Exercise 4

Learning Python language fundamentals

Work with numbers

Python can be used as a powerful calculator. Practicing math calculations in Python will help you not only perform these tasks, but also show you how Python works with different types of numbers.

1 **Start ArcMap. On the Standard toolbar, click the Python button.**

2 **In the Python window, type the following code and press ENTER:**

__Note:__ Do not type the prompt (>>>), since it is already provided by the Interactive Window. Whenever sample code in this exercise is preceded by the prompt, it means the code should be entered in the Interactive Window.

```
>>> 12 + 17
```

This code should give you the result 29. All basic calculator functions work just as you would expect, with one exception, which follows.

3 **Run the following code:**

```
>>> 10 / 3
```

And the result is 3? What went wrong? The inputs to the calculation (10 and 3) are both integers, and therefore the result is, by default, also an integer. This results in rounding. If you want ordinary division, the solution is to use real numbers or floats—that is, numbers with decimals. If either one of the inputs in a division is a float, the result will also be a float.

4 Run the following code:

```
>>> 10.0 / 3.0
```

The result is 3.3333333333333335—notice that the very last number does not make sense because it has reached the limit of the number of decimal places Python uses.

***Note:** The exact number of decimal places depends on the version of Python you are using.*

Is there a similar upper limit to the size of integers? Yes, ordinary integers cannot be larger than 2147483647 or smaller than -2147483647. However, if you want larger values, you can use a long integer, commonly referred to as "long."

5 Run the following code:

```
>>> 12345678901
```

The result is 12345678901L, with the letter L indicating that Python converted the input value to a long integer.

Basic arithmetic operations such as addition, subtraction, multiplication, and division are relatively straightforward. Many more operations are possible, but take a look at just one more for now: the exponentiation, or power, operator (**).

6 Run the following code:

```
>>> 2 ** 5
```

And the result is 32.

Although you are not very likely to use Python directly as a calculator, the examples here show you how Python handles numbers, which will be useful as you start writing scripts.

Work with strings

Now you can take a look at strings. You have already seen a very simple example, which follows.

1 Run the following code:

```
>>> print "Hello World"
```

The code prints `Hello World` to the next line. This is called a *string*, as in a string of characters. Strings are values, just as numbers are.

Python considers single and double quotation marks the same, making it possible to use quotation marks within a string.

2 Run the following code:

```
>>> print 'Let's go!'
```

The code results in a syntax error because Python does not know how to distinguish the quotation marks that mark the beginning and end of the string from the quotation marks that are part of the string—in this case, in the word "Let's." The solution is to mix the type of quotation marks used, with single and double quotation marks.

3 Run the following code:

```
>>> print "Let's go!"
```

The code prints `Let's go!` to the next line.

Strings are often used in geoprocessing scripts to indicate path and file names, so you will see more examples of working with strings throughout the exercise.

Strings can be manipulated in a number of ways, as you will see next.

Note: *Quotation marks in Python are "straight up," and there is no difference between opening quotation marks and closing quotation marks, as is common in word processors. When you type quotation marks directly in Python, they are automatically formatted properly, but be careful when copying and pasting from other documents. Quotation marks in Python have to look like this (' ') or this (" "), not like this (‘ ’) or this (“ ”).*

4 Run the following code:

```
>>> z = "Alphabet Soup"
>>> print z[7]
```

The code returns the letter *t*, the seventh letter in the string where the letter A is located at index number 0. This system of numbering a sequence of characters in a string is called *indexing* and can be used to fetch any element within the string. The index number of the first element is 0.

5 Run the following code:

```
>>> print z[0]
```

The code returns the letter A. The index number of the last element depends on the length of the string itself. Instead of determining the length, negative index numbers can be used to count from the end backward.

6 Run the following code:

```
>>> print z[-1]
```

The code returns the letter *p*.

To fetch more than one element, you can use multiple index numbers. This is known as *slicing*.

7 Run the following code:

```
>>> print z[0:8]
```

The reference `z[0:8]` returns the characters with index numbers from 0 up to, but not including, 8, and therefore the result is Alphabet.

As you have seen, you can use an index to fetch an element. You can also search for an element to obtain its index.

8 Run the following code:

```
>>> name = "Geographic Information Systems"
>>> name.find ("Info")
```

The result is 11, the index of the letter I. In this example, `find` is a method that you can use on any string. Methods are explored later in this exercise.

Work with variables

All scripting and programming languages work with variables. A variable is basically a name that represents or refers to a value. Variables store temporary information that can be manipulated and changed throughout a script. Many programming languages require that variables be declared before they can be used. Declaring means that you first create a variable and specify what type of variable it is—and only then can you actually assign a value to that variable. In Python, you immediately assign a value to a variable (without declaring it), and from this value, Python then determines the nature of the variable. This typically saves a lot of code and is one reason why Python scripts are often much shorter than code in other programming languages.

Next, you can try a simple example using a numeric value.

1 Run the following code:

```
>>> x = 12
>>> print x
```

The value of 12 is now printed to the next line. The code line `x = 12` is called an *assignment*. The value of 12 is assigned to the variable *x*. Another way of putting this is to say that the variable *x* is bound to the value of 12. Implicitly, this particular line of code results in variable *x* being an integer, but there is no need to explicitly state this with extra code.

Once a value is assigned to a variable, you can use the variable in expressions, which you'll do next.

2 Run the following code:

```
>>> x = 12
>>> y = x / 4
>>> print y
```

The result is 3.

Variables can store many different types of data, including numbers (integers, longs, and floats), strings, lists, tuples, dictionaries, files, and many more. So far, you have seen only integers. Next, you can continue with strings.

3 Run the following code:

```
>>> k = 'This is a string'
>>> print k
```

Note: *Variable names can consist of letters, digits, and underscores (_). However, a variable name cannot begin with a digit.*

Work with lists

Lists are a versatile Python data type used to store a sequence of values. The values themselves can be numbers or strings.

1 Run the following code:

```
>>> w = ["Apple", "Banana", "Cantaloupe", "Durian", "Elderberry"]
>>> print w
```

This prints the contents of the list.

Lists can be manipulated using indexing and slicing techniques, very much like strings.

2 Run the following code:

```
>>> print w[0]
```

This returns `Apple` because the index number of the first element in the list is 0. You can use negative numbers for index positions on the right side of the list.

3 Run the following code:

```
>>> print w[-1]
```

This returns `Elderberry`.

Slicing methods using two index numbers can also be applied to lists, which you'll try next.

4 Run the following code:

```
>>> print w[2:-1]
```

The reference `w[2:-1]` returns the elements from index number 2 up to, but not including, -1, and therefore the result is `['Cantaloupe', 'Durian']`.

Notice the difference here between indexing and slicing. Indexing returns the value of the element, and slicing returns a new list. This is a subtle but important difference.

Use functions

A function is like a little program you can use to perform a specific action. Although you can create your own functions, Python has functions already built in, referred to as *standard functions*.

1 Run the following code:

```
>>> d = pow (2, 3)
>>> print d
```

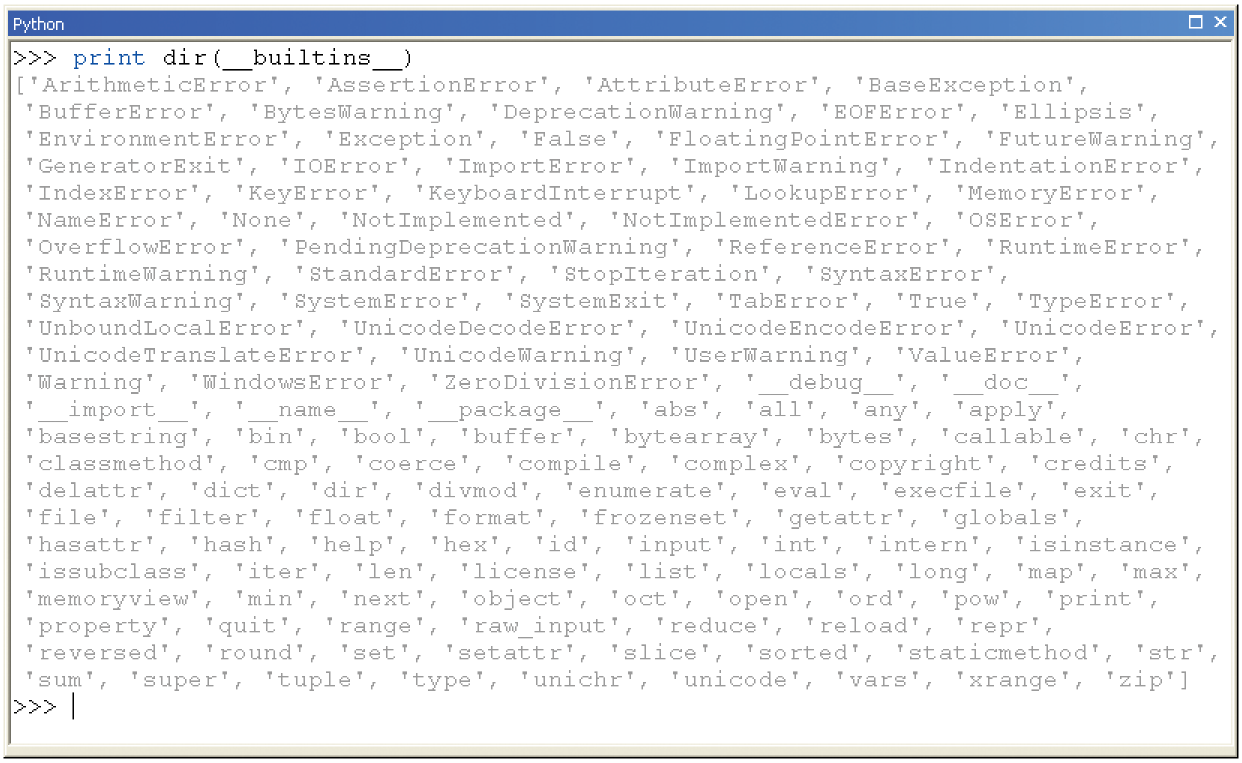
So instead of using the exponentiation operator (`**`), you can use a power function called `pow`. Using a function this way is referred to as *calling* the function. You supply the function with *parameters*, or *arguments* (in this case, 2 and 3), and it *returns* a value.

Numerous standard functions are available in Python. You can view the complete list by using the `dir(__builtins__)` statement.

***Note:** There are two underscores on either side of the word "builtins," not just one.*

2 Run the following code:

```
>>> print dir(__builtins__)
```



Python

```
>>> print dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning',
 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'WindowsError', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'apply',
 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit',
 'file', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals',
 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max',
 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
 'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr',
 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str',
 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
>>> |
```

It may not be immediately intuitive as to what many of these functions are used for, although some are straightforward. For example, `abs` returns the absolute value of the numeric value.

3 Run the following code:

```
>>> e = abs(-12.729)
>>> print e
```

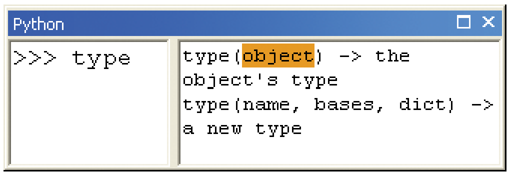
This returns the value of 12.729.

Since it may not be immediately clear how many of these functions work and what the parameters are, it should be helpful to take a look at the Help function next.

4 First, clean up the Python window by removing all the code so far. Right-click in the Python section of the window and click Clear All.

5 Make sure the Help and syntax panel is visible by dragging the divider in place.

6 Type the function `type` and don't press ENTER yet. Notice that the syntax appears in the adjacent panel. ➔



```
>>> type
```

```
type(object) -> the
object's type
type(name, bases, dict) ->
a new type
```

You can find similar descriptions in the Python manuals, but having it right where you are coding in the Python window is convenient. Notice that a section of the syntax is highlighted. It specifies the parameters of the function. When you call the function, you need to supply these parameters for the function to work, although some parameters are optional.

Next, you can try out this function.

7 Run the following code:

```
>>> type(123)
```

The result is `<type 'int'>`—that is, the input value is an integer.

8 Run the following code:

```
>>> type(1.23)
```

The result is `<type 'float'>`—that is, the input value is a float, or floating point.

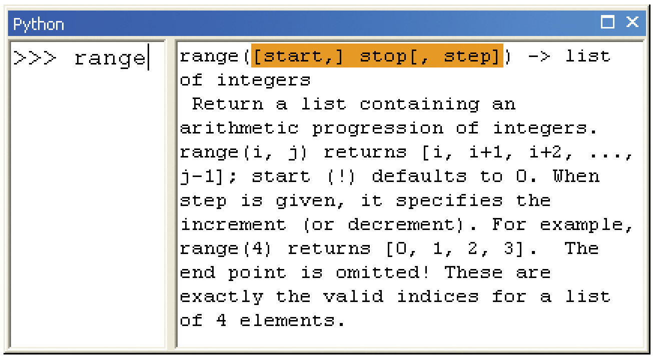
9 Run the following code:

```
>>> type("GIS")
```

The result is `<type 'str'>`—that is, the input value is a string.

Multiple parameters are separated by commas. Optional parameters are shown between square brackets ([ ]). For example, take a look at the function `range`.

10 In the Python window, type the function `range` and notice that the syntax Help appears in the adjacent panel.



```
>>> range
```

```
range([start,] stop[, step]) -> list
of integers
 Return a list containing an
arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ...,
j-1]; start (!) defaults to 0. When
step is given, it specifies the
increment (or decrement). For example,
range(4) returns [0, 1, 2, 3].  The
end point is omitted! These are
exactly the valid indices for a list
of 4 elements.
```

Notice the syntax: `range([start,] stop[, step])`. The function `range` has three parameters: `start`, `stop`, and `step`.

11 Run the following code:

```
>>> range(10, 21, 2)
```

The result is `[10, 12, 14, 16, 18, 20]`.

The function returns a list of integers from 10 to 20, with an increment of 2. However, the only required parameter is the endpoint (`stop`).

12 Run the following code:

```
>>> range(5)
```

The result is `[0, 1, 2, 3, 4]`.

The function returns a list of integers using the default values of 0 for the start parameter and 1 for the increment (`step`) parameter.

Use methods

Methods are similar to functions. A method is a function that is closely coupled with an object—for example, a number, a string, or a list. In general, a method is called as follows:

```
<object>.<method>(<arguments>)
```

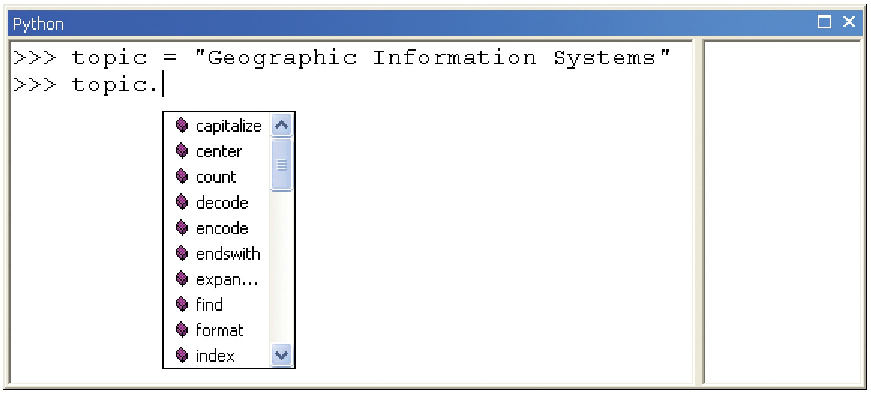
Calling a method looks just like calling a function, but now the object is placed before the method, with a dot (.) separating them. Next, take a look at a simple example.

1 Run the following code:

```
>>> topic = "Geographic Information Systems"
>>> topic.count("i")
```

The code returns the value of 2 because that is how often the letter *i* occurs in the input string.

A number of different methods are available for strings. Notice that when you start calling methods by typing a dot after the variable, a list of methods is provided for you to choose from. The syntax Help is also context sensitive.



```
>>> topic = "Geographic Information Systems"
>>> topic.
```

Next, try this out by using the `split` method.

2 Run the following code:

```
>>> topic.split(" ")
```

The result is a list of the individual words in the string:

```
['Geographic', 'Information', 'Systems']
```

Next, you can see how to apply the `split` method to work with paths. Say, for example, the path to a shapefile is c:\data\part1\final. How would you obtain just the last part of the path?

3 Run the following code:

```
>>> path = "c:/data/part1/final"
>>> pathlist = path.split("/")
>>> lastpath = pathlist[-1]
>>> print lastpath
```

The result is `final`.

So what happened exactly? In the first line of code, the path is assigned as a string to the variable `path`. In the second line of code, the string is split into four strings, which are assigned to the list variable `pathlist`. And in the third line of code, the last string in the list with index -1 is assigned to the string variable `lastpath`.

Methods are also available for other objects, such as lists.

4 Run the following code:

```
>>> mylist = ["A", "B", "C"]
>>> mylist.append("D")
>>> print mylist
```

The result is `['A', 'B', 'C', 'D']`.

Very few built-in methods are available for numbers, so in general, you can use the built-in functions of Python or import the `math` module (see next section) to work with numeric variables.

Use modules

Hundreds of additional functions are stored in modules. Before you can use a function, you have to import its module using the `import` function. The functions you used in the preceding sections are part of Python's built-in functions and don't need to be imported. One of the most common modules to import is the `math` module, so you'll start with that one.

1 Run the following code:

```
>>> import math
>>> h = math.floor (7.89)
>>> print h
```

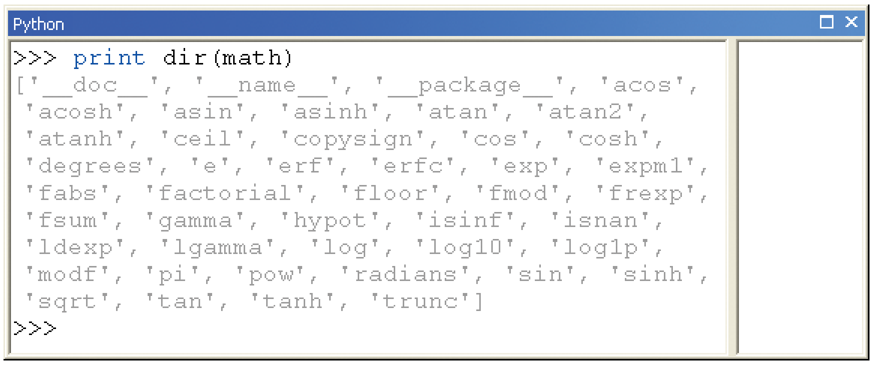
The result is 7.0.

Notice how the `math` module works: you import a module using `import`, and then use the functions from that module by writing `<module>.<function>`. Hence, you use `math.floor`. The `math.floor` function always rounds down, whereas the built-in `round` function rounds to the nearest integer.

You can obtain a list of all the functions in the `math` module using the `dir` statement.

2 Run the following code:

```
>>> print dir(math)
```

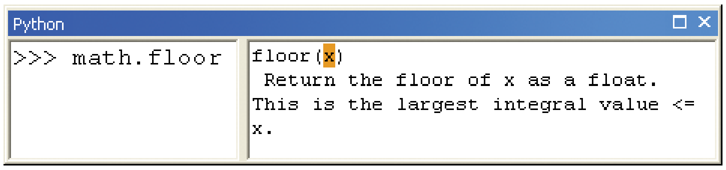


```
>>> print dir(math)
['__doc__', '__name__', '__package__', 'acos',
 'acosh', 'asin', 'asinh', 'atan', 'atan2',
 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

You can learn about each function in the Python manuals, but remember that you can also see the syntax in the Python window's Help and syntax panel.

3 Type the following code and do not press ENTER:

```
>>> math.floor
```



```
>>> math.floor
```

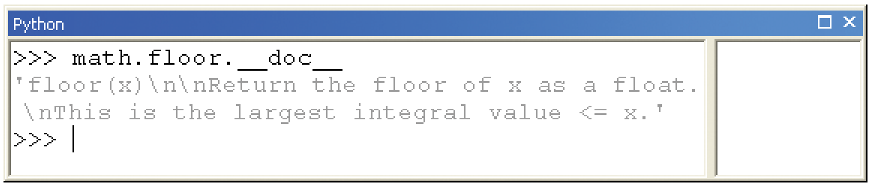
```
floor(x)
 Return the floor of x as a float.
This is the largest integral value <=
x.
```

Another way to see the documentation is to use the `__doc__` statement directly in Python.

4 Run the following code:

```
>>> print math.floor.__doc__
```

The result is a printout of the same syntax Help but now directly within the interactive Python interpreter.


Python

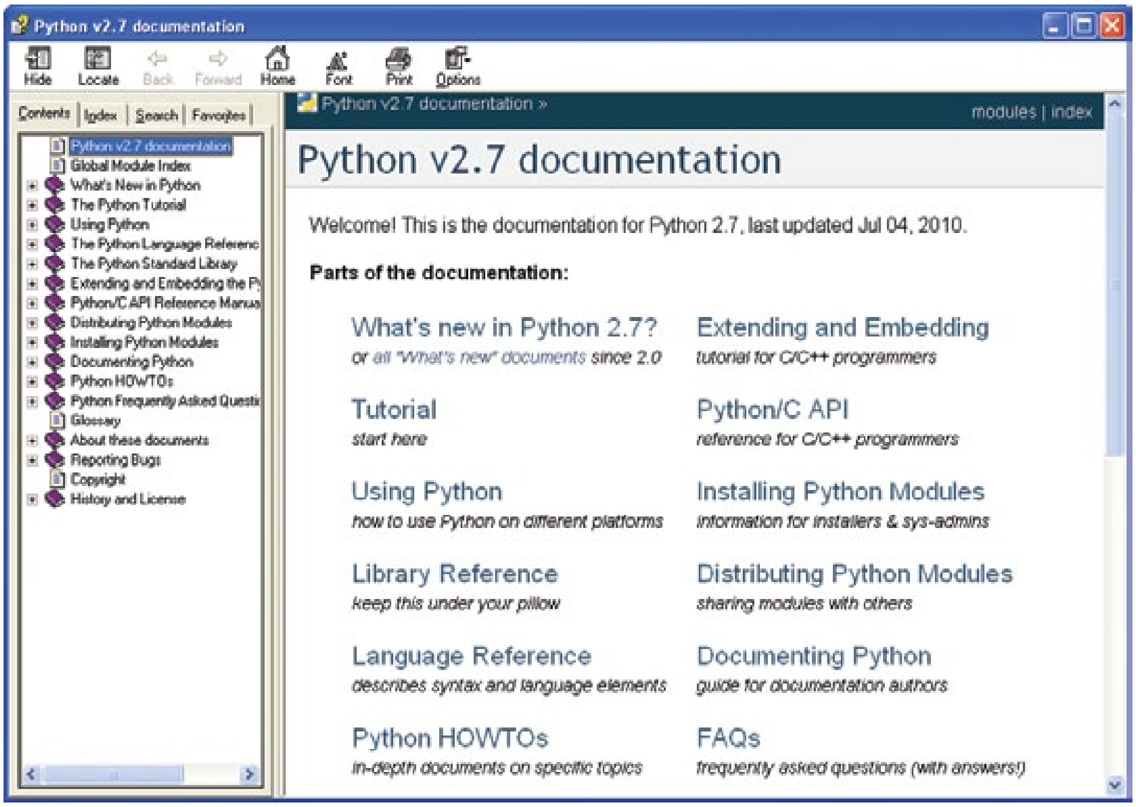
```
>>> math.floor.__doc__
'floor(x)\n\nReturn the floor of x as a float.
\nThis is the largest integral value <= x.'
>>> |
```

The syntax is `floor(x)`, which means the only parameter of this function is a single value. The function returns a float (such as 7.0), not an integer (such as 7). This information allows you to determine whether the function is really what you are looking for and how to use it correctly.

__Note:__ Remember that you need to add a prefix to any non-built-in functions using the name of the module, as in `math.floor(x)`, not just `floor(x)`. If you do not use a prefix, you will get an error stating that the name `floor` is not defined.

There are numerous modules available in Python. A complete list can be found in the Python manuals, which you'll look at next.

5 To access the Help documentation, on the taskbar, click the Start button, and then, on the Start menu, click All Programs > ArcGIS > Python 2.7 > Python Manuals.



Python v2.7 documentation
Global Module Index
What's New in Python
The Python Tutorial
Using Python
The Python Language Referenc
The Python Standard Library
Extending and Embedding the P
Python/C API Reference Manua
Distributing Python Modules
Installing Python Modules
Documenting Python
Python HOWTOs
Python Frequently Asked Questi
Glossary
About these documents
Reporting Bugs
Copyright
History and License

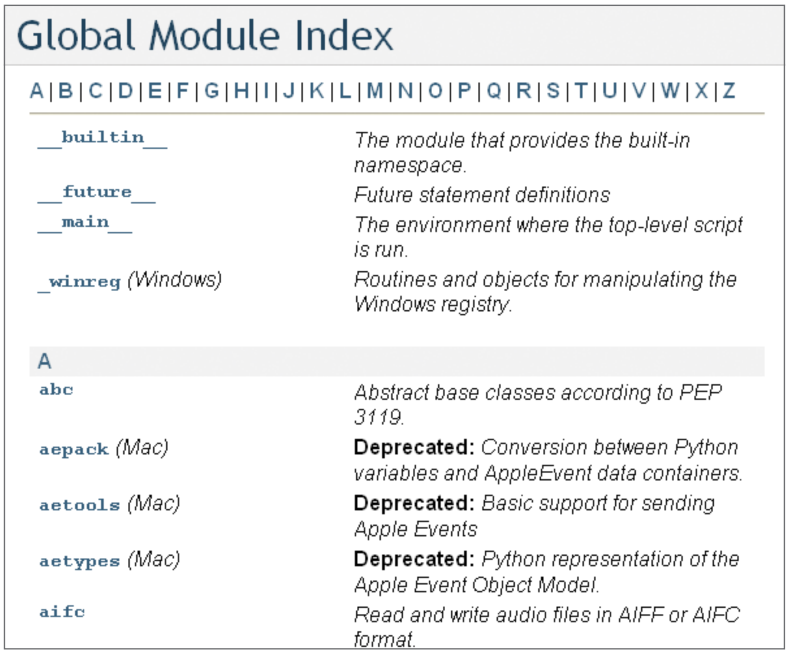
Python v2.7 documentation

Welcome! This is the documentation for Python 2.7, last updated Jul 04, 2010.

Parts of the documentation:

What's new in Python 2.7? *or all "What's new" documents since 2.0*	Extending and Embedding *tutorial for C/C++ programmers*
Tutorial *start here*	Python/C API *reference for C/C++ programmers*
Using Python *how to use Python on different platforms*	Installing Python Modules *information for installers & sys-admins*
Library Reference *keep this under your pillow*	Distributing Python Modules *sharing modules with others*
Language Reference *describes syntax and language elements*	Documenting Python *guide for documentation authors*
Python HOWTOs *in-depth documents on specific topics*	FAQs *frequently asked questions (with answers!)*

6 In the documentation table of contents, under "Indices and tables," click Global Module Index. The index provides an alphabetical list of all the available modules. →

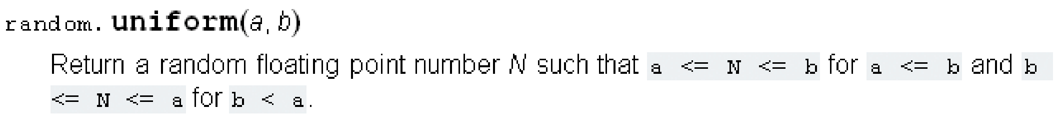


Global Module Index

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Z

`__builtin__`	*The module that provides the built-in namespace.*
`__future__`	*Future statement definitions*
`__main__`	*The environment where the top-level script is run.*
`_winreg` *(Windows)*	*Routines and objects for manipulating the Windows registry.*
A	
`abc`	*Abstract base classes according to PEP 3119.*
`aepack` *(Mac)*	**Deprecated:** *Conversion between Python variables and AppleEvent data containers.*
`aetools` *(Mac)*	**Deprecated:** *Basic support for sending Apple Events*
`aetypes` *(Mac)*	**Deprecated:** *Python representation of the Apple Event Object Model.*
`aifc`	*Read and write audio files in AIFF or AIFC format.*

There are many specialized modules, and in a typical Python script, you may use several. Try taking a look at just one more. For example, scroll down to the `random` module and click the link. Scroll down to the `uniform` function and read the description.



`random.`**`uniform`**`(`*a*, *b*`)`

Return a random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

Notice that the `uniform` function has two required parameters, a and b. You will try this function next in Python.

7 **Close the documentation and return to the Python window.**

8 **Run the following code:**

```
>>> import random
>>> j = random.uniform(0, 100)
>>> print j
```

The result is a float from 0 to 100.

9 **Close ArcMap. There is no need to save your map document.**

Save Python code as scripts

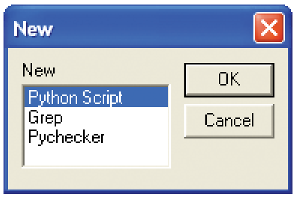
So far in this exercise, you have only worked from within the Python window, or interactive Python interpreter. This works great to practice writing Python code and to run relatively simple code. However, once code gets a bit more complex, you'll typically want to save your work to a Python script. Although you can save your code from the Python window to a script file, you will first practice creating, writing, and saving scripts using the PythonWin editor.

1 **On the taskbar, click the Start button, and then, on the Start menu, click Python 2.7 > PythonWin.** Notice that, by default, PythonWin opens with an Interactive Window, which works very much like the Python window in ArcMap.

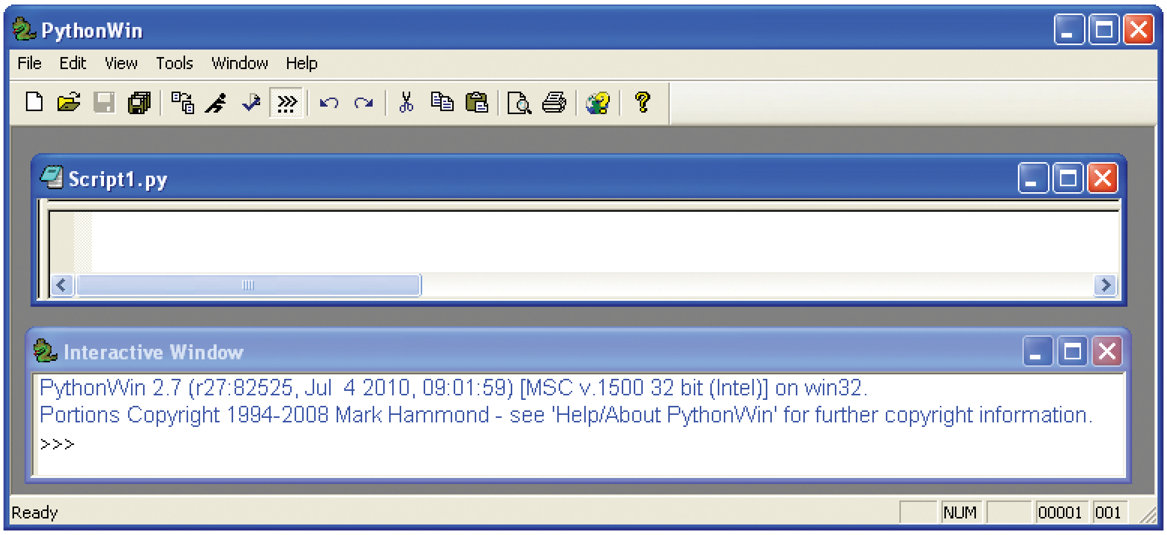
Next, you will create a new script window.

2 On the PythonWin Standard toolbar, click the New button. On the New dialog box, click Python Script and click OK.

Note: Using the New button in PythonWin is the same as using File > New from the menu bar.



3 Rearrange the windows so that both the Interactive Window and the new script window are visible.



```
PythonWin 2.7 (r27:82525, Jul  4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on win32.
Portions Copyright 1994-2008 Mark Hammond - see 'Help/About PythonWin' for further copyright information.
>>>
```

Next, you will enter the same code you worked with in the Python window.

4 In the script window, type the following code:

```
path = "c:/data/part1/final"
pathlist = path.split("/")
lastpath = pathlist[-1]
print lastpath
```

Notice that the lines of code in the script window are not preceded by the prompts (>>>) found in the Interactive Window. Also notice that nothing is printed when you press ENTER—the cursor simply jumps to the next line as in a text editor. What this means is that in the script window, the Python code is not actually executed until you run it.

5 **First, save the script. On the PythonWin Standard toolbar, click the Save button 💾. On the Save As dialog box, navigate to the C:\EsriPress\Python\Data\Exercise04\Results folder and save your file as** path.py**.** The extension .py indicates the file is a Python script.

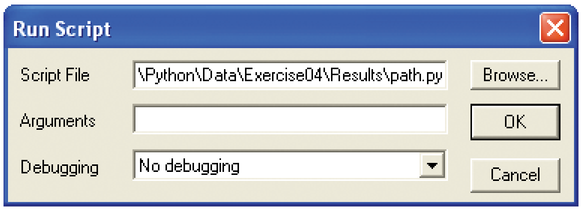
Note: *Using the Save button in PythonWin is the same as using File > Save from the menu bar.*

6 **With your cursor still placed anywhere within the code in the path.py script window, click the Run button on the PythonWin Standard toolbar.**

>>> TIP

To run a script, you can also click File > Run on the menu bar or press CTRL+R.

7 **This brings up the Run Script dialog box. For now, leave the default settings and click OK.**



Script File: \Python\Data\Exercise04\Results\path.py
Arguments:
Debugging: No debugging

The result `final` is printed to the Interactive Window. Notice that the syntax for Python code in the script window is the same as in the interactive interpreter. The main difference is that the script window allows you to write and save scripts without the code being run. In a typical workflow, you may use both the interactive interpreter, such as the Python window in ArcGIS or the Interactive Window in PythonWin, and the script window in a Python editor, such as PythonWin. Later exercises show examples of using both in a single workflow.

8 **Close the path.py script and leave PythonWin open.**

Write conditional statements

The scripts you have worked with so far use a sequential flow. In many cases, you'll want to selectively run certain portions of your code instead. That's where branching and looping statements come in.

1 **On the PythonWin Standard toolbar, click the New button and confirm that you want a new script. Then save as** branching.py **to the Results folder for exercise 4.**

2 Write the following code to generate a random number between 1 and 6:

```
import random
p = random.randint(1, 6)
print p
```

3 Run the script to confirm that it works correctly.

Next, you will add an `if` structure to run code based on the value of *p*.

4 Replace the line `print p` with the following:

```
if p == 6:
```

The code `p == 6` is an example of a condition—the answer is either `True` or `False`. If the answer is true, the code following the `if` statement runs. If the answer is false, there is no code left to run, and the script simply ends.

A few things to remember about the `if` structure: First, the `if` statement ends with a colon (:). Second, the lines following the `if` statement are indented. When you indent a line, the code becomes a *block*. A block consists of one or more consecutive lines of code that have the same indentation.

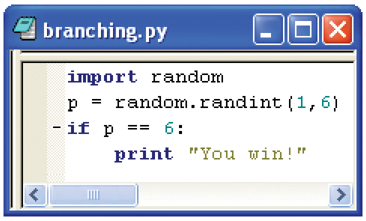
PythonWin assists with automatic indentation. When you press ENTER following the `if` statement colon, PythonWin automatically indents the next line of code.

5 Write the following line of code following the `if` statement:

```
print "You win!"
```

Your script should now look like the example in the figure.

Note: Indentation is required in Python. You can use tabs or spaces to create indentation—the style you pick is partly a matter of preference, but you should be consistent. Using either two spaces or four spaces is most common. By default, Python uses four spaces for indentation and also converts a tab to four spaces.



```
import random
p = random.randint(1,6)
if p == 6:
    print "You win!"
```

6 Run the script. Running the script may result in a print statement in the Interactive Window, or nothing at all, depending on your value for *p*.

Notice that the if structure, in this case, is not followed by anything else. If you are familiar with other programming languages, you may have expected something to follow, such as "else" or "end". In Python, the if structure can be used on its own or expanded by follow-up statements.

7 **In the branching.py script, place your pointer at the end of the line of code that reads `print "You win!"` and press ENTER.** Notice that the next line of code is automatically indented under the assumption you are continuing your block of code. However, in this case, you want to continue with an else statement, and the indentation needs to be removed.

8 **Press BACKSPACE to remove the indentation.**

9 **For the next lines of code, enter the following:**

```
else:
    print "You lose!"
```

Notice again the automatic indentation following the else statement. Now, your code is ready to handle both a true and a false condition.

>>> TIP

Correct indentation is key here. In this example, the if and the else statements should line up, and the two print statements should also line up.

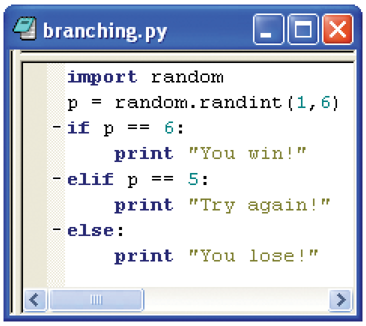
10 **Save and run the script.** By using the if-else structure, you account for all possible outcomes and the script prints a value to the screen every time you run it, not just when the if statement is True.

One more variant on this is the if-elif-else structure, which you'll use next.

11 **Insert a line above the else statement and enter the following code:**

```
elif p == 5:
    print "Try again!"
```

Your code should now look like the example in the figure.



```
import random
p = random.randint(1,6)
if p == 6:
    print "You win!"
elif p == 5:
    print "Try again!"
else:
    print "You lose!"
```

12 Run the script a few times until the results include all three conditions. The `elif` statement is evaluated only if the `if` statement is `False`. You can use `elif` multiple times, so in principle you could specify an action for every unique possible value of the variable *p*. Like the `if` statement, the `elif` statement does not need an `else` statement to follow. You do, however, need to start this type of branching structure with an `if` statement—that is, you can't use `elif` or `else` without first using an `if` statement. Also notice that all three statements end with a colon (:) and that there is no "end" statement as there is in some programming languages.

13 Save your branching.py script and close it.

Use loop structures

There are other structures to control workflow, including the `while` loop and `for` loop structures.

1 On the PythonWin Standard toolbar, click the New button and confirm that you want a new script. Then save as whileloop.py **to the Results folder for exercise 4.**

2 Write the following code:

```
i = 0
while i <= 10:
    print i
    i += 1
```

***Note:** The syntax uses the plus-equal symbol (+=), which adds a specified amount to the input value. This could also be written as* `i = i + 1`*.*

3 Run the script. The result is a print of the numbers 0 to 10. With each iteration over the `while` loop, the value of the variable `i` is increased by 1. The variable `i` is referred to as a *counter*. The `while` loop keeps going until the condition becomes false—that is, when the counter reaches the value of 11.

The `while` loop structure uses a syntax similar to the `if` structure: the `while` statement ends with a colon (:), and the next line of code is indented to create a block.

4 Save and close your whileloop.py script.

Next, you can try a `for` loop.

5 Create a new Python script and save as forloop.py **to the Results folder for exercise 4.**

6 Write the following code:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print number
```

7 Run the script. The block of code is run for each element in the list.

8 Save and close your forloop.py script.

Looping, or iterating, over a range of numbers is a common task, and Python has a built-in function to create ranges. The `range` function has three arguments: `start`, `stop`, and `step`. The `range` function generates a list of integers, beginning with `start` and up to, but not including, `stop`, using an increment of size `step`. For example, the next code prints the numbers 0 to 100, with a step of 10.

9 Create a new Python script and save as range.py **to the Results folder for exercise 4.**

10 Write the following code:

```
for number in range(0, 101, 10):
    print number
```

11 Run the script. Iterating using a `for` loop is much more compact than using the `while` loop.

12 Save and close your range.py script.

Usually, iterating over a loop simply runs a block of code until it has used up all the sequence elements. Sometimes, however, you may want to interrupt a loop to start a new iteration or to end the loop. You can use the `break` statement to accomplish this, which you'll do next.

13 Create a new Python script and save as breakloop.py **to the Results folder for exercise 4.**

14 Write the following code:

```
from math import sqrt
for i in range(1001, 0, -1):
    root = sqrt(i)
    if root == int(root):
        print i
        break
```

15 **Run the script.** The code determines the largest square below 1,000. A range of integers is created, starting at 1,000 and counting down to zero (0). The negative step is used to iterate downward. When the square root of the integer is identical to the integer of the square root, you have the solution, and there is no need to continue. The solution is printed, and the loop ends.

Comment scripts

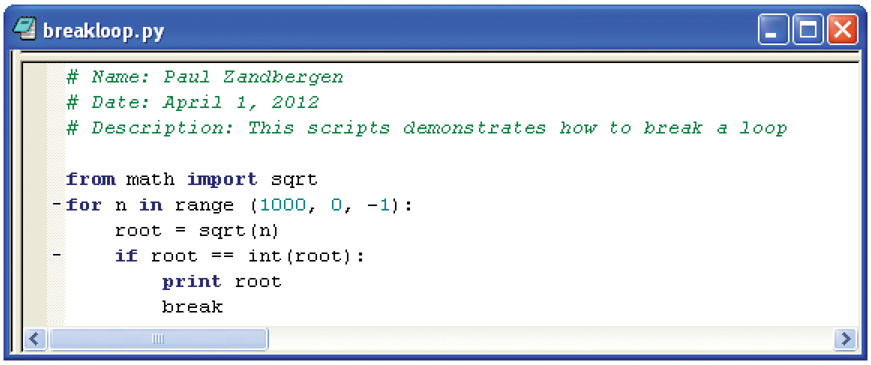
Well-developed scripts include comments that provide documentation about the script. Typically, the first few lines of a script consist of comments, but comments also occur throughout a script to explain how the script works. Comments are not executed when the script is run. In Python, a comment is preceded by the number sign (#). Any text that comes after the number sign is ignored during the execution of the script.

Next, you will add some comments that could prove useful in almost any script you write.

1 **In the breakloop.py script, place your pointer at the very beginning of the code and press ENTER. At the top of the script, type the following code:**

```
# Name: <your name>
# Date: <current date>
# Description: This script demonstrates how to break a loop
```

Your script window should now look like the example in the figure. Notice that the PythonWin editor recognizes comments and shows them in green italics.



```
# Name: Paul Zandbergen
# Date: April 1, 2012
# Description: This scripts demonstrates how to break a loop

from math import sqrt
for n in range (1000, 0, -1):
    root = sqrt(n)
    if root == int(root):
        print root
        break
```

Adding a comment just before a particular line or block of code can help other users understand it, as well as serve as a personal reminder about the code's meaning.

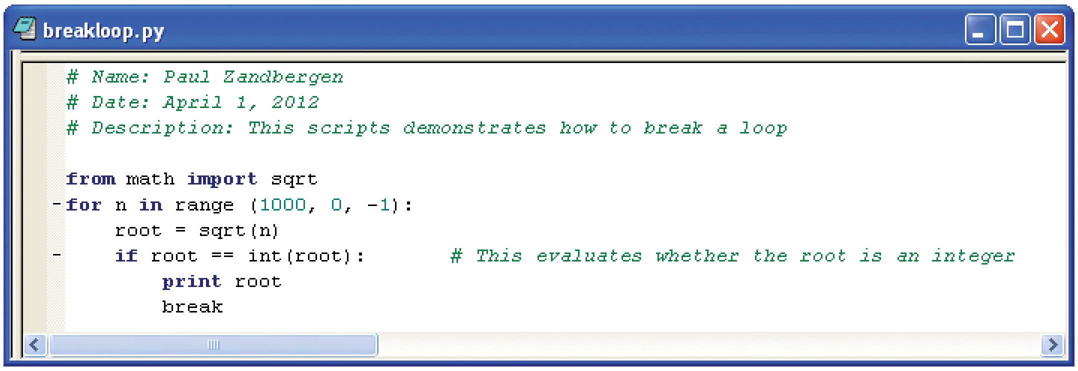
>>> TIP

Adding a line of space in your code is optional and has no effect on running the script. Typically, lines of space are added for readability. For example, it is common to add a line before or after comments or to keep lines of related code separate from other sections. This becomes more important as your scripts get longer.

2 After the `if root == int(root)` **code, enter a few tabs and then the code:**

```
# This evaluates when the root is an integer
```

Inserting comments allows you to enter specific comments to explain very specific parts of your code.



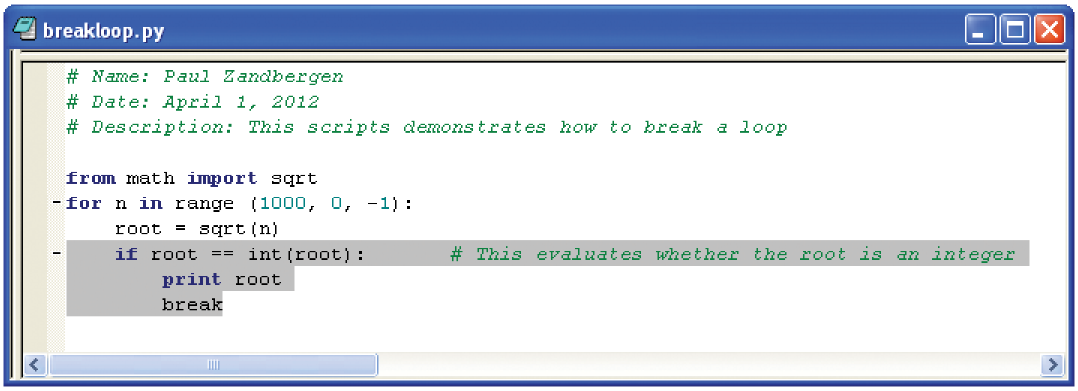
breakloop.py

```
# Name: Paul Zandbergen
# Date: April 1, 2012
# Description: This scripts demonstrates how to break a loop

from math import sqrt
for n in range (1000, 0, -1):
    root = sqrt(n)
    if root == int(root):       # This evaluates whether the root is an integer
        print root
        break
```

A related technique is *commenting out* several lines of code all at once. Say, for example, you have written some code and you've tested it. Now you want to try another approach without having to delete the code you already have.

3 In the breakloop.py script, highlight the last three lines of code.

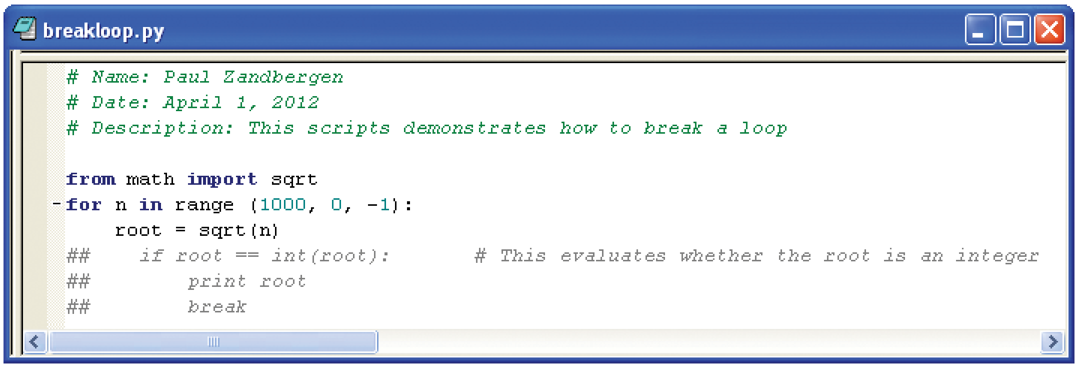


breakloop.py

```
# Name: Paul Zandbergen
# Date: April 1, 2012
# Description: This scripts demonstrates how to break a loop

from math import sqrt
for n in range (1000, 0, -1):
    root = sqrt(n)
    if root == int(root):       # This evaluates whether the root is an integer
        print root
        break
```

4 Right-click the highlighted section of code and click Source code > Comment out region. Notice that this places double number signs (##) in front of the lines of code (you could do this manually as well, but it is much faster to comment them out all at one time). Now these lines of code are skipped when the script is run. To make the code active again, highlight the commented lines, right-click, and click Source code > Uncomment region.



```
# Name: Paul Zandbergen
# Date: April 1, 2012
# Description: This scripts demonstrates how to break a loop

from math import sqrt
for n in range (1000, 0, -1):
    root = sqrt(n)
##    if root == int(root):       # This evaluates whether the root is an integer
##        print root
##        break
```

5 Save your script and close PythonWin.

Check for errors

It is relatively easy to make small mistakes in your Python code as a result of spelling and other syntax errors. Next, you can take a look at some simple ways to identify and correct errors. More advanced techniques are covered in chapter 11.

Start with some simple syntax errors.

1 Start ArcMap and open the Python window. Run the following code, in which "print" is intentionally misspelled:

```
>>> pint "Hello World!"
```

There is a typo in the print statement, and the result is a syntax error:

```
Parsing error SyntaxError: invalid syntax
(line 1)
```

Notice that the error message indicates both the type of error (parsing error) and where it occurred (line 1). You can try to minimize typos by using the prompts provided as you start typing.

2 Try a small variation of your print statement by running the following code:

```
>>> print "Hello World!
```

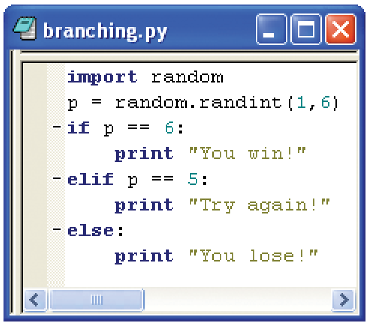
There are no closing quotation marks at the end of the line of code, resulting in a syntax error:

```
Parsing error SyntaxError: EOL while scanning
string literal (line 1)
```

Notice that the error message provides specific details on the nature of the error. EOL stands for End of Line, so you know where to look to correct the error. There are a lot of different types of error messages—too many to worry about at this point, but the basic idea is that error messages provide information on both the nature of the error and where the error occurs. Next, you will look at how this works for scripts consisting of multiple lines of code.

3 Close ArcMap. There is no need to save your map document.

4 Start PythonWin and open your branching.py script from earlier in the exercise. It should look like the example in the figure.



```
import random
p = random.randint(1,6)
if p == 6:
    print "You win!"
elif p == 5:
    print "Try again!"
else:
    print "You lose!"
```

Next, try adding a small error.

5 Place your pointer at the end of the third line of code and remove the colon (:) at the end of the `if` statement:

```
if p == 6
```

6 Place your pointer anywhere in the first line of code and click the Check button ✓. This checks the syntax of the code without running it. Notice that the cursor is placed at the end of the third line of code where the syntax error is located, and the status bar at the bottom of the PythonWin interface reads, "Failed to check – syntax error – invalid syntax." The Check option provides a quick way to test the syntax prior to running your script.

7 Now try running the script. In this case, running the script returns the same result: the cursor is placed at the end of the third line of code, and the status bar at the bottom indicates a syntax error. For more complicated scripts, it is useful to first identify and remove syntax errors using the Check option, and then run the script to see if there are other errors.

8 Correct the syntax error by placing a colon (:) at the end of the `if` statement:

```
if p == 6:
```

9 Now introduce a different error by placing a typo in the `randint` function:

```
p = random.randinr(0, 6)
```

10 Check the syntax of your script by clicking the Check button. Clicking Check confirms that there are no syntax errors in your script. Syntax checking examines the statements and expressions in your code but does not check for other errors, such as naming a function incorrectly. Only when you run the code will you discover that the function `randinr` does not exist.

11 Try running your script. Notice a fairly lengthy error message that is printed to the Interactive Window. The last three lines read as follows:

```
File "......\branching.py", line 2, in <module>
p = random.randinr(0,6)
AttributeError: 'module' object has no attribute 'randinr'
```

The error message provides a clear indication of where the error is located (line 2 of the code in which you are working with a module) and what the error is (`randinr` is not a function of this module).

12 Close PythonWin. There is no need to save the changes to your script.

Challenge exercises

Challenge 1

Create a script that examines a string for the occurrence of a particular letter, as you did previously in this exercise for "Geographic Information Systems." If the letter occurs in the text (for example, the letter Z), the string "Yes" should be printed to the Interactive Window. If the letter does not occur in the text, the string "No" should be printed.

Challenge 2

Create a script that examines a list of numbers without duplicates (for example, 2, 8, 64, 16, 32, 4) and determines the second-largest number.

Challenge 3

Create a script that examines a list of numbers (for example, 2, 8, 64, 16, 32, 4, 16, 8) to determine whether it contains duplicates. The script should print a meaningful result, such as "The list provided contains duplicate values" or "The list provided does not contain duplicate values."

An optional addition is to remove the duplicates from the list.

Challenge 4

Consider the following list:

```
mylist = ["Athens", "Barcelona", "Cairo", "Florence", "Helsinki"]
```

Determine the results of the following:

a) `len(mylist)`
b) `mylist[2]`
c) `mylist[1:]`
d) `mylist[-1]`
e) `mylist.index("Cairo")`
f) `mylist.pop(1)`
g) `mylist.sort(reverse = True)`
h) `mylist.append("Berlin")`

These operations are all to be performed on the original list—that is, not as a sequence of operations. Try to determine the answer manually first, and then check your result by running the code.