



Learning to Program with Python

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

Version 2020-01

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

| | |
|------------------------------------------------------|-----------|
| Licence | 2 |
| Table of Contents | 3 |
| Introduction | 6 |
| Why learn Python? | 6 |
| What this course covers | 6 |
| Python 2 or Python 3? | 7 |
| A note about learning a programming language | 7 |
| Chapter 1: getting started with Python | 8 |
| How to install Python | 8 |
| <i>On Linux/Unix/MacOSX etc.</i> | 8 |
| <i>On Windows</i> | 8 |
| What do I use to write Python code? | 9 |
| Thonny | 10 |
| Hello World! and getting started with Python | 11 |
| Chapter 2: data types and expressions | 12 |
| Integers (int) | 12 |
| Floats | 13 |
| Booleans (bool) | 14 |
| Strings (str) | 16 |
| <i>Operations on strings</i> | 17 |
| Compound expressions | 20 |
| Chapter 3: names, functions and methods | 21 |
| Writing scripts using Thonny | 21 |
| Names | 22 |
| F-strings | 24 |
| Functions | 25 |
| <i>Built-in functions</i> | 25 |
| <i>Functions and data types</i> | 27 |
| <i>Creating Functions</i> | 28 |
| Methods | 33 |
| Chapter 4 – Collections | 34 |
| Sets | 34 |
| Sequences | 36 |
| <i>Ranges</i> | 36 |
| <i>Tuples</i> | 37 |
| Lists | 38 |
| Mappings | 40 |
| <i>Dictionaries</i> | 40 |
| Streams | 42 |

| | |
|----------------------------------------------------------------------------|-----------|
| Files..... | 42 |
| Generators..... | 43 |
| Copying Collections..... | 43 |
| Further points on manipulating collections with functions and methods..... | 45 |
| <i>The key parameter</i> | 45 |
| <i>Key expressions</i> | 45 |
| <i>Anonymous functions (the lambda function)</i> | 46 |
| Chapter 5 – Conditionals, loops and iterations | 47 |
| Conditionals..... | 47 |
| <i>Simple, one-alternative and multi-test conditionals</i> | 47 |
| Loops..... | 48 |
| <i>While Loops</i> | 48 |
| <i>Processing file input using a loop</i> | 50 |
| Iterations..... | 50 |
| <i>Simple Iteration</i> | 50 |
| <i>Iterating over a file</i> | 51 |
| <i>Iterating over a dictionary</i> | 51 |
| <i>Enumerating iterations</i> | 51 |
| <i>Iterating over ranges</i> | 52 |
| <i>Nested iterations</i> | 52 |
| <i>Iterating over a string</i> | 52 |
| <i>Passing iterables to functions</i> | 53 |
| Comprehensions..... | 54 |
| <i>List Comprehensions</i> | 54 |
| <i>Set Comprehensions</i> | 54 |
| <i>Dictionary Comprehensions</i> | 55 |
| <i>Conditional Comprehensions</i> | 55 |
| Generators..... | 55 |
| A note on scope..... | 56 |
| Chapter 6 – Exception handling | 60 |
| Chapter 7 – Object-oriented programming..... | 63 |
| The concepts of object-oriented programming..... | 63 |
| Defining classes..... | 63 |
| Instance Attributes..... | 64 |
| Access Methods..... | 64 |
| Predicate Methods..... | 66 |
| Initialisation Methods..... | 67 |
| String Methods..... | 67 |
| Modification Methods..... | 68 |
| Additional Methods..... | 69 |
| Class Attributes..... | 69 |
| Static methods..... | 70 |

| | |
|-------------------------------------------------|-----------|
| Inheritance | 71 |
| Inheritance and super() | 72 |
| A brief note on creating exceptions | 73 |
| Chapter 8 – Modules | 74 |
| Introducing modules | 74 |
| <i>The datetime module</i> | 74 |
| <i>The math module</i> | 77 |
| <i>The sys module</i> | 77 |
| <i>The time module</i> | 78 |
| <i>The optparse module</i> | 78 |
| <i>The subprocess module</i> | 78 |
| <i>The os module</i> | 79 |
| <i>The tempfile module</i> | 79 |
| <i>The glob module</i> | 80 |
| <i>The textwrap module</i> | 80 |
| <i>The string module</i> | 82 |
| <i>The csv module</i> | 82 |
| <i>The zlib and gzip modules</i> | 82 |
| Installing Modules and Packages | 83 |
| <i>Installation</i> | 83 |
| <i>Installation locations</i> | 84 |
| Virtual Environments | 85 |
| Biopython | 85 |
| Chapter 9 – Regular expressions | 86 |
| Introducing the re module | 86 |
| Simple String Matching with the re Module | 86 |
| Querying the match object | 87 |
| Metacharacters | 88 |
| <i>Character classes</i> | 88 |
| <i>Start and ends</i> | 88 |
| <i>Groups</i> | 88 |
| <i>Backslash</i> | 89 |
| <i>Escaping Using Backslashes</i> | 90 |
| <i>Raw String Notation</i> | 90 |
| <i>Repetition</i> | 90 |
| <i>Greedy vs non-greedy matching</i> | 91 |
| <i>Compilation flags</i> | 92 |
| Modifying Strings | 93 |
| Concluding remarks | 94 |

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 will often 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 tools 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 or Flask are the most noteworthy of these web creation tools.

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

Good things about Python:

- It's free
- 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 numerical "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

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 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, the most valued of skills are not easy to acquire, and demand 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 is not installed, you will need to download it from *The Python Software Foundation* website at: www.python.org.

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.

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:

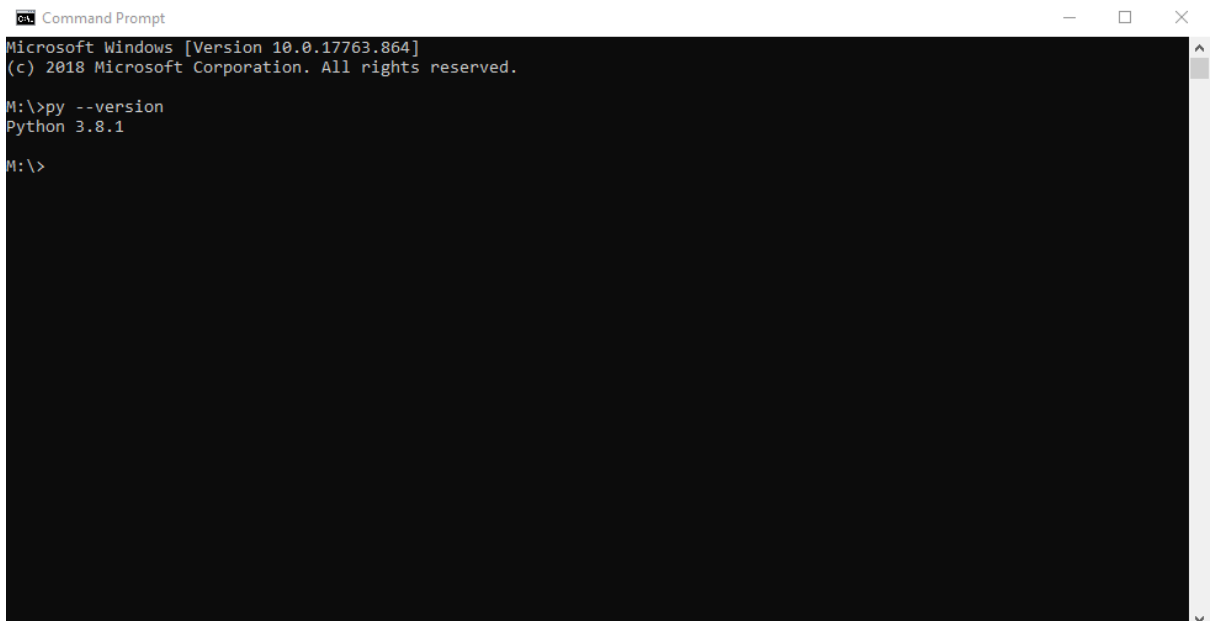
- 1) Try right-clicking on your desktop and it's probably listed as one of the options
- 2) Look in Start > Programs for an entry called “MS-DOS Prompt”
- 3) Look in Start > Programs > Accessories for an entry called “Command Prompt”
- 4) Go to Start > Run. In the box type “cmd” and press return

One of these alternatives should get you a command prompt.

At the command prompt type in the command

```
python --version
```

If python is installed you should see something like this:



```
Command Prompt
Microsoft Windows [Version 10.0.17763.864]
(c) 2018 Microsoft Corporation. All rights reserved.

M:\>py --version
Python 3.8.1

M:\>
```

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 and boast a large number of features to make this process easier. For example, what is almost immediately obvious 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/>

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/>

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 Code

This is another versatile text editor available for the three main types of operating system. Although produced by Microsoft, this is a free piece of software.

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.

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.

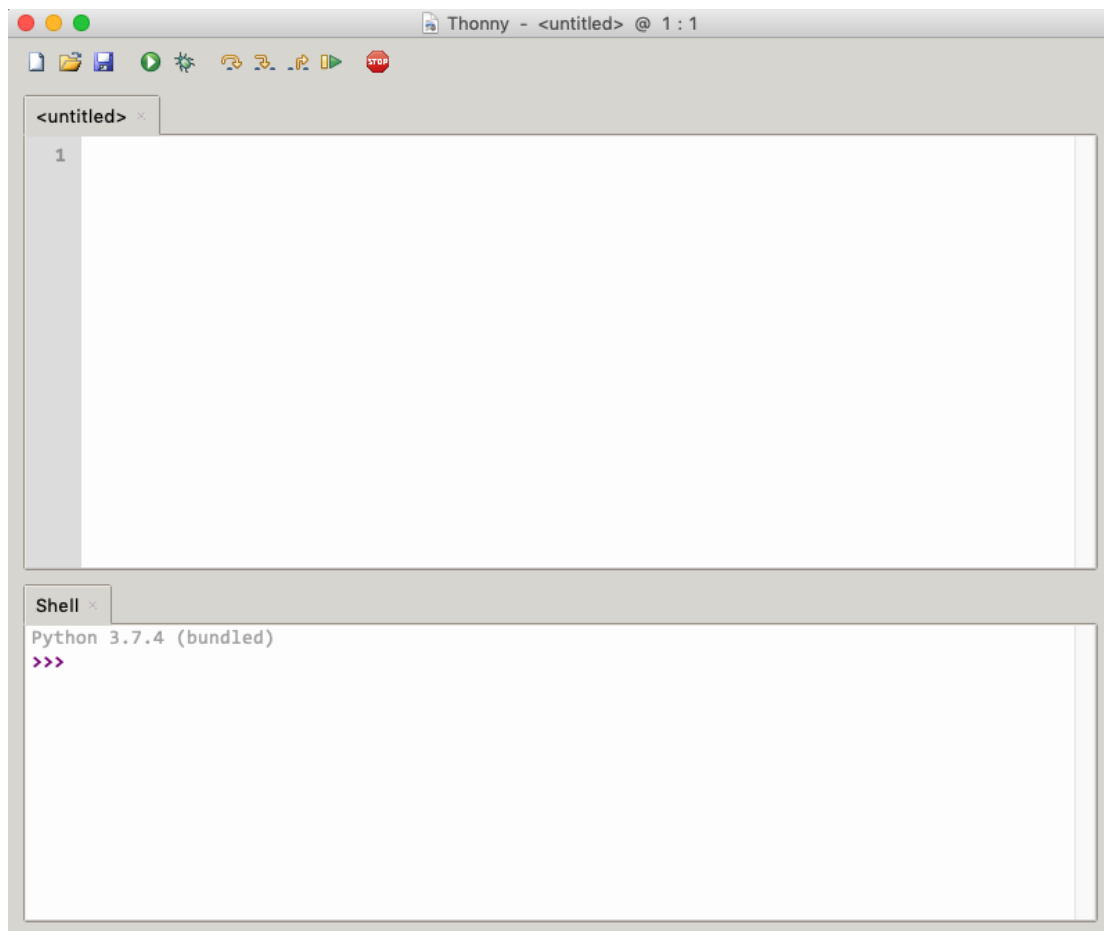
Thonny

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 a *.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 shell 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.



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:

```
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 $\sqrt{2}$ 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:

```
>>> 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:

```
>>> 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:

```
>>> 2 * 2
4
```

```
>>> 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:

```
>>> 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 10...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 significant of 1.0 and an exponent of 3, but storing integers in this way uses more of the available memory.

| Value | Significand | Exponent |
|-------|-------------|----------|
| 0.5 | 5 | -1 |
| 0.001 | 1 | -3 |
| -10.5 | -10.5 | 1 |

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.

```
>>> 3.0E10
30000000000.0
```

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:

```
>>> 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.

```
>>> 10 / 4
2.5
>>> 10 // 4
2
>>> 10 % 4
2
```

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

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**.

| Comparison Operator | Description |
|---------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>==</code> | If the values of two operands are equal, then the condition is <code>True</code> |
| <code>!=</code> | If values of two operands are not equal, then condition is <code>True</code> |
| <code>></code> | If the value of left operand is greater than the value of right operand, then condition is <code>True</code> |
| <code><</code> | If the value of left operand is less than the value of right operand, then condition is <code>True</code> |
| <code>>=</code> | If the value of left operand is greater than or equal to the value of right operand, then condition is <code>True</code> |
| <code><=</code> | If the value of left operand is less than or equal to the value of right operand, then condition is <code>True</code> |

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

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> True and True
True
>>> True and False
False
```

In contrast, the `or` operator behaves differently:

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

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

```
>>> 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.

```
>>> 1 and 0
0
>>> 1 and 2
2
>>> 1 or 0
1
>>> 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.

```
>>> 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.

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**.

```
>>> '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**:


```
>>> 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
```

Placing a backslash before certain letters cause them to be interpreted differently by Python. As you can see above, `\n` is interpreted as a new line, whereas `\t` is interpreted as a tab:

```
>>> 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"
```

Both the above lines will generate the desired string. This use of the backslash may seem a little bit odd at the moment, but its mode of action should become clear in a later chapter discussing regular expression metacharacters – so please be patient until then.

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.

```
>>> '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 'HaystackHaystackNeedleHaystackHaystack'
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 'HaystackHaystackNeedleHaystackHaystackHaystackHaystack'
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 begins 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 0, and then consequently the Python interpreter returns the first 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:

```
>>> '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.

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

```
'Bab'
```

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

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

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

```
>>> '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:

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

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:

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

Simple...but wait, shouldn't the answer be 30 and not 20? Well, Python follows orders of precedence and will perform the multiplication before the addition. Thus, $5 * 3 = 15$. 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), 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:

```
>>> (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 edit, 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:

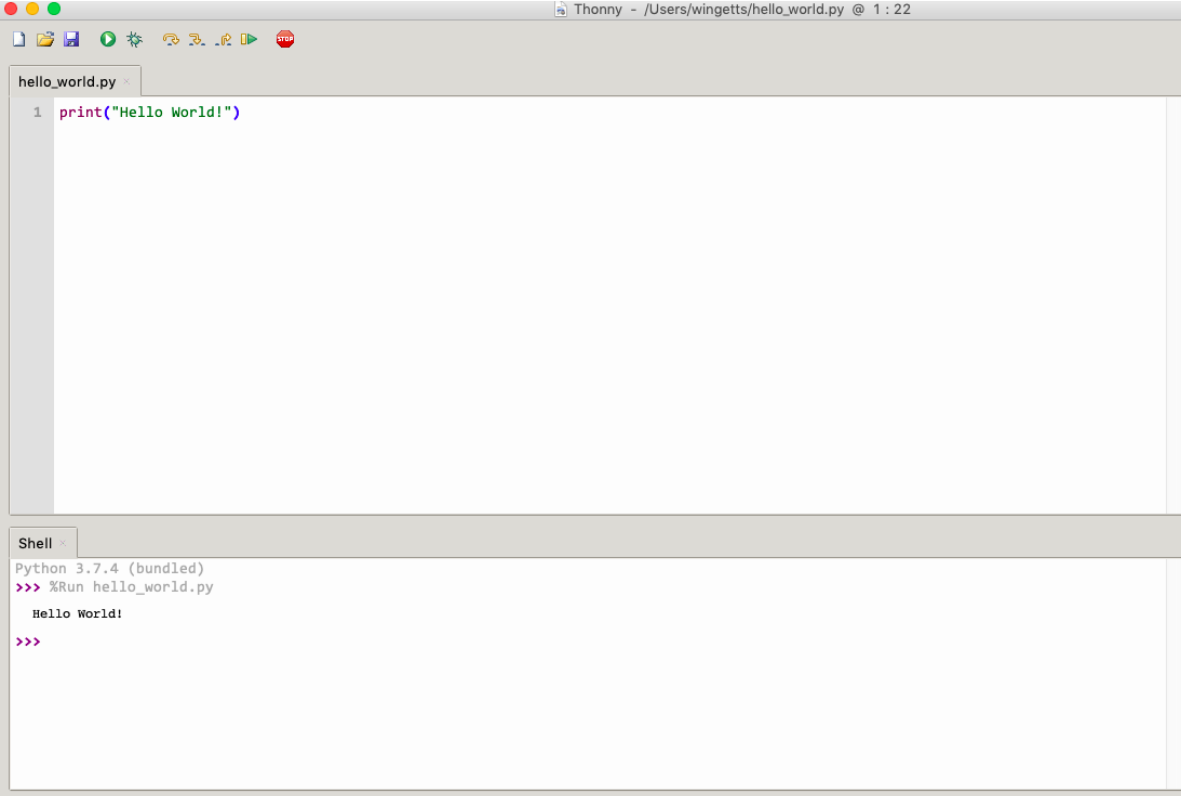
```
print("Hello World!")
```

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 field, 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:

```
>>> %Run hello_world.py
Hello World!
```



```
Thonny - /Users/wingetts/hello_world.py @ 1 : 22

hello_world.py
1 print("Hello World!")

Shell
Python 3.7.4 (bundled)
>>> %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 notice 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_word.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:

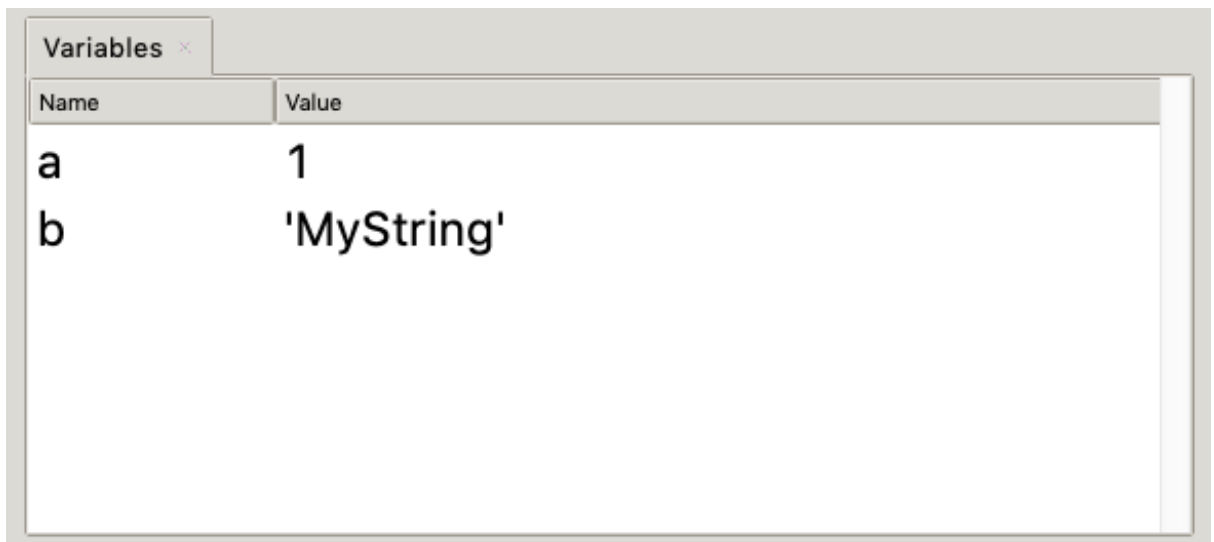
```
>>> a = 1
```

This assigns the **name** '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 **statement**, which unlike expressions in the previous section, do not return values. What you should also notice is that the **names** appear in the **Variables window**, along with their associated values



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

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

It is possible to assign a name to an existing name

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

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

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

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

```
>>> 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:

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

In this example, the value of `c` is deduced to be 2, 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 lower- or 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:

```
>>> 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:

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


Similarly,

```
>>> 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:

```
>>> 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).

```
>>> 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.

```
>>> 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 `<`, `>` and `^` denote *left-*, *right-* and *centre-aligned formatting*, respectively.

```
>>> 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:

```
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 are 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:

```
>>> 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:

```
>>> len('Supercalifragilisticexpialidocious')
34
```

`min(arg...)` – returns the minimum value of a list of numerical arguments:

```
>>> min(2, 4, 100, 205.3, -4)
-4
```

`max(arg...)` – returns the maximum value of a list of numerical arguments:

```
>>> 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. Alternative, 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.

| | | | | |
|----------------------------|--------------------------|---------------------------|---------------------------|-----------------------------|
| <code>abs()</code> | <code>delattr()</code> | <code>hash()</code> | <code>memoryview()</code> | <code>set()</code> |
| <code>all()</code> | <code>dict()</code> | <code>help()</code> | <code>min()</code> | <code>setattr()</code> |
| <code>any()</code> | <code>dir()</code> | <code>hex()</code> | <code>next()</code> | <code>slice()</code> |
| <code>ascii()</code> | <code>divmod()</code> | <code>id()</code> | <code>object()</code> | <code>sorted()</code> |
| <code>bin()</code> | <code>enumerate()</code> | <code>input()</code> | <code>oct()</code> | <code>staticmethod()</code> |
| <code>bool()</code> | <code>eval()</code> | <code>int()</code> | <code>open()</code> | <code>str()</code> |
| | | | | |
| <code>breakpoint()</code> | <code>exec()</code> | <code>isinstance()</code> | <code>ord()</code> | <code>sum()</code> |
| <code>bytearray()</code> | <code>filter()</code> | <code>issubclass()</code> | <code>pow()</code> | <code>super()</code> |
| <code>bytes()</code> | <code>float()</code> | <code>iter()</code> | <code>print()</code> | <code>tuple()</code> |
| <code>callable()</code> | <code>format()</code> | <code>len()</code> | <code>property()</code> | <code>type()</code> |
| <code>chr()</code> | <code>frozenset()</code> | <code>list()</code> | <code>range()</code> | <code>vars()</code> |
| <code>classmethod()</code> | <code>getattr()</code> | <code>locals()</code> | <code>repr()</code> | <code>zip()</code> |
| <code>compile()</code> | <code>globals()</code> | <code>map()</code> | <code>reversed()</code> | <code>__import__()</code> |
| <code>complex()</code> | <code>hasattr()</code> | <code>max()</code> | <code>round()</code> | |

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.

```
>>> 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**.

```
>>> 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.)

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.

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 an argument and then prints its value 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 for 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 begin 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 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. 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:

```
>>> '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'. As for functions, Python has a large number of built-in methods for its datatypes. Listed below are the commonly used string methods. These are by no means all the methods available, just this relatively small list allows a programmer to substantially increase their ability to manipulate strings.

| Method | Purpose |
|---------------------------|------------------------------------------------------------------------------------------|
| <code>capitalize()</code> | Converts the first character to upper case |
| <code>count()</code> | Returns the number of times a specified value occurs in a string |
| <code>find()</code> | Searches the string for a specified value and returns the position of where it was found |
| <code>isalnum()</code> | Returns <code>True</code> if all characters in the string are alphanumeric |
| <code>isalpha()</code> | Returns <code>True</code> if all characters in the string are in the alphabet |
| <code>isdecimal()</code> | Returns <code>True</code> if all characters in the string are decimals |
| <code>join()</code> | Joins the elements of an iterable to the end of the string |
| <code>lower()</code> | Converts a string into lower case |
| <code>replace()</code> | Returns a string where a specified value is replaced with a specified value |
| <code>split()</code> | Splits the string at the specified separator, and returns a list |
| <code>splitlines()</code> | Splits the string at line breaks and returns a list |
| <code>strip()</code> | Returns a trimmed version of the string |
| <code>title()</code> | Converts the first character of each word to upper case |
| <code>upper()</code> | Converts a string into upper case |

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 order 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:

```
>>> 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)

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

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

Intersection (identify elements common to both sets)

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

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

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

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

```
>>> beatles.difference(wilburys)
{'Paul', 'John', 'Ringo'}
```

Symmetric Difference (items present that are not common to both sets)

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

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

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

```
empty = 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:

```
>>> 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:

```
>>> 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:

```
>>> 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

```
>>> 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 be empty, but if you choose a negative step value, then stop will need to be less than then start:

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

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:

```
>>> (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**.

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

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

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

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

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

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

```
>>> 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:

```
>>> 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 in a single expression using comma-separated lists:

```
>>> 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.

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

```
>>> 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).

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

| Method | Purpose |
|----------------------|-----------------------------------------------------------------------------------------|
| <code>count()</code> | Returns the number of times a specified value occurs in a tuple |
| <code>index()</code> | Searches the tuple for a specified value and returns the position of where it was found |

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 rounds brackets. The code below illustrates this point:

```
>>> 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:

```
>>> 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) 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:

```
>>> temps = ('Jimmy', 'Eric')
>>> beatles[3:4] = temps
>>> beatles
['John', 'Paul', 'George', 'Jimmy', 'Eric']
```

Here we have replaced the George and 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.

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

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

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

Note that calling the desired method in the above example **modifies the list immediately** and the list does not have to be re-assigned to itself i.e. we do **not** do the following

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

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

Mappings

Mapping are **mutable** structures that store data in what is known as “**key/value**” **pairs**. Conceptually, you may think of these data types as a chest of drawers, in which each drawer is labelled. Into each drawer may be placed one – and not more than one – item. The drawer label corresponds to the key and the item in the drawer corresponds to the value. Such data structures are **unordered** and the **keys must be unique**.

Dictionaries

The only mapping datatype encountered in Python is known as a **dictionary** (`dict`), which is an intuitive name: the key serves as the dictionary word to look-up, and the value serves as the definition of that word. To declare a dictionary, use the following syntax:

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

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:

```
>>> 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:


```
>>> 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:

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

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

```
>>> 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:

```
>>> 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:

```
>>> 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.

```
>>> 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.

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

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.

```
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 (`w` 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 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 values 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`.

```
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)
```

```
>>> %Run mypy.py
['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
```

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:

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

```
>>>
['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>.)

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:

```
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

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

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

```
>>> %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:

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

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

Chapter 5 – Conditionals, loops and iterations

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:

```
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 of 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**:

```
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**.

```
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 as the Python interpreter would do when executing the program. The name “ball” was assigned the integer 22. The `if` statement checks whether the value of the ball as 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:

```
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:


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

```
1
2
3
4
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:

```
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

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

```
2
4
6
8
10
```

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.

```
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 `\n`) 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

Iterations 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”.)

Simple Iteration

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

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

```
>>>
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.

```
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.

```
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 + "\n" + person)
```

Enumerating iterations

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

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

```
>>>
0
Adam
1
Bob
2
Charlie
```

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.

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

```
>>>  
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.

```
base_list = ['A', 'G', 'C', 'T']  
  
for base1 in base_list:  
    for base2 in base_list:  
        print(base1 + base2)
```

```
>>>  
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.

Comprehensions

We have now introduced conditional expressions, loops and the closely related iterations. When used together these key components of the Python language prove extremely versatile when it comes to modifying data. In addition, Python has an elegant way achieving these same tasks but in single lines of code in what are known as **comprehensions**. These comprehensions can be used to create sets, lists or dictionaries from using a collection as a starting point. It is worth taking the time to become familiar with the syntax of comprehensions, since by using them you should be able to write more succinct elegant code.

List Comprehensions

List comprehensions generate **lists as output** and have the following template syntax:

```
[expression for item in collection]
```

That is to say, this comprehension will perform the specified expression on every item in the collection. To illustrate this point, look at the next example in which we generate square numbers using 0 to 10 as the root values. Sure, we could use a loop to achieve the same task, but a one-line list comprehension is a succinct Pythonic way to manipulate collections.

```
S = [x**2 for x in range(11)]
print(S)

>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

We created a collection object with the term `range(11)`, and we iterated over this range, assigning each element to `x`. Each value of `x` is subsequently squared and added to the list generated by the comprehension. This list is then assigned to the name `S`.

Set Comprehensions

Set comprehensions are similar to list comprehensions and have a similar syntax, but are enclosed in curly brackets rather than round brackets. As their name suggests, they generate **sets** as output.

```
{expression for item in collection}
```

It is worth bearing in mind that sets only contain unique items, and so you cannot be sure that the output will have the same number of elements as the input. We see in the following example a list containing duplicates of the letter 'C'. Using the set comprehension, every value from the list (assigned the name `x`) is allocated to the newly generated set. The set has no duplicate values.

```
my_list = ['A', 'B', 'C', 'C', 'C']
my_set = {x for x in my_list}
print(my_set)
print(type(my_set))

>>>
{'B', 'C', 'A'}
```

```
<class 'set'>
```

Dictionary Comprehensions

Essentially the same rules apply to **dictionary comprehensions** as for set comprehensions, but of course dictionaries are produced as output. Dictionary comprehensions have the template:

```
{key-expression: value-expression for key, value in collection}
```

Consider the code below that transposes dictionary keys for values.

```
my_dict = {1:"Red", 2:"Green", 3:"Blue"}
print(my_dict)
print( {value:key for key, value in my_dict.items()} )
```

```
>>>
{1: 'Red', 2: 'Green', 3: 'Blue'}
{'Red': 1, 'Green': 2, 'Blue': 3}
```

Conditional Comprehensions

Conditional comprehensions are an excellent way to filter a comprehension. These comprehensions have the code structure:

```
[expression for element in collection if test]
```

In the example below a range of integers from 0 to 9 is generated, but only those with values less than 3 are printed to the screen, owing to the conditional comprehension.

```
print([x for x in range(10) if x < 3])
>>>[0, 1, 2]
```

Generators

We previously alluded briefly to the Python object known as a generator, which produces a stream of data. In many cases this is preferable to an alternative strategy of keeping a very large list in memory, and then selecting a value from the list as required. The syntax for making a generator is almost identical to that of declaring a function, except that code defining a generator ends with the keyword `yield` instead of `return`. The keyword `next` instigates each iteration of the generator.

In the following example we create a generator that produces even numbers. Each time the `next` command is run in the loop, the generator yields the next even number in the sequence. Importantly, notice that once we leave the loop, but call the generator once more, we produce the next even number in the sequence (i.e. the generator keeps track of “where it has got to”).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2

even_number_generator = all_even()

for i in range(5):
    even_number = next(even_number_generator)
    print(even_number)

print("And again...")
print(next(even_number_generator))

>>>
0
2
4
6
8
And again...
10
```

Generators don't have to produce an infinite number of values. We could specify some a limit to the number of values that can be returned. Once this is reached, further iteration will not be possible and a `StopIteration` error will occur.

So to recap, generators constitute a clear and concise way to make data. They are memory efficient as compared to regular functions that need to create an entire sequence in memory before returning the result. Owing to these reduced memory overheads, generators are generally more efficient in such instances. Generators may also be joined to one another (i.e. the output of one taken as input by another), making data pipelines.

A note on scope

We have discussed that assigning names to values and then using these values in subsequent calculations and manipulations. However, just because we have created a name, it does not mean it is accessible to every part of a script. We list some example of how scoping works in Python. These may be particularly interesting for people familiar with other languages (such as Perl) who may get caught out by Python's behaviour.

Variables created in out outer bloc of code will be accessible to the inner bloc of code

You can see below that `h` is in scope inside the `for` loop, since it is successfully added to the value of `i` and then printed to the screen.


```
h = 10
for i in range(2):
    print(h+i)
```

```
>>> %Run mypy.py
1
10
11
```

Likewise, variables declared outside a function are available within that function:

```
def test_function():
    print(x)
```

```
x = 1
test_function()
```

```
>>> %Run mypy.py
1
```

Loops can modify variables “external” variables, but functions cannot

You see in the example below that, as expected, the toplevel name `x` is accessible within a loop and a function. However, these control structures have different behaviour with regards to modifying `x`.

```
x = 1

for i in range(2):
    x = x + 1
print(x)
```

```
>>> %Run mypy.py
3
```

```
def test_function():
    x = x + 1
    print(x)
```

```
x = 1
test_function()
```

```
>>>
```

```
Traceback (most recent call last):
```

```
File "C:\Users\wingetts\Desktop\mypy.py", line 8, in <module>
    test_function()
File "C:\Users\wingetts\Desktop\mypy.py", line 5, in test_function
    x = x + 1
UnboundLocalError: local variable 'x' referenced before assignment
```

Variables created within a function will not be accessible outside that function

In the two examples below, the code prints the names `x` or `y`, which have been declared within a function. In both cases an error is generated.

```
def add_one(x):
    return(x + 1)
```

```
print(add_one(1))
print(x)
```

```
>>>
```

```
2
```

```
Traceback (most recent call last):
```

```
File "C:\Users\wingetts\Desktop\mypy.py", line 5, in <module>
    print(x)
```

```
NameError: name 'x' is not defined
```

```
def add_one(x):
    y = x + 1
    return(y)
```

```
print(add_one(1))
print(y)
```

```
2
```

```
Traceback (most recent call last):
```

```
File "C:\Users\wingetts\Desktop\mypy.py", line 6, in <module>
    print(y)
```

```
NameError: name 'y' is not defined
```

Variables created in a loop will be accessible outside that loop

In the example below, the names `i`, `j` and `k` are declared within a loop (or a pair of nested loops). Despite this, all these names are accessible to the toplevel coding area.

```
for i in (range(1)):  
    for j in (range(10, 11)):  
        print(i)  
        print(j)  
        k = i + j  
        print(k)  
        print()  
  
print("Outside loop i:" + str(i))  
print("Outside loop j:" + str(j))  
print("Outside loop k:" + str(k))  
  
>>> %Run mypy.py  
0  
10  
10  
  
Outside loop i:0  
Outside loop j:10  
Outside loop k:10
```

Just to make you aware that there is a keyword `global` that can modify Python scoping, but we shall not discuss this further in this course.

Chapter 6 – Exception handling

No doubt by now when trying to run code you have written, the Python interpreter will have responded by outputting to the screen a **traceback** message of some kind, meaning that there is some kind of error in your code. It is worth pointing out that taking the time to read the traceback message is certainly worthwhile, as the output gives an explanation why the code failed. Although it not always obvious what the output means, with time and experience the traceback message should help you identify most bugs (coding errors) in a relatively short amount of time. Despite any initial misgivings you may have, you will soon find out that the traceback message is the Python coders' friend.

Even if you write perfect code, however, there are events which will be entirely beyond a programmer's control. Suppose you write a script that takes input from a file, but unfortunately, for whatever reason, that file was deleted from the computer running the Python script and so when the program tries to open the file, an error will result. When using a script you have written, or moreover using a script someone else has written, there are countless ways in which the script could fail. However, a good programmer will plan for those eventualities and although you cannot stop your script failing, when it does fail, you can ensure it fails as gracefully as possible.

Errors can be categorised into different types: trying to open a file that is not on your system you would generate an `IOError`. In contrast, a `NameError` would result if trying to use a name that had not yet been declared within your script.

Fortunately, Python has a feature available to a programmer to deal with such unexpected errors. This capability is known as **exception handling**, and enables a programmer to provide instructions to the script to be on the "look out" for certain types of error, and provides a "contingency plan" of what to do if such an error is encountered. So, an **exception is an error that is handled by the Python code**, and will be dealt with in a pre-specified manner. **Unhandled exceptions** are errors and will result in a **traceback** message.

The basic way to deal with errors is to use a `try` statement followed by an `except` clause. Listed below is a standard template for handling exceptions:

```
try:
    statements we try to run
except ErrorClass:
    what to do if an error of this class is encountered
```

Let's illustrate this with an example:

```
my_list = ['A', 'B', 'C']
print(my_list[2])

>>>
Traceback (most recent call last):
  File "C:\Users\wingetts\Desktop\thonny.py", line 4, in <module>
    print(my_list[3])
IndexError: list index out of range
```

We have generated a “list index out of range error” by trying to get the value at index 3 from `my_list`. Although `my_list` has 3 elements, the numbering of indices starts at 0, so there is no value at position 3 in our list. We could write an exception handler for this:

```
try:
    my_list[3]
except IndexError:
    print("That index is outside the list!")

>>>
That index is outside the list!
```

Now, thanks to the exception handler, the code does not terminate with a traceback, but rather prints to the screen a user-friendly description of what went wrong. Furthermore, if this code were part of a larger program, the program should carry on running while conversely, without the exception handler, the program would terminate abruptly.

Many error handlers are more complex than the example above, catching many different types of errors, and potentially handling each type of error differently. Look at the following template:

```
try:
    statements we try to run
except ErrorClass1:
    what to do if an error of class 1 is encountered
except ErrorClass2:
    what to do if an error of class 2 is encountered
except (ErrorClass3, ErrorClass4):
    what to do if an error of class 3/class 4 is encountered
except ErrorClass5 as err:
    what to do if an error of class 5 is encountered
except:
    statement to execute if error occurs
finally:
    statement carried out if exception is/is not made
```

The above template shows that the error handling reporting process can be tailored to the type of error generated, for example `ErrorClass1` and `ErrorClass2` will result in different responses. `ErrorClass3` and `ErrorClass4` will generate the same response. The response to `ErrorClass5` generates an error object called `err` using the `as` keyword. This object may then be processed by the script as required. The statement following the last `except`, in which the error type is not defined, will be performed if an error takes place, but has not been specified previously in the error handling block of code. The `finally` statement will be carried out regardless, irrespective of the type of error, or indeed if any error, has occurred. To illustrate a more complex error handling process, look at the code below:

```
my_dict = {"Aardvark":1, "Baboon":2, "Cougar":3}
try:
    value = my_dict["Dandelion"]
except IndexError:
    print("This index does not exist!")
except KeyError:
    print("This key is not in the dictionary!")
except:
    print("Some other error occurred!")
finally:
    print("Let's carry on")
```

```
>>>
This key is not in the dictionary!
Let's carry on
```

The code creates a dictionary containing three values which are animals, but then tries to retrieve a value (i.e. "Dandelion") not in the dictionary. This returns a `KeyError` leading to message "This key is not in the dictionary!" being printed to the screen. The `finally` clause is then executed after the rest of the code.

It is also possible to write code to create an **exception object**, which may be returned should an error occur. But to do that, you first need to understand more about Python objects and classes, and indeed become familiar with **object-orientated programming**. The next section discusses these topics.

Chapter 7 – Object-oriented programming

The concepts of object-oriented programming

A strength of Python and a feature that makes this language attractive to so many, is that Python is what is known as an **object-oriented programming language (OOP)**. (You may occasionally see this written as “orientated” in British English.)

The alternative programming style is procedural, which may be thought of as a set of ordered instructions. Giving someone geographical directions makes a good analogy to procedural instructions: e.g. 1) take the second right, 2) go straight on at the roundabout and 3) turn left at the lights. This style is what most people think of by the term programming and indeed, this is how we have approached programming up until now in this course, since it is a simple and effective way to complete tasks of basic-to-intermediate complexity. As you build more complex programs, however, you may find it becomes ever more difficult to keep track in your own mind as to what is going on. What does a particular function or variable do? How should I arrange my many pages of code? Should I make a value accessible to all parts of my code? These questions you may ask yourself as your codebase increases in size.

OOP is easier for humans to understand, particular as a program increases with size, because it models our everyday world. That is to say, it categorises its components into objects, which may be thought of as self-contained entities that have their own properties. Different objects may interact with one another and related objects constitute groups know as classes.

In reality, the distinction between an OOP language and a procedural language is somewhat blurred. Perl (previously the most popular bioinformatics language) for example has an OOP component, but it is quite common for even experienced aficionados to hardly ever use this aspect of the language. The statistical programming language R is similar in this regard, but many users will only explicitly deal with R objects when processing the output from external modules. In contrast, Java was designed as OOP from the ground up, and learners will be introduced to these concepts right from the start. Python falls between Perl and Java in that it is quite possible for programmers to write code with only a passing familiarity with objects, such as when executing methods on particular objects. However, with a little bit more experience it is quite possible to build complex object-orientated software in a style more typical to Java.

Defining classes

As mentioned before, **classes** are groups of related objects. For example, a particular dog is an **instance** but of the dog class. If we wanted to create a dog in our program, we would define the dog class, and then make a *specific* dog from that class. Each dog would constitute a separate Python object, modelling the real world. (Technically speaking, in Python even the abstract concept of a class is an object in its own right, but nevertheless you should get the idea that when using this programming style we create discrete data structures analogous to physical objects.)

So, we would define our dog class using the keyword `class`, as shown in the simple example below:

```
class dog:
    pass
```

All the dog class contains is the keyword `pass`, the placeholder value that allows a block of code to do nothing, without generating an error. If you were now to type `dog()` into the interpreter, you should see a message similar to this:

```
<__main__.dog object at 0x0341D7B0>
```

The text “__main__” is the name of the module to which the dog class belongs (main is the Python interpreter). Next is the name of the class followed by an internal memory address (written in hexadecimal).

To make an instance of the dog class, simply call the class as you would a function:

```
snoopy = dog()
```

This instance of the dog class is named snoopy. You may view its memory location as well:

```
>>> dog
<__main__.dog object at 0x0410D7F0>
```

Instance Attributes

Instances of a class may have **methods** (such as already seen with built-in objects) and store information in what is known as **fields**. Collectively, methods and fields are known as **attributes**. Both of these may be accessed using the dot notation.

Suppose we wanted to set a field for our dog, snoopy, we would do the following:

```
snoopy.colour = 'White'
print(snoopy.colour)
```

All other instances of the dog class will not have a colour field; only snoopy will be changed by this statement. Although this is a simple and quick way to edit the snoopy instance, there are better ways to do this. We shall now work through the commonly used attributes of an instance, building our dog class as we go.

Access Methods

This type of method returns values based on the fields of an instance. The code below re-writes the dog class so that now instead of simply the pass keyword, the class now has a method named `get_colour`. To define a method within a class, use the `def` keyword which we encountered when creating functions. You can see that calling this method returns the value `self.colour`. But where does `self.colour` come from? Well, `self` refers to the current instance of a class, and so the return statement is in effect saying “return the value of colour associated with **this instance** (i.e. snoopy) of the dog class”.

```
class dog:
    def get_colour(self):
        return self.colour
```

```
>>> snoopy.get_colour()
'White'
```


You may be wondering as to the point of writing such a method. Wouldn't it be easier simply to type the following?

```
>>> snoopy.colour
'White'
```

And you would be correct, this is easier and quicker to do and will return the correct answer. Suppose, however, that at a later date you, or someone else, changes how the dog colour values are stored within a class. Maybe you decide to store all useful variables in a dictionary. This will mean that code that interacted directly with the colour name will no longer work. Having methods to enable your class instance to interact with the outside world enables programmers to modify the internal structure of such an object, while still allowing the object to function correctly.

While access methods retrieve values based on the current state of an instance of a class, these methods do not simply have to return a value. They may, for example, perform a test of some kind before returning a value. In the code printed below, we have modified the dog class once more to include an action method that will evaluate the mood of the dog and return a different string response depending on that mood. Consequently, when snoopy is happy he wags his tail, but when he is angry you need to watch out, because he will bite!

```
class dog:
    def get_colour(self):
        return self.colour

    def action(self):
        if self.mood == 'Happy':
            return('Wag Tail')
        elif self.mood == 'Angry':
            return('Bite')
        else:
            return('Bark')
```

```
snoopy = dog()
```

```
snoopy.mood = "Happy"
print((snoopy.action()))
snoopy.mood = "Angry"
print((snoopy.action()))
```

```
>>>
Wag Tail
Bite
```

Predicate Methods

A **predicate method** returns either a `True` or `False` value. By convention, such methods begin with an `is_` prefix (or sometimes `has_`, depending on the grammatical context of the method name).

In the example below, we have modified the `dog` class to contain a predicate method that reports whether a dog is hungry (for brevity, we have removed the other methods from the class). The degree to which the dog's stomach is full is associated with the name `stomach_full_percentage`. If this value drops below 30, the `is_hungry` predicate method will return `true`.

```
class dog:
    stomach_full_percentage = 20
    def is_hungry(self):
        if(self.stomach_full_percentage < 30):
            return True
        else:
            return False

snoopy = dog()
print(snoopy.is_hungry())
```

An import method to add to a class is the ability to sort instances when compared to one other. By convention, a way to do this is to contract an `__lt__` method, which evaluates whether one class is less than another class. We have added this method to the new version of the `dog` class. The method takes as arguments: itself and another object of the same type (it then checks whether the arguments passed are indeed of the same type). The method sorts dogs by their ages. We create two dogs, to which we allocate ages and then sort using the `__lt__` method. Running the script confirms that `snoopy` is older than `scooby`.

```
class dog:
    def get_age(self):
        return self.age

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception(
                'Incompatible argument to __lt__: ' +
                str(other))
        return self.get_age() < other.get_age()

snoopy = dog()
snoopy.age = 9

scooby = dog()
scooby.age = 6
```

```
print(snoopy.__lt__(scooby))
```

```
>>>
```

```
False
```

Initialisation Methods

When creating a new class, it is often useful to set (or **initialise**) its variables at time of creation. This is done using a special initialisation method: `__init__`. This is the usual way to assign values to all fields in the class (even if they are assigned to None). By convention and ease of use, the `__init__` method should be at the top of the code in a class.

You will see we have rewritten the dog class below, but now with an `__init__` method that sets the dog's age. As you can see, we then create an instance of a dog called snoopy with an age initialised to 10 years old.

```
class dog:
    def __init__(self, data):
        self.age = data

    def get_age(self):
        return self.age
```

```
snoopy = dog(10)
print(snoopy.get_age())
```

```
>>>
```

```
10
```

String Methods

Sometimes it is useful to be able to print a class to the screen to read its contents. To be able to do this, you need to write a method that defines how the output should be displayed on printing. There are special Python methods named `__str__` and `__repr__` explicitly for this purpose. The `__str__` will be returned after calling print, whereas `__repr__` would be returned by the interpreter.

If you look at the new version of the dog class printed below, in which the name of the dog is set during the initialisation step. Passing the instance of the class (dog1) to the interpreter – or indeed printing the class – causes the memory location to be returned.

```
class dog:
    def __init__(self, data):
        self.name = data

dog1 = dog("Snoopy")
print(dog1)

>>> dog1
<__main__.dog object at 0x0405D6B0>
>>> print(dog1)
<__main__.dog object at 0x0405D6B0>
>>>
```

However, after adding `__init__` and `__str__`, a human-readable name printed to the screen, which is defined within the class.

```
class dog:
    def __init__(self, data):
        self.name = data

    def __str__(self):
        return 'Dog:' + self.name

    def __repr__(self):
        return self.name

>>> dog1
Snoopy
>>> print(dog1)
Dog:Snoopy
```

Modification Methods

So, we have methods that access fields within a class. We also have methods that can modify fields within a class. In the example below the dog's mood by default is set to "Sad". However, the modification method `set_mood` will adjust the mood of the dog. In this example, we change the mood of the dog from "Sad" to "Happy" by using the modification method.

```
class dog:
    def __init__(self):
        self.mood = "Sad"

    def get_mood(self):
        return self.mood

    def set_mood(self, data):
        self.mood = data

dog1 = dog()
print(dog1.get_mood())
dog1.set_mood("Happy")
print(dog1.get_mood())
```

Additional Methods

In addition to the methods described above, there are **action methods**, which will exert some kind of effect outside their class. There are also **support methods** that are used internally within the class, to assist methods that interact with code outside that class. The code is subdivided in this way for readability and preventing the re-use of the same chunks of code. Remember earlier in the course we mentioned how it is often useful to break down large functions into several smaller functions. Well, the same is true of class methods.

Class Attributes

Up until now we have looked at attributes that work at the level of each instance of a class. Their impact is restricted to their own instance and do not affect the other instances of the same class. In contrast, there are attributes whose scope, or namespace, operate at the wider level of the whole class.

It is quite common and simple need for class attributes is in the recording of the number of instances of a class. Of course, the wider program could keep track of this, but it is much neater if the class itself records this value. The code below (generating a sheep class this time) does just this task.

You will notice there is a top-level field called `Counter`. Fields declared here work at the class-level and by convention begin with a capital letter. Having made a **class field**, we now need a **class method** to modify it. To make class methods, simply follow the standard way of making an instance method but place the special indicator **@classmethod** on the line immediately above the definition. The class method `AddOne` simply increments the `Counter` value by one after being called.

The first sheep instantiated is `dolly`. The initialisation method calls the `AddOne` class method and then assigns the value of the `Counter` to the instance field `id`. Consequently, the `id` of `dolly` will be set to 1. Repeating this process for `flossy` further increment the class field `Counter`, and consequently `flossy` will have an `id` of 2.

```
class sheep:
    Counter = 0

    @classmethod
    def AddOne(self):
        self.Counter += 1

    def __init__(self):
        self.AddOne()
        self.id = self.Counter

    def get_id(self):
        return self.id

dolly = sheep()
flossy = sheep()
print(dolly.get_id())
print(flossy.get_id())
```

```
>>>
1
2
```

Static methods

It is just worth briefly mentioning static methods. These methods are different in that they can be called directly from a class, without the need for creating an instance of that class. This is illustrated in the code below. Similar to before, to make a static method place the special indicator `@staticmethod` on the line immediately above the definition.

```
class Utilities:
    @staticmethod
    def miles_to_km(miles):
        return(miles * 1.60934)

journey = 10
journey_km = Utilities.miles_to_km(journey)
print(journey_km)
```

```
>>> %Run mypy.py
16.0934
```

Static methods are useful when we need to make use of a class's functionality but we will not need that class at any other point in the code. When (or indeed whether) to use a static method is often a case of coding style, but they do help to simplify code.

Inheritance

The concept of **inheritance** is central to object orientated programming and allows programmers to write code much more efficiently. The rationale owes much to the phenomenon of the same name observed in biology, in which organisms with a certain set of traits produce offspring with largely the same characteristics. In OOP, once we have defined a class, we can easily define a **subclass** that automatically "inherits" the code of its parent class (now referred to as the **superclass**). We can then change the properties of the subclass, so while it resembles the superclass in many ways, it also has its own distinct functionality.

This ability of OOP is advantageous as it allows coders to produce objects (remember, all classes are objects) with a wide range of functions with a much-reduced code base. It also prevents duplication of code, which is good since if we subsequently need to make changes, we should only have to make the modification in one place and not in many different locations. The process of inheritances may take place over many generations i.e. it is possible to make a subclass and then make a subclass of that.

To illustrate this idea, let's revisit one of the dog classes we generated previously in the section:

```
class dog:
    def __init__(self):
        self.mood = "Sad"

    def get_mood(self):
        return self.mood

    def set_mood(self, data):
        self.mood = data
```

The class `dog` contains the field `mood` which may be set by the method `get_mood`, or may be modified by the method `set_mood`. The initial value is set to "Sad". As we have seen before, running the following code:

```
dog1 = dog()
print(dog1.get_mood())
```

Will return the results:

```
>>>
Sad
```

Now, let's suppose we want to create a subclass of `dog`. To illustrate this concept of inheritance, let's suppose we want to create a breed of `dog`, for example a `Rottwieler`. The code to do this is actually quite simple and entails using the `class` keyword, followed by the new class name to be made, followed in parentheses by the superclass.

```
class rottweiler(dog):  
    pass
```

So, we have now generated the `rottweiler` subclass (for the code to function correctly, we need to place the keyword `pass` after the indentation, since the contents of the subclass cannot be left empty). The subclass `rottweiler` has inherited the properties of the superclass `dog`.

```
rottweiler1 = rottweiler()  
print(rottweiler1.get_mood())
```

```
>>>
```

```
Sad
```

Inheritance and super()

The previous example with the `dog` and `rottweiler` demonstrates how to make a subclass, but at this stage the benefits of this may not seem apparent. We have simply created a class that, for all intents of purposes, is identical to the parent class. Why not simply instantiate a new member of the `dog` class? Well, we shall now illustrate some of the power of class inheritance.

The `Rectangle` class represents rectangles that we may encounter in the everyday world, or in mathematics. Everything in the code should look familiar. The `__init__` method allows the user to specify the `length` and `width` of the rectangle, which is all that is needed to define this shape. Having instantiated a rectangle, there are a couple of methods at our disposal to report the area and perimeter of any given rectangle.

```
class Rectangle:  
    def __init__(self, length, width):  
        self.length = length  
        self.width = width  
  
    def area(self):  
        return self.length * self.width  
  
    def perimeter(self):  
        return 2 * self.length + 2 * self.width
```

This is all well and good, but what about the special case where the length and width of a rectangle are equal? The above code may work fine in such eventualities, but it is probably easier to have a separate `Square` class to deal with these shapes. Not only does a `Square` class make the code easier to read (since it will be obvious we are working with a square and rectangle), such classes should be easier to instantiate, since we only need to know one side length for a square (as opposed to two side lengths for a rectangle). We could write a separate `Square` class from scratch, but a more parsimonious strategy is to create a `Square` subclass of `Rectangle`:


```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

The first line of code generates the `Square` class and specifies that this will inherit its properties from the `Rectangle` class. On the second line we now need a new initialisation method, since we only need to specify the length of one side of a square. The block of code within the initialisation method comprises one line, where we introduce the keyword `super`. As the name suggests, this is used to refer to the superclass. So this line of code references the `__init__` method in the superclass of `Square` (which is `Rectangle`). We then pass `length` twice to this initialisation method, which is exactly what we want to do, for if we define a rectangle in which the length was the same as the width, we will have defined a square. This process of taking a generalised class and then creating more specific subclass from it is a central concept in object oriented programming.

You will see that we have an `__init__` method in our subclass as well as our superclass. If we require a method in the child (sub) class to do something different from the parent, simply define the method in the child class. The method defined in the child class will take priority over the parent class, and this feature of object orientated programming – known as **overriding** – applies to *any* method.

OOP is a huge area in computing, and although to become an expert there is still a great deal to learn, this chapter and the accompanying examples should make you familiar with the main concepts of this programming schema.

A brief note on creating exceptions

In the exception handling chapter we discussed how to deal with errors and enable programs to fail gracefully. While Python has a wide range of built-in errors, when developing more complex code there may be times when you need to define custom errors. We shall not cover this in detail in this course, but if you ever do this, you will need to understand OOP. Exceptions in Python are instances of the built-in class `Errors`. To create your own error, you would need to import that class and then define your custom error as a subclass.

Chapter 8 – Modules

Introducing modules

By using the Python techniques already described in this course, and by making use of the language's built-in functions and methods, it is possible to perform a wide variety of calculations and manipulations with a small amount of code. This capability is extended substantially by making use of **modules**.

Modules make available to the user a much greater range of data types, functions, and methods. Under the hood, modules are actually files placed in a library directory during Python installation. Modules can also be obtained from external sources and added to Python's library. There are also a wide variety of bioinformatics modules available that can be installed to assist with your data analysis. To import a module, use the import command:

```
>>> import os
```

This will import the `os` module, which allows Python to communicate with your operating system. Now the `os` module is installed it is possible to run functions within this module:

```
>>> os.getcwd()
'/Users/wingetts'
```

The `getcwd()` function returns the user's current working directory. To call a function within a module, enter the module name (e.g. `os`), followed by a dot, then the required function's name (e.g. `getcwd`) and then a pair of round brackets as above.

Sometimes you may want to just import specific names from a module, rather than everything found within that module. The following example shows how to import only `getcwd` from the `os` module. Use this syntax for other importing specific names from other modules.

```
from os import getcwd
```

Importing only what you need often makes more sense as it can be overkill to import a whole module simply for one function, or similar. Importing in this way means you can refer to `getcwd` directly, rather than by using `os.getcwd`. Anyway, don't worry about this distinction, the take home message is that there are many modules available that can be imported by a script to instantly extend the functionality of that script. This chapter discusses some of these modules and once you have become familiar with these, you should be able to use a wide variety of additional modules. Some of these modules have bioinformatics functionality, while others are linked solely computational in nature, extending the capabilities of a Python script, or how it interacts with your computer.

The datetime module

Some programs may need to retrieve the current time and date information, or they may need to manipulate times and dates in some way. The `datetime` module was written specifically for such tasks. Suppose then that you wanted to use this `datetime` module, how would you go about using it? Well, in the first place you should check the documentation for the module, which can be found at: <https://docs.python.org/3/library/datetime.html>

The documentation is quite technical and there will undoubtedly be parts that you don't understand, but with growing familiarity and practice you will be able to understand much of what is being described and

extract from the document all that is required to write your own code. The text below is taken from the documentation (for Python 3.2). The text lists the classes contained within the `datetime` module.

Available Types

class datetime.date

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.time

*An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of “leap seconds” here.) Attributes: hour, minute, second, microsecond, and tzinfo.*

class datetime.datetime

A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.timedelta

A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.tzinfo

An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.timezone

A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

The module is versatile and enables the user to perform a wide variety of calculations. Here are some examples of how to use the module to achieve some commonplace date/time-related tasks. In this first example we have imported the `datetime` module, we then instantiate a `datetime` object using the current time as time stored by the object. This value is then printed to the screen as a single-line timestamp, using the `print` function.

```
import datetime

my_datetime_object = datetime.datetime.now()
print(my_datetime_object)

>>>
2019-12-16 17:40:35.218147
```

It is worth pointing out that all the attributes of a module can be achieved using the `dir` function. You should see that the resulting list of attributes corresponds to that listed in the documentation

```
import datetime

print(dir(datetime))

>>>
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime',
 'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

It is also possible to create a date object that corresponds to a user specified date:

```
import datetime
my_date = datetime.date(1966, 7, 30)
print(my_date)

>>>
1966-07-30
```

In addition, the module can be used to add or subtract dates to determine the amounts of time accumulated in different situations. In the example below, we import the `datetime` and `date` classes. We then create to date objects, set to dates of our choosing. Then, by simply using the minus operator, we are able to deduce the length of time between them.

```
from datetime import datetime, date

my_start = date(year = 2016, month = 7, day = 12)
my_end = date(year = 2019, month = 7, day = 24)
my_term = my_end - my_start
print(my_term)

>>>
1107 days, 0:00:00
```

As mentioned before, when you want to perform a task with an additional module, there is often some degree of research involved. Firstly, you may need to find out if a module that meets your programming requirements actually exists. Typically, programmers will refer to modules they have used previously to see if their required functionality is available. If that proves not to be the case, it is common then to search the Python documentation or using Google (or another favourite search engine) to find a module that appears suitable. Next will be the phase of reading the module's documentation to see how that particular module works, and then finally trying the module out and incorporating it into the codebase.

The exercises for this chapter were designed to help you go through this process yourself. The rest of the chapter briefly discusses other useful modules and how they are typically used. We shall not go into detail with regards to their implementation, since this will be handled as part of the exercises.

The math module

As the name implies, this module provides many trigonometric and hyperbolic functions, powers and logarithms, angular conversions and well as mathematical constants such as `pi` and `e`. Further information on this module can be found at: <https://docs.python.org/3/library/math.html>. This module should be your starting point if you wish to perform some kind of mathematical operation beyond that available to you in standard Python. With what you know already, much of the syntax of the module should be relatively straight forward to pick up, such as the following code used to calculate the base-10 logarithm of 1000.

```
import math

print(math.log10(1000))
```

```
>>> %Run test.py
3.0
```

The sys module

The **sys module** contains information pertaining to the Python implementation being used. Below is a list of commonly used values and examples of the output you would expect to see if you print them to the screen.

sys.argv

Description: this name is a list of strings containing the elements of the command line used to run the script. As we might expect, the first element of `sys.argv` (`sys.argv[0]`) reports the name of the script being run.

Printing to the screen:

```
['thonny.py']
```

sys.modules

Description: This is a dictionary in which the keys name the currently loaded modules.

Printing to the screen:

```
thonny.py ('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn',
'_codecs_hk', '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw',
'_collections', '_csv',...
```

The first part, `thonny.py`, is the name of the script and then the modules are listed between the brackets.

sys.path

Description: A list of the directories into which Python looks for modules following the `import` command in a script.

Printing to the screen:

```
['C:\\Users\\wingetts\\Desktop',  
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\python37.zip',  
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\DLLs',  
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\lib',  
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny',  
'C:\\Users\\wingetts\\AppData\\Local\\Programs\\Thonny\\lib\\site-  
packages']
```

sys.exit()

This is a function that will cause a Python program to exit. It is the standard practice to pass the argument 0 if the program exits without error. Numerical values other than 0 signify some kind of error.

The time module

Although the time module contains many time-related functions, you are most likely to encounter the function: `time.sleep()`. This instructs the Python script to wait for the specified number of seconds passed as an argument to the function. Why would you want to do this? Well, sometimes slowing down a program is a useful way to check output written to the screen at a pace that is readable by a human. In addition, sometimes we may add a delay to a script to ensure some other process has finished before we run the next part of the script.

The optparse module

Maybe you have seen already when running scripts on the command line that they may take as input a series of options, each one prefixed by one or two hyphens. Python scripts may also be passed options this way, and this is made possible by using the `optparse` module. The module enables the user to specify what flags may be used by a script, what (if any) argument these flags take, and the type of arguments associated with each flag. For example, the flag `--length` may only take integers while the flag `--mode` may take a string. What is more, the `optparse` module will automatically generate instructions regarding the script and its options, which can be viewed by specifying `--help` (or `-h`) when running the script.

```
python3 myPythonScript.py --length 500 --mode fast file1.txt file2.txt  
file3.txt
```

The subprocess module

The command line is a versatile and convenient environment in which to manipulate your system and run scripts and software. Since pipeline development (the joining separate processes into a single workflow) is a common use of Python for bioinformaticians, it would be useful to incorporate some of the functionality of the command line into a Python script. The `subprocess` module makes this possible by allowing a shell command to be executed directly from a Python script.

For example, the following Python script could be run directly from the command line, and will print "Hello World" as output.

```
import subprocess

print(subprocess.getoutput('echo "Hello World!"))
```

```
user$ python3 subprocess_example.py
Hello World!
```

The os module

The `os` module is an operating system interface and is most commonly used in Python scripts for interacting with files and directories in the filesystem. The selected example below illustrates how the `os` module may be used to interact with your filesystem.

```
os.chdir(path)
Sets path to the working directory
```

```
os.getcwd()
Lists the current working directory
```

```
os.mkdir(path)
Creates a directory at specified location
```

```
os.makedirs(path)
Creates all the directories in the path specified
```

```
os.rmdir(path)
Removes the specified directory
```

```
os.removedirs(path)
Removes all the directories in the path specified
```

```
os.remove(path)
Delete the specified file
```

```
os.rename(sourcepath, destpath)
Rename the file at sourcepath to destpath
```

```
os.path.dirname(path)    Return the directory name of pathname path.  So, if path =
'/home/User/Desktop/myfile.py', then '/home/User/Desktop' would be returned.
```

```
os.path.basename(path)  Returns the basename of the path.  So, if path =
'/home/User/Desktop/myfile.py', then 'myfile.py' would be returned.
```

The tempfile module

In the future you may write a script that during processing writes out a temporary file to a specified directory. This may sound straightforward, but what happens if you are running multiple instances of the same script? There is a danger that one instance of the script could overwrite a temporary file

created by another instance of the script. The solution may seem simple, such as numbering the temporary output files. However, in reality, these solutions that may seem logical can fail and guaranteeing that output files from one process are not overwritten by another process is not a trivial process. Fortunately, the `tempfile` module handles this all for you.

The glob module

Suppose there is a file or a list of files on your computer that needs processing in some way. Unfortunately, however, you don't know the names of the files in advance (you may only know, for example, that the files are found in a certain folder). You will therefore need some way for your script to search the filesystem, return the relevant filenames and process these. The `glob` module allows you to do this.

To use `glob`, simply specify the pattern you wish to match:

```
glob.glob(pattern)
```

The operation of the pattern matching is similar to using the command line in that it also allows **wildcards**:

| | |
|---------------------|----------------------------|
| * | match 0 or more characters |
| ? | match a single character |
| [agct] | match multiple characters |
| [0-9] | match a number range |
| [a-z], [A-Z], [a-Z] | match an alphabet range |

In the simple example below, the script will identify all files in the current working directory, and then all text files in that folder. The `glob.glob` command generates a list, which is then printed to the screen.

```
import glob

all_files = glob.glob('*')
text_files = glob.glob('*.txt')
print(all_files)
print(text_files)

>>>
['make a better figure.pptx', 'one_hundred_lines.txt',
'subprocess_example.py']
['one_hundred_lines.txt']
```

The textwrap module

Writing code to format text in a consistent, neat and logical manner is surprisingly difficult, but once again Python modules come to the rescue. The `textwrap` module does just that, providing a list of functions to format text in a variety of ways.

Perhaps the most commonly used components of the `textwrap` module are the `textwrap.wrap` and the `textwrap.fill` functions that serve as convenient ways to handle, format and print out long strings, such as encountered in FASTA files. In the example below you will see how a lengthy string

may be converted into a list by `textwrap.wrap`, which may be subsequently printed out using a `for` loop. In contrast, the input may be formatted into a new string using `textwrap.fill`. Both these functions take optional integer arguments specifying the maximum character length of each line.

```
import textwrap

quote = """It was the best of times, it was the worst of times, it was the
age of wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were all
going direct to Heaven, we were all going direct the other way - in short,
the period was so far like the present period, that some of its noisiest
authorities insisted on its being received, for good or for evil, in the
superlative degree of comparison only."""

formatted_quote = textwrap.wrap(quote, 50)
for line in formatted_quote:
    print(line)

print("\n")

print(textwrap.fill(quote , 70))
```

```
>>> %Run subprocess_example.py
It was the best of times, it was the worst of
times, it was the age of wisdom, it was the age of
foolishness, it was the epoch of belief, it was
the epoch of incredulity, it was the season of
Light, it was the season of Darkness, it was the
spring of hope, it was the winter of despair, we
had everything before us, we had nothing before
us, we were all going direct to Heaven, we were
all going direct the other way - in short, the
period was so far like the present period, that
some of its noisiest authorities insisted on its
being received, for good or for evil, in the
superlative degree of comparison only.
```

```
It was the best of times, it was the
worst of times, it was the age of
```

wisdom, it was the age of foolishness,
it was the epoch of belief, it was the
epoch of incredulity, it was the season
of Light, it was the season of Darkness,
it was the spring of hope, it was the
winter of despair, we had everything
before us, we had nothing before us, we
were all going direct to Heaven, we were
all going direct the other way - in
short, the period was so far like the
present period, that some of its
noisiest authorities insisted on its
being received, for good or for evil, in
the superlative degree of comparison
only.

The string module

The **string module** (not to be confused with the `str` data type) contains a range of useful values and functions. For example, it contains all the upper- and lowercase letters in the names **`string.ascii_uppercase`** and **`string.ascii_lowercase`** respectively. There are also names representing digits, punctuation and a range of other string values.

The csv module

If you have ever worked with data file storing text values, the chances are you will have encountered the **comma-separated values (CSV)**. This format stores a table or matrix (with layout similar to that of *MS Excel*), only it uses commas to separate columns. (As may be expected, rows are separated by newlines.) A very similar format uses tabs instead of commas to separate columns; this is known as **tab-separated values** (or **tab-delimited**) format. The **csv module** enables a Python script to read from or write to these types of file.

The zlib and gzip modules

Modern life science data files are often are very large. To assist with their storage, they are often compressed as either zip or gzip files. The `zlib` and `gzip` modules are available for reading from or writing to these file types, respectively. The syntax for their usage is generally similar to that for opening a regular file. One important difference is that data returned from the file will be a sequence of `bytes` (note the `b` before the quotation mark when printing to the screen in the example below.) The `bytes` datatype can be decoded to a string with the `decode` method `decode('utf-8')`. UTF-8 is the type of Unicode format to which the `bytes` should be decoded.

```
import gzip

with gzip.open('test_file.txt.gz', 'rb') as f:
    file_content = f.read()

print(type(file_content))
print(file_content)
print()
file_content_string = file_content.decode('utf-8')
print(type(file_content_string))
print(file_content_string)

<class 'bytes'>
b'Hello.\nI am a compressed file.\n'

<class 'str'>
Hello.
I am a compressed file.
```

Installing Modules and Packages

Up until now the modules discussed are distributed with Python and so can be simply be made accessible via the `import` command. There are many modules, however, particularly in the fields of bioinformatics or some other specialist area of biology, that require downloading and installation. If all goes well, this process should be quite simple thanks to the **pip** installer program, which is included by default with Python binary installers since version 3.4.

Installation

The following command shows how to use pip to install the latest version of a **package** (a collection of modules) and its dependencies:

```
python3 -m pip install SomePackage
```

If you have Python2 installed, you may need to type “python3” instead of “python” to clarify the version of python that should be used; we used python3 here for clarity. The flag `-m` instructs the python to run the module `install`. After running this command, and if everything proceeds successfully, you should see a message similar to: “Successfully installed SomePackage-1.2.3”. (It is possible to install packages using the `pip3` command directly, but we shall use the methods described above.)

It is possible to specify an exact or minimum version directly on the command line. To install a specific version (say 1.1.2) try:

```
python3 -m pip install SomePackage==1.1.2
```

To specify a minimum version (say, 1.0.2), try:

```
python3 -m pip install "SomePackage>=1.0.2"
```

(Don't forget the double quotes as the character ">" may cause unintended behaviour when run on the command line.)

Should the desired module be already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python3 -m pip install --upgrade SomePackage
```

It is worth bearing in mind that up-to-date copies of the `setuptools` and `wheel` projects are useful to ensure you can also install from source archives. To update these, run the command below:

```
python3 -m pip install --upgrade pip setuptools wheel
```

Installation locations

So, where are Python packages installed? Well, to find this out, type the command:

```
python3 -m site
```

This should return information, similar to that listed below:

```
sys.path = [  
    '/Users/wingetts',  
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python38.zip',  
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8',  
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/lib-  
dynload',  
    '/Users/wingetts/Library/Python/3.8/lib/python/site-packages',  
    '/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-  
packages',  
]  
USER_BASE: '/Users/wingetts/Library/Python/3.8' (exists)  
USER_SITE: '/Users/wingetts/Library/Python/3.8/lib/python/site-packages'  
(exists)  
ENABLE_USER_SITE: True
```

The Python packages will be stored in the `sys.path` section, in the path ending `site-packages`. If you take a look in that folder, you should see your installed packages.

It is quite likely, however, that on the system you are using, you will not have administration rights to install the package you desire i.e. you may not write to the `site-packages` folder listed above. Not to worry, you should be able to install packages that are isolated for you by adding the `--user` flag:

```
python3 -m pip install SomePackage --user
```

This package will now be installed in the `site-packages` folder, to which you will have access, listed in the `USER_SITE:` section.

Using `pip` is the most common way to install modules and should work in most instances. However, when this does not work, the module required should give instructions on to how it should be installed.

Virtual Environments

Although not described in this course in detail, you should be aware of **Python virtual environments** that allow Python packages to be installed in an isolated location for a particular application, rather than being installed globally. This means that you can install Python packages expressly for one (or several) particular applications without changing the global setup of your system. You then simply start the virtual environment when trying the application of interest, and then exit when done.

Biopython

Biopython is a set of freely available Python tools for biological computation and as such provides a useful resource for the life sciences community. This may be one of the packages that are not distributed with Python that you will need to install to help you complete some computational task. Of course, with your new-found skills in Python you may be able to write the necessary application yourself, but there is no point re-inventing the wheel and if what you require is already available using Biopython, then this should be a good starting point for your analysis. The Biopython homepage is found at: <https://biopython.org/>

As you might expect, to install the package, enter on the command line one of the following commands (in accordance with your administration privileges):

```
python3 -m pip install biopython
python3 -m pip install biopython --user
```

Below is an example given in the Biopython documentation, which shows how the `SeqIO` standard Sequence Input/Output interface for Biopython may be used to parse a Genbank format file into a FASTA file. In a relatively small amount of code a useful task can be accomplished, highlighting how Python and its large, active community of developers can be harnessed so you can achieve tasks much more efficiently than if you were starting writing code from scratch.

```
from Bio import SeqIO

with open("cor6_6.gb", "rU") as input_handle:
    with open("cor6_6.fasta", "w") as output_handle:
        sequences = SeqIO.parse(input_handle, "genbank")
        count = SeqIO.write(sequences, output_handle, "fasta")

print("Converted %i records" % count)
```

Chapter 9 – Regular expressions

Sequence data – whether that is derived from DNA, RNA or protein samples – constitutes an ever more important and burgeoning area of research. Identifying, extracting or modifying specific sequences from the vast pools of data that is commonly generated in modern experiments is no trivial task. Fortunately, the world of computer science has already developed a toolkit with the pattern matching capabilities suited to this formidable challenge. These tools, which constitute a language in their own right, are known as **regular expressions** (or more familiarly known as **regexes** or **REs**). There is a substantial amount of new and dense material to cover in this chapter, so this section may take some effort and practice to become confident at pattern matching. But do take heart, even though regexes may first appear to be impenetrable lines of hieroglyphics, once you can use them, they will form a powerful and succinct way to scrutinise sequences. Moreover, the same ideas and notation are used in different programming languages (e.g. R, Perl and Java), allowing you to also build skills outside of Python.

Introducing the re module

The Python `re` module provides an interface with the regular expression engine, allowing you to compile regexes and then perform matching operations. The commands listed in the table below allow you to convert string to pattern objects and then perform the desired matching operation.

| Method/Attribute | Purpose |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>match()</code> | Determine if the regex matches at the beginning of the string. |
| <code>search()</code> | Scan through a string, looking for any location where this regex matches. |
| <code>findall()</code> | Find all substrings where the regex matches, and returns them as a list. |
| <code>finditer()</code> | Find all substrings where the regex matches, and returns them as an iterator (whose values may returned element by element in a for loop) |

As you may expect, to use the `re` module, simply enter `import re` towards the top of your Python script.

Simple String Matching with the re Module

So now we have introduced the `re` module, let's try some simple pattern matching of strings. Take a look at the code below in which we intend to identify the HindIII restriction enzyme cut site (AAGCTT) in three different sequences.

```
import re

pattern = 'AAGCTT'
p = re.compile(pattern)

seq1 = 'AAGCTTNNAAAGCTT'
seq2 = 'GGGGGG'
seq3 = 'NNAAGCTT'

print(p.match(seq1))
print(p.match(seq2))
print(p.match(seq3))
```

```
print(p.search(seq3))

>>>
<re.Match object; span=(0, 6), match='AAGCTT'>
None
None
<re.Match object; span=(2, 8), match='AAGCTT'>
```

We firstly import the `re` module. After that, we then compile the regular expression (the pattern we wish to identify) into a **compiled pattern** object (named `p`). We now run the `match` method on this object, taking `seq1`, `seq2` and then `seq3` as arguments. The first time we do this (with `seq1`), we achieve a match and a **match object** is returned. We can tell this has happened by printing out this newly generated object, for we can now see displayed the match string and the position of the match. In contrast, no match was achieved for `seq2` and `seq3`, resulting in `None` being returned. It is quite clear that `seq2` does not contain the HindIII site, but `seq3` does – so why wasn't a match object returned? Well, the method `match` only matches at the *start* of sequence. In contrast, the `search` method allows matches *anywhere* in the sequence, and so the match was successful.

Querying the match object

The match object returned after a successful match has methods which may be executed to retrieve useful information, as listed in the following table.

| Method/Attribute | Purpose |
|----------------------|-------------------------------------------------------------------|
| <code>group()</code> | Return the string matched by the RE |
| <code>start()</code> | Return the starting position of the match |
| <code>end()</code> | Return the ending position of the match |
| <code>span()</code> | Return a tuple containing the (start, end) positions of the match |

The code printed below illustrates this ability to query the match object with these methods. The code is a modification of the previous example, and here we use `search` to identify a HindIII site within our DNA sequence. We then print matched sequences, the start position of the match, the end position of the match by retrieving this information from the match object.

```
import re

pattern = 'AAGCTT'
p = re.compile(pattern)

seq = 'NNAAGCTT'

m = p.search(seq)
print(m.group())
print(m.start())
print(m.end())
```

```
>>>
AAGCTT
2
8
```

You may be wondering at this point as to the usefulness of being able to print the matched sequence; after all we already know the identity of our sequence, since we passed this to our regular expression object to achieve the match. Well, that is true for this simple example, but often we will not know the exact sequence that will be matched in advance since we may be trying to match a range of different sequences. In the next sections we shall describe how to construct these more complex regex look-up terms.

Metacharacters

There are special characters, termed **metacharacters**, in regular expressions that represent other types of characters. The list of metacharacters is:

```
. ^ $ * + ? { } [ ] \ | ( )
```

We shall now provide an overview of how these are used.

Character classes

The metacharacters `[` and `]` are used to define classes of characters. For example `[abc]` means match the letters a, b or c. The character class `[a-c]` will achieve the goal. Similarly, you may match against all lowercase characters with the class `[a-z]`. Metacharacters generally lose their special properties inside the square brackets and instead become literal representations of themselves.

The caret character `^` is used to complement the set. So, for example, `[^7]`, will match any character *except* 7.

On a related note, the pipe character `|` essentially means 'or' within a regex. The pattern `a|b` therefore will constitute a match on either a or b.

Start and ends

When found outside of the character class, the caret (`^`) denotes the start of the string. Similarly, the dollar symbol (`$`) denotes the end of a string. So, for example, the regex `^ABC$` would mean a matching pattern should contain ABC and have nothing either side (i.e. the start character is A, and the end character is B). Or for example, the regex `^pi` would match `pi`, `pike`, and `pill`, but not `spill`. This is because a successful match requires the string to *start* with a p.

Groups

Suppose we want to capture individual components (i.e. groups) of a pattern. For example, let's imagine that a particular identifier comprises a word, then a numerical value and then another word. Well we can do this by creating groups within the regex using round brackets. See the code below:


```
import re

pattern= '^[A-z+)(\d+)([A-z+)$'
p = re.compile(pattern)
seq = 'The6Hundred'

m = p.search(seq)

print(m)
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(3))

>>>
<re.Match object; span=(0, 11), match='The6Hundred'>
The6Hundred
The
6
```

We have developed a more complex regex than encountered before, but once you break it down, it is quite intelligible. The first component of our serial are letters, as represented by the character class `[A-z]`. Since there will be at least one of these letters, we suffix the character class with a `+`. Then we have 1 or more digits, represented by `\d+`. Finally, we end with `[A-z]+` once more, representing another word. Since there should be nothing else within our serial number we ensure that only this pattern was retrieved (i.e. there is nothing either side) by delimiting the regex with `^` and `$`. We now define our groups by surrounding `[A-z]+` and `\d+` with rounds brackets.

To retrieve the value associated with each of the three groups, use the group method of the match object. The value at position 0 will be the whole matched term (i.e. `The6Hundred`), while values 1-3 correspond to each of the predefined groups, proceeding in order, from left to right of the regex.

Backslash

The backslash is commonly used in regular expressions and serves multiple purposes. Firstly, the backslash is often followed by a letter character, and together those two characters represent a whole range of different characters. As seen before, the character pair `\d` represents numerical (base-10) digits. The table below describes the backslash letter pairs.

| Backslash Letter | Meaning |
|------------------|----------------------------------------------------------------------------------------------------|
| <code>\d</code> | Matches any decimal digit; this is equivalent to the class <code>[0-9]</code> |
| <code>\D</code> | Matches any non-digit character; this is equivalent to the class <code>[^0-9]</code> |
| <code>\s</code> | Matches any whitespace character; this is equivalent to the class <code>[\t\n\r\f\v]</code> |
| <code>\S</code> | Matches any non-whitespace character; this is equivalent to the class <code>[^\t\n\r\f\v]</code> |
| <code>\w</code> | Matches any alphanumeric character; this is equivalent to the class <code>[a-zA-Z0-9_]</code> |
| <code>\W</code> | Matches any non-alphanumeric character; this is equivalent to the class <code>[^a-zA-Z0-9_]</code> |

The above pattern matches may be used inside character classes to increase their power.

Another very useful metacharacter is the **dot** (`.`), which matches any character, except newline characters.

Escaping Using Backslashes

You may have noticed a problem with the regular expression we have introduced so far. One such problem involves using the dot (`.`) as a metacharacter. While this is undoubtedly useful, what happens if we want to explicitly match a full stop (or period)? We can't use this character itself, since it will match everything. The solution is a technique known as escaping. Preceding a metacharacter with backslash will cause it to be interpreted literally, rather than as a metacharacter. For example:

```
pattern1 = '.'
pattern2 = '\.'
```

The regular expression `pattern1` will match everything except new line character, while `pattern2` matches full stops. This “**escaping**” technique may be used for other metacharacters, even the backslash itself, as shown that `pattern3` matches the backslash character itself.

```
pattern3 = '\\'
```

Raw String Notation

An alternative to escaping characters (which may become very difficult to read) is to use **raw string notation**. This is quite easy to implement and all that one needs to do is place the character `r` before the string for it be interpreted literally.

For example, to read the text `\\matchme` literally, use the notation: `r"\\matchme"`.

Repetition

Regular expressions allow the user to denote that a phrase needs to be repeated a specified number of times for a match to occur. Specifically, the metacharacters **asterisk** (`*`), **plus** (`+`) and **question mark** (`?`) achieve this in closely related, but slightly different ways (see the table below).

| Metacharacter | Action |
|----------------|-----------------------------------------------------------------------|
| <code>*</code> | The preceding character should be occur zero or more times |
| <code>+</code> | that the preceding character should be occur one or more times |
| <code>?</code> | The preceding character should be occur zero or one times |

The small sample code below should illustrate this functionality. The code creates a regular expression to match the pattern `CAR*T`. As should be expected, matching against the letters `CART` will generate a successful match. Essentially the required pattern is `CART`, although the letter `R` may be “present” zero or more times. In the second pattern match we test whether the word `CAT` matches the specified criteria. It does, since the asterisk means the `R` may be omitted. Removing the letter `R` from `CART` makes `CAT`, and so we create a match object.

```
import re

pattern = 'CAR*T'
p = re.compile(pattern)

letters1 = 'CART'
letters2 = 'CAT'

m1 = p.search(letters1)
m2 = p.search(letters2)

print(m1)
print(m2)

>>> %Run regext.py
<re.Match object; span=(0, 4), match='CART'>
<re.Match object; span=(0, 3), match='CAT'>
```

Another method to denote repetition involves placing two integers between curly brackets: $\{m, n\}$. This repeat qualifier means there must be at least m repetitions, and at most n of the preceding character. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, nor `a////b`, which has four. (Note that `\a` is not a metacharacter and so the backslash will be interpreted literally as a backslash.) In fact, you don't need both the m and n , since by default omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.

So, we have at our disposal different techniques to denote the repetition of individual characters. But these techniques are even more powerful, since they may **used to denote repetition of character classes (rather than simply individual characters)**.

Greedy vs non-greedy matching

In addition to the techniques of specifying repetition, there are a set of related techniques that work in the same way, except these perform non-greedy matching as opposed to greedy matching. The non-greedy qualifiers are the same as those for greedy matching (described previously), except that they are followed by a question mark:

```
*?
+?
??
{m,n}?
```

Greedy matching matches as many characters and possible, while nongreedy matching matches the fewest allowed characters. Please review the code below to see how adding the qualifier `?` to the pattern `Pneu.*s` modifies its matching behaviour.

```
import re

pattern_greedy = 'Pneu.*s'
pattern_nongreedy = 'Pneu.*?s'

p_greedy = re.compile(pattern_greedy)
p_nongreedy = re.compile(pattern_nongreedy)

word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'

m_greedy = p_greedy.search(word)
m_nongreedy = p_nongreedy.search(word)

print(m_greedy)
print(m_nongreedy)

>>>
<re.Match object; span=(0, 45),
match='Pneumonoultramicroscopicsilicovolcanoconiosis'>
<re.Match object; span=(0, 19), match='Pneumonoultramicros'>
```

Compilation flags

Compilation flags allow you to modify how certain aspects of a regular expression works. The more commonly used flags are `IGNORECASE` (`I`) and `MULTILINE` (`M`). The former causes the regular expression to perform case-insensitive matches on text characters. The latter flag modifies the mode of action of the metacharacters `^` and `$`. The caret (`^`) now matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. In a similar fashion, the `$` metacharacter matches either at the end of a string and at the end of each line. Add the appropriate flag when creating the regular expression, for example:

```
import re

pattern= 'AAGCTT'
p = re.compile(pattern)
p_ignore_case = re.compile(pattern, re.IGNORECASE)

seq = 'aagctt'

m = p.search(seq)
```

```
m_ignore_case = p_ignore_case.search(seq)

print(m)
print(m_ignore_case)

>>> %Run regext.py
None
<re.Match object; span=(0, 6), match='aagctt'>
```

You can see how applying the `IGNORECASE` has modified the action of the regular expression, causing `aagctt` to match `AAGCTT`.

Modifying Strings

Regular expressions can also be used to modify strings in a variety of ways. Perhaps the two most commonly used methods to perform such a modification are `split()` and `sub()`.

The `split()` method splits a string into a list, subdividing it wherever the regex matches. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. The regex may also be given a `maxsplit` value, which limits the number of components returned by the splitting process.

Another task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value – which can be either a string or a function – and the string to be processed. Similar to the `split()` method, an optional argument `count` may be passed in the regex specifying the maximum number of pattern occurrences to be replaced.

If you look at the example below you will see code that modifies the last word in our serial ID (from before) into `Thousand`. The regex identifies the last word in the string. We then perform the `sub()` method on the pattern object, passing the replacement value as well. This will replace `Hundred` to `Thousand` in the string.

```
import re

pattern= '[A-z]+$'
p = re.compile(pattern)
seq = 'The6Hundred'

m = p.sub('Thousand', seq)

print(m)

>>>
The6Thousand
```

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 complex 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. In addition, you now have an understanding of object orientated programming for writing more complex software.

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.

www.jupyter.org – many bioinformaticians and computational biologist are adopting Jupyter notebooks to write code and share results in a structured and reproducible fashion.

www.matplotlib.org – a popular resource for using Python to produce graphs and charts.

www.biopython.org – a set of freely available tools for biological computation.

Also, don't forget the Babraham Bioinformatics pages listing available courses and providing training materials: <https://www.bioinformatics.babraham.ac.uk/training>

Happy coding!

The Babraham Bioinformatics Team