University of Strathclyde Glasgow

# An introduction to Python for those new to computer programming

W. H. Bell

October 10, 2023

**Abstract**

Basic aspects of programming in the Python computer programming language are introduced. Concepts are introduced slowly, explaining how data are stored and manipulated within the computer's memory. This document assumes no prior programming experience.

# Contents

# 1 Introduction

Python is a popular computer programming language that can be used for many different applications. It can be used with simple computers, to create games, to build web services and to perform data analysis.

Python can be used on Windows, Mac (OSX), Linux and other operating systems. It can be combined with other programming languages to produce final applications.

This document introduces some of the basic building blocks of the Python computer programming language. It is assumed that the reader may have no prior experience of computer programming. When it is necessary to use technical words, these words are defined within a glossary that is given later in this document.

The best way to understand a computer programming language is to use it. Examples are provided within the following sections of this document. The purpose of these examples is to discuss how the Python programming language works. To understand these examples, they should be run, where the line-by-line functionality can be verified by using the Python debugger.

# 2 Software Installation

**This document assumes that Python 3.6 or higher has been installed.**

Python 2.7 is still used for several purposes and is packaged with Linux distributions. However, the functionality of Python 2.7 is different to Python 3, especially concerning text manipulation. Python is being actively developed from Python 3. Therefore, this course focuses on Python 3. Python 3 can be installed on Microsoft Windows, Linux, Mac (OSX) and other operating systems.

Python can be downloaded from https://www.python.org/. On Linux, it can be installed with the Linux package manager. When Python is installed, it must be added to the `PATH` environment variable by selecting the appropriate option during installation. Once Python has been installed, additional packages can be installed with the package installer for Python (pip).

Anaconda provides Python and a range of Python packages. It can be used instead of the standalone Python installation. Anaconda can be downloaded from https://www.anaconda.com/products/individual.

## 2.1 Editors

The Python programming language is written as text that is saved in text files that end with the ".py" suffix. These text files can be edited with a range of different text editors. Some text editors colour the Python programming syntax, such that it can be more easily understood by developers. Editors may also suggest functions or corrections to programs to aid the developer. Finally, editors may provide integration with tools to run and debug Python programmes. These fully functional editors are referred to as integrated development environments (IDEs).

Visual Studio Code provides a simple IDE environment for python development and can be installed on Windows, MacOS or Linux. It is available at https://code.visualstudio.com/. It should be installed in

the default location, such that it can be found by the Anaconda navigator. The system installer should be used on Windows. On MacOS, the application should be installed into the Applications folder. After Visual Studio Code has been installed, it will be visible in the Anaconda navigator when the Anaconda navigator is restarted. It should be started from the Anaconda navigator if it is used with Anaconda. On Linux, it can be started from the applications menu.

# 3 Using Python

The Python programming language is interpreted by the `python` program. The `python` program can be started to allow a user to type Python commands or it can be used to process Python commands that are saved in a text file.

## 3.1 Using Python interactively

Python can be started interactively by typing `python` into a command prompt or terminal or by using Anaconda. An example of the interactive prompt is given in Listing 1.

Listing 1: The Python prompt.

```
>>>
```

Python commands can then be typed into the window, similar to those given in Listing 2.

Listing 2: Using the Python prompt.

```
>>> print("Hello World")
Hello World
>>> x = 4
>>> x + 2
6
```

When Python is used interactively, information that is printed to the screen is given below the Python command. Likewise, information that is returned is also given below the command. In Listing 2, the text `"Hello World"` is printed to the screen, whereas the value `6` is returned from `x + 2`.

## 3.2 Using Python files

Python can be written into text files that have a ".py" suffix. The Python interpreter is used to execute these text files. This can be achieved by running the program using an IDE such as Visual Studio Code, or by typing `python` followed by the name of the text file.

A simple Python file is given in Listing 3. This file contains one line of Python that prints the text `"Hello World"` to the screen.

Listing 3: hello_world.py

```
print("Hello world")
```

# 4 Defining variables

Information can be copied into variables. A variable is a space within the computer's memory. The size of the space that is associated with the variable depends on the type of the variable and the number of elements that are associated with it. Once information has been copied into a variable, the computer can operate on the variable and update its value.

In the Python programming language, a variable is created when it is first assigned a value. The type of the value that is assigned sets the type of the variable.

## 4.1 Defining an integer

An integer is a whole number, which does not have a decimal fraction following it. An integer variable is created and assigned the value `4` in Listing 4.

Listing 4: Creating an integer variable.

```
x = 4
```

## 4.2 Defining a float

A float or floating point number is a number that may have a decimal fraction that is associated with it. For example, `3.14159` is a floating point number. A float variable is created and assigned the value `3.14159` in Listing 5.

Listing 5: Creating a float variable.

```
x = 3.14159
```

## 4.3 Defining a string

A string is a variable that is used to store a series of text characters. For example, `"This is some text"` is a text string that comprises 17 characters. A string variable is created and assigned the value `"This is some text"` in Listing 6.

Listing 6: Creating a string variable.

```
s = "This is some text"
```

## 4.4 Defining a list

A list is a sequential data container, where each element is stored in memory after the previous one. It can be empty or contain several entries. The entries of a list can be any Python variable type, including a list. An empty list is defined in Listing 7.

Listing 7: Creating an empty list.

```
values = []
```

A list can be defined with some initial values. For example, Listing 8 demonstrates two ways of defining a list with four values. The text layout for `more_values` is slightly different than for `values`, but the functionality is the same.

Listing 8: Creating lists with initial values.

```
values = [1, 2, 3, 4]
more_values = [
    1,
    2,
    3,
    4
]
```

The structure of the list `values` and `more_values` is illustrated in Figure 1. Each value is stored in a separate element of the list. The list index is used to access the list value. The index is an offset, where an index value of `0` refers to the first element in the list. The size of list element is set by the type of value stored within the list element.

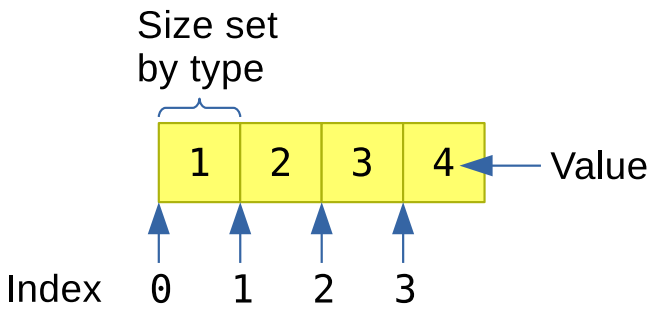


Figure 1: The elements of a list, indices and values.

## 4.5 Defining a dictionary

A dictionary is a collection of key and value pairs. The keys have to be unique within the dictionary, whereas the values can hold duplicate values. The keys are used to access the values within the dictionary. They are stored randomly, such that data access is as rapid as possible.

A dictionary can be empty or contain several key and value pairs. An empty dictionary is defined in Listing 9.

Listing 9: Creating an empty dictionary.

```
data = {}
```

A dictionary can be defined with some initial values. For example, Listing 10 demonstrates two ways of defining a dictionary with three key and value pairs. The text layout for `more_data` is slightly different than for `data`, but the functionality is the same.

Listing 10: Creating dictionary with initial values.

```
data = {1: "a", 26: "z", 10: "q"}
more_data = {
    1: "a",
    26: "z",
    10: "q"
}
```

The structure of the list `data` and `more_data` is illustrated in Figure 2.

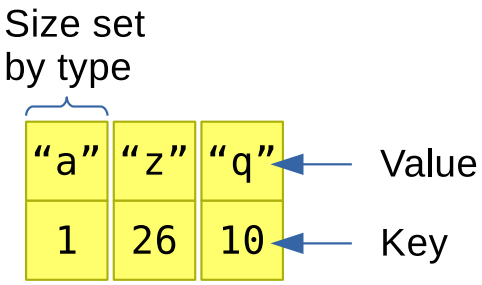


Figure 2: A dictionary, showing the relationship between keys and values.

### 4.6 Redefining variables

If a variable is assigned a new value that does not match its current type, then the variable is redefined with the new type. In Listing 11, a series of assignments are used to define the type of the variable `x`. Line numbers are included to allow discussion of the Python, but are not part of the program.

In Listing 11, the variable `x` is defined on Line 1 as an integer. Then on the lines that follow the variable `x` is redefined as a float, string, list and dictionary, respectively. While the Python programming language allows redefinition of variable types, it should be avoided to make the program easier to understand and debug.

Listing 11: Redefining the type of a variable.

```
1 x = 10
2 x = 2.3
3 x = "20"
4 x = []
5 x = {}
```

## 5 Using and updating variables

Once a variable has been defined its value can be used or updated. The value of the variable is accessed using the name of the variable.

## 5.1 Using an integer variable

The Python program in Listing 12 demonstrates how to set, update and use an integer variable. The program includes:

- Line 1: A variable `x` is defined as an integer, with an initial value 10.
- Line 2: The value of the variable `x` is printed on the screen by providing the `print` function with the variable name.
- Line 3: The value of `x` is updated by adding 2 to it and then putting the resulting value back into `x`.
- Line 4: The value of `x` is updated by adding 2 to it and then putting the resulting value back into `x`. Line 4 is functionally the same as Line 3, where the shorthand notation of "`x +=`" is equivalent to "`x = x +`".
- Line 5: The value of the variable `x` is printed on the screen by providing the `print` function with the variable name.

Listing 12: Using and updating an integer variable.

```
1 x = 10
2 print(x)
3 x = x + 2
4 x += 2
5 print(x)
```

A float variable can be used in a similar manner as an integer variable.

## 5.2 Using a string variable

The Python program in Listing 13 demonstrates how to set, update and use a string variable. The program includes:

- Line 1: A string variable named `line` is defined, with an initial value `"Some text"`.
- Line 2: The value of the variable `line` is printed on the screen by providing the `print` function with the variable name.
- Line 3: The value of `line` is updated by appending another string to it and then putting the resulting value back into `line`.
- Line 4: The value of `line` is updated by appending another string to it and then putting the resulting value back into `line`. Line 4 appends text, in a similar manner as Line 3, where the shorthand notation of "`line +=`" is equivalent to "`line = line +`".
- Line 5: The value of the variable `line` is printed on the screen by providing the `print` function with the variable name.
- Line 6: The syntax `line[0:4]` selects a substring from `line`, from the first character (index 0) to the fourth character (index 3). The fifth character (index 4) is not included in the substring. The resulting substring is returned and passed to the `print` function.

Listing 13: Using and updating an string variable.

```
1 line = "Some text"
2 print(line)
3 line = line + ", some more text"
4 line += "."
5 print(line)
6 print(line[0:4])
```

Strings are stored in memory as a series of characters values. This is similar to a list, which is illustrated in Figure 1.

## 5.3 Using a list

The Python program in Listing 14 demonstrates how to set, update and use a list. The program includes:

- Line 1: An empty list named `values` is defined.
- Line 2: The contents of the list is printed by passing its name to the `print` function.
- Line 3: The value `2` is appended to the list.
- Line 4: The value `10` is appended to the list.
- Line 5: A new list is created that contains the values `100` and `1000`. This list is appended to the list `values`. The "`+`" operator appends one list to another one.
- Line 6: A new list is created that contains the value `10000`. This list is appended to the list `values`. The shorthand syntax "`values +=`" is equivalent to "`values = values +`".
- Line 7: The contents of the list is printed by passing its name to the `print` function.
- Line 8: The value in the first element of the list is set to `1`.
- Line 9: The contents of the list is printed by passing its name to the `print` function.

Listing 14: Using and updating a list.

```
1 values = []
2 print(values)
3 values.append(2)
4 values.append(10)
5 values = values + [100, 1000]
6 values += [10000]
7 print(values)
8 values[0] = 1
9 print(values)
```

The structure of a list and associated indices is illustrated in Figure 1. The list index 0 refers to the first element of the list.

## 5.4 Using a dictionary

The Python program in Listing 15 demonstrates how to set, update and use a dictionary. The program includes:

- Line 1: An empty dictionary named `data` is defined.
- Line 2: The contents of the dictionary is printed by passing its name to the `print` function.
- Line 3: A new dictionary element is added, using the key `"a"` and the value `10`.
- Line 4: A new dictionary element is added, using the key `"f"` and the value `15`.
- Line 5: The contents of the dictionary is printed by passing its name to the `print` function.
- Line 6: A new dictionary is defined that contains two key and value pairs. The new dictionary is passed to the `update` function, which combines the two dictionaries together.
- Line 7: The contents of the dictionary is printed by passing its name to the `print` function.
- Line 8: A new dictionary element is added, using the key `"a"` and the value `1`.
- Line 9: The contents of the dictionary is printed by passing its name to the `print` function.
- Line 10: The value that is associated with the key `"c"` is retrieved and passed to the `print` function.

Listing 15: Using and updating a dictionary.

```
1 data = {}
2 print(data)
3 data["a"] = 10
4 data["f"] = 15
5 print(data)
6 data.update({"b": 11, "c": 12})
7 print(data)
8 data["a"] = 1
9 print(data)
10 print(data["c"])
```

The structure of a dictionary is illustrated in Figure 2. A key is used to access the value that is associated with it.

## 6 Logic conditions

Python allows many types of logic conditions to be implemented. Each logic condition results in a `True` or `False` value being returned.

Listing 16 demonstrates a simple greater than condition. This program contains:

- Line 1: An integer variable named `x` is defined and given an initial value 2.
- Line 2: A logic condition is used to verify if the value of `x` is greater than 0. This logic condition returns `True`. The value `True` is passed to the `print` function, which prints `True` on the screen.

The variable `x` must be assigned a value before it is used within a logic condition. This value could be assigned somewhere else in the program, where the value might be less than or equal to zero instead.

Listing 16: A simple logic condition.

```
1 x = 2
2 print(x > 0)
```

Table 1: Comparison operators.

| Operator | Name |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

A list of comparison operators are given in Table 1.

Two logic conditions can be combined together using the AND (`and`) and OR (`or`) logic operators. The behaviour of the AND and OR logic operator is given in Table 2.

Table 2: The behaviour of the logic operators AND (`and`) and OR (`or`) for different input values

| Input 1 | Input 2 | Operator | Output |
|---|---|---|---|
| False | False | and | False |
| True | False | and | False |
| False | True | and | False |
| True | True | and | True |
| False | False | or | False |
| True | False | or | True |
| False | True | or | True |
| True | True | or | True |

A logic condition that includes two conditions is given in Listing 17. In this example program:

- Line 1: An integer variable named `x` is defined and given an initial value 2.
- Line 2: The comparison `x > 0` is evaluated and found to be `True`. The comparison `x != 2` is found to be `False`. The `True` and `False` Boolean values are combined by the AND (`and`) operator, resulting in `False`. The value `False` is passed to the `print` function.

Listing 17: Two logic conditions.

```
1 x = 2
2 print(x > 0 and x != 2)
```

The result of one logic condition can be converted from `True` to `False` or `False` to `True` by using the NOT (`not`) operator.

# 7 Conditional statements

The `if` keyword can be used to run an indented section of Python, when an associated Boolean value is `True`. This is illustrated in Listing 18.

Listing 18: A simple `if` statement.

```
1 if True:
2     print("This is always printed")
```

Instead of explicitly providing a Boolean value, a logic condition can be used with an `if` statement. This causes a piece of a program to run when the associated condition is `True`. This is demonstrated in Listing 19. In this example program:

- Line 1: An integer variable named `x` is defined and given an initial value 3.
- Line 2: Checks if the value of `x` is greater than zero.
- Line 3: This line is only executed if the logic condition at Line 2 is `True`. It prints some text that describes the logic condition.
- Line 4: Checks if the value of `x` is greater than zero and less than 10.
- Line 5: This line is only executed if the logic condition at Line 4 is `True`. It prints some text that describes the logic condition.
- Line 6: Checks if the value of `x` is equal to 3.
- Line 7: This line is only executed if the logic condition at Line 6 is `True`. It prints some text that describes the logic condition.
- Line 8: Checks if the value of `x` is not equal to 4.
- Line 9: This line is only executed if the logic condition at Line 8 is `True`. It prints some text that describes the logic condition.

Listing 19: Several `if` conditions.

```
1 x = 3
2 if x > 0:
3     print("x > 0")
4 if x > 0 and x < 10:
5     print("0 < x < 10")
6 if x == 3:
7     print("x == 3")
8 if x != 4:
9     print("x != 4")
```

The intended Python that follows an `if` statement can comprise one or many lines of Python.

In Listing 19, each `if` statement is evaluated separately. Instead of evaluating them separately, they can be part of an `if-else` structure. This is illustrated in Listing 20. In this program:

- Line 1: An integer variable named `x` is defined and given an initial value 3.
- Line 2: If the value of `x` is less than zero, the program runs Line 3 and then skips to Line 8.
- Line 4: The keyword `elif` implies “else if”. The program reaches this line if the condition at Line 2 is `False`. If the value of `x` is less than 10, the program runs Line 5 and then skips to Line 8.

- Line 6: The program reaches this line if the condition at Line 4 is `False`. It runs Line 7 and then Line 8.

Listing 20: Using `if`, `elif` and `else`.

```
1 x = 3
2 if x < 0:
3     print("x < 0")
4 elif x < 10:
5     print("0 < x < 10")
6 else:
7     print("x >= 10")
8 print("The rest of the program")
```

The intended Python that follows `if`, `elif` and `else` can comprise one or many lines of Python.

# 8 Loops

Loops allow programmers to design programs that process repetitive tasks, using a short program. Python supports `for` and `while` loops. These loops are executed zero or more times, depending on their associated conditions.

## 8.1 For loops

A `for` loop is given in Listing 21. In this program:

- Line 1: `range(5)` defines a series of integer values 0, 1, 2, 3, 4. The `for` loop takes the next value from `range(5)`. If there are no values left, the program skips to Line 3. If a value is available, it is put into the variable x. Then Line 2 is run.
- Line 2: The value of the variable `x` is printed. Then the program skips back to Line 1.

Listing 21: A `for` loop that iterates over values from `range`.

```
1 for x in range(5):
2     print(x)
3 print("After the loop")
```

A `for` loop is similar to someone dealing cards from a deck of playing cards. The dealer continues to provide cards until there are none left in the deck.

A `for` loop can be used to iterate over the elements within a list. This is demonstrated in Listing 22. Similar to Listing 21, the `for` loop continues to run until it has run out of list elements. It starts with the first list element, which is copied into the variable `value`. It continues to copy list elements into `value`, until there are none left.

Listing 22: A `for` loop that iterates over values from a list.

```
1 values = [2, 3, 4, 2]
2 for value in values:
3     print(value)
```

Rather than directly iterate over list elements, an index can be used within a `for` loop to select each element in turn. Listing 23 demonstrates this idea. In this program:

- Line 1: A list named `values` is defined that contains four values.
- Line 2: The length of the list `values` is assigned to the integer variable `n`.
- Line 3: The integer variable `n` is used with `range` to iterate over the list indices from 0 to `n-1`. If there are no more values, the program skips to Line 5. If there is another value, the program runs Line 4.
- Line 4: The value of `value` is printed. The program then skips back to Line 3.

Listing 23: A `for` loop that iterates over list index values.

```
1 values = [2, 3, 4, 2]
2 n = len(values)
3 for i in range(n):
4     print(values[i])
5 print("After the loop")
```

In some cases, action may be needed for specific list indices. In these cases, using the list index within the `for` loop may prove to be more efficient.

### 8.1.1 For loop with continue

In some cases, it is necessary to skip part of the program that is associated with a loop and continue to the next loop iteration. This is possible using `continue`, which is demonstrated in Listing 24. In this program:

- Line 1: A list named `values` is defined that contains four values.
- Line 2: The `for` loop takes the next list element or exits if there are no more list elements.
- Line 3: If the value from the list element is equal to 4, then Line 4 is executed.
- Line 4: The `continue` causes the program to skip back to Line 2.
- Line 5: The value in the list element is printed.

Listing 24: A `for` loop that demonstrates `continue`.

```
1 values = [2, 3, 4, 2]
2 for value in values:
3     if value == 4:
4         continue
5     print(value)
```

### 8.1.2 For loop with break

It may be necessary to break out of a loop, without reaching the end of the loop. This is possible using `break`, which is demonstrated in Listing 25. In this program:

- Line 1: A list named `values` is defined that contains four values.
- Line 2: The `for` loop takes the next list element or exits if there are no more list elements.
- Line 3: If the value from the list element is equal to 4, Line 4 is executed.

- Line 4: The `break` causes the program to exit the loop and skip to Line 6.
- Line 5: The value in the list element is printed.

Listing 25: A `for` loop that demonstrates `break`.

```
1 values = [2, 3, 4, 2]
2 for value in values:
3     if value == 4:
4         break
5     print(value)
```

## 8.2 While loops

A `while` loop continues to execute while its associated Boolean condition is `True`. The condition also needs to be true for the program to enter the while loop.

A `while` loop is demonstrated in Listing 26. In this program:

- Line 1: An integer variable `x` is defined that contains 0.
- Line 2: If the condition `x <= 3` is `False`, then the program skips to Line 5. If the condition `x <= 3` is `True`, then the program runs Line 3.
- Line 3: The value of the variable `x` is printed.
- Line 4: 1 is added to the current value of `x` and the result is assigned to `x`. The program then skips back to Line 2.

Listing 26: A `while` loop.

```
1 x = 0
2 while x <= 3:
3     print(x)
4     x += 1
5 print("After the loop")
```

### 8.2.1 While loop with continue

The keyword `continue` can be used to cause the program to skip back to the `while` condition. This is demonstrated in Listing 27. In this program:

- Line 1: An integer variable `x` is defined that contains 0.
- Line 2: If the condition `x <= 3` is `False`, then the program skips to Line 8. If the condition `x <= 3` is `True`, then the program runs Line 3.
- Line 3: If `x` is equal to 2, then Line 4 is run. If `x` is not equal to 2, then the program skips to Line 6.
- Line 4: 1 is added to the current value of `x` and the result is assigned to `x`.
- Line 5: The `continue` keyword causes the program to skip back to Line 2.
- Line 6: The value of the variable `x` is printed.
- Line 7: 1 is added to the current value of `x` and the result is assigned to `x`. The program then skips back to Line 2.

If Line 4 was deleted, the program would loop forever. This is because the value of `x` would not increase. Therefore, when the `continue` sends the program back to Line 2, the value of `x` would still be 2. Care needs to be taken when using `continue` within a `while` loop to avoid unwanted behaviour, such as infinite loops.

Listing 27: A `while` loop that demonstrates `continue`.

```
1 x = 0
2 while x <= 3:
3     if x == 2:
4         x += 1
5         continue
6     print(x)
7     x += 1
8 print("After the loop")
```

### 8.2.2 While loop with break

Sometimes it is necessary to enter a `while` loop and only break out of it if a condition is met within the loop. The `break` keyword can be used to perform this, as demonstrated in Listing 28. In this program:

- Line 1: An integer variable `x` is defined that contains 0.
- Line 2: The `while` loop condition is `True`. Therefore, the program runs Line 3.
- Line 3: The value of the variable `x` is printed.
- Line 4: If the value of `x` is greater than 3, then Line 5 is run. If the value is less than or equal to 3, then the program skips to Line 6.
- Line 5: The `break` keyword causes the program to skip to Line 7.
- Line 6: 1 is added to the current value of `x` and the result is assigned to `x`. The program then skips back to Line 2.

Listing 28: A `while` loop that demonstrates `break`.

```
1 x = 0
2 while True:
3     print(x)
4     if x > 3:
5         break
6     x += 1
7 print("After the loop")
```

# 9 Functions

A function can be thought of as a box that has zero or more inputs and an optional return value. Information may be passed in or returned from a function. A function should be used to encapsulate specific tasks that need to be repeated. For example, if data need to be loaded from a file, then a specific function for loading data might be needed. The name of a function should suggest what it is used for. For example, a data loading function might be called `load_data_sample`.

## 9.1 Return values

A function may or may not have a return value. If it does not declare a return value, then the return value is `None`. A simple function is given in Listing 29. In this program:

- Line 1: Defines a function called `get_number`. This function has no input arguments. Therefore, there is nothing between the parentheses `()`.
- Line 2: The function always returns the value `True`.
- Line 3: An empty line needed to separate the function from the rest of the program.
- Line 4: An empty line needed to separate the function from the rest of the program.
- Line 5: Calls the function `get_number` and assigns the return value to a variable named `x`.
- Line 6: Prints the value of `x` on the screen, using the `print` function.

Listing 29: A function with no input arguments and a return value.

```
1 def get_number():
2     return True
3
4
5 x = get_number()
6 print(x)
```

When a function returns a value, it is passed back to the calling program. If it is not assigned to a variable, then the value is disposed of.

If a return value is not assigned to a variable and the program is run interactively, the value is printed on the screen. This is a special feature of an interactive Python session. Using Python interactively is discussed in Section 3.1.

## 9.2 Input arguments

One or more input arguments may be passed into a function. The function receives these input arguments, performs some actions and may return a resulting value. Listing 30 illustrates how to define a function with two input arguments and a return value. In this program:

- Line 1: Defines a function called `multiply`. This function has two input arguments that are named `x` and `y`.
- Line 2: The value of `x` and `y` are multiplied together. The function returns the result of the multiplication.
- Line 3: An empty line needed to separate the function from the rest of the program.
- Line 4: An empty line needed to separate the function from the rest of the program.
- Line 5: Calls the function `multiply` and assigns the return value to a variable named `result`.
- Line 6: Prints the value of `result` on the screen, using the `print` function.

Listing 30: A function with two input arguments and a return value.

```
1 def multiply(x, y):
2     return x*y
3
4
```

```
5 result = multiply(2, 3)
6 print(result)
```

In Listing 30, the variables `x` and `y` become integer variables since they are passed integer values 2 and 3, respectively. When an integer is passed into the function, the value is copied into the variable that is defined within the function. For example, the value 2 is copied into the variable `x`.

The variable `x` exists inside the function `multiply`, but does not exist outside of it. At the end of a function call, any variables that were defined when it was called are by default deleted and removed from the computer's memory.

# 10 Immutable and mutable

Python variables are either immutable or mutable. An immutable variable is defined in memory, used and then automatically deleted. A mutable variable is defined and can then be modified elsewhere within a program. Integers, floats, strings and boolean values are all immutable. Lists and dictionaries are mutable.

## 10.1 Assigning a reference

When an immutable variable is assigned to another variable, the value is copied from one variable to another. When a mutable variable is assigned to another variable, a reference to the original variable is assigned. The effects of this are demonstrated in Listing 31. In this program:

- Line 1: A variable named `x` is defined and assigned the value 1.
- Line 2: A variable named `y` is defined and assigned the value that is stored in `x`. The value is copied from `x` to `y`. The variable `x` still exists in memory and is separate from the variable `y`.
- Line 3: The value stored in `y` is incremented by 1.
- Line 4: The value stored in `x` is printed. The value in `x` is 1, since the variable `y` only received a copy of the value that was in `x`.
- Line 5: A list named `values` is defined with two elements.
- Line 6: A reference to `values` is assigned to `more_values`. This reference is similar to a sign post, which points at the original memory that is associated with the list `values`. The variable `more_values` is a reference to `values`. Therefore, when `more_values` is updated, `values` is updated.
- Line 7: An element is appended to the list `more_values`, which updates the number of elements in `values`.
- Line 8: The contents of the list `values` is printed. The list contains `[1, 2, 3]`, since a reference was assigned to `more_values`.

Listing 31: Assigning a value and a reference.

```
1 x = 1
2 y = x
3 y += 1
4 print(x)
5 values = [1, 2]
6 more_values = values
7 more_values.append(3)
8 print(values)
```

## 10.2 Breaking a reference link

If a mutable instance is assigned to a variable and a new instance is then assigned, the reference to the first instance is overridden. This is demonstrated by Listing 32. In this program:

- Line 1: A list named `values` is defined with two elements.
- Line 2: A reference to `values` is assigned to `more_values`.
- Line 3: A reference to a new list is assigned to `more_values`.
- Line 4: An element is appended to the list `more_values`.
- Line 8: The contents of the list `values` is printed. The list contains `[1, 2]`.

Listing 32: Breaking a reference link

```
1 values = [1, 2]
2 more_values = values
3 more_values = []
4 more_values.append(3)
5 print(values)
```

## 10.3 Mutable arguments

When mutable instances are passed as arguments of a function, the function may update the original instance since it is operating on a reference. This is demonstrated by Listing 33. In this program:

- Line 1: A function named `update_lists` is defined that has two arguments and no return value.
- Line 2: One element is added to the list `numbers`. Since `numbers` is assigned a reference to `values` at Line 9, when `numbers` is updated `values` is updated.
- Line 3: A reference to a new list is assigned to `more_numbers`. The variable `more_numbers` is assigned a reference to `more_values` at Line 9. However, the assignment of a reference to new list at Line 4 breaks this connection.
- Line 4: One element is added to the list `more_numbers`.
- Line 5: An empty line needed to separate the function from the rest of the program.
- Line 6: An empty line needed to separate the function from the rest of the program.
- Line 7: A list named `values` is defined with two elements.
- Line 8: A list named `more_values` is defined with two elements.
- Line 9: References to the lists `values` and `more_values` are passed to the function `update_lists`.
- Line 10: The contents of the list `values` is printed. The list contains `[1, 2, 3]`.

- Line 10: The contents of the list `more_values` is printed. The list contains `[1, 2]`.

Listing 33: A function with mutable arguments.

```
1  def update_lists(numbers, more_numbers):
2      numbers.append(3)
3      more_numbers = []
4      more_numbers.append(3)
5
6
7  values = [1, 2]
8  more_values = [1, 2]
9  update_lists(values, more_values)
10 print(values)
11 print(more_values)
```

## 11 Comments

Comments should be used to explain the purpose of a Python program. A Python file or module should include a comment that describes the module. There should be a comment that explains the purpose of each program, function or class. Comments might be needed to explain particular lines within a program. These comment styles are demonstrated in Listing 34. In this program:

- Line 1 to 4: A multiline comment used to describe the module as a whole. The multiline comment starts and ends with `"""`.
- Line 5 and 6: Empty lines after the module comment, which are required by the coding standard used for these examples.
- Line 7: A function definition that has two arguments.
- Line 8 to 11: A multiline comment that describes the purpose of the function. The multiline comment starts and ends with `"""`.
- Line 12: The return value of the function.
- Line 15: A single-line comment that describes the purpose of one or more steps of the program.

Listing 34: Python module, function and single line comments.

```
"""
This example module demonstrates different comments types that
are available in Python.
"""


def multiply(x, y):
    """
    A function that multiplies two values together and returns
    the result.
    """
    return x*y


# Multiply two numbers together and print the result.
result = multiply(2, 3)
print(result)
```

Comments should not explain the syntax of the program, but tell the reader why the program has been written. The reader should be able to read the Python program syntax.

## 12 Coding standards

The Python interpreter allows Python to be written with different text formatting. For example, a developer could use two, three, four or some other number of spaces to indent lines of Python. A developer could name variables or classes with a mixture of upper and lower case letters. Class members and functions can also be named with a mixture of upper and lower case letters.

To provide other developers with a clearer understanding of the function of a program, coding standards are often used. The pep8 coding standard is followed for this document. More details on pep8 are given at https://pep8.org/.

## 13 Further reading

This document provides a basic introduction to programming the in Python programming language. More information can be found in:

- W. H. Bell, “Python programming examples”.
- https://docs.python.org/3/
- https://www.w3schools.com/python/

# 14 Glossary

- **Accessor function** - An accessor function is used to get or set the value of private or protected class data members. Accessor functions should be used sparingly with the Python programming language, since the Python interpreter does not reduce their associated processing overheads.
- **Argument** - Used when referring to program and function inputs or outputs. For example, a function can be said to have no input arguments, implying that nothing is passed into the function. Alternatively, a function can be described as having two input arguments, implying that two values are separately passed into the function. Programs may have zero or more input arguments that are supplied at a command line. These values are passed into a program.
- **Bug** - A software defect. It is feature of software that is not intended by the program designer. The bug could be a simple feature that causes a bad value to be returned, the program to behave erratically or crash.
- **Call** - A function call is where the program execution runs a function. The computer jumps to another part of the program to run the function and then returns to the program where it was before the function call.
- **Cast or casting** - The process of converting a value from one type to another. For example, a text string can be cast into an integer value.
- **Class** - A structure that contains zero or more data members and zero or more member functions. A class is normally written such that state is stored within data members that are associated with functions in the class. An object is created using the class name to call the class constructor function.
- **Coding standard** - A set of rules that govern the naming and layout of a computer programming language. The purpose of a coding standard is to provide uniformity, which allows other developers to more quickly understand the structure of a computer program.
- **Debugging** - The process of stepping through the execution of the program and verifying its functionality. When a bug is present within a program, debugging is often required to find it and understand its features.
- **Element** - A list, tuple and array are made up of elements. Each element is one piece of memory. Each element is referred to using an index, which corresponds to the offset within the computer's memory.
- **Execution** - The process of running a program.
- **Float** - A floating point number. A floating point number is a number that may have a decimal fraction associated with it. For example, `3.14159` is a floating point number.
- **Function** - A structure that is used to encapsulate one or more lines of Python. The function may have zero or more input arguments and one or zero return values.
- **Integer** - A whole number. For example, `3` is an integer number.
- **Integrated development environment (IDE)** - A text editor that provides integration with a programming language interpreter or compiler. The IDE may provide a connection to debugging tools. It may also support syntax checks and suggestions that are made available to a developer.
- **Immutable** - The state cannot be changes after it has been created. If a variable is immutable, then the value is passed into a function. Likewise, when it is assigned to another variable, the value is assigned.

- **Interpreter** - A program that reads lines of instructions and performs some functions in response to the instructions that are provided.
- **Loop** - A program continues to run over the contents of a loop, while the condition that is associated with the loop is true. It is said to be a loop, since the execution of the program loops back to the beginning of the loop. For and while are both types of loop.
- **Module** - A module comprises one or more Python files. A module can be imported, such that associated data, functions and classes can be accessed.
- **Mutable** - The state can be changed after it has been created. If a variable is mutable, then a reference is passed into a function. Likewise, when it is assigned, a reference is assigned rather than a value.
- **Mutator function** - An mutator function is used to set the value of private or protected data members within an object. Mutator functions should be used sparingly with the Python programming language, since the Python interpreter does not reduce their associated processing overheads.
- **Nesting** - Where one structure is inside another one. For example, an `if` statement could include another `if` statement or a `for` loop.
- **Object** - Objects are created as instances of a given class. Two objects do not share the same memory location.
- **Operating system (OS)** - Software that runs on a computer and manages computer hardware and software. It supports the running of other applications on the computer and provides a user interface.
- **Operator** - An operator operates on values that are stored in variables. For example, to multiply two values together the multiplication operator is used. The multiplication operator performs the mathematical multiplication and returns the result.
- **Operator overloading** - The process of writing functions for a class that allow objects to be used with operators.
- **Present working directory** - This is the directory that the operating system is in when the program runs. This directory is normally associated with a process window that is used to run a Python program.
- **String** - In computer programming language text is stored within “string” variables. A string is a series of text characters. The text characters can be directly accessed, using index syntax that is similar to that used with a list.
- **Reference** - A reference refers to a memory location that has been defined elsewhere in a program. For example, a reference to a list can be assigned to a variable. When the list is updated through the reference the original list is updated.
- **Type** - A type refers to the nature of a variable. For example, a variable may be an integer. In this case, the variable is said to be of type integer.
- **User interface (UI)** - A means for a user to interact with a device. This could be a graphical user interface that comprises windows, icons and buttons, or a text interface that requires the user to provide text input. The user interface could be via a remote connection to the local machine, rather than directly to it.
- **Variable** - A section of the computers memory that can be used to hold a value or reference. The size of the variable will depend on the type of data that is associated with it.