Introduction to Python

An introduction to the Python programming language for those who haven’t programmed before

Version 2020-08
Licence

This manual is © 2020, Steven Wingett & Simon Andrews.

This manual is distributed under the creative commons Attribution-Non-Commercial-Share Alike 2.0 licence. This means that you are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.
- Non-Commercial. You may not use this work for commercial purposes.
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

Please note that:

- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Full details of this licence can be found at
http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode
# Table of Contents

**Introduction** .................................................................................................................. 5
  - Why learn Python? .......................................................................................................... 5
  - What this course covers ............................................................................................... 6
  - Python 2 or Python 3? .................................................................................................. 6
  - A note about learning a programming language ....................................................... 6

**Chapter 1: getting started with Python** .......................................................................... 7
  - How to install Python .................................................................................................. 7
    - On Linux/Unix/MacOSX etc. .................................................................................... 7
    - On Windows ............................................................................................................ 7
  - What do I use to write Python code? ......................................................................... 8
  - Thonny ....................................................................................................................... 10
  - Hello World! and getting started with Python ......................................................... 11

**Chapter 2: data types and expressions** ......................................................................... 12
  - Integers (int) ............................................................................................................. 12
  - Floats ......................................................................................................................... 13
  - Operations on strings ............................................................................................... 14
    - Operations on strings ............................................................................................ 15
  - Booleans (bool) ......................................................................................................... 18
  - Compound expressions ............................................................................................. 21

**Chapter 3: names, functions and methods** .................................................................... 22
  - Writing scripts using Thonny ..................................................................................... 22
  - Names ......................................................................................................................... 23
  - F-strings ..................................................................................................................... 25
  - Functions ................................................................................................................... 26
    - Built-in functions .................................................................................................. 26
    - Functions and data types ...................................................................................... 28
    - Creating Functions ............................................................................................... 29
  - Methods ....................................................................................................................... 34

**Chapter 4 – Collections** ............................................................................................... 36
  - Sets ............................................................................................................................... 36
  - Sequences .................................................................................................................. 38
    - Ranges ..................................................................................................................... 39
    - Tuples ..................................................................................................................... 39
    - Tuples and functions ............................................................................................ 41
  - Lists ............................................................................................................................. 42
  - Dictionaries ............................................................................................................... 43
  - Streams ....................................................................................................................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>46</td>
</tr>
<tr>
<td>Generators</td>
<td>46</td>
</tr>
<tr>
<td>Copying collections</td>
<td>46</td>
</tr>
<tr>
<td>Operations on collections</td>
<td>48</td>
</tr>
<tr>
<td>Using the keyword <code>in</code> with collections</td>
<td>49</td>
</tr>
<tr>
<td>Further points on manipulating collections with functions and methods</td>
<td>50</td>
</tr>
<tr>
<td>The key parameter</td>
<td>50</td>
</tr>
<tr>
<td>Key expressions</td>
<td>50</td>
</tr>
<tr>
<td>Anonymous functions (the lambda function)</td>
<td>51</td>
</tr>
<tr>
<td>Chapter 5 – Conditionals, Loops and Iterators</td>
<td>52</td>
</tr>
<tr>
<td>Conditionals</td>
<td>52</td>
</tr>
<tr>
<td>Simple, one-alternative and multi-test conditionals</td>
<td>52</td>
</tr>
<tr>
<td>Loops</td>
<td>53</td>
</tr>
<tr>
<td>While Loops</td>
<td>54</td>
</tr>
<tr>
<td>Processing file input using a loop</td>
<td>55</td>
</tr>
<tr>
<td>Iterations</td>
<td>56</td>
</tr>
<tr>
<td>Simple Iteration</td>
<td>56</td>
</tr>
<tr>
<td>Iterating over a file</td>
<td>57</td>
</tr>
<tr>
<td>Iterating over a dictionary</td>
<td>57</td>
</tr>
<tr>
<td>Enumerating iterations</td>
<td>57</td>
</tr>
<tr>
<td>Iterating over ranges</td>
<td>58</td>
</tr>
<tr>
<td>Nested iterations</td>
<td>58</td>
</tr>
<tr>
<td>Iterating over a string</td>
<td>58</td>
</tr>
<tr>
<td>Passing iterables to functions</td>
<td>59</td>
</tr>
<tr>
<td>Concluding remarks</td>
<td>60</td>
</tr>
</tbody>
</table>
Introduction

Why learn Python?

In recent years, the programming language Python has become ever more popular in the bioinformatics and computational biology communities and indeed, learning this language marks many people’s first introduction to writing code. This success of Python is due to a number of factors. Perhaps most importantly for a beginner, Python is relatively easy to use, being what we term a “high-level” programming language. Don’t let this terminology confuse you however: “high-level” simply means that much of the computational tasks are managed for you, enabling you to write shorter and simpler code to get your jobs done.

Since Python is widely used, there is a large community of people who potentially can give advice or co-author code with you. Many people are familiar with the language and so if you write something in Python, it is more likely to be used than if you wrote the same program in a more obscure language. The wide adoption of Python also means there are add-ons that can be installed to increase the flexibility of the language. For example, numerous mathematical and scientific Python packages can be used for analysis and data presentation. There are also platforms available – such as Jupyter – to help you present and share your Python code along with your results, providing an excellent way to disseminate your findings in a transparent and reproducible fashion. In addition, Python is a leader in the burgeoning field of machine learning.

Although Python may be regarded as relatively easy to learn, that does not mean it is restricted to simple scripts. The language also allows the user to write “object-orientated” programs – a style of coding designed for larger and multifaceted applications. In addition to all these benefits, learning Python will enable you to perform a variety of tasks outside of the field of bioinformatics. A common use of Python is in the building of websites, using what are termed frameworks. Django and Flask are the most noteworthy of these web creation tools.

Another important factor in the success of Python is that is available to all for free. Actually, that is not as surprising as it may first sound to those new to programming, for the major languages (e.g. Perl, Java, C) are distributed at no cost to the end user. This does not however mean that these are sub-standard products, with better “paid-for” alternatives readily available to those wishing to open their wallets. This is just the way the computing ecosystem works and has evolved over the decades. These freely available languages are the mainstay of the software environment and furthermore, much of the commercial software sold today is written using these languages.

So, to summarise, there are many reasons why Python is a good language to learn and use:

Good things about Python:
- It works on the vast majority of computers
- It’s relatively easy to write
- There is a large community of developers who may have already written a program you require, or may be able to help you write your own code
- There are extensive scientific and mathematical “add-ons” to Python to help you with your code analysis. The field of Machine Learning is one area where a knowledge of Python is a distinct advantage.
- It allows you to develop programs in a short amount of time
- It’s free
Bad things about Python
- Its flexibility means that, in some situations, it can be slower than other languages

What this course covers
This course introduces the basic features of Python. At the end you should be able to write moderately complicated programs, and be aware of additional resources and wider capabilities of the language to undertake more substantial projects. The course tries to provide a grounding in the basic theory you’ll need to write programs in any language as well as an appreciation of the right way to do things in Python.

Python 2 or Python 3?
Python 3 was released in 2008 to supersede Python 2, which underwent its last update in 2018. Although these versions of Python are very similar – essentially dialects of the same language – there may be compatibility problems if trying to use the two interchangeably. Moving forwards, we will see most new software being written in the newer version of Python, replacing many of the scripts and modules written in the previous language. This development was a major change in the short history of Python, but pleasingly, no such major changes are planned for the future. We therefore recommend learning Python 3, which is the version taught in this course.

A note about learning a programming language
While Python may be considered less demanding than some other programming languages, this does NOT mean that learning Python is easy. In fact, beware of courses with dubious titles such as “Master Python in 60 Seconds”. Learning a programming language is akin to learning a foreign language, and while gaining useful skills may come quickly, a solid understanding of programming takes time and commitment. That said, acquiring the most valued skills is not easy, but instead demands effort and dedication.
Chapter 1: getting started with Python

How to install Python

On Linux/Unix/MacOSX etc.
It may already be installed! Python is made available on many Unix-based operating systems. To check whether Python is installed, go to the command line (open the “Terminal” on a Mac) and type:

```
python --version
```

Alternatively, on some systems, you may need to enter:

```
py --version
```

If Python is installed, you should then see a message stating the exact version of Python installed, similar to that below:

```
Python 3.7.0
```

If Python 2 version is now displayed, then you may check whether Python3 is also installed with the command:

```
python3 --version
```

The Python 3 version installed will now be reported, else your system will display an error message.

If Python 3 is not installed, you will need to download it from The Python Software Foundation website at: [www.python.org](http://www.python.org). (Note: from now onwards in this manual, you should assume that the word Python refers to Python 3, unless specified otherwise.)

When there, click on the “Downloads” link and you will be navigated to a page from which different versions of Python may be downloaded. This webpage should automatically detect the operating system you are running and display a button which, when pressed, will initiate the downloading process of the latest Python version compatible with your setup.

Should you wish to download another version of Python, simply click on the relevant operating system and version you wish to download. For a variety of reasons (such as security and efficiency), we recommend downloading the latest stable release of the software.

Once you have downloaded the software, follow the standard procedure you would follow on your system for installing a program. Once finished, follow the steps above to check that Python is indeed installed.

On Windows
If you’re not sure whether you have Python installed, you can easily find out by opening the Command Prompt. Depending on your version of Windows, there are different ways to open the Command Prompt:

1) Try right-clicking on your desktop and it’s probably listed as one of the options

2) Look in Start > Programs for an entry called “MS-DOS Prompt”

3) Look in Start > Programs > Accessories for an entry called “Command Prompt”
4) Go to Start > Run. In the box type “cmd” and press return

One of these alternatives should get you a command prompt.

At the command prompt type in the command

```bash
python --version
```

or

```bash
py --version
```

If python is installed you should see something like this:

![Command Prompt]

---

**What do I use to write Python code?**

Complete beginners to programming may be surprised to learn that Python programs are actually just plain text files. Consequently, you can use any plain text editor to write Python code. On a PC for example, you could create Python scripts with Notepad. While this is possible, you really don’t want to write code this way. This is because there are text editors created specifically for the purpose of writing code that boast a large number of features to make this process easier. For example, what is almost obvious immediately when using such an editor for the first time is that the text is displayed in a variety of different colours. Each of these colours will have a different meaning and will help you understand at a glance the basic structure of a line of code. There are a variety of editors available and what you end up using is often a result of personal taste and familiarity. We summarise below some of the most frequently encountered text editors commonly used for writing Python.

**Vi**

The strength of this software is that it should be installed on almost all Unix/Linux distributions. So, if you are working on such a system you should be able to start coding without downloading any additional software. For this reason, and owing to its established history in computer science, this is a commonly used program. However, we would advise those new to coding to start somewhere else. Vi is not the
most user-friendly environment (particularly to those only familiar with MS Windows). It uses a command line interface and requires the user to learn a number of commands to run the program.

**Emacs**
This is well-known software to those familiar with Linux and while it is commonly used on this platform, it can also be run on Windows and Mac OS. While it is powerful, many new users find the shortcuts a little unintuitive (again, particularly with those familiar to Microsoft software). Emacs can be downloaded from: [https://www.gnu.org/software/emacs/](https://www.gnu.org/software/emacs/)

**Notepad++**
Is a popular free text editor and very easy to use for beginners and its numerous add-ons make it powerful to use. However, it is only available for Windows. It may be downloaded from [https://notepad-plus-plus.org/downloads/](https://notepad-plus-plus.org/downloads/)

**Sublime Text**
Like Notepad++, this is a simple-to-use free application with a powerful choice of add-on tools. In addition, it is also available for Windows, Mac and Linux.

**Visual Studio Code**
This is another versatile text editor available for the three main types of operating system. It is a rising star in the software development community and although produced by Microsoft, it is a free to use. Visual Studio Code also has many useful extensions that help with programming. I recommend this software, although it is not quite so user-friendly initially as Notepad++ or Sublime Text.

**Atom**
Atom is another text editor popular with Python programmers and, similar to Visual Studio code, has available a wide range of add-ons to assist development.

**PyCharm**
PyCharm is popular with professional developers and is a good choice for coding more substantial projects. Again, it is free and available for Windows, Mac and Linux. Furthermore, it is what is termed an Integrated Development Environment (IDE) allowing the users to run their scripts directly using the program. While it is a very good application, it is perhaps a little overwhelming for the complete novice.

**IDLE**
Another popular IDE is IDLE, which is bundled along with standard Python installations.

**For this course, we shall be using:**

**Thonny**
It is similar to the tools mentioned previously in a variety of ways: being free and available for Mac, Windows and Linux operating systems. It also has an IDE capability similar to PyCharm, but it has a much simpler interface making it much friendlier for those new to coding. If you have ever used a Raspberry Pi, you may have already come across Thonny. Another benefit of Thonny (certainly as far as this course is concerned), is that it comes with its own version of Python. So, once you have installed Thonny, you are ready to start writing Python code.
**Thonny**

Thonny can be downloaded from its homepage at: https://thonny.org. Here, there should be links which will commence the file transfer of the latest version of the software, as either an *.exe file for Windows systems, or a *.dmg file for use with MacOS. After downloading, simply follow the usual method you use to install such files (this typically involves double-clicking the files and then following the on-screen instructions). Clicking the link for the Linux version of Thonny will display a pop-up giving instructions on how to install the software via the command line. On Linux Ubuntu systems and its derivatives, the download and installation process is achieved with the command:

```
sudo apt install thonny
```

After installing Thonny, double-click on the software icon (a graphical representation of a “Th”) to start the program. After doing this, a window similar to that shown below should appear on your screen. The functionality of Thonny is described in greater detail as the course progresses. All you need to know at the moment is that the Thonny window has three components. At the top of the window there is a panel of icons which are used for opening or saving files or for running new Python programs you have created. Below this is a window into which you may type Python code. Text written here may be saved as a file, in a similar fashion to how you would save any text file. The bottom panel also allows you to enter Python commands. However, text entered here may not be saved, for this is an interactive window which displays to the screen the computer’s responses to your Python code. This is explained in more detail in the next section, in which you will write your first Python commands.

Thonny has a helpful **tab-autocomplete** coding aid feature. The program will predict the intended code if the user presses the tab key. Should Thonny predict more than one likely possibility, a drop-down menu will be displayed from which the user may select the correct option. This feature is beneficial, for it speeds up coding, prevent typos being introduced, and allows users to see potential coding options.

(This interactive window is known technically as a **shell**. A shell allows users to pass text commands to a computer, enabling those proficient to perform complex tasks efficiently. Most people using computers are familiar with the contrasting Graphical User Interface (GUI), that provides the capability to interact with a computer by moving an on-screen pointer via a mouse.)
Hello World! and getting started with Python

Now that you have Python installed, we shall get started with writing Python code. Traditionally, when beginning a new language, the first program a beginner writes instructs the computer to display “Hello World!” on the screen. Seeing no good reason to break this convention, we shall do the same.

For these early examples we shall use the interactive functionality of Thonny by typing into the bottom panel of the window. Type in the following text and then press enter:

```python
print("Hello World!")
```

You should now see “Hello World!” printed to the screen. Congratulations, you have now written your first Python program! We shall now look some more at running simple one-line commands in this interactive fashion as we learn about Python data types in the next section.
Chapter 2: data types and expressions

In this section we introduce four basic data types used by Python. The Integer and Float data types store numbers, the Boolean data type stores logical values and String data type stores characters (such as text). While we will encounter other data types later in the course we shall start with these most basic of data types. (In computing terminology, such simple data types are known as Primitives.)

A programmer can manipulate these data types in Python using an operator. A command which includes one or more data type and an operator is known as an expression – essentially a small sentence telling the computer to do something. We shall start by looking at the integers.

**Integers (int)**

Integers in Python have the same meaning as in mathematics: these are whole numbers and can be positive, negative or zero. For example: 10, 23 and -18 are all integers. In contrast: 1.5, -0.2 and √2 are not integers, since they have values after the decimal point.

If you type an integer in the Thonny interactive window, you will see that it will be displayed back to you on the subsequent line. Try typing in 100:

```python
>>> 100
100
```

Integers can be manipulated by operators. The addition and subtraction operators will be familiar to you from basic maths. Enter the following in the Thonny interpreter window:

```python
>>> 1 + 2
3

>>> 3 - 5
-2
```

When a data type is manipulated by an operator, the data type is referred to as an operand. In the 1 + 2 example above, the values 1 and 2 are operands to the addition operator (+). Since plus and minus take two operands, they are known as binary operators. (If you think about this a little more, the minus operator can also take only one operand. For example: when entering -1, the minus operator is modifying the positive integer to become negative. In this case the minus operator is known as a unary operator.)
Python can also be instructed to perform multiplication. The multiplication symbol used by Python is not the same as used in standard maths textbooks, for it is an asterisk (*). Try entering the following in the Python interpreter in Thonny:

```python
>>> 2 * 3
6

>>> 5 * 5
25
```

As you may expect, it is possible in Python to raise a number to a given power. This is denoted by a double asterisk operator (**). Try the following to square a number:

```python
>>> 5 ** 2
25
```

(Integers may also be represented in what is known as hexadecimal notation, which is base-16 instead of base-10. The letters A...H represent the decimal system numbers 0...15. Hexadecimal notation begins with 0x. 0xB50 would be represented in base-10 as 2896.)

After having covered addition, subtraction and multiplication, naturally the next operator to describe is division. Before we do this, we need to introduce the Float.

**Floats**

Floats, or to use the full term: “floating point numbers”, represent numbers in a more complex way than integers. Floats are used by computers to store non-integer numbers. You may never have thought about this before, but there is a limit to how much precision a computer can store a non-integer value. For example, one-third is represented as a decimal as 0.333 recurring. A computer obviously cannot store an infinite number of 3s in memory, so the computer will need to round to an appropriate level of precision.

Floats have two components: the *significand* and the *exponent*. The former stores the significant numbers (these can be negative), while the exponent defines the position of the decimal place. For example, the value 0.5 would have a significand of 5 and an exponent of -1. In fact, integer values may be represented as floats, for example 1000 would have a significand of 1.0 and an exponent of 3, but storing integers in this way uses more of the available memory. (To calculate the value, multiply the significand by 10 raised to the power of the exponent.)

<table>
<thead>
<tr>
<th>Value</th>
<th>Significand</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>0.001</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>-10.5</td>
<td>-1.05</td>
<td>1</td>
</tr>
</tbody>
</table>

Having said all this, you generally won’t notice any difference in the Thonny interactive window when entering floats as compared to integer variables. One difference, however, is that floats may be entered using scientific notation. Type the following in the bottom window in Thonny to show that scientific notation is interpretable by Python.

```python
>>> 3.0E10
```
We started discussing floats because they are important for division. This is a result of the fact that many division operations (even involving only integers) will return fractional values. Indeed, division operations in Python always return the float type (even if the value returned is not fractional).

To divide in Python, use the forward slash (/) character:

```python
>>> 1 / 3
0.3333333333333333
```

There are related operators in Python3, such as floor division (//=), which performs a division and subsequently returns the resulting integer, after the remainder removed. The modulo (%) operator works in a similar fashion, but instead returns the remainder.

```python
>>> 15 / 7
2.142857142857143

>>> 15 // 7
2

>>> 15 % 7
1
```

Just to reiterate: be aware that operations involving integer types and floats will always return floats.

**Strings (str)**

These store Unicode characters, whether that is a letter, number or a symbol of some kind. Essentially, they constitute a “string” of characters – although a single character all on its own is allowed. To create a string you will need to enclose your text within either single or double quotes.

```python
>>> 'abcde12345'
'abcde12345'

>>> "abcde12345"
'abcde12345'
```

It is possible for a string to span multiple lines, but it will need to be enclosed within a pair of three single quotes or a pair of 3 double quotes:

```python
>>> print("""line1
line2""")
line1
line2
```

A more convenient way to do this is to include backslash n (\n) in the string:
>>> print("line1\nline2")
line1
line2

This rather strange notation of \n to represent a line ending constitutes what is known as a Python **metacharacter**. As you become more experienced with Python you will encounter these metacharacters again and again. They are a way of using characters to represent something other than simple characters. Metacharacters form an import part of the Python vocabulary and you should take time to become familiar with using them.

Many metacharacters have a backslash immediately before a letters, causing them to be interpreted differently by Python. As you can see above, \n is interpreted as a new line, whereas \t shown below is interpreted as a tab:

```python
>>> print("line1\tline2")
line1   line2
```

You may be wondering whether it is possible to include quotation marks in a string. Well, the answer is yes and the way to achieve this is to use different types of quotation marks (i.e. single vs double) to denote the literal text as compared to the characters that should be included within the string. For example, compare:

'You\'re gonna need a bigger boat'
"You\'re gonna need a bigger boat"

The first line will fail since the apostrophe in You’re will actually be interpreted as the end of the string. In the second example, this will not happen since a double quotation mark (instead of a single) is used to define the string’s contents.

An alternative way to achieve the same thing is to place a backslash before the speech marks or apostrophe found within the string:

'You\\'re going to need a bigger boat'
"You\\'re going to need a bigger boat"

**Operations on strings**

**The plus (+) operator**

Although you may think of as plus as solely to be used with numbers, in Python it can also be used to **concatenate** (i.e. join) two strings.

```python
>>> 'Love' + 'Marriage'
'LoveMarriage'
```

**Multiplication (*) of strings**

In a similar fashion, the multiplication operator can be used on strings. Multiplying a string by an integer causes the string to be repeated.
>>> 'Go forth ' * 5
'Go forth Go forth Go forth Go forth Go forth ' 

The in operator
Strings may also be evaluated by operators. Two useful string functions are in and not. These check for the presence (or absence) of a string (first operand) within another string (second operand), returning the appropriate Boolean value.

>>> 'Needle' in 'HaystackHaystackNeedleHaystackHaystackHaystackHaystack'
True

>>> 'Elvis' not in 'Building'
True

In the first example, True is returned since the text contains the string ‘Needle’. Notice in the second example that the presence of not reverses the result.

Be aware that this string lookup function is case sensitive:

>>> 'NEEDLE' in 'HaystackHaystackNeedleHaystackHaystackHaystackHaystackHaystack'
False

Subscription and slicing
Parts of a string may be retrieved by specifying the position within the string that is required. This is achieved by placing square brackets after the string and entering within them the numerical value of the desired position. For example, suppose you wanted the first character from a string:

>>> 'Babraham'[1]
'a'

Hmmm, that didn’t work since the second letter was returned. That was because numbering strings actually begins at 0! This may seem a little odd, but this numbering convention is used on multiple occasions in Python and in countless other programming languages. So then, try this instead:

>>> 'Babraham'[0]
'B'

Similarly, the third character can be retrieved with:

>>> 'Babraham'[2]
'b'

That worked, but watch out for this numbering convention in future since it catches out many who are new to Python.

The numbering system described above works in a left-to-right direction, but it is also possible to number in a right-to-left direction by using negative values in the square brackets:
>>> 'Babraham'[-1]
'm'

>>> 'Babraham'[-3]
'h'

(Somewhat confusingly, to return the right-most character requires [-1], whereas [0] is used to retrieve the left-most character.)

Numerical operations can be performed within the square brackets:

>>> 'Babraham'[100 - 99]
'a'

The numerical calculation is performed first, returning a value of 1, and then consequently the Python interpreter returns the second character of the input string. If you were to specify beyond the end of the string, the interpreter would return an error message.

It is also possible to extract multiple characters from a string, known as a slice. To extract a slice, simply type the range of positions you wish to extract in the square brackets, separating the start and end positions with a colon:

>>> 'Babraham'[1:4]
'abr'

You will see that the slice does indeed begin at position 1 (the second character in the string). However, the string goes up to, but does not include the end position.

As you may expect, you can enter negative values in the splice:

>>> 'Babraham'[-4:-2]
'ah'

Or a mixture of negative and positive values:

```python
>>> 'Babraham'[2:-2]
'brah'
```

A shortcut is to leave the space before or after the colon empty. The space is an instruction to retrieve values from the start of the string, or up until the end of the string, respectively.

```python
>>> 'Babraham'[:3]
'Bab'
```

```python
>>> 'Babraham'[5:]
'ham'
```

```python
>>> 'Babraham'[:]
'Babraham'
```

It is also possible to specify a third value between the square brackets, which gives instructions on the step size to employ:

```python
>>> 'Babraham'[:3]
'Bra'
```

In the above example, the start and end position are missing, which instructs every character. However, the 3 instructs Python to only return every third character.

It is also worth noting that “impossible” splices will return empty strings:

```python
>>> 'Babraham'[2:1]
''
```

### Booleans (bool)

The simplest Python data type is the Boolean, which has only two values: either True or False. Boolean values are generated when performing tests using the **comparison operators**.

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>If the values of two operands are equal, then the condition is True</td>
</tr>
<tr>
<td>!=</td>
<td>If values of two operands are not equal, then condition is True</td>
</tr>
<tr>
<td>&gt;</td>
<td>If the value of left operand is greater than the value of right operand, then condition is True</td>
</tr>
<tr>
<td>&lt;</td>
<td>If the value of left operand is less than the value of right operand, then condition is True</td>
</tr>
</tbody>
</table>
If the value of left operand is greater than or equal to the value of right operand, then condition is True

If the value of left operand is less than or equal to the value of right operand, then condition is True

To demonstrate the use of these operators, please see the results below using the Thonny interactive window.

```python
>>> 1 == 1
True
>>> 1 == 0
False
>>> 2 > 5
False
```

(Please note that the comparison operator to check whether two values are equal is ‘==’ and not the more familiar ‘=’, which has a different meaning in Python.)

These datatypes follow Boolean logic when used in expressions. Try entering the values below into the Thonny interactive window:

```python
>>> True
True
>>> False
False
```

As should be expected from previous examples using different data types, entering a value in the window causes Thonny to return the same value on the next line.

Please note that True and False values are case sensitive:

```python
>>> true
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'true' is not defined
```

Entering “true” instead of “True” has resulted in an error message, since ‘true’ is not recognised by the Python interpreter.

The commonly used Boolean operators are: and/or/not. These binary operators make intuitive sense when considering how we use these words in the English language:

```python
>>> True and True
True
>>> True and False
False
```
In contrast, the or operator behaves differently:

```python
>>> True or True
True
>>> True or False
True
```

The not operator converts True to False, and vice versa.

```python
>>> not True
False
>>> not False
True
```

Maybe a little surprisingly, the and, or and not operators can take values other than True or False as operands. Python considers almost everything as True, except for a few exceptions. The number zero is such an exception, and consequently is considered as False. The following examples illustrate this point. At first it may not be clear what is going on, but a logic is being followed to generate the results. Firstly, it is worth noting that when using logical operators this way, an input argument is returned, which may not necessarily be a Boolean value.

When using the and operator in an expression that evaluates to False, the first value corresponding to False will be returned. If there are no False values, the last value will be returned. In contrast, when using the or operator, Python will return the first value evaluating to True. Should no such value be present, the last value in the expression will be returned.

These above rules are, at first, a little confusing, but once you understand the logic that is being followed the rules become easier to remember. Essentially, Python reads from left to right and returns a value once a decision can be made about the Boolean test being performed. For example, when we use the and operator, the values both side of the operator need to be True. Consequently, even if we encounter a value that evaluates to False on the left-hand side of the operator, we already know the statement is False and need not evaluate any further. In contrast, when using the or operator, we need to process the right-hand side as well, for if either side evaluates to True then according to Boolean logic, the expression will be True. *The value returned is the value at which a decision is made.*

An expression that evaluates to False:

```python
>>> 1 and 0
0
```

An expression that evaluates to True:

```python
>>> 1 and 2
2
```

An expression that evaluates to True:

```python
>>> 1 or 0
1
```
An expression that evaluates to `True`:

```python
>>> 1 or 2
1
```

There are non-numeric values that evaluate to `False`. One such example is a special data type referred to as `None`. The data type essentially mean “nothing”. Enter it on the command line and you will see nothing is returned.

```python
>>> None
>>> 1 and None
>>> 1 or None
1
```

Another value that evaluates to `False` is an empty string datatype. But to understand this we shall need to introduce strings.

**Compound expressions**

All the expressions so far listed involve one operator and one value, but a single expression may contain many values and expressions. Here is a simple example involving only integers:

```python
>>> 5 + 5 * 3
20
```

If one performed the above calculation in a left-to-right order, the answer be 30 and not 20 (i.e. $5 + 5 = 10$ and then $10 \times 3 = 30$). However, Python follows orders of precedence and will perform the multiplication before the addition (which are the formal rules in mathematics). Thus, $5 \times 3 = 15$. Then add 5, which makes 20. You could of course learn the rules of precedence and write code accordingly, but virtually no programmer does this. The way around this extra complexity is to use parentheses in your code. The expression inside the parentheses is evaluated first and then this passed to values outside the parentheses. Python also allows nested parentheses (brackets within brackets), which are useful in complex expressions. Don’t worry about using multiple parentheses in your code; in fact, their inclusion should help make your code easier to read.

Consequently, you could re-write the above expression above as:

```python
>>> (5 + 5) * 3
30
```

Which now gives the expected answer.
Chapter 3: names, functions and methods

Writing scripts using Thonny

So far in this course we have been writing single-line expressions and running them in the Thonny interactive window. While this is fine, it is not the best way to handle more complex pieces of code. Although in this section we shall not be writing any particularly complex code, this is nevertheless a good place to introduce Python scripts.

Python scripts are text documents in which code is stored. Writing code to a script makes it easier to navigate and edit the particular line in your code you wish to change, as compared to the interactive mode. Moreover, code written into a script can be saved for use at a later date.

Open Thonny and type into the top window the Hello World! Python code:

```python
print("Hello World!")
```

Now you need to save the code as a script. To do this, in the top navigation bar, click on File -> Save as... A pop-up window should now appear. In the “Save As” text file, type “hello_world.py”, which will be the name of your script. Get into the habit of using the “.py” file extension for Python scripts. Now click “Save”. You should now see that this file has been saved on your PC.

Once you have done this, close the window tab containing Hello World! Script and then close Thonny. Now, open Thonny and go to File -> Open. Use the pop-up window that appears to open the Hello World! script you just created. If this has worked, you should see your code displayed in the top window.

You can now run this script by going to Run -> Run current script in the top window menu bar. This should result in the text below being displayed in interactive Shell window:

```bash
>>> %Run hello_world.py
Hello World!
```
You should have noticed that when writing code, Thonny kindly colours the text for you. This is not an aesthetic exercise, for writing components of Python code in different colours should help you comprehend code more quickly. You will see that in your Hello World! script that the print command is written in magenta, while the text to print to screen is written in green and the brackets are coloured in black. If you move the cursor next to one of the brackets you should notice that the pair of brackets are then highlighted using a blue font. We shan’t go into detail on how this colour scheme works since it should become familiar to you with usage (also, the colour scheme followed may vary from text editor to text editor). The more you use Thonny, the more helpful you should find this way of presenting code.

To illustrate the point, open the hello_world.py script in a simple text editor (such as Notepad on Windows systems). You can imagine how hundreds or even thousands of lines of code are easier to read with consistent and intelligent colouring.

Incidentally, you could run the same script using the command line via the Windows Dos prompt, the MacOS Terminal or a Linux Shell. Simply open the interactive window found on your operating system, navigate to your script and type:

```
python3 hello_world.py
```

(If you have python2 installed on your system, simply typing “python” may run this older version of the language. Specifying the version number, as done here with python3, can help prevent this confusion.)

**Names**

You are probably already aware that in algebra, letters are used to represent numbers. This idea is a central concept of Python and is achieved using an assignment statement. For example:
>>> a = 1

This assigns the name \texttt{a} to the value 1. Names can be assigned to integers, strings and other data types. As you may expect, when assigning to a string you will need to surround your string with quotation marks:

>>> b = 'MyString'

Try doing this in your interactive window in Thonny, but before you start typing, open the “Variables” window (View -> Variables).

When you type in the text and press enter you should notice two things. Firstly, nothing is displayed on the line after your input text. This is because what you have entered is a \texttt{statement}, which unlike expressions in the previous section, do not return values. What you should also notice is that the names appear in the \textbf{Variables window}, along with their associated values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>'MyString'</td>
</tr>
</tbody>
</table>

If you now type \texttt{a} or \texttt{b} in the interactive window, you will see the value of that name is returned:

```python
>>> b
'MyString'
```

It is possible to assign a name to an existing name

```python
>>> c = b
>>> c
'MyString'
```

It is also possible to assign multiple names to the same value in a single statement:

```python
>>> d = e = f = 'ManyToOne'
>>> e
'ManyToOne'
```
'ManyToOne'

Another useful feature of Python is that it is possible to assign a name to value as it is being calculated:

```python
>>> g = 10 * 5
>>> g
50
```

This is an important concept in programming languages, for it now means that **names may be manipulated as one would manipulate the values** to which they are assigned. For example:

```python
>>> a = 1
>>> b = 2
>>> c = a + b
>>> c
3
```

In this example, the value of `c` is deduced to be 3, since it is the sum of `a` and `b` (which have been assigned the values 1 and 2 respectively). Also note that you do not have to put quotation marks around the names since they are not strings, although of course they may represent strings.

**F-strings**

Now that we have introduced the concept of names, this marks a good point to discuss f-strings. F-strings provide a simple yet powerful way to format strings, making life easier for those reading the text or numbers displayed on the screen.

The syntax for f-string formatting requires a lowercase $f$ or an uppercase $F$, followed immediately by some text or expression encapsulated by quotation marks. For example:

```
f"Hi {user}"
```

The `$f$ before the quotation marks denotes that we are generating an f-string. The text in the quotation marks forms the literal part of the f-string, while anything in the curly brackets is the expression part of the string. The expression component may be thought of as a placeholder for what will appear when the code executes. Let's elaborate on the example to illustrate this:

```python
>>> user = 'Octavian'
>>> f"Hi {user}"
'Hi Octavian'
```

The string 'Octavian' is assigned to the name `user`, which is rendered in the f-string.

It is not simply names that may be rendered between these curly brackets, since they are essentially expressions. Calculations for example may be performed here:

```python
>>> F'Two plus two is {2 + 2}'
'Two plus two is 4'
```
Similarly,

```python
>>> f'I have £{1000 * 1000}!'
'I have £1000000!'
```

That's nice, but it would be even nicer if we could format these large numbers. Well, with f-strings you can! To add commas to separate the thousands, add a colon and a comma:

```python
>>> f'I have £{1000 * 1000:,}!'
'I have £1,000,000!'
```

You could also add the pennies by specifying the number of decimal places to display with `.2f`. The `.2f` means “two decimal places, fixed” (never any more or less than two decimal places).

```python
>>> f'I have £{1000 * 1000:,.2f}!'
'I have £1,000,000.00!'
```

In a similar fashion, f-strings allow you to manipulate percentages. In the example below, the `.1%` is an instruction to convert the value to a percentage and display one decimal place.

```python
>>> vat = 0.2
>>> f'VAT rate: {vat:.1%}'
'VAT rate: 20.0%'
```

It is also possible to control the width and alignment of f-strings. If you look at the f-string below, you will see the number after the colon fixes the width of the string, while the characters `< > ^` denote left-, right- and centre-aligned formatting, respectively.

```python
>>> pointer = '<->'
>>> f'Left-aligned{pointer:<50}'
'Left-aligned<->
>>> f'Right-aligned{pointer:>50}'
'Right-aligned <->
>>> f'Centre-aligned{pointer:^50}'
'Centre-aligned <->
```

F-strings may also encompass multiple lines, as per regular strings.

**Functions**

**Built-in functions**

Previously in this manual we ran a line of code that printed “Hello World!” to the screen:

```python
print("Hello World!"))
```
This is an example of a function. A function comprises several components: firstly, every function has a name, which in this case is `print`. Following the function name comes a pair of brackets. There may be nothing between these brackets, or alternatively there may be a list of one or more items termed arguments. In this example, the argument passed to the `print` function is the “Hello World!” string. If there is more than one argument, they should be separated from one another using commas. Every function returns a value and so consequently a function is a type of expression. More specifically, a function is a type of expression known as a call (another type of call is a method, which we shall meet later).

Notice what happens when printing a name using this function:

```python
>>> forPrinting = 'Print Me!

>>> print(forPrinting)
Print Me!
```

The string ‘Print Me!’ is assigned to the name ‘forPrinting’. Calling the function print with the argument ‘forPrinting’ causes the value associated with that name to be printed.

The print function is one of Python's built-in functions. There many other built-in functions:

- `len(arg)` – returns the length characters in a string:
  ```python
  >>> len('Supercalifragilisticexpialidocious')
  34
  ```

- `min(arg...)` – returns the minimum value of a list of numerical arguments:
  ```python
  >>> min(2, 4, 100, 205.3, -4)
  -4
  ```

- `max(arg...)` – returns the maximum value of a list of numerical arguments:
  ```python
  >>> max(2, 4, 100, 205.3, -4)
  205.3
  ```

The table below shows the full list of Python built-in functions. Another built-in function that is worth knowing about at this stage is `help()`. If you type this function on the command line, an interactive help dialog box will start. Alternatively, you can pass to the help function the name of a function or value as an argument, and that will result in information on that argument being printed to the screen.
Functions and data types
We have discussed already that there are different datatypes in Python (i.e. int, float, str,...). Up until now we have not been able to ascertain directly the data type of a given object. However, this is possible using the `type()` function.

```python
>>> type(378163771)
<class 'int'>

>>> type('Hello World!')
<class 'str'>

>>> type('378163771')
<class 'str'>
```

That may be all well and good, but should we really care how Python is storing these values? Well the answer is yes. Suppose we have the integer 378163771 stored as a string. Any numerical calculations we want to do with this will then fail. To get around this we would need to convert a string to a number and then perform our numerical operations. This process of converting one data type to another is called **casting**.

```python
>>> int('378163771') - 1
378163770
```

In this example we have converted a string to an integer, but we could have course converted to a float.

Casting in python is therefore done using constructor functions:
int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)

float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Creating Functions
In addition to using Python’s built-in functions, it is possible to build your own functions. In fact, the ability to write your own functions is highly advised for all but the most basic programs.

The function definition statement has the following general structure:

def function_name (parameters list):
    function code...
    ...
    ...

The def command in Python means you are defining a new function. You then specify in brackets the parameters that the function takes, although some functions may take no parameters. There is then a colon followed by the code that constitutes the working part of the function on the adjacent lines immediately beneath the function definition. (Just to clarify, a parameter is a name in a function definition. When a function is called, the arguments are the data you pass into the function’s parameters.)

Indentation
You will also notice that the function code is indented as compared to the line above. The indentation is a key concept in Python. Indenting code in this way tells the Python interpreter that this indented code is part of the same function. When indenting code, use 4 spaces for each indentation (do not use tabs*). We shall revisit this concept of creating code blocks to scope your code again and again in this course.

*Some text editors automatically convert a tab into 4 spaces to assist with writing Python.

We shall now learn more about functions using a series of different examples.

Example 1
def minimalist():
    pass

minimalist()

The code defines a function called “minimalist”. It is passed no arguments and when run, returns nothing. The command pass is an instruction for the function to end and return nothing. Although this function is far from useful in a practical sense, it illustrates the minimal requirements of a function.
You will notice there is a line below the function, using the left-most indentation once again, on which is entered “minimalist ()”. This is how functions are called i.e. the function name followed by a pair of brackets, just like built-in functions. Should a function take arguments, then these arguments will be entered between the brackets.

**Example 2**

def HelloWorld():
    print ("Hello World!")

HelloWorld()

>>> %Run functions.py
Hello World!

This second example takes no arguments; however the function does a little more by printing “Hello World!” to the screen once the function is called.

**Example 3 – passing arguments**

def HelloUser(name):
    print ("Hello " + name)

HelloUser("Bob")

>>> %Run functions.py
Hello Bob

This third example is a slight modification of the previous function. Here the function also prints to the screen, but rather than simply printing a fixed message the function prints the argument it receives to the screen. So, when the function is called and the string “Bob” is passed as an argument, the function subsequently prints out “Hello Bob”.

**Example 4 – returning values**

def percentageCalculator(value, total):
    percentage = (100 * value / total)
    return(percentage)

score = percentageCalculator(9, 18)
print(score)

>>> %Run functions.py
50.0

In this example we define a percentage calculator. The function is intended to take two arguments, both numbers. On running, the function calculates the percentage value of the first number as compared to the second number. The function is then called and calculates what percentage 9 is of 18. Rather than printing to the screen, the function returns the percentage value, which is then assigned to the name “score”. The value of score is then printed to the screen.
Example 5 – documenting functions
# We could write a comment here describing
# our function, but it is more useful to
# enter this information in the docstring
def percentageCalculator(value, total):
    """Takes a value and a total and
    returns the percentage value"
    return(100 * value / total)

help(percentageCalculator)

>>> %Run functions.py
Help on function percentageCalculator in module __main__:

percentageCalculator(value, total)
    Takes a value and a total and
    returns the percentage value

Documenting code is good practice and will save you from all sorts of future problems, especially with larger projects. Documentation entails interleaving your code with descriptions written in intelligible English (or your language of choice). Try to give as much detail as possible, providing an overview of what the code does, the purpose of each function and clarify points in the code which may not be obvious as to their intent. Doing this will not only help other people who may try to run or build upon your code, but more often than not it will be your future self who is the main beneficiary as you try to decipher what you were trying to achieve many months ago.

A simple way to document code, which is common in many languages, is to incorporate comments prefixed with a hash symbol. Python, being a very stylish language, has a better way to document functions. Immediately after declaring a function, write a comment between a pair of triple speech marks. Writing comments this way will allow users to read more about a particular function when running the help function.

(You may have noticed a modification in the above example, as compared to the previous function. Here, a percentage object was not created in the function itself and the result of the “100 * value / total” is returned directly. Either option is valid. As you learn to write code you will often have to decide what is the appropriate trade-off between brevity and clarity.)
Example 6 – Default parameter values

def calculateVAT(initial, rate=20):
    """Calculates the cost of an item after Value Added Tax. The function takes an initial value (integer/float) as input and an optional VAT percentage rate (integer/float), else defaults to 20%."""
    vat = initial / 100 * rate
    new_cost = initial + vat
    return(new_cost)

print(calculateVAT(500, 10))
print(calculateVAT(500))

This time the function is a simple VAT calculator to determine the amount of tax to add to a given value. As mentioned in the documentation, the first argument passed to the function is the value to which VAT should be added, while the second argument is the VAT percentage rate. The function is then called twice, with the result printed directly to the screen:

>>> %Run functions.py
550.0
600.0

In the first example a value of 550 is returned, which makes sense since adding 10% to 500 will make 550. However, for the second call, 600 is printed to the screen, but no rate parameter was specified. How was the calculation made without this necessary piece of information? The answer lies in the function itself; in the first line of the function you will see rate parameter is assigned the value 20. This tells the function that if no argument is specified for this parameter, then the rate should default to 20. Reassuringly, adding 20% to 500 does make indeed 600.

Example 7 – assertions

def HelloUser(name):
    """Says hello to the named user. Takes a string as input."""
    assert isinstance(name, str), 'The input needs to be a string'
    print ("Hello " + name)

HelloUser(1)

>>> %Run functions.py
Traceback (most recent call last):
  File "/Users/wingetts/functions.py", line 25, in <module>
    HelloUser(1)
  File "/Users/wingetts/functions.py", line 22, in HelloUser
assert isinstance(name, str), 'The input needs to be a string'

AssertionError: The input needs to be a string

So far, we have passed the expected arguments to a function. But you may have already wondered as to what would happen if we pass something incorrect to the function. Well, the function may simply fail, causing the program to terminate. This is not ideal, but not as bad as if the function is passed something that carries on working but results in the wrong value being returned by the function. While we cannot always check for every potential problem, we can check that arguments passed to a function meet pre-specified criteria by using assertions.

After encountering an assertion, the Python interpreter will check whether the following statement is true. For example:

```
assert 0==1
```

Will evaluate to False, resulting in an error message. These error message will begin with the word “Traceback” and provide further details of why the program failed, including the line number of the code that returned the error. It is also possible to add text after the assertion statement that will be included in the traceback message:

```
assert 0==1, 'Zero is not the same as one!
```

In Example 7, the assertion is followed by the built-in function `isinstance()`. This checks the datatype of a value. In this case, for the function to evaluate to true, the “name” variable will have to be a string (str). However, we pass the integer 1 to the new `HelloUser` function, the assertion will fail resulting in the traceback error message shown. By doing this, we have forced the `HelloUser` function to only accept string input.

**Example 8 – passing multiple arguments to a function**

It is possible to pass multiple arguments to a function, simply use the syntax:

```
def function_name(*args):
```

This will create a tuple (you will encounter this datatype later in the course, but for completeness it makes sense to include this example in this section) named `args` containing the passed arguments, as demonstrated in the example below. Note that in the example the script passes to the function four integers as well as a list (another datatype discussed later). This generates a tuple comprising five elements: one for each integer and a final element containing the whole list.

```
def accept_multiple_arguments(*args):
    print(type(args))
    print(args)

my_list =['a', 'b', 'c']
accept_multiple_arguments(1,2,3,4, my_list)

>>> 
<class 'tuple'>
(1, 2, 3, 4, ['a', 'b', 'c'])
```
Methods

Methods are very similar to functions. Specific data types in Python have the capability to run different methods. To call the method, simply place a dot after the name of the datatype and then enter the method name. Similar to functions, methods may also take arguments. For example:

```python
>>> 'abracadabra'.count('a')
5

>>> magic = 'abracadabra'
>>> magic.count('a')
5
```

In the example above, we have used the `count` method of the string datatype to count the number of times the character 'a' occurs in the word abracadabra. In the first example the method is run directly on the string; in the second the method is run on the name assigned to 'abracadabra'. It is also possible to "chain" together multiple methods:

```python
>>> magic = 'abracadabra'
>>> magic.upper().count('A')
5
```

As for functions, Python has a large number of built-in methods for its datatypes. Listed below are the string methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>capitalize()</td>
<td>Converts the first character to upper case</td>
</tr>
<tr>
<td>casefold()</td>
<td>Converts string into lower case</td>
</tr>
<tr>
<td>center()</td>
<td>Returns a centred string</td>
</tr>
<tr>
<td>count()</td>
<td>Returns the number of times a specified value occurs in a string</td>
</tr>
<tr>
<td>encode()</td>
<td>Returns an encoded version of the string</td>
</tr>
<tr>
<td>endswith()</td>
<td>Returns true if the string ends with the specified value</td>
</tr>
<tr>
<td>expandtabs()</td>
<td>Sets the tab size of the string</td>
</tr>
<tr>
<td>find()</td>
<td>Searches the string for a specified value and returns the position of where it was found</td>
</tr>
<tr>
<td>format()</td>
<td>Formats specified values in a string</td>
</tr>
<tr>
<td>format_map()</td>
<td>Formats specified values in a string</td>
</tr>
<tr>
<td>index()</td>
<td>Searches the string for a specified value and returns the position of where it was found</td>
</tr>
<tr>
<td>isalnum()</td>
<td>Returns True if all characters in the string are alphanumeric</td>
</tr>
<tr>
<td>isalpha()</td>
<td>Returns True if all characters in the string are in the alphabet</td>
</tr>
<tr>
<td>isdecimal()</td>
<td>Returns True if all characters in the string are decimals</td>
</tr>
<tr>
<td>isdigit()</td>
<td>Returns True if all characters in the string are digits</td>
</tr>
<tr>
<td>isidentifier()</td>
<td>Returns True if the string is an identifier</td>
</tr>
<tr>
<td>islower()</td>
<td>Returns True if all characters in the string are lower case</td>
</tr>
<tr>
<td>Method</td>
<td>Purpose</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isnumeric()</td>
<td>Returns True if all characters in the string are numeric</td>
</tr>
<tr>
<td>isprintable()</td>
<td>Returns True if all characters in the string are printable</td>
</tr>
<tr>
<td>isspace()</td>
<td>Returns True if all characters in the string are whitespaces</td>
</tr>
<tr>
<td>istitle()</td>
<td>Returns True if the string follows the rules of a title</td>
</tr>
<tr>
<td>isupper()</td>
<td>Returns True if all characters in the string are upper case</td>
</tr>
<tr>
<td>join()</td>
<td>Joins the elements of an iterable to the end of the string</td>
</tr>
<tr>
<td>ljust()</td>
<td>Returns a left justified version of the string</td>
</tr>
<tr>
<td>lower()</td>
<td>Converts a string into lower case</td>
</tr>
<tr>
<td>lstrip()</td>
<td>Returns a left trim version of the string</td>
</tr>
<tr>
<td>maketrans()</td>
<td>Returns a translation table to be used in translations</td>
</tr>
<tr>
<td>partition()</td>
<td>Returns a tuple where the string is parted into three parts</td>
</tr>
<tr>
<td>replace()</td>
<td>Returns a string where a specified value is replaced with a specified value</td>
</tr>
<tr>
<td>rfind()</td>
<td>Searches the string for a specified value and returns the last position of where it was found</td>
</tr>
<tr>
<td>rindex()</td>
<td>Searches the string for a specified value and returns the last position of where it was found</td>
</tr>
<tr>
<td>rjust()</td>
<td>Returns a right justified version of the string</td>
</tr>
<tr>
<td>rpartition()</td>
<td>Returns a tuple where the string is parted into three parts</td>
</tr>
<tr>
<td>rsplit()</td>
<td>Splits the string at the specified separator, and returns a list</td>
</tr>
<tr>
<td>rstrip()</td>
<td>Returns a right trim version of the string</td>
</tr>
<tr>
<td>split()</td>
<td>Splits the string at the specified separator, and returns a list</td>
</tr>
<tr>
<td>splitlines()</td>
<td>Splits the string at line breaks and returns a list</td>
</tr>
<tr>
<td>startswith()</td>
<td>Returns true if the string starts with the specified value</td>
</tr>
<tr>
<td>strip()</td>
<td>Returns a trimmed version of the string</td>
</tr>
<tr>
<td>swapcase()</td>
<td>Swaps cases, lower case becomes upper case and vice versa</td>
</tr>
<tr>
<td>title()</td>
<td>Converts the first character of each word to upper case</td>
</tr>
<tr>
<td>translate()</td>
<td>Returns a translated string</td>
</tr>
<tr>
<td>upper()</td>
<td>Converts a string into upper case</td>
</tr>
<tr>
<td>zfill()</td>
<td>Fills the string with a specified number of 0 values at the beginning</td>
</tr>
</tbody>
</table>
Chapter 4 – Collections

We previously discussed the primitive data types found in Python which are useful in a wide number of situations. These simple data types, however, are not adequate for the more complex tasks performed by software. This section describes the compound data types that contain multiple objects in structures called collections or containers. Each object in a collection is referred to as an element or item. There are three groups of collections in Python, namely sets, sequences and mappings. Becoming familiar with their usage will enable you to dramatically increase the range of tasks to which you can code solutions.

Sets

A set is a collection of unordered unique elements. Sets can be created in a couple of ways. Firstly, a string can be passed to a keyword to create a set:

>>> set('ABBA')
{'B', 'A'}

You will see that passing the string ‘ABBA’ results in a set containing only ‘A’ and ‘B”, i.e. the string has been deduplicated at the character level. Also, don’t make any assumptions about the order in which the individual characters are returned since they need not correspond to the ordered in which they were entered.

Another way to create a set is to use curly brackets:

>>> beatles = {'John', 'Paul', 'George', 'Pete'}

>>> beatles
{'Paul', 'John', 'Pete', 'George'}

Here we have assigned the set to the name beatles. Notice that this time the input is not broken down to individual characters, but instead this method of declaration preserves to the individual strings between the speech marks.

It is possible to add elements to existing sets:

>>> beatles.add('Ringo')

>>> beatles
{'Ringo', 'Pete', 'John', 'Paul', 'George'}

And remove items:

>>> beatles.remove('Pete')

>>> beatles
{'Ringo', 'John', 'Paul', 'George'}

Using remove to delete an entry not found in a set will generate an error:
>>> beatles.remove('Stuart')
Traceback (most recent call last):
  File "<pyshell>" , line 1, in <module>
    KeyError: 'Stuart'

However, using **discard** will not:

>>> beatles.discard('Stuart')

Choose **remove** or **discard** as appropriate. In some situations you may want to be informed if an item expected in a set turns out not to be present, since this suggests some kind of logical error in your code. Conversely, in other situations you may not be able to predict the elements of a set, and so no error should be returned if trying to delete a non-existent element.

Python sets can be combined or compared in ways that may be familiar to people who have worked with sets in mathematics. For example, when considering 2 sets:

```python
>>> beatles = {'John', 'Paul', 'George', 'Ringo'}
>>> wilburys = {'Bob', 'George', 'Jeff', 'Roy', 'Tom'}
```

The following set calculations can be performed, either by i) using an operator; or ii) by using a method call:

**Union (identify elements found in either set)**

```python
>>> beatles | wilburys
{'Ringo', 'Roy', 'Bob', 'John', 'Jeff', 'Paul', 'Tom', 'George'}
```

```python
>>> beatles.union(wilburys)
{'Ringo', 'Roy', 'Bob', 'John', 'Jeff', 'Paul', 'Tom', 'George'}
```

**Intersection (identify elements common to both sets)**

```python
>>> beatles & wilburys
{'George'}
```

```python
>>> beatles.intersection(wilburys)
{'George'}
```

**Difference (identify items present in set one, but not set two)**

```python
>>> beatles - wilburys
{'Paul', 'John', 'Ringo'}
```

```python
>>> beatles.difference(wilburys)
{'Paul', 'John', 'Ringo'}
```
Symmetric Difference (items present that are not common to both sets)

```python
>>> beatles ^ wilburys
{'Paul', 'Roy', 'Tom', 'Bob', 'John', 'Ringo', 'Jeff'}
```

```python
>>> beatles.symmetric_difference(wilburys)
{'Paul', 'Roy', 'Tom', 'Bob', 'John', 'Ringo', 'Jeff'}
```

Also, please note that to create an empty set you may only use the syntax using the keyword set (i.e. you cannot simply use the curly brackets):

```python
empty = set()
```

The table below lists the Set Methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add()</td>
<td>Adds an element to the set</td>
</tr>
<tr>
<td>clear()</td>
<td>Removes all the elements from the set</td>
</tr>
<tr>
<td>copy()</td>
<td>Returns a copy of the set</td>
</tr>
<tr>
<td>difference()</td>
<td>Returns a set containing the difference between two or more sets</td>
</tr>
<tr>
<td>difference_update()</td>
<td>Removes the items in this set that are also included in another, specified set</td>
</tr>
<tr>
<td>discard()</td>
<td>Remove the specified item</td>
</tr>
<tr>
<td>intersection()</td>
<td>Returns a set, that is the intersection of two other sets</td>
</tr>
<tr>
<td>intersection_update()</td>
<td>Removes the items in this set that are not present in other, specified set(s)</td>
</tr>
<tr>
<td>isdisjoint()</td>
<td>Returns whether two sets have a intersection or not</td>
</tr>
<tr>
<td>issubset()</td>
<td>Returns whether another set contains this set or not</td>
</tr>
<tr>
<td>issuperset()</td>
<td>Returns whether this set contains another set or not</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes an element from the set</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes the specified element</td>
</tr>
<tr>
<td>symmetric_difference()</td>
<td>Returns a set with the symmetric differences of two sets</td>
</tr>
<tr>
<td>symmetric_difference_update()</td>
<td>Inserts the symmetric differences from this set and another</td>
</tr>
<tr>
<td>union()</td>
<td>Return a set containing the union of sets</td>
</tr>
<tr>
<td>update()</td>
<td>Update the set with the union of this set and others</td>
</tr>
</tbody>
</table>

For completeness, please be aware that there is also a collection known as a frozenset. This datatype is created in a similar way to a set, only using the name `frozenset`. Unlike `set`, the values added to a `frozenset` may not be altered after creation. There are also fewer methods available to a `frozenset` than the more commonly encountered `set`.

**Sequences**

Sequences in Python are an ordered list of elements and, unlike sets, may contain duplicate elements.
Ranges
Ranges contain an ordered list of **integers**. You may create a range in a number of ways, of which the simplest is to specify the **stop** value. This will create a range from 0 to, but not including, that stop value. If you try in the Thonny interactive window you should see:

```python
>>> range(3)
range(0, 3)
```

The returned value will show that a range has been created and also **display the start and stop values**. If we pass this range to a set, you shall see that the values of the range are 0, 1, and 2 but not 3:

```python
>>> set(range(3))
{0, 1, 2}
```

If you wish the range so start at a value other than 0, then simply add this before the stop value:

```python
>>> set(range(100, 111))
{100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110}
```

You may also pass a third argument when creating a range, namely a **step** value which sets the number by which the range should be incremented until it reaches the stop value

```python
>>> set(range(2, 10, 2))
{8, 2, 4, 6}
```

If you choose an end value equal or less than the start value then the range will but empty, but if you choose a negative step value, then stop will need to be less than then start:

```python
>>> set(range(3, -3, -1))
{0, 1, 2, 3, -2, -1}
```

Passing a range to the **set** command introduces an important new concept: **Python data structures may be generated from pre-existing data structures**. Using this technique is often an efficient way to generate the desired collection. For example, by creating a set from a range is quick way to generate a **set** with a large number of elements (you wouldn’t want to type out every number in a set, now would you?).

**Tuples**
Tuples are sequences that contain **any type of element** and **may contain duplicates**. Tuples are ordered and **immutable** (meaning that when they have been created they cannot be adjusted). To create a tuple, simply specify a comma-separated list of the elements of the tuple:

```python
>>> (1, 2, 3)
(1, 2, 3)
```
When assigning a comma separated list to a name, omitting the round brackets will still lead to a tuple being created. Creating a tuple from such a list is known as **tuple packing**.

```python
>>> turtles = 'Leonardo', 'Michelangelo', 'Donatello', 'Raphael'
>>> turtles
('Leonardo', 'Michelangelo', 'Donatello', 'Raphael')
```

However, for more complex tuples which may contain other data structures rather than strings or number, the brackets will need to be specified. In this manual we include round brackets when creating tuples.

You can declare an empty tuple simply with placing nothing between the brackets:

```python
>>> nothing = ()
>>> nothing
()
```

But you will need to **add a trailing comma to a one-element tuple**:

```python
>>> alone = ('Crusoe', )
>>> alone
('Crusoe',)
```

This avoids ambiguity between tuples and mathematical operations. For example:

```python
>>> ambiguous = (10 * 10)
```

Are we creating an integer or a tuple? Well, we are actually creating an integer of value 100. Please contrast with the line of code below:

```python
>>> not_ambiguous = (10 * 10,)
```

Now we are definitely making a tuple!

On a related note, it is possible to assign multiple values to multiple names (tuple unpacking) in a single expression using comma-separated lists:

```python
>>> batman, robin = 'Bruce', 'Dick'
>>> batman
'Bruce'
>>> robin
'Dick'
```

Here, the first element in the right-hand side comma-separated list is assigned the value of the first value in the left-hand side of the comma-separated list. This assignment carries on for all elements of the two lists either side of the assignment operator (=), so long as the two lists are of equal length. This
trick can help you write more succinct code. One such instance where tuple unpacking makes for more simple code is when transposing values:

```python
>>> b, a = a, b
```

As you may have already predicted, you may also create tuples using the tuple function:

```python
>>> tuple('Leonardo')
('L', 'e', 'o', 'n', 'a', 'r', 'd', 'o')
```

In a similar fashion to sets, using the function causes the individual characters to be considered separate elements of the tuple (in contrast to above, where ‘Leonardo’ would form a single element of a tuple).

You may reference a specific element in a tuple using the syntax used for indexing strings:

```python
>>> turtles[0]
'Leonardo'
```

Similar to strings, tuples have built-in methods, but whereas string have a multiplicity of methods, tuples have only 2.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Returns the number of times a specified value occurs in a tuple</td>
</tr>
<tr>
<td>index()</td>
<td>Searches the tuple for a specified value and returns the position of where it was found</td>
</tr>
</tbody>
</table>

**Tuples and functions**

In a previous section we discussed in detail how to pass multiple arguments to a given function. What we did not discuss however, was how to retrieve multiple values from a function. A common way to do this is to use a tuple, as shown in the example below:

```python
def currency_convert(pounds):
    euro = pounds * 0.9
    dollars = pounds * 0.8
    return(euro, dollars)

my_money = 10
conversions = currency_convert(my_money)
print(f'I have €{conversions[0]} or ${conversions[1]}')
```

```python
>>> I have €9.0 or $8.0
```

Now multiple values have been returned from the `currency_convert` function, by virtue of being contained within a single tuple.
Lists
Lists are in many ways similar to tuples, for they constitute a sequence of any type of element. Unlike tuples, however, lists are mutable. Lists can be created in a similar fashion to tuples, only using square brackets instead of round brackets. The code below illustrates this point:

```python
>>> a = (1, 2, 3)
>>> type(a)
<class 'tuple'>

>>> b = [1, 2, 3]
>>> type(b)
<class 'list'>

>>> b
[1, 2, 3]
```

Since lists may be changed after they have been created, there are assignment expressions or methods that may be used to modify a list. In fact there are actually a wide of ways to modify a list, a small subset of these possible options is shown below.

Replacing an element in a list:
```python
>>> beatles = ['John', 'Paul', 'George', 'Pete']
>>> beatles[3] = 'Ringo'
>>> beatles
['John', 'Paul', 'George', 'Ringo']
```

Here the element at position 3 (the fourth value in the list, using the now familiar index notation) is changed from Pete to Ringo. Alternatively, it is possible to replace multiple elements in a list with multiple elements from another type of collection.

Replacing elements in a list using another collection:
```python
>>> temps = ('Jimmy', 'Eric')
>>> beatles[3:4] = temps
>>> beatles
['John', 'Paul', 'George', 'Jimmy', 'Eric']
```

Here we have replaced Ringo in the list `beatles` with elements from the tuple `temps`.

As mentioned before, there are many related ways lists may be modified. Looking at the Python documentation or detailed cheat sheet is a good way to get an overview of these. We shall not discuss all these ways in detail, since they are largely variations on the same idea. We shall, however, allow you to become familiar with these ideas by trying out several examples in the exercises.

As for other types of collections, we can modify lists using methods. For example, to reverse the list use the method ‘reverse’:
Using methods on lists:
Lists may be modified using a method call.

```python
>>> beatles.reverse()
>>> beatles
['Eric', 'Jimmy', 'George', 'Paul', 'John']
```

The developers of Python wisely chose intuitive names for the built-in methods:

```python
>>> beatles.sort()
>>> beatles
['Eric', 'George', 'Jimmy', 'John', 'Paul']
```

Another important point that you may have noticed is that the `sort()` method has not returned a value, instead the original `beatles` list was sorted directly. The list does not have to be re-assigned to itself i.e. we do not do the following:

```python
beatles = beatles.sort()
```

You should bear this in mind in the future: **some methods return values, while others manipulate “their” object directly.**

The table below displays the `list` methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append()</code></td>
<td>Adds an element at the end of the list</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all the elements from the list</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td>Returns a copy of the list</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Returns the number of elements with the specified value</td>
</tr>
<tr>
<td><code>extend()</code></td>
<td>Add the elements of a list (or any iterable), to the end of the current list</td>
</tr>
<tr>
<td><code>index()</code></td>
<td>Returns the index of the first element with the specified value</td>
</tr>
<tr>
<td><code>insert()</code></td>
<td>Adds an element at the specified position</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>Removes the element at the specified position</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>Removes the item with the specified value</td>
</tr>
<tr>
<td><code>reverse()</code></td>
<td>Reverses the order of the list</td>
</tr>
<tr>
<td><code>sort()</code></td>
<td>Sorts the list</td>
</tr>
</tbody>
</table>

**Dictionaries**

Dictionaries (dict) are **mutable** structures that store data in what is known as **“key/value” pairs**. The name is of this datatype is quite intuitive name: the key serves as the dictionary word to look-up, and the value serves as the definition of that word. As for dictionaries, the definitions (keys) must be **unique**. Importantly, each value (definition) may only hold one object. Unlike real dictionaries however, the dict keys are actually unordered.
A dictionary is actually a type of computing datatype known as a **mapping**. Unlike some other languages, dictionaries are the only mapping in Python. Mapping datatype encountered in Python is known as a **dictionary** *(dict)*. To declare a dictionary, use the following syntax:

```python
>>> a_team = {'Hannibal': 'Lieutenant Colonel John Hannibal Smith', 'Face': 'Lieutenant Templeton Arthur Peck', 'BA': 'Sergeant Bosco Albert Baracus', 'Murdock': 'Captain H.M. Murdock')
```

This creates a dictionary named *a_team* which holds 4 key/value pairs, referring to an individual’s nickname / real name. For example: the key ‘Hannibal’ is paired with the real name value ‘Lieutenant Colonel John Hannibal Smith’.

You may have noticed that declaring a dictionary using curly brackets is similar to creating sets, only when making dictionaries there are colons to separate the keys and elements. You may wonder, therefore, what happens if we use this naming system and place nothing between the brackets: do we create a set or a dictionary? Well, we actually create a dictionary:

```python
>>> type({})
<class 'dict'>
```

This form of declaration uses colons to separate the key/value pairs, while different entries are separated from one another by commas.

As for other datatypes, there is an alternative way to create dictionaries:

```python
>>> a_team = dict((('Hannibal', 'Lieutenant Colonel John Hannibal Smith'), ('Face', 'Lieutenant Templeton Arthur Peck'), ('BA', 'Sergeant Bosco Albert Baracus'), ('Murdock', 'Captain H.M. Murdock')
``` }

Values maybe retrieved entering the dictionary name and then entering the sought key between square brackets:

```python
>>> a_team['Face']
'Lieutenant Templeton Arthur Peck'
```

Entering a key that does not exist will result in an error:

```python
>>> a_team['Jaffo']
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
KeyError: 'Jaffo'
```

You may add a new entry to a dictionary as follows:

```python
>>> a_team['Amy'] = 'Amy Allen'
```
>>> a_team
{'Hannibal': 'Lieutenant Colonel John Hannibal Smith', 'Face': 'Lieutenant Templeton Arthur Peck', 'BA': 'Sergeant Bosco Albert Baracus', 'Murdock': 'Captain H.M. Murdock', 'Amy': 'Amy Allen'}

The same method is also used to edit an existing value:

a_team['Murdock'] = 'Captain Murdock'

To delete an entry from the dictionary, use the `del` command:

```python
>>> del a_team['BA']
>>> a_team
{'Hannibal': 'Lieutenant Colonel John Hannibal Smith', 'Face': 'Lieutenant Templeton Arthur Peck', 'Murdock': 'Captain Murdock', 'Amy': 'Amy Allen'}
```

Once again, as for other datatypes, it is possible to modify dictionaries using built-in object methods.

```python
>>> a_team.get('Face')
'Lieutenant Templeton Arthur Peck'
```

Be aware that using this method on a non-existent key returns the `None` type, rather than an error message.

Look at the table below of Python built-in dictionary methods to appreciate how these data structures may be manipulated.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>Removes all the elements from the dictionary</td>
</tr>
<tr>
<td>copy()</td>
<td>Returns a copy of the dictionary</td>
</tr>
<tr>
<td>fromkeys()</td>
<td>Returns a dictionary with the specified keys and values</td>
</tr>
<tr>
<td>get()</td>
<td>Returns the value of the specified key</td>
</tr>
<tr>
<td>items()</td>
<td>Returns a list containing a tuple for each key value pair</td>
</tr>
<tr>
<td>keys()</td>
<td>Returns a list containing the dictionary's keys</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes the element with the specified key</td>
</tr>
<tr>
<td>popitem()</td>
<td>Removes the last inserted key-value pair</td>
</tr>
<tr>
<td>setdefault()</td>
<td>Returns the value of the specified key. If the key does not exist: insert the key, with the specified value</td>
</tr>
<tr>
<td>update()</td>
<td>Updates the dictionary with the specified key-value pairs</td>
</tr>
<tr>
<td>values()</td>
<td>Returns a list of all the values in the dictionary</td>
</tr>
</tbody>
</table>

(As a brief aside, a dictionary key does not have to be a string, integer or float (which are actually immutable), since tuples may be used as dictionary keys as well (so long as all its elements are immutable). Whether an object may be allowed to be used as a dictionary key is known as hashability.)
Streams

Obviously, the term stream does not refer to the flow of water, but instead refers to the flow of data. Despite this, comparing data processing to the flow of water is actually a good analogy. In both examples the flowing ‘material’ moves from a source to a sink. In a house water flows from a source (a tap) to a sink (the, erm, sink). In data terms, the source could be a file that is opened and read, a network connection or a specialised Python data structure termed a generator.

Files

To read a file, you will need to write code to create a Python object representing the file to be processed. The two lines of code below will open a file named ‘Poetry.txt’ and then write the output to the screen.

```python
with open('/Users/wingetts/Desktop/Poetry.txt', 'r') as anthology:
    print(anthology.readlines())
```

The command `open` serves as an instruction to open a specified file. Where necessary the path to the file should be specified i.e. the location of the file on the computer running the script. In this example, the file Poetry.txt sits on the Desktop, which in turn is found in the wingetts folder, and so on. After the path to the file is the letter `r`. This is an instruction to read the file (in contrast would be an instruction to write to a named file). By default, this command is expecting the file of interest to contain text. If this were not the case, and the file contained binary data, then the following open function should be passed `rb`. The `with` function is not necessary, but is desirable since its inclusion ensures the file is closed in the event of an error (it is bad practice to leave files open unnecessarily and can potentially lead to problems.) The `as` `anthology` tells Python to create an object named `anthology` to represent this file.

On the subsequent line the method `readlines()` is called on the `anthology` object, which causes all the text in the file to be read. This is then printed to the screen with the `print` command.

There are many different method calls that can be run on a file object, depending on what is required. For example, `anthology.readline(10)` would return the first 10 characters of the first lines of the file.

Conversely, if we were writing to the file, we could pass the `writelines()` method a collection in which every element is a string.

Generators

A generator is a Python object that creates, or indeed generates, a stream of values. The generator can keep making new values de novo as many times as is required. (In many situations this is preferable to an alternative strategy of keeping a very large list in memory, and then selecting a value from the list as necessary.) We shall discuss generators in more detail later in the Advanced Python Course.

Copying collections

Up until now, when dealing with primitive data types, we have seen that the assignment operator can be used to make an independent copy of a name. If you look the example below, `string1` and `string2` have final vales of ‘A’ and ‘B’ respectively. The name `string1` always had the value ‘A’, whereas `string2` was initialised with the same value as `string1` (i.e. ‘A’), but this was subsequently updated to ’B’.

```python
string1 = 'A'
```
string2 = string1
string2 = 'B'

print(string1)
print(string2)

>>> A
B

This is all well and good, but the situation is different for collections:

list1 = ['A', 'B', 'C']
list2 = list1

print(list1)
print(list2)
print()

list2.append('D')

print(list1)
print(list2)

>>> ['A', 'B', 'C']
['A', 'B', 'C']

[['A', 'B', 'C', 'D'],
['A', 'B', 'C', 'D']

Unlike for primitive datatypes, modifying list2 will also modify list1. This is because we have actually created 2 different names referencing the same data structure. Changing list1 will therefore modify list2, and vice versa. You may want this functionality in your code, or you may not. If you don’t, the way around this is to use the copy() method of a collection:

list1 = ['A', 'B', 'C']
list2 = list1
list3 = list1.copy()

print(list1)
print(list2)
print(list3)
print()

list2.append('D')
list3.append('E')

print(list1)
print(list2)
print(list3)

>>> ['A', 'B', 'C']
['A', 'B', 'C']
['A', 'B', 'C']

['A', 'B', 'C', 'D']
['A', 'B', 'C', 'D']
['A', 'B', 'C', 'E']

You will see in the above code, appending 'E' to list3 has not resulted in this character being appended to list1.

As an aside, we can illustrate this point further with the id() built-in function. The function accepts a single parameter and returns the identity of that object. This identity will be unique and constant during the “lifetime” of the object. (Although two objects with non-overlapping lifetimes may have the same id() value.) You can see below you will see that list1 and list2 actually reference the same object, while in contrast list3 is a separate entity.

print(id(list1))
print(id(list2))
print(id(list3))

>>> 60937640
60937640
65185072

**Operations on collections**

As for strings, tuples may be concatenated using the + operator, or repeated with the * operator:

```python
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
```
>>> t1 + t2
(1, 2, 3, 4, 5, 6)

>>> t1 * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

The + and * operators work on lists in the same fashion as they do for tuples i.e. perform concatenation or repetition.

>>> 11 = [1, 2, 3]
>>> 12 = [4, 5, 6]
>>> 11 + 12
[1, 2, 3, 4, 5, 6]

>>> 11 * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

**Using the keyword `in` with collections**

In the section discussing strings, we encountered the `in` keyword, which was used to evaluate whether a specified character was found in a given string. The `in` keyword may be used in a similar fashion with collections: namely to check for the presence of a specified term. In the examples below, you will see how the `in` keyword can be used to assess whether a value is present in a range, tuple or list.

```python
r = range(10)
print(5 in r)
```

```
True
```

```python
t = ('Cat', 'Dog')
print('Cat' in t)
print('Pig' in t)
```

```
True
False
```

```python
l = [14, 7, 0, -3]
print(10 in l)
print(-3 in l)
```

```python
False
```
The `in` keyword may also be used on dictionaries. The keyword checks whether a specified key is found in the dictionary (i.e. the dictionary value is not evaluated), as shown below.

```python
da_team = dict({('Hannibal', 'Lieutenant Colonel John Hannibal Smith'),
    ('Face', 'Lieutenant Templeton Arthur Peck'),
    ('BA', 'Sergeant Bosco Albert Baracus'),
    ('Murdock', 'Captain H.M. Murdock')})

print('Hannibal' in a_team)
print('Lieutenant Colonel John Hannibal Smith' in a_team)
```

```bash
True
False
```

**Further points on manipulating collections with functions and methods**

**The key parameter**

Now that we are familiar with Python collections, this marks a good place to return once more to functions to discuss a slightly more complex mode of their action. Let's consider this with an example in which we want to sort a list of names alphabetically:

```python
managers = ['Ferguson', 'Wenger', 'de Boer', 'Ancelotti']
managers.sort()  # Sorts the list in place
print(managers)
```

```bash
['Ancelotti', 'Ferguson', 'Wenger', 'de Boer']
```

That's not quite what we wanted, for ‘de Boer’ was positioned at the end of the list. Why was that? Well the answer is that the `sort()` method uses ASCII character values to perform the sort, and while these are arranged alphabetically, all the lowercase letters come after all the uppercase letters. (More information on ASCII is found at: [https://en.wikipedia.org/wiki/ASCII](https://en.wikipedia.org/wiki/ASCII).)

**Key expressions**

This is an appropriate point to mention that methods or functions can be passed other functions as arguments. This may sound strange, so let's look at our example some more:

```python
managers = ['Ferguson', 'Wenger', 'de Boer', 'Ancelotti']
managers.sort(key=len)  # Sorts the list in place
```
print(managers)

>>> %Run misc.py
['Wenger', 'de Boer', 'Ferguson', 'Ancelotti']

We have modified the `sort()` method via the `key` parameter, which is used to pass `len` as an argument to the function (i.e. `key=len`). You may have guessed already, but `len` in the built-in Python function that determines the length of a string. By using this `key` parameter, the `sort()` method will use the function `len` as the basis to sort the list managers i.e. the lengths of the strings are used to sort the list, but the original values in the list are returned.

So, how does this help us with our original task? Well, let’s define a function that converts strings to lowercase (only functions may be passed via the `key` parameter and although there is a built-in method to convert strings to lowercase, there is not a built-in function)

```python
def to_lowercase(my_string):
    return my_string.lower()
```

```python
managers = ['Ferguson', 'Wenger', 'de Boer', 'Ancelotti']
managers.sort(key=to_lowercase)
print(managers)
```

```python
>>> %Run misc.py
['Ancelotti', 'de Boer', 'Ferguson', 'Wenger']
```

The sort method now converts the names to lowercase, sorts these values, and then returns the original values of the sorted list. We have now solved our original problem.

**Anonymous functions (the lambda function)**

It is possible in Python to declare functions with no name i.e. they are *anonymous*. (These anonymous functions are sometimes also known as *lambda functions*.)

The syntax required for declaring a lambda function is:

```
lambda arguments : expression
```

Here is the previous code re-written using the more concise lambda function notation:

```python
managers = ['Ferguson', 'Wenger', 'de Boer', 'Ancelotti']
managers.sort(key=lambda s: s.lower())
print(managers)
```

```python
>>> %Run misc.py
['Ancelotti', 'de Boer', 'Ferguson', 'Wenger']
```
Chapter 5 – Conditionals, Loops and Iterators

Conditionals

Simple, one-alternative and multi-test conditionals

A central concept in computing is the capability to evaluate a particular value, or set of values, and then make a “decision” based on this input. In Python, such conditional statements are created using the `if` keyword. The example below is a simple conditional expression:

```python
if 1==1:
    print('One is equal to one')

One is equal to one
```

You can think of the `if` keyword much the same as the English language equivalent, for it asks the question: “is the following true?” In the example, the expression that follows asks whether one is equal to one – which of course it is. The statement evaluating to `True` causes the code on the next line to be executed, whereas conversely `False` would cause the next line to be ignored. Notice at this point that the indentation scheme we encountered before. The conditional expression ends with a colon and the `print` statement that may be performed depending on the outcome of the `if` statement, is indented by 4 spaces.

Let’s illustrate the structure once again, but this time with a one-alternative conditional:

```python
if (2 + 2 == 5):
    print('Two plus two is five')
else:
    print('Two plus two is not equal to five')

Two plus two is not equal to five
```

In this example the condition expression evaluates to false. As before, this will cause the statement after the `if` keyword to be ignored, but the statement after the `else` keyword will be printed to the screen. This should make intuitive sense: if true do this, else do that. On many occasions, clear Python can almost be read as instructions written in the English language.

Building on the two previous examples, there may be occasions when you need to test many different conditional expressions. Such type of code is known as a multi-test conditional.

```python
ball = 22

if(ball == 13):
    print('Unlucky for some')
elif(ball == 14):
    print('Lawnmower')
elif(ball == 22):
    print('Two little ducks!')
```
else:
    print(ball)

Two little ducks!

For instructional purposes, we work our way through the code, from top to bottom. The name “ball” was assigned the integer 22. The if statement checks whether the value of the ball has a value of 1. Since it does not, we move to the elif keyword on the next line. This keyword is an abbreviation for “else if”, i.e. the previous expression evaluated to false, but does this expression evaluate to true? Well, the ball does not have a value of 4, and so we move to the next elif statement which checks whether the ball has a value of 22. Since the ball is equal to 22, the program prints “Two little ducks!” to the screen. We now exit this code block and so we never reach the final else keyword.

Loops

Loops are a central concept in many programming languages. They cause the same block of code to be repeated (i.e. looping) until a fixed number of repetitions has been achieved, or some other criterion is has been satisfied. There are many different types of loops in Python, each one suited to slightly different situations.
**While Loops**

As alluded to previously in this course, with much of the Python code it is possible for a beginner to make a reasonable guess at what a statement does simply from the meaning of the words in English. The **while loop** is another such example, and in simple terms the statement means: keep looping while this is true.

For example:

```python
x = 1
while(x < 5):
    print(x)
    x += 1
```

Is this code, \(x\) is assigned the value 1. We then proceed to the loop which will be carried out for so long as \(x\) is less than 5. At the end of this line of code we have the typical colon followed by the four-space line indentation on the next line. This indented code will be performed for each loop. The print statement will display the value of \(x\) on the screen, and then the value of \(x\) is incremented by 1. This will cause the values 1 to 4 (not 5, since \(x\) need to be less than 5) to be printed to the screen. It is worth pointing out that if the increment statement had not been included, \(x\) would have the value of 1 and the loop would be repeated forever; we would have created an **infinite loop**. Watch out for these when writing your code. If a program or operation has not terminated after a much longer time than expected, re-check that you have not inadvertently created an infinite loop.

You may append an **else** statement to your while loop which will be executed when the loop condition evaluates to **False**:

```python
x = 1
while(x < 5):
    print(x)
    x += 1
else:
    print("Not less than 5")
```

1
2
Not less than 5

These while loops can be structured in a different way. See the code below which also prints 1 to 4 to the screen, but is structured quite differently from before:

```python
x = 1
while(1):
    print(x)
    x += 1
    if(x >= 5):
        break
```

Again, \( x \) is set to 1. Now, however, we enter a while loop that runs while \( 1 \) is true. But all numbers other than 0 evaluate to True and 1 does not change its value, so we have created an infinite loop! This will indeed go on forever unless we break out, which is exactly what we do if \( x \) is greater than equal to 5. This break command is very useful when working with loops.

There is a related command to break named continue, which causes the code program to continue to the next loop, but importantly does not break out of the loop. Look at the code below

```python
x = 0
while(x < 10):
    x += 1
    if x % 2:
        continue
    print(x)
```

This causes the even numbers present in the range 1 to 10 to be printed to the screen. In this example, the integer \( x \) is initialised to 0 and is then incremented by 1 in a while loop, which recurs while \( x \) is less than 10. There is a conditional expression in the loop which results in a continue command if \( x \) is assigned to an odd integer (i.e. \( x \% 2 \) returns a remainder other than 0). The continue command causes a new loop starts, and consequently the value of \( x \) is not printed to the screen. If, however, \( x \) has an even value, then that value will be printed to the screen.

**Processing file input using a loop**

Loops are also useful to evaluate data received when reading a file (or produced by a generator). See the example below, in which a text file listing the integers from 1 to 100 on separate lines, is read one
line at a time. However, only the line numbers divisible by 10 (e.g. 10, 20, 30 etc) are subsequently printed to the screen by the Python script.

```python
with open(filename) as file:
    line = file.readline()
    count = 1
    while line:
        count += 1
        line = file.readline()
        if (count % 10 == 0):
            print(line)
```

Most of the code should look familiar, but please not the code `while line:`. This is a conditional `while` loop requiring the value of “line” to evaluate to true. Any line in a file (even an empty line – which is actually represented by the `\n` metacharacter) will correspond to true in such an evaluation, as consequently the loop will continue until the end of the file. Using this trick, you may read and process an input file on a line-by-line basis.

**Iterations**

**Iterators** are conceptually very similar to loops and indeed the terms are often used interchangeably. Loops run so long as the obligatory `while` statement evaluates to true. Iterations in contrast take place, by definition, on collection objects using the `for` keyword. The next few paragraphs give examples of how iterations are used on different containers or in different contexts.

(Please note that in many places the term iteration is used interchangeably with “loop”, or “for loops”.)

```
for every item
    Do something
```

**Simple Iteration**

The example below shows how to iterate over a collection object:

```python
people = ['Adam', 'Bob', 'Charlie']
for person in people:
    print(person)
```

```text
>>> Adam
Bob
```
Charlie

The for keyword tells the Python interpreter to iterate over the list ‘people’ (this need not be a list and could be another type of container object). The first element of ‘people’ is then assigned to the name ‘person’. The script then executes the block of indented code, printing the name of the person (i.e. ‘Adam’) to the screen. The iteration then continues with the second element of the list.

Iterating over a file

Iterating over a file object has a similar syntax, as shown in the example below which prints all lines in a file to the screen.

```python
filename = '/Users/wingetts/Desktop/one_hundred_lines.txt'
with open(filename) as file:
    for line in file:
        print(line)
```

Iterating over a dictionary

When iterating over a dictionary, it is possible to process either the keys, values or both. In the example below we have used the “a_team” dictionary from earlier in the course. The code iterates through the dictionary, assigning the keys to the alias name and the dictionary values to the name person. The code then prints these to the screen sequentially.

```python
a_team = dict({('Hannibal', 'Lieutenant Colonel John Hannibal Smith'),
               ('Face', 'Lieutenant Templeton Arthur Peck'),
               ('BA', 'Sergeant Bosco Albert Baracus'),
               ('Murdock', 'Captain H.M. Murdock')})
for alias, person in a_team.items():
    print(alias + '
    ' + person)
```

Enumerating iterations

A nice feature of Python is that it is easy to count the number of iterations performed using the enumerate function:

```python
people = ['Adam', 'Bob', 'Charlie']
for n, person in enumerate(people):
    print(n)
    print(person)
```

```text
>>> 0
Adam
1
Bob
2
```
The above code iterates over the list “people” as before, but in addition to assigning “Adam”, “Bob”, “Charlie” to “person”, the iteration number (starting at 0) is assigned to “n”.

**Iterating over ranges**

It is also possible to iterate over a range. In the example below, the range function generates integers 0 to 3. The resulting range may be iterated over using the `for` command, and the results are printed to the screen.

```python
for n in range(3):
    print(n)
```

```plaintext
0
1
2
```

**Nested iterations**

A nested iteration is an iteration within an iteration. It is quite common to see nested iterations in Python code for they provide an excellent way to combine values to create permutations of those values. For example, the nested iteration below generates the 16 di-nucleotide permutations of the 4 DNA nucleotide bases.

```python
base_list = ['A', 'G', 'C', 'T']

for base1 in base_list:
    for base2 in base_list:
        print(base1 + base2)
```

```plaintext
AA
AG
AC
AT

. .

. .
```

**Iterating over a string**

While strings are primitive data types, in some ways they may be thought of as more complex data structures comprising multiple different elements (i.e. a string of components). Owing to this, it is possible to iterate across a string in a similar fashion to a collection. See how we iterate across the string of eight letters in the example printed below.
my_string = 'ABCDEFGH'

for element in my_string:
    print(element)

>>> %Run misc.py
A
B
C
D
E
F
G
H

Passing iterables to functions

It is possible in Python to pass iterables to a function. Consider this code that defines and uses a function to convert to uppercase every string in a list:

def list_upper(original_list=[]):
    upper_list = []
    for element in original_list:
        element = upper_list.append(element.upper())

    return upper_list

my_list = ['a', 'b', 'c', 'd']
capitalised_list = list_upper(my_list)
print(capitalised_list)

The code should be understandable at this point: we have passed a list to a function, which subsequently iterates over the list while converting each element of the list to uppercase. Importantly, Python does not allow for a function to edit the original list, so we have to append the edited elements to a new list declared within the function. The function then returns the new edited list.

As an aside, it is worth noting that in the first line of the function definition we have specified an empty string as the default argument. This is a good idea, since if an iterable is not passed to the function from the main body of the script, then setting an empty list to the default value will prevent the function call from failing.
Concluding remarks

Well, that brings the course to an end. We have covered a lot of material, taking you from the absolute basics of learning a programming language to being able to write quite useful code in Python. You should now be familiar with the Python datatypes, understand concepts such as functions and methods and how programs are controlled using loops and conditional operators.

Now that these new skills are fresh in your mind, we strongly recommend that you go out of your way to find reasons to write code over the next few weeks. If you don’t build upon your current knowledge, as the weeks turn into months, you will become less familiar with what you have learned over the past few days. Maybe there is some analysis that you could now perform with Python? Even if it is easier to do the tasks in, say, MS Excel, reinforcing your new-found skills now will pay dividends in the future.

As mentioned previously, learning Python is akin to learning a foreign language. There is a great deal to take in and becoming fluent takes practice, practice, practice.

We would like to bring to your attention the following resources that may help you in your future Python career:

www.python.org – the homepage of Python. This should often be your first port of call for Python-related queries.

Babraham Bioinformatics also produces an Advanced Python training course where we delve more into this language. After completing this course, you should be able to construct relatively sophisticated code. For more details, please go to: https://www.bioinformatics.babraham.ac.uk/training

Happy coding!
The Babraham Bioinformatics Team