Exercise 4

Learning Python language fundamentals

Exercise data

Exercise 4 does not require any data to be downloaded.

Work with numbers

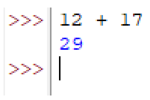
Python can be used as a powerful calculator. Practicing math calculations in Python will help you not only perform these tasks, but also show you how Python works with different types of numbers. The following instructions use IDLE as the Python editor, but you can use the same code in another editor as well as the Python window in ArcGIS Pro.

1. **Start IDLE.**

2. **In the interactive interpreter (i.e., the Python Shell), type the following code and press Enter:**

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

The result 29 prints to the next line.



```
>>> 12 + 17
29
>>>
```

Note: Do not type the prompt (`>>>`) because it is already provided by the interactive interpreter. Whenever sample code in this exercise is preceded by the prompt, the code should be entered in the interactive interpreter. PyCharm uses a prompt that looks like this: `In [1]:`. Although this prompt looks different, it works the same as the prompt in IDLE. The Python window does not show a prompt but can be used as well.

All basic calculator functions work as you would expect, but there are some issues to be aware of.

3. **Run the following code:**

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

The result is 3.3333333333333335. Notice that the last number, 5, does not make sense because it has reached the limit of the number of decimal places Python uses.

4. **Run the following code:**

```
>>> 10 / 5
```

The result is 2.0 instead of 2. The result of a division is always a float, even if both inputs are integers and the result is a whole number.

Basic arithmetic operations such as addition, subtraction, multiplication, and division are relatively straightforward. Many more operations are possible. Consider the exponentiation, or power, operator (`**`).

5. **Run the following code:**

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

The result is 32.

Also consider the floor division operators (//).

6. **Run the following code:**

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

The result is 3. Floor division is similar to regular division, but the result is an integer and the decimals are removed.

Although you are not 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

Next, you will look at strings. You have previously seen a simple example, which follows.

1. **Run the following code:**

```
>>> print("Hello World")
```

The code prints `Hello World` to the next line. This code 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 itself. 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.

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 a Python editor, they are automatically formatted properly, but be careful when copying and pasting from other documents. Quotation marks in Python must look like this (`'  '`) or this (`"  "`), not like this (‘ ’) or this (“ ”).

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

4. **Run the following code:**

```
>>> z = "Alphabet Soup"

>>> print(z[7])
```

The code returns the letter *t* that has index number 7 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 obtain 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 of the string backward.

6. **Run the following code:**

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

The code returns the letter *p* from `soup`.

To obtain 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 number to fetch an element. You can also search for an element to obtain its index number.

8. **Run the following code:**

```
>>> name = "Geographic Information Systems"

>>> name.find ("Info")
```

The result is 11, the index number of the letter *I*. Note that spaces are also counted as characters. 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 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 feature typically saves a lot of code and is one reason why Python scripts are often much shorter than code in other programming languages.

Next, try a simple example using a numeric value.

1. **Run the following code:**

```
>>> x = 12

>>> print(x)
```

The value of 12 is printed to the next line. The code line `x = 12` is called an assignment. The value of 12 is assigned to the variable x. Or you could say that the variable x is bound to the value of 12. Implicitly, this line of code results in variable x being an integer, but there is no need to explicitly state it 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 3.0.

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

3. **Run the following code:**

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

>>> print(k)
```

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

Work with lists

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

1. **Run the following code:**

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

>>> print(w)
```

Running the code prints the contents of the list.

Lists can be manipulated using indexing and slicing techniques, the same as 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 `Durian`.

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

4. **Run the following code:**

```
>>> print(w[1:-1])
```

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

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

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)
```

The result is 8.

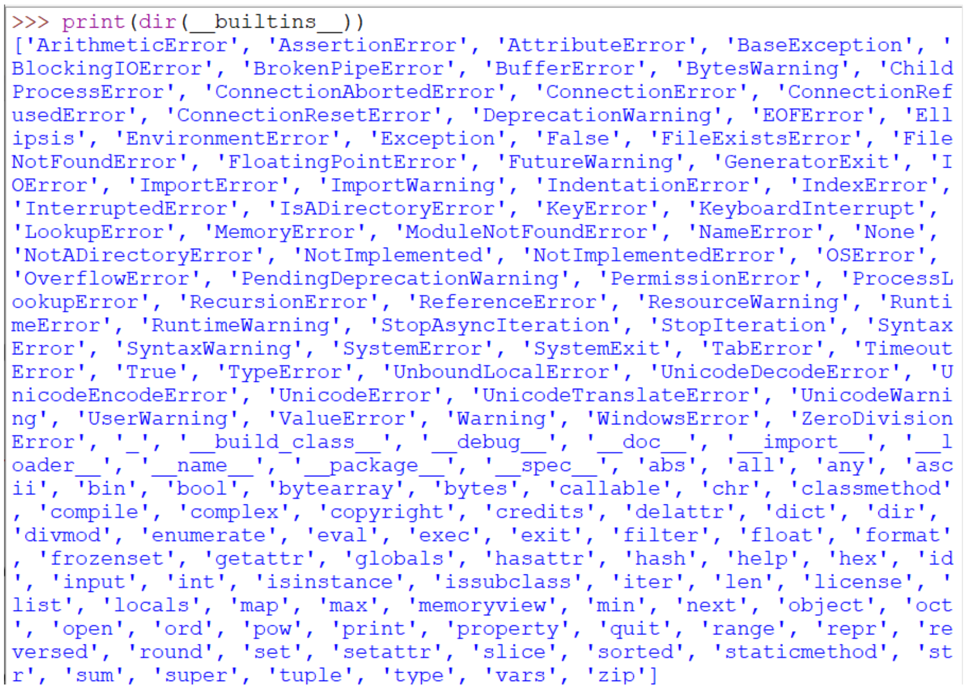
Instead of using the exponentiation operator (`**`), you can use a power function called `pow()`. Using a function this way is referred to as *calling the function*. You supply the function with parameters, or arguments (in this case, 2 and 3), and it returns a value.

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

2. Run the following code:

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

Note that there are two underscores on either side of the word "builtins," not just one. The result prints all the built-in identifiers of Python, not just the functions. The built-in functions start at `abs()` and go until `zip()`.



```
>>> print(dir(__builtins__))
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', '
BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'Child
ProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRef
usedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ell
ipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'File
NotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'I
OError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessL
ookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'Runti
meError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'Syntax
Error', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeout
Error', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'U
nicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarni
ng', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivision
Error', '_', '__build_class__', '__debug__', '__doc__', '__import__', '__l
oader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'asc
ii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod'
, 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format'
, 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id
', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', '
list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct
', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 're
versed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'st
r', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

It may not be immediately intuitive what many of these functions can be used for, but 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)
```

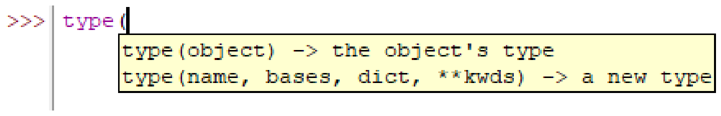
This returns the value of 12.729.

For many other functions, however, it may not be immediately clear how they work and what their parameters are. So next, use the built-in syntax help.

Note: In many of the figures in this exercise, the results from previous lines of code are not shown unless they are relevant to the example. They are not necessary for the code to work, and you can continue to work in the interactive interpreter without restarting.

4. **Enter the following code but stop after the opening parentheses.**

```
>>> type(
```



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

Notice that the syntax appears as a pop-up. In this case, the only argument is an object, and the function returns the type of the object. You can find similar descriptions in the Python manuals but having it right where you are coding is convenient.

Next, you can try out this function.

5. **Run the following code:**

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

The result is `<class 'int'>`, which means that the input value is an integer.

6. **Run the following code:**

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

The result is `<class 'float'>`, which means that the input value is a float, or floating-point number.

7. **Run the following code:**

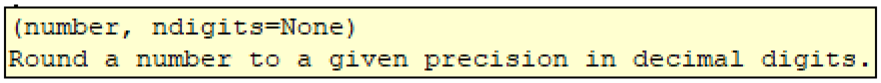
```
>>> type("GIS")
```

The result is `<class 'str'>`, which means that the input value is a string.

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

8. **Enter the following code but stop after the opening parentheses.**

```
>>> round(
```



```
(number, ndigits=None)
Round a number to a given precision in decimal digits.
```

Notice the syntax: `round(number, ndigits = None`. The function has two parameters: `number` and `ndigits`. The `number` parameter is required whereas the `ndigits` parameter is optional.

9. Run the following code:

```
>>> round(1.234567, 4)
```

The result is 1.2346.

The function rounds the number to the specified number of decimals. However, the only required parameter is the number itself.

10. Run the following code:

```
>>> round(1.234567)
```

The result is 1.

The function rounds the number, and if the number of decimals is not specified, the function returns an integer. In effect, the default value for the parameter ndigits is zero (0).

Use methods

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

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

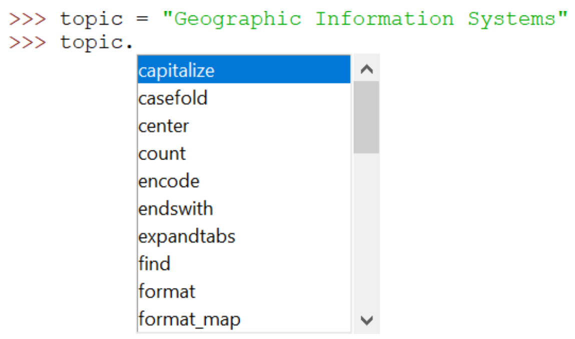
Calling a method looks like calling a function, but now the object is placed before the method, with a dot separating them. Next, consider a simple example.

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 character *i* occurs in the input string. The uppercase *I* is not counted.

Several 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.



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

Note: The autocompletion prompts are sometimes a bit slow to pop up, so be patient. When the prompt disappears too quickly, remove the dot, type the dot again, and wait a second.

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. Suppose, for example, that 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`.

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 must import its module using the `import` statement. 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)
```

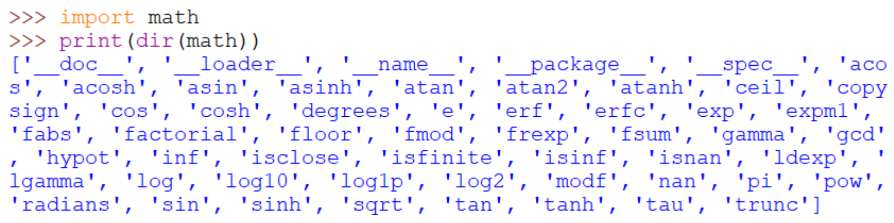
The result is 7.

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()` function.

2. **Run the following code:**

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



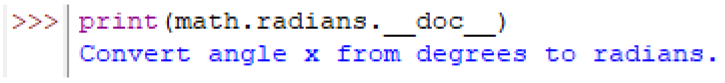
```
>>> import math
>>> print(dir(math))
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'aco
s', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copy
sign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd'
, 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', '
lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

You can learn about each function in the Python manuals but remember that you can also see the syntax when you are typing the code. Another way to see the documentation is to use `__doc__` directly in Python.

3. **Run the following code:**

```
>>> print(math.radians.__doc__)
```

The result is a printout of the syntax with a brief explanation directly within the interactive Python interpreter.



```
>>> print(math.radians.__doc__)
    Convert angle x from degrees to radians.
```

The syntax is `radians(x)`, which means the only parameter of this function is a single value. The function converts an angle from degrees to radians. This information allows you to determine whether the function is what you are looking for and how to use it correctly.

Note: Remember that you must reference the module before using any non-built-in functions, as in `math.radians(x)`, not `radians(x)`. If you do not reference the module, you will get an error: `NameError: name 'radians' is not defined`.

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

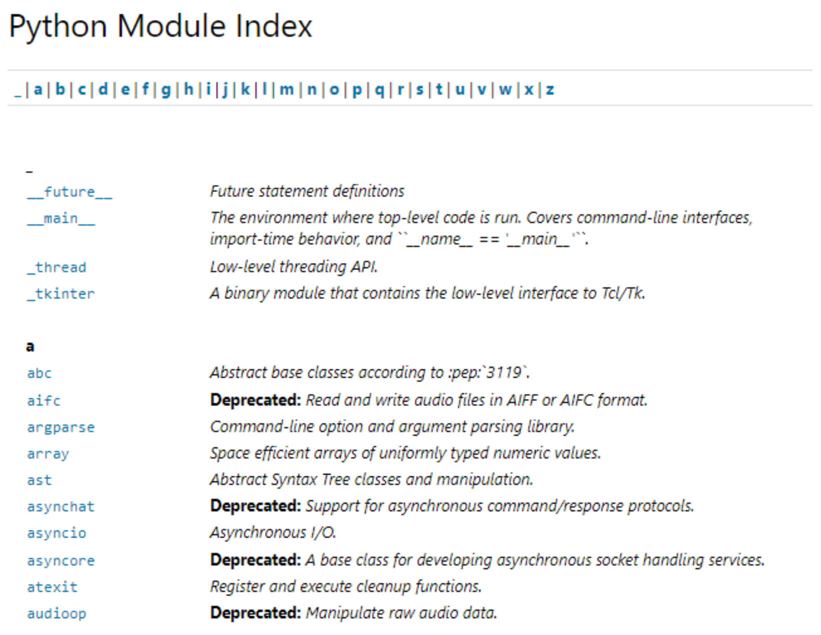
4. **In IDLE, click Help > Python Docs.**

The documentation appears in your default browser. You can also navigate directly to docs.python.org/3.11/.

Note: The latest version of Python at the time of writing is 3.11, with 3.13 in development. ArcGIS Pro works with Python 3.11, so that is the documentation you should look at. However, the changes between these versions are relatively minor, so the version of the online documentation you consult is not critical.

5. **In the documentation table of contents, under Indices, Glossary, and Search, click Global Module Index.**

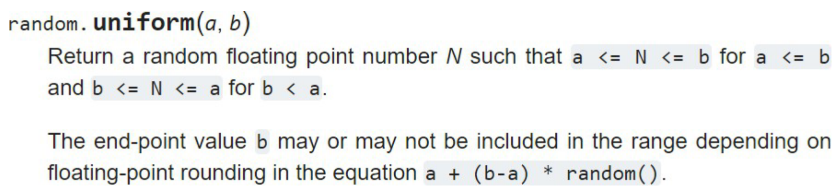
The index provides an alphabetical list of all the available modules.


Python 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

_	
`__future__`	*Future statement definitions*
`__main__`	*The environment where top-level code is run. Covers command-line interfaces, import-time behavior, and ``__name__ == '__main__'``.*
`_thread`	*Low-level threading API.*
`_tkinter`	*A binary module that contains the low-level interface to Tcl/Tk.*
a	
`abc`	*Abstract base classes according to :pep:`3119`.*
`aifc`	**Deprecated:** *Read and write audio files in AIFF or AIFC format.*
`argparse`	*Command-line option and argument parsing library.*
`array`	*Space efficient arrays of uniformly typed numeric values.*
`ast`	*Abstract Syntax Tree classes and manipulation.*
`asynchat`	**Deprecated:** *Support for asynchronous command/response protocols.*
`asyncio`	*Asynchronous I/O.*
`asyncore`	**Deprecated:** *A base class for developing asynchronous socket handling services.*
`atexit`	*Register and execute cleanup functions.*
`audioop`	**Deprecated:** *Manipulate raw audio data.*

There are many specialized modules, and in a typical Python script, you may use several. 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`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

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

6. **Close the documentation, and return to your Python editor.**

7. **Run the following code:**

```
>>> import random

>>> j = random.uniform(0, 100)

>>> print(j)
```

The result is a float between 1 and 100. You will use this function again later in this exercise.

Save Python code as scripts

So far in this exercise, you have worked only with the interactive Python interpreter. The Python window works great for 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. You will first practice creating, writing, and saving scripts using IDLE.

1. **Start IDLE if it is not already open.**

By default, IDLE open as the Python Shell, or interactive interpreter. Next, you will create a new script window.

2. **In the Python Shell, click File > New File.**

This opens a new script window. When using IDLE, the interactive interpreter and individual scripts are separate windows, whereas in editors such as PyCharm, these components are integrated into a single interface.

Next, you will enter the same code you worked with in the interactive interpreter.

3. **In the script window, type the following code:**

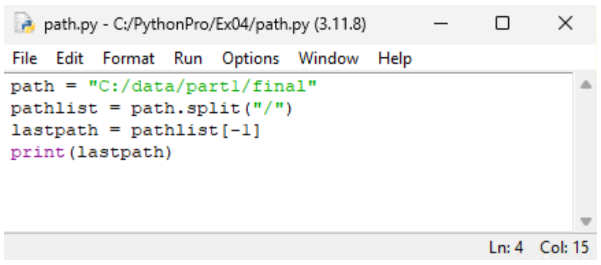
```
path = "C:/data/part1/final"

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 Python interpreter. Also notice that nothing happens 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.

Next, save the script.

4. **Click File > Save As, and navigate to C:\PythonPro\. Create a new folder, and name it Ex04. Inside that folder, save the script as path.py.**

The file extension .py indicates the file is a Python script. There is no need to type the .py file extension when the file type is set to Python files. Once the script file is saved, the path and file name appear at the top of the script window.

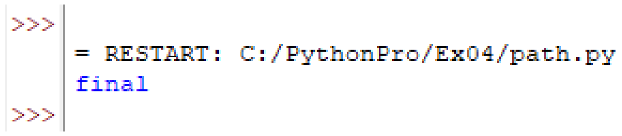

path.py - C:/PythonPro/Ex04/path.py (3.11.8)
File Edit Format Run Options Window Help
Ln: 4 Col: 15

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

Next, you will run the script.

5. **Click Run > Run Module.**

This checks the script for any syntax issues, and then runs the script. The Python Shell opens, and the result prints to the interactive interpreter. Notice that there is also a message that the path.py script was run.



```
>>>
= RESTART: C:/PythonPro/Ex04/path.py
final
>>>
```

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

Note: If you accidentally close the Python Shell but your script window is still open, running the script brings up the Python Shell again.

without running the code. In a typical workflow, you may use both the interactive interpreter and the script window. Later exercises show examples of using both in a single workflow.

6. **Close the path.py script but leave the Python Shell 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. **In the Python Shell, click File > New File.**

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

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

3. **Save the script as branching.py to the C:\PythonPro\Ex04 folder.**

4. **Run the script to confirm that it works correctly.**

   Every time you run the script, the result will be different.

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

5. **Replace line 3 with the following:**

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

   The code `p == 6` is an example of a condition where 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.

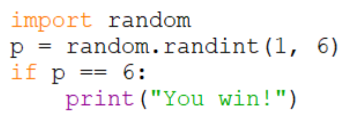
A few things to remember about the `if` structure: First, the `if` statement ends with a colon. Second, the lines following the `if` statement are indented. When you indent a line, the code becomes a block. A block consists of one or more consecutive lines of code that have the same indentation.

Python editors such as IDLE assist with automatic indentation. When you press Enter following the `if` statement colon, the next line of code is automatically indented.

6. **Write the following line of code following the `if` statement:**

```
print("You win!" )
```

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



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

7. **Run the script. If the script was not saved since any edits were made, you will be prompted to first save the source code (i.e., the script file).**

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 editors typically use four spaces for indentation and convert a tab to four spaces.

Running the script may result in a message in the interactive interpreter, 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 using follow-up statements.

8. **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.

9. **Press Backspace to remove the indentation.**

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

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

11. Save and run the script.

By using the `if-else` structure, the code accounts 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.

12. Above the else statement, insert a line 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!")
```

13. 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 using an `if` statement first. Also, notice that all three statements end with a colon and that there is no "end" statement as there is in some programming languages.

14. **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. **In the Python Shell, click File > New File.**
2. **Save the script as while_loop.py to the C:\PythonPro\Ex04 folder.**

   Write the following code:

```
i = 0

while i <= 10:

    print(i)

    i += 1
```

3. **Run the script.**

The result is a printout 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, which is when the counter reaches the value of 11.

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`.

The `while` loop structure uses a syntax like 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 while_loop.py script.**

   Next, you can try a `for` loop.

5. **Create a new script, and save it as for_loop.py to the C:\PythonPro\Ex04 folder.**

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 for_loop.py script.**

Looping, or iterating, over a list is a common task. In the example code, the list consists of numbers, but it could consist of a list of files or datasets as well.

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 end the loop. You can use the `break` statement to accomplish this, which you'll do next.

9. **Create a new script, and save it as break_loop.py to the C:\PythonPro\Ex04 folder.**

10. **Write the following code:**

```
from math import sqrt

for i in list(range(1000, 0, -1)):

    root = sqrt(i)

    if root == int(root):

        print(i)

        break
```

11. **Run the script.**

The result prints 961.

The code determines the largest number below 1,000 where the square root of the number is an integer. A range of integers is created as a list, starting at 1,000 and counting down to zero (0). The negative step is used to iterate downward. The `range()` function does not return a list but a range object, so the `list()` function is used to create the list. 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.

Without the `break` statement, the code would continue to determine all the numbers where the square root of the number is an integer—i.e., 961, 900, 841, and so on—which is not asked for here.

Comment scripts

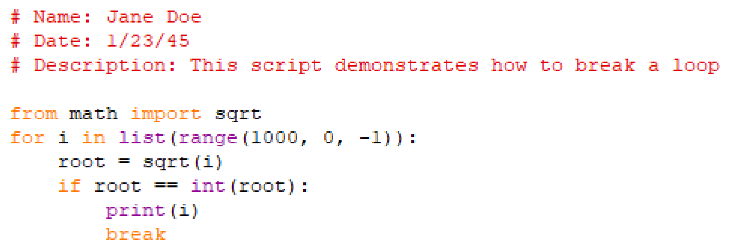
Well-developed scripts include comments that provide documentation about the script. Typically, the first few lines of a script consist of comments, but comments also occur throughout a script to explain how the script works. Comments are not executed when the script runs. In Python, a comment is preceded 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 break_loop.py script, place your pointer at the beginning of the code, and press Enter. At the top of the script, type the following lines of code:**

```
# Name: <your name>
# Date: <current date>

# Description: This script demonstrates how to break a loop
```

Enter your name and date. Your script window should now look like the example in the figure. Notice that the editor recognizes comments and shows them in red. Syntax highlighting using colors varies by editor.



```
# Name: Jane Doe
# Date: 1/23/45
# Description: This script demonstrates how to break a loop

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

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

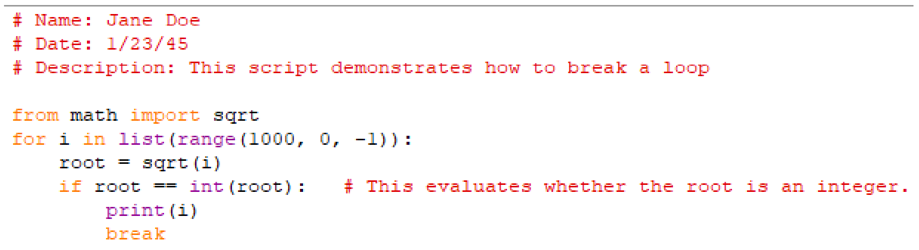
Note: Adding empty lines in your code is optional and has no effect on running the script. Typically, empty lines are added for readability. For example, it is common to add a line of space before or after comments or to keep lines of related code separate from other sections. This separation becomes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

You can also add comments at the end of a line of code.

2. **At the end of the line `if root == int(root)`, enter a few spaces and then the following comment:**

```
# This evaluates whether the root is an integer.
```

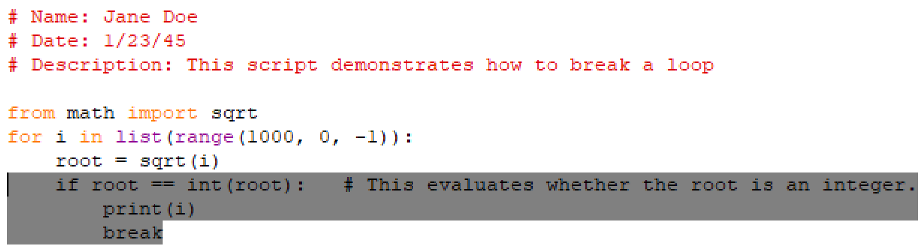
Inserting comments allows you to enter specific comments to explain specific parts of your code.



```
# Name: Jane Doe
# Date: 1/23/45
# Description: This script demonstrates how to break a loop

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

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

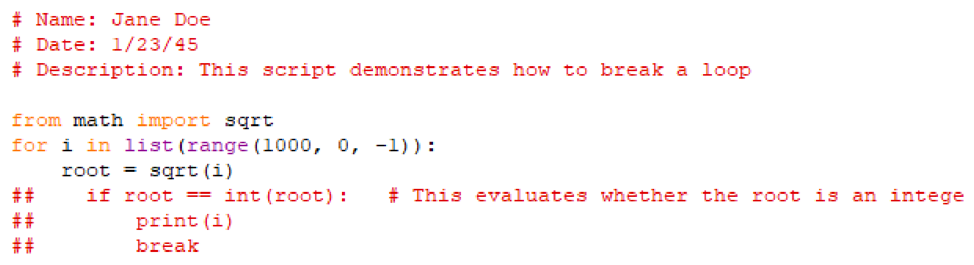
3. **In the break_loop.py script, select and highlight the last three lines of code.**



```
# Name: Jane Doe
# Date: 1/23/45
# Description: This script demonstrates how to break a loop

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

4. **Click Format > 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 runs.



```
# Name: Jane Doe
# Date: 1/23/45
# Description: This script demonstrates how to break a loop

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

5. **To make the code active again, select and highlight the commented lines, and click Format > Uncomment region.**

6. **Save your break_loop.py script.**

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 will review simple ways to identify and correct errors.

Start with some simple errors.

1. **Return to the Python Shell. If you closed it, you can open it from a script window by clicking Run > Python Shell.**

2. **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 an error:

```
NameError: name 'pint' is not defined
```

3. **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 the string, resulting in a syntax error:

```
SyntaxError: unterminated string literal
```

Notice that the error message provides specific details on the nature of 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.

4. **Open your branching.py script from earlier in the exercise. If you closed it, you can open it by clicking File > Open, and navigating to the C:\PytonPro\Ex04 folder.**

The script 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!")
```

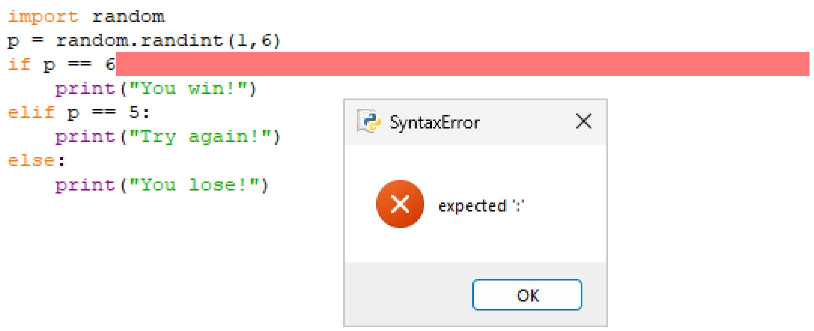
Next, try adding a small error.

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. **Click Run > Check Module.**

This checks the syntax of the code without running the code. The result reports an error:



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

```
SyntaxError
expected ':'
```

Notice that there is a red bar at the end of the third line of code where the syntax error is located, and the message reads: `SyntaxError: expect ;`. This option provides a quick way to test the syntax prior to running your script.

Note: If there are multiple syntax errors, only the first one is flagged.

7. **Correct the syntax error by placing a colon at the end of the if statement:**

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

8. **Now introduce a different error by placing a typo in the `randint()` function:**

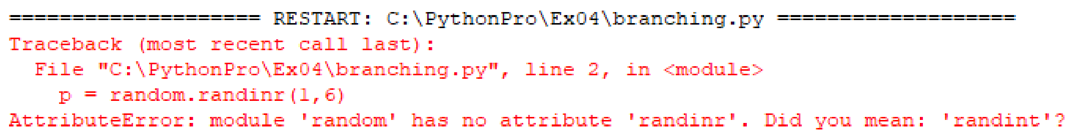
```
p = random.randinr(0, 6)
```

9. **Check the syntax of your script by clicking Run > Check Module.**

When there are no syntax errors, the Python Shell opens, but nothing prints. This 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.

10. **Try running your script.**

Notice a lengthy error message that is printed to the Interactive Window. The last three lines read as follows:



```
==================== RESTART: C:\PythonPro\Ex04\branching.py ===================
Traceback (most recent call last):
  File "C:\PythonPro\Ex04\branching.py", line 2, in <module>
    p = random.randinr(1,6)
AttributeError: module 'random' has no attribute 'randinr'. Did you mean: 'randint'?
```

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).

Error messages are not always helpful, but in most cases, they provide some insight into the location and nature of the issue you need to address.

End of Exercise 4.