University of Strathclyde Glasgow

Python programming examples

W. H. Bell

November 14, 2023

Abstract

The Python programming language is discussed using a series of example programs. These example programs introduce concepts, such as data types, functions, classes, mathematical and logical operators, and input and output functionality. The document uses simple examples and provides a systematic discussion of their functionality.

Contents

1 Introduction

This document discusses concepts and functionality of the Python programming language through a series of worked examples. The document assumes that the reader is familiar with computer programming. Readers who are not familiar with computer programming should read “An introduction to Python for those new to computer programming” before continuing to read the rest of this document.

A computer programming language is only properly understood by experimentation and writing new programs. The examples that are given in this document should be run, debugged and modified to better understand the Python programming language.

A glossary of terms that are used in this document is given in Section 24.

2 Further reading

Further reading is available at:

- W. H. Bell, “An introduction to Python for those new to computer programming”.
- https://www.w3schools.com/python/
- https://www.tutorialspoint.com/python3/index.htm
- https://docs.python.org/3/

It may aid the reader to refer to these references for comparison or further discussion of some of the concepts that are introduced in this document.

3 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.

3.1 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.

4 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.

4.1 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`.

4.2 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")
```

5 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/.

6 Variable types

6.1 Type from assignment

When a Python variable is first assigned a value, the type of the variable is set by the type of the value. Some examples are given in Listing 4. In this Listing:

- Line 1: An integer (whole number) is assigned to the variable `x`. Therefore, `x` is an integer variable.
- Line 2: A float (number with decimal fraction) is assigned to the variable `y`. Therefore, `y` is a float variable.
- Line 3: A string (series of text characters) is assigned to the variable `text`. Therefore, `text` is a string.
- Line 4: A Boolean value (True or False) is assigned to the variable `b`. Therefore, `b` is a Boolean.
- Line 5: A list that contains three elements is assigned to the variable `values`. Therefore, `values` contains a reference to a list. A list is a sequential data container that can contain zero or more elements.
- Line 6: A dictionary that contains two key and value pairs is assigned to the variable `data`. Therefore, `data` contains a reference to a dictionary. A dictionary is an associative container. The key is before the ":" and is used to access the value that follows the ":".

Listing 4: Defining variable types by the assignment of values.

```
1 x = 2
2 y = 1.2
3 text = "Text"
4 b = True
5 values = [1, 2, 3]
6 data = {2: "d", 4: "z"}
```

6.2 Mutability

Depending on their type, variables are immutable or mutable. The mutability of some common variable types is given in Table 1.

Table 1: Mutability of variable types.

Variable type	Mutability
Boolean	Immutable
Float	Immutable
Integer	Immutable
String	Immutable
Tuple	Immutable
Dictionary	Mutable
List	Mutable
Set	Mutable

The memory that is associated with an immutable variable cannot be changed, whereas the memory that is associated with a mutable variable can be changed. When an immutable variable is assigned,

the value is assigned. When a mutable variable is assigned, a reference is assigned to the original version of the data structure. This can be illustrated with an integer and a list, as demonstrated in Listing 5. In this Listing:

- Line 1: An integer variable named `x` is defined.
- Line 2: The value that is stored in `x` is assigned to the variable `y`.
- Line 3: The value that is stored in `y` is incremented by 1.
- Line 4: The value that is stored in `x` is printed. It contains 1, since the value was assigned to `y` rather than a reference.
- Line 5: A variable named `values` is defined, which is assigned the reference to a list that has two elements.
- Line 6: A reference to the list `values` is assigned to the variable `more_values`.
- Line 7: An additional list element is appended to `more_values`. This updates the list that the variable `values` also refers to.
- Line 8: The elements in the list `values` are printed. There are three elements in the list that the variable `values` refers to.

Listing 5: Assigning a value and a reference.

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

7 Operators

Python supports numeric and bitwise operators that can be used to operate on numeric or boolean data types. The addition operator can be used with strings to append strings together.

7.1 Assignment operator

The assignment operator = can be used with all types, where its meaning depends on the type used. For immutable types, the assignment operator copies the value into another variable. For mutable types, such as lists, dictionaries and objects that are mutable, the assignment operator copies a reference to the original variable.

7.2 Numeric operators

Numeric operators can be used with both floating point and integer numbers. The list of available operators is summarised in Table 2.

Table 2: Numeric operators.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation (power)
//	Floor division

Some of the available numeric operations are demonstrated in Listing 6. In this Listing:

- Line 1: A variable named `n_bits` is defined and assigned 8.
- Line 2: A variable named `n_bytes` is defined and assigned the value that is returned from `n_bits`/8.
- Line 3: The value of `n_bytes` is printed.
- Line 4: The result of $2^8 - 1$ which is 255 is assigned to a variable named `max_value`.
- Line 5: The value of `max_value` is printed.

Listing 6: Numeric operators.

```
1 n_bits = 8
2 n_bytes = n_bits/8
3 print(n_bytes)
4 max_value = 2**n_bits - 1
5 print(max_value)
```

7.3 Comparison and logical operators

Comparison and logical operators can be used with numeric and boolean types, where a subset of them are useful with boolean types. The comparison and logical operators are summarised in Table 3.

Some of the comparison and logical operators are demonstrated in Listing 7. In this Listing:

- Line 1: A variable named `x` is defined and assigned the value 3.
- Line 2: A variable named `y` is defined and assigned the value 2.

Table 3: Comparison and logical operators.

Operator	Name
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
and	True if both statements are true
or	True if one of the statements is true
not	Returns the opposite of the statement state

- Line 3: The result of the comparison `x > y` is assigned to `result`. The result of the comparison is the Boolean value `True`.
- Line 4: The value that is stored in `result` is printed.
- Line 5: The result of the comparison `x != y and x < 0` is assigned to `result`. The result of the comparison is the Boolean value `False`.
- Line 6: The value that is stored in `result` is printed.

Listing 7: Comparison and logical operators.

```
1 x = 3
2 y = 2
3 result = x > y
4 print(result)
5 result = x != y and x < 0
6 print(result)
```

7.4 Identity and membership operators

Identity and membership objects are useful when operating on lists, dictionaries or other objects. For example, searching for a value within a list can be achieved using the membership operator `in`. The identity and membership operators are summarised in Table 4.

Table 4: Comparison and logical operators.

Operator	Name
is	True if both variables are same object
is not	True if both variables are not the same object
in	True if found in object
not in	True if not found in object

The two membership operators are demonstrated in Listing 8. In this Listing:

- Line 1: A variable named `values` is defined and assigned a reference to a list that contains three values.
- Line 2: The `not in` membership operator is used to verify that the value 2 is not in the list. The variable `result` is assigned `True`, since the value 2 is not in the list `values`.
- Line 3: The value of `result` is printed.
- Line 4: The `in` membership operator is used to verify that the value 9 is in the list. The variable `result` is assigned `True`, since the value 9 is in the list `values`.

Listing 8: Membership operators.

```
1 values = [1, 4, 9]
2 result = 2 not in values
3 print(result)
4 result = 9 in values
5 print(result)
```

7.5 Bitwise operators

Bitwise operators can be used with numeric and boolean types, where a subset of them are applicable to booleans. The bitwise operators are summarised in Table 5.

Table 5: Bitwise operators.

Operator	Name
&	AND
|	OR
^	Exclusive OR (XOR)
~	NOT
«	Left shift
»	Right shift

Some of the bitwise operators are demonstrated in Listing 9. In this Listing:

- Line 1: A variable named `bits` is assigned 20. The value 20 comprises $2^2 + 2^4$. The first binary bit is 2^0. Therefore, a value of 20 implies that the third and fifth binary bit are `True`.
- Line 2: A variable named `bit` is assigned 1.
- Line 3: The expression `bit & bits` is 0, since the value 20 does not include 2^0. Therefore, the `result` variable is assigned `False`.
- Line 4: The value of `result` is printed.
- Line 5: The value of `bit` is left shifted by two places, from 2^0 to 2^2.
- Line 6: The expression `bit & bits` is 4, since the value 20 does include 2^2. Therefore, the `result` variable is assigned `True`.

- Line 7: The value of `result` is printed.
- Line 8: The value of `bit` is right shifted by one place, from 2^2 to 2^1.
- Line 9: The expression `bit | bits` is 22, since it is the sum of 2^1, 2^2 and 2^4. Therefore, the `result` variable is assigned `True`.
- Line 10: The value of `result` is printed.

Listing 9: Bitwise operators.

```
1  bits = 20
2  bit = 1
3  result = (bit & bits) > 0
4  print(result)
5  bit = bit << 2
6  result = (bit & bits) > 0
7  print(result)
8  bit = bit >> 1
9  result = (bit | bits) > 0
10 print(result)
```

7.6 Assignment operator short versions

Operations that involve updating a variable can be implemented using an abbreviated version of Python syntax. The abbreviated versions and their longer equivalent forms are summarised in Table 6.

Table 6: Abbreviated and equivalent assignment operators.

Abbreviated operator	Equivalent operator
`variable += value`	`variable = variable + value`
`variable -= value`	`variable = variable - value`
`variable *= value`	`variable = variable * value`
`variable /= value`	`variable = variable / value`
`variable %= value`	`variable = variable % value`
`variable **= value`	`variable = variable ** value`
`variable //= value`	`variable = variable // value`
`variable &= value`	`variable = variable & value`
`variable |= value`	`variable = variable | value`
`variable ^= value`	`variable = variable ^ value`
`variable >>= value`	`variable = variable >> value`
`variable <<= value`	`variable = variable << value`

Unlike some other programming languages, Python does not allow `variable++` or `variable-`.

8 Copying mutable variables

As discussed in Section 7.1, when the assignment operator is used with a mutable variable it copies a reference to the original variable into another variable. Instead of copying a reference, the `copy` function can be used to create a separate copy of an input mutable variable. Mutable variables can either be copied using a shallow or deep copy. A shallow copy implies that the top-level mutable variable is copied, whereas any mutable variables inside are not copied. Alternatively, a deep copy copies all layer of a mutable variable.

The `copy` and `deepcopy` functions are demonstrated in Listing 10. In this Listing:

- Line 1: The `copy` module is imported, such that it can be used at Line 10 and Line 11.
- Line 3: A list is created that contains another list in the first two elements and an integer in the third element.
- Line 9: A shallow copy is made of the list, which is equivalent to Line 10.
- Line 10: A shallow copy is made of the list.
- Line 11: A deep copy is made of the list and the lists inside the list.
- Line 13: The value in the first element of the list in `more_values[0]` and `values[0]` is updated. This occurs since the first element of `more_values` contains a reference to the list that is in the first element of `values`.
- Line 14: The value in the third element of `more_values` is updated. This has no effect on `values`, since the integer in `more_values[2]` is a copy of the integer in `values[2]`.
- Line 15: The value in the first element of the list in `other_values[0]` is updated. This has no effect on `values`, since `other_values` is a completely separate copy of `values`.
- Line 16: The value in the third element of `other_values` is updated. This has no effect on `values`, since the integer in `other_values[2]` is a copy of the integer in `values[2]`.

Listing 10: Shallow and deep copies of mutable variables.

```
 1 import copy
 2
 3 values = [
 4     [1, 2],
 5     [3, 4],
 6     5
 7 ]
 8
 9 new_values = values.copy()
10 more_values = copy.copy(values)
11 other_values = copy.deepcopy(values)
12
13 more_values[0][0] = 8
14 more_values[2] = 9
15 other_values[0][0] = 10
16 other_values[2] = 11
17
18 print(values)
19 print(more_values)
20 print(other_values)
```

9 Conditional statements

Conditional statements are used to control the flow of a program. For example, if an input changes, then a program might behave differently. The `if`, `elif` and `else` keywords are used to construct conditional statements. Conditional statements can be very long or nested several layers deep. To allow developers to understand programs more easily and debug them, conditional statements should use a minimum depth of nesting and be as short as possible.

The `if`, `elif` and `else` keywords are demonstrated in Listing 11. In this Listing:

- Line 1: A variable named `x` is defined and assigned 2.
- Line 2: A variable named `y` is defined and assigned 3.
- Line 3: This condition is `False`, since `x` contains 2.
- Line 5: This condition is `False`, since `y` contains 3.
- Line 7: The programs reaches the `else`, since the previous conditions were `False`.
- Line 8: The condition that was fulfilled is printed.

Listing 11: Conditional statements.

```
1 x = 2
2 y = 3
3 if x < 0:
4     print("x < 0")
5 elif y < 0:
6     print("x >= 0 and y < 0")
7 else:
8     print("x >= 0 and y >= 0")
```

The `if` keyword can be used on its own, without `elif` and `else`. The `elif` keyword may follow another `elif` or a `if`, where there must be an `if` before the first `elif`. The `else` keyword may follow an `elif` or an `if`, where there must be an `if` before the `else` and there can only be one `else` keyword within a conditional expression.

10 Loops

Loops can be used to perform repetitive tasks. For example, many files may need to be treated in the same way, running similar functionality for each file. Loops are especially useful when used with sequential data containers, such as lists. A loop can process each list element, re-using the program logic to do so.

10.1 For loops

For loops function rather like the dealer of a deck of playing cards. The `for` loop is given a list or an object that can be treated in the same way as a list and iterated over. The `for` loop continues to run, while there are elements in the input data. Once there are no more elements, the `for` loop finishes. If there are no elements in the input data to start with, then the execution does not go into the `for` loop.

A `for` loop is demonstrated in Listing 12, which calculates the factorial of the value that is stored in the variable `x`. In this Listing:

- Line 1: A variable named `x` is defined and assigned 3.
- Line 2: A variable named `result` is defined and assigned 1. It is assigned 1, since 0 factorial is 1.
- Line 3: A `range(x)` is defined, which generates the series of numbers 0, 1 and 2. The `for` loop copies 0 into `value` the first time that the program reaches Line 3, then 1 and finally 2. When there are no more numbers, the program skips to Line 5.

- Line 4: This line is indented and is therefore executed each time the `for` loop takes a value from the `range`. One or more lines must be indented, such that they are associated with a `for` loop. The result of the addition `value + 1` goes from 1 to 3, as the `for` loop takes values from the `range`. The result of the addition is multiplied with the number that is already in the variable `result` and is then assigned to `result`.
- Line 5: The result of `x` factorial is printed on the screen.

Listing 12: Using a `for` loop to calculate the factorial of `x`.

```
1 x = 3
2 result = 1
3 for value in range(x):
4     result *= value + 1
5 print(result)
```

Similar to conditional statements, `for` loops can be nested within other loops or conditional statements. To allow a developer to understand and debug a program, the minimal level of nesting should be used.

The keywords `continue` and `break` can be used with `for` loops to skip to the next item in the series and break out of the loop, respectively. This is demonstrated in Listing 13. In this Listing:

- Line 1-8: A list named `fruits` is defined that contains a series of fruits.
- Line 9: The `for` loop iterates over the fruits, copying each of the names into the variable `fruit`.
- Line 10: If the fruit name is `"kumquat"`, then Line 11 is executed.
- Line 11: The `continue` keyword causes the program to skip to Line 9.
- Line 12: If the fruit name is `"kiwi"`, then Line 13 is executed.
- Line 13: The `break` keyword causes the program to skip to after Line 14.
- Line 14: The value in `fruit` is printed to the screen.

Listing 13: Using `continue` and `break` with a `for` loop.

```
 1 fruits = [
 2     "pear",
 3     "apple",
 4     "kumquat",
 5     "banana",
 6     "kiwi",
 7     "orange"
 8 ]
 9 for fruit in fruits:
10     if fruit == "kumquat":
11         continue
12     if fruit == "kiwi":
13         break
14     print(fruit)
```

Due to the use of the keywords `continue` and `break`, Line 14 only prints `pear`, `apple` and `banana`.

10.2 While loops

While loops continue to loop while an associated Boolean condition is `True`. If the condition is not `True`, then the program will not enter the loop. If the condition is true, then the program will enter the loop. Once the condition becomes not `True`, the while loop will exit.

A `while` loop is demonstrated in Listing 14. This program calculates the factorial of the variable `x`, similar Listing 12. In the example `while` loop program:

- Line 1: A variable named `x` is defined and assigned 3.
- Line 2: A variable named `result` is defined and assigned 1. It is assigned 1, since 0 factorial is 1.
- Line 3: The program enters the `while` loop, since `x > 0` is `True`. The `while` loop continues, while `x > 0` is `True`.
- Line 4: The current value of `result` is multiplied by `x` and the output value is assigned to `result`.
- Line 5: The current value of `x` is decremented by 1.
- Line 6: The result of `x` factorial is printed on the screen.

Listing 14: Using a `while` loop to calculate the factorial of `x`.

```
1 x = 3
2 result = 1
3 while x > 0:
4     result *= x
5     x -= 1
6 print(result)
```

The keywords `continue` and `break` can be used with a `while` loop too. However, care is needed when using `continue` to avoid an infinite loop.

11 Exceptions

When an operation occurs that is not expected, a function may throw an exception. Exceptions are thrown by standard functions, but can also be thrown by other functions that are developed. These exceptions have a type that can be used to identify them. When writing robust software, expected exceptions should be caught. If an exception is thrown that is not expected, the program should exit such that a developer can investigate. Therefore, unexpected exceptions should not be caught within developed software.

11.1 Exception when casting values

When a string variable is converted into an integer value, the function may throw a `ValueError` exception. This occurs when the string value is not a valid integer and cannot be converted into an integer value. This is demonstrated in Listing 15. In this Listing:

- Line 1: A string variable named `text` is assigned a string that contains numbers but also contains a comma.
- Line 2: The program enters the `try` section.
- Line 3: The function `int` converts the variable `text` from a string to an integer value. The function throws an exception, since the string contains a comma. The program skips to Line 5.
- Line 4: Is not run, since `text` contains a comma.
- Line 5: The exception thrown is a `ValueError`. Therefore, the program proceeds to Line 6.
- Line 6: A message is printed on the screen.
- Line 7: A message is printed on the screen.

Listing 15: Catching a `ValueError` exception.

```
1 text = "1234,"
2 try:
3     value = int(text)
4     print(value)
5 except ValueError:
6     print("Cannot cast text to an integer.")
7 print("Finished.")
```

11.2 Out of range index

An integer number can be used to select an element within a list. The elements in the list are numbered from 0 to $n-1$, where n is the number of elements in the list. If an index is used that is outside of this range, an `IndexError` is thrown. This is demonstrated in Listing 16. In this Listing:

- Line 1: A list named `values` is defined that has three elements.
- Line 2: An integer variable `i` is assigned 3.
- Line 3: The program enters the `try` section.
- Line 4: This line attempts to retrieve the 4th element, since the value of `i` is 3. An `IndexError` exception is thrown and the program skips to Line 6.
- Line 5: This line not run, since the exception causes the program to skip to Line 6.
- Line 6: The exception is an `IndexError`. Therefore, the program proceeds to Line 7.
- Line 7: A message is printed on the screen.
- Line 8: A message is printed on the screen.

Listing 16: Catching an `IndexError` exception.

```
1 values = [2, 41, 3]
2 i = 3
3 try:
4     value = values[i]
5     print(value)
6 except IndexError:
7     print("The index is out of range.")
8 print("Finished")
```

11.3 Unknown dictionary key

A dictionary contains unique keys, together with associated values. A key can be used to access the associated value. If the key is not present within the dictionary, then a `KeyError` exception is thrown. This is demonstrated in Listing 17. In this Listing:

- Line 1: A dictionary named `data` is defined that has two key and value pairs.
- Line 2: An integer variable `key` is assigned 3.
- Line 3: The program enters the `try` section.
- Line 4: This line attempts to retrieve a value, using the value of `key` as the key. Since a key with value 3 does not exist in the dictionary, a `KeyError` is thrown and the program proceeds to Line 6.
- Line 5: This line not run, since the exception causes the program to skip to Line 6.
- Line 6: The exception is an `KeyError`. Therefore, the program proceeds to Line 7.
- Line 7: A message is printed on the screen.
- Line 8: A message is printed on the screen.

Listing 17: Catching a `KeyError` exception.

```
1 data = {2: "Address", 4: "Name"}
2 key = 3
3 try:
4     value = data[key]
5     print(value)
6 except KeyError:
7     print("The key is not in the dictionary.")
8 print("Finished.")
```

12 Mathematics

There are several mathematical libraries that can be used with the Python programming language. In this section, some of the standard libraries are introduced.

12.1 Math module

The `math` module contains standard mathematical functions, such as trigonometric and log functions. Listing 18 demonstrates how to sample a sine wave with the math library. In this Listing:

- Line 1: The `math` module is imported.
- Line 2: A variable named `x_min` is assigned 0.
- Line 3: A variable named `x_max` is assigned 2π.
- Line 4: A variable named `n_steps` is assigned 20.
- Line 5: The size of the step along the x-axis is calculated and assigned to `x_step`.
- Line 6: A variable named `x` is assigned the value that is stored in `x_min`.
- Line 7: The `while` loop continues, until the value of `x` is greater than or equal to `x_max`.

- Line 8: The position on the y-axis is calculated as $\sin(x)$.
- Line 9: The value of `x` and `y` is printed on the screen.
- Line 10: The value of `x` is incremented by `x_step`. The program then skips back to Line 7.

Listing 18: Sampling a sine wave.

```
1 import math
2 x_min = 0
3 x_max = 2*math.pi
4 n_steps = 20
5 x_step = (x_max-x_min)/n_steps
6 x = x_min
7 while x < x_max:
8     y = math.sin(x)
9     print(x, y)
10    x += x_step
```

12.2 Random module

Random numbers are useful when working with probabilities and simulations of discrete events. The `random` module provides basic pseudorandom number generation functions. Listing 19 demonstrates two of the random number generation functions that are part of the `random` module. In this Listing:

- Line 1: The `random` module is imported.
- Line 2: The random number seed is set to 12345. Random numbers that are generated by a computer are said to be pseudorandom, since they are part of a series. The series starts with an input seed. When the same random number seed is used, the same set of random numbers is generated.
- Line 3: An integer random number is generated within the set $\{1, ..., 6\}$. The value is allocated to the variable `die`.
- Line 4: The value of `die` is printed on the screen.
- Line 5: A random number is generated between 0 and 1, where the probability of a number being selected is uniform between 0 and 1. The result is assigned to the variable named `flat`.
- Line 6: The value of `flat` is printed on the screen.

Listing 19: Using random functions.

```
1 import random
2 random.seed(12345)
3 die = random.randint(1, 6)
4 print(die)
5 flat = random.uniform(0., 1.)
6 print(flat)
```

12.3 Statistics module

When performing data analysis, it is often necessary to examine the basic statistical behaviour of distributions. The `statistics` module contains functions to calculate the mean, median, variance and standard deviations. Listing 20 demonstrates how to use two of these functions. In this Listing:

- Line 1: The `statistics` module is imported.
- Line 2: A list named `values` is created with five values.
- Line 3: The mean of the values that are stored in `values` is calculated and assigned to a variable named `mean_value`.
- Line 4: The sample standard deviation is calculated for the values that are stored in `values`, where the result is assigned to the variable `stdev_value`.
- Line 5: The value of `mean_value` and `stdev_value` is printed on the screen.

Listing 20: Using statistical functions.

```
1 import statistics
2 values = [2.1, -0.2, 1.2, 3.2, -2.3]
3 mean_value = statistics.mean(values)
4 stdev_value = statistics.stdev(values)
5 print(mean_value, stdev_value)
```

13 String operations

One of the most powerful things about the Python programming language is its string functions. This document introduces some of string operations that are available.

13.1 Appending to a string

Once a string variable has been created, another string can be appended or prepended to it. The resulting value can then be assigned to the original string or another string. Listing 21 demonstrates these operations. In this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python"`.
- Line 2: The string `" is great"` is appended to the value that is stored in the variable `text` and the result is assigned to the variable `text`. The "+" operator is used to append one string to another string. As discussed in Section 7.6, `text +=` is equivalent to `text = text +`.
- Line 3: The string `"»"` is prepended to the value that is stored in the variable `text` and the result is assigned to the variable `text`.
- Line 4: The value of `text` is printed on the screen.

Listing 21: Appending strings.

```
1 text = "Python"
2 text += " is great"
3 text = ">>" + text
4 print(text)
```

13.2 Accessing substrings

A string is a series of text characters. The text characters or substrings can be accessed in the same manner as list entries. This is demonstrated in Listing 22. n this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python"`.
- Line 2: The length of the string is returned by using the `len` function, where the result is stored in the variable named `n`.
- Line 3: The number of characters within the string is printed, together with some description. The "+" operator is used to combine the strings before they are passed to the `print` function.
- Line 4: The first two characters are printed together with some description. The syntax `text[0:2]` implies that characters 0 and 1 are returned in the substring.
- Line 5: The last two characters are returned together with some description. The syntax `text[n-2:]` implies that the substring starts from the penultimate character and continues to the end of the string, where the absence of a number after the colon indicates that the substring continues to the end of the string from the starting position.

Listing 22: Accessing part of a string.

```
1 text = "Python"
2 n = len(text)
3 print("\"" + text + "\" contains " + str(n) + " characters.")
4 print("First two characters: \"" + text[0:2] + "\"")
5 print("Last two characters: \"" + text[n-2:] + "\"")
```

13.3 Matching with a substring

The value of a string variable can be compared to a complete string using the "==" operator. The value of a substring can be compared to the beginning or ending of a string, using the `startswith` and `endswith` functions, respectively. This is demonstrated in Listing 23. In this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python is great"`.
- Line 2: The `endswith` function is used to test if the string `text` ends with `"great"`. If this comparison is successful, Line 3 is executed.
- Line 3: A message is printed on the screen.

- Line 4: The value of `text` is converted to lower case and the result is assigned to the variable `text`.
- Line 5: The `startswith` function is used to test if the string `text` starts with `"python"`. If this comparison is successful, Line 6 is executed.
- Line 6: A message is printed on the screen.

Listing 23: Matching strings.

```
1 text = "Python is great"
2 if text.endswith("great"):
3     print("Ends with great")
4 text = text.lower()
5 if text.startswith("python"):
6     print("Starts with python")
```

13.4 Finding a substring

It is often necessary to find a substring within a string. A programmer may need to find a substring within a string and then use the string before or after where the substring is found. Listing 24 demonstrates how to find the position of a substring within another string and access the substring that follows the search string. In this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python is great"`.
- Line 2: A string variable named `search_text` is defined and assigned `"is"`.
- Line 3: The length of the string in `search_text` is assigned to the variable `offset`.
- Line 4: The position of the search string is found in the variable `text`. The position is returned as an integer value that is assigned to the variable `pos`.
- Line 5: If the value of `pos` is greater than 0, the program runs Line 6. The value of `pos` is -1 if the substring is not found. The number -1 is not a valid index. Therefore, the `if` statement is used to prevent it from being used as an index.
- Line 6: The value in `pos` is incremented by the value that is stored in `offset`. This implies that `pos` now refers to the first character after the `search_text` position.
- Line 7: The text that follows `search_text` within `text` is printed on the screen, together with some description.

Listing 24: Finding a substring within a string.

```
1 text = "Python is great"
2 search_text = "is"
3 offset = len(search_text)
4 pos = text.find(search_text)
5 if pos > 0:
6     pos += offset
7     print("Remaining text: \"" + text[pos:] + "\"")
```

13.5 Replacing a substring

It may be necessary to replace a search string within another string. Listing 25 demonstrates two ways of performing string replacement. In this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python is great"`.
- Line 2: The text `"great"` is replaced with `"a useful language"` by calling the `replace` function. This function can also be used to replace a character with another character or a string with an empty string.
- Line 3: The value in `text` is shortened to the seventh character onwards.
- Line 4: The string `"python3"` is prepended to the value in `text` and the result is assigned to `text`.
- Line 5: The value of `text` is printed on the screen.

Listing 25: Replacing a substring within a string.

```
1 text = "Python is great"
2 text = text.replace("great", "a useful language")
3 text = text[6:]
4 text = "python3" + text
5 print(text)
```

13.6 Splitting and joining strings

A string can be split into a list of substrings, using a specific character. A list of substrings can also be joined together using a string. Listing 26 demonstrates the separation of a string by spaces and the combination of substrings using a comma. In this Listing:

- Line 1: A string variable named `text` is defined and assigned the value `"Python is great"`.
- Line 2: The value stored in `text` is split into three substrings, using a space as the separating character. The resulting list of substrings is assigned to the variable `fragments`.
- Line 3: The first substring within the list is printed on the screen.

- Line 4: The substrings are joined back together, using a comma as a separator.
- Line 5: The value of `text` is printed on the screen.

Listing 26: Splitting and joining a string.

```
1 text = "Python is great"
2 fragments = text.split()
3 print(fragments[0])
4 text = ",".join(fragments)
5 print(text)
```

13.7 Removing white space

Input text that is supplied in an input file, database or through a user interface may contain leading or trailing white spaces. It is often necessary to remove these characters to avoid duplicate values within a program. The Listing 27 demonstrates how to remove leading and trailing white space. In this listing:

- Line 1: A string variable named `text` is defined and assigned the value `"\t Python is great \n"`. The `"\t"` is a tab character, whereas `"\n"` is a newline character. Spaces, tabs and newline characters are all classified as white space characters.
- Line 2: The value of `text` is printed on the screen.
- Line 3: Leading and trailing white space characters are removed from `text` and the result is assigned to `text`.
- Line 4: The value of `text` is printed on the screen.

Listing 27: Stripping leading and trailing white space.

```
1 text = "\t" + " Python is great " + "\n"
2 print(text)
3 text = text.strip()
4 print(text)
```

13.8 String formatting

The values of other variable types can be formatted as strings. Listing 28 demonstrates how to create a string that includes values from two variables and how to format a floating point value as a string with limited precision. In this Listing:

- Line 1: A variable named `a` is defined and assigned an integer value.
- Line 2: A variable named `b` is defined and assigned a floating point value.

- Line 3: A text string is created that includes text and the values of the variables `a` and `b`, where the names of the variables are given within the `{}` brackets. The resulting string is assigned to the variable `text`.
- Line 4: The value of `text` is printed on the screen.
- Line 5: The value that is stored in `b` is formatted as a floating point number with two decimal places. This causes the floating point number to be rounded up to the nearest value and converted to a string. The resulting value is assigned to `text`.
- Line 6: The value of `text` is printed on the screen.

Listing 28: String formatting operations.

```
1 a = 10
2 b = 23.45534232
3 text = f"a:{a}, b:{b}"
4 print(text)
5 text = "{:.2f}".format(b)
6 print(text)
```

13.9 Using regular expressions

Regular expressions can be used to search for matching text strings. The `re` module provides regular expression functionality, which is demonstrated in Listing 29. In this Listing:

- Line 1: The `re` module is imported.
- Line 3: A string variable named `meal_1` is given a value.
- Line 4: A string variable named `meal_1` is given a value.
- Line 5: The regular expression is defined, such that the string must start with the word `fish`.
- Line 7: The regular expression is compiled. Compiling the regular expression causes its syntax to be verified. If the regular expression is not valid, the `compile` function raises an `re.error` exception. Once the expression has been compiled it can be reused as needed.
- Line 8: The regular expression is compared with the first string. The function `match` returns `True`.
- Line 9: The value of `meal_1` is printed.
- Line 10: The regular expression is compared with the second string. The function `match` returns `True`.

Listing 29: Regular expression matching using the `re` module.

```
1  import re
2
3  meal_1 = "fish and chips"
4  meal_2 = "tuna salad"
5  search_string = "fish."
6  try:
7      match_alg = re.compile(search_string)
8      if match_alg.match(meal_1):
9          print(f"Found: \"{meal_1}\".")
10     if match_alg.match(meal_2):
11         print(f"Found: \"{meal_1}\".")
12 except re.error as e:
13     print(f"Could not compile the regular expression.  {e}")
```

14 Basic data structures

The Python programming language includes several types of basic data structures. These include lists, dictionaries, tuples and sets. A data structure can include another data structure within its elements. For example, a list can include a list as each of its elements. In the following subsections, some of the functionality of basic data structures is discussed.

14.1 Lists

A list is a sequential data container that can contain zero or more values. The values can be accessed using an index, where the first list entry is accessed with an index value of 0. Elements can be prepended or appended to a list. The following subsections demonstrate some of the features of a list.

14.1.1 Declaring and clearing a list

A list can be defined that has zero more more elements. Once a list has been defined, one or more elements can be removed. Listing 30 demonstrates how to declare a list with several elements and clear these elements. In this Listing:

- Line 1: A list is created with five elements that all contain the value 0. This syntax copies the value with the `[]` brackets for as many times as specified after the "`*`" character.
- Line 2: The elements within the list `values` are printed on the screen.
- Line 3: All elements are removed from the list `values` by calling the `clear` function.

- Line 4: A list is created that contains the values `[0, 1, 2, 3, 4]`. A reference to this list is assigned to `values`. The function `list` is called to convert the `range` into a list.
- Line 5: The elements within the list `values` are printed on the screen.

Listing 30: Declaring and clearing a list.

```
1 values = [0]*5
2 print(values)
3 values.clear()
4 values = list(range(5))
5 print(values)
```

14.1.2 Appending to a list

A list is a dynamic data container that allows one or more elements to be appended or prepended to it. Listing 31 demonstrates how to append one or more values to a list. In this Listing:

- Line 1: An empty list is created and a reference to it is assigned to `values`.
- Line 2: One element is appended to the list `values`.
- Line 3: A list that contains two elements is defined. The list is appended to the list `values`, by using the "+=" operator. As discussed in Section 7.6, `values +=` is equivalent to `values = values +`.
- Line 4: The elements within the list `values` are printed on the screen.

Listing 31: Appending to a list.

```
1 values = []
2 values.append(3)
3 values += [5, 6]
4 print(values)
```

14.1.3 Sorting and shuffling a list

A list of values may be read from a data source and then need to be sorted for analysis. In some cases, it is necessary to shuffle list values such that they are drawn at random. Listing 32 demonstrates sorting and shuffling of list values. In this Listing:

- Line 1: The `random` module is imported.
- Line 2: A list that contains six elements is assigned to `values`.
- Line 3: The elements that are stored in `values` are sorted into numerically ascending order. The `sort` function updates the list itself and does not return a result.
- Line 4: The elements within the list `values` are printed on the screen.

- Line 5: The `shuffle` function is called to shuffle the values randomly. The `shuffle` function updates the list itself.
- Line 6: The elements within the list `values` are printed on the screen.

Listing 32: Sorting and shuffling a list.

```
1 import random
2 values = [3, 2, 4, 5, 3, 1]
3 values.sort()
4 print(values)
5 random.shuffle(values)
6 print(values)
```

14.1.4 Finding a value within a list

Listing 33: Finding the index of a value in a list.

```
1 values = [4, 5, 1, 3]
2 try:
3     pos = values.index(2)
4     print("Found 2 at position " + str(pos))
5 except ValueError:
6     print("Cannot find 2")
```

14.2 Dictionaries

A dictionary is an associative container, where a unique key is used to access values that are stored in a dictionary. A dictionary may contain zero or more key and value pairs. The following subsections demonstrate some features of a dictionary.

14.2.1 Declaring and clearing a dictionary

A dictionary may be declared with zero or more key and value pairs. The content of an existing dictionary may be cleared, removing all key and value pairs. These two actions are demonstrated in Listing 34. In this Listing:

- Line 1 to 4: A dictionary is created that contains two key and value pairs. A reference to this dictionary is assigned to the variable `data`.
- Line 5: The key and value pairs within `data` are printed to the screen.
- Line 6: All key and value pairs are removed from the dictionary by calling the `clear` function.
- Line 7: The key and value pairs within `data` are printed to the screen.

Listing 34: Declaring and clearing a dictionary.

```
1 data = {
2     "a": 23,
3     "b": 34
4 }
5 print(data)
6 data.clear()
7 print(data)
```

14.2.2 Merging dictionaries

A dictionary can be combined with another dictionary, such that the resulting dictionary contains a superset of keys. When the two sets keys overlap, the value within the original dictionary is updated using the value from the input dictionary. These features are demonstrated in Listing 35. In this Listing:

- Line 1: A dictionary is created that contains two key and value pairs. A reference to this dictionary is assigned to the variable `data`.
- Line 2: A dictionary is created that contains two key and value pairs. A reference to this dictionary is assigned to the variable `more_data`.
- Line 3: The `more_data` dictionary is combined with the `data` dictionary. Since two sets of keys do not overlap, the resulting dictionary will contain four key and value pairs.
- Line 4: The key and value pairs within `data` are printed to the screen.
- Line 5: A dictionary is created that contains two key and value pairs. This dictionary is combined with `data`. Since the keys are already present within the dictionary `data`, the values that are associated with these keys are updated.
- Line 6: The key and value pairs within `data` are printed to the screen.

Listing 35: Merging two dictionaries together.

```
1 data = {"a": 23, "b": 34}
2 more_data = {"d": 54, "c": 67}
3 data.update(more_data)
4 print(data)
5 data.update({"a": 22, "b": 23})
6 print(data)
```

14.2.3 Retrieving a value from a dictionary

Values can be retrieved from a dictionary, either by using the key within `[]` or by calling the `get` function. This is demonstrated in Listing 36. In this Listing:

- Line 1: A dictionary is created that contains two key and value pairs. A reference to this dictionary is assigned to the variable `data`.
- Line 2: The value that is associated with the key `"a"` is retrieved and assigned to `value`.
- Line 3: The content of the variable `value` is printed on the screen.
- Line 4: The value that is associated with the key `"b"` is retrieved by calling the `get` function. The value is assigned to the variable `value`.
- Line 5: The content of the variable `value` is printed on the screen.
- Line 6: The function `get` is used to try to get a value that is associated with the key `"c"`. The key `"c"` does not exist in the dictionary `data`. Therefore, `None` is assigned to the variable `value`.
- Line 7: The content of the variable `value` is printed on the screen.

Listing 36: Getting a value using a key.

```
1 data = {"a": 23, "b": 34}
2 value = data["a"]
3 print(value)
4 value = data.get("b")
5 print(value)
6 value = data.get("c")
7 print(value)
```

If `[]` is used with a key value, then an exception is thrown if the key does not exist within the dictionary. This is discussed in Section 11.3.

14.2.4 Iterating over dictionary keys and values

The keys and the values can be retrieved from the dictionary, such that they can be used within a `for` loop or some other logic. Listing 37 demonstrates how to retrieve the values and keys from a dictionary and loop over them. In this Listing:

- Line 1: A dictionary is created that contains two key and value pairs. A reference to this dictionary is assigned to the variable `data`.
- Line 2: The values are retrieved from `data` by calling the function `values`. For each value, the `for` loop executes Line 3.
- Line 3: The value of `value` is printed on the screen.
- Line 4: The keys are retrieved from `data` by calling the function `keys`. For each key, the `for` loop executes Line 5.
- Line 5: The value of `key` is printed on the screen.
- Line 6: The keys and values are retrieved from `data` by calling the function `items`. Each pair of key and value is assigned to the `key` and `value` variable.

- Line 7: The value of `key` and `value` is printed on the screen.

Listing 37: Iterating over keys and values.

```
1 data = {"a": 23, "b": 34}
2 for value in data.values():
3     print(value)
4 for key in data.keys():
5     print(key)
6 for key, value in data.items():
7     print(f"key={key}, value={value}")
```

14.3 Tuples

A tuple is an immutable container. Once it has been created, elements cannot be added or removed. It can be useful as a simple container to pass values into a function or to return values from a function. A tuple is demonstrated in Listing 38. In this Listing:

- Line 1: A tuple is created that contains three strings. It is assigned to the variable `values`.
- Line 2: The content of `values` is printed on the screen.
- Line 3: The second and third element of `values` are printed on the screen. The colon syntax and its meaning is similar to that used with a string variable or a list.

Listing 38: Declaring and using a tuple.

```
1 values = ("Database", "WebService", "Client")
2 print(values)
3 print(values[1:3])
```

14.4 Sets

Sets contain a unique group of values. They can be useful when operating on lists, since they will reject duplicate values. Sets are demonstrated in Listing 39. In this Listing:

- Line 1: A set is defined that has three string values. A reference to the set is assigned to the variable `values`.
- Line 2: A value is removed from the set by calling the `pop` function.
- Line 3: A value is added to the set by calling the `add` function.
- Line 4: The content of the set `values` is printed on the screen.
- Line 5: A list is created that contains four string values. A reference to the list is assigned to the variable `list_values`.

- Line 6: The list is converted to a set by calling the `set` function. This removes the duplicate entries. The set is converted to a list by calling the `list` function. A reference to the resulting list is assigned to the variable `list_values`.
- Line 7: The content of the list `list_values` is printed on the screen.

Listing 39: Declaring and using sets.

```
1 values = {"Query", "Sort", "Save"}
2 values.pop()
3 values.add("Randomise")
4 print(values)
5 list_values = ["Query", "Sort", "Query", "Sort"]
6 list_values = list(set(list_values))
7 print(list_values)
```

15 Date and time

A date and time value can be stored in a `datetime` variable, where the `datetime` type is defined in the `datetime` module. Listing 40 demonstrates some of the `datetime` module's functionality. In this Listing:

- Line 1: The `datetime` class is imported from the `datetime` module.
- Line 3: A `datetime` value is created and assigned to `some_date`, using a year, month and day. Values for the number of hours, minutes and microseconds can also be provided. If values are not provided, they are set to zero.
- Line 4: The year, month and day are printed.
- Line 7: The current time is retrieved and stored in `current_time`.
- Line 8: The current time is printed.
- Line 10: The difference between `current_time` and `some_date` is calculated.
- Line 11: The value of `time_diff` is printed.
- Line 13: The `datetime` that is stored in `current_time` is converted into a text string, using the ISO format.
- Line 14: The ISO formatted text string is printed.
- Line 15: The ISO formatted text string is used to create a new `datetime`.
- Line 17: The value of `time_from_str` is verified to be equal to `current_time`.

Listing 40: Declaring and using sets.

```
1 from datetime import datetime
2
3 some_date = datetime(2023, 4, 29)
4 print("Year hour minute:",
5       some_date.year, some_date.month, some_date.day)
6
7 current_time = datetime.now()
8 print("The current datetime is:", current_time)
9
10 time_diff = current_time - some_date
11 print("The time different is:", time_diff)
12
13 time_str = current_time.isoformat()
14 print("isoformat:", time_str)
15 time_from_str = datetime.fromisoformat(time_str)
16
17 if current_time == time_from_str:
18     print("current_time == time_from_str")
```

16 Functions

Functions should be used to encapsulate functionality, where there is a clear purpose to a specific function. A function should have a name that indicates its purpose. A function can be passed zero or more arguments. It can return one or no arguments. The process of defining and calling functions is demonstrated in the following subsections.

16.1 Defining a function

A simple function is given in Listing 41. This function has one input argument, which it returns as a return value. The purpose of this function is to demonstrate the process of passing in a value and returning a value. Normally, a function should have several lines of content that provide the functionality of the function. In Listing 41:

- Line 5: The function `pass_through` is given the value 10.
- Line 1: The function `pass_through` receives the input value and assigns it to a variable named `input`. The `input` variable only exists within the function `pass_through`.
- Line 2: The value that is contained in `input` is returned.
- Line 5: The returned value is assigned to the variable `value`.
- Line 6: The value of `value` is printed on the screen.

Listing 41: Defining and calling a function.

```
1 def pass_through(input):
2     return input
3
4
5 value = pass_through(10)
6 print(value)
```

16.2 Two functions

A program can be constructed from a set of user-defined functions. Listing 42 contains two functions, where one function calls the other one. In this Listing:

- Line 10: The function `print_value` is passed 10.
- Line 5: The function `print_value` receives the value 10 into the variable `input`. This version of the variable `input` only exists within the function `print_value`.
- Line 6: The function `pass_through` is passed the value that is stored in the variable `input`.
- Line 1: The function `pass_through` receives the value and stores it within a variable named `input`. This version of the variable `input` only exists within the function `pass_through`.
- Line 2: The value that is stored in the variable `input` is returned.
- Line 6: The returned value is received into the variable `value`.
- Line 7: The value that is stored in `value` is printed on the screen.

Listing 42: Calling one function from another one.

```
1 def pass_through(input):
2     return input
3
4
5 def print_value(input):
6     value = pass_through(input)
7     print(value)
8
9
10 print_value(10)
```

16.3 Mutable arguments

The mutability of different variable types is discussed in Section 6.2. Passing a value or reference into a function behaves in the same manner as an assignment, which is discussed in Section 6.2. For

example, when an integer is passed into a function the value is passed, whereas a list is passed by reference.

Listing 43 demonstrates the behaviour of two mutable and one immutable variable type, when they are passed to function. In this Listing:

- Line 8: An empty list is created and a reference to it is assigned to `values`.
- Line 9: An empty dictionary is created and a reference to it is assigned to `data`.
- Line 10: An integer variable named `i` is defined and assign the value 10.
- Line 11: The function `update_data` is called, passing it the reference to `values` and `data`, and the value of `i`.
- Line 1: The references and value are received into the variables `values`, `data` and `i`, respectively. The variable `values` contains a reference to the empty list that was created at Line 8. The variable `data` contains a reference to the empty dictionary that was created at Line 9. The variable `i` contains the value that was passed to it. It does not contain a reference to the variable `i` that is defined at Line 10.
- Line 2: A list containing three values `[0, 1, 2]` is created and appended to the list `values`.
- Line 3: A dictionary key named `"a"` is created and associated with the value 12.
- Line 4: A dictionary key named `"b"` is created and associated with the value 10.
- Line 5: The value of `i` is incremented by 1.
- Line 12: The contents of the list `values` is printed on the screen. There are three list elements, since the list is passed by reference into the function `update_data`.
- Line 13: The contents of the dictionary `data` is printed on the screen. There are two key and value pairs, since the dictionary is passed by reference into the function `update_data`.
- Line 14: The value of `i` is printed on the screen. The value is still 10, since the value was passed into the function `update_data`.

Listing 43: Updating mutable variables.

```
1 def update_data(values, data, i):
2     values += list(range(3))
3     data["a"] = 12
4     data["b"] = 10
5     i += 1
6
7
8 values = []
9 data = {}
10 i = 10
11 update_data(values, data, i)
12 print(values)
13 print(data)
14 print(i)
```

16.4 Recursion

Functions can be used to call themselves. This is defined as recursion. Recursion can be useful when traversing a hierarchical data structure. It is important to limit the number of recursive calls, since there is a maximum permitted depth for the Python programming language. This number is large and configurable, but is not infinite.

Listing 44 demonstrates how to create a recursive function. The function continues to call itself until it reaches the recursive depth 5. In this Listing:

- Line 10: The function `recursion` is called without passing it a value.
- Line 1: The function `recursion` is defined with one argument that has a default value 0. The default value is used when no value is passed into the function. The value 0 is assigned to the variable `depth`.
- Line 2: The value of `depth` is incremented by 1.
- Line 3: The value of `depth` is printed, together with some description.
- Line 4: If the value of `depth` is equal to 5, Line 5 is run.
- Line 5: A message is printed to the screen.
- Line 6: The function exits and returns `None`.
- Line 7: If the value of `depth` is less than 5, the function `recursion` is called passing it the current value of the variable `depth`. This causes the program to skip to Line 1.

Listing 44: Using recursion.

```
1 def recursion(depth=0):
2     depth += 1
3     print("Depth: " + str(depth))
4     if depth == 5:
5         print("Reached maximum depth.")
6         return
7     recursion(depth)
8
9
10 recursion()
```

17 Classes

Classes can contain zero or more member functions and zero or more data members. A class is instantiated to create an object. The object is a separate instance, rather like two lists are stored in different sections of the computer's memory.

Classes and objects are the building blocks of the object-oriented programming paradigm. In this programming paradigm, functions are clustered together with the data that they operate on. In other programming languages, assessor functions are used to set or get the data that belongs to an object. In the Python programming language, accessor functions should be used sparingly since they can add a considerable overhead in terms of the performance of the final program. The examples given in the following subsections demonstrate some of the object-oriented functionality that is available in the Python programming language.

17.1 Class and object

Once a class has been defined, objects of the specific class can be instantiated. The data members of each object occupy a separate section of the computer's memory. This is demonstrated in Listing 45. In this Listing:

- Line 10: A `Tree` object is instantiated, passing the constructor the values 20 and 4.
- Line 2: The constructor receives the values 20 and 4 into the variables `nuts` and `branches`. The variable `self` is needed for the constructor to refer to a specific object. A value is not explicitly passed to `self` at Line 10 or 11.
- Line 3: The value of `nuts` is copied into the variable `self.nuts`. The `self.nuts` variable is a data member of the class, where the `self` refers to a specific object.
- Line 4: The value of `branches` is copied into the variable `self.branches`. The `self.branches` variable is a data member of the class, where the `self` refers to a specific object.

- Line 10: A reference to the object that is returned is assigned to the variable `tree`.
- Line 11: A `Tree` object is instantiated, passing the constructor the values 10 and 4. This calls the constructor at Line 2. The reference to the object that is returned is assigned to the variable `another_tree`. The objects `tree` and `another_tree` are stored in different sections of the computer's memory.
- Line 12: The `nuts_per_branch` function is called, from the `tree` object.
- Line 6: The `self` refers to a specific object. It does not need to be explicitly passed a value.
- Line 7: The value of nuts per branch is calculated and returned. The variable `self.nuts` and `self.branches` are data members. Therefore, they are accessible within every function that belongs to the class. They do not need to be passed into the function `nuts_per_branch`, since they are accessible from the `self` variable.
- Line 12: The value returned from the function `nuts_per_branch` is assigned to the variable `value`.
- Line 13: The value that is contained in `value` is printed to the screen. The value printed is the result of the division 20/4.
- Line 14: The `nuts_per_branch` function is called, from the `another_tree` object. The return value is assigned to `value`.
- Line 15: The value that is contained in `value` is printed to the screen. The value printed is the result of the division 10/4.

Listing 45: Defining a class and using objects.

```
1  class Tree:
2      def __init__(self, nuts, branches):
3          self.nuts = nuts
4          self.branches = branches
5
6      def nuts_per_branch(self):
7          return self.nuts/self.branches
8
9
10 tree = Tree(20, 4)
11 another_tree = Tree(10, 4)
12 value = tree.nuts_per_branch()
13 print(value)
14 value = another_tree.nuts_per_branch()
15 print(value)
```

17.2 Static data members and functions

In Section 17.1, data members and functions are defined that are associated with a specific object. It is also possible to define data members and functions that are specific to a class, rather than to one object. This is referred to as a static data member or function.

- Line 9: The static data member `nuts` of the `Tree` class is assigned 20.
- Line 10: The static data member `branches` of the `Tree` class is assigned 4.
- Line 11: The static member function `nuts_per_branch` of the `Tree` class is called.
- Line 5: The `nuts_per_branch` function receives no input arguments.
- Line 6: The ratio is calculated and returned.
- Line 11: The returned value is assigned to the variable `value`.
- Line 12: The value that is contained in `value` is printed on the screen.

Listing 46: A class with static data members and a static function.

```
1 class Tree:
2     nuts = 0
3     branches = 0
4
5     def nuts_per_branch():
6         return Tree.nuts/Tree.branches
7
8
9 Tree.nuts = 20
10 Tree.branches = 4
11 value = Tree.nuts_per_branch()
12 print(value)
```

17.3 Inheritance

Classes can inherit member functions or data members from other classes. This implies that functionality that is common to two or more classes can be factorised out into a base class that the other classes inherit.

Inheritance is introduced in Listing 47. In this Listing:

- Line 20: A `WebServer` object is created, by passing the constructor function an IP address and a port number.
- Line 16: The constructor function receives the IP address and port number into the variables `ip_address` and `port_number`, respectively.

- Line 17: The `WebServer` class is defined as inheriting from the `Server` class at Line 15. Therefore, the `WebServer` constructor is able to call the constructor for the `Server` class. The IP address, port number and protocol is passed to the `Server` class constructor function.
- Line 2: The IP address, port number and protocol are received into the variables `ip_address`, `port_number` and `protocol`, respectively.
- Line 3: The value stored in the variable `ip_address` is copied into the data member `self.ip_address`.
- Line 4: The value stored in the variable `port_number` is copied into the data member `self.port_number`.
- Line 5: The value stored in the variable `protocol` is copied into the data member `self.protocol`.
- Line 20: A reference to the `WebServer` object is assigned to the `web_server` variable.
- Line 21: The `as_dict` function is called to return the contents of the object as a dictionary.
- Line 7: The `as_dict` receives no input arguments, beyond its association with an object.
- Line 8: An empty dictionary is created and a reference to it is assigned to `dict_version`.
- Line 9: A key `"ip_address"` is created and assigned the value of `self.ip_address`.
- Line 10: A key `"port_number"` is created and assigned the value of `self.port_number`.
- Line 11: A key `"protocol"` is created and assigned the value of `self.protocol`.
- Line 12: A reference to the dictionary is returned.
- Line 21: The dictionary is passed to the print function, which prints the values on the screen.

Listing 47: A program to demonstrate class inheritance.

```
1 class Server:
2     def __init__(self, ip_address, port_number, protocol):
3         self.ip_address = ip_address
4         self.port_number = port_number
5         self.protocol = protocol
6 
7     def as_dict(self):
8         dict_version = {}
9         dict_version["ip_address"] = self.ip_address
10        dict_version["port_number"] = self.port_number
11        dict_version["protocol"] = self.protocol
12        return dict_version
13 
14 
15 class WebServer(Server):
16     def __init__(self, ip_address, port_number):
17         Server.__init__(self, ip_address, port_number, "http")
18 
```

```
19
20 web_server = WebServer("127.0.0.1", 80)
21 print(web_server.as_dict())
```

17.4 Public and private

Python classes support public, protected and private data members and member functions. A public data member or member function can be accessed from anywhere within the program, inside an object of the class or from another object or function. Protected members can be accessed within objects of the specific class or from objects of a class that inherits from a class where the protected members are defined. Private members can only be accessed within object of the specific class. Therefore, they can be used to encapsulate data and state within the object.

Listing 48 demonstrates how to declare a private and a public data member. A public member has no "_" character after "`self.`", a protected member has a single "_" character after "`self.`" and a private member has two "_" characters after "`self.`". These rules apply to the declaration of data members and member functions.

In Listing 48:

- Line 7: A `Car` object is creating, passing the constructor the values `4` and `"red"`.
- Line 2: The constructor receives the values into the variables `n_wheels` and `colour`, respectively.
- Line 3: The value of `n_wheels` is assigned to the public data member `self.n_wheels`.
- Line 3: The value of `colour` is assigned to the private data member `self.__colour`.
- Line 8: The value of the public data member `n_wheels` is printed.
- Line 9: The program attempts to print the value of the private data member `__colour`. However, this fails, since `__colour` is not accessible outside of the class `Car`.

Listing 48: A class with a public and private data member.

```
1 class Car:
2     def __init__(self, n_wheels, colour):
3         self.n_wheels = n_wheels
4         self.__colour = colour
5
6
7 car = Car(4, "red")
8 print(car.n_wheels)
9 print(car.__colour)
```

17.5 Polymorphism

Polymorphism allows the definition of one or more functions that may be associated with more than one object type. This implies that a program may be written assuming that an associated set of functions will be available, but without knowing the type of the object that is associated with these functions.

In the Python programming language, polymorphism may be implemented through inheritance or through the definition of the same function in two or more classes. In both cases, the function must have the same number of required input arguments. Listing 49 demonstrates how to implement polymorphism by defining the same function in two different classes. In this Listing:

- Line 20: An empty list is created and reference to it is assigned to `shapes`.
- Line 21: A `Cube` object is created by calling the constructor function in the `Cube` class.
- Line 5: The constructor receives the value `2` into the variable `x`.
- Line 6: The value of the variable `x` is copied into the data member `self.x`.
- Line 21: The reference to the `Cube` object is appended to the `shapes` list.
- Line 22: A `Sphere` object is created by calling the constructor function in the `Sphere` class.
- Line 13: The constructor receives the value `1` into the variable `r`.
- Line 14: The value of the variable `r` is copied into the data member `self.r`.
- Line 22: The reference to the `Sphere` object is appended to the `shapes` list.
- Line 23: Each element of the `shapes` list is assigned to the variable `shape`, until there are no more elements left.
- Line 24: The `volume` function of the `Cube` and `Sphere` object is called. On the first iteration, the `for` loop calls the function `volume` that is defined at Line 8. On the second iteration, the `for` loop calls the function `volume` that is defined at Line 16. The return value from the `volume` function is assigned to the variable `value`.
- Line 25: The value that is stored in `value` is printed on the screen.

Listing 49: A program to demonstrate function polymorphism.

```
1 import math
2
3
4 class Cube:
5     def __init__(self, x):
6         self.x = x
7
8     def volume(self):
9         return self.x**3
10
```

```
11
12 class Sphere:
13     def __init__(self, r):
14         self.r = r
15
16     def volume(self):
17         return 4*math.pi*(self.r**3)/3
18
19
20 shapes = []
21 shapes.append(Cube(2))
22 shapes.append(Sphere(1))
23 for shape in shapes:
24     value = shape.volume()
25     print(value)
```

17.6 Operator overloading

The Python operators are summarised in Section 7. These operators can be used with simple variable types, such as integers or floats. Operators can also be used with objects, provided that an associated function has been implemented within the class. The use of operator functions within a class is referred to as operator overloading. Each operator is associated with a specific member function name and input arguments. Beyond the function name and input arguments, the function is a normal Python function and can perform actions that the programmer decides to implement.

Operator overloading is demonstrated in Listing 50. In this Listing:

- Line 21: A `Rectangle` object is created by calling the constructor at Line 2. A reference to the object is assigned to `rect1`.
- Line 21: A `Rectangle` object is created by calling the constructor at Line 2. A reference to the object is assigned to `rect2`.
- Line 23: The equality operator is used to test if the two objects are equal to each other.
- Line 17: The function `__eq__`, which is associated with the equality operator, receives the object to the left of "==" into `self` and the object that is to the right of "==" into `other`.
- Line 18: The function returns `True` if the width and the height of the two objects is the same.
- Line 24: If the rectangles have the same dimensions, a message is printed to state this.
- Line 25: The less than operator is used to compare the two objects.
- Line 14: The function `__lt__`, which is associated with the less than operator, receives the object to the left of "<" into `self` and the object that is to the right of "<" into `other`.

- Line 15: The function returns `True` if the width and the height of the first object is less than the width and the height of the other object.
- Line 26: If the first rectangle has a smaller width and height than the second object, a message is printed to state this.
- Line 27: The two objects are combined to create a new object, using the addition operator.
- Line 9: The function `__add__`, which is associated with the addition operator, receives the object to the left of "+" into `self` and the object that is to the right of "+" into `other`.
- Line 10: The width of the two objects is added together and assigned to `width`.
- Line 11: The height of the two objects is added together and assigned to `height`.
- Line 12: A new `Rectangle` object is created and a reference to it is returned.
- Line 27: The reference to the resulting object is assigned to `rect3`.
- Line 28: The reference to the `rect3` object is passed to the `print` function, which casts the object to a string. When the `rect3` object is cast to a string the `__repr__` function is called.
- Line 6: The function `__repr__` receives no input arguments, beyond its association with an object.
- Line 7: The function returns a string that includes the value of the `self.width` and `self.height` data members. The string formatting is discussed in Section 13.8.
- Line 28: The string returned is printed on the screen.

Listing 50: A program to demonstrate operator overloading.

```
1  class Rectangle:
2      def __init__(self, width, height):
3          self.width = width
4          self.height = height
5
6      def __repr__(self):
7          return f"width:{self.width}, height:{self.height}"
8
9      def __add__(self, other):
10         width = self.width + other.width
11         height = self.height + other.height
12         return Rectangle(width, height)
13
14     def __lt__(self, other):
15         return self.width < other.width and self.height < other.height
16
17     def __eq__(self, other):
18         return self.width == other.width and self.height == other.height
19
20
```

```
21 rect1 = Rectangle(1, 2)
22 rect2 = Rectangle(2, 3)
23 if rect1 == rect2:
24     print("The rectangles have the same dimension.")
25 if rect1 < rect2:
26     print("rect1 is smaller than rect2 in both dimensions.")
27 rect3 = rect1 + rect2
28 print(rect3)
```

18 Input/Output operations

Python libraries exist to write and read many different data formats. In this section, a few of the common data formats are discussed. These have been selected, since they are supported by basic Python libraries.

18.1 Text files

It may be necessary to read data from a text file, which does not have a common structure. Python provides simple functionality that can be used to read a text file line by line or all lines at once into memory.

Listing 51 demonstrates how to write a text file line by line and read it back into an array of lines. In this Listing:

- Line 18: A variable named `file_name` is defined and assigned a text string.
- Line 19: The function `write_file` is passed the value of `file_name`.
- Line 1: The function `write_file` receives the file name value into the variable `file_name`.
- Line 2: A new file is opened in the present working directory. The name of the file is taken from the `file_name` variable. The `"w"` flag implies that the file is written and if the file already exists then it is truncated to be an empty file.
- Line 3: A text string and a newline character `"\n"` are written to the file by calling the `write` function.
- Line 4: A text string and a newline character `"\n"` are written to the file by calling the `write` function.
- Line 5: The output file is closed by calling the `close` function.
- Line 20: The function `read_file` is passed the value of `file_name`.
- Line 8: The function `read_file` receives the file name value into the variable `file_name`.
- Line 9: A file is opened for read access by providing the `"r"` flag. The file is assumed to be found in the present working directory and have the name that is stored in the `file_name` variable.

- Line 10: All of the lines are read into a list by calling the `readlines` function. A reference to this list is assigned to `lines`.
- Line 11: The input file is closed by calling the `close` function.
- Line 12: An empty list is created and assigned to `ip_addresses`.
- Line 13: The `for` loop iterates over each element of the `lines` list.
- Line 14: The trailing newline character is removed from the string by calling the `strip` function. The resulting string is appended to the `ip_addresses` list.
- Line 15: The contents of the `ip_addresses` list is printed on the screen, together with some description text.

Listing 51: Writing to and reading from a text file.

```
1  def write_file(file_name):
2      output_file = open(file_name, "w")
3      output_file.write("127.0.0.1" + "\n")
4      output_file.write("8.8.8.8" + "\n")
5      output_file.close()
6
7
8  def read_file(file_name):
9      input_file = open(file_name, "r")
10     lines = input_file.readlines()
11     input_file.close()
12     ip_addresses = []
13     for line in lines:
14         ip_addresses.append(line.strip())
15     print("IP addresses:" + str(ip_addresses))
16
17
18 file_name = "file_io.txt"
19 write_file(file_name)
20 read_file(file_name)
```

18.2 CSV files

Comma separated value (CSV) files are often used as an intermediate data format. CSV files can be read from or written to using functions from the `csv` module. The `csv` module provides a simple interface to the rows in a CSV file and a dictionary based interface that uses column names.

Listing 52 demonstrates how to write and read a CSV file, using the standard Excel format. In this Listing:

- Line 1: The `csv` module is imported.
- Line 28: A variable named `file_name` is defined and assigned a text string.
- Line 29: The function `write_csv` is passed the value of `file_name`.
- Line 4: The function `write_csv` receives the file name value into the variable `file_name`.
- Line 5: A new file is opened in the present working directory. The `"w"` flag implies that the file is written and if the file already exists then it is truncated to be an empty file. The `newline=''` implies that newline characters are not automatically appended. This is selected such that the `writer` function on Line 6 appends the appropriate newline character.
- Line 6 to 8: A CSV writer is created by calling the `writer` function. This function is part of the `csv` module. The function is passed the open output file, together with some optional settings. These settings are to set the delimiter between the values to be a comma, the quotation character around values to be a double quote and quoting to be used for any non-numeric values.
- Line 9: A row that contains two columns is written to the output file by calling the `writerow` function and passing it a list of two values.
- Line 10: A row that contains two columns is written to the output file by calling the `writerow` function and passing it a list of two values.
- Line 11: A row that contains two columns is written to the output file by calling the `writerow` function and passing it a list of values.
- Line 12: The output file is closed by calling the `close` function.
- Line 30: The function `read_csv` is passed the value of `file_name`.
- Line 15: The function `read_csv` receives the file name value into the variable `file_name`.
- Line 16: A file is opened for read access by providing the `"r"` flag. The file is assumed to be found in the present working directory and have the name that is stored in the `file_name` variable. The use of newlines is disabled, such that it it is used by the `reader` function instead.
- Line 17 to 18: A CSV reader is created by calling the `reader` function. This function is part of the `csv` module. The function is passed the open input file, together with some optional settings. These settings are to set the delimiter between the values to be a comma and the quotation character around values to be a double quote. It is not necessary to set the quoting style, since the reader will detect the style.
- Line 19: The `for` loop iterates over each row within the `csv_reader`.
- Line 20: If this is the first line in the file, then Line 21 is run.
- Line 21: A description and the values from the first row are printed on the screen.
- Line 22: The program skips to Line 19.
- Line 23: A description and the values from the current row are printed on the screen.
- Line 24: The CSV file is closed by calling the `close` function.

Listing 52: Writing to and reading from a CSV file.

```
1  import csv
2
3
4  def write_csv(file_name):
5      csv_file = open(file_name, "w", newline='')
6      csv_writer = csv.writer(csv_file, delimiter=',',
7                              quotechar='"',
8                              quoting=csv.QUOTE_NONNUMERIC)
9      csv_writer.writerow(["Host name", "IP address"])
10     csv_writer.writerow(["localhost", "127.0.0.1"])
11     csv_writer.writerow(["GoogleDNS", "8.8.8.8"])
12     csv_file.close()
13
14
15 def read_csv(file_name):
16     csv_file = open(file_name, "r", newline='')
17     csv_reader = csv.reader(csv_file, delimiter=',',
18                             quotechar='"')
19     for row in csv_reader:
20         if csv_reader.line_num == 1:
21             print("Column names:" + str(row))
22             continue
23         print("Data row:" + str(row))
24     csv_file.close()
25
26
27 file_name = "output.csv"
28 write_csv(file_name)
29 read_csv(file_name)
```

Listing 53 demonstrates how to use the `DictWriter` and `DictReader` to create the same file structure as produced by Listing 52. Concerning Listing 53:

- Line 7: The `DictWriter` must be given a list of `fieldnames`. These are the keys within the dictionary rows that will be written to the output file. An empty list can be provided, as long as the list of `fieldnames` is updated before the first row is written.
- Line 10: The `fieldnames` are used to write the column names into the file. If the `writeheader` function is not called, the column names are not written.
- Line 11-12: Each row must be provided as a dictionary that contains a key for each field name, with an associated value.
- Line 20: The `fieldnames` are available as soon as the `DictReader` has been created.

- Line 22: Each line is read as a dictionary that includes field name keys and associated values that are taken from the CSV file.

Listing 53: Using the CSV `DictWriter` and `DictReader`.

```
1  import csv
2
3
4  def write_csv(file_name):
5      csv_file = open(file_name, "w", newline='')
6      fieldnames = ["Host name", "IP address"]
7      dict_writer = csv.DictWriter(csv_file, fieldnames=fieldnames,
8                                   delimiter=',', quotechar='"',
9                                   quoting=csv.QUOTE_NONNUMERIC)
10     dict_writer.writeheader()
11     dict_writer.writerow({"Host name": "localhost", "IP address": "
           127.0.0.1"})
12     dict_writer.writerow({"Host name": "GoogleDNS", "IP address": "
           8.8.8.8"})
13     csv_file.close()
14
15
16 def read_csv(file_name):
17     csv_file = open(file_name, "r", newline='')
18     dict_reader = csv.DictReader(csv_file, delimiter=',',
19                                  quotechar='"')
20     print("Column names:" + str(dict_reader.fieldnames))
21     for row in dict_reader:
22         print("Data row:" + str(row))
23     csv_file.close()
24
25
26 file_name = "dict_output.csv"
27 write_csv(file_name)
28 read_csv(file_name)
```

18.3 JSON files

JavaScript Object Notation (JSON) files are often used to send data to and from web services. They are also used within NoSQL databases and for data storage. The `json` module contains functions to create and read JSON strings.

Listing 54 demonstrates how to write and read two classes from a JSON file. In this Listing:

- Line 1: The `json` module is imported.
- Line 38: A variable named `file_name` is defined and assigned a text string.
- Line 39: The function `write_json` is passed the value of `file_name`.
- Line 20: The function `write_json` receives the file name value into the variable `file_name`.
- Line 21: A new `DataObject` object is created by calling the constructor.
- Line 5: The input string is received into the variable `data`.
- Line 6: The value of `data` is assigned to the data member `self.data`.
- Line 21: The reference to the object is assigned to `data_object`.
- Line 22: The `to_json` function is called to convert the object into a format that can be written to a JSON file.
- Line 11: The `to_json` function receives no arguments, except for its association with an object.
- Line 12: A dictionary is returned that contains a single key and value pair.
- Line 22: A reference to the dictionary is assigned to `json_data`.
- Line 23: An output file is opened. The `"w"` flag implies that the file is written and if the file already exists then it is truncated to be an empty file. The `"utf-8"` character encoding is used to allow a greater range of characters to be written to the file.
- Line 24: The `dump` function from the `json` module is called to save the `json_data` as a JSON file. The function is passed the data to save, a reference to the output file, `ensure_ascii=False` to allow binary data to be within the data payload and `indent=4` to improve the readability for a developer.
- Line 26: The output file is closed by calling the `close` function.
- Line 40: The function `read_json` is passed the value of `file_name`.
- Line 29: The function `read_json` receives the file name value into the variable `file_name`.
- Line 30: An input file is opened, where the `"r"` flag implies that the file is opened with a read-only connection. The encoding is chosen to match the encoding that was used when the file was written.
- Line 31: The contents of the file is loaded into `json_data` by calling the `load` function of the `json` module. The `load` function is passed a reference to the input file.
- Line 32: The input file is closed by calling the `close` function.
- Line 33: A `DataObject` object is created by calling the constructor.
- Line 5: The constructor does not receive an input argument. Therefore, the default value for `data` is used, which is an empty string.
- Line 6: The value of `data` is assigned to the data member `self.data`.
- Line 33: The reference to the `DataObject` object is assigned to `data_object`.
- Line 34: A reference to `json_data` is passed to the function `from_json`.
- Line 14: The reference is received into `json_data`.

- Line 15: The `self.data` member is assigned an empty string.
- Line 16: An `if` statement is used to test if the `"data"` key exists within `json_data`. If the key is found, then Line 17 is run.
- Line 17: The value that is associated with the key `"data"` is assigned to the data member `self.data`.
- Line 35: The `data_object` is cast to a string.
- Line 8: The `__repr__` function receives no input arguments, beyond its association with an object.
- Line 9: The contents of the object is returned as a string.
- Line 35: The string value is printed on the screen.

Listing 54: Writing to and reading from a JSON file.

```
1  import json
2
3
4  class DataObject():
5      def __init__(self, data=""):
6          self.data = data
7
8      def __repr__(self):
9          return str(self.to_json())
10
11     def to_json(self):
12         return {"data": str(self.data)}
13
14     def from_json(self, json_data):
15         self.data = ""
16         if "data" in json_data:
17             self.data = json_data["data"]
18
19
20 def write_json(file_name):
21     data_object = DataObject("DataString:12345")
22     json_data = data_object.to_json()
23     output_file = open(file_name, "w", encoding="utf-8")
24     json.dump(json_data, output_file, ensure_ascii=False,
25               indent=4)
26     output_file.close()
27
28
29 def read_json(file_name):
```

```
30      input_file = open(file_name, "r", encoding="utf-8")
31      json_data = json.load(input_file)
32      input_file.close()
33      data_object = DataObject()
34      data_object.from_json(json_data)
35      print(data_object)
36
37
38  file_name = "json_data.json"
39  write_json(file_name)
40  read_json(file_name)
```

The `json` module supports Python lists, dictionaries and simple variable types. The data that is converted to and from a JSON file can contain many layers of nested lists or dictionaries.

18.4 Pickle files

Pickle files provide a mechanism to save and restore Python objects. They are written to or read from binary files. Pickle files should not be read from an untrusted source, since they can be used maliciously to attack the functionality of a program. The `pickle` module contains functions to write to and read from pickles.

Listing 55 demonstrates how to write and read an object from a pickle file. In this Listing:

- Line 1: The `pickle` module is imported.
- Line 26: A variable named `file_name` is defined and assigned a text string.
- Line 27: The function `write_pickle` is passed the value of `file_name`.
- Line 12: The function `write_pickle` receives the file name value into the variable `file_name`.
- Line 13: A `DataObject` object is created by calling the constructor.
- Line 5: The constructor is passed no arguments, except its association with an object.
- Line 6: The private data member `self.__private` is assigned a string value.
- Line 14: An output file is opened. The flag `"w"` implies that the output file is written and truncated if it already exists. The flag `"b"` implies that a binary output file is written, rather than a human-readable text file.
- Line 15: The `dump` function from the `pickle` module is called to save the contents of the `data_object` to the output file.
- Line 16: The output file is closed by calling the `close` function.
- Line 28: The function `read_pickle` is passed the value of `file_name`.
- Line 19: The function `read_pickle` receives the file name value into the variable `file_name`.

- Line 20: An input file is opened. The flag `"r"` implies that the file is opened with a read-only connection. The flag `"b"` implies that the file is opened as a binary file.
- Line 21: The `load` function from the `pickle` module is called to load the input file as an object. A reference to the object is assigned to `data_object`.
- Line 22: The input file is closed by calling the `close` function.
- Line 23: The `data_object` is cast to a string.
- Line 8: The `__repr__` function receives no arguments, except for its association with an object.
- Line 9: The value of the private data member is returned as a string.
- Line 23: The string value is printed on the screen, together with some description.

Listing 55: Writing to and reading from a Pickle file.

```
1  import pickle
2
3
4  class DataObject():
5      def __init__(self):
6          self.__private = "DataString:12345"
7
8      def __repr__(self):
9          return f"__private:{self.__private}"
10
11
12 def write_pickle(file_name):
13     data_object = DataObject()
14     output_file = open(file_name, "wb")
15     pickle.dump(data_object, output_file)
16     output_file.close()
17
18
19 def read_pickle(file_name):
20     input_file = open(file_name, "rb")
21     data_object = pickle.load(input_file)
22     input_file.close()
23     print("dataObject=" + str(data_object))
24
25
26 file_name = "pickle.bin"
27 write_pickle(file_name)
28 read_pickle(file_name)
```

19 Modules

A module can be defined as one Python file or a collection of Python files. Each Python file should contain a group or related classes, functions and variables. The name of the module should be chosen to reflect its purpose.

Modules can be loaded if they are available in the present working directory or if the directory that contains them is within the `PYTHONPATH` environment variable.

19.1 Files as modules

Listing 56 corresponds to a file named `module_my.py`. The Python file is a module. The module is imported in Listing 57. In this Listing:

- Line 1: The `my_module` file is imported. This loads the contents of the `module_my.py` file into memory.
- Line 2: The `hello_world` from the `module_my.py` file is called.

Listing 56: An example module that contains a single function. The file is named `module_my.py`

```
1 def hello_world():
2     print("Hello World")
```

Listing 57: An example program that imports the `module_my.py` module and calls its function.

```
1 import my_module
2 my_module.hello_world()
```

19.2 Defining a main

When a single Python file is imported, the lines of source code outside of functions or classes are run. While this may be the intended behaviour for a program, it is often necessary to prevent sections of a Python file from being run when the file is imported. To separate what is run when the file is executed from when the file is imported, a main entry point can be defined.

Listing 60 demonstrates how to define a main entry point. The contents of the `main.py` file can be executed as demonstrated in Listing 58. When the file is executed, Line 1 and 3 of Listing 60 is run.

Listing 58: Running the main.

```
python main.py
```

The file `main.py` can be imported as demonstrated in Listing 59. When the file is imported, Line 1 of Listing 60 is run, but Line 3 is not run.

Listing 59: Importing the main.

```
import main
```

Listing 60: Demonstrating the main entry point within a file named `main.py`

```
1 print("Importing or running the program.")
2 if __name__ == "__main__":
3     print("Running the program.")
```

19.3 Directories as modules

An example module named `my_module` is defined using the file structure that is given in Listing 61.

Listing 61: Directory as a module.

```
1 my_module/
2 my_module/__init__.py
3 my_module/__main__.py
4 my_module/another.py
```

In this Listing:

- Line 1: The directory name `my_module` is the module name.
- Line 2: A directory is considered as a Python module if a file named `__init__.py` is present.
- Line 3: The module can be executed if a file named `__main__.py` is present in the same directory as the `__init__.py` file.
- Line 4: A module may include zero or more additional Python files.

An example `my_module/__init__.py` file is given in Listing 62. Similar to any Python file, it can contain functions and classes.

Listing 62: An example file named `my_module/__init__.py`

```
1 def fun():
2     print("Running function \"fun\".")
3
4 print("Loaded module \"my_module\".")
```

An example `my_module/__main__.py` file is given in Listing 63.

- Line 1: Functions or modules can be imported from the same module by using a relative import statement. The "." character refers to the current module. The function `fun` is defined in the `__init__.py` file.
- Line 2: Similar to other Python files several lines of Python code can be included.
- Line 3: The function `fun` that is defined in `__init__.py` is called.

Listing 63: An example file named `my_module/__main__.py`

```
1 from . import fun
2 print("Executing module \"my_module\".")
3 fun()
```

The module `my_module` can be executed by typing the command given in Listing 64. This causes the file named `my_module/__main__.py` to be run.

Listing 64: Running the `my_module` module.

```
  python -m my_module
```

An example program that uses `my_module` is given in Listing 65.

- Line 1: The function `fun` is imported from the `my_module/__init__.py` file.
- Line 2: The function `another_fun` is imported from the `my_module/another.py` file.
- Line 3: The function `fun` that is defined in `my_module/__init__.py` is called.
- Line 4: The function `another_fun` that is defined in `my_module/another.py` is called.

Listing 65: An example program that imports the `my_module.py` module and calls its functions.

```
1 from my_module import fun
2 from my_module.another import another_fun
3 fun()
4 another_fun()
```

20 Unit tests

Sections of a program are referred to as units. These sections could comprise functions or classes. To ensure that these functions or classes are behaving as expected, unit tests are written. A unit test should include an expect response that should be returned when a function is given a fixed input value. The returned value is then compared with the expected value to determine if the test passes. One or more requirements can be implemented within one test, such that a test has to pass all of the requirements for the test to pass.

Listing 66 contains the definition of a `factorial` function that is present within a file named `unit_test_module.py`. In this Listing:

- Line 1: The `factorial` function receives a value into the variable `x`.
- Line 2: If the variable type of `x` is not an integer, then Line 3 is run.
- Line 3: The value `-1` is returned.
- Line 4: The result is assigned `1`, since factorial of zero is 1.

- Line 5: The `for` loop iterates over values from `0` to `x-1`.
- Line 6: The value of `result` is calculated as the factorial.
- Line 7: The value of `result` is returned.

Listing 66: A function to safely calculate the factorial.

```
1 def factorial(x):
2     if not isinstance(x, int):
3         return -1
4     result = 1
5     for value in range(x):
6         result *= value + 1
7     return result
```

Listing 66 contains a unit test that verifies that the `factorial` function behaves as expected. In this listing:

- Line 1: The `unittest` module is imported.
- Line 2: The `unit_test_module` module is imported. This imports the Python that is given in Listing 66.
- Line 14: The `main` function of the `unittest` module is run.
- Line 6: The class `TestModule` inherits from the `unittest.TestCase` class. Therefore, each test function that is defined in the `TestModule` class is run when the `main` function of the `unittest` module is called.
- Line 7: The `factorial` function is passed the value `3` and the resulting value is assigned to `value`.
- Line 8: If the value that is stored in `value` is `6`, then the test continues. If the value is not `6`, then the test fails.
- Line 9: The `factorial` function is passed the value `3.14159` and the resulting value is assigned to `value`. The value `3.14159` is a floating point number. Therefore, the `factorial` function should return `-1`.
- Line 10: If the value that is stored in `value` is `-1`, then the test passes. If the value is not `-1`, then the test fails.

Listing 67: Using a unit test to verify expected performance.

```
1 import unittest
2 import unit_test_module
3
4
5 class TestModule(unittest.TestCase):
6     def test_factorial(self):
7         value = unit_test_module.factorial(3)
8         self.assertEqual(6, value)
9         value = unit_test_module.factorial(3.14159)
10        self.assertEqual(-1, value)
11
12
13 if __name__ == '__main__':
14     unittest.main()
```

21 Command-line arguments

It is sometimes necessary to pass command-line arguments into a program when it is run. These command-line arguments are accessible within a Python program and can be used to control how the program behaves. A Python program can be passed command-line arguments by supplying them after the name of the Python file, as shown in Listing 68. In this Listing, three command line arguments follow the name of the Python file.

Listing 68: Providing command-line arguments.

```
python cmd_line_args.py first second third
```

Listing 69 demonstrates a simplistic way of accessing command-line arguments. In this Listing:

- Line 1: The `os` module is imported.
- Line 2: The `main` entry point is required, which is discussed in Section 19.2.
- Line 3: The number of command-line arguments is calculated as the length of the `os.sys.argv` list minus 1. The name of the executable is the first entry within the `argv` list and any command-line arguments follow.
- Line 4: The number of command-line arguments is printed with some description.
- Line 5: The name of the python file that was run is printed with some description.
- Line 6: If there are command-line arguments, then Line 7 is run.
- Line 7: The command-line arguments are printed together with some description.

Listing 69: Accessing command-line arguments.

```
import os
if __name__ == "__main__":
    n = len(os.sys.argv) - 1
    print(str(n) + " command line arguments.")
    print("Program name: \"" + os.sys.argv[0] + "\"")
    if n > 0:
        print("Arguments: " + str(os.sys.argv[1:]))
```

For more complex command-line options, the `getopt` module can be used. The `getopt` module is documented at https://docs.python.org/3/library/getopt.html.

22 Comments

When a program is written, it should be accompanied by comments. These comments are not evaluated by the Python interpreter, but are useful for the developers and for the `pydoc` program. Comments should explain the purpose of the code, rather than state exactly what the Python syntax represents. Developers should be able to read the Python syntax themselves.

Python allows multiline and single-line comments. Multiline comments start and end with """. They should be indented in line with the lines of program that they refer to. Single-line comments start from a # character and continue to the end of the line. This implies that a single-line comment may be used on the same line as some functional code. Single-line comments should be used to explain the purpose of the source code within a section of the program.

A Python program comprises modules, functions and classes. At its most basic, a Python program is a single module or Python file. Multiline comments should be used to document modules, functions and classes. These comments should be written such that they are useful to the `pydoc` program. They should be indented within a function or class that they belong to.

Listing 70: Demonstrating comment types

```
"""
A module to demonstrate inheritance.
"""


class Server:
    """
    A base class for a server.
    """

```

```
11      def __init__(self, ip_address, port_number, protocol):
12          self.ip_address = ip_address
13          self.port_number = port_number
14          self.protocol = protocol
15
16      def as_dict(self):
17          """
18          A function to return a dictionary form of the object.
19          """
20          dict_version = {}
21          dict_version["ip_address"] = self.ip_address
22          dict_version["port_number"] = self.port_number
23          dict_version["protocol"] = self.protocol
24          return dict_version
25
26
27  class WebServer(Server):
28      """
29      A specialisation of the server class, which has a fixed
30      protocol "http".
31      """
32      def __init__(self, ip_address, port_number):
33          Server.__init__(self, ip_address, port_number, "http")
34
35
36  """
37  A program to create a web server and print its
38  data members.
39  """
40  web_server = WebServer("127.0.0.1", 80)
41
42  # Provide the user with the current configuration.
43  print(web_server.as_dict())
```

23 Summary

This document introduces some of the features of the Python computer programming language. Many other supporting libraries exist that can be used for data analysis or data access. The Python programming language can be used to interface with other computer programming languages, such as the C computer programming language. Further reading and programming is needed to explore these features.

24 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.
- **Deep copy** - All layers of a mutable variable or object are copied. For example, a deep copy of a list creates two separate lists, where the mutable variables or objects in the first list are copied too such that the two lists are completely separate.

- **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.
- **Shallow copy** - The top layer of a mutable variable or object is copied. For example, a shallow copy of a list creates two separate lists, where the mutable variables or objects in the first list are not separate from those in the second list.
- **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.