# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to Python Programming</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Execution of Python Programs</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.2 Integrated Development Environment (IDE) for Python</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.3 Running Python with Jupyter Notebook</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Fundamental Concepts of Programming</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.1 The Concept of Abstraction</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Basic Elements of Python</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3.1 Objects and Variables</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3.2 Core Data Types in Python</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3.3 Operators and Expressions</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>3.4 Statements and Assignments</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Standard Input and Output</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>4.1 Comments</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>Summary for Module 1</td>
<td>29</td>
</tr>
</tbody>
</table>
1 Introduction to Python Programming

Overview of Python

Why Python? Python is a general-purpose programming language that can be used to literally build any kind of programs. It is considerably a simple programming language that is easy to learn and use. With the support of a rich collection of libraries (or packages), Python can be used for developing relatively sophisticated programs, ranging from data analytics and website development to gaming and robotic applications.

Python is not a new programming language and it has been around for over two decades. It was first conceptualised in the late 1980s by a Dutch programmer, Guido van Rossum. The language was named after Guido's personal favourite comedy series "Monty Python's Flying Circus".

Python is in fact a living language that constantly evolving since its first introduction. At the present, there are two major versions of Python: Python 2 and Python 3. Python 3 was released to mainly address various design decisions and inconsistency in Python 2 and its subsequent releases, Python 2.x. However, Python 3 is not backward compatible with Python 2.x; this means that programs written in Python 2.x generally could not run properly using the Python 3 implementations. The current version is Python 3.6, which is also the version that we use here. (Note that Python 3.5 will work just fine.)

High-level Programming Languages

Python is a high-level programming language. A high-level programming language (such as C, C++, Java, or Ruby) is generally not interpretable or directly executable by the Central Processing Unit (CPU), the hardware that is responsible for program execution. Thus, programs written in high-level programming languages have to be translated into a low-level programming language or the machine language in order to be executable by the CPU.

There are a number of advantages writing or implementing programs in a high-level programming language. Comparing with low level programming languages such as the machine language or the assembly language, high-level language programs are easier to implement and generally require shorter amount of time to write since high-level programming languages are very English like and hence more readable and comprehensible by programmers. Furthermore, programs written in a high-level programming language are known to be portable and platform independent. As such, the program just has to be written once and it can then be run on any types of computer platform (Windows, Macintosh, or Linux) without much modification effort.
Compiled and Interpreted Languages

High-level language programs have to be translated into the machine language for execution. This can be achieved in two ways: it is either by using a compiler or an interpreter.

With the use of the compiler, high-level language programs written in the form of source code are first compiled into the so-called object code of a specific program language. This object code is then executable at any time without further compilation or translation. Hence, this type of programs is compiled once and can then be run at any time through the executable object code. On the other hand, the interpreted-type of programs have to be translated every time they need to be executed. The interpreter of a specific programming language reads the instructions of the program code one at a time, and interprets the intended meaning of the instruction before executing it.

Is Python a compiled or interpreted programming language?
1.1 Execution of Python Programs

When you execute a Python program, the source code of your program (with the .py file extension) is compiled or translated into a format known as "byte code". The byte code is some form of low-level, platform-independent representation of the source code of your Python program. (Note that byte code the instructions are not the actual machine instructions.) The byte code of your Python programs are stored as .pyc files and are often found in a subdirectory named "__pycache__" which is located on the same directory as where the source code of your programs are stored. Once a program has been compiled into byte code, it can be loaded for execution at any time. During the execution, each of the byte code instructions will be interpreted and executed by the runtime execution engine known as the Python Virtual Machine (PVM).

In general, programs written in Python can be interpreted and executed by using the Python interpreter in two modes: interactive mode or script mode. Let's see how these two modes of execution work.

Interactive Mode

To run a Python program with the interactive mode, we need to start up the Python interpreter by running the command python at the prompt of a command-line terminal. The interpreter will start running in an interactive mode and it should appear similarly to the figure shown below (Figure 1.1), depending on the type of computer platform that the Python interpreter is running on.

![Figure 1.1 Python interpreter running in the interactive mode](image)

Interactive mode is usually used to test out individual Python statements or small segments of code. For instance, the figure below (Figure 1.2) demonstrates the execution of a simple Python statement that computes the addition of two integer numbers. Notice that the Python statement is typed at the interactive mode prompt represented by the symbol >>>. Once the statement is entered, the Python interpreter will interpret the statement and execute it. The output of the execution is presented on a separate line; as for this example, the value of 3 as the result of the addition of the numbers 1 and 2 is displayed.
1.1 Execution of Python Programs

Python 3.5.2 |Anaconda 2.5.0 (x86_64)| (default, Jul 2 2016, 17:52:12)
[ GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28) ] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> 1 + 2
3

Figure 1.2 An example of Python statement executed in the interactive mode

---

Script Mode

Alternatively, Python programs can be executed in the script mode. The source file (also known as the module file) of a Python program with the file extension of .py can be created using any form of text editors or one from a Python integrated development environment (IDE), which we will explain next. To execute a program, the source file (e.g. program.py) is passed on as an argument to the python command at the command-line prompt as follow:

```
python program.py
```

The Python interpreter will interpret and execute the contents of the source file. If there are any outputs, they will be presented on the terminal. Suppose that the program.py program contains the following lines of code:

```
a = 1
b = 2
print ("The addition of a and b is", a + b)
```

When the program.py program is executed, the output of the program "The addition of a and b is 3" is displayed on the terminal. Note that a and b used in program.py are referred as variables in programming, that can serve as a reference to a specific type of value, such as an integer value in this example. We will discuss more on variables in subsequent sections.

As a quick note, the interactive mode generally enables you to run or test certain sections of your program, while the script mode runs the entire program written in the Python script in one go.
1.2 Integrated Development Environment (IDE) for Python

In order to execute Python programs, you need to at least have the Python interpreter installed on your machine. Note that not all the computer systems would have Python preinstalled (e.g. Windows), you may have to perform the installation by yourself. Given that coding examples presented here are largely based on Python 3, install Version 3 if you have to. (The Python installation package can be downloaded from the Python official download site (https://www.python.org/downloads/).)

Python programs can be created using any form of editors and then executed by the Python interpreter. However, it is more manageable and convenient to develop Python programs under some form of integrated development environment (IDE) that comes with a set of utilities and libraries (or commonly known as packages in Python) that assist programmers to edit, build, and debug programs. For instance, the IDLE is the default of the Python IDE bundled with the Python installation package. IDLE provides the standard Python library.

Another more popular Python IDE known as Anaconda, provides many more useful packages for scientific computation and data analysis, such as NumPy, SciPy, Pandas and others, which you will be introduced throughout this course. Conveniently, Anaconda also installs an interactive web-based authoring application called Jupyter Notebook (formerly known as IPython Notebook) that enables programmers to create and execute Python programs interactively using a browser. (The Anaconda distribution package can be downloaded from the Anaconda official download site (https://www.continuum.io/downloads).)

For information about Anaconda and Jupyter Notebook visit the following sites:

- Anaconda Distribution  
  https://docs.continuum.io/anaconda/

- Jupyter/IPython Notebook Quick Start Guide  
1.3 Running Python with Jupyter Notebook

Once you have installed Anaconda on your machine, the Jupyter Notebook can be launched by running the command `jupyter notebook` at the prompt of a command-line terminal. Alternatively, the Jupyter Notebook can be launched from the Anaconda Launcher window (Figure 1.3) by clicking the “Launch” icon at the bottom of “Jupyter Notebook” and the dashboard will be shown (Figure 1.4).

![Anaconda Launcher window](image)

*Figure 1.3 The Anaconda Launcher window*
To start a new notebook document that would allow you to write Python code and execute in an interactive mode, click on the "New" button on the left of the dashboard and select "Python [default]" - the default kernel or the computational engine for executing Python code. You should now see a new Python notebook document named "Untitled" as depicted in Figure 1.5. (Note that the notebook documents can always be renamed.)

You may now start entering the Python code into this newly created notebook document. Using the previous examples that we have seen earlier, type in $1 + 2$ in the first line or the first cell (Figure 1.6). To execute this single line of Python code, click on the "run" button $\text{⌘}$. (Note that you should not get confused with the other button $\text{⌘}$ which is meant for "show presentation".)

You should now see the instruction was interpreted and the result of 3 is displayed before the next empty cell (Figure 1.7).
1.3 Running Python with Jupyter Notebook

Figure 1.6 Entering Python code in Jupyter Notebook

Figure 1.7 Executing Python code interactively in Jupyter Notebook
2 Fundamental Concepts of Programming

Programs and Algorithms

The art of programming is to design and implement a solution for solving a computational problem. An algorithm provides a general solution to a problem by specifying a sequence of step-by-step instructions with a flow of control indicating how each of the instructions is executed for solving the problem under consideration. The instructions of an algorithm are implemented or written, using a particular programming language, as a form of computer program that can be both understandable and executable by computers.

In a nutshell, a computer program consists of the following essential components:

- **Input**: data read in from an input device (such as a keyboard or an input file) that may be needed for solving a specific problem;
- **Output**: results from the computation that may be displayed on an output device (such as a terminal screen or an output file);
- **Operations**: a set of computational operations (such as arithmetic or logical) specified as computer instructions that are to be performed for solving the problem;
- **Control structures**: a set of control structures that determine when a set of instructions should be executed and whether a set of instructions should be repeated.
2.1 The Concept of Abstraction

When implementing a solution for a real-world problem, you may only be interested in modelling the essential properties of the solution rather than accommodating every single detail of the solution. At the very least, the resulting computational model should be able to solve the given problem by producing the expected outcome. This is the notion of abstraction which intends to ignore irrelevant details but focusing only on the important ones.

Take the Python program, `program.py`, that we have seen earlier as an example.

```python
a = 1
b = 2
print("The addition of a and b is", a + b)
```

This simple program is aimed at solving one of the fundamental arithmetic problems that sums two numbers.

To implement a solution for this arithmetic problem as a computational program, we need some form of representation for modeling the input for the program; and we also need to know how we could model the addition operation. To achieve this, the `program.py` was implemented with two representative symbols (referred as the variables) for denoting the two numbers respectively, which serve as the input data for the program. These two symbols have a specific set of properties associated with them. In this example, they are used for representing the integer type of data, and the type of operations that can be performed on integers are typically the arithmetic type of operations.

From an abstraction point of view, all a programmer needs to know is that the variables are representing integer values, and for that only arithmetic operations can be performed on this type of values. However, the programmer does not necessarily need to know how integers are modelled at the hardware or machine level as a binary representation; what the programmer should really care about is the operations or functions that are available in Python for manipulating integers and the ways to invoke these operations, such as using the `+` symbol (operator) to perform the computation. The former concept is typically referred as data abstraction; while the latter is referred as procedural or functional abstraction.

The concept of abstraction attempts to separate the logical aspect from the physical aspect of a computational problem and its solution as a realisation of abstraction. With the logical aspect, we meant how the elements of the solution are perceived by the users of the abstraction; where the physical aspect is about the actual implementation of those elements of the solution. In the subsequent sections, you will be introduced with some built-in data types in Python and the operations associated with each of these data types. You do not necessarily have to know the underlying implementation but simply to know how to use them to construct your own programs for solving computational problems.
3 Basic Elements of Python

Analogous to natural languages, each programming language is defined by its syntax and semantics, which can be varied from one language to another.

Syntax and Semantics

The syntax of a programming language is regarded as a set of rules that defines how program instructions can be constructed from various valid symbols and structures of that specific language. For instance, a Python program can only be executed by the Python interpreter if it is syntactically well formed by using the valid Python's syntax. If the program could not be executed, the interpreter will display error messages on the terminal indicating the existence of some form of syntax errors.

The semantics of a programming language, on the other hand, is referred to the meaning associated with a program or its individual instructions. Note that a syntactically well-formed program can always be executed without errors; however, it might not be doing the right things or achieving its intended tasks. In other words, if a legal program did run successfully but did not produce the expected result, this indicates that there might be a semantic error in the program. This type of errors is generally not recognised by the interpreter as it is usually due to the mistakes made by the programmer on the logic of the program. Thus, semantic errors are also known as logic errors.

We will look at some basic syntax of Python here in particular the core elements that are essential for constructing Python programs. Before we proceed, it would be useful to have a high-level overview of the composition of a Python program.

- A Python program contains one or more modules, i.e. source files (with the file extension of .py).
- Each module or source file contains one or more statements.
- Each statement contains one or more expressions.
- Each expression is composed of objects and operators (where objects are data values that operators manipulate on).

We will take a bottom-up approach here to introduce each of these Python components in turn.
3.1 Objects and Variables

Objects and Values

The core elements that Python programs are manipulating on are referred as objects. Objects are essentially pieces of memory in the computer that containing a certain type of data value or literal (in the programming context). Thus, each object is associated with a specific data type or object type (such as integer or string) that denotes how a program can manipulate it and what kind of operations that a program can perform on this type of data.

Recall that the simple Python program that we have seen earlier, program.py, which contains of the following lines of code:

```
a = 1
b = 2
print ("The addition of a and b is", a + b)
```

The literal value 1 is in fact a Python object itself of type integer (int). The sequence of characters "The addition of a and b is" is another object of type string (str). You can always find out the type of an object by using the built-in Python function called type. Refer to the code examples presented in Figure 1.8.

```
In [1]: type(1)
Out[1]: int

In [2]: type('The addition of a and b is')
Out[2]: str
```

Figure 1.8 Finding the type of an Python object
Variables and Names

In every programming language, variables are a means of associating a name to a value stored in the computer's memory. Hence, a variable in Python is just a name that references an object. In program.py, a and b are both variables that reference to an object of the same type (int) but with a different value. This is achieved through the assignment statement, such as a = 1, that binds the int object of value 1 on the right hand side of the assignment operator '=' to the variable name (a) on the left hand side of the symbol '='.

A same variable can be associated with a number of different objects that could be of a different literal value or even of a different type. The example below illustrates the variable a which was initially has the integer literal 1 assigned to it. It is then being assigned with another integer literal 100 in the second statement; and the third statement binds it with a string literal "Hi Python".

```
 a = 1
 a = 100
 a = "Hi Python"
```

Note that Python variables themselves do not have a data type but the associated objects do; and the type of the object is determined by the literal that it contains. Thus, in the first two statements above, the variable a is associated with objects of type int while in the third statement with an object of str.

Naming Rules and Conventions

When defining a variables in Python, the rule is that variable names can only contain lowercase letters (a-z), uppercase letters (A-Z), digits (0-9), and the underscore (_). They are case sensitive and may not begin with a digit. However, there is no restriction in length. Note that certain variable names are prohibited in Python; these are those that known as reserved words (or keywords). The following is a list of reserved words in Python:

```
and    as       assert    break    class    continue
def    del      elif       else      except    finally
for    from     global    if        import    in
is     lambda   nonlocal  not       or        pass
```
Note the following variable names (Figure 1.9) are illegal and the Python interpreter would flag these variable definitions as syntax errors.

![Figure 1.9 Illegal variable names in Python](image)

Variable names are not restricted to a single character or a single word. The variable `a` that we have been using as example although is legal; but it is not the best practice in the programming context. Variable names should be meaningful and usually self-explained with the kind of data that it represents. It would be more sensible to name variables as "first_number" and "second_number" rather than "a" and "b".

When using multiple words as variable names, there are two common naming conventions adopted in the Python community: use either a single underscore (_) as the delimiter between words (e.g. `first_number`) or the so-called camelCase style (e.g. `firstName`). It is always a good programming practice to maintain consistency throughout your programs with the one same style of naming convention.
3.2 Core Data Types in Python

Python supports a number of built-in primitive data types (or object types) that can generally be classified as atomic and collective. The atomic types are considered indivisible which represents only a single data value; an example of this is the integer type (int) that we have seen earlier. Collective types, on the other hand, are comprised of more than one data values; the string type (str) is an example of this type where a string may contain more than one character.

Numbers

Two fundamental representations for numbers are available in Python: integers (type int) that represent whole numbers and floating-point or real numbers (type float) with a fractional part. The common arithmetic operations, including addition, subtraction, multiplication, and division, are supported by Python for both integers and real numbers. We will discuss these operations in more details when we introduce operators and expressions in the next section.

Below are some valid literals for denoting integers and real numbers in Python. Both positive and negative numbers can be represented. The last real literal denotes $2.718 \times 10^3$.

```python
first_number = 12
second_number = -1234
pi = 3.1416
gamma = 0.577215
e_multiple_1000 = 2.718e3
```

Booleans

Logical values are represented by Python as the Boolean data type (type bool). There are only two possible values of a bool object: True or False.

Python supports the standard logical operations on Boolean values, such as AND, OR, and NOT. Boolean
type objects are often used as the results for comparison, such as checking for equality of and relation between data values (see Figure 1.11). More details on these logical and relational operations later.

![Image of evaluating two data objects on equality and relation of less than](image1)

Note that you should not get confused with the two operators or symbols: ‘=’ and ‘==’. The former is meant for assigning a value to a variable (as we have seen earlier); while the latter is meant for checking whether the two variables on both sides of the ‘==’ are equal to each other. As demonstrated by the example in Figure 1.11, ‘yes’ is not the same as ‘no’ given that they are assigned with a different Boolean value (True and False respectively).

### Strings

We have seen a couple of examples for the string data type (`str`). This is a collective data type as compared to `int`, `float` and `bool`. A String type object contains a collection of characters or letters. Literally, a character can be any keystroke including, a letter ('a'), a number ('12'), a punctuation ('!'), a space (' '), a tab ('	'), or even a newline ('
'). Note that a single character is indeed a string object itself. We could also have an empty string, which contain zero characters. In Python, strings are recognised as any number of characters within a pair of quotation marks, where both single ('') and double quotes (" ") are acceptable.

```python
last_name = "Smith"
first_name = 'John'
middle_name = "K"
message = "Welcome to Python Programming!"
```

Strings are useful for presenting the computational results produced by your programs to some form of output devices, typically the terminal screen. Recall the example that we have seen much earlier that computes the sum of two integer variables `a` and `b`, the results of the addition was presented on the terminal using another Python built-in method called `print()`, as depicted below (Figure 1.12).
Python provides a number of built-in methods for string manipulation. Briefly, methods are generally defined to perform certain operations on a specific type of objects. A method is a "self-contained" set of instructions (i.e. multiple lines of code) that defined to accomplish a specific task. A method may take in values as arguments to be processed and return some form of result. Once a method is defined, it can be then re-used in other programs. We will return to the concept of methods under the section of decomposition. For now, let's explore some of the pre-defined methods by Python for the String data type.

As for instance, `len()` can be used to find out the size of the string (i.e. the number of characters within the string). Another example, you may want to capitalize all the characters of a string. This can be achieved by invoking the built-in method `upper()`. Alternatively, `lower()` is for converting all the characters into lowercase. If you would like to find out the number of occurrences of a specific character in a string object, `count()` is the built-in method that enables this.
3.3 Operators and Expressions

Data objects and operators are combined to form expressions, which are evaluated by the Python interpreter during the execution, to a value of a specific data type or object type. The result type is largely dependent on the type of operators and operands (i.e. the data objects) that the operators perform on. The evaluated result can often be assigned to a variable object for further evaluation or manipulation. A simple expression has the following syntax:

```
result_variable = <operand> <operator> <operand>
```

### Arithmetic Operators

For integers and real numbers, these operators (+, -, *, /) are available to perform the basic arithmetic operations in Python. Each of these expressions are evaluated to a value of either type integer (int) or floating-point (float) depending on the two operands themselves. For most instances, if both of the operands are integer, the result will usually be an integer; if one of the two operands is a floating-point, the result will then be a floating-point.

![Figure 1.14 The basic arithmetic operators (+, -, *, /)](image)

For the first three expressions (in Figure 1.14) involving addition (+), subtraction (-), and multiplication (*), the evaluated results are as expected which are of type int. In the last expression, the result of the division is of type float although both the operands were integers. This is an exception in Python where the operator '/' is for real division. To perform floor division with the result rounded down to the nearest integer value, Python provides the floor division operator '//'. To obtain the remainder from a division, the modulo operator '%' is available. Another useful operator for manipulating numbers is the power operator '**'. 
In addition, arithmetic expressions can involve multiple operators as well as operands of mixed type. Take note of the examples below, the evaluated result of the expression in Line 2 is different from the one for the expression in Line 3.

\[
\begin{align*}
6 \times 2 / 1.5 \\
6 - 2 \times 3 \\
(6 - 2) \times 3
\end{align*}
\]

The arithmetic operators are associated with certain order of precedence; such that \(\ast\) and \(/\) are of higher precedence than \(+\) and \(-\). Hence, the expression in Line 2 is evaluated as first multiplying 2 with 3 and then subtracting the multiplication result from 6 yielding the result of 0. However, the level of precedence can be overridden by applying parentheses (or brackets) to form sub-expressions. This is demonstrated by the expression in Line 3, which produces a different result of value 12. (For operators having the same order of precedence, evaluation proceeds from left to right.)

### Relational Operators

Relational operators are used for determining the relationship that exists between two data values. The following relational operators are supported by Python to form logical expressions which are evaluated to a result of Boolean type, either True or False.

\[
\begin{align*}
a < b \\
a > b \\
a \leq b \\
a \geq b \\
a == b \\
a \neq b
\end{align*}
\]

Do not get confused with the double equals symbol (==) and of the single equal symbol (=); the former is the equality operator while the latter is indeed the assignment operator.

The operands that the relational operators can be applied on can be of any Python built-in data types. If the object values are of type int or float, the values are compared with respect to the relative numerical order; if the object values are of type str, the comparison is based on the lexicographical
3.3 Operators and Expressions

order (as demonstrated in Figure 1.16).

The operands do not have to be a single value; they can be in the form of arithmetic expressions (see Figure 1.17). Given that arithmetic operators have a higher order of precedence as compared to the relational operators; the arithmetic expressions at both sides of a relational operator are first evaluated and the comparison is then made on the resulting values.

Logical Operators

Logical operators or Boolean operators can be combined with relational operators to form more complex logical expressions, also known as compound expressions. Python supports the three basic logical operators: and, or, and not.

- `a and b` is evaluated to True if and only if both the operands `a` and `b` are True
- `a or b` is evaluated to True if either `a` or `b` is True or both `a` and `b` are True
- `not a` is used to invert the Boolean value of the operand `a` (i.e. if `a` is True, `not a` will turn `a` into False)

Note that the `not` operator is a unary operator since it can only apply on a single operand.

Logical operators generally have the lowest order of precedence. Hence, relational expressions are always
evaluated before applying the logical operators in a complex logical expression. Some examples of these are demonstrated below (Figure 1.18).

![Figure 1.18 Examples of compound logical expressions](image)

Logical expressions are useful in controlling the flow of execution for a program. They serve as the conditions that determine whether a particular segment of a program should be executed repeatedly for a number of times, or the program should branch out to another segment of the program instead of executing the program instructions in a sequential order. We will explore this programming concept in the next module under the control structures of programs.
3.4 Statements and Assignments

Fundamentally, a collection of statements put together forms a program. Instructions of a Python program are literally statements (or commands) that are interpretable and executable by the Python interpreter. We have already seen some examples of statements. Recall the following example that binds an object contains a specific type of value to a variable; it is a statement of assignment.

```python
message = "Welcome to Python Programming!"
```

Other examples of assignment could be that the result of an expression (be it simple or compound, arithmetic or logical) can be assigned to a variable for further evaluation or to be used in the later part of a program.

```python
temperature_fahrenheit = temperature_celsius * 9 / 5 + 32
boolean_result = counting_value > 0 and counting_value < 100
```

Apart from assignment, calling or invoking a method is another type of statement in Python. For instance, displaying some string on the terminal screen by invoking the Python built-in `print()`. Note that the `message` variable is passed on as an argument to the `print` method as the string to be displayed.

```python
message = "Welcome to Python Programming!"
print(message)
```

Generally, each Python statement spans a single line within a Python program (or in the source file). However, we could split a single statement across multiple lines for readability reason. This can be achieved by appending a backslash (`\`) at the end of the line and continue on in the next line.

```python
boolean_result = counting_value > 0 \ 
                and \ 
                counting_value < 100
```

Python programs may consist of certain types of programming construct that span across multiple lines, forming a block of statements. These constructs are the control structures that determine the flow of execution for a program. As mentioned earlier, we are deferring the detailed discussion of this programming concept to the next module.

However, let's take a quick look at an example of statement blocks in Python, in which a block of statements is repeatedly executed based on a condition that governs the number of repetitions.
counter = 5
while counter >= 0:
    print(counter)
    counter -= 1

The example above repeatedly prints out the value of the counter variable, where the value of counter is decremented by 1 for each repetition (as shown in Figure 1.19).

The while keyword is the control structure that allows a block of statements to be repeatedly executed; and the logical expression counter >= 0 is the governing condition. You may not be able to understand this code structure completely at this point; all you need to know is that the last two statements will keep repeating so long as the 'condition' after the while keyword is evaluated to True. (Note that the last statement counter -= 1 is essentially equivalent to the statement counter = counter + 1; the operator -= is known as the compound assignment operator.)

Figure 1.19 An example of statement block

A key point to note is that the symbol ':' is syntactically important in Python as it denotes the beginning of a statement block, i.e. the subsequent statements are considered as part of the block. The statement block is concluded with a new blank line. Hence, indentation is semantically meaningful in Python programs. The indentation can be a fixed number of spaces '; however, it is usually auto-indented with a 'tab' keystroke while editing within an 'IDE.
4 Standard Input and Output

Input and output are two of the essential components for a computer program; while input is the data that is needed for solving a specific computational problem, output is the means to present the computational result.

Recall again the simple program (program.py) that we have been referring, the addition of two numbers represented by the variables a and b.

```
a = 1
b = 2
print("The addition of a and b is", a + b)
```

**Standard Input**

Instead of limiting the program for summing two 'hard-coded' values of 1 and 2, we could make the program more flexible and useful for performing addition of any two values. These values can be obtained through the standard input using the keyboard. The program should be implemented as below.

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
print("The addition of a and b is ", a + b)
```

input() is the Python built-in method for obtaining data externally. Note that the input value is returned as an object of type str. In order to perform an arithmetic operation (addition in this case), the input has to be first converted to the type int by using the conversion method int() with the str input data passed onto it as an argument. This is called type conversion or type casting in programming.

**Standard Output**

We have indeed seen a number of examples for the standard output in Python. Basically, the print() function is used for displaying information as a form of string on the terminal screen. By default, the print() function appends a newline character '\n' at the end of each string to be displayed. Hence, each print statement will display the output on a separate line (as demonstrated in Figure 1.20).
Note that the only type of data that can be displayed is of type string (str). However, when you attempt to output data of other built-in types in Python (such as int or float), the print() function will convert the data values to strings implicitly. As you have seen in the example presented above for the standard input, the print statement will result in the following output:

However, this automated type conversion is only possible when multiple output arguments supplied to the print() function are separated by a comma ','. If you attempt to use the '+' operator to join or concatenate the output arguments as a string (as in Figure 1.22), an explicit type casting is required with the use of the str() function.

You may notice that you have to insert a space ' ' at the end of the first string argument to space out the string value of the variable result. Unlike for the case where commas are used for combining multiple arguments, spaces are included in between by default.

In some situations, you may want to present the output in a certain format. This is particular useful when displaying floating-point numbers with a specific number of decimal points. The '%’ operator is overloaded in Python for this purpose, see the example below (Figure 1.23). This is known as the printf-style
formatting commonly used in other programming languages such as C and Java. (For more details, refer to Python documentation (https://docs.python.org/3/library/stdtypes.html#old-string-formatting).)

```
In [1]: pi = 3.14159
   print ('The value of pi is %5.4f' % pi)
   The value of pi is 3.1416
```

*Figure 1.23 An example of formatted output with the printf style*
4.1 Comments

To enhance the readability of your program, you are encouraged to include some form of documentation or comments throughout your program. This is particularly useful when you have a complex segment of code that requires some explanation. Comments in Python are denoted by the symbol `#`. Any part of your code following this symbol is ignored by the Python interpreter, that means it will not be interpreted or executed.

The documented version of the running example `program.py` would look like this:

```python
a = 1   # initialise variable a with value 1
b = 2   # initialise variable b with value 2
# print the result of addition
print ("The addition of a and b is", a + b)
```

A key point that you should keep in mind when documenting your code: comments should really be used for explaining the meaning or functionality of the code rather than just providing a mere literal interpretation of the code.
5 Summary for Module 1

By now, you should be able to recognise various fundamental programming constructs of Python, such as the built-in data types, the basic arithmetic and logical operations, as well as the standard input and output. With these programming constructs, you should be able to implement simple programs in Python for solving some basic computational problems.

**Prescribed Reading**

- How to Think Like a Computer Scientist: Learning with Python 3 *(HTML Edition)*: Chapters 1
  (http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html) and 2
  (http://openbookproject.net/thinkcs/python/english3e/variables_expressions_statements.html)
- How to Think Like a Computer Scientist *(Interactive Edition)*: Chapters 1
  (https://interactivepython.org/runestone/static/thinkcspy/GeneralIntro/toctree.html) and 2
  (https://interactivepython.org/runestone/static/thinkcspy/SimplePythonData/toctree.html)
- Problem Solving with Algorithms and Data Structures *(Interactive Edition)*: Chapter 1
  (https://interactivepython.org/runestone/static/thinkcspy/GeneralIntro/toctree.html)

**Additional Reading**

- Introducing Python by Bill Lubanovic *(O'Reilly, 2014)*: Chapters 1
- Learning Python by Mark Lutz *(O'Reilly, 2013)*: Chapters 5

**Useful Resources**

- [Python 3 Documentation on Built-in Types](https://docs.python.org/3/library/stdtypes.html)
- [Python 3 Documentation on Input and Output](https://docs.python.org/3/tutorial/inputoutput.html)
- [Python's Official Download Site](https://www.python.org/downloads/)
- [Anaconda's Official Download Site](https://www.continuum.io/downloads)
- [Python 3 Tutorial (Python Online Course)](http://python-course.eu/python3_course.php)