Chapter 3
A First Python Program

3.1 Introduction

In this chapter we will return to the Hello World program and look at what it is doing. We will also modify it to become more interactive and will explore the concept of Python variables.

3.2 Hello World

As mentioned in the previous chapter, it is traditional to get started in a new programming language with writing a *Hello World* style program. This is very useful as it ensures that your environment, that is the interpreter, any environmental settings, your editor (or IDE) etc. are all set up appropriately and can process (or compile) and execute (or run) your program. As the 'Hello World' program is about the simplest program in any language, you are doing this without the complexities of the actual language being used.

Our 'Hello World' program has already been presented in the first chapter of this book, however we will return to it here and take a closer look at what is going on.

In Python the simplest version of the *Hello World* program merely prints out a string with the welcome message:

```
print('Hello World')
```

You can use any text editor or IDE (Integrated Development Editor) to create a Python file. Examples of editors commonly used with Python include Emacs, Vim, Notepad++, Sublime Text or Visual Studio Code; examples of IDEs for Python include PyCharm and Eclipse. Using any of these tools we can create file with a

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_3

`.py` extension. Such a file can contain one or more Python statements that represent a Python program or Script.

For example, we can create a file called `hello.py` containing the above `print()` function in it.

One question this raises is where does the `print()` function come from?

In fact, `print()` is a predefined function that can be used to *print things out*, for example to the user. The output is actually printed to what is known as the output stream. This handles a stream (sequence) of data such as letters and numbers. This output stream of data can be sent to an output window such as the terminal on a Mac or Command Window on a Windows PC. In this case we are printing the string `'Hello World'`.

By *predefined* here we mean that it is built into the Python environment and is understood by the Python interpreter. This means that the interpreter knows where to find the definition of the `print()` function which tells it what to do when it encounters the `print()` function.

You can of course write your own functions and we will be looking at how to do that later in this book.

The `print()` function actually tries to print whatever you give it,

- when it is given a string it will print a string,
- if it is given an integer such as `42` it will print `42` and
- if it is a given a floating point number such as `23.56` then it will print 23.56.

Thus, when we run this program, the string 'Hello World' is printed out to the console window.

Also note that the text forming the Hello World string is wrapped within two single quote characters; these characters delimit the start and end of the string; if you miss one of them out then there will be an error.

To run the program, if you are using an IDE such as PyCharm, then you can select the file in the left hand tree and from the right mouse button select Run. If you want to run this program again you can merely click on the green arrow in the tool bar at the top of the IDE.

If you are running it from the command line type in `python` followed by the name of the file, for example:

```
> python hello.py
```

This should be done in the directory where you created the file.

3.3 Interactive Hello World

Let us make our program a little more interesting; lets get it to ask us our name and say hello to us personally.

The updated program is:

```
print('Hello, world')
user_name = input('Enter your name: ')
print('Hello ', user_name)
```

Now after printing the original 'Hello World' string, the program then has two additional statements.

The result of running this program is:

```
Hello, world
Enter your name: John
Hello John
```

We will look at each of the new statements separately.

The first statement is:

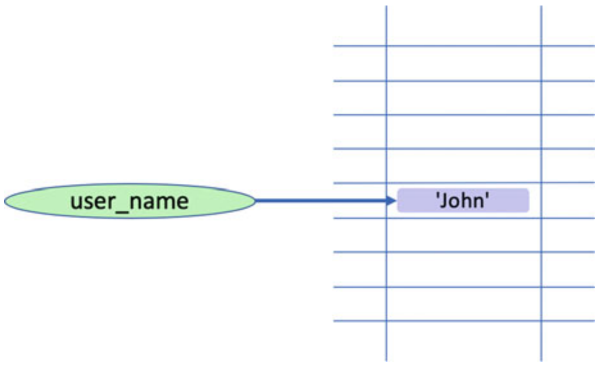
```
user_name = input('Enter your name: ')
```

This statement does several things. It first executes another function called `input()`. This function is passed a string—which is known as an argument—to use when it prompts the user for input.

This function `input()`, is again a *built-in* function that is part of the Python language. In this case it will display the string you provide as a prompt to the user and wait until the user types something in followed by the return key.

Whatever the user types in is then returned as the result of executing the `input()` function. In this case that result is then stored in the *variable* `user_name`.

A *variable* is a named area of the computers' memory that can be used to hold things (often referred to as data) such as strings, numbers, boolean such as True/False etc. In this case the variable *user_name* is acting as a label for an area of memory which will hold the string entered by the user. The basic idea is illustrated in the following diagram:



This simplified picture illustrates how a variable, such as user_name, can reference an area of memory containing actual data. In this diagram the memory is shown as a two dimensional grid of memory *locations*. Each location has an address associated with it. This address is unique within the memory and can be used to return to the data held at that location. This address is often referred to as the memory address of the data. It is this memory address that is actually held in the variable user_name; this is why the user_name variable is shown as pointing to the area in memory containing the string 'John'.

Thus the variable user_name allows us to access this area of memory easily and conveniently.

For example, if we want to get hold of the name entered by the user in another statement, we can do so merely by referencing the variable user_name. Indeed, this is exactly what we do in the second statement we added to our program. This is shown below:

```
print('Hello', user_name)
```

The last statement again uses the built-in print() function, but this time it takes two arguments. This is because the print() function can actually take a variable number of arguments (data items that we pass into it). Each argument is separated by a comma. In this case we have passed in the string 'Hello' and whatever value is referenced by (present at the memory address indicated by) the variable user_name.

3.4 Variables

You may wonder why the element holding the user's name above is referred to as a *variable*. It is a called a variable because the value it references in memory can vary during the lifetime of the program.

For example we can modify our *Hello World* program to ask the user for the name of their best friend and print out a welcome message to that best friend. If we want to, we can *reuse* the variable to hold the name of that best friend. For example:

```
print('Hello, world')
name = input('Enter your name: ')
print('Hello', name)
name = input('What is the name of your best friend: ')
print('Hello Best Friend', name)
```

When we run this version of the program and the user enters 'John' for their name and 'Denise' for their best friends' name we will see:

```
Hello, world
Enter your name: John
Hello John
What is the name of your best friend: Denise
Hello Best Friend Denise
```

As you can see from this when the string 'Hello Best Friend' is printed, it is the name 'Denise' that is printed alongside it.

This is because the area of memory that previously held the string 'John' now holds the string 'Denise'.

In fact, in Python the variable name is not restricted to holding a string such as 'John' and 'Denise'; it can also hold other types of data such as numbers or the values `True` and `False`. For example:

```
my_variable = 'John'
print(my_variable)
my_variable = 42
print(my_variable)
my_variable = True
print(my_variable)
```

The result of running the above example is

```
John
42
True
```

As you can see `my_variable` first holds (or references the area of memory containing) the string '`John`', it then holds the number `42` and finally it holds the Boolean value `True` (Boolean values can only be `True` or `False`).

This is referred to in Python as *Dynamic Typing*. That is the *type* of the data held by a variable can *Dynamically* change as the program executes. Although this may seem like the obvious way to do things; it is not the approach used by many programming languages such as Java and C# where variables are instead *Statically Typed*. The word static is used here to indicate that the type of data that a variable can hold will be determined when the program is first processed (or compiled). Later on it will not be possible to change the type of data it can hold; thus if a variable is to hold a number it cannot later on hold a String. This is the approach adopted by languages such as Java and C#.

Both approaches have their pros and cons; but for many people the flexibility of Pythons variables are one of its major advantages.

3.5 Naming Conventions

You may have noticed something above some of the variable names we have introduced above such as `user_name` and `my_variable`. Both these variable names are formed of a set of characters with an underbar between the 'words' in the variable name.

Both these variable names highlight a very widely used naming convention in Python, which is that variable names should:

- be all lowercase,
- be in general more descriptive than variable names such as *a* or *b* (although there are some exceptions such as the use of variables *i* and *j* in looping constructs).
- with individual words separated by underscores as necessary to improve readability.

This last point is very important as in Python (and most computer programming languages) spaces are treated as separators which can be used to indicate where one thing ends and another starts. Thus it is not possible to define a variable name such as:

- user name

As the space is treated by Python as a separator and thus Python thinks you are defining two things 'user' and 'name'.

When you create your own variables, you should try to name then following the Python accepted style thus name such as:

- `my_name, your_name, user_name, account_name`
- `count, total_number_of_users, percentage_passed, pass_rate`
- `where_we_live, house_number,`
- `is_okay, is_correct, status_flag`

are all acceptable but

- `A, Aaaaa, aaAAAaa`
- `Myname, myName, MyName or MYName`
- `WHEREWELIVE`

Do not meet the accepted conventions.

However, it is worth mentioning that these are merely commonly adhered to conventions and even Python itself does not always comply with these conventions. Thus if you define a variable name that does not conform to the convention Python will not complain.

3.6 Assignment Operator

One final aspect of the statement shown below has yet to be considered

```
user_name = input('Enter your name: ')
```

What exactly is this '=' between the `user_name` variable and the `input()` function?

It is called the *assignment* operator. It is used to assign the value returned by the function `input()` to the variable `user_name`. It is probably the most widely used operator in Python. Of course, it is not just used to assign values from functions as the earlier examples illustrated. For example, we also used it when we stored a string into a variable directly:

```
my_variable = 'Jason'
```

3.7 Python Statements

Throughout this chapter we have used the phrase statement to describe a portion of a Python program, for example the following line of code is a statement that prints out a string 'Hello' and the value held in `user_name`.

```
print('Hello', user_name)
```

So what do we mean by a statement? In Python a statement is an instruction that the Python interpreter can execute. This statement may be formed of a number of elements such as the one above which includes a call to a function and an assignment of a value to a variable. In many cases a statement is a single line in your program but it is also possible for a statement to extend over several lines particularly if this helps the readability or layout of the code. For example, the following is a single statement but it is laid out over 6 lines of code to make it easier to read:

```
print('The total population for',
      city,
      'was',
      number_of_people_in_city,
      'in',
      year)
```

As well as statements there are also expressions. An expression is essentially a computation that generates a value, for example:

```
4 + 5
```

This is an expression that adds 4 and 5 together and generates the value 9.

3.8 Comments in Code

It is common practice (although not universally so) to add comments to code to help anyone reading the code to understand what the code does, what its intent was, any design decisions the programmer made etc.

Comments are sections of a program that are ignored by the Python interpreter—they are not executable code.

A comment is indicated by the '#' character in Python. Anything following that character to the end of the line will be ignored by the interpreter as it will be assumed to be a comment, for example:

```
# This is a comment
name = input('Enter your name: ')
# This is another comment
print(name)  # this is a comment to the end of the line
```

In the above, the two lines starting with a # are comments—they are for our human eyes only. Interestingly the line containing the `print()` function also has a comment—that is fine as the commend starts with the # and runs to the end of the line, anything *before* the # character is not part of the comment.

3.9 Scripts Versus Programs

Python can be run in several ways:

- Via the Python interpreter by entering the REPL; the interactive Python session.
- By having the Python interpreter run a file containing stored Python commands.
- By setting up an association at the operating system level so that any files ending with `.py` are always run by the Python interpreter.
- By indicating the Python interpreter to use at the start of the Python file. This is done by including a first line in a file with something similar to '`#!/usr/bin/env python`'. This indicates that the rest of the file should be passed to the Python interpreter.

All of these can be defined as a way to run a Python program or indeed a Python script.

However, for the purposes of this book we will treat a file containing a first line specifying the Python interpreter to use as a script. All other Python code that represents some code to execute for a specific purpose will be called a Python program. However, this is really only a distinction being made here to simplify terminology.

3.10 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/reference/simple_stmts.html For information on statements in Python.

3.11 Exercises

At this point you should try to write your own Python program. It is probably easiest to start by modifying the *Hello World* program we have already been studying. The following steps take you through this:

1. If you have not yet run the *Hello World* program, then do so now. To run your program you have several options. The easiest if you have set up an IDE such as PyCharm is to use the 'run' menu option. Otherwise if you have set up the Python interpreter on your computer, you can run it from a command prompt (on Windows) or a Terminal window (on a Mac/Linux box).
2. Now ensure that you are comfortable with what the program actually does. Try commenting out some lines—what happens; is that the behaviour you expected? check that you are happy with what it does.
3. Once you have done that, modify the program with your own prompts to the user (the string argument given to the input function). Make sure that each string is surrounded by the single quote characters (''); remember these denote the start and end of a string.
4. Try creating your own variables and storing values into those instead of the variable `user_name`.
5. Add a `print()` function to the program with your own prompt.
6. Include an assignment that will add two numbers together (for example `4 + 5`) and then assign the result to a variable.
7. Now print out that variables value once it has been assigned a value.
8. Make sure you can run the program after each of the above changes. If there is an error reported attempt to fix that issue before moving on.

You must also be careful with the indentation of your program—Python is very sensitive to how code is laid out and at this point all statements should start at the beginning of the line.