Part I Introduction to Coding Concepts Using Python

Chapter 1 Using Python Interactively

Python is a very versatile language that includes a lot of built-in functionality, as well as many add-ons. Python code is ***interpreted***, which means that it is executed one line at a time. There are several environments in which Python can be used. The main ones are the Python ***interpreters***, creation of ***scripts***, and Jupyter notebooks.

- The Python interpreters allow you to enter and execute code one line at a time.
- Scripts allow you to save multiple lines of code in a file and then execute them sequentially, which gives the results all at once rather than executing and getting the results one line at a time. To create a script, use an editor and save a file with an extension of .py on the filename.
- Jupyter notebooks allow combinations of text that can be formatted and executable code (and the results) in one document; the code is entered in ***cells***.

The details of installation and implementation will not be covered in this book, but it is assumed that the reader will have access to some form of Python and will be able to follow along and test the code. This book uses Python 3, which is quite different from Python 2.

Since Python is interpreted, it is possible to use Python ***interactively***. This means that you (the ***user***) enter an expression, and then Python responds with the result. What this looks like depends on the environment in which you are using Python.

In some cases, such as using the Python interpreter, you will see a ***prompt***, which is generally three greater than signs in a row, e.g.

```
>>>
```

When you see this prompt, you can enter an expression, e.g. 3 + 5, and then hit the Return key.

```
>>> 3 + 5
8
>>>
```

The Python interpreter responds with the result (in this case, 8), and then displays another prompt so that you can enter another expression. Note that this is the prompt for a one-line expression. When code takes multiple lines, the >>> is the first or ***primary prompt***. This is followed by lines starting with an ellipsis, which is 3 dots in a row (...), which is called the ***secondary prompt***.

In some cases, such as in Jupyter notebooks or ipython, the prompt will be the word "In" and then an integer in square brackets followed by a colon, and the response will consist of the word "Out" and then the same integer in square brackets with a colon, e.g.

```
In[1]: 3 + 5
Out[1]: 8
```

The integers increase sequentially as you enter and execute expressions.

Some environments, such as Jupyter notebooks, allow more than one expression to be entered into a ***cell***, and then ***executed*** sequentially together.

In this text, although the actual prompt looks different in different environments, the prompt(s) will be shown as >>>. The prompt and the user input will be shown in a code font in italics, and the Python response will not have italics, e.g.

```
>>> 3 + 5
8
```

In order to be concise, the prompt that would appear after the result will not be shown.

Note that the prompt will be shown for short, simple codes. Longer code will instead be shown in boxes.

All expressions (for now) should begin in the first column; they should not be indented. Spacing and indentation is important in Python; following this rule will avoid some errors.

Longer expressions can be extended to another line by either putting a '\' at the end of the first line, or putting the expression in parentheses. This can be extended to multiple lines, also.

```
>>> 1 + 2 + 3 + \
...4 + 5
15

>>> (1 + 2 + 3
...+ 4 + 5)
15
```

Using parentheses is generally preferred. In these examples, the addition operator (+) can be at the end of one line, or at the beginning of the next, as shown. Also, note the primary prompt >>> on the first line, and the secondary prompt … on the second line of these examples.

1.1 Variables and Assignment Statements

If it is desired to store a value for subsequent use, a ***variable*** can be used. Values can be stored in variables using an ***assignment statement***. The general form of an assignment statement is

```
variable = expression
```

where the name of the variable is on the left, the ***assignment operator*** is the single equal sign, and the expression is on the right. For simplicity we can assume that the way it works is that first the expression is evaluated, and then the value of the expression is assigned to the variable. For example, the statement

```
>>> mynumber = 29 + 4
```

stores the number 33 in the variable called *mynumber*. This result is not displayed by default. However, just typing the name of a variable at the prompt will display the value, e.g.

```
>>> mynumber
33
```

The terminology is that the assignment ***binds*** the variable name *mynumber* to the value 33. The 33 is stored in a location, and the variable *mynumber* references that location.

As we will see later, what we are calling variables here are actually ***objects***. While a variable simply stores a value, an object has both ***attributes*** such as the data stored in it, as well as functions that can operate on the data (these are called ***methods***). For now, however, we will keep it simple and refer to them as variables.

Values of some types of variables can change. Types that can change are called ***mutable***, and types that cannot change are ***immutable***. Number variables are immutable types. When the variable *mynumber* is used or displayed, it refers to the value that is stored in it. The following will take the current value of the variable *mynumber* (which is 33), subtract 2 from it, to result in the number 31, which is stored in a new location. The assignment binds the variable *mynumber* to this new location. It appears as though the value of *mynumber* has changed, and it is easiest to think of it that way. In reality, *mynumber* now refers to a new location in which 31 is stored.

```
>>> mynumber = mynumber – 2
>>> mynumber
31
```

This demonstrates that the order of operations in an assignment statement is important: first the expression on the right is evaluated, and then the result is stored in a location that is bound to the variable on the left (which may or may not be the same variable). This means that the equal sign, which is the assignment operator, does NOT mean equality! One way to read this out loud, instead of using the word "equals", is to say "*mynumber* gets the value of *mynumber* minus 2".

Adding a value to a variable is called ***incrementing*** the variable, and subtracting from it is called ***decrementing***.

There are rules for variable names: they must start with a letter of the alphabet or the underscore character (_), and then can have any combination of letters, digits, and the underscore character. Spaces may not be used in variable names. It is common to use all lower case letters. Variable names should be ***mnemonic***, which means that the name should help to explain what is stored in the variable. For example, to store the radius of a circle in a variable, the variable name *radius* would be mnemonic; a variable called *x* would not be descriptive.

Python also has a default variable, which is just the underscore character. Whenever an expression is entered interactively, and the result is not stored in a variable, Python stores it in the default variable _. This variable is reused every time just an expression is entered. For example,

```
>>> 3 + 5
8

>>> _ + 4
12

>>> _ * 2
24
```

It is best practice, however, to avoid using this default variable.

Multiple assignments can be made in one statement, e.g.

```
>>> a, b = 5, 33
```

This will assign the value of 5 to the variable *a* and the value of 33 to the variable *b*. It is easier to read code in which every assignment is on a separate line, however, so this usage is generally discouraged. An exception might be assigning 0's to several variables at once, and we will see later that there are some interesting applications for this ***simultaneous assignment***.

The word 'variable' means something that can change, and that is how variables are used in coding. ***Constants*** are values that cannot change during the execution of the code. Some languages provide a way to define constants, but Python does not. Instead, a value that is meant to be a constant in Python code is by convention stored in a variable named with all upper-case letters. For example, a tax rate might be defined in an assignment statement as follows.

>>> TAX_RATE = 0.05

To delete any variable that has been created, use the **del** statement:

```
>>> del mynumber
```

1.2 Built-in Functions

Python has many built-in functions that can be used. As an example of a built-in function, we will first examine the **round** function. An example of using it is:

```
>>> round(7.6)
8
```

This rounds the real number 7.6 up to the integer 8.

The terminology when using a function is that the function is ***called***; the ***function call*** includes the name of the function and ***argument(s)*** that are ***passed*** to it in parentheses. If there are multiple arguments, they are separated by commas. In this example, the argument 7.6 was passed to the function named **round**. Most functions ***return*** a result; in this case, the rounded result of 8 is returned.

Another, different type of function is the **print** function. For example, the value stored in a variable can be displayed using the **print** function, as in:

```
>>> val = 5 * 2
>>> print(val)
10
```

Of course, the value of the variable can also be displayed by just typing the name of the variable. The **print** function, however, can be used to display more than one result, e.g.

```
>>> print(val, 2*3)
10  6
```

Note that a space is automatically printed between the two values.

With the **print** function, the values of the expression arguments are displayed, and it might not seem like any result is returned. The **print** function, however, actually returns the special value **None**, since it does not really need to return anything. Normally this value is not seen or used, but can be displayed by assigning the result of calling the **print** function to a variable (which is not something that would typically be done!). We will see that all Python functions return something (even if it is just None, as with **print**).

Names of functions should never be used as variable names. Although it is technically possible to do this, it would eliminate the use of the function itself (at least temporarily, until the variable is deleted). For example, if the word 'print' is used as a variable, it could not then be used to print anything. The second statement here would create an error message because *print* is now a variable, not a function that can be called.

```
>>> print = 11
>>> print(8-3)
TypeError: 'int' object is not callable
```

Note that the actual error message may be different in the various Python programming environments.

Deleting the *print* variable would, however, restore the use of the **print** function:

```
>>> del print
>>> print(8-3)
5
```

1.3 Types and Operators

There are several built-in types for expressions and variables. Each type has different operators and functions that can be used with expressions of that type.

Variables in Python are actually ***objects***, which are created from ***classes***. We will see more on objects and object oriented programming (OOP) in a later chapter.

1.3.1 Numbers

Simple numbers can be integers (whole numbers, with no decimal point), or real numbers (which have a decimal point). Integers are of the type **int** in Python, and real numbers are of the type **float**. There are also complex numbers, which are of the type **complex** (and are in the form a + bj or a+bJ). The type of an expression can be determined using the **type** function. For example,

```
>>> print(type(1+3), type(5.2))
<class 'int'> <class 'float'>
```

Python shows that the type (or ***class***) of the first expression is **int**, whereas the type of 5.2 is **float**.

Numbers can be written in scientific notation using "e", as in 2e3, which is equivalent to $2*10^3$. The result is always a float.

```
>>> 2e3
2000.0
```

There are operators and functions that work with numbers.

We have seen some basic math operators, which include:

```
Addition            +
Subtraction         -
Multiplication      *
Negation            -
```

The addition, subtraction, and multiplication operators are called ***binary operators***, as they operate on two ***operands***. Negation is a ***unary operator***, as it operates on only one operand.

When using these operators, if the operand(s) are integers, the result will be an integer (type **int**). If anything in the expression is a **float**, however, the result will be a **float**.

```
>>> 3 + 5
8
>>> 5.2 - 3.2
2.0
```

Other operators include:

```
Exponentiation      **
Float division      /
Integer division    //
Remainder division  %
```

Exponentiation, or raising one number to a power, is accomplished using the ** operator. For example, 5 ** 2 is 5^2, which is 5 raised to the second power, or 5 squared, or 25.

```
>>> 5 ** 2
25
```

There are several division operators.

The / division operator always results in a **float**, regardless of the types of the operands. For example,

```
>>> 5.2/2
2.6

>>> 6/3
2.0
```

The // division operator always results in an integer (conceptually, anyway!), regardless of the types of the operands. The // operator results in the whole number part of the division, and is sometimes called ***floor division*** since it gets rid of the fractional part. The following examples show that 2 divides into 7 three times, 4 divides into 7 once, and 3 divides into 7.5 two times.

```
>>> 7//2
3

>>> 7//4
1

>>> 7.5//3
2.0
```

Notice the types of the results. With the expression 7.5//3, the result that is returned is conceptually the integer 2, but it is stored as a **float** and not the type **int** since the expression contains a **float**.

The % operator will show the remainder (or ***modulus***) of these divisions, e.g.

```
>>> 7.5%3
1.5

>>> 7%4
3
```

So, 3 goes into 7.5 two times, with a remainder of 1.5 (7.5-3*2), and 4 goes into 7 one time with a remainder of 3. Again, notice the types of the results.

In addition to the built-in operators, there are math functions that are in core Python. These include rounding functions such as **round**, which we have already seen:

```
>>> round(7.5)
8
```

The **round** function can also round floats to a specified number of decimal places. For example, we can round 8/3 to 2 decimal places. To do this, a second argument is passed to the **round** function, which is the number of decimal places.

```
>>> 8/3
2.6666666666666665

>>> round(8/3,2)
2.67
```