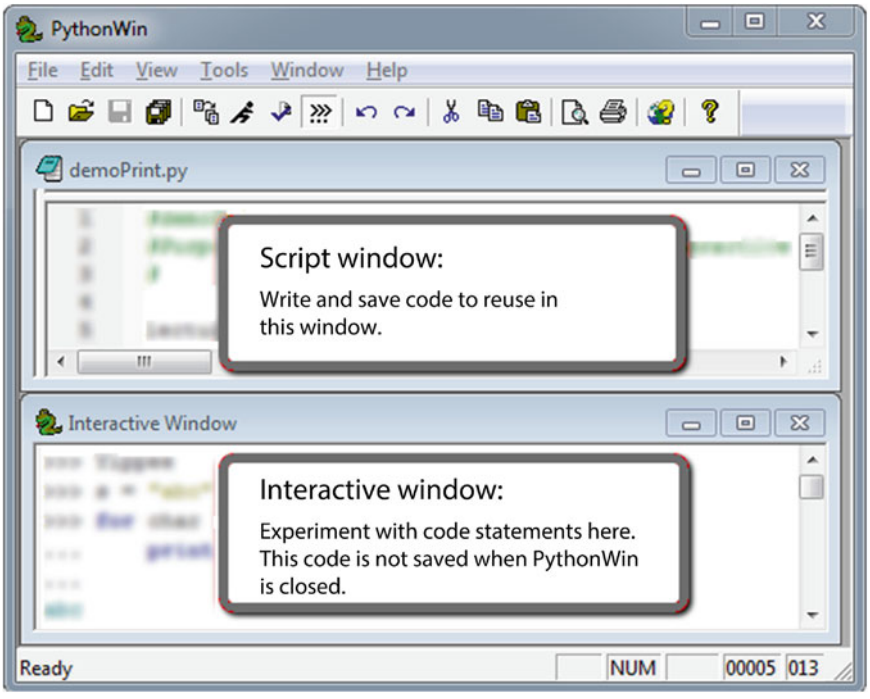


Figure 2.1 PythonWin has two windows: one for scripts and one for interactive coding.

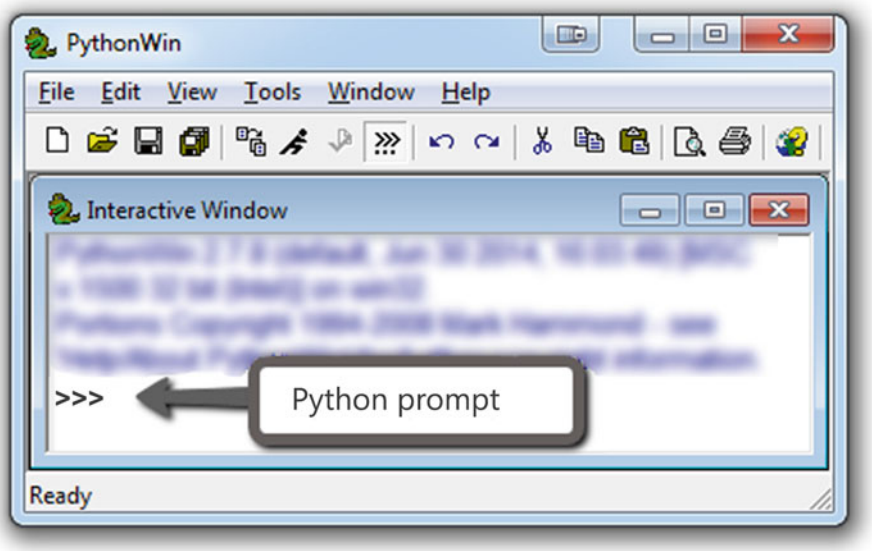


Figure 2.2 The Interactive window opens when PythonWin is launched.

when the IDE is closed. Often we want to save lines of related code for reuse, in other words, we want to save scripts. A *Python script* is a program (a set of lines of Python code) saved in a file with the '.py' extension. A script can later be reopened in an IDE and run in its entirety from there.

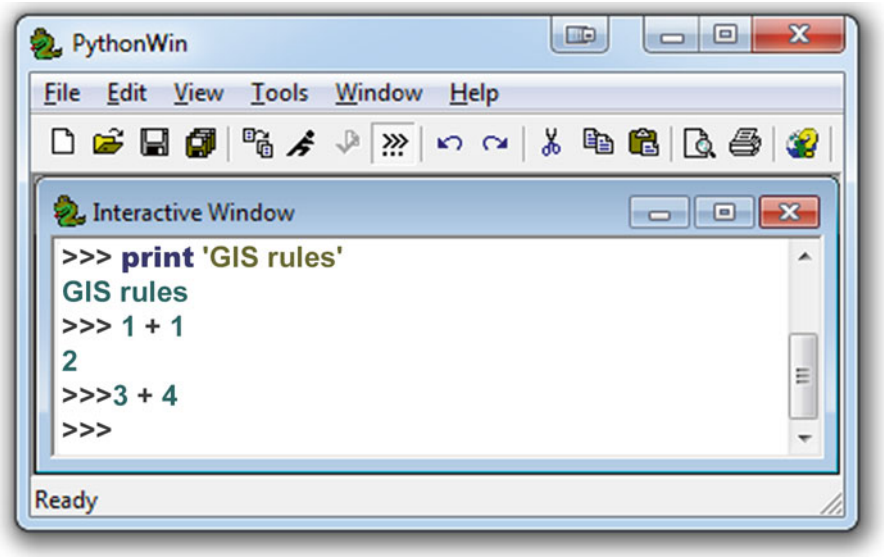
Python is installed automatically when ArcGIS is installed, but PythonWin is not. To follow along in the rest of the chapter, install PythonWin and PyScripter based on the instructions in Exercise 1. Then launch PythonWin and locate the Python prompt, as shown in Figure 2.2.

PyScripter is also a good choice as a Python IDE. PyScripter's equivalent of PythonWin's Interactive Window is the Python Interpreter window. PyScripter is more complex than PythonWin, but has additional functionality, such as interactive syntax checking, window tabs, variable watch tools, and the ability to create projects.

2.2 How to Run Code in PythonWin and PyScripter

Once you have installed PythonWin, you'll need to understand how to use the Interactive Window and script windows. When PythonWin is first opened, only the Interactive Window is displayed, because you're not editing a script yet. Python statements in the window labeled 'Interactive Window' are executed as soon as you

finish typing the line of code and press the 'Enter' key. The >>> symbol in the Interactive Window is the *Python prompt*. Just like the prompt in the ArcGIS Python Window, it indicates that Python is waiting for input.



Type something at the Python prompt and press the 'Enter' key. On the following line, Python displays the result. Enter `print "GIS rules"` and it displays 'GIS rules'. Enter `1 + 1` and Python displays `2`. Backspace and then enter `3 + 4`. Python doesn't display `7` because there is no space between the prompt and the Python code. There must be exactly one space between the prompt and the input. This problem only occurs if you 'Backspace' too far. When you press the 'Enter' key, the cursor is automatically placed in the correct position, ready for the next command. PyScripter avoids this issue by not allowing you to remove that space after the prompt.

Instead of showing screen shots of the Interactive Window, this book usually uses text to display these interactions. For example, the screen shot above would be replaced with this text:

```
>>> print 'GIS rules.'
GIS rules.
>>> 1 + 1
2
>>> 3 + 4
>>>
```

Table 2.1 PythonWin buttons, keyboard shortcuts, and menu instructions.

Action	Button	Keyboard shortcut	Menu
Create a new script		Ctrl + n, Enter	File > New > OK
Open a script		Ctrl + o	File > Open...
Save a script		Ctrl + s	File > Save
Close script			File > Close
Run a script		Ctrl + r, Enter	File > Run > OK
Tile the windows			Window > Tile
Show line numbers			View > Options > Editor > Set Line numbers > – 30
Shift focus between windows		Ctrl + Tab	
Open/close Interactive Window			View > Interactive Window
Clear the window		Ctrl + a, Delete	Edit > Select All, Edit > Delete
Toggle whitespace		Ctrl + w	View > Options > Tabs and whitespace > check 'View whitespace'

Note Interactive Window examples are given as text with Python prompts. To try the examples yourself, note that the prompt indicates that you should type what follows on that line in the Interactive Window, but don't type the >>> marks.

The interactive mode is helpful for testing small pieces of code and you'll continue to use this as you develop code, but ultimately you will also be writing Python scripts. Editing scripts in an IDE is similar to working with a text editor. You can use buttons (or menu selections or keyboard shortcuts) for creating, saving, closing, and opening scripts. Table 2.1 shows these options. If you're unfamiliar with PythonWin, it would be useful to walk through the following example, to practice these actions.

Create a new blank script as follows:

1. Choose File > new or press **Ctrl + n** or click the 'New' button  .
2. Select 'Python script'.
3. Click 'OK'.

A new blank script window with no >>> prompt appears. Next, organize the display by tiling the windows and turning on line numbers. To tile the windows as shown in Figure 2.1, click in the script window, then select: Window menu>Tile. When you have two or more windows open in PythonWin, you need to be aware of the focus. The *focus* is the active window where you've clicked the mouse most recently. If you click in the Interactive Window and tile the windows again, the Interactive Window will be stacked on top, because the 'Tile' option places the focus window on top. To display the line numbers next to the code in the script window, select the View menu>Options>Editor and set 'Line numbers' to 30. This sets the width of the line numbers margin to 30 pixels. Next, add some content to the script, by typing the following lines of code in the new window:

```
print 'I love GIS...'
print 'and soon I will love Python!'
```

Text font and color in the script window is used to differentiate code elements as we will discuss in an upcoming section. Next, save the Python script in 'C:/gispy/scratch'. To save the script:

1. Click File>Save or press **Ctrl+s** or click the 'Save' button  .
2. A 'Save as' window appears. Browse to the 'C:\gispy\scratch' directory.
3. Set the file name to 'gisRules.py'.
4. Click 'Save'.

Beware of your focus; if you select **Ctrl+s** while your focus is in the Interactive Window, it will save the contents of the Interactive Window instead of your script. Confirm that you can view the 'gisRules.py' file name in the 'C:\gispy\scratch' directory in ArcCatalog and the file extension in Windows Explorer. If not, see the box titled "Listing Python scripts in ArcCatalog and Windows Explorer."

Listing Python Scripts in ArcCatalog and Windows Explorer

I. By default, ArcCatalog does not display scripts in the TOC. To change this setting, use the following steps:

1. Customize menu>ArcCatalog Options>File Types tab>New Type
2. Enter *Python* for Description and *py* for File extension
3. Import File Type From Registry…
4. Click 'OK' to complete the process.

It may be necessary to restart ArcCatalog in order to view the python files.

II. By default, Windows® operating system users can see Python scripts in Windows Explorer, but the file extension may be hidden. If you don't see the '.py' extension on Python files, change the settings under the Windows Explorer tools menu. The procedure varies depending on the Windows® operating system version. For example, in Windows 7, follow these instructions:

1. Tools>Folder Options…>View.
2. Uncheck 'Hide extensions for known file types'.
3. Click 'Apply to All Folders'.

and in Windows 8, click View, then check 'File name extensions'.

Back in PythonWin, run the program:

1. Select File>Run or press **Ctrl+r** or click the 'Run' button  .
2. A 'Run Script' window appears. Click 'OK'.

PythonWin will run the script and you should see these results in the Interactive Window:

```
>>> I love GIS...
and soon I will love Python!
```

Code from the script prints output in the Interactive Window. PythonWin prints the first line in black and the other in teal; in this case, the text coloring is inconsequential and can be ignored.

With the focus in the 'gisRules.py' script window, select File>Close or click the X in the upper right corner of the window to close the script. Next, we'll reopen 'gisRules.py' to demonstrate running a saved script. To open and rerun it:

1. Select File>Open or press **Ctrl+o** or click the 'Open' button  .
2. Browse to the file, 'gisRules.py' in C:\gispy\scratch.
3. Select File>Run or press **Ctrl+r** or click the 'Run' button  .
4. A 'Run Script' window appears. Click 'OK'.

You will see the same statements printed again in the Interactive Window. Clearing the Interactive Window between running scripts to make it easier to identify new output. To do so, click inside the Interactive Window to give it focus. Then select all the contents (**Ctrl+a**) and delete them. PythonWin allows you to open multiple script windows, so you can view more than one script at a time, but it only opens one Interactive Window. To open and close the Interactive Window, click on the button that looks like a Python prompt  . Table 2.1 summarizes the actions described in this example.

The buttons and keyboard shortcuts are similar in PyScripter. PyScripter uses Ctrl+n, Ctrl+o, Ctrl+s, and Ctrl+Tab in the same way. One notable difference is that running a script is triggered by the green arrow button  and keyboard shortcut, Ctrl+F9 (instead of Ctrl+r). Closing a window is Ctrl+F4. Additional options can be configured by going to Tools>Options>IDE Shortcuts.

2.3 How to Pass Input to a Script

Now you know how to run lines of code directly in the Interactive Window and how to create and run Python scripts in PythonWin. You also need to know how to give input to a script in PythonWin. Getting input from the user enables code reuse without editing the code. User input, referred to as *arguments*, is given to the script in PythonWin through the 'Run Script' window. To use arguments, type them in the 'Run Script' window text box labeled 'Arguments'. To use multiple arguments, separate them by spaces.

This example runs a script 'add_version1.py' with no arguments and then runs 'add_version2.py', which takes two arguments:

1. In PythonWin open (**Ctrl+o**) 'add_version1.py', which looks like this:

```
# add_version1.py: Add two numbers.
a = 5
b = 6
c = a + b
# format(c) substitutes the value of c for {0}.
print 'The sum is {0}.'.format(c)
```

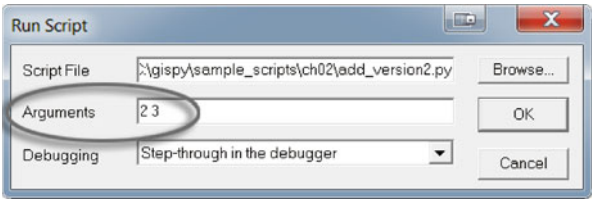
2. Run the script (**Ctrl+r>OK**). The output in the Interactive Window should look like this:

```
>>> The sum is 11.
```

3. 'add_version1.py' is so simple that it adds the same two numbers every time it is run. 'add_version2.py' instead adds two input numbers provided by the user. Open 'add_version2.py', which looks like this:

```
# add_version2.py: Add two numbers given as arguments.
# Use the built-in Python sys module.
import sys
# sys.argv is the system argument vector.
# int changes the input to an integer number.
a = int(sys.argv[1])
b = int(sys.argv[2])
c = a + b
print 'The sum is {0}.'.format(c)
```

4. This time, we will add the numbers 2 and 3. To run the script with these two arguments, select **Ctrl+r** and in the 'Run Script' window 'Arguments' text box, place a 2 and 3, separated by a space.



5. Click '**OK**'. The output in the Interactive Window should look like this:

```
>>> The sum is 5.
```

The beginning of 'add_version2.py' differs from 'add_version1.py' so that it can use arguments. The new lines of code in 'add_version2.py' use the system argument vector, `sys.argv`, to get the values of the arguments that are passed into the script. `sys.argv[1]` holds the first argument and `sys.argv[2]` holds the second one. We'll revisit script arguments in more depth in an upcoming chapter. For now, you know enough to run some simple examples.

2.4 Python Components

The color, font, and indentation of Python code, as it appears in an IDE, highlights code components such as comments, keywords, block structures, numbers, and strings. This special formatting, called *context highlighting*, provides extra cues to help programmers. For example, the keyword `import` appears as bold blue in a script in PythonWin, but if the word is misspelled as 'improt', it will have no special formatting. The script, 'describe_fc.py', in Figure 2.3, shows several highlighted code components. 'describe_fc.py' prints basic information about each feature class in a workspace. To try this example, follow steps 1–5:

1. In ArcCatalog, preview the two feature classes ('park.shp' and 'fires.shp') in 'C:/gispy/data/ch02'. Before moving on to step 2, click on the ch02 directory in the ArcCatalog table of contents and select F5 to refresh the view and release the locks on the feature classes.
2. Open the 'describe_fc.py' in PythonWin.
3. Launch the 'Run Script' window (**Ctrl+r**).
4. Type "C:/gispy/data/ch02" in the Arguments text box. The script uses this argument as the data workspace.
5. Click 'OK' and confirm that the output looks like the output shown in Figure 2.4.

```
# describe_fc.py
# Purpose: Print information about each feature class in a workspace.
# Usage: Workspace
# Example input: C:/gispy/data/ch02
# Output: A list of basic information about each feature class.
# Author: Lou Lou Who 7/20/2055

import arcpy, sys

# GET the input workspace from the user.
arcpy.env.workspace = sys.argv[1]

# GET a list of the feature classes in the workspace.
fcs = arcpy.ListFeatureClasses()

# PRINT basic information about each feature class in the folder.
print 'Feature classes in folder {0}:'.format(arcpy.env.workspace)
for fc in fcs:
    desc = arcpy.Describe(fc)
    print 'Name:         {0}'.format(fc)
    print 'Data type:    {0}'.format(desc.dataType)
    print 'Data class:   {0}'.format(desc.dataSetType)
    print 'Type:         {0}'.format(desc.featureType)
    print 'Shape type:   {0}'.format(desc.shapeType)
    print 'Has M:        {0}'.format(desc.hasM)
    print 'Has Z:        {0}'.format(desc.hasZ)
    print
print 'Feature class list complete.'

```

Figure 2.3 Python script 'describe_fc.py' as it appears in an IDE.

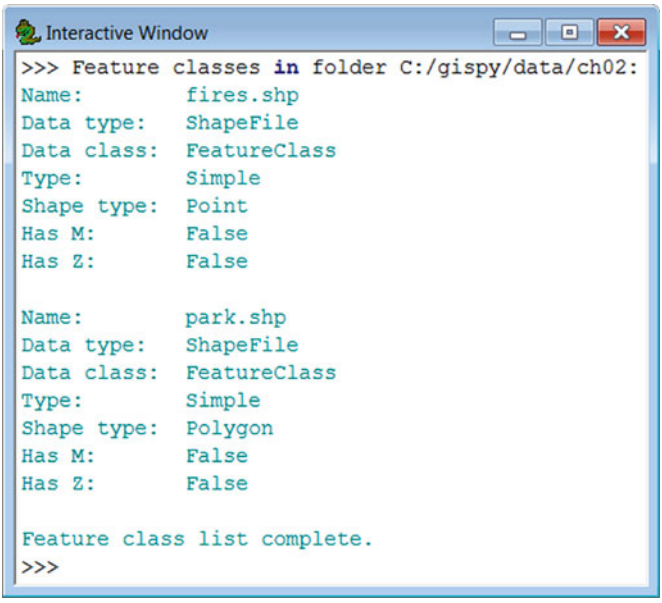


Figure 2.4 Output printed in the PythonWin Interactive Window when 'describe_fc.py' (Figure 2.2) runs.

Figure 2.3 shows the script as it is displayed with the default settings in PythonWin. The italic text, bold text, and indentation correspond to comments, keywords, and block structures, respectively. These components along with variables and assignment statements are discussed next.

2.4.1 Comments

Text lines shown in italics by the IDE are *comments*, information only included for humans readers, not for the computer to interpret. Anything that follows one or more hash sign (#) on a line of Python code is a comment and is ignored by the computer. By default, PythonWin displays comments as italicized green text when the comment starts with one hash sign and italicized gray text when the comment starts with two or more consecutive hash signs. Comments have several uses:

- Provide metadata—script name, purpose, author, data, usage (input), sample input syntax, expected output. These comments are placed at the beginning of the script (lines 1–6 in 'describe_fc.py').
- Outline—for the programmer to fill in the code details and for the reader to glean the overall workflow. For example, an outline for 'describe_fc.py' looks like this:

```
# GET the input workspace from the user.
# GET a list of the feature classes in the workspace.
# PRINT basic information about each feature class in the folder.
```

- Clarify specific pieces of code—good Python code is highly readable, but sometimes comments are still helpful. Skilled Python programmers use expository commenting selectively.
- Debug—help the programmer isolate mistakes in the code. Since comments are ignored by the computer, creating 'commented out' sections, can help to focus attention on another section of code. Comment out or uncomment multiple lines of code in PythonWin by selecting the lines and clicking **Alt+3/Alt+4** (or right-click and choose **Source code>Comment out region/Uncomment region**).

2.4.2 Keywords

Python keywords, words reserved for special usage, are shown by default in bold blue in PythonWin. 'describe_fc.py' in Figure 2.3 uses keywords `import`, `print`, `for`, and `in`. Table 2.2 gives a list of Python keywords. Python is case sensitive; keywords must be written in all lower case.

Table 2.2 Python 2.7 keywords.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

2.4.3 Indentation

Indentation is meaningful in Python. Notice that lines 19–27 are indented to the same position in 'describe_fc.py.' The code in line 18, which starts with the `for` keyword, tells Python to repeat what follows for each feature class listed in the input directory. Lines 19–27 are indented because they are the block of code that gets repeated. The Name, Data type, and so forth are printed for each feature class. The `for` keyword structure is an example of a Python *block structure*, a set of related lines of code (a block of code). Block structures will be discussed in more detail later, but for now, it's useful to have some understanding of the significance of indentation in Python. Items within a block structure are sequential code statements indented the same amount to indicate that they are related. The first line of code dedented (moved back a notch) after a block structure does not belong to the block structure. 'describe_fc.py' prints 'Feature class list complete' only once, because line 28 is dedented (the opposite of indented). Python does not have mandatory statement termination keywords or characters such as 'end for' or curly brackets to end the block structure; indentation is used instead. For example, if lines 20–27 were dedented, only one feature class description would be printed, the last one in the list. Indentation and loops will be discussed in more detail in an upcoming chapter.

2.4.4 Built-in Functions

'describe_fc.py' uses the `print` keyword to print output to the Interactive Window. This works because `print` is both a keyword and a built-in function. A *function* is a named set of code that performs a task. A *built-in function* is a function supplied by Python. Use it by typing the name of the function and the input separated by commas, usually inside parentheses. A code statement that uses a built-in function has this general format:

```
functionName(argument1, argument2, argument3,...)
```

The built-in `print` function, discussed in more detail in Chapter 3, is an exception to this rule; it does not require parentheses. In Python 2.7, which is the version of Python currently used by ArcGIS, the parentheses are optional in print statements. In Python 3.0 and higher, they are required.

Programming documentation uses special terminology related to dealing with functions, such as 'calling functions', 'passing arguments', and 'returning values'. These terms are defined here using built-in function examples:

- A code statement that invokes a function is referred to as a *function call*. We say we are 'calling the function', because we call upon it by name to do its work. Think of a function as if it's a task assigned to a butler. If you want him to make tea, you need to call on the butler to do so. You don't need to know any more details about how the tea is made, because he does it for you. There is no Python `make_tea` function, so we'll look at the built-in `round` function instead. The following line of code is a function call that calls the `round` function to round a number:

```
>>> round(1.7)
2.0
```

- Providing input for the function is referred to as *passing arguments* into the function.
- When you call the butler to make tea, you need to tell him if you want herbal or green tea or some other kind. The type of tea would be provided as an *argument*. The term 'parameter' is closely related to arguments. *Parameters*, are the pieces of information that can be specified to a function. The `round` function has one required parameter, a number to be rounded. The specific values passed in when calling the function are the arguments. The number `1.7` is used as the argument in the example above. An argument is an expression used when calling the function. The difference between the terms 'argument' and 'parameter' is subtle and often these terms are used interchangeably. The following line of code calls the built-in `min` function to find the minimum number. Here, we pass in three comma separated arguments and the function finds the smallest of the three values:

```
>>> min(1, 2, 3)
1
```

- Some functions come up with results from the actions they perform. Others do some action that may make changes or print information, but don't send a result to the caller. Those that do are said to *return a value*. When we ask our butler to do something, he goes away and does his work. Sometimes he returns with a result—like the cup of tea we asked for. Other times, he just performs his duty and there's no value returned—he dimmed the lights, if that's what you requested, but he doesn't bring your slippers if you only asked him to adjust the lighting. The built-in `round` function returns the rounded number and the `min` function returns the minimum value. These return values are printed in the Interactive

Window, but would not be printed in a script. By contrast, the `help` function is designed to print help about an argument, but it returns nothing. The following line of code calls the built-in `help` function to print help documentation for the `round` function. The name of the function is passed as an argument:

```
>>> help(round)
Help on built-in function round in module __builtin__:

round(...)
round(number[, ndigits]) -> floating point number

Round a number to a given precision in decimal digits (default
0 digits).
This always returns a floating point number. Precision may be
negative.
```

The help prints a *signature*, a template for how to use the function which lists the required and optional parameters. The first parameter of the `round` function is the number to be rounded. The second parameter, `ndigits`, specifies the number of digits for rounding precision. The square brackets surrounding `ndigits` mean that it is an optional argument. The arrow pointing to 'floating point number' means that this is the type of value that is returned by the function.

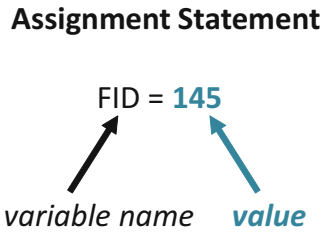
Other sections of this book employ additional built-in functions (e.g., `enumerate`, `int`, `float`, `len`, `open`, `range`, `raw_input`, and `type`). Search online with the terms, 'Python Standard Library built-in functions' for a complete list of built-in functions and their uses.

2.4.5 Variables, Assignment Statements, and Dynamic Typing

Viewing 'describe_fc.py' in PythonWin also shows script elements in cyan-blue and olive-green. By default in PythonWin and PyScripter, numeric data types are colored cyan-blue and string data types are colored olive-green. All objects in Python have a data type. To understand data types in Python you need to be familiar with variables, assignment statements, and dynamic typing. Variables are a basic building block of computer programming. A programming *variable*, is a name that gets assigned a value (it's like an algebra variable, except that it can be given non-numeric values as well as numeric ones). The following statement assigns the value 145 to the variable FID:

```
>>> FID = 145
```

This line of code is called an assignment statement. An *assignment statement* is a line of code (or statement) used to set the value of a variable. An assignment statement consists of the variable name (on the left), a value (on the right) and a single equals sign in the middle.



To print the value of a variable inside a script, you need to use the `print` function. This works in the Interactive Window too, but it is not necessary to use the `print` function in the Interactive Window. When you type a variable name and press the 'Enter' key in the Interactive Window, Python prints its value.

```
>>> inputData = 'trees.shp'
>>> inputData
'trees.shp'
```

A programming variable is similar to an algebra variable, except that it can be given non- numeric values, so the data type of a variable is an important piece of information. In other programming languages, declaration statements are used to specify the type. Python determines the data type of a variable based on the value assigned to it. This is called *dynamic typing*. You can check the data type of a variable using the built-in `type` function. The following tells us that `FID` is an `'int'` type variable and `inputData` is an `'str'` type variable:

```
>>> type(FID)
<type 'int'>
>>> type(inputData)
<type 'str'>
```

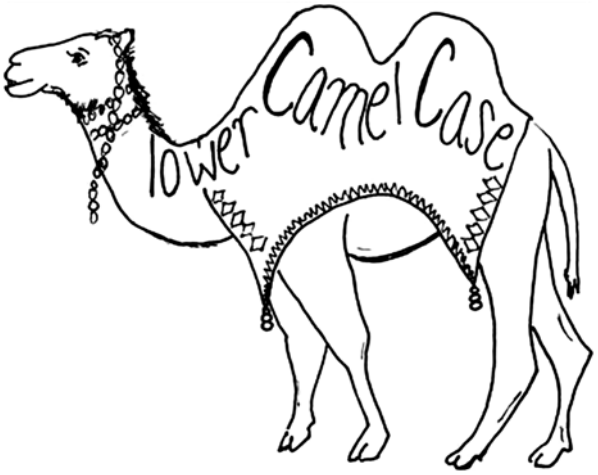
`'int'` stands for integer and `'str'` stands for string, (variable types which are discussed in detail in the Chapter 3). Due to dynamic typing, the type of a variable can change within a script. These statements show the type of a variable named `avg` changing from integer to string:

```
>>> avg = 92
>>> type(avg)
<type 'int'>
>>> avg = 'A'
>>> type(avg)
<type 'str'>
```

First, Python dynamically sets the data type of `avg` as an integer, since 92 is an integer. Python considers characters within quotes to be strings. So when `avg` is set to `'A'`, it is dynamically typed to string. Dynamic typing is agile, but beware that if you inadvertently use the same name for two variables, the first value will be overwritten by the second.

2.4.6 Variables Names and Tracebacks

Variable names can't start with numbers nor contain spaces. For names that are a combination of more than one word, underscores or capitalization can be used to break up the words. Capitalizing the beginning of each word is known as camel case (the capital letters stick up like the humps on a camel). This book uses a variation called lower camel case—all lower case for the first word and capitalization for the first letter of the rest. For example, `inputRasterData` is lower camel case.



Variable names are case sensitive. Though `FID` has a value of `145` because we assigned it in a previous example, in the following code, Python reports `fid` as undefined:

```
>>> fid
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
NameError: name 'fid' is not defined
>>> FID
145
```

When we attempt to check the value of `fid`, PythonWin prints an error message called a *traceback*. The traceback traces the steps back to where the error occurred. This example was typed in the Interactive Window, so it says 'interactive input', line 1. The last line of the traceback message explains the error. Python doesn't recognize 'fid'. It was never assigned a value, so it is considered to be an undefined object. When Python encounters an undefined object, it calls this kind of error a `NameError`.

Tip Look for an explanation of the error in the last line of a traceback message.

Keywords cannot be used as variable names. The following code attempts to use the keyword **`print`** as a variable name:

```
>>> print = 'inputData'
Traceback ( File "<interactive input>", line 1
print = 'inputData'
        ^
SyntaxError: invalid syntax
```

Again, a traceback error message is printed, but this time, the message indicates invalid syntax because the keyword is being used in a way that it is not designed to be used. This statement does not conform with the rules about how print statements should be formed, so it reports a `SyntaxError`.

Python ensures that keywords are not used as variable names by reporting an error; however, Python does not report an error if you use the name of a built-in function as variable name. Making this mistake can cause unexpected behavior. For example, in the code below, the built-in `min` function is working correctly at the outset. Then `min` is used as a variable name in the assignment statement `min = 5` Python accepts this with no error. But this makes `min` into an integer variable instead of a built-in function, so we can no longer use it to find the minimum value:

```
>>> type(min)
<type 'builtin_function_or_method'>
>>> min(1, 2, 3)
1
>>> min = 5
>>> min(1, 2, 3)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: 'int' object is not callable
>>> type(min)
<type 'int'>
```

A `TypeError` is printed the second time we try to use `min` to find the minimum. `min` is now an `int` type so it can no longer be called as a function. To restore `min` as a built-in function, you must restart PythonWin.

2.4.7 Built-in Constants and Exceptions

In addition to built-in functions, Python has built-in constants and exceptions. No built-in names should be used as variable names in scripts to avoid losing their special functionality. Built-in constants such as, `None`, `True`, and `False`, are assigned their values as soon as Python is launched, so that value is available to be used anywhere in Python. The built-in constant `None` is a null value placeholder. The data type of `True` and `False` is `'bool'` for boolean. Boolean objects can either be `True` or `False`:

```
>>> type(True)
<type 'bool'>
>>> True
True
```

The built-in constants `True` and `False` will be used in upcoming chapters to set variables that control aspects of the geoprocessing environment. For example, the following lines of code change the environment to allow geoprocessing output to overwrite existing files:

```
>>> import arcpy
>>> arcpy.env.overwriteOutput = True
```

The built-in exceptions, such as `NameError` or `SyntaxError`, are created for reporting specific errors in the code. Built-in exceptions are common errors that Python can identify automatically. An exception is *thrown* when one of these errors is detected. This means a traceback message is generated and the message is printed in the Interactive Window; if the erroneous code is in a script, the script will stop running at that point.

New programmers often encounter `NameError`, `SyntaxError`, and `TypeError` exceptions. The `NameError` usually occurs because of a spelling or capitalization mistake in the code. The `SyntaxError` can occur for many reasons, but the underlying problem is that one of the rules of properly formed Python has been violated. A `TypeError` occurs when the code attempts to use an operator differently from how it's designed. For example, code that adds an integer to a string generates a `TypeError`.

There are many other types of built-in exceptions. The names of these usually have a suffix of 'Error'. A traceback message is printed whenever one of these exceptions is thrown. The final line of traceback error messages states the name of the exception. We'll revisit exceptions and tracebacks in upcoming chapters.

The built-in `dir` function can be used to print a list of all the built-ins in Python. Type `dir(__builtins__)` to print the built-ins in PythonWin (there are two underscores on each side).

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', ..., 'str', 'sum', 'super', 'tuple', 'type',
'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Only a few of the items in the list are printed here, as you'll see when you run the code, there are over 100 built-ins. The built-in constants and built-in exceptions are interspersed in the beginning of this list. The built-in functions are listed next, starting with `abs`.

2.4.8 Standard (Built-in) Modules

When Python is installed, a library of standard modules is automatically installed. A *module* is a Python file containing related Python code. Python's standard library covers a wide variety of functionality. Examples include math functions, file copying, unzipping files, graphical user interfaces, and even Internet protocols for retrieving online data. To use a module you first use the `import` keyword. The import statement can be applied to one or more modules using the following format:

```
import moduleName1, moduleName2, ...
```

Once a module is imported, its name can be used to invoke its functionality. For example, the following code imports the standard `math` module and then uses it to convert 180 degrees to radians:

```
>>> import math
>>> math.radians(180)
3.141592653589793
```

The online 'Python Standard Library' contains documentation of the vast set of standard modules. For now, we'll just highlight two modules (`sys` and `os`) that are used in early chapters and introduce others as the need arises.

The `sys` module gives access to variables that are used by the *Python interpreter*, the program that runs Python. For example, the following code imports the `sys` module and prints `sys.path`:

```
>>> import sys
>>> sys.path
['C:\\gispy\\sample_scripts\ch02', u'c:\\program files
(x86)\\ArcGis\\desktop10.3\\arcpy', 'C:\\Windows\\system32\\python27.
zip', 'C:\\Python27\\ArcGIS10.3\\DLLs', 'C:\\Python27\\ArcGIS10.3\\lib',
'C:\\Python27\\ArcGIS10.3\\lib\\plat-win']
```

Notice that `sys.path` is a list of directory paths (only a few are shown here—your list may differ). These are the locations used by the Python interpreter when it encounters an import statement. The Python interpreter searches within these directories for the name of the module being imported. If the module isn't found in one of these directories, the import won't succeed.

The following code prints the file name of the script that was run most recently in PythonWin.

```
>>> sys.argv[0]
'C:\\gispy\\sample_scripts\\ch02\\describe_fc.py'
```

The `sys` module is used to get the workspace from user input on line 11 in Figure 2.3 (`arcpy.env.workspace = sys.argv[1]`). Chapter 7 discusses this useful `sys` module variable in more depth.

The `os` module allows you to access operating system-related methods (`os` stands for operating system). The following code uses the `os` module to print a list of the files in the 'C:/gispy/data/ch02' directory:

```
>>> import os
>>> os.listdir('C:/gispy/data/ch02')
['fires.dbf', 'fires.prj', 'fires.sbn', 'fires.sbx', 'fires.shp',
'fires.shp.xml', 'fires.shx', 'park.dbf', 'park.prj', 'park.sbn',
'park.sbx', 'park.shp', 'park.shp.xml', 'park.shx']
```

2.5 Key Terms

Integrated development environment (IDE)
Python script
PythonWin
PythonWin Interactive Window
PythonWin script window
PyScripter
Python prompt (>>>)
Python interpreter
Window focus
Script arguments
Context highlighting
Code comments
Hash sign
Python keywords
Variables
Assignment statement
Dynamic typing
Block structure, block of code
Dedented
Tracebacks
Built-in functions, constants, and exceptions
Function arguments
Function signatures
Function parameters
Exceptions thrown
`True`, `False`, `None`
`NameError`, `SyntaxError`, and `TypeError`
Module
Standard modules
`math`, `sys`, `os`, `shutil`

2.6 Exercises

1. Set up your integrated development environment (IDE) software and check the functionality as described here:

   (a) To install PythonWin,

   1. Browse 'C:\gispy\sample_scripts\ch02\programs\PythonWin'.
   2. Double-click on the executable ('exe') file.
   3. Launch PythonWin. A PythonWin desktop icon should be created during installation. If not, search the machine for 'PythonWin.exe' and create a shortcut to this executable. When you first launch PythonWin, it will display the 'Interactive Window' with a prompt that looks just like the one in the ArcGIS Python window as shown in Figure 2.2.
   4. Test PythonWin. Type `import arcpy` in the Interactive Window and press the 'Enter' key. If no error appears, everything is working. In other words, if you see only a prompt sign on the next line (>>>), then it worked. If, instead, you see red text, this is an error message. Check your ArcGIS software version. This book is designed for ArcGIS 10.1-10.3, which use the 2.7 version of Python. If you are using a different version of ArcGIS, you will need to get a different PythonWin executable. Start by searching online for 'pywin32 download', then navigate to the version you need.

ArcGIS version	Python version
10.1, 10.2, 10.3	2.7
10	2.6
9.3	2.5
9.2	2.4.1
9.1	2.1

   (b) Install PyScripter. To install PyScripter, browse to 'C:\gispy\sample_scripts\ch02\programs\PyScripter'. Double-click on the '.exe' file. Install using the defaults. PyScripter's equivalent of PythonWin's Interactive Window is the Python Interpreter window. Confirm that it is working correctly by typing `import arcpy` in the Python Interpreter window. Press the 'Enter' key. If you see only a prompt sign on the next line (>>>), then it worked. If, instead, you see red text, this is an error message. The ArcGIS Resources Python Forum is a good place for additional trouble-shooting.

2. Type the code statements below in the Interactive Window. Notice that the built-in `type` function gives two different results—first `str` and second `int`. Which

word or phrase in the 'Key terms' list at the end of this chapter explains this phenomenon?

```
>>> month = 'December'
>>> type(month)
>>> month = 12
>>> type(month)
```

3. Match each key term with the most closely related statement. There are two distracter statements that should not be used.

Key term	Statement
1. Python keyword	A. This is a built-in boolean constant.
2. IDE	B. If this isn't in the Script Window, you might save the Interactive Window by mistake.
3. Hash sign	C. Special font formatting (such as color) based on meaning of text in the code.
4. Tracebacks	D. The controversial debates surrounding Python versus Perl scripting languages.
5. Window focus	E. PythonWin shows **`print`** in blue because it is one of these.
6. Script arguments	F. If a variable name is misspelled, this can occur.
7. Context highlighting	G. A specialized editor for developing scripts.
8. Assignment statement	H. Information passed into a script by the user.
9. Dynamic typing	I. Automatically set the data type of a variable when a value is assigned.
10. False	J. `x = 5` is an example of one of these.
11. NameError	K. Messages printed when exceptions are thrown.
12. Dedented code	L. This signals the end of a related block of code.
13. `math`, `os`, and `sys`	M. The shape of a camel's back.
	N. Code following this character is only meant for the human reader.
	O. These are examples of Python standard modules.

4. Type the following lines of code in the Interactive Window. Each line of code should raise a built-in exception. Report the *name* of the built-in exception that is raised (The first row is done for you).

Python code	Exception name
`class = 'combustibles'`	`SyntaxError`
`'five' + 6`	
`Min`	
`int('five')`	
`5/0`	
`input file = 'park.shp'`	

5. **times.py** Write a script named 'times.py' that finds the product of two input integers and prints the results. Start by opening 'add_version2.py', modifying the last two lines and saving the file with the name 'times.py'. An asterisk is used as the multiplication operator in Python, so you'll use `c = a * b` to multiply. Also change the comment on line 1 to reflect the modified functionality. Run the script using two integer arguments to check that it works. The example below shows the expected behavior. Check your results against this example.

   Example input:
   2 3

   Example output:
   ```
   >>> The product is 6.
   ```