# An introduction to the Python programming language

W. H. Bell

2020/08/19

**Abstract**

The basics of the Python programming language are 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.

# Contents

# 1 Introduction

The Python programming language is widely used for data analysis and data science. It is an interpreted programming language that has a large range of libraries associated with it. These libraries provide graphical user interface, web service, data analysis, mathematical and data visualisation functionality. There are also database drivers to allow connections to traditional relational databases and NoSQL solutions.

Python is not compiled, but is executed with the Python Interpreter. This implies that algorithms that have been written in Python are approximately twice as slow as those that have been implemented in a compiled language, such as C or C++. Python libraries are often written in C, such that more CPU or I/O intensive functionality is implemented efficient. The Python language supports coupling Python to C, C++ and other languages, allowing existing libraries to be reused.

This document discusses a series of programming examples that illustrate the functionality of the Python programming language. Each example introduces new concepts and builds on concepts that are introduced in previous examples. These examples should be run and experimented with by modifying the source code, such that the functionality is understood.

# 2 Reference material

Reference material is available for Python 3 online at:

- https://docs.python.org/3/ [1]
- https://www.w3schools.com/python/ [2]

The w3schools web site provides both example code and the ability to try out some of the functionality.

# 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 around string 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 and Mac (OSX) operating systems.

## 3.1 Microsoft Windows

Python can be installed on Microsoft Windows using Anaconda [3]. Anaconda provides a large range of libraries that are used for data science and other activities.

### 3.2 Linux

Python 3 can be installed using the package manager of all major Linux distributions. Refer to documentation for specific Linux distributions to perform the installation.

## 4 Using Python

Python can be used interactively or to execute a Python script.

### 4.1 Interactive shell

To start Python interactively, a Python shell can be started from Anaconda or on Linux or OSX. This opens a prompt, as shown in Listing 1.

Listing 1: The Python prompt.

```
1 >>>
```

Once a Python prompt has been opened, Python programming commands can be typed as illustrated in Listing 2.

Listing 2: Using the Python prompt.

```
1 >>> print("Hello World")
2 Hello World
3 >>> x = 4
4 >>> x + 2
5 6
```

When a Python command prints information to the standard output or standard error, it is shown below the command. This can be seen at Line 1 and 2, where the print command is used to print to the standard output.

When a Python operator or function returns a value and the value is not assigned to a variable, the value is printed below the operator or function. This can be seen at Line 4 and 5. The value of 4 is assigned to the variable x at Line 3. When 2 is added to the value of 4, it is not assigned to a variable. Therefore, it is printed on the screen at Line 5.

### 4.2 Executing scripts

While the Python shell provides an ideal user interface to experiment with commands, it is often necessary to re-run commands or particular Python scripts. Python scripts can be created with a text editor or integrated development environment, such as offered using Anaconda. Once the script has been created it can be executed using the Python interpreter.

A first script is given in Listing 3. Line 1 contains a comment that informs Linux or OSX how to execute the script. Line 3 prints a text string to the standard output. This implies that the text is printed to the screen, unless it is redirected to a file by the user.

Listing 3: HelloWorld.py

```
1 #!/usr/bin/env python3
2
3 print("Hello World")
```

The rest of this document assumes that the examples provided are run as Python scripts. Some more examples of commenting Python programs are given in Section 14.

# 5 Functions

## 5.1 The main function

When a Python script is executed using the Python interpreter, it evaluates each command in order starting from the top of the Python script file. When a Python script is within a package, it may be necessary to execute the script or include it within another script by importing functionality. To cause the script to be executed when called, a main function is defined using the syntax given in Listing 4.

Listing 4: Main.py

```
1 def main():
2     print("def main")
3
4 if __name__ == "__main__":
5     main()
```

A function called `main` is defined at Line 1 of Listing 4. This function includes one `print` function call to print a text string on the screen. Unlike other languages, such as C, C++, C#, Java and PHP, Python relies on indenting to describe the structure of the program. The `print` function at Line 2 is indented by four spaces. Python will tolerate different numbers of spaces being used to indent source code, as long as the indenting is consistent within a Python script file. This document and the examples presented use four spaces for each indent, since this is commonly used by other Python programmers.

Once a function is defined, it can be called elsewhere in a program. The `main` function is called at Line 5 of Listing 4. The `__name__` variable is automatically set to be equal to `"__main__"` when the script is executed. Therefore, the if condition is true when the script is executed. Similar to the function definition, Line 5 is indented since the `main` function should only be called if the `if` statement is true. Conditional statements are discussed more in section 8.1.

## 5.2 Defining other functions

Listing 5 contains three functions. The `main` function calls `func1`, passing the text string `"Functions"` to `func1`. Then `func1` passes the text string to `func2`, using a variable called `stringValue`. The function `func2` returns a text string value that is assigned to the variable `returnValue`, which is then printed to the standard output. The function `func2` returns the text string that is passed to it.

Listing 5: Functions.py

```
1 def func1(stringValue):
2     returnValue = func2(stringValue)
3     print(returnValue)
4
5 def func2(stringValue):
6     return stringValue
7
8 def main():
9     func1("Functions")
10
11 if __name__ == "__main__":
12     main()
```

The purpose of Listing 5 is to demonstrate how to pass a text string to a function and return a value. Python functions may or may not return a value. If a function does not explicitly return a value, then the value `None` is returned. The value `None` is commonly referred to as null in other programming languages and is the absence of a value.

# 6 Basic data types

## 6.1 Text strings

When writing computer programs, variables that contain text are normally referred to as strings. An example of string variables is given in Listing 6.

Listing 6: StringVariables.py

```
1 def stringVariables(stringValue):
2     stringValue = "A value"
3     print(stringValue)
4
5     stringValue += ",Another value"
6     print(stringValue)
7
8     firstValue = "First value"
9     stringValue = firstValue + "\t" + stringValue + "  \n" + "A
          new line."
10    print(stringValue)
11
12    stringValue = stringValue.replace("First","Primary")
13    stringValue = stringValue.replace("\t",",")
14    stringValue = stringValue.split("\n")[0]
15    stringValue = stringValue.strip()
16    print(stringValue)
17
18    searchString = "value"
19    print("The string \"" + searchString + "\" was found at
          index " + str(stringValue.find(searchString)))
20
21 if __name__ == "__main__":
22    stringValue = ""
23    stringVariables(stringValue)
24    print("stringValue=\"" + stringValue + "\"")
```

The Listing 6 contains a single function called `stringVariables`, which is passed a single string variable called `stringValue`. The name used for this variable within the function does not have to match the name used outside the function.

Within the function `stringVariables`, a new value is assigned to `stringValue` at Line 2. Then more characters are appended to `stringValue` at Line 5. Python supports string appending using both the `+` and `+=` operator. The `+=` operator appends to the string variable and assigns the result to the original variable.

The `+` operator is used at Line 9 to append several characters together, these includes a tab character (`\t`) and a newline character (`\n`).

Some of the string functions are demonstrated from Line 12 to 15. These are the string `replace` function, the `split` function and the `strip` function. The `replace` function performs a string find and replace operation, returning the resulting string. The `split` function splits the string into a list of strings, using a delimiter which can be specified. In this example, the string is split using the newline character. The first element of the resulting list is used by specifying it with `[0]`. (Lists are discussed in Section 7.1 of this document.) The `strip` function removes leading and trailing white space from the string.

Line 19 demonstrates how to use the `find` function to find a string within a string. The `find` function returns -1 if the string is not found or the position of the string within the string if the string is found. Since the value returned by `find` is a numeric value, it is cast to a string using the `str` function before it can be combined with a string. Once the value has been cast to a string, it is appended to other strings and printed to the standard output.

After the function `stringVariables` has been called, the value of the variable `stringValue` is printed out at Line 24. None of the modifications that are made within the function `stringVariables` affect this output, since a string variable is passed by value into the function rather than by reference. String variables are therefore said to be immutable.

## 6.2 Numeric types

An example of numeric types is given in Listing 7. The `math` library is imported to allow the use of `math.cos` function and the `math.pi` constant. The sample program includes a single function called `numericVariables`. The function `numericVariables` is passed two numeric variables, called `firstValue` and `secondValue`. The values of these two variables are modified within the function `numericVariables`. However, this has no effect on their values in the calling code, since the variables are passed by value rather than by reference. The `print` statement at Line 20 demonstrates this by printing the values of the variables. Similar to strings, numeric variables are also immutable.

Listing 7: NumericVariables.py

```
1 import math
2
3 def numericVariables(firstValue, secondValue):
4     firstValue = firstValue + 1
5     firstValue += 1
6     firstValue *= 2
7     firstValue /= 3
```

```
 8     print(str(firstValue))
 9
10     firstValue = int(firstValue)
11     print(firstValue)
12
13     secondValue = math.cos(math.pi/2.)
14     print(secondValue)
15
16 if __name__ == "__main__":
17     firstValue = 0
18     secondValue = 0.
19     numericVariables(firstValue, secondValue)
20     print("firstValue=" + str(firstValue) + ", secondValue=" +
          str(secondValue))
```

The variables `firstValue` and `secondValue` are defined at Line 17 and 18, by assigning a numeric value to these variables. Once a value has been assigned, the two variables are numeric variables. Therefore, to combine them with a text string, they have to be cast to text strings by using the function `str`.

The `numericVariables` function adds 1 to the value of `firstValue` at Line 4 and 5. The operator `+` adds the two values together, whereas the operator `+=` is used to add the value to the existing `firstValue` variable and assigns the result back to `firstValue`.

Listing 7 also demonstrates how to cast a floating point number to an integer by using the `int` function. This rounds down to the nearest whole number. Line 13 demonstrates how to use the cosine function and print its result.

## 6.3 Boolean type

Python provides a boolean type that can be used to store the value `True` or `False`. A boolean value can be filled with `True` or `False` or using a conditional statement. Listing 8 demonstrates how a boolean can be used.

Listing 8: BooleanVariables.py

```
 1 def booleanVariable(booleanValue):
 2     booleanValue = True
 3     booleanValue &= False
 4     print("booleanValue = " + str(booleanValue))
 5
 6     booleanValue = True | False
 7     print("booleanValue = " + str(booleanValue))
 8
 9     booleanValue = 1 > 0
10     print(booleanValue)
11
12 if __name__ == "__main__":
13     booleanValue = False
14     booleanVariable(booleanValue)
15     print("booleanValue = " + str(booleanValue))
```

Listing 8 creates a variable called `booleanValue`. This variable is assigned `False`. The function `booleanVariable` is then called. Following the function call, the value of `booleanValue` is printed to the standard output.

The function `booleanVariable` assigns the value `True` to `booleanValue`. This is then combined with `False`, using a bitwise logic AND operator `&=`. This forms the bitwise and with the value in `booleanValue` and then assigns the result to `booleanValue`.

Similar to strings and simple numeric types, booleans are immutable. Therefore, the value of `booleanValue` remains equal to `False` after the `booleanVariable` function call.

# 7 Basic data structures

Python programmers can access many different data structures, where this documentation only covers a few of the basic structures.

## 7.1 Lists

A list provides sequential storage of zero or more values. Elements of a list can be removed and new ones can be assigned. Lists can be sorted or searched for particular values. Unlike basic variable types such as strings, integers or floats, lists are passed by reference into functions. They are said to be mutable. Therefore, a copy of a list should be made if the original values need to be retained after a function call performs some modifications.

The Listing 9 introduces several features of Python lists. The example, includes a single function called `lists`. Before this function is called, the list `listValues` is initialised with two zeros at Line 20. The syntax `[v]*n` causes a list to be created with `n` elements, where each element contains a copy of the value `v`. The list is copied at Line 22 to demonstrate that lists are passed by reference to functions.

Listing 9: NumericVariables.py

```
1  def lists(listValues):
2      print(">> Start of lists")
3
4      del listValues[:]
5
6      listValues += [ 15.0 ]
7      print(listValues)
8
9      listValues.pop()
10     print(listValues)
11
12     listValues += range(5)
13     print(listValues)
14
15     listValues += range(4)
16     listValues.sort()
17     print(">> End of lists")
18
19 if __name__ == "__main__":
20     listValues = [0]*2
21
22     copyList = listValues.copy()
23
24     print("List values before: " + str(listValues))
```

```
25      lists(listValues)
26      print("List values after: " + str(listValues))
27      print("Copy of original list: " + str(copyList))
28
29      print("The list contains " + str(len(listValues)) + "
            elements.")
30      print("The third element: " + str(listValues[2]))
31      print("The last element: " + str(listValues[-1]))
32      print("The fourth and fifth elements: " + str(listValues
            [5:7]))
```

The function `lists` deletes all of the elements of the `listValues` list at Line 4. Then a new element is appended at Line 6. The new element is then removed at Line 9. Line 12 uses the `range` function to create a list that contains the values 0 to 4. These values are appended to `listValues`. Then another range of 0 to 3 is appended to `listValues` at Line 15. The values in `listValues` are then sorted.

Python lists can be used to store basic types or more complicated types, such as lists, dictionaries or objects.

## 7.2 Dictionaries

Python dictionaries are used to store a key and value pair. The key value might be an integer or string value, whereas the value could be a simple or more complicated type. The Listing 10 introduces dictionaries and some of their functionality.

Listing 10: Dicts.py

```
1  def dicts(dictValues):
2      print(">> Start of dicts")
3
4      dictValues.clear()
5
6      dictValues["localhost"] = "127.0.0.1"
7      dictValues["googleDNS1"] = "8.8.8.8"
8      dictValues["googleDNS2"] = "8.8.4.4"
9      dictValues["myMachine"] = "192.168.1.66"
10     print(dictValues)
11
12     dictValues.pop("myMachine")
13     print(dictValues)
14
15     keys = list(dictValues.keys())
16     print("Keys : " + str(keys))
17
18     values = list(dictValues.values())
19     print("Values : " + str(values))
20
21     print(">> End of dicts")
22
23 if __name__ == "__main__":
24     dictValues = {}
25     dictValues["AnotherMachine"] = "10.0.0.12"
26
27     copyDict = dictValues.copy()
```

```
28
29      print("Dictionary values before: " + str(dictValues))
30      dicts(dictValues)
31      print("Dictionary values after: " + str(dictValues))
32      print("Copy of original dictionary: " + str(copyDict))
```

The Listing 10 creates a dictionary called `dictValues`. One element is added that contains a host name and an IP address string. The dictionary is then copied, such that it can be used after the function call. The `dicts` function is called to update the dictionary. After this function, the values in `dictValues` and `copyDict` are printed to the standard output.

The `dicts` function clears the dictionary that has been passed to it. Similar to lists, dictionaries are passed by reference into functions and are said to be mutable. Therefore, the `dicts` function clears the single element that was added before the function was called. The `dicts` function then adds four elements, removes a selected element and prints the key and values separately.

## 7.3 Tuples

A tuple is an ordered collection that cannot be modified once it has been created. They are said to be immutable. Tuples may be returned by functions or used in function calls. An example of a tuple is given in Listing 11.

Listing 11: Tuples.py

```
1 def buildTuple():
2     return ("Database", "WebService", "Client")
3
4 if __name__ == "__main__":
5     tupleValues = buildTuple()
6
7     print(tupleValues)
8     print(tupleValues[1:3])
```

Listing 11 calls the function `buildTuple`, which returns a tuple. All of the values and a selected range of values are then printed to the standard output.

## 7.4 Sets

A set is an unordered and unindexed collection. Listing 12 demonstrates some of the features of sets. Similar to lists, they are passed by reference and are said to be mutable.

Listing 12: Sets.py

```
1 def sets(setValues):
2     print(">> Start of sets")
3     print(setValues)
4
5     setValues.pop()
6     print(setValues)
7
8     setValues.add("Randomise")
9
```

```
10      print(">> End of sets")
11
12  if __name__ == "__main__":
13      setValues = { "Query", "Sort", "Save"}
14      copySet = setValues.copy()
15
16      print("Set values before: " + str(setValues))
17      sets(setValues)
18      print("Set values after: " + str(setValues))
19      print("Copy of original set: " + str(copySet))
```

Listing 12 creates a set called `setValues`. This set is copied, such that it can be compared with the original set after modifications have taken place. Then the function `sets` is called to update the values of the set. The function `sets` removes a random element from the set and adds a new one.

# 8 Conditions and loops

## 8.1 Conditions

Conditions are used to control the flow of a program. Python supports several different logic conditions, where some of these are used in Listing 13.

Listing 13: Conditions.py

```
1  def returnsNone():
2      return
3
4  def conditionalOutput(inputValue):
5      if inputValue > 0:
6          return inputValue * 2
7      return None
8
9  def valueInList(inputList, value):
10     if value in inputList:
11         return value
12
13 def powerState(powerName):
14     powerValue = 0
15     if powerName == "On":
16         powerValue = 1
17     elif powerName == "Off":
18         powerValue = 0
19     else:
20         return None
21     return powerValue
22
23 def ipAddress(machineName):
24     addresses = {}
25     addresses["localhost"] = "127.0.0.1"
26     addresses["GoogleDNS"] = "8.8.8.8"
27     if machineName in addresses:
28         return addresses[machineName]
29     return None
```

```
30
31 def notNone(value):
32     return not value is None
33
34 def notZero(value):
35     return value != 0
36
37 def conditions():
38     if returnsNone() is None:
39         print("returnsNone returned None")
40
41     inputValue = 2
42     print("conditionalOutput(" + str(inputValue) + ") = " + str(
           conditionalOutput(inputValue)))
43
44     inputValue = -1
45     print("conditionalOutput(" + str(inputValue) + ") = " + str(
           conditionalOutput(inputValue)))
46
47     powerName = "On"
48     print("powerState(\"" + powerName + "\") = " + str(
           powerState(powerName)))
49
50     machineName = "localhost"
51     print("ipAddress(\"" + machineName + "\") = \"" + str(
           ipAddress(machineName)) + "\"")
52
53     value = None
54     print("notNone(" + str(value) + ") = " + str(notNone(value))
           )
55
56     value = -5
57     print("notZero(" + str(value) + ") = " + str(notZero(value))
           )
58
59 if __name__ == "__main__":
60     conditions()
```

Listing 13 calls the `conditions` function. This function calls other functions to demonstrate some commonly used conditions.

The `returnsNone` function always returns `None`. This is the absence of a value, sometimes referred to as null in other programming languages. The value returned from `returnsNone` is compared with `None` at Line 38.

The `conditionalOutput` function returns a value or `None` depending on the input value. This is achieved using a numeric greater than (`>`) comparison.

The `valueInList` function returns the value if the value is found in the list and `None` if the value is not found in the list. If a function does not implement a specific return type, then it returns `None`.

The `powerState` function compares the input `powerName` variable with different values and either returns a numeric value or `None`.

The `ipAddress` function creates a dictionary that contains two host names and their

associated IP addresses. Then it tests to see if the `machineName` provided has a matching dictionary key. If the dictionary key is found, the associated IP address is returned. If the dictionary key is not found, then the function returns `None`.

The `notNone` function returns `True` if the value is not `None` and `False` if it is `None`.

The `notZero` function returns `True` if the supplied value is not equal to zero.

The conditional operators are summarised in section 9.3, whereas the membership operators are given in section 9.4.

## 8.2 Loops

Operations that need to be repeated can be efficiently implemented using loops. Python allows several styles of loop, where two are shown in Listing 14.

Listing 14: Loops.py

```
1  def loops():
2      for i in range(10):
3          print("i = " + str(i))
4
5      values = range(10)
6      i = len(values) - 1
7      while i > 0:
8          if values[i] == 3:
9              break
10
11         print("values[" + str(i) + "] = " + str(values[i]))
12         i -= 1
13
14     sum = 0
15     for i in range(10):
16         for j in range(4):
17             if sum > 20:
18                 break
19             sum += j
20
21         print("sum = " + str(sum))
22
23 if __name__ == "__main__":
24     loops()
```

The Listing 14 calls the function `loops`. This function contains several `for` loops and a `while` loop. The first `for` loop uses values of `i` from 0 to 9. The `range(10)` function creates a list of values from 0 to 9. The loop prints the values of `i` to the standard output.

Before the `while` loop, `range` is used to initialise a list called `values`. The value of `i` is set to be the index of the last element in the `values` list. The `while` loop continues while the value of `i` is greater than zero. However, within the loop there is a conditional statement that breaks out of the loop if the value of `i` is equal to 3. While the loop continues, the value of `i` and the value within the `values` list is printed. At the end of the while loop, the value of `i` is reduced by 1.

After the `while` loop, there is a `for` loop at Line 15. This `for` loop contains another `for` loop at Line 16. The inner `for` loop exits when the `sum` variable is greater than 20. The

purpose of this loop is to demonstrate that `break` only breaks out from the current loop. The value of `sum` is printed to demonstrate this.

# 9 Operators

Similar to other programming languages, Python supports numeric and bitwise operators that can be used to operate on numeric or boolean data types.

As discussed in Section 6.1, when the numeric addition operator is used with string variables it appends one string to another.

## 9.1 Assignment operator

The assignment operator `=` can be used with all types, where its meaning depends on the type used. For simple types that are mutable, such as numbers, string and booleans, the assignment operator copies the value into another variable. However, for more complex types that are mutable, the assignment operator copies a reference to the original variable. Therefore, an explicit copy may be needed to avoid unintentionally updating data values.

An operation that results in a modification of a variable and re-assignment, can be shortened:

- `variable += value` $\implies$ `variable = variable + value`
- `variable -= value` $\implies$ `variable = variable - value`
- `variable *= value` $\implies$ `variable = variable * value`
- `variable /= value` $\implies$ `variable = variable / value`
- `variable %= value` $\implies$ `variable = variable % value`
- `variable **= value` $\implies$ `variable = variable ** value`
- `variable //= value` $\implies$ `variable = variable // value`
- `variable &= value` $\implies$ `variable = variable & value`
- `variable |= value` $\implies$ `variable = variable | value`
- `variable ^= value` $\implies$ `variable = variable ^ value`
- `variable >>= value` $\implies$ `variable = variable >> value`
- `variable <<= value` $\implies$ `variable = variable << value`

Unlike some other programming languages, Python does not allow `variable++` or `variable--`. The allowed operators are defined in the following subsections.

## 9.2 Numeric operators

Numeric operators can be used with both floating point and integer numbers. The list of available operators is summarised in Table 1.

## 9.3 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

Table 1: Numeric operators.

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

summarised in Table 2.

Table 2: 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 |

## 9.4 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 3.

Table 3: 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 |

## 9.5 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 4.

Table 4: Bitwise operators.

| Operator | Name |
|---|---|
| & | AND |
| \| | OR |
| ^ | Exclusive OR (XOR) |
| ~ | NOT |
| << | Left shift |
| >> | Right shift |

# 10 Classes

In a similar manner as other languages that support object orientated programming, Python allows classes to be created. A class can contain just data members, just functions or a mixture of both. A class should have a clear purpose within the design of a program, where some classes are more data rich than others.

The Listing 15 demonstrates how to implement classes. The example program contains two classes `Computer` and `SuperComputer`. The class `SuperComputer` inherits from the class `Computer`. The class `Computer` contains three public data members `address`, `cpu` and `ram` and one private data member `__privateInfo`. The private data member is only accessible within an object of the class `Computer`, whereas the other data members are accessible to other source code.

Python classes can contain several standard functions that have specific purposes. The constructor is called `__init__`. A non-static class function, such as the constructor, must have `self` as its first argument in a function definition. This refers to the current object instance. For a default constructor, `__init__` only has an argument `self`. In the presented example, the constructors of `Computer` and `SuperComputer` both require a single value `cpu`.

When a Python object is cast to a string by using the `str` function, the `__str__` member function is called. Python expects that this function will return a string. Both `Computer` and `SuperComputer` classes contain a `__str__` member function to allow the contents of an object to be printed easily.

If a Python list or other compound object is cast to a string then Python calls the `__repr__` function. This function should also return a string. This function has been implemented for `Computer` and `SuperComputer` to allow the contents to be printed.

When referring to data members or member functions within a class, the `self` prefix must be used. This is needed to differentiate between data and functions that belong to the class or are only defined within the specified scope.

Listing 15: Classes.py

```
1 class Computer():
2     def __init__(self, cpu):
3         self.address = ""
4         self.cpu = cpu
5         self.ram = 0
6         self.__privateInfo = "secrets"
7 
8     def __str__(self):
```

```
 9         stringValue = "{address=\"" + self.address + "\""
10         stringValue += ", cpu=" + str(self.cpu)
11         stringValue += ", ram=" + str(self.ram)
12         stringValue += ", privateInfo=" + str(self.__privateInfo
               )
13         stringValue += "}"
14         return stringValue
15
16     def __repr__(self):
17         return self.__str__()
18
19 class SuperComputer(Computer):
20     def __init__(self, cpu):
21         Computer.__init__(self, cpu * 10)
22         self.ram = 100
23
24 def classes():
25     pc = Computer(2)
26     pc.address = "127.0.0.1"
27     pc.ram = 10
28     pc.__privateInfo = "hacked"
29     print("pc=" + str(pc))
30
31     superPC = SuperComputer(3)
32     print("superPC=" + str(superPC))
33
34     farm = []
35     for i in range(10):
36         pc = Computer(4)
37         pc.ram = 4
38         pc.address = "192.168.0." + str(10 + i)
39         farm += [ pc ]
40
41     print("farm=" + str(farm))
42
43 if __name__ == "__main__":
44     classes()
```

The Listing 15 calls the function `classes`. The `classes` function creates a `Computer` and `SuperComputer` object. The code attempts to set the private data member from outside the class, which fails. Then the values stored in the objects are printed to the screen.

The list `farm` is used to demonstrate how to create a list of objects. Once the list has been filled, the contents of the list is printed to the standard output.

## 11 Unit tests

Software should be developed with matching unit tests. The purpose of a unit test is to verify that a unit of software works as expected. This is particularly useful when several developers are working on the same piece of software or when a piece of software is developed over a long period of time.

The Listing 16 demonstrates how to use unit tests in Python. Normally, one unit test class

should be created to match one class that has been implemented in a program. Ideally, the name of the test class should match the name of the class it is testing, where a "Test" prefix is added to the test class name. In this example program, existing mathematical functions have been used for simplicity.

Listing 16: UnitTests.py

```
import unittest
import math

class TestMathMethods(unittest.TestCase):
    def test_true(self):
        self.assertTrue(True)

    def test_sin(self):
        self.assertEqual(1.0, math.sin(math.pi/2.0))

    def test_cos(self):
        self.assertEqual(0.0, math.cos(math.pi/2.0))

    def test_sqrt(self):
        self.assertEqual(3, math.sqrt(9.0))

if __name__ == '__main__':
    unittest.main()
```

The Listing 16 calls the `unittest.main` function. Since the class `TestMathMethods` inherits from the class `unittest.TestCase`, each of the test functions in the class are called in turn.

The `unittest.TestCase` provides the functions `assertTrue` and `assertEqual`. The function `assertTrue` requires that the value passed to it is `True`. This could be the result of a logical comparison. In this simple example, `True` is passed to the function.

The function `assetEqual` is used to compare two values and require that they must be equal to each other. In this example, results from three mathematical operators are compared with expected values.

## 12 Command-line inputs

It can be useful to be able to control a program using command-line inputs. Python supports command-line inputs in a similar manner as C or C++. The Listing 17 demonstrates some of this functionality.

Listing 17: CmdInputs.py

```
import getopt
import os
import sys

def usage(scriptName):
    print("Usage: " + scriptName + " [options]" + "\n")
    print("where [options] includes:")
    print("\t" + "-h, --help")
    print("\t" + "-o, --output <file name>" + "\n")
```

```
10
11 def main():
12     print("argv=" + str(sys.argv))
13     if len(sys.argv) == 1:
14         return
15
16     scriptName = os.path.basename(sys.argv[0])
17     print("scriptName=\"" + scriptName + "\"")
18     print()
19
20     try:
21         opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help"
               , "output="])
22     except getopt.GetoptError as err:
23         usage(scriptName)
24         sys.exit(2)
25
26     output = ""
27     for option, value in opts:
28         if option in ("-h", "--help"):
29             usage(scriptName)
30             sys.exit()
31         elif option in ("-o", "--output"):
32             output = value
33         else:
34             assert False, "unhandled option"
35
36     print("output=\"" + output + "\"")
37
38
39 if __name__ == "__main__":
40     main()
```

The Listing 16 calls the `main` function. The `main` function retrieves the command-line arguments using `sys.argv`. When a script is run without additional command-line arguments, `sys.argv` contains one element which is the path of the script that was run. Any command-line arguments are stored in subsequent elements of `sys.argv`.

The Listing 16 demonstrates how to use `getopt` to specify and read the options `"-h"`, `"--help"`, `"-o"` and `"--output"`. If the `"-o"` or `"--output"` option is not followed by a string or the user provides `"-h"` or `"--help"`, the usage is printed by calling the function `usage`.

The example program includes a `try` and `except` clause. This is needed to catch a possible exception. Several commands in Python may throw exceptions. A Python program should catch exceptions that might occur and provide the user with information as to what occurred. More documentation on possible exceptions is given in function reference documentation [1] and [2].

## 13 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.

## 13.1 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. The Listing 18 demonstrates how to write a text file line by line and read it back into an array of lines.

Listing 18: FileIO.py

```
import datetime

def writeFile(fileName):
    outputFile = open(fileName, "w")
    outputFile.write("Data file" + "\n")
    outputFile.write("Date:" + str(datetime.datetime.utcnow()) +
        "\n")

    ipAddresses = []
    ipAddresses += [ "127.0.0.1" ]
    ipAddresses += [ "8.8.8.8" ]
    ipAddresses += [ "8.8.4.4" ]

    values = ""
    for ipAddress in ipAddresses:
        if len(values) == 0:
            values += ipAddress
        else:
            values += "\t" + ipAddress

    outputFile.write("Values:" + values + "\n")
    outputFile.close()

def readFile(fileName):
    inputFile = open(fileName, "r")
    allLines = inputFile.readlines()
    inputFile.close()

    dataRead = {}
    dataRead["Date"] = ""
    dataRead["Values"] = ""
    for line in allLines:
        line = line.strip()

        for key in dataRead.keys():
            if line.startswith(key + ":"):
                dataRead[key] = line[len(key)+1:]
                break

    print("dataRead=" + str(dataRead))
    values = dataRead["Values"].split("\t")
    print("values=" + str(values))

def fileIO():
```

```
44      fileName = "fileIO.txt"
45      writeFile(fileName)
46      readFile(fileName)
47
48  if __name__ == "__main__":
49      fileIO()
```

The Listing 18 calls the function `writeFile` to write a text file that contains some data values. Then it calls `readFile` to read the text file and print the values to the standard output.

The `writeFile` function creates a file and then writes data to it. If the file already exists, it is truncated and overwritten. The `write` function is used to write a string to the output file. To complete the line, a newline character is appended to the string. The output data comprises a title, the current date and three IP addresses. The text file `fileIO.txt` is written in the present working directory.

The `readFile` function opens the file written by `writeFile` as a file to be read. It then reads all of the lines into a list and the input file is closed. A dictionary is created to hold the data labels `"Date"` and `"Values"` and the values that are associated with them. The function `strip` is used to remove the newline character. The `startswith` function is used to match the dictionary key against the start of a line. Then the rest of the line is assigned as the dictionary value. At the end of the data input, the values that have been read are printed to the standard output.

## 13.2 CSV files

Comma separated value (CSV) files are often used to store data, taken from equipment or as an intermediate format. The Listing 19 demonstrates how to write a CSV file using the standard Excel format and read the CSV data back in again.

Listing 19: CsvIO.py

```
1  import csv
2
3  def writeCSV(fileName):
4      csvFile = open(fileName, "w", newline='')
5      csvWriter = csv.writer(csvFile, delimiter=',', quotechar='"'
           , quoting=csv.QUOTE_NONNUMERIC)
6
7      dataTable = {}
8      dataTable["Host name"] = ["localhost", "GoogleDNS"]
9      dataTable["IP address"] = ["127.0.0.1", "8.8.8.8"]
10     columnNames = list(dataTable.keys())
11
12     csvWriter.writerow(columnNames)
13
14     nRows = len(dataTable[columnNames[0]])
15
16     for i in range(nRows):
17         row = []
18         for columnName in columnNames:
19             row += [ dataTable[columnName][i] ]
20         csvWriter.writerow(row)
```

```
21      csvFile.close()
22
23  def readCSV(fileName):
24      csvFile = open(fileName, "r", newline='')
25      csvReader = csv.reader(csvFile, delimiter=',', quotechar='"'
            )
26
27      dataTable = {}
28      columnNames = []
29      numberOfColumns = 0
30      for row in csvReader:
31          if len(columnNames) == 0:
32              columnNames += row
33              numberOfColumns = len(columnNames)
34              for columnName in row:
35                  dataTable[columnName] = []
36              continue
37
38          for i in range(numberOfColumns):
39              if i >= len(row):
40                  break
41              dataTable[columnNames[i]] += [ row[i] ]
42
43      csvFile.close()
44
45      print("dataTable=" + str(dataTable))
46
47  def csvIO():
48      fileName = "data.csv"
49      writeCSV(fileName)
50      readCSV(fileName)
51
52  if __name__ == "__main__":
53      csvIO()
```

The Listing 19 calls the function `writeCSV` to write two columns to a CSV file. The function opens an output text file without a new line character. Then the `csv.writer` function is used to configure the output to use a comma delimiter and double quotes for non-numeric values. These are the settings that Excel uses for CSV files by default, which are robust to additional commas being written within output strings.

The data to be written are stored in a dictionary, where the key name of the dictionary is the column name in the CSV file and the values are lists of entries for the given column. The rows of the CSV file are written by calling the `writerow` function, first for the header row and then for each row of values that is stored within the dictionary. The CSV file is written within the present working directory.

The `readCSV` function is called to open the CSV file and read the data back into a dictionary. The first row of the file is assumed to contain the column names. The order of the column names in the first row is then used to assign the values to the dictionary. A conditional statement is used to prevent the index that is used with the `row` list from being out of range.

## 13.3 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 Listing 20 demonstrates how to write and read two classes from a JSON file.

Listing 20: JsonIO.py

```
1  import json
2
3  class DataSet():
4      def __init__(self):
5          self.name = ""
6          self.dataSlices = []
7
8      def __str__(self):
9          stringValue = "{name=" + self.name
10         stringValue += ",dataSlices=" + str(self.dataSlices) + "
               }"
11         return stringValue
12
13     def __repr__(self):
14         return self.__str__()
15
16     def toJson(self):
17         dataSet = {}
18         dataSet["name"] = self.name
19         dataSet["dataSlices"] = []
20         for dataSlice in self.dataSlices:
21             dataSet["dataSlices"] += [ dataSlice.toJson() ]
22         return dataSet
23
24     def fromJson(self, jsonData):
25         self.name = ""
26         del self.dataSlices[:]
27         if "name" in jsonData.keys():
28             self.name = jsonData["name"]
29         if "dataSlices" in jsonData.keys():
30             for dataSliceDef in jsonData["dataSlices"]:
31                 dataSlice = DataSlice()
32                 dataSlice.fromJson(dataSliceDef)
33                 self.dataSlices += [ dataSlice ]
34
35 class DataSlice():
36     def __init__(self):
37         self.name = ""
38         self.dataTable = {}
39
40     def __str__(self):
41         stringValue = "{name=" + self.name
42         stringValue += ",dataTable=" + str(self.dataTable) + "}"
43         return stringValue
44
45     def __repr__(self):
46         return self.__str__()
47
```

```
48      def toJson(self):
49          dataSlice = {}
50          dataSlice["name"] = self.name
51          dataSlice["dataTable"] = self.dataTable.copy()
52          return dataSlice
53
54      def fromJson(self, jsonData):
55          self.name = ""
56          self.dataTable.clear()
57          if "name" in jsonData.keys():
58              self.name = jsonData["name"]
59          if "dataTable" in jsonData.keys():
60              self.dataTable = jsonData["dataTable"].copy()
61
62  def writeJson(fileName):
63      dataSet = DataSet()
64      dataSet.name = "Input data"
65      dataSet.dataSlices += [ DataSlice() ]
66      dataSet.dataSlices += [ DataSlice() ]
67      dataSet.dataSlices[0].name = "Training sample"
68      dataSet.dataSlices[0].dataTable["x-axis"] = [ 1.0, 2.0, 3.0,
            4.0, 5.0 ]
69      dataSet.dataSlices[0].dataTable["y-axis"] = [ 11.0, 16.3,
           12.1, 9.2, 4.2 ]
70      dataSet.dataSlices[1].name = "Analysis data"
71      dataSet.dataSlices[1].dataTable["x-axis"] = [ 1.0, 2.0, 3.0,
            4.0, 5.0 ]
72      dataSet.dataSlices[1].dataTable["y-axis"] = [ 13.0, 15.7,
           10.4, 10.1, 3.9 ]
73      jsonData = dataSet.toJson()
74
75      outputFile = open(fileName, "w", encoding="utf-8")
76      json.dump(jsonData, outputFile, ensure_ascii=False, indent
           =4)
77      outputFile.close()
78
79  def readJson(fileName):
80      inputFile = open(fileName, "r", encoding="utf-8")
81      jsonData = json.load(inputFile)
82      inputFile.close()
83
84      dataSet = DataSet()
85      dataSet.fromJson(jsonData)
86      print(dataSet)
87
88  def jsonIO():
89      fileName = "jsonData.json"
90      writeJson(fileName)
91      readJson(fileName)
92
93  if __name__ == "__main__":
94      jsonIO()
```

The Listing 20 contains the definition of two classes `DataSet` and `DataSlice`. The `DataSet` has a name and a dictionary of `DataSlice` objects. The key for the dictionary entries is

the name of the `DataSlice` object. The `DataSet` and `DataSlice` class contain functions to return their contents as a string and functions to read and write to JSON objects.

The example program calls `writeJson` to write a JSON file in the present working directory. Then `readJson` is called to load the information back into memory and print it to the standard output.

The `writeJson` function creates a `DataSet` object that contains two `DataSlice` objects. The (toJson) function in the `DataSet` class is then called to convert these data objects to a format that can be written to a JSON file. An output text file is opened and the JSON file is written with options that make it easier for a human to read. The output file is then closed.

The `readJson` function opens the existing JSON file and loads the data into memory. The file is then closed. A new `DataSet` object is created and the `fromJson` member function is called to fill it from the JSON data. Finally, the values in the `DataSet` and `DataSlice` objects are printed to the standard output.

## 13.4 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 Listing 21 demonstrates how to write and read an object from a pickle file.

Listing 21: PickleIO.py

```
1 import pickle
2
3 class DataObject():
4     def __init__(self):
5         self.__private = "DataString:12345"
6
7     def __str__(self):
8         return "{ __private=" + str(self.__private) + "}"
9
10 def writePickle(fileName):
11     dataObject = DataObject()
12     outputFile = open(fileName, "wb")
13     pickle.dump(dataObject, outputFile)
14     outputFile.close()
15
16 def readPickle(fileName):
17     inputFile = open(fileName, "rb")
18     dataObject = pickle.load(inputFile)
19     inputFile.close()
20
21     print("dataObject=" + str(dataObject))
22
23 def pickleIO():
24     fileName = "pickle.bin"
25     writePickle(fileName)
26     readPickle(fileName)
27
28 if __name__ == "__main__":
```

```
29      pickleIO()
```

The Listing 21 contains a single class definition, which has a constructor and a string function. The constructor assigns a data value to a private data member `__private`. The string method returns the value of this data member with its name.

The function `writePickle` is called to write the Pickle file. It creates a new `DataObject` object, opens a binary output file, dumps the pickled object into the output file and closes the output file.

The function `readPickle` is called to read the Pickle back into memory and print the value of the `__private` data member. The function opens the binary input file, loads the pickle creating the class, closes the input file and prints the value that is stored in `__private` data member.

## 14 Comments

In this document, comments are not used within the source code since the code is discussed within the document. However, one should normally add comments to Python code. Comments should explain the function of the code, rather than replicate the Python code itself. Python comments can be converted into documentation using the `pydoc` program [4]. Therefore, comments are useful beyond just when reading the code.

Python allows single line comments to be made with `#` and multiple line comments to be made with three `"` characters. An example of single and multiple line comments is given in Listing 22. The comments in this Listing are verbose, since the code is part of a program that was written for education purposes. Normally, comments can assume that the reader understands the Python programming language itself.

Listing 22: Python comments.

```
1  import random
2
3  """
4  A function to simulate rolling two dice.  The function returns
5  a random value between 2 and 12.
6  """
7  def rollTwoDice():
8      return random.randint(1,6) + random.randint(1,6)
9
10 """
11 A program to simulate the rolling of two dice.
12 """
13 def main():
14     # A list to contain the total value rolled.
15     # There are 12 elements, because the total value that can be
16     # rolled is 12.
17     counters=[0.]*12
18
19     # Roll two dice 100 times.
20     nRolls = 100
21     for i in range(nRolls):
22         totalValue = rollTwoDice()  # roll the dice
23
```

```
24          # Python list indices count from zero.  Therefore, have
25          # to remove one from the totalValue to put it into the
26          # right index in the list.
27          counters[totalValue-1] = counters[totalValue-1] + 1. #
                count this total value
28
29      # Total probability is always defined as 1.
30      # Therefore, have to divide by the total number of counted
31      # values.
32      for i in range(len(counters)):
33          counters[i] = counters[i] / float(nRolls)
34
35      # Now print out the probabilities for each of the combinations
36      print("The probabilities of rolling a total value using two
            dice:")
37      for i in range(len(counters)):
38          # Need to add one, since Python counts from zero.
39          print(" P("+str(i+1)+")="+str(counters[i]))
40      print("where P(n) is the probability of rolling a total of n
            on two dice.")
```

## 15 Conclusions

This document introduces some features of the Python programming language, using a series of example programs. The Python programming language provides developers with a large range of libraries and additional functionality that are not discussed within this document. However, this document provides a good starting point to explore and understand more of the Python programming language.

## References

[1] Official Python reference documentation. `https://docs.python.org/3/`. Accessed: 2020/08/19.

[2] W3C schools Python reference. `https://www.w3schools.com/python/`. Accessed: 2020/08/19.

[3] ANACONDA. `https://www.anaconda.com/products/individual`. Accessed: 2020/08/19.

[4] pydoc. `https://docs.python.org/3/library/pydoc.html`. Accessed: 2020/08/19.