A Short Introduction to Computer Programming Using Python

Carsten Fuhs and David Weston
(based on earlier documents by
Sergio Gutierrez-Santos, Keith Mannock, and Roger Mitton)
Birkbeck, University of London

v1.4

This document forms part of the pre-course reading for several MSc courses at Birkbeck.
(Last revision: December 4, 2019.)

Important note: If anything is unclear, or you have any sort of problem (e.g., downloading or installing Python on your computer), please send an email to Carsten Fuhs (*carsten@dcs.bbk.ac.uk*).

1 Introduction

This reading material serves as a preparation for an aptitude test for several MSc programmes at Birkbeck, University of London. This test is an online test that one can take via a web browser, and it only assumes content presented in this document. The questions on the test are similar in style to those in Section 5.

1.1 Recommended further reading

This booklet is designed to be self-contained. However, we will point you to several external resources which we recommend if you are interested in further explanations, additional background, and (also interactive) exercises.

We recommend two electronic textbooks in particular: "Python for Everybody – Exploring Data Using Python 3" by Charles R. Severance [1] and "Think Python – How to Think like a Computer Scientist" by Allen Downey [2]. These textbooks have been made available officially for free download. There are also interactive editions that can be used online, which will give you immediate feedback.

1.2 Motivation

Computing increasingly permeates everyday life, from the apps on our phones to the social networks that connect us. At the same time, computing is a general tool that we can use for solving real-world problems. For you to get a computer to do something, you have to tell it what you want it to do. You give it a series of instructions in the form of a *program*. The computer then helps us by carrying out these instructions very fast, much faster and more reliably than a human. In this sense, a program is a description of how to carry out a process automatically.

Often programs that others have written will do the job. However, we don't want to be limited to the programs that other people have created. We want to tailor our program to the needs of the problem that we have at hand. Writing programs is about expressing our ideas for solving a problem clearly and unambiguously. This is a skill increasingly sought also outside of the software industry. The programs that you can write may then help solve problems in areas such as business, data science, engineering, and medicine.

As a pointer to further reading for this part, you can find an extended description of the purpose and use of writing computer programs in [1, Chapter 1] and [2, Chapter 1].

1.3 Programming language

You write a program in a *programming language*. Many programming languages have been devised over the decades. The language we will use in this document is named *Python*. More precisely, we are using version 3 of the Python language. Python hits a sweet spot: it makes writing simple programs easy and is robust enough for a wide uptake in a variety of industries.

Programming languages vary a lot, but an instruction in a typical programming language might contain some English words (such as `while` or `return`), perhaps a mathematical expression (such as `x + 2`) and some punctuation marks used in a special way.

Programs can vary in length from a few lines to thousands of lines. Here is a complete, short program written in Python:

Example

```
print("Given a series of words, each on a separate line,")
print("this program finds the length of the longest word.")
print("Please enter several sentences, one per line.")
print("Finish with a blank line.")
maxi = 0
word = "."
while len(word) > 0:
    word = input()
    if len(word) > maxi:
        maxi = len(word)

if maxi == 0:
    print("There were no words.")
else:
    print("The longest sentence was " + str(maxi) + " characters long.")
```

When you write a program, you have to be very careful to keep to the syntax of the programming language, i.e., to the grammar rules of the language. For example, the above Python program would behave incorrectly if in the 12^{th} line we wrote

```
if maxi = 0:
```

instead of

```
if maxi == 0:
```

or if in the 9^{th} line we omitted the colon in

```
if len(word) > maxi:
```

The computer would refuse to accept the program if we added dots or colons in strange places or if any of the parentheses (or colons) were missing, or if we wrote `length` instead of `len` or even `Len` instead of `len`.

1.4 Input and output

Almost all programs need to communicate with the outside world to be useful. In particular, they need to read input and to remember it. They also need to respond with an answer.

To get the computer to take some input as text and to store it in its memory, we write in Python:

```
word = input()
```

`input()` is a phrase which has a special meaning to the computer. The combination of these instructions means "Take some input which is a sequence of characters and put it into the computer's memory."

`word` by contrast, is a word that I (the programmer) have chosen. I could have used `characters` or `thingy` or `breakfast` or just `s` or almost any word I wanted. (There are some restrictions which I will deal with in the next section.)

Computers can take in all sorts of things as input — numbers, letters, words, records and so on — but, to begin with, we will write programs that generally handle text as sequences (or *strings*) of characters (like "I", "hi", "My mother has 3 cats", or "This is AWESOME!"). We'll also assume that the computer is taking its input from the keyboard, i.e., when the program is executed, you key in one or more words at the keyboard and these are the characters that the computer puts into its memory.

You can imagine the memory of the computer as consisting of lots of little boxes. Programmers can reserve some of these boxes for use by their programs and they refer to these boxes by giving them names.

```
word = input()
```

means "Take a string of characters input using the keyboard and terminated when the user presses RETURN, and then put this string into the box called `word`." When the program runs and this instruction gets executed, the computer will take the words which you type at the keyboard (whatever you like) and will put them into the box which has the name `word`.

Each of these boxes in the computer's memory can hold only one string at a time. If a box is holding, say, "hello", and you put "Good bye!" into it, the "Good bye!" replaces the "hello". In computer parlance these boxes are called *variables* because the content inside the box can vary; you can put an "I" in it to start with and later change it to a "you" and change it again to "That was a hurricane" and so on as often as you want. Your program can have as many variables as you want.

In Python (and many other programming languages), you have to make sure that you have put something into a variable before you read from it, e.g., to print its contents. If you try to take something out of a box into which you have not yet put anything, it is not clear what that "something" should be. This is why Python will complain if you write a program that reads from a variable into which nothing has yet been put.

If you want the computer to display (on the screen) the contents of one of its boxes, you use `print(thingy)` where instead of `thingy` you write the name of the box. For example, we can print the contents of `word` by writing:

```
print(word)
```

If the contents of the box called `word` happened to be "Karl", then when the computer came to execute the instruction `print(word)` the word "Karl" would appear on the screen.

So we can now write a program in Python (not a very exciting program, but it's a start):

Example

```
word = input()
print(word)
```

This program takes some text as input from the keyboard and displays it back on the screen.

In many programming languages it is customary to lay out programs like this or in our first example with each instruction on a line of its own. In these languages, the indentation, if there is any, is just for the benefit of human readers, not for the computer which ignores any indentations.

Python goes a step further and uses indentation as a part of the language to structure the programs. We will see more about this a bit later.

1.5 Running a program

You can learn the rudiments of Python from these notes just by doing the exercises with pencil and paper. However, it is a good idea to test your programs using a computer after you have written them so you can validate that they do what you intended. If you have a computer and are wondering how you run your programs, you will need to know the following, and, even if you don't have a computer, it will help if you have some idea of how it's done.

First of all you type your program into the computer using a text editor. Then, to run your program, you use a piece of software called an *interpreter*. The interpreter first checks whether your program is acceptable according to the syntax of Python and then runs it. If it isn't acceptable according to the syntax of Python, the interpreter issues one or more error messages telling you what it objects to in your program and where the problem lies. You try to see what the problem is, correct it and try again. You keep doing this until the interpreter runs your program without further objections.

To make the computer run your program using the interpreter, you need to issue a command. You can issue commands[1] from the command prompt in Windows (you can find the command prompt under *Start* → *Accessories*), or from the terminal in Linux and Mac OS/X.

If you are lucky, your program does what you expect it to do first time. Often it doesn't. You look at what your program is doing, look again at your program and try to see why it is not doing what you intended. You correct the program and run it again. You might have to do this many times before the program behaves in the way you wanted. This is normal and nothing to worry about at this stage.

As I said earlier, you can study this introduction without running your programs on a computer. However, it's possible that you have a PC with a Python interpreter and

[1] In a modern operating system, you can click on an icon to execute a program. However, this only makes sense for graphical applications and not for the simple programs that you will write at first.

will try to run some of the programs given in these notes. If you have a PC but you don't have a Python interpreter, I attach a few notes telling you how you can obtain one (see Section 7).

If you do not wish to install Python on your machine, you can use the online Python editor and interpreter at

`https://repl.it/languages/python3`

Have a look at the intro video at

`https://replit.github.io/media/quick-start/simple-repl.mp4`

and the quick-start guide

`https://repl.it/site/docs/misc/quick-start`

1.6 Outputting words and ends of lines

Let's suppose that you manage to get your program to run. Your running of the above program would produce something like this on the screen if you typed in the word Tom followed by RETURN:

```
Tom
Tom
```

The first line is the result of you keying in the word Tom. The system "echoes" the keystrokes to the screen, in the usual way. When you hit RETURN, the computer executes the `word = input()` instruction, i.e., it reads the word or words that have been input. Then it executes the `print` instruction and the word or words that were input appear on the screen again.

We can also get the computer to display additional words by putting them in quotes[2] in the parentheses after the `print`, for example:

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

We can use this to make the above program more user-friendly:

Example

```
print("Please key in a word:")
word = input()
print("The word was:")
print(word)
```

An execution, or "run", of this program might appear on the screen thus:

[2] In Python, we can choose between " and ', but we cannot use both together.

```
Please key in a word:
Tom
The word was:
Tom
```

Here each time the computer executed the `print` instruction, it also went into a new line. If we use `print(word, end = "")` instead of `print(word)` then no extra line is printed after the value.[3] So we can improve the formatting of the output of our program on the screen:

Example

```
print("Please key in a word: ", end = "")
word = input()
print("The word was: ", end = "")
print(word)
```

Now a "run" of this program might appear on the screen thus:

```
Please key in a word: Tom
The word was: Tom
```

Note the spaces in lines 1 and 3 of the program after `word:` and `was:`. This is so that what appears on the screen is `word: Tom` and `was: Tom` rather than `word:Tom` and `was:Tom`.

It's possible to output more than one item with a single `print` instruction. For example, we could combine the last two lines of the above program into one:

```
print("The word was: " + word)
```

and the output would be exactly the same. The symbol "`+`" does not represent addition in this context, it represents concatenation, i.e., writing one string after the other. We will look at addition of numbers in the next section.

Let's suppose that we now added three lines to the end of our program, thus:

Example

```
print("Please key in a word: ", end = "")
word = input()
print("The word was: " + word)
print("Now please key in another: ", end = "")
word = input()
print("And this one was: " + word)
```

After running this program, the screen would look something like this:

```
Please key in a word: Tom
The word was: Tom
Now please key in another: Jane
And this one was: Jane
```

[3]The reason is that with `end = ""` we are telling Python that after printing the contents of `word`, we want to print nothing at the end instead of going to a new line.

Exercise A

For those of you who prefer a "learning by doing" approach, the University of Waterloo provides an interactive tutorial to programming in Python 3 with a similar structure to this booklet:

`https://cscircles.cemc.uwaterloo.ca/`

This tutorial gives you further exercises and checks whether the output obtained from your solutions is correct. We will occasionally refer to exercises in this online tutorial. You are welcome to create an account if you wish to record your progress. However, as this booklet is designed for self-study, we will not provide any assistance for this system.

As your first exercise, read through Section "0: Hello" at the above link and attempt the exercises in your browser as you go along. The web page will provide you with direct feedback.

Exercise B

Now pause and see if you can write:

1. a Python instruction which would output a blank line.
2. an instruction which would output

   ```
   Hickory, Dickory, Dock
   ```

3. a program which reads in two words, one after the other, and then displays them in reverse order. For example, if the input was

   ```
   First
   Second
   ```

   the output should be

   ```
   Second
   First
   ```

1.7 Assignment and initialisation

There is another way to get a string into a box apart from using `input()`. We can write, for instance:

```
word = "Some text"
```

This has the effect of putting the string "Some text" into the `word` box. Whatever content was in `word` before is obliterated; the new text replaces the old one.

In programming, this is called *assignment*. We say that the value "Some text" is assigned to the variable `word`, or that the variable `word` takes the value "Some text". The "`=`" symbol is the assignment operator in Python. We are not testing whether `word` has the value "Some text" or not, nor are we stating that `word` has the value "Some text"; we are *giving* the value "Some text" to `word`.

An assignment instruction such as this:

```
word = word + " and some more"
```

looks a little strange at first but makes perfectly good sense. Let's suppose the current value of `word` (the contents of the box) is "Some text". The instruction says, "Take the value of `word` ("Some text"), add " and some more" to it (obtaining "Some text and some more") and put the result into `word`". So the effect is to put "Some text and some more" into `word` in place of the earlier "Some text".

The `=` operator is also used to "initialise" variables. In Python, a variable is *defined* by the first assignment to that variable, the first time you put something into it. This first assignment is often also called *initialisation*. You can read from a variable only after it has been initialised.

Finally a word about terminology. I have used the word "instruction" to refer to lines such as `print(word)` and `word = "Some text"`. It seems a natural word to use since we are giving the computer instructions. But the correct word is actually "statement". `print(word)` is an output statement, and `word = "Hello"` is an assignment statement.

Exercise C

Now see if you can write a program in Python that takes two words from the keyboard and outputs one after the other on the same line. E.g., if you keyed in "Humpty" and "Dumpty" it would reply with "Humpty Dumpty" (note the space in between). A run of the program should look like this:

```
Please key in a word: Humpty
And now key in another: Dumpty
You have typed: Humpty Dumpty
```

2 Variables, identifiers and expressions

In this section, we will look at the fundamentals of how Python stores and manipulates information. To this end, we will discuss integer numbers, arithmetic expressions, identifiers, comments, and strings.

2.1 Integer numbers

As well as strings, we can also have integer numbers (often just *integers*) as values: whole numbers. So variables can contain not only strings, but also any integer value like 0, -1, -200, or 1966.[4]

We can assign an integer value, say, `0`, to a variable, say, `i`:

```
i = 0
```

or, if we had two variables `i` and `j`, we could assign:

```
i = j
```

We can even say that `i` takes the result of adding two numbers together:

```
i = 2 + 2
```

which results in `i` having the value 4.

Integers and strings

You have probably noticed that, when dealing with integer values the symbol "`+`" represents addition of numbers, while – as we saw in the last section – when dealing with strings the same symbol represents concatenation. Therefore, the statement

```
word = "My name is " + "Inigo Montoya"
```

results in `word` having the value "My name is Inigo Montoya", while the statement

```
n = 10 + 7
```

results in `n` having the value 17 (not 107). What happens if we mix integer values and string values when using "`+`", say in the following statement?

```
answer = "Answer: " + 42
```

In that case, Python will give us an error message similar to the following one:

[4] In contrast to many other programming languages, Python does not have a limit to how large an integer value can become.

```
TypeError: cannot concatenate 'str' and 'int' objects
```

This means that Python does not know how to connect ("concatenate") the string "Answer: " (Python uses the name "str" for the *type* of strings) and the integer 42 ("int"). To clarify that we want to use "+" to concatenate two strings (and not to use "+" to add two values), we can convert the "int" value 42 to the "str" value "42" with the command `str()`:

```
answer = "Answer: " + str(42)
```

Similarly, the small program

```
word = "My name is " + "Inigo Montoya"
n = 10 + 7
text = word + " and I am " + str(n)
print(text)
```

will print on the screen "My name is Inigo Montoya and I am 17". It is important to know that the same symbol can be used for different things, but we will come to this later again; now let's go back to writing programs with a bit of maths in them using integers.

2.2 Reading integers from the keyboard

In the last section, we saw how we can read a string of characters from the keyboard, using `input()`. We can use the same command to read a number... but the computer will not know it is a number, it will think it is a string of characters. If we want to tell the computer that a sequence of characters *is* a number, we need to convert it. This is similar to the way we needed to tell the computer that we wanted to treat a number as a character string in the previous example. We can do this easily by *parsing* it using the command `int()`:

Example

```
print("Please introduce a number: ", end = "")
word = input()
n = int(word)
print("The number was " + word)
next = n + 1
print("The next number is " + str(next))
```

When we parse a string that contains an integer we obtain an integer with the correct value. If we try to parse a string that is not an integer (for example, the word "Tom") the program will terminate with an error message on the screen. If we do not parse the string and use it as if it was an integer, the results will not be what you'd expect. This is a common source of errors in programs. You can check for yourself what happens if you do not parse the string in the former example, e.g., what happens with this program:

Example

```
print("Please introduce a number: ", end = "")
word = input()
n = word
print("The number was " + word)
next = n + 1
print("The next number is " + str(next))
```

Now, assuming that you read your integers and always remember to parse them, what maths can you do with them?

2.3 Operator precedence

Python uses the following arithmetic operators (amongst others):

`+`	addition
`-`	subtraction
`*`	multiplication
`//`	division
`%`	modulo

The last one is perhaps unfamiliar. The result of `x % y` ("x mod y") is the remainder that you get after dividing the integer x by the integer y. For example, `13 % 5` is `3`; `18 % 4` is `2`; `21 % 7` is `0`, and `4 % 6 is 4` (6 into 4 won't go, remainder 4). `num = 20 % 7` would assign the value 6 to `num`.

How does the computer evaluate an expression containing more than one operator? For example, given `2 + 3 * 4`, does it do the addition first, thus getting `5 * 4`, which comes to 20, or does it do the multiplication first, thus getting `2 + 12`, which comes to 14? Python, in common with other programming languages and with mathematical convention in general, gives precedence to `*`, `//` and `%` over `+` and `-`. This means that, in the example, it does the multiplication first and gets 14.

If the arithmetic operators are at the same level of precedence, it takes them left to right. `10 - 5 - 2` comes to 3, not 7. You can always override the order of precedence by putting brackets into the expression; `(2 + 3) * 4` comes to 20, and `10 - (5 - 2)` comes to 7.

Some words of warning are needed about division. First, remember that integer values are whole numbers. So the result of a division with `//` is only the integer part without the decimal part. Try to understand what the following program does:

Example

```
num = 5
print(str(num))
num = num + 2
print(str(num))
num = num // 3 * 6
print(str(num))
print(str(7 + 15 % 4))
num = 24 // 3 // 4
print(str(num))
num = 24 // (num // 4)
print(str(num))
```

A computer would get into difficulty if it tried to divide by zero (it is not clear what the result should be). Consequently, the system makes sure that it never does. If a program tries to get the computer to divide by zero, the program is unceremoniously terminated, usually with an error message on the screen.

Exercise A

Write down the output of the above program without executing it.

Now execute it and check if you got the right values. Did you get them all right? If you got some wrong, why was it?

Exercise B

Optional: read through the following section and try to solve its exercises:

`https://cscircles.cemc.uwaterloo.ca/1-variables/`

2.4 Identifiers and comments

I said earlier that you could use more or less any names for your variables. I now need to qualify that.

The names that the programmer invents are called *identifiers*. The rules for forming identifiers are that the first character can be a letter (upper or lower case, usually the latter) and subsequent characters can be letters or digits or underscores. (Actually the first character can be an underscore, but identifiers beginning with an underscore are often used by system programs and are best avoided.) Other characters are not allowed. Python is case-sensitive, so `Num`, for example, is a different identifier from `num`.

The only other restriction is that you cannot use any of the language's keywords as an identifier. You couldn't use `if` as the name of a variable, for example. There are many keywords but most of them are words that you are unlikely to choose. Ones that you might accidentally hit upon are `assert`, `break`, `class`, `continue`, `def`, `del`,

`except`, `finally`, `global`, `import`, `pass`, `raise`, `return`, `try` and `yield`. You should also avoid using words which, though not technically keywords, have special significance in the language, such as `int` and `print`.

Programmers often use very short names for variables, such as `i`, `n`, or `x` for integers. There is no harm in this if the variable is used to do an obvious job, such as counting the number of times the program goes round a loop, and its purpose is immediately clear from the context. If, however, its function is not so obvious, **it should be given a name that gives a clue as to the role it plays in the program**. If a variable is holding the total of a series of integers and another is holding the largest of a series of integers, for example, then call them `total` and `maxi` rather than `x` and `y`.

The aim in programming is to write programs that are "self-documenting", meaning that a (human) reader can understand them without having to read any supplementary documentation. A good choice of identifiers helps to make a program self-documenting.

Comments provide another way to help the human reader to understand a program. Anything on a line after "`#`" is ignored by the interpreter, so you can use this to annotate your program. You might summarise what a chunk of program is doing:

```
# sorts numbers into ascending order
```

or explain the purpose of an obscure bit:

```
x = x * 100 // y   # x as percent of y
```

Comments should be few and helpful. Do not clutter your programs with statements of the obvious such as:

```
num = num + 1   # add 1 to num
```

Judicious use of comments can add greatly to a program's readability, but they are second-best to self-documentation. Note that comments are ignored by the computer: their weakness is that it is all too easy for a programmer to modify the program but forget to make any corresponding modification to the comments, so the comments no longer quite go with the program. At worst, the comments can become so out-of-date as to be positively misleading, as illustrated in this case:

```
num = num + 1  # decrease num
```

What is wrong here? Is the comment out of date? Is there a bug in the code and the plus should be a minus? Remember, use comments sparingly and make them matter. A good rule of thumb is that comments should explain "why" and not "how": the code already says *how* things are done!

Exercise C

Say for each of the following whether it is a valid identifier in Python and, if not, why not:

```
BBC, Python, y2k, Y2K, old, new, 3GL, a.out,
first-choice, 2nd_choice, third_choice
```

2.5 A bit more on strings

We have done already many things with strings: we have created them, printed them on the screen, even concatenated several of them together to get a longer value. But there are many other useful things we can do with strings.

For example, if you want to know how long a string is, you can find out using the *function* `len()`.[5] You could say, for example:

```
print("Please enter some text: ", end = "")
word = input()
length = len(word)
print("The string " + word + " has " + str(length) + " characters.")
```

You can obtain a substring (or a "slice") of a string as in the following example: If the variable `s` has the string value "`Silverdale`", then `s[0:6]` will give you the first six letters, i.e., the string "`Silver`". The first number in square brackets after the string says where you want the substring to begin, and the second number says where you want it to end (to be precise, the second number is the position of the first character that we *don't* want in the substring). *Note that the initial character of the string is treated as being at position 0, not position 1.* Similarly, `s[2:6]` will give you the string `lver`.

If you leave out the second number, you get the rest of the string. For example, `s[6:]` would give you "`dale`", i.e., the tail end of the string beginning at character 6 ("d" is character 6, not 7, because the "S" character is character 0).

You can output substrings with `print` or assign them to other strings or combine them with the "`+`" operator. For example:

Example

```
s = "software services"
s = s[0:4] + s[8:9] + s[13:]
print(s)
```

will output "soft ices".

Exercise D

Optional: read through the following section and try to solve its exercises:

`https://cscircles.cemc.uwaterloo.ca/7a-strings/`

Exercise E

Say what the output of the following program fragment would be:

[5] A function in Python is a named sequence of instructions. We *call* the function using its name in order to execute its instructions. We can call a function without knowing what instructions it consists of, so long as we know what it does.

Example

```
s = "artificial reality"
print(s[11:15] + " " + s[0:3])
length = len(s[11:])
print(str(length))
```

Then run the program and see if you were right.

3 Conditionals (`if` statements)

To write anything more than very straightforward programs we need some way of getting the computer to make choices. We do this in Python with the keyword `if`. We can write, for example:

Example

```
if num == 180:
    print("One hundred and eighty!")
```

When the computer executes this, it first sees whether the variable `num` currently has the value 180 or not. If it does, the computer displays its message; if it doesn't, the computer ignores the `print` line and goes on to the next line.

Note that the conditional expression (`num == 180`) has to be followed by a colon (":"), and that we must put some additional space in front of the code that we want to execute if the conditional expression is true. This additional space is also called *indent*, and it tells Python that this code is "inside" the `if` statement. The usual number of spaces to add to the indent is 4.

Note also that, to test whether `num` has the value 180 or not, we have to write `if num == 180:` and not `if num = 180:`. We have to remember to hit the "=" key twice. This can be a serious nuisance in Python, especially for beginners. It comes about because the language uses the "=" operator for a different purpose, namely assignment. `num = 180` does not mean "`num` is equal to 180", it means "Give the value 180 to `num`". You may feel that it is obvious that assignment is not what is intended in `if num = 180:`, but unfortunately that is how the computer will interpret it – and will complain. You have been warned.

The following program takes two numbers and displays a message if they happen to be the same:

Example

```
print("Please key in a number: ", end = "")
s = input()
num1 = int(s)
print("And another: ", end = "")
s = input()
num2 = int(s)
if num1 == num2:
    print("They are the same.")
```

3.1 Conditional expressions

Conditional expressions – the kind that follow an `if` – can be formed using the following operators:

`==`	is equal to
`!=`	is not equal to
`>`	is greater than
`<`	is less than
`>=`	is greater than or equal to
`<=`	is less than or equal to

When these operators are used with integers, their meaning is obvious, but they can also be used with strings. Here their meaning corresponds to something like alphabetical order (the order used for entries in a printed dictionary). For instance, `if s < t:`, where `s` and `t` are strings, means "If `s` comes before `t` in alphabetical order". So it would be true if `s` had the value "`Birkbeck`" and `t` had the value "`College`". All the upper-case letters come before the lower-case, so `s < t` would still be true if `s` had the value "`Zebra`" and `t` had the value "`antelope`" (upper-case "Z" comes before lower-case "a").

But what about strings that contain non-alphabetic characters? Would `s` come before `t` if `s` had the value "`#+*`" and `t` had the value "`$&!`"? To find the answer we have to consult the *UNICODE table* – the Universal Character Set. UNICODE defines a particular ordering of all the characters on the keyboard. (There are other orderings in use, notably EBCDIC which is used on IBM mainframe computers, and ASCII, which was adopted by PCs and became the de facto standard for English-based languages.) The UNICODE table tells us that the character "#" comes before the character "$", for instance. The latest version of Unicode[6] consists of a repertoire of more than 120,000 characters covering 129 different scripts (including Latin, Cyrillic, Arabic, all the Japanese ones, and many more). Some points worth remembering are:

- The space character comes before all the printable characters.
- Numerals come in the order you'd expect, from "0" to "9".
- Letters come in the order you'd expect, from "A" to "Z" and from "a" to "z".
- Numerals come before upper-case letters and upper-case letters come before lower-case.

[6]The full list can be downloaded from many places, including Wikipedia (`http://en.wikipedia.org/wiki/List_of_Unicode_characters`).

Exercise A

Say, for each of the following pairs of strings, whether `s < t` would be true or false, assuming that `s` had the value on the left and `t` had the value on the right:

`"A"`	`"9"`
`"Zurich"`	`"acapulco"`
`"Abba"`	`"ABBA"`
`"long_thing_with_a_$"`	`"long_thing_with_a_&"`
`"King@example.invalid"`	`"King Kong"`

3.2 Two-way branches (`if ... else`)

The following program fragment tells students whether they have passed their exam:

Example

```
print("Please key in your exam mark: ", end = "")
s = input()
examMark = int(s)
if examMark >= 50:
    print("A satisfactory result!")
```

What happens, in the case of this program, if a student's mark is less than 50? The program does nothing. This kind of `if` statement is a one-way branch. If the condition is true, we do something; if not, we do nothing. But in this case this seems unsatisfactory. If the exam mark is less than 50, we would like it to display "Sorry, you have failed." We could arrange this by including another test – `if examMark < 50:` – or, better, we could do it by using the keyword `else`, thus:

Example

```
if examMark >= 50:
    print("A satisfactory result!")
else:
    print("Sorry, you have failed.")
```

The `else` turns a one-way branch into a two-way branch. If the condition is true, do this; otherwise, do that.

A note about the indent (the spaces we put before the statements "inside" the `if` statement). It has the effect of grouping all the statements with the same indentation level (the same amount of whitespace before them) into a programming unit called a *block*. Look at this example:

Example

```
if examMark >= 50:
    print("A satisfactory result!")
    print("You may proceed with your project.")
else:
    print("Sorry, you have failed.")
```

If the exam mark is greater than or equal to 50, the whole of the block is executed and both lines "A satisfactory result!" and "You may proceed with your project." will be printed. If the mark is lower than 50, the computer skips to the `else` and executes the "Sorry" line.

We can reduce the indent level again to tell Python where e.g. the block inside the `else` ends. We might add a line to the previous example to print a "Good bye." message in all cases:

Example

```
if examMark >= 50:
    print("A satisfactory result!")
    print("You may proceed with your project.")
else:
    print("Sorry, you have failed.")
print("Good bye.")
```

Suppose now that we wanted to give a different message to candidates who had done exceptionally well. Our first thought might be as follows:

Example

```
if examMark >= 70:
    print("An exceptional result!")
    print("We expect a first-class project from you.")
if examMark >= 50:
    print("A satisfactory result!")
    print("You may proceed with your project.")
else:
    print("Sorry, you have failed.")
```

But this would not work quite right. It's OK for candidates with marks below 70, but candidates with marks greater than or equal to 70 would give the following output:

```
An exceptional result
We expect a first-class project from you.
A satisfactory result
You may proceed with your project.
```

The problem is that if a mark is greater than or equal to 70, it is also greater than 50. The first condition is true, so we get the "exceptional" part, but then the second condition is also true, so we get the "satisfactory" part. We want to proceed to the greater than 50 test only if the mark is below 70. Here Python offers us the keyword `elif`, which is short for "else if" and which allows us to avoid excessive indentation:

Example

```
if examMark >= 70:
    print("An exceptional result!")
    print("We expect a first-class project from you.")
elif examMark >= 50:
    print("A satisfactory result!")
    print("You may proceed with your project.")
else:
    print("Sorry, you have failed.")
```

We can write more complex conditional expressions by joining them together using the operator **or**. For example, the following additional condition at the start of the last program checks if the exam mark keyed in is invalid, i.e., less than 0 or greater than 100:

Example

```
if examMark < 0 or examMark > 100:
    print("Invalid mark entered - needs to be in the range 0..100")
elif examMark >= 70:
    print("An exceptional result!")
    print("We expect a first-class project from you.")
elif examMark >= 50:
    print("A satisfactory result!")
    print("You may proceed with your project.")
else:
    print("Sorry, you have failed.")
```

Exercise B

Optional: read through the following section and try to solve its exercises:

`https://cscircles.cemc.uwaterloo.ca/6-if/`

Exercise C

Write a program that takes two numbers as input, one representing a husband's salary and the other representing the wife's, and prints out whether or not their combined income makes them due for tax at the higher rate (i.e., it exceeds £40,000).

Exercise D

Extend the program about students' marks so that all the candidates get two lines of output, the unsuccessful ones getting "Sorry, you have failed." and "You may re-enter next year."

Exercise E

Write a program which takes two integers as input. If the first is exactly divisible by the second (such as 10 and 5 or 24 and 8, but not 10 and 3 or 24 and 7) it outputs "Yes", otherwise "No", except when the second is zero, in which case it outputs "Cannot divide by zero". Remember you can use the modulo operator ("`%`") to find out whether one number is divisible by another.

Exercise F

Write a program which takes an integer as its input, representing the time using the 24-hour clock. For example, 930 is 9.30 am; 2345 is 11.45 pm. Midnight is zero. The program responds with a suitable greeting for the time of day (e.g., "Good morning", "Good afternoon", or "Good evening"). If you want to make this a bit harder, make the program respond with a "?" if the time represented by the number is impossible, such as 2400, -5 or 1163.

4 Loops and booleans

How would you write a program to add up a series of numbers? If you knew that there were, say, four numbers, you might write this program:

Example

```
print("Please key in four numbers: ")
print("> ", end = "")
s = input()
num1 = int(s)
print("> ", end = "")
s = input()
num2 = int(s)
print("> ", end = "")
s = input()
num3 = int(s)
print("> ", end = "")
s = input()
num4 = int(s)
total = num1 + num2 + num3 + num4
print("Total: " + str(total))
```

But a similar program to add up 100 numbers would be very long. More seriously, each program would need to be tailor-made for a particular number of numbers. It would be better if we could write one program to handle *any* series of numbers. We need a *loop*.

One way to create a loop is to use the keyword `while`. For example:

Example

```
num = 0
while num < 100:
    num = num + 5
    print(str(num))
print("Done looping!")
```

Similar to the `if` statement we saw earlier, it is essential to indent the lines inside the loop. This allows Python to see which lines are inside the loop and which lines are "after" the loop.

Having initialised the variable `num` to zero, the program checks whether the value of `num` is less than 100. It is, so it enters the loop. Inside the loop, it adds 5 to the value of `num` and then outputs this value (so the first thing that appears on the screen is a `5`). Then it goes back to the `while` and checks whether the value of `num` is less than 100. The current value of `num` is 5, which is less than 100, so it enters the loop again. It adds 5 to `num`, so `num` takes the value 10, and then outputs this value. It goes back to the `while`, checks whether 10 is less than 100 and enters the loop again. It carries on doing

this with `num` getting larger each time round the loop. Eventually `num` has the value 95. As 95 is still less than 100, it enters the loop again, adds 5 to `num` to make it 100 and outputs this number. Then it goes back to the `while` and this time sees that `num` is not less than 100. So it stops looping, goes on to the line after the end of the loop, and prints "Done looping!". The output of this program is the numbers 5, 10, 15, 20 and so on up to 95, 100, followed by "Done looping!".

Note the use of indent to mark the start and end of the loop. Each time round the loop the program does everything inside the indented block. When it decides not to execute the loop again, it jumps to the point beyond the indented block.

What would happen if the `while` line of this program was `while num != 99:`? The value of `num` would eventually reach 95. The computer would decide that 95 was not equal to 99 and would go round the loop again. It would add 5 to `num`, making 100. It would now decide that 100 was not equal to 99 and would go round the loop again. Next time `num` would have the value 105, then 110, then 115 and so on. The value of `num` would never be equal to 99 and the computer would carry on for ever. This is an example of an *infinite loop*.

Note that the computer makes the test *before* it enters the loop. What would happen if the `while` line of this program was `while num > 0:`? `num` begins with the value zero and the computer would first test whether this value was greater than zero. Zero is not greater than zero, so it would not enter the loop. It would skip straight to the end of the loop and finish, producing no output.

Exercise A

Optional: read through the following section and try to solve its exercises:

`https://cscircles.cemc.uwaterloo.ca/7c-loops/`

For our purposes, it suffices to look at the "`while` Loops" part.

Exercise B

Write a program that outputs the squares of all the numbers from 1 to 10, i.e., the output will be the numbers 1, 4, 9, 16 and so on up to 100.

Exercise C

Write a program that asks the user to type in 5 numbers, and that outputs the largest of these numbers and the smallest of these numbers. So, for example, if the user types in the numbers 2456 457 13 999 35, the output will be as follows:

```
The largest number is 2456
The smallest number is 13
```

4.1 Booleans (True/False expressions)

So far we have just used integer and string values. But we can have values of other types and, specifically, we can have *boolean* values.[7] In Python, this type is called "bool". Its values are also called "truth values", and they are `True` and `False`.

```
positive = True
```

Note that we do not have quote marks around the word `True`. The word `True`, without quote marks, is not a string; it's the name of a boolean value. Contrast it with:

```
positive = "True"
```

Here `positive` is assigned the four-character string "`True`", not the truth value `True`. (This is similar to the difference between, e.g., the number 42 and the string "42" that we saw earlier.)

You have already met boolean expressions. They are also called conditional expressions and they are the sort of expression you have after `if` or `while` before the "`:`". When you evaluate a boolean expression, you get the value `True` or `False` as the result.

Consider the kind of integer assignment statement with which you are now familiar:

```
num = count + 5
```

The expression on the right-hand side, the `count + 5`, is an integer expression. That is, when we evaluate it, we get an integer value as the result.

Now consider a similar-looking boolean assignment statement:

```
positive = num >= 0
```

The expression on the right-hand side, the `num >= 0`, is a boolean expression. That is, when we evaluate it, we get a boolean value (True/False) as the result. You can achieve the same effect by the following more long-winded statement:

```
if num >= 0:
    positive = True
else:
    positive = False
```

[7] The word "boolean" was coined in honour of an Irish mathematician of the nineteenth century called *George Boole*.

The variable `positive` now stores a simple fact about the value of `num` at this point in the program. (The value of `num` might subsequently change, of course, but the value of `positive` will not change with it.) If, later in the program, we wish to test the value of `positive`, we need only write

```
if positive:
```

You can write `if positive == True:` if you prefer, but the `== True` is redundant. `positive` itself is either True or False. We can also write

```
if not positive:
```

that is exactly the same as `if positive == False`. If `positive` is True, then `not positive` is False, and vice-versa.

Boolean variables are often called *flags*. The idea is that a flag has basically two states – either it's flying or it isn't.

So far we have constructed simple boolean expressions using the operators introduced in the last chapter – `x == y`, `s >= t` and so on – now augmented with negation (`not`). We can make more complex boolean expressions by joining simple ones with the operators `and` and `or`. For example, we can express "if x is a non-negative odd number" as `if x >= 0 and x % 2 == 1:`. We can express "if the name begins with an A or an E" as `if name[0:1] == "A" or name[0:1] == "E":`. The rules for evaluating *and* and *or* are as follows:

	left	`and`	right	left	`or`	right
1	True	**True**	True	True	**True**	True
2	True	**False**	False	True	**True**	False
3	False	**False**	True	False	**True**	True
4	False	**False**	False	False	**False**	False

Taking line 2 as an example, this says that, given that you have two simple boolean expressions joined by *and* and that the one on the left is True while the one on the right is False, the whole thing is False. If, however, you had the same two simple expressions joined by *or*, the whole thing would be True. As you can see, *and* is True if and only if both sides are True, otherwise it's False; *or* is False if and only if both sides are False, otherwise it's True.

Exercise D

Given that `x` has the value 5, `y` has the value 20, and `s` has the value "`Birkbeck`", decide whether these expressions are True or False:

```
x == 5 and y == 10
x < 0 or y > 15
y % x == 0 and len(s) == 8
s[1:3] == "Bir" or x // y > 0
```

4.2 Back to loops

Returning to the problem of adding up a series of numbers, have a look at this program:

Example

```
total = 0
finished = False
while not finished:
    print("Please enter a number (end with 0):")
    s = input()
    num = int(s)
    if num != 0:
        total = total + num
    else:
        finished = True
print("Total is " + str(total))
```

If we want to input a series of numbers, how will the program know when we have put them all in? That is the tricky part, which accounts for the added complexity of this program.

The variable `finished` is being used to help us detect when there are no more numbers. It is initialised to `False`. When the computer detects that the last number ("0") is introduced, it will be set to `True`. When `finished` is True, it means that we have finished reading in the input. The `while` loop begins by testing whether `finished` is true or not. If `finished` is not true, there is some more input to read and we enter the loop. If `finished` is true, there are no more numbers to input and we skip to the end of the loop.

The variable `total` is initialised to zero. Each time round the loop, the computer reads a new value into `num` and adds it to `total`. So `total` holds the total of all the values input so far.

Actually the program only adds `num` to `total` if a (non-zero) number has been entered. If the user enters something other than an integer – perhaps a letter or a punctuation mark – then the parsing of the input will fail and the program will stop, giving an error message.

A real-life program ought not to respond to a user error by aborting with a terse error message, though regrettably many of them do. However, dealing with this problem properly would make this little program more complicated than I want it to be at this stage.

You have to take some care in deciding whether a line should go in the loop or outside it. This program, for example, is only slightly different from the one above but it will perform differently:

Example

```
finished = False
while not finished:
    total = 0
    print("Please enter a number (end with 0):")
    s = input()
    num = int(s)
    if num != 0:
        total = total + num
    else:
        finished = True
print("Total is " + str(total))
```

It resets `total` to zero *each time round the loop.* So `total` gets set to zero, has a value added to it, then gets set to zero, has another value added to it, then gets set to zero again, and so on. When the program finishes, `total` does not hold the total of all the numbers, just the value zero.

Here is another variation:

Example

```
total = 0
finished = False
while not finished:
    print("Please enter a number (end with 0):")
    s = input()
    num = int(s)
    if num != 0:
        total = total + num
        print("Total is " + str(total))
    else:
        finished = True
```

This one has the `print` line inside the loop, so it outputs the value of `total` each time round the loop. If you keyed in the numbers 4, 5, 6, 7 and 8, then, instead of just getting the total (30) as the output, you would get 4, 9, 15, 22 and then 30.

Exercise E

Write a program that reads a series of numbers, ending with 0, and then tells you how many numbers you have keyed in (other than the last 0). For example, if you keyed in the numbers 5, -10, 50, 22, -945, 12, 0 it would output "You have entered 6 numbers.".

5 Extra exercises

This section provides some additional exercises for you, for more practice with Python. Use only the concepts presented in this booklet to solve the exercises.

Exercise X1

What does the following program do? This question is not about describing the instructions in the program one by one. Instead, try to find a *descriptive name* as a "summary" to tell the user of the program what output it will compute based on the user's inputs.

For example, for the first program in Section 4.2, we could give the summary: *"This program prints the sum of the numbers that the user has entered."* A good name for that program could be *"sum"*.

Try to answer this question without using a computer.

Example

```
m = 0
finished = False
while not finished:
    print("Enter another integer number (0 to finish): ", end = "")
    s = input()
    num = int(s)
    if num != 0:
        if num > m:
            m = num
    else:
        finished = True
print(str(m))
```

Exercise X2

If you have worked out what the above program does, can you see that, for certain series of integers, it will not produce the correct output? In what circumstances will it not work correctly, and how could you change the program to make it work properly? You may assume that the user will enter at least one number that is not 0 before the final 0.

Exercise X3

Write a program that takes a series of numbers (ending in 0) and counts the number of times that the number 100 appears in the list. For example, for the series 2, 6, 100, 50, 101, 100, 88, 0, it would output 2.

Exercise X4

Write a program that takes a series of lines of text (ending by an empty string) and, at the end, outputs the longest line. You may assume there is at least one line in the input.

Exercise X5 (this one is a bit harder)

Write a program that takes a series of numbers (ending in 0). If the current number is the same as the previous number, it says "Same"; if the current number is greater than the previous one, it says "Up", and if it's less than the previous one, it says "Down". It makes no response at all to the very first number. For example, its output for the list 9, 9, 8, 5, 10, 10, 0, would be Same, Down, Down, Up, Same (comparing, in turn, 9 and 9, 9 and 8, 8 and 5, 5 and 10, 10 and 10). You may assume there are at least two numbers in the input.

```
Enter the first number: 9
Enter the next number (0 to finish): 9
Same
Enter the next number (0 to finish): 8
Down
Enter the next number (0 to finish): 5
Down
Enter the next number (0 to finish): 10
Up
Enter the next number (0 to finish): 10
Same
Enter the next number (0 to finish): 0
```

Exercise X6 (still a bit harder)

Write a solution for exercise X5 that prints all the "Down", "Same", and "Up" messages together at the end.

```
Enter the first number: 3
Enter the next number (0 to finish): 5
Enter the next number (0 to finish): 4
Enter the next number (0 to finish): 4
Enter the next number (0 to finish): 6
Enter the next number (0 to finish): 8
Enter the next number (0 to finish): 2
Enter the next number (0 to finish): 6
Enter the next number (0 to finish): 7
Enter the next number (0 to finish): 5
Enter the next number (0 to finish): 6
Enter the next number (0 to finish): 6
Enter the next number (0 to finish): 7
Enter the next number (0 to finish): 0
Up Down Same Up Up Down Up Up Down Up Same Up
```

6 Summary of the language features mentioned in this introduction

`#`	Introduces a comment
`num = 99`	Assigns the integer value 99 to a variable called `num`
`word = input()`	Takes a string from the input and puts it into the variable `word`
`n = int(word)`	Parses the number in the variable `word` and puts its integer value into the variable `n`
`word = str(n)`	Converts the number in the variable `n` to its string representation and puts it into the variable `word`
`print(word)`	Outputs the value of `word`
`print(word, end = "")`	The same, but without the return-of-line at the end
`print("The answer is " + x)`	Outputs "The answer is " followed by the string `x`
`+`, `-`, `*`, `//`, `%`	Arithmetic operators. `*`, `//` and `%` take precedence over `+` and `-`.
`len(s)`	Gives length of string `s`
`s[x:y]`	Gives substring of `s` starting with the character at position `x` and ending with the last character before position `y`. (The first character of the string is at position zero.)

`if A:`
`  B`
`elif C:`
`  D`
`else:`
`  E`
	If `A` is True,
	  do `B`,
	else if `C` is True,
	  do `D`,
	else
	  do `E`.

`x == y`	Tests whether `x` is equal to `y`. Note the double "=".
`!=`, `>`, `<`, `>=`, `<=`	Not equal to, greater than, less than, etc.
`while A:` `  B`	While `A` is True, do `B`. `B` can be a block with several statements.
`True`, `False`	The boolean values *True* and *False*.
`not`, `and`, `or`	The boolean operators *not*, *and* and *or*.

7 Obtaining, installing and running Python on a PC

You can download a free copy of Python from the web:

`https://www.python.org/downloads/`

It is important that you download version 3.. . . , which is what we use in this introduction. Python is available for many operating systems, including Windows, Linux, and Mac.

Python will be used intensively in several of the MSc courses at Birkbeck, so it is a good idea to install it if you do not have it yet. If you do not know whether you have Python installed in your system, you can open a command prompt[8] and type `python3 --version`. If Python is installed, the result should be something similar to this:

```
> python3 --version
Python 3.4.3
```

Running Python programs

Python is run from the command line.

You first type your program in a text editor. Notepad (in Windows) or gedit (in Linux) are good options, but any simple editor will do. Note: If you use a word-processor (like Microsoft Word or LibreOffice Writer), make sure you save the file as plain text.

Give the filename a `.py` extension.

Open the command prompt. Go to the folder where you have saved your Python program using the command "cd" (change directory). (You do not have to do this but it is easier if you do, otherwise you would have to type the full path for your source code file to run it.)

Suppose your program is in a file called `Myprog.py` in the current folder. You can run your program by typing

```
python3  Myprog.py
```

If the Python interpreter accepts your program, you will see the result of your program on the screen. If there are errors, you will get error messages and you need to go back to the text editor, correct the program and save it again, then try to run your program again.

If you cannot install Python. . .

If you have problems installing Python, you may find this webpage useful:

`https://repl.it/languages/python3`

[8] You can find it in Windows in "Accessories".

The page provides you with an editor on the left and a console on the right. It will allow you to test your programs online. It may be quite slow compared to using a local machine, but it may help if you find problems installing Python.

Additionally, if you have any problem installing or running Python, feel free to write to Dr. Carsten Fuhs (carsten@dcs.bbk.ac.uk). We will come back to you as soon as possible.

References

[1] Charles R. Severance. *Python for Everybody – Exploring Data Using Python 3*. 2016. Available online at `https://www.py4e.com/book` and `http://do1.dr-chuck.com/pythonlearn/EN_us/pythonlearn.pdf`; interactive edition at `https://books.trinket.io/pfe/`.

[2] Allen Downey. *Think Python – How to Think like a Computer Scientist*. Green Tea Press, 2nd edition, 2015. Available online at `http://www.greenteapress.com/thinkpython2/index.html` and `http://greenteapress.com/thinkpython2/thinkpython2.pdf`; interactive edition at `https://runestone.academy/runestone/books/published/thinkcspy/index.html`.