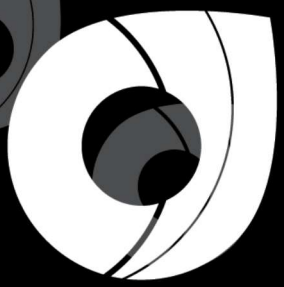
Corndel DevOps Engineering Programme

in association with Softwire

Module 1: General Purpose Coding - Part I



Corndel Digital.

A4N

Introduction

DevOps as a discipline demands proficiency in an extremely wide variety of tools, practices, skills, and techniques. It is perhaps unsurprising that DevOps engineers are motivated to enter the field from an equally vibrant palette of backgrounds and experiences.

The key facilitator of most DevOps principles is the idea of automation: eliminating manual tasks by commanding a computer to perform these tasks for you. The skill of designing such a system is critical, and so we start here. It is, however, also the area in which prospective DevOps students differ the most in terms of existing experience. If you are one of those learners entering this module with some measure of familiarity, feel free to take the material on at the pace most comfortable to you. Even for those partially comfortable using the specific technologies (Python, Flask) covered in these early two modules, there is embedded stretch material: Alongside the sidebars that contain additional information, you will find many opportunities to augment your understanding and stretch your skills in the Additional Material section.

In this module, we'll cover the fundamentals of the **Python** programming language, discuss the importance of source control and show how you can use distributed version control tools, such as **Git**, to enable collaborative software development that can scale to large team sizes. We'll also cover the basics of package management in Python and introduce the **Flask** web framework, which we'll be relying on extensively during this course.

By the end of this module, you should be able to:

- Write your first programs in Python, using flow control, functions and basic data types.
- Create a Git repository to track changes to a codebase, create and merge branches, and push your commits to a remote repo hosted on GitHub.
- Build your first web app in Python using the Flask framework, making use of key features such as routing, HTML templates and form submission.

The reading material is divided into **Core Material** and **Additional Material**. You must complete everything in the **Core Material** section for each module. Having done so, you should undertake the **Project Exercise**. This will demonstrate your mastery of the core concepts of the module. After that, you can undertake the Additional Material. This section will further your skills as a DevOps Engineer, but completing it is not a requirement for the course, so don't worry if the Core Material and Project Exercise is all you manage to complete each module.
It also really helps to try out the examples as you go along, since these will give you hands-on practice with each of the concepts. DevOps skills are gained through practice and getting your hands dirty, so to speak, rather than through reading alone. So - **don't skip the examples.**

Okay, let's get started!

Table of Contents

Core Material

Additional Material

About the sidebars

This is a sidebar!
Yes, this blue box on the side of the page.
In these boxes, you should find material additional to or extending the topic currently being discussed. Content in these sidebars is recommended, but optional. So don't worry if the content in a sidebar is out of your reach for now! Just make a note to revisit it a few months down the line.

Also, this document contains many URLs. You could type or copy & paste them into your browser… but if you're viewing this as a PDF, you can click on the links directly!

Chapter 1

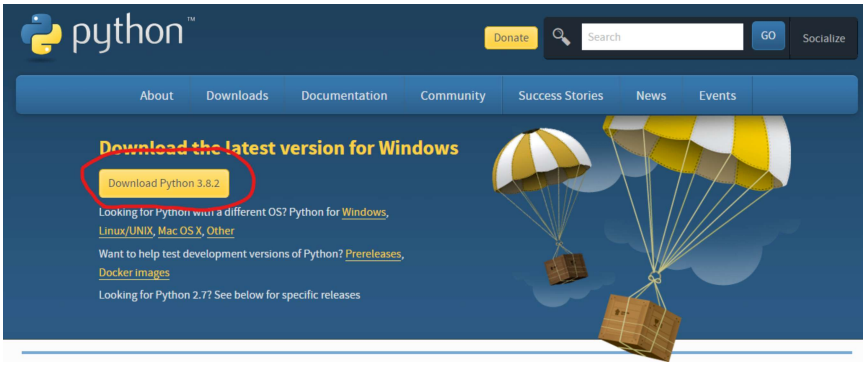
The Fundamentals of Python

Getting Started

Python

Python is a general-purpose scripting language which has skyrocketed in popularity due to its intuitive syntax and ease to learn. Its popularity and flexibility make it a great choice for a first general-purpose programming language to learn. In addition, as a language, it is very suitable for automating tasks, and other purposes related to DevOps.

You can download the latest version of Python from https://www.python.org/downloads/.



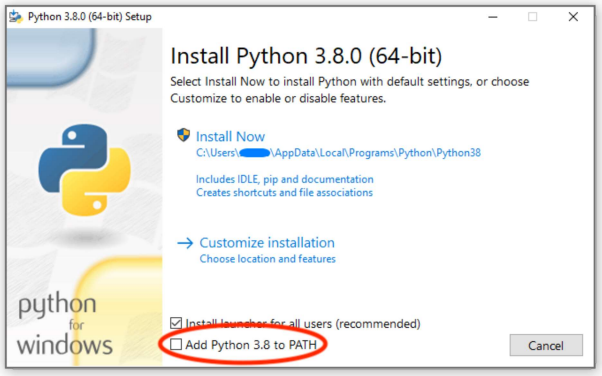
You should download the latest version recommended for your operating system.

If you're using Windows, make sure to select the "Add Python to PATH" option when installing (see screenshot below). This will make it easier to use Python from the command line.

The first step is always the hardest

Getting software set up and installed is by far one of the most error prone processes, especially since everyone's machine can have different settings that could break an installation.

If you're encountering unexpected errors when running these examples, please don't hesitate to contact your technical Professional Development Expert for help.



We will be using Python very soon, so for your **first exercise, please install Python.**

Alongside this introduction to Python, we also have a supplementary set of introductory Python materials on GitHub that visit similar topics in a more hands-on fashion, which can be particularly valuable if you're fairly new to programming in this fashion - getting your hands dirty and learning through doing can be really beneficial when you don't have a great amount of context!

Using the Python REPL

Let's write a few simple expressions in Python and play around with some of its basic features. To start with, open up a command line / terminal window (see the sidebar for more detail). On Windows, you can do this by searching for "PowerShell" or "Git Bash" in the Start menu.

With your terminal window open, and having installed Python, type `python -i` into your terminal and press Enter. You should see something like this — called the **Python REPL**:

Command Lines and Terminals

The topic of command line prompts and terminals will be covered in more detail in **Module 4: Command Line Tasks**. For now, if you are unfamiliar with these applications, you can accept them simply as a text-based alternative to the graphical user interface on a computer.

For example, on Windows, you would normally open Notepad by moving the mouse to the Notepad icon, and double-clicking it. However, in a command prompt window, you can accomplish the same task by typing "notepad" and pressing Enter. In this module, we will be using the command line to run the Python REPL.

You can open a command line on **Windows** by either:

- Pressing the Windows key, typing "powershell", then selecting "Windows PowerShell"
- (Or for older versions of Windows) Pressing Win + R, typing "cmd", pressing Enter

You can open a terminal on **Mac** by pressing Cmd + Space, typing "terminal", then selecting the "Terminal" application.

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019,
19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

This is an interactive window where you can type expressions, and the Python interpreter will evaluate them and show you the result.
Try the following commands and see what you get:

```
>>> 1 + 2
>>> 7 - 5
>>> 2 * 3
>>> 18 / 2
>>> 2 ** 3
```

Hopefully the first four of these are self-explanatory. The final example, which uses **, is the Python version of the exponentiation operator, which you might know as the "power of" operator: it calculates 2 to the power of 3.

Python can work with more than just numbers. It can also work with strings of text surrounded by quotes (usually just called **strings**):

```
>>> 'apples ' + 'bananas'
>>> len('coconuts')
>>> len('grapes')
>>> 'DrAGoNfRuiT'.lower()
>>> 'eLDeRbeRrY'.upper()
```

Try each of those commands out. What does each one do?

- `'apples ' + 'bananas'` attaches the text "bananas" to the end of "apples" to make the text string `'apples bananas'`

REPL Environments

"REPL" stands for "Read-eval-print loop".
A REPL environment for a programming language is effectively a window in which you can type single lines of code, and a runtime environment for the language will read the code you've written as an expression, parse and evaluate the result, then print the result to the screen. It will then provide you another prompt to start the process all over again. Hence, "read-eval-print loop".

This is incredibly useful for experimenting with code, but typically only scripting languages like Python come with REPL environments.

- `len('coconuts')` is programming code for "get the number of letters in the string `'coconuts'`", which returns the number 8. Here, `len` is shorthand for the English word "length" — please note, though, that writing `length('coconuts')` will not work!
- Similarly, `len('grapes')` returns the number of characters in the string `'grapes'`, i.e. the number 6.
- `'DrAGoNfRuiT'.lower()` returns the lowercase (uncapitalised) version of the string `'DrAGoNfRuiT'`, which is `'dragonfruit'`.
- `'eLDeRbeRrY'.upper()` returns the uppercase (capitalised) version of the string `'eLDeRbeRrY'`, which is `'ELDERBERRY'`.

Do they all make sense to you?

Python won't let you do just anything, though!

```
>>> 'apples' + 7
>>> 'bananas' / 'coconuts'
```

Try each of those commands out. What happened?
Each of those examples should have failed with an error message. That's because the Python language doesn't have a definition for adding a number to a string, or dividing a string by another string.

Common sense need not apply

Sometimes a piece of code will not behave as you might expect:

```
>>> 'one apple' + 'one apple'
'one appleone apple'
>>> 'bananas' - 'ananas'
File "<stdin>", line 1
    >>> 'bananas' - 'ananas'
```

Whitespace in Python

Whitespace characters are characters that only represent horizontal or vertical space, such as spaces, tabs, and newlines.
In most programming languages, whitespace is (mostly) ignored.
Python is fairly unique for being a language in which whitespace is significant.
You will understand this better soon, but for now, you just need to make sure that your indentation is consistent.

Compiled vs Interpreted Languages

Languages like Python are "interpreted". This means that programming in Python means writing a text file containing Python code, then to execute your code, running an application (the `python` application) that opens your text file, and *interprets* and executes your code line by line.
Other languages are "compiled". C++ code is written to a text file just as Python code is, but C++ code then needs to be passed through an application (called a compiler) that converts this code into a binary file. You then need to run this binary file to execute your program.

```
    ^
SyntaxError: invalid syntax
```

As a result, it's always a good idea to either:

- test or play around with the code to confirm it behaves as you would expect it to; or
- check the Python documentation (https://docs.python.org/3/) to clarify how a given feature works.

You can now exit the Python REPL by typing `exit()` into your terminal window — or just closing your terminal!

More than just a calculator

Python is far more than just a calculator! The REPL is useful for seeing the result of single expressions, but the power of Python comes from being able to string together multiple expressions in various ways. Let's write our first simple program.
Open up a text editor — we recommend **VS Code** (you can install it from https://code.visualstudio.com/) as it has several features that help with writing Python code — and copy the following lines into a new file. Make sure you copy the lines ***exactly*** as they appear here. Python allows for some flexibility in the number of spaces around operators; however, as we'll see later on, using different indentation has a very specific meaning in Python and can change how your code is executed.

```
users_name = input('What is your name? ')
greeting = 'Hello, ' + users_name + '!'
print(greeting)
```

Once you've copied that, save the file as `greeting.py` (the `.py` extension is standard for Python files).
Before we run it, have a think about what you would expect this program to do. Don't worry if you're not sure at this point – you'll find out soon enough! To run this program, you'll need to open

Visual Studio Code

VS Code is a Chromium-based text editor made by Microsoft, and is distinct from Visual Studio, Microsoft's mainline IDE. It boasts a lot of useful features, and benefits from a large ecosystem of extensions that, for most languages, provide the functionality you expect from IDEs. It is the recommended tool for creating and editing scripts and config in this course.

Programs vs Scripts

So far we've been referring to the text file containing your Python code (e.g. `greeting.py`) as a program.
Arguably, this isn't false, but you may find that it's more common for people to refer to these files as **scripts**.
In general, it's most common to describe a text file containing application code for an interpreted language (like Python) as a **script**, a text file containing application code for a compiled language (like Java) as **source code**, and a compiled executable binary file as a **program**.
This nomenclature isn't strict - some languages are both interpreted and compiled!

another command line, but this time, it needs to be opened *in the folder that you saved your program in*.

To do that in Windows, navigate to the folder where you saved your program in the File Explorer, and then click "File --> Open Windows PowerShell". On a Mac, you would instead need to open a Terminal window (which will by default be placed in your User directory), then type `cd` (with a space at the end), then click the folder you saved your program in, and drag it to the terminal window. The terminal window should then say `cd /Users/yourname/Desktop/DevOpsCourse` or something similar-looking that reflects where you created your folder. Pressing Enter will bring the terminal to that location.
Now, with a terminal in the correct directory, type `python greeting.py`. This tells Python that, instead of starting the REPL, you would like it to run all of the code that can be found in `greeting.py`. When you run it, you'll see something like this:

```
$ C:\Work> python greeting.py
What is your name?
```

That's the program asking you to provide some input. Type your name on the last line, press Enter, and see what happens. The program should give you a personalised greeting!

How does it do that? The first line of code is what lets you enter your name, using the `input` function to prompt you to type something. Once you've entered your name, it saves that information in a *variable* called `users_name`. On the next line of code, it constructs a custom greeting, using your name that it saved. Finally, it *prints* (displays) that greeting on the screen. That's the last line of code, so the program exits at that point.

Navigating folders via command-line

We're going to explore how to use command line utilities in more detail in **Module 4: Command Line Tasks**. For now, it might just be of interest to know that on **Windows command prompt**:

- `cd` With no arguments, this command prints the directory you're currently in (the *working directory*)
- `dir` Lists the contents of the working directory
- `cd foo` Changes directory to the subfolder “foo”

On **Mac**, or **Windows PowerShell**:

- `pwd` Prints the current working directory
- `ls` Lists the contents of the working directory
- `cd` With no arguments, changes the working directory to your user's home directory
- `cd foo` Changes directory to the subfolder “foo”

A comment about comments

Throughout this module you will find code extracts that contain a hash (#) followed by some text, e.g.

```
# Addition
>>> six = 2 + 4
```

This is a **comment.** Comments in code are used to convey information to those who are reading it (for all intents and purposes they never change the behaviour of code).

A # denotes the start of a line comment. This means anything that is written after it *on the same line* will not be processed by the Python interpreter.
If you want to write a multi-line comment then you can use a # for each line, or use three double quotes before and afterwards:

```
# One way of writing
# a multiline comment

"""
Another way of writing
a multiline comment
"""
```

Moving on

As we look in more detail at the features that Python has to offer, you should try out the code examples either using the REPL for short expressions, or by writing a simple program for longer snippets (like we did with `greeting.py`).

When you see code on a line that starts with >>>, that means you should type it into the REPL.

- In this case, code is fed to the Python interpreter line by line.
- This is referred to as **interactive mode** and can be used to debug programs while they are running.

When you see code snippets without >>>, then you should type the code into a file and run it as a program.

- In this case, the whole file is fed to the Python interpreter.
- This is also closer to how a program (or script) would run in production, as it does not need to continuously stop/start to read and execute individual lines.

Feel free to try out your own code too, if you have questions that aren't answered by the material here. Perhaps you'll find yourself curious about what happens if you modify the examples in some way.

If at any point your code gives you an error message, don't panic — you haven't done anything wrong! In fact, thousands of people before you have probably had the same problem. Read the message carefully and see if you can work out why it has happened. If you can't, then try searching online for the error

message; look particularly for results from stackoverflow.com. As an example, take the following error:

```
>>> 'apples' + 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to
str
```

The actual error here is a `TypeError` — it means that we've used the wrong type of data for something. In this case, we've used a number (an "int" in Python speak, short for "integer") where Python was expecting a string (a "str"). Since a number cannot be added to a string, we get this error.

Data Types and Variables

Data Types

In Python, as in most programming languages, the values that you work with can be of various *data types*. By a “data type”, we mean the sort of value that something is — either a number, or a piece of text, or maybe something else entirely. Here, we'll become acquainted with the most common Python data types. We've already briefly seen the examples of `number` and `string`.

Variables

A *variable* is a way of giving a name to a value so that it can be saved and used later. In Python, the way you assign a value to a variable is using an assignment expression like this:

```
>>> my_name = 'John Smith'
>>> my_age = 42
```

Restrictions on variable names

Earlier, we said that variable names are bound by a couple of rules, namely the fact that they need to start with a letter or underscore, and to be composed purely of alphanumeric characters or underscores. There is another restriction. Python has a set of reserved keywords that are often used elsewhere in the language, and so you're not allowed to use them as variable names.

For Python 3.7, this list is:

```
False       del         lambda
None        elif        nonloc
True        else        al
and         except      not
as          finally     or
assert      for         pass
async       from        raise
await       global      return
break       if          try
class       import      while
continue    in          with
def         is          yield
```

You can see this list is by typing the following into your REPL:

```
>>> import keyword
>>> print(keyword.kwlist)
```

These lines of code create two variables (called `my_name` and `my_age`) and assign to the former the `string` value `'John Smith'`, and to the latter the `number` value `42`. The rules for what names you are allowed to use for variables are:

- Variable names can only contain letters, numbers and underscores.
- A variable name must not start with a number.

Other code can then refer to those values using the names we have given them:

```
>>> double_my_age = 2 * my_age
>>> my_name_in_capitals =
my_name.upper()
```

- The first line takes the value assigned to `my_age` (42), multiplies it by 2 and then assigns it to `double_my_age`.
- The second line takes the value assigned to `my_name` (`'John Smith'`), converts it to uppercase (`'JOHN SMITH'`) and then assigns it to the variable `my_name_in_capitals`.

When using the Python REPL, you can see what value is held by a variable by simply typing the name of the variable into the interactive shell. Python will evaluate the expression you have typed, which in this case simply means getting the value of the variable!

```
>>> my_variable = 42
```

```
>>> my_variable
42
```

Common Data Types

Here are the most common Python data types, and ways that you can use them.

Number

Numeric literals

In Python, you can use both whole numbers (known as *integers*) and decimal numbers (known as *floating point* numbers). You write numerical values, including negative numbers, just as you would normally:

```
>>> an_integer_value = 7
>>> a_negative_value = -13
>>> a_floating_point_value = 1.34
```

Numeric expressions

You can also use all of the mathematical operations that you're used to:

```
# Addition
>>> six = 2 + 4

# Subtraction
>>> four = 6 - 2

# Multiplication
```

```
>>> twelve = 2 * 6

# Division
>>> eight = 16 / 2

# Power of
>>> twenty_seven = 3 ** 3

# Brackets to change the order of operations
>>> ten = 2 * (1 + 4)
```

String

String literals

The *string* data type is used to hold pieces of text. They can be written enclosed in either single quotes or double quotes:

```
>>> a_string = 'Ham'
>>> another_string = "Eggs"
```

Character access

You can access the individual characters in a string by using square brackets. Python indexes from zero, meaning that the first letter is considered to be at index `0`, and index 1 is the second letter, and so on.
You can use a negative index to get characters from the back of the string instead of the front. The index –1 is the first character from the back, and so on.

```
>>> a_string = 'Example'

>>> a_string[0] # E
>>> a_string[1] # x
>>> a_string[6] # e

>>> a_string[-1] # e
```

Unicode strings in Python

Unicode (https://www.unicode.org/) is a specification that aims to list every character used by human languages and give each character its own unique code. Earlier programming languages used ASCII (American Standard Code for Information Interchange) to encode text. However, it suffered from a number of limitations, the most significant being that it only really catered to the English language. Unicode is now considered the standard character set used to store text, and most modern programming languages support its use.

Since Python 3.0, strings are stored as Unicode characters. The default encoding for Python source code is UTF-8, so you can simply include a Unicode character in a string literal:

```
>>> message = "I ❤️ Python"
>>> print(message)
```

Python also supports using Unicode characters in variable names, so you could even use emoji in your code (though it's generally not recommended – the meaning of an emoji might seem obvious to you, but not be obvious to others)!

```
>>> a_string[-3] # p
```

String slices

You can obtain a string consisting of the part of a string between a start point and an end point – this is called a *slice* of the original string. The character at the start index is included in the slice, and the character at the end index is *not* included. You should think of the slice as stopping just *before* that character.
If you leave out either the start (or the end) index, then Python assumes that you want to start from the beginning of the string (or go all the way to the end).

```
>>> original_string =
'Supercalifragilistic'

>>> original_string[0:5] # 'Super'
>>> original_string[5:9] # 'cali'

>>> original_string[:5]  # 'Super'
>>> original_string[9:]  #
'fragilistic'
```

Concatenation

You can add two different strings together to get a string made up of both of them - this is called concatenation. You can also join multiple copies of a string to itself using the multiplication operator.

```
>>> first_string = 'Hello'
>>> second_string = 'World'
```

```
>>> first_string + second_string #
'HelloWorld'

>>> na = 'Na'
>>> batman = 'Batman'
>>> 8 * na                #
'NaNaNaNaNaNaNaNa'
>>> 8 * na + batman       #
'NaNaNaNaNaNaNaNaBatman'
```

Length

You can work out the length of a string using the `len` function, which returns the number of characters:

```
>>> my_string =
'Antidisestablishmentarianism'
>>> len(my_string) # 28
```

Further string operations

There are many more string operations you can do, some of which are listed on Python's documentation (https://docs.python.org/3/library/stdtypes.html#string-methods) – but don't worry about learning them all! It's worth seeing this page once just so you know that, in the future, when you are wondering what the best way is to do something in particular with a string, there is probably something listed there which will help you!

Boolean

Boolean literals

A *Boolean* value has exactly two possibilities: `True` and `False`. They are written in Python with capital letters:

```
>>> a_true_boolean = True
>>> a_false_boolean = False
```

Boolean operators

There are some operators in Python that produce a Boolean value as their result. The most important ones are the *equality* and *comparison* operators. The *equality* operators == (equals) and != (not equals) check whether two values are the same as each other. The comparison operators check whether one value is less than or greater than another value:

```
>>> 2 + 3 == 5 # True
>>> 2 + 3 == 6 # False

>>> 2 + 3 != 5 # False
>>> 2 + 3 != 6 # True

>>> 1 + 2 < 3  # False
>>> 1 + 2 <= 3 # True
>>> 2 + 3 > 5  # False
>>> 2 + 3 >= 5 # True

>>> "apple" < "banana" # True, using alphabetical
                       order
>>> "coconut" == "coco" + "nut" # True
```

Why “Boolean”?

“Boolean” is actually named after a person! Boolean algebra was introduced by George Boole, a mathematician from Lincolnshire. Not only do millions of lines of code bear his name, but so does an asteroid, and a crater on the moon, both named after him.
He married Mary Everest (niece of George Everest, for whom Mount Everest is named), and is believed by some to be the inspiration for Professor James Moriarty (the arch-villain in Sherlock Holmes). Apparently, being a mathematician is very exciting!

It is important to note that the equality operator uses *two* equals signs, ==. This is because it is different from the *assignment* operator, with one equals sign =, that is used when you assign a value to a variable:

```
>>> one = 1
>>> two = 2

# This line of code checks if 1 is equal to two
>>> one == two

# This line of code assigns the value 2 to the
variable `one` - not what you want!
>>> one = two
>>> one # 2
```

Combining Boolean expressions

There are three more important Boolean operators in Python: `not`, `and` and `or`. These behave as you expect given the English meaning of these words:

- `not x` is `True` if `x` is `False` and vice-versa
- `x and y` is `True` only when both `x` and `y` are `True`
- `x or y` is `True` if either one of `x` and `y` are `True`, or if both of them are `True`

These can be combined with the other Boolean operators in complicated ways:

```
>>> not (1 == 2)                        # True
>>> (1 == 1) or (2 == 2)                # True
>>> (1 == 1) and (2 == 2)               # True

>>> (2 + 3 == 4) or (4 + 5 == 9)        # True
>>> (2 + 3 == 4) and (4 + 5 == 9)       # False
>>> not (2 + 3 == 4)                    # True

>>> (1 == 2) or (not (2 + 3 == 4))      # True
```

De Morgan's Laws

There's a whole branch of mathematics devoted to Boolean algebra, but you don't need to have studied it to make use (and sense!) of Boolean expressions in programming. There are, however, a couple of transformation laws worth learning which can help to simplify the logic in your code. They are named after **Augustus De Morgan**, a 19th-century British mathematician, and can be expressed as follows:

For any Boolean expressions **A** and **B**,

- not (**A** or **B**) = not **A** and not **B**
- not (**A** and **B**) = not **A** or not **B**

Why is this helpful? It turns out that trying to understand *negative conditionals* — expressions containing lots of "nots" (e.g. "not this or not not that") — is quite mentally taxing, so it's better to avoid this where possible. Using De Morgan's laws, whenever you see an expression like:

```
not is_big or not is_heavy
```

you'll know that this can be simplified to:

```
not (is_big and is_heavy)
```

Operator precedence

The use of parentheses (brackets) in the above is actually entirely optional, but has been left in because it makes it clear what order the various numerical and Boolean operators are working in.

For example, the first line without parentheses:

```
>>> not 1 == 2 # True
```

Could be interpreted as either

```
>>> not (1 == 2) # True
```

Or

```
>>> (not 1) == 2 # Syntax Error
```

However, the operator `not` is of a lower precedence (or priority) than == which means that the former occurs and not the latter.

Another important example is the combination of the `or` and `and` keywords:

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

In this case the `and` operator has higher precedence than the `or` keyword which means the above becomes:

```
>>> (True) or (False and False) # True
```

The complete list of operators grouped by precedence can be found here: https://docs.python.org/3/reference/expressions.html#operator-precedence

Strongly vs Weakly Typed Languages

In programming languages, all values have a data ***type***. It's possible to convert data between different types — a process known as ***casting***. For example, you can convert the number 18 into the string “18”. However, there aren't conversions between all data types (the string “banana” can't be converted to a number).

The difference between a “strongly typed” language and a “weakly typed” one is that a weakly typed language makes conversions between types *implicitly*, while a strongly typed language does not allow implicit conversions (and will raise an error). Python is a strongly typed language, so will raise a `TypeError` if, for example, you try to add together two values of different types:

```
>>> 'Total = ' + 18
Traceback (most recent call
last): File "<stdin>", line
1, in <module>
TypeError: can only
concatenate str (not "int")
to str
```

Instead, you would need to explicitly cast the number to a string using the `str()` function before it can be added:

```
>>> 'Total = ' + str(18)
'Total = 18'
```

Complex Data Types

While the numeric, Boolean and string data types are fundamental, they are also quite limited in what they can express. Python has several complex data types, which are formed out of combinations of these fundamental data types, that are more useful in writing practical programs.

List

List literals

A *list* is a value that contains multiple other values in an ordered sequence. A list can hold any other types of value inside it, and they are typed using square brackets that surround the list, with commas between the items of the list:

```
>>> list_of_numbers = [1, 2, 3, 4, 5]
>>> list_of_strings = ['one', 'two', 'three',
'four', 'five']
>>> list_of_booleans = [False, True, True, False,
True]
```

Lists can also mix type — not everything in the list needs to be of the same type — but be warned that the circumstances under which you'd do this in “real” programming are very rare indeed!

```
>>> mixed_list = [1, 'two', 'three', 4, 'five']
```

You can also have an empty list — that is, a list that contains no values. It is written how you might expect:

```
>>> empty_list = []
```

Where are all the arrays?

Python also provides an **`array`** type (in a module also called **`array`**, which comes bundled with Python by default). This data type shares some similarities with the **`list`** type, with the key difference that you can — *and must* — fix the type of value that can be inserted into the array. Furthermore the choices of type are restricted to numerical types and single characters.

So why would somebody want to use this when a list is far more flexible?

One answer is **performance**; the interpreter won't need to check the type of each item in an array, unlike with a list. Another is **convenience**; if you know that every item in a given array is a number, you won't need to worry about somebody putting a string in there by accident.

Some add-ons to Python (called libraries, which we'll discuss later) provide their own array types with additional mathematical-oriented capabilities, such as being able to divide one array by another (item by item). If you are interested in these sorts of tools, you can check out the **NumPy** (Numerical Python) and **SciPy** (Scientific Python) libraries.

Accessing individual values

Similarly to accessing individual characters in a string, you can access the individual items in a list using the same syntax. Remember that in Python, indices start from 0, so the item at index 0 is the first item, and the item at index 1 is the second item, and so on.

Just like with strings, you can use a negative index to refer to items starting from the end of the list, with −1 as the last item in the list and working backwards.

```
>>> list_of_animals = ['armadillo', 'bear',
'crocodile', 'deer', 'elephant']
>>> list_of_animals[0] # 'armadillo'
>>> list_of_animals[1] # 'bear'
>>> list_of_animals[4] # 'elephant'

>>> list_of_animals[-1] # 'elephant'
>>> list_of_animals[-3] # 'crocodile'
```

Accessing sublists

The syntax used to create string slices is also used to create sublists of a list. A sublist is a portion of a list in between a specified starting element and ending element. Just like with string slices, a sublist *includes* the item at the starting index, but *does not* include the item at the ending index - think of it as the portion of the list starting from the start index, and ending just before the end index.

Again, if you leave out either the start index or the end index, Python assumes that you want to start from the very beginning of the list (or go all the way to the end).

```
>>> my_list = ['These', 'are', 'some', 'words',
'in', 'a', 'list']

>>> my_list[0:3] # ['These', 'are', 'some']
>>> my_list[3:5] # ['words', 'in']

>>> my_list[:3]  # ['These', 'are', 'some']
>>> my_list[3:]  # ['words', 'in', 'a', 'list']
```

Modifying a list

Unlike strings, which are *immutable* (see the sidebar), you can modify the individual items in a list without needing to overwrite the whole thing:

```
>>> my_list = [1, 2, 3]
>>> my_list[0] = 4
>>> my_list # [4, 2, 3]

>>> my_string = 'abc'
>>> my_string[0] = 'd' # This won't work!
```

Removing items from a list

You can remove items from a list using the `del` statement. You can remove either individual items or whole slices at once:

```
>>> list_one = [1, 2, 3, 4, 5]
>>> del list_one[1]
>>> list_one # [1, 3, 4, 5]

>>> list_two = [1, 2, 3, 4, 5]
>>> del list_two[1:3]
>>> list_two # [1, 4, 5]
```

If you know which value you want to remove from the list, but not which index it is at, you can instead use the `remove` method.

Immutability

As we have seen, Python strings are *immutable*, meaning that they can't be changed once they have been created. Instead, if we want to alter a string we need to build a modified copy of the original:

```
my_string = 'd' + my_string[1:]
```

Even though we are storing the result in `my_string`, this is a *different* string object. This same idea — cannot be changed once created, copy to modify — can be applied to more complex data types and structures, and there are other built-in immutable Python data types that we will not cover in this module.

By limiting what we can do to an object, immutability allows useful assumptions and optimisations, particularly for *parallelisation*, where we split a task up between multiple threads running code simultaneously. Functional programming languages such as Haskell take the idea further. The trade-off is extra memory management work for copied and discarded objects, but Python deals with that for us.

Should you make your own Python data structures immutable? This is not required by Python, but is a design decision for your specific project.

This method looks for a particular value in the list, and removes it from wherever it happens to be. If it appears more than once, only the first instance is removed:

```
>>> list_of_numbers = [3, 1, 4, 5, 7, 2, 6]
>>> list_of_numbers.remove(4)
>>> list_of_numbers # [3, 1, 5, 7, 2, 6]

>>> list_with_repeats = [1, 2, 1, 3, 2]
>>> list_with_repeats.remove(2)
>>> list_with_repeats # [1, 1, 3, 2]
>>> list_with_repeats.remove(1)
>>> list_with_repeats # [1, 3, 2]
```

Adding items to a list

You can add an item onto the end of a list using the `append` method, and add an item somewhere in the middle of the list using the `insert` method. When using `insert`, as well as the item you are inserting, you need to first tell Python where in the list you want to insert the item using an index.

```
>>> my_list = [1, 2, 3]
>>> my_list.append(4)
>>> my_list # [1, 2, 3, 4]

>>> my_list.insert(1, 5)
>>> my_list # [1, 5, 2, 3, 4]
```

Concatenation

You can join together two lists to make a longer list using the addition operator, and join together multiple copies of a list with itself using the multiplication operator. This is known as list *concatenation*.

```
>>> first_list = [1, 2, 3]
>>> second_list = [4, 5, 6]
>>> first_list + second_list # [1, 2, 3, 4, 5, 6]

>>> other_list = [1, 2, 3]
>>> other_list * 3 # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Length

You can get the length of a list using the `len` function, just like for strings:

```
>>> my_list = [1, 2, 3, 4]
>>> len(my_list) # 4
```

Dictionary

Dictionary literals

Lists are useful when we want to store a sequence of data in a particular order. Dictionaries are another useful method of storing multiple pieces of data, but structured very differently to lists. They are so named because they work similarly to an English Dictionary that you use to look up words and definitions.
An English Dictionary is a book containing a definition of every word. But it wouldn't be very useful to just have an enormous book consisting of definitions - instead, each definition is associated with the word that it defines, and you can look up the definition you are interested in by first finding the right word, and then looking at the definition alongside the word.
Dictionaries in Python work similarly. They hold various pieces of data, called the *values* (like the definitions in an English Dictionary), and each value is associated with a particular *key* (like the words in an English Dictionary). You write a dictionary out using curly braces, with a key and its value being separated by a colon, and different entries being separated by commas:

```
>>> dictionary_of_favourite_colours
= {'Alice': 'Purple', 'Bob':
'Green', 'Charlie': 'Scarlet'}
```

You can also have an empty dictionary:

```
empty_dictionary = {}
```

You would use a dictionary rather than a list when you don't care about the order of the items, but instead you want to be able to look up a particular item easily using its key.

Dictionaries, Maps and Associative Arrays

What we call **dictionaries** in Python are also known as **maps**, **associative arrays** or **symbol tables** in other programming languages. They are all names for a fundamental data type that stores a collection of ***key-value*** pairs, where each key is unique.

These data types all have a common set of operations you can perform on them:

- **Add** a new key and associated value
- **Remove** a key and its value
- **Modify** the value for an existing key
- **Look up** a value for a particular key

These data types are designed to make these operations very fast and they are best used for data where you want to store and look up values associated with specific labels (their keys).

Accessing individual items

You would use a dictionary rather than a list when you don't care about the order of the items, but instead you want to be able to look up a particular item easily using its key.

The syntax is the same as for accessing an item from a list, but instead of using an index, you use the key. Taking our example from earlier:

```
>>> dictionary_of_favourite_colours
= {'Alice': 'Purple', 'Bob':
'Green', 'Charlie': 'Scarlet'}
>>> dictionary_of_favourite_colours['Alice'] #
'Purple'
>>> dictionary_of_favourite_colours['Bob'] # 'Green'
>>> dictionary_of_favourite_colours['Charlie']
# 'Scarlet'
```

Adding to and modifying a dictionary

To add a new value to a dictionary, you won't be surprised to see that you use the following syntax:

```
>>> favourite_colours = {}
>>> favourite_colours['Alice'] = 'Purple'
>>> favourite_colours['Bob'] = 'Green'
>>> favourite_colours # {'Alice':
'Purple', 'Bob': 'Green'}
```

If you try to add a value to a key that already exists, then it will simply be overwritten with the new value:

```
>>> favourite_colours = {}
>>> favourite_colours['Alice'] = 'Yellow'
>>> favourite_colours['Alice'] = 'Purple'
>>> favourite_colours # {'Alice': 'Purple'}
```

Key restrictions

You can't just use *any* data type for keys in a dictionary. As a general rule, any common data type is acceptable as a key for a dictionary, but many complex data types are not by default (e.g. **lists** and **dictionaries**).

But what happens if you want to use a complex data type for a key?

There are essentially two approaches to achieve this:

- One approach is to convert the data into something called a **tuple** (which works as a key in dictionaries by default).
- Another approach is only applicable to special complex data types called **classes** and involves something called **magic methods** (we'll talk about these in the group call for this module).

Removing items from a dictionary

You can remove an item from a dictionary as long as you know its key, and you do so using the `del` statement:

```
>>> favourite_colours = {'Alice': 'Purple', 'Bob':
'Green', 'Charlie': 'Scarlet'}
>>> del favourite_colours['Bob']
>>> favourite_colours # {'Alice': 'Purple',
'Charlie': 'Scarlet'}
```

Note that a `KeyError` will be thrown if the key is missing when using `del`.

Length

You can work out how many different items are in a dictionary using the `len` function:

```
>>> favourite_colours = {'Alice': 'Purple', 'Bob':
'Green', 'Charlie': 'Scarlet'}
>>> len(favourite_colours) # 3
```

Sets: dictionaries in disguise

Python also provides a data type called a **set**. This is essentially the same as a dictionary, except that there are just keys and no values:

```
>>> basket = {'apple',
'orange', 'apple', 'pear',
'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear',
'apple'}
# duplicates have been removed
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

On the surface this looks a little similar to a list, but there are major differences:

- Sets cannot store duplicates, unlike a list.
- Looking up an item by key in a set is relatively fast compared to looking up an item by value in a list.

Where might this data type be useful? One example would be a shopping list, where you're adding items but don't want to add something again if it's already on the list.

Flow Control

So far, all of the code you have written has been completely sequential – the statements you write are executed in the order you write them, and are evaluated straight away. The real advantage of programming, though, isn't just being able to execute one line of code after another. Programming is at its most powerful when you utilise a concept called *flow control* – a concept that means your program could take one of several courses of action depending on certain conditions, a bit like a flowchart.

Elements of Flow Control Statements

In Python, there are several types of flow control statement. Most of them consist of two important parts – the *condition*, and the *code block*.

Conditions

A *condition* in Python is any Boolean expression – that is, anything that evaluates to either `True` or `False`. It could be a simple variable that we know to hold a Boolean value, or it could be a more complicated expression involving several Boolean operators. The following could all be used as conditions in a flow control statement:

- True
- 1 < 2
- 1 > 2 or 3 > 4

Code Blocks

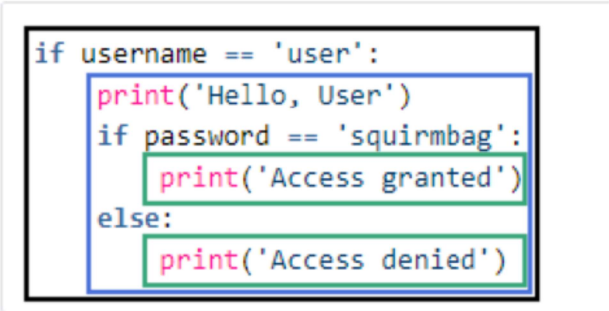
In Python, lines of code can be grouped together in blocks. Python marks blocks of code using *indentation* — that is, by putting spaces at the start of each line in a block. Lines with the same level of indentation belong to the same block, and you start a new code block within the current block by *increasing* the indentation level. When you want to return to the outer code block, simply return the indentation level to match the outer block.

In Python, it's conventional for programs to begin with no indentation, and for each nested code block to have four more spaces of indentation than its parent block.

Can you identify the blocks in this example?

```
if username == 'user':
    print('Hello, User')
    if password == 'squirmbag':
        print('Access granted')
    else:
        print('Access denied')
```

In this example, the code blocks are:

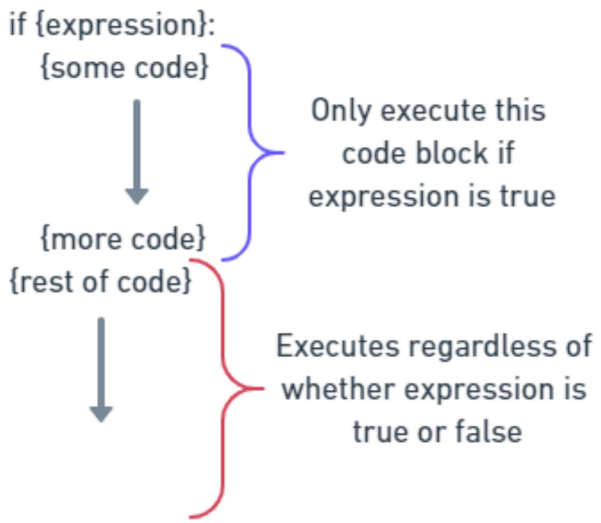


Flow Control Statements

If Statements

The most common type of flow control statement is the `if` statement. An `if` statement's code block will be executed when the if statement's condition evaluates to `True`, and will be skipped otherwise.

An `if` statement in Python consists of the following:



In order, it has:

- The `if` keyword
- An expression
- A colon
- A code block, starting on the line after the expression

What will be the output of the following code? Run it to check your understanding.

```
if True:
    print('First if statement')

if False:
    print('Second if statement')

if 1 + 2 == 3:
    print('Third if statement')

if 1 > 2 or 3 < 4:
    print('Fourth if statement')
```

Answers by statement:

1. This is printed.
2. This is not printed.
3. `1 + 2 == 3` evaluates to `True` so this is printed.
4. `3 < 4` is true so this is printed.

Else Statements

The code block of an `if` statement can, optionally, be followed by an `else` statement. The `else` acts as a counterpart to the `if` — it will be executed exactly when the `if` statement's condition is `False`. An `else` statement doesn't have its own condition, because it relies entirely on the resolution of the `if` condition.

An `else` statement simply consists of the keyword `else` and a colon, and then a code block starting on the next line.

If we add `else` statements to each of the `if` statements from above, what will be the output of this code?

```
if True:
    print('First if statement')
else:
    print('First else statement')

if False:
    print('Second if statement')
else:
    print('Second else statement')

if 1 + 2 == 3:
    print('Third if statement')
else:
    print('Third else statement')

if 1 > 2 or 3 < 4:
    print('Fourth if statement')
else:
    print('Fourth else statement')
```

Answers by `if/else` statement:

1. First **if** statement.
2. Second **else** statement.
3. Third **if** statement.
4. Fourth **if** statement.

Elif Statements

The `if` and `else` statements are great for when you have two possible courses of action to choose between. They are enhanced further by introducing the `elif` statement.

An `elif` statement looks exactly like an `if` statement, but using the keyword `elif` instead of `if`. It can only follow an `if` statement or another `elif` statement, as in the following example.

```
if age < 2:
    print('You are a baby')
elif age < 18:
    print('You are a child')
elif age < 100:
    print('You are an adult')
else:
    print('You are really old!')
```

In an example like this, which begins with an `if`, followed by multiple `elif` statements, the important thing to remember is that at most one of the code blocks will execute. Each condition is checked in turn, and when one of the conditions evaluates to `True`, the corresponding code block is executed, and all subsequent blocks are skipped. If none of the conditions evaluate to true, then the code block belonging to the `else` statement on the end is executed, if there is such a statement.

Can you tell the difference between the following two examples? Run them to check your understanding.

```
age = 16

if age < 18:
    print('You are a child')
elif age < 100:
    print('You are an adult')
else:
    print('You are really old!')
```

```
age = 16

if age < 18:
    print('You are a child')
if age < 100:
    print('You are an adult')
else:
    print('You are really old!')
```

While Loops

Using a `while` loop, you can make a block of code execute over and over again. As long as the condition in the `while` statement continues to evaluate to `True`, the code block will continue to execute.

A `while` statement consists of:

- The `while` keyword
- A condition
- A colon
- A code block beginning on the next line

The only difference syntactically between a `while` statement and an `if` statement is that the keyword `if` is replaced by `while`. They behave very differently, though! The `if` statement checks its condition only once, and executes the code block once if the condition is `True`. On the other hand, the `while` statement checks its condition multiple times — once at the start, and once after each execution of the code block — and for as long as the condition continues to be `True`, it will execute the code block over and over again.

What does this program do?

```
times_run = 0
while times_run < 10:
    print('Hello!')
    times_run = times_run + 1
```

How about this one?

```
password = ''
while password != 'secret':
    password = input('What is the password?')
print('Access granted')
```

Break and Continue Statements

Sometimes, we want to change the way that a `while` loop runs. There are two keywords in Python that let you do this: `break` and `continue`.

If your program encounters the keyword `break` (on its own line) within the body of a `while` loop, then when that line gets executed, it completely exits the `while` loop — even if the loop's condition is still `True`.

On the other hand, the `continue` keyword tells the program to stop the *current* iteration of the loop, and go back to check the condition again. If the condition is still `True`, then the loop will continue with the next execution.

Here are two examples of using the `break` and `continue` keywords. Run them and try to follow what is happening in each case.

```
while True:
    password = input('What is the password?')
    if password == 'secret':
        break
print('Access granted')

while True:
    username = input('What is your username?')
    if username != 'admin':
        continue
    password = input('What is the password?')
    if password == 'topsecret':
        break
print('Access granted')
```

For Loops and Ranges

Whereas the `while` loop keeps looping for as long as its condition is `True`, a `for` loop is for running some code a certain number of times.

A `for` loop consists of the following parts:

- The `for` keyword
- A variable name (it doesn't need to exist yet)
- The `in` keyword
- An iterable (something consisting of a collection of values, e.g. a list)
- A colon
- A code block, starting on the next line

The variable that you use as part of the `for` loop is often called the *loop variable* (i.e. the second bullet point above). A `for` loop executes once for each item in the iterable you provide it, and in each iteration, the loop variable will be set to the next item in the

Iterating over lists

You'll often want to do something with every element in a list. The process of retrieving and operating on every element in a collection (e.g. a list) is known as "iterating over" its elements.

Python provides a number of ways to iterate over the elements of a list.

- You can use a **for loop**:

```
fruits = ['apple', 'orange',
'pear']
for fruit in fruits:
    print(fruit)
```

- You can use a list comprehension:

```
[print(fruit) for fruit in fruits]
```

- If you also need to refer to the *index* of each element, you can use the **`enumerate()`** function:

```
for index, value in
enumerate(fruits):
    print (index, ": ", value)
```

iterable. For example, the following code will print out the names of all of the Teletubbies:

```
teletubbies = ['Tinky Winky', 'Dipsy', 'Laa-Laa',
'Po']
for name in teletubbies:
    print(name)
```

This example will print each Teletubby's name in order (from `'Tinky Winky'` to `'Po'`).

One very common use of the `for` loop is to iterate over a range of numbers, and this is done using the built-in `range` function. The `range` function lets you specify a starting number and an ending number, and will iterate over all numbers between them — including the start number, but *not* including the end number.

What does this program do?

```
teletubbies = ['Tinky Winky', 'Dipsy', 'Laa-Laa',
'Po']
for index in range(0, len(teletubbies)):
    print('The Teletubby at index ' + str(index) + '
is ' + teletubbies[index])
```

Like `while` loops, the `for` loop can also use the `break` and `continue` statements. The `break` statement ends the loop completely, and the `continue` statement moves on to the next item in the iterable.

References, references, everywhere!

All variables in Python are what are known as **references**. This means that when you assign a variable to a value like this:

```
a_variable = 1
```

The variable `a_variable` doesn't really become 1 — instead, the name `a_variable` becomes an **alias** for 1.

This distinction isn't really important for common data types like numeric literals, but for complex data types like lists, you can end up in the following situation:

```
a_variable = [1, 2]
another_variable = a_variable
# another_variable is an alias
# to the same list as a_variable
a_variable[0] = 3
```

What's the value of `another_variable` ? Is it [1, 2] or [3, 2]? Can you guess?

The answer is [3, 2]. This is because `another_variable` refers to the same underlying list as `a_variable`, so when we changed `a_variable`, we ended up changing `another_variable` as well!

If you really want `another_variable` to refer to a separate list with the same values as `a_variable`, you can write:

```
a_variable = [1, 2]
another_variable=a_variable.copy()
```

Try, Except and Finally

As you've seen already, it's possible for code to throw errors; in particular we've seen:

- **Syntax errors:** this is where code isn't valid Python
- **Type errors:** this is where the type of some data does not fit what is expected from a function or operator — we saw this when trying to add a number to a string: `'hello' + 12`

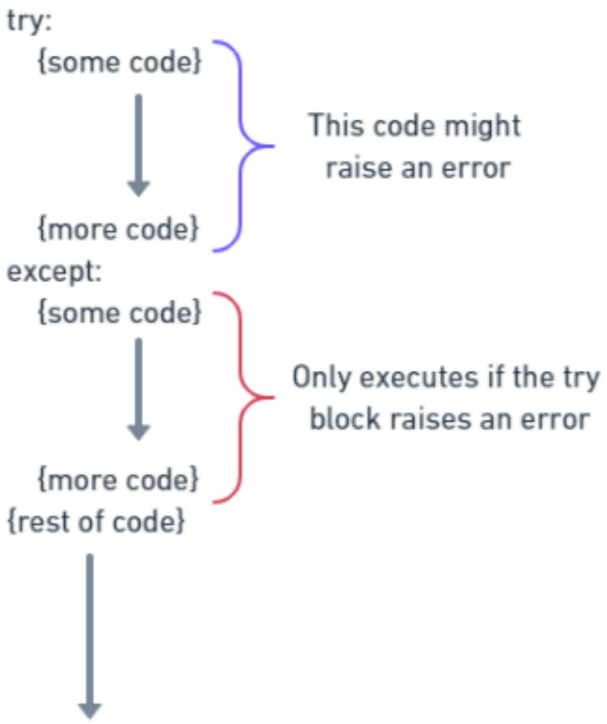
While the former should never occur in production code, it is plausible that the latter might happen if, say, data is being passed in directly by end users.

In the case of type errors, we probably don't want to completely stop execution of the code (which is what happens by default when an error occurs). However, we might want to log what happened, present an error message to the user, or fall back to some other code.

But how do we prevent the interpreter from stopping execution of the code?

The answer is that Python provides special syntax to allow us to “try” to execute some code, “catch” errors and take action.

The simplest possible structure for this is:



And here's a very simple code example:

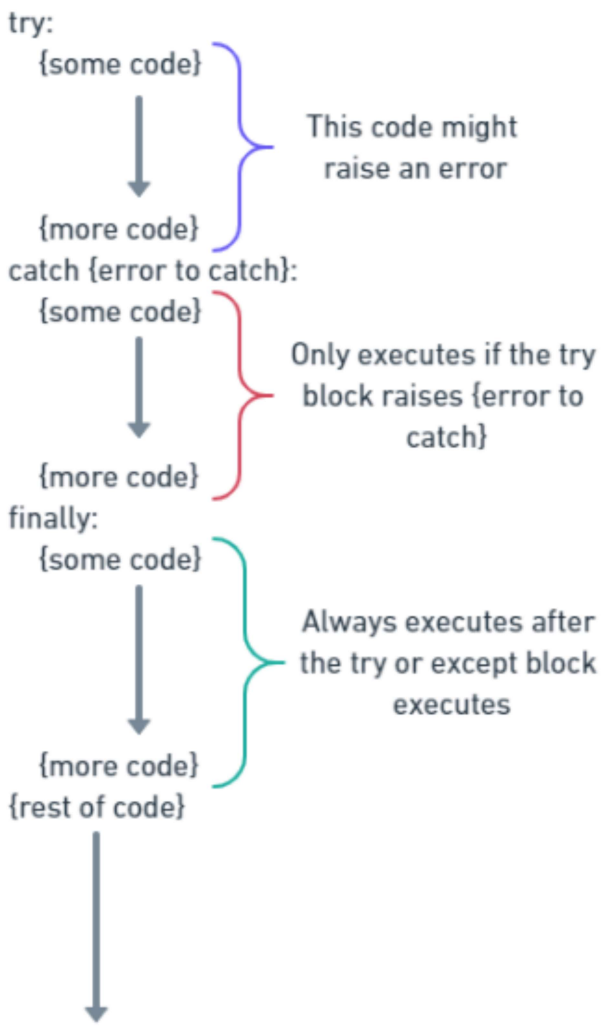
```
number = 12

try:
    msg = 'hello' + number
except:
    print('Something went wrong!')

# The lines below this will still be executed
```

In this example, a `TypeError` will be raised in the `try` block, causing execution of the `except` block, which prints the message "Something went wrong!”.

Here's an example of more complex error handling syntax:



Two things have been added here:

- The option of including a specific error to catch.
- A whole new code block has been added using the `finally` keyword.

And here's an example:

```
number = 12

try:
    msg = 'hello' + number
except TypeError:
    print('Something went wrong!')
finally:
    print('Exiting the try block')
```

This will behave the same as before, except that we'll also print the second message as well.

If we instead wrote:

```
number = 12
dict = {}

try:
    dict['Apple']
except TypeError:
    print('Something went wrong!')
finally:
    print('Exiting the try block')
```

The following would happen:

1. A `KeyError` would be raised in the `try` block.
2. The `except` block would ***not*** be executed as it only catches a `TypeError`.
3. The `finally` block would then execute.
4. The original `KeyError` would be raised.

Running the previous example would produce output similar to the following:

```
exiting the try block
Traceback (most recent call last):
  File ".\greeting.py", line 6, in <module>
    dict['Apple']
KeyError: 'Apple'
```

A finally block always executes, even if you try to escape!

In nearly all circumstances, the `finally` block will execute (assuming we enter its corresponding `try` block). Even if the `try` block is in a `for` loop and we try to use `break` or `continue`, the `finally` block will still execute before breaking or continuing the loop.

One rare exception is if you write invalid Python in the `try` block, because the interpreter won't be able to see there's a `finally` block to process afterwards.

Functions

Functions as reusable blocks

We've learnt about some of the basic components of Python code, but to go from a simple program to more complex behaviour, we need to start breaking up that program into meaningful blocks that can be used repeatedly, which helps to structure our code as it grows. Consider the following code sample, which formats various items along with their prices:

```
item_name = 'Milk'
price_in_pennies = 85
formatted_price = '£{:.2f}'.format(price_in_pennies / 100.0)
print('Item: ' + item_name)
print('Price: ' + formatted_price)

item_name = 'Coffee'
price_in_pennies = 249
formatted_price = '£{:.2f}'.format(price_in_pennies / 100.0)
print('Item: ' + item_name)
print('Price: ' + formatted_price)

item_name = 'Orange Juice'
price_in_pennies = 110
formatted_price = '£{:.2f}'.format(price_in_pennies / 100.0)
print('Item: ' + item_name)
print('Price: ' + formatted_price)
```

While this code achieves its goal, there's a lot of duplication, making it lengthy and hard to read. Also, if we wanted to change the format of what was printed for each item, we'd have to change that in three different places. Wouldn't it be nice if we could take the duplicated code and convert it into a reusable block? That's where **functions** come in:

```
def print_item(name, price_in_pennies):
    formatted_price = '£{:.2f}'.format(price_in_pennies / 100.0)
    print('Item: ' + name)
    print('Price: ' + formatted_price)

print_item('Milk', 85)
print_item('Coffee', 249)
print_item('Orange Juice', 110)
```

A function is a block of code which only runs when it is called. You can pass data, known as **arguments**, into a function and it can return data as a **result**.

In the above example, we've defined the function `print_item`, which has two parameters (`name` and `price_in_pennies`). This function is then run (or **called**) three times, once for each shopping item.

Functions are a convenient way to divide your code into useful blocks, allowing us to better structure our code, make it more readable, reuse it and save time.

Creating a function

At its simplest, a function definition in Python consists of the following, in order:

- The `def` keyword
- A name for the function
- A pair of parentheses
- A colon
- A code block, starting on the next line

Here's a simple example:

```
def hello_world():
    print('Hello World!')
```

If you run a program containing the code above, what will happen? Try it and see.

You'll notice that nothing is printed. That's because the code block in a function definition is not executed until the function is called. In this case, you'd call the above function as follows:

Function Objects

Functions in Python can be treated just like any other data object. In particular they:

- Have types:

```
def square(x):
    return x * x

print(type(square))
# <class 'function'>
```

- Can be sent as arguments to another function:

```
def apply(func, x):
    return func(x)

print(apply(square, 3)) # 9
```

- Can become part of various data structures like dictionaries:

```
def compose(func1, func2):
    def composed_func(x):
        return func1(func2(x))
    return composed_func

methods = { 'square': square,
'quad': compose(square,
square) }

for method in methods.values():
    print(method(2)) # 4, 16
```

The example above also shows how you can use function objects to build new functions (see Decorator Syntax for an advanced use of this).

```
hello_world()
```

Add a call to the `hello_world` function at the bottom of your program and run it again. You should now see "Hello World!" printed on the screen.

Passing data to functions

Data can also be passed into functions as *arguments*.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. Those arguments can then be used by name in the function's code block.

Have a look at the following function and try to determine what it does:

```
def greet_user(first_name, last_name):
    full_name = first_name + ' ' + last_name
    print('Hey there, ' + full_name)
```

When this function is called, the values for the two arguments are also specified inside the parentheses:

```
# Function declaration
def greet_user(first_name, last_name):
    full_name = first_name + ' ' + last_name
    print('Hey there, ' + full_name)

# Calling the function
greet_user('Jane', 'Doe')
```

Parameters or Arguments?

You may sometimes hear the terms **parameter** and **argument** being used for the same thing: data passed into a function. However, there is a distinction. From a function's perspective:

- A parameter is the *variable* listed inside the parentheses in the function definition.
- An argument is the *value* that is sent to the function when it is called.

In our `greet_user` example function, `first_name` and `last_name` are the parameters. `'Jane'` and `'Doe'` are the arguments passed to the function when it was called.

You will also often see "arguments" shortened to "args" in Python documentation.

What happens if you don't specify both values? Try it and you'll see an error similar to this:

```
TypeError: greet_user() takes exactly 2 arguments (1 given)
```

What happened to the REPL?

From now on we'll be working mostly with Python files rather than the REPL. This is because we're starting to work more and more with multi-line code blocks that are not as easy to copy into a terminal directly.

For each example, you'll need to copy the code into a new file and run it as a program.

Make arguments optional by specifying a default value

As we've just seen, when a function definition contains arguments, ***all*** those arguments must be specified when calling the function. However, there are scenarios where you might want an argument to be optional. You can do this by specifying a *default value* for that parameter:

```
def greet_user(first_name, last_name = ''):
    full_name = first_name + ' ' + last_name
    print('Hey there, ' + full_name)

greet_user('Jane', 'Doe')
greet_user('John')
```

When we call `greet_user` without the second argument, it uses the default value we've specified in the function definition. All parameters can be given default values.

Restrictions on default arguments

Note that a non-default argument cannot follow a default argument, e.g.

```
def greet_user(first_name = '', last_name):
```

would be invalid syntax.

Keyword arguments

In the examples above, the order of the arguments matters when calling the function: you have to remember which is which when passing in values. That's fine for functions that only take a few parameters, but what happens if you have a function with many parameters? It can become confusing what each value means. Python also allows you to send arguments using `key = value` syntax, called *keyword arguments* (as opposed to *positional arguments*). Keyword arguments can also be specified **in any order**:

```
greet_user(first_name = 'Jane', last_name = 'Doe')
greet_user(last_name = 'Doe', first_name = 'Jane')
```

Note that functions can be called with a mix of positional and named arguments:

```
greet_user('Jane', last_name = 'Doe')
```

That might not seem particularly useful in the above example, but becomes very powerful when creating more complex functions.

There's a lot more you can do with positional and keyword arguments — it's one of the things that makes Python so flexible and expressive as a language. If you're keen to find out more, there's an in-depth discussion you can read at https://treyhunner.com/2018/04/keyword-arguments-in-python/.

Types need not apply

Python (as of version 3.5) allows the addition of **return type hints** in function definitions, in the form of an arrow and then the return type, i.e. `-> {type}`

For example:

```
def multiply(a, b) -> int:
    return a * b
```

However, keep in mind that these are completely ignored by the interpreter and can be broken when calling the function, e.g.

```
multiply('hello', 3)
```

which Python will still treat as valid!

Returning data from functions

So far, the functions we've seen have accepted input data as arguments, but haven't returned anything. Functions are able to return a value using the `return` statement:

```
def square(number):
    return number * number

result = square(5)
print(result)
print(square(3))
```

The `return` statement will return immediately from the function, even if there are additional lines below it in the code block. This allows for early return — for example, as part of a flow control statement:

```
def greet_user(first, surname):
    if surname == '':
        print('You are missing a
surname ' + first + '!')
        return
    full_name = first + ' ' + surname
    print('Hey there, ' + full_name)

# This would print 'You are missing a
surname Jane!' (and nothing else)
greet_user('Jane')
```

What does an empty return do?

What happens if your function returns nothing?

```
def empty_return():
    return

variable = empty_return()
print(variable) # ???
```

The answer is that Python returns a value called `None`. This is a special value (somewhat related to `null` in other languages) and is used to indicate a *lack* of value.

Another way of explaining the above is that `return` is just shorthand for `return None`.

Exercise: FizzBuzz+

Introduction

This is a spin on a classic programming warm-up exercise. By now, you should have some experience handling Python, mostly through a REPL. We will now iterate on a basic application until it becomes… well, considerably more convoluted.

Learning Goals

The goals of this exercise are:

- Practice writing Python code.
- Look at implementing the same problem in a few different ways.
- Think about how programs can evolve from simple beginnings to something quite complex.

Setup

1. Download the latest version of Python (https://www.python.org/downloads/).
    - Select the option to "Add Python to PATH" during installation.
2. Download VS Code (https://code.visualstudio.com/) and the Python extension (using the in-app package manager).
    - If you prefer, you can use your alternative Python IDE of choice.
3. Pick a location on your machine to save your work (at Softwire we tend to use `C:\Work`). Create a sub-folder called "Exercises".
4. Create a new folder inside "Exercises" called "FizzBuzz" and open this folder in VS Code / your preferred IDE.

Part 1

Write a Python program that prints the numbers from 1 to 100. If a number is a multiple of three, print "Fizz" instead of the number. If the number is a multiple of five, print "Buzz" instead of the number. For numbers which are multiples of both three and five, print "FizzBuzz" instead of the number.

This exercise should give you plenty of practice using loops and conditional flows. Look out for opportunities to use functions to make your code easier to follow.

While we're here with a simple program, let's take some time to see some of the debugging features of your development environment. The following instructions cover debugging using Visual Studio Code (more info available here) but the instructions will be similar for other IDEs, such as PyCharm.

1. Put a break point on a line early in your code, click the "Debugging" tab on the left, and run your code by pressing the "Start Debugging" button (indicated by a green "Play" icon). Note that the program stops when execution hits that break point. While your program is paused, look at the variables in the pane at the top of the debug window that has opened at the left of the screen.
2. You'll now see a floating control panel at the top of the screen, containing (among others) the items 'Continue', 'Step Over', 'Step Into' and 'Step Out'. Find out the difference between them. You'll use these a lot, so learn their keyboard shortcuts. Until then, maybe write them on a post-it for easy reference. You can also access these commands from the "Debug" menu. Step through your code and notice how the variables in the "Variables" panel change as you go.
3. Hover your mouse over the name of a variable in the editor tab. Notice that a small tooltip pops up showing its value.
4. Find the 'Call Stack' pane. This shows which method is currently executing, and all of the methods that called it. This will be very useful later.

Part 2

FizzBuzz is pretty simple as programs go. But it's interesting to see what happens if you try adding new rules. Work through these in order, adding one at a time. How easy is it? How neat and tidy is the resulting code? Can you make changes to your program to make these sorts of enhancements easier, or cleaner?

- If a number is a multiple of 7, print "Bang" instead of the number. For numbers which are multiples of seven and three / five, append Bang to what you'd have printed anyway. (E.g. 3 * 7 = 21: "FizzBang").

- If a number is a multiple of 11, print "Bong" instead of the number. *Do not* print anything else in these cases. (E.g. 3 * 11 = 33: “Bong").

- If a number is a multiple of 13, print "Fezz" instead of the number. For multiples of most other numbers, the Fezz goes *immediately in front* of the first thing beginning with B, or at the end if there are none. (E.g. 5 * 13 = 65: "FezzBuzz", 3 * 5 * 13 = 195: "FizzFezzBuzz"). Note that Fezz should be printed *even if* Bong is also present (E.g. 11 * 13 = 143: “FezzBong").

- If a number is a multiple of 17, reverse the order in which any fizzes, buzzes, bangs etc. are printed. (E.g. 3 * 5 * 17 = 255: “BuzzFizz")

You will obviously need to display more than 100 numbers in order to test out some of these later cases.

Are there any ambiguities in the instructions?

How much of a mess has your code become — how can you make it clear what's supposed to be happening in the face of so many rules?

Stretch goals

Try these further enhancements if you want to challenge yourself in your spare time.

Prompt the user for a maximum number

Read a value in from the console, then print output up to that number.

Allow the user to specify command-line options

Let the user pass in which rules to implement (e.g. any combination of 3, 5, 7, 11, 13, 17) as a command line parameter (or via some other means of your choice).

If you wanted to go wild and let the user define their own rules, how would you do that?…

Chapter 2

Version Control

What is it and why is it good?

Version control is a system that records changes to a file or set of files over time, allowing you to track changes and recall specific versions later.

Though we often talk about version control in the context of software source code, in reality nearly any kind of computer file can be version controlled. Let's say you have a set of spreadsheets that you update regularly, but you want to keep every previous version of the documents (which you would most certainly want to). In this case, a Version Control System (VCS) would allow you to do so. It allows you to revert selected files (or the entire project) back to a previous state, compare changes over time, see who last modified something that might be causing a problem, and more. Using a VCS also acts like a safety net, meaning that if you mess things up or accidentally delete a file, you can easily recover. Even better, a modern VCS provides all this with very little time and effort overhead, both to set up and to use regularly.

Version control is also sometimes referred to as ***revision control*** or ***source control*** — though the latter deals primarily with text (source) files, as opposed to non-text or “binary” data files, which means that it is easier to compare versions (since differences in text files are generally easier to understand than changes in binary files).

Types of Version Control System

Often the simplest method of version control that people use is to copy files into another directory (and possibly rename them with a version or timestamp suffix, if they remember). This method is popular because it's simple, easy to understand and doesn't require installing and learning new software. But it is also

very error prone! It's easy to forget which directory you're in and accidentally make changes to the wrong file or copy over files you don't mean to — or simply forget to save a version.

This is not a new problem, and VCSs were developed long ago to deal with this issue. The first of these were *local* version control systems that used a simple database to store all the changes to files under revision control, stored alongside the files on the same system.

Local version control works fine when it's just you working on those files. But things become more complicated when you need to collaborate with others (for example, other developers) who are working on different machines. Enter **Centralised Version Control Systems** (CVCSs), which have a single server that contains all the versioned files, and a number of clients that *check out* files from that central server. This has been the standard for version control for many years.

Centralised version control offers many benefits over local VCSs. For example, everyone knows what everyone else on the project is working on, and administrators can control who can do what. However, this setup also has some serious downsides. First, the central server represents a single point of failure: if it goes down, then nobody can share or save versioned changes during that period. If the server's hard disk becomes corrupted (and there aren't proper backups), then the central database is lost and you lose absolutely everything — the entire history of the project, save whatever snapshots people may have on their local machines.

This single point of failure is what **Distributed Version Control Systems** (DVCSs) are designed to avoid. In a DVCS, clients don't just check out the latest snapshot of the files — they check out the whole repository, including its full history, known as ***cloning***.

SCM, VCS, what?

You'll often come across the terms **Software Configuration Management** (SCM) and **Version Control System** (VCS) being used in a similar context, and the two can often be confused as being the same thing, which they aren't. This isn't helped by the fact that the acronym SCM can also be used to mean **Source Control Management** (or Source Code Management), which *is* the same as version control.

Software Configuration Management is a broader term that encompasses all the processes needed to build, package and deploy software. It covers processes to manage *all* the changes of the software: its development, delivery release, bug tracking, software settings, host/network settings, the version/settings of the other software it interacts with, etc., as opposed to VCS, which principally covers versioning a set of files.

Every clone is really a full backup of all the data. As such, if any server dies, any of the clients can copy their repository back to the server to restore it. In addition, the idea of a “central server" is replaced by the concept of having multiple remote repositories you can work with, so you can collaborate with different groups of people simultaneously within the same project.

Notable VCSs

Some of the more well-known tools currently in use are listed below:

Tool	Type	Concurrency	Comments	Free?
Git	Distributed	Merge	Created by Linus Torvalds (of Linux fame).	✅
Subversion (SVN)	Centralised	Merge or lock	Probably the most widely used centralised VCS.	✅
CVS	Centralised	Merge	One of the older VCS tools still in use.	✅
Mercurial	Distributed	Merge	Popular alternative to Git, offering functionality that spans both SVN and Git workflows.	✅
Perforce Helix	Centralised	Merge or lock		❌
Team Foundation Version Control (TFVC)	Centralised	Merge or lock	Part of Microsoft's Azure DevOps / Team Foundation Server (TFS) platform. Not free for commercial use.	❌
ClearCase	Centralised	Merge or lock	IBM product as part of its Rational software suite. Considered quite outdated these days; mainly used by large enterprises with legacy applications.	❌

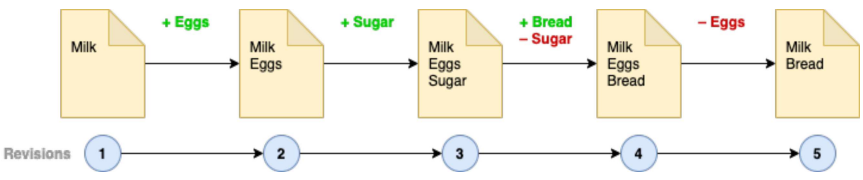
Key concepts in version control

The set of files managed by a VCS, together with the database that records changes to them, is known as a ***repository***, or ***repo*** for short. A VCS can manage multiple repositories, each tracking changes to a different set of files. In many systems, the set of files is defined as a directory on the file system, so that all files that are added, changed or removed from that directory (and its subdirectories) are considered part of that repository.

Changes stored in the VCS database are usually identified by a number or letter code, termed the *revision number*, *revision level*, or simply ***revision***. For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

Basic workflow

The simplest workflow in a VCS is to make changes to a file and to save them as different revisions over time. The process of saving a version is known as ***checking in*** or ***committing*** a file. Consider a text file containing a shopping list. Checking in changes to this file over time might look like this:



The blue dots indicate each revision (numbered 1–5) of the shopping list text file, and the green and red labels show the

Jargon Buster

Repository: an instance of a VCS with its own separate copy of the history of the versioned files. Often shortened to **'repo'**.

Version/revision: a specific version of a file (or files) that were saved to the VCS repository, usually assigned a unique code to identify it ("v1", "v2", etc., or a sequence of hexadecimal digits, e.g. “87a3f38”, known as a *hash*).

Diff/delta: determine the difference between two files, useful for seeing what has changed between revisions. Often displayed with additions in green and deletions in red.

Branch: create a separate copy of a repository for a specific purpose (developing a new feature, fixing a bug, testing, etc). Branch is both a verb ("branch the code") and a noun ("which branch is it in?").

HEAD: a pointer to the currently checked out commit in a git repository.

changes made to the file in each revision: green for added text, red for removed text. For example, we added 'Eggs' to the shopping list in revision 2, and removed them again in revision 5.

Trunks, branches and merges

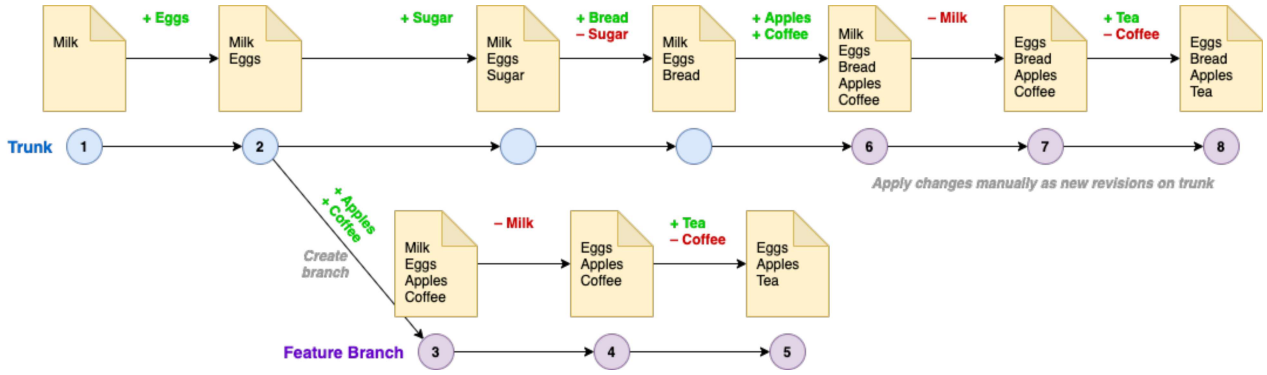
Perhaps the most important concept to become familiar with when using source control in modern software development is that of trunks, branches and merges. Think of the history of all changes to a VCS repository as a family tree, where the ***trunk*** is the main line, tracking its primary history. This is the central version of the files or software, where everyone's changes will eventually come together, and is typically the basis for new releases.

It's possible to create a branch off of that main line — essentially a duplicate of the trunk with a shared history up to a certain revision. This becomes a stable starting point to make new changes, as a way of avoiding interference from or interfering with other people making separate changes, and as a way of grouping a set of changes for independent testing or review.

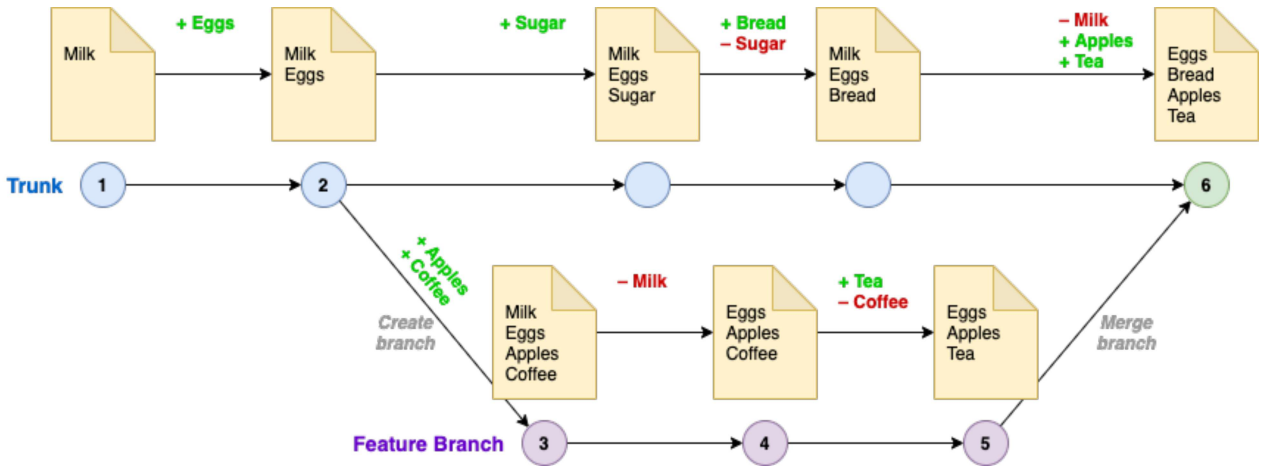
You can then make file changes and save them as new revisions on that branch, but those revisions are absent from the trunk. At the same time, someone else (perhaps another developer on your team) may have made other changes and checked them in to the trunk. At this point, the trunk and your branch have diverged and combining them becomes more of a challenge. Let's say you wanted to apply the changes you had made on your branch to the trunk, making them part of the "primary history" of the repository. There are several ways you could do this — you can apply each of those changes manually on the trunk copy of the repository and check them in as equivalent revisions; or you can ***merge*** the branch into the trunk, meaning that the two histories are preserved but the changes are

combined into a new revision on the trunk (sometimes referred to as a *merge revision*).

Approach 1: Apply revisions from branch individually to trunk.



Approach 2: Merge all changes from branch into trunk as a merge revision.



Many VCS tools are able to perform a merge between branches automatically, determining which changes have been applied on each branch since they split from a common history, then combining those changes together and applying them on the destination branch (the trunk, in the above example). This works well a lot of the time, but when changes have been made to the same area of a file in both branches, such that those changes might overlap — or *conflict* —

then the automatic merge process plays it safe and hands over the reigns to you instead, asking you to manually decide how to combine the overlapping changes. This is called a ***merge conflict***. Fortunately, there are tools dedicated to helping resolve merge conflicts, providing a visual representation of the changes that are being combined from the two branches and giving the option to choose one, the other, or both in the merged output.

Welcome to Git

Git has pretty much become the standard VCS for most new software development projects, because it has some great features that make collaboration easier.

Some Git-related terminology:

- Revisions are referred to as **commits**.
- The main or trunk branch is called **master** by default.
- Checking out the latest changes from a remote repository is called **pulling** from that repository.
- Similarly, checking in your changes to a remote repository is called **pushing** to that repository.

Basic commands

Let's start by creating a new repository. First, create a new folder called `git-playground` to contain the repo (if you are using Windows, we recommend using `C:\Work` as the root folder for creating project and exercise folders for this course). Open that folder in File Explorer, right-click anywhere within it and select "Git Bash Here" from the menu that appears. This will open a Git Bash terminal with that folder set as the current working directory (we'll discuss terminals, command lines and working directories in depth in Module 4).

Installing Git

If you are using Windows and don't already have a version of Git installed (it doesn't come installed as standard), then download and install Git for Windows (https://gitforwindows.org/), accepting all the default options.

If you are using Linux or macOS, follow the instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

On a Mac, you instead need to open the Terminal application, then type `cd ` (with a space at the end), then click on the `git-playground` folder you just created and drag it to the terminal window. Pressing Enter in the terminal will then set that folder as the current working directory.

Let's go ahead an initialise a repository. Type the following into the terminal and then press Enter:

```
$ git init
```

You should see a message confirming that an empty Git repository has been initialised. We now have a VCS to experiment with.

Before you can go ahead and commit some changes, you first need to tell Git who you are, so that your details are added to the commit. To do that, type the following commands into the terminal, replacing `John Doe` with your name and `name@example.com` with your preferred email address:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

You only need to do this once if you include the `--global` option, because then Git will always use that information for anything you do on your system.

Now we need a file to commit. Create a text file in that folder (it doesn't matter what the contents is) and save it as `file_a.txt`

A note about notation

Within the terminal, you should see a cursor with a **$** symbol just to the left of it (and possibly some other characters before it). This is called a **command prompt**, and indicates it is ready to accept commands.

In this course, whenever we list commands that should be entered in a terminal, we'll indicate that by prefixing the command with a **$** symbol, but **don't include the $ when typing the command!**

The .git folder

Git stores its repository state in a hidden folder called `.git` that it has created inside our `git-playground` folder.

As long as you're in `git-playground` or a descendant folder, then Git commands will act on this repository.

Let's check the status of our repository — type the following into the terminal:

```
$ git status
```

You should see a message similar to this:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will
be committed)

    file_a.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

This tells us a number of things:

- What branch we are currently on (`master` by default)
- Whether there are any changed or untracked files (in this case, `file_a.txt`)
- The status of the **index**, or staging area for changes before they are committed (nothing has been added to commit yet, but Git gives a helpful suggestion of how to do so)

Before you can commit changes, you need to **stage** those changes by adding them to the **index**. In this case, we want to add the new file to be tracked:

```
$ git add file_a.txt
```

`git add` and the index

Note that `git add` means "add this file and its changes to the index" — i.e., include these changes in the next commit.

By coincidence, we are adding a new file, but we would also need to `git add` an existing file that we had modified, not just new files.

To delete a file and stage that deletion so it appears in the index, you would use the `git rm` command instead.

If you run `git status` again, Git will tell you that there are now changes to be committed and lists `file_a.txt` as a new file (appearing in green).

Now that we have staged the change we want to commit as a new revision, let's actually commit it. Type the following into the terminal:

```
$ git commit -m "My first commit!"
```

Git should print out the details of that commit, including the message and what file(s) have changed. Hurray!

Let's not stop there! Next we'll make a change to that text file and commit that, too. Modify the contents of `file_a.txt` and run `git status` again. The status message now indicates that the file has been modified. As above, we need to stage that change before we can commit it, and the command is also `git add`:

```
$ git add –all
```

(Here we've used the `––all` option to add all outstanding changes listed by `git status` to the index, rather than adding each file individually as we did in the example above.)

Now commit the change and run `git log` to see a history of all changes (revisions) in the repository:

```
$ git commit -m "Update text file"
$ git log
```

Writing good commit messages

Git doesn't impose any restrictions on the message that you add when committing changes, but writing good commit messages is important, as these will form the historic record of all changes to a project. A change might seem obvious now, because you're aware of the context, but imagine coming back to that change a year from now: chances are you'll have forgotten all about what you were working on at the time and all you'll have is that message to make sense of it! You can find some good guidelines on how to write useful commit messages here: https://chris.beams.io/posts/git-commit/

The log lists the revision ID (called a “commit hash” in Git), along with the author, date and message for each commit, similar to the following:

```
commit 35189d8b7744d72b341e6cabd43869e0c232ac53
(HEAD -> master)
Author: John Doe <johndoe@example.com>
Date:   Wed Apr 15 10:30:00 2020 +0100

    Update text file

commit 79cd50d1348192a3fe085fd955b7f72875fdad0f
Author: John Doe <johndoe@example.com>
Date:   Wed Apr 15 10:15:00 2020 +0100

    My first commit!
```

Connecting to a remote repository

So far, all these changes have been made to a local repository on your machine, but in most cases you will want to share them with others — whether that's your team mates on a shared project, or with a wider developer community. This is achieved in Git by connecting your local repository to one or more *remote repositories*. But before you can do that, you first need to create a remote repository...

One of the most popular cloud-based hosting services for Git repositories is GitHub. We'll be using it extensively during this course to share project exercises and example code. Go ahead and create an account if you don't have one already. Once signed in, click the green "New" button to create a new repository. For consistency, call it `git-playground` (the same name as your local repo). GitHub gives you the option to initialise the repo with a readme file, or add a gitignore or standard open-

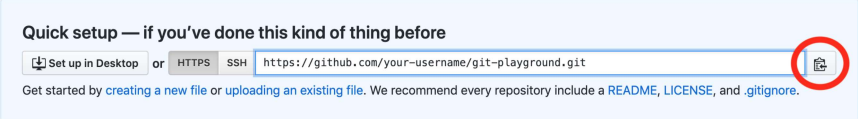
Commit hash identifiers

Note that the commit hash identifiers in your log will be different: this is by design. They are a function of all of the contents of the files in each commit, the timestamp of the commit and your name and email address (plus a few other values), and are computed as a SHA-1 cryptographic hash — hence the name. It is extraordinarily unlikely that two different commits will ever have the same hash (a “hash collision”), even when working on two separate clones of a single remote repository. Hence these hashes will remain unique identifiers for each commit even when many commits are merged back into a single remote repository from many different sources.

However these 40-digit hexadecimal identifiers are long and unwieldy. Therefore, we typically work with branches and tags, which serve as human-readable aliases to the longer hash identifiers. Git will often just quote the first seven digits of a hash as long as that is enough to be unique (try using `git log --oneline` to see this in action).

source software license, but we'll skip those steps for now (you can always add them later if you want).

Once you've created the repository, you'll see a page giving you various setup options for using it. The most important bit of info is the URL of the repository, which you'll use to connect it to your local repository:



Click the copy button (circled in red on the above screenshot) to copy the address to your clipboard, then add it as a remote to your local repo using the following command in the terminal, replacing the URL with the one you copied from GitHub:

```
$ git remote add origin https://github.com/your-username/git-playground.git
```

You can confirm that a new remote repository has been added using the `git remote --verbose` command, which will list back the URL you have just configured. By convention, we usually name the main remote repository `origin`, since it's the one everyone will be using as the origin for their local copies. Now that we've added the GitHub repository as a remote, we can **"push"** our local changes to it:

```
$ git push --set-upstream origin master
```

You might be prompted for your GitHub username and password if you haven't used them previously with Git. Once authenticated, you should see a message similar to the following:

Excluding files from source control with .gitignore

You won't always want to commit all files in your working directory into your VCS. For example, Python will compile scripts that you "import" into `.pyc` files; Java compilers generate `.class` files and `.jar` files, and these are binary files which do not need to be (and should generally not be) version controlled. You might also want to exclude test reports, temporary output files and per-user IDE settings files.

To tell Git to automatically ignore these files, create a `.gitignore` file in your working directory — see the link in the main text for more information. You can also find example `.gitignore` files for many scenarios online, e.g. at gitignore.io

```
Counting objects: 4, done.
Delta compression using up to 2
threads.
Compressing objects: 100% (2/2),
done.
Writing objects: 100% (3/3), 266
bytes, done.
Total 3 (delta 1), reused 1 (delta 0)
To https://github.com/your-username/
git-playground.git
   0e78fdf..e6a8ddc  master -> master
```

Note that the `--set-upstream` option is only needed the first time you push to a new remote repository, and tells Git to link your local `master` branch to the `master` branch of the `origin` remote. Once you have made more commits, you can upload them to the remote repository in the future using just `git push`, without having to specify the remote repo and branch name.

If you refresh the repository page on `github.com`, the file you committed previously should now be listed, and the "commits" tab should show the two commits that you've just pushed. This shows that the master branch is synced properly between the local and remote repositories.

The opposite of pushing to a remote repository is **pulling** from it, which retrieves any new commits that might have been pushed there by someone else on your team and updates your local version of the repo to include them. If you run `git pull` now, you'll see a message saying there are no new commits, because you're already up-to-date.

It's possible to add additional remote repos using the `git remote add` command, specifying a name and URL for the new remote. You can then pull or push changes to that remote repository by specifying its name when using `git pull` or `git push`.

Using branches to make team development easier

Congratulations! You can now make commits and push them to a remote repository. However, when you're working as part of a team on the same project, having everyone commit and push their changes directly to the master branch can soon lead to headaches when you try to reconcile the differences between everyone's local version of master.

For that reason, one of the most common workflows that is followed when using Git is to create a new branch, make your changes as a series of commits on that branch, and then merge that branch into `master` when done. This has a number of benefits, allowing related changes to be grouped together, and simplifying the process of integrating changes from different developers.

We'll walk through the steps of that workflow, creating a branch called `my-feature` to add

some commits that we'll then merge into `master`. First, let's see what branches currently exist:

```
$ git branch --all
```

(The `--all` option lists all *remote* branches, as well as local ones; that is, any in remote repositories, such as `origin`.)

The output will look something like this:

```
* master
  remotes/origin/master
```

The asterisk next to `master` in the first line indicates that we are currently on that branch. The second line simply indicates that on our remote, named `origin`, there is a single branch, also called `master`. If you have recently run `git push` or `git pull` then these two master branches will be in sync.

Now that we know how to view branches, it's time to create our first one. To create a branch called "my-feature", type the following:

```
$ git checkout -b my-feature
```

The command should print a message to confirm that a new branch has been created, and that you are now using that branch:

```
Switched to a new branch 'my-feature'
```

Remote branches

The `remotes/origin/master` branch is the position of the `master` branch in the `origin` repository at the point that you last ran `git pull` (or `git fetch`, which is similar to `git pull` but does not automatically update your local master branch with the changes it has fetched) or made a successful `git push`.

It will not automatically pick up changes other developers have merged to the origin repository without running `git pull` or `git fetch`.

You can use `git branch` to confirm that you are, indeed, on the "my-feature" branch (as indicated by the asterisk):

```
  master
* my-feature
```

You can also see this at the top of the `git status` output, and recent versions of Git Bash include the current branch in the command prompt.

Now let's add a new text file, called `feature.txt`:

```
$ echo "This is my new feature." > feature.txt
```

Stage and commit that new file:

```
$ git add --all
$ git commit -m "Add new feature file"
```

If you open the folder in File Explorer, you should see the two files `file_a.txt` and `feature.txt`. However, we've only added the `feature.txt` file as a commit on the "my-feature" branch — what happens if we switch back to the master branch? You can check out (change to) a different branch using the `git checkout` command:

```
$ git checkout master
```

Check the contents of the folder again, and the `feature.txt` file will have vanished! Don't worry, though: Git hasn't lost it, it's just that the current state of the repository has been updated to match the latest revision on the master branch, which doesn't

include the commit we just made to add that file. Switch back to the "my-feature" branch again and you'll see that the file reappears.

Now that we've made our changes on a separate branch and are happy with them, we want to add them back into the master branch, to incorporate them with the changes that other developers on our team have been making. (In reality, your changes would likely consist of multiple commits on your branch, not just the one we've made here.)

The process of moving commits between branches (often from a feature branch to master) is known as **merging**. It's important to remember when merging that we need to be on the branch that we want to merge ***to*** before performing the merge, as the merge will be represented as a new commit on that branch.

In this case, we want to merge from the "my-feature" branch, where we've made our changes, to the master branch. Switch back to the master branch if you aren't already on it (you can use `git branch` to check).

Now, all we have to do is run the merge command, specifying the branch that we want to merge from:

```
$ git merge my-feature
```

If you check the contents of the folder again, you should see both files are now present on the master branch.

The last thing we need to do is to push our latest changes to the remote repository (`origin`):

```
$ git push
```

Merge strategies

When merging changes from a branch into master, Git provides several ways to achieve it, each of which causes the history of commits to appear differently on the master branch once complete:

- **Fast-forward:** the individual commits from the branch are applied on top of master, producing a linear history. This happens when there are new remote changes but no new local changes.
- **Three-way:** a dedicated merge commit is created on master, producing a history where the branch is preserved.
- **Rebase:** this isn't really a merging strategy, but provides a way to "move" one branch on top of another (e.g. master), to allow a fast-forward merge to then be performed.

If you want to read more about this topic, this article (https://www.atlassian.com/git/tutorials/using-branches/git-merge) covers the difference between fast-forward and 3-way merges in detail, and this article (https://www.atlassian.com/git/tutorials/merging-vs-rebasing) provides an in-depth discussion of merging and rebasing, and when it makes sense to use each.

You'll see output similar to following, confirming that your merge into the master branch has been pushed to the remote server:

```
Counting objects: 4, done.
Delta compression using up to 2
threads.
Compressing objects: 100% (3/3),
done.
Writing objects: 100% (3/3), 332
bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/your-username/
git-playground.git
   9af2dcb..53649cf  master -> master
```

Wrapping up

Phew! That was a lot to cover, but we've seen how to:

- create a new repository (both on your local machine and on GitHub)
- commit changes to files
- pull and push changes to a remote repository
- and use branches to make team development work easier.

GUI clients and other tools

We've been using the Git command line tool to interact with the repository. However, there are a number of GUI tools available that make it easier to visualise commits and branch history, review changes and amend previous commits (using Git's interactive rebase functionality).

A few of the most popular GUI clients for Git are:

- Git GUI and gitk (which are installed as part of Git for Windows)
- SourceTree
- GitKraken

You can find a much longer list of available GUI clients for Git listed on their website (https://git-scm.com/downloads/guis).

There are also plenty of GUI clients available for other VCS tools, both those included with the VCS itself (e.g. Perforce Helix Core) and third-party clients, such as TortoiseSVN and SmartSVN.

Finally, many modern IDEs (Visual Studio, VS Code, IntelliJ, Xcode, etc.) also integrate version control functionality, including the ability to review and commit changes, view revision history for open files, push and pull from remote repositories, and even resolve merge conflicts.

Chapter 3

Python libraries and the Flask web framework

Libraries and frameworks

So far we've covered some of Python's core features, which are provided by its built-in data types and functions. However, Python provides a lot more functionality as part of its **standard library**, including access to files and directories, date and time data types, and network communication. These capabilities are available in the form of **modules** which you can access using the `import` statement in your code. For example, the following script imports the `datetime` module and then uses the `today()` function on its `date` object to get today's date:

```
import datetime
today = datetime.date.today()
print(today)
```

Python's standard library is one example of a collection of code that extends a programming language's basic features to provide you with much more complex functionality. These collections of code are known as **libraries** or **frameworks** and are intended to make our lives as developers easier by providing "off-the-shelf" functions that we can use directly, rather than having to write them from scratch ourselves. Libraries and frameworks are essentially reusable code written by someone else, designed to help solve common problems in easier ways.

When writing software programs, chances are you'll make use of many libraries or frameworks so that you can leverage the existing functionality they provide. Many of these libraries won't come installed by default with the programming language you're using, but are instead provided by other developers — together, these make up what is known as an "ecosystem" of libraries and frameworks that are written and made available by the developer community for a particular language.

Python packages

Python has a rich ecosystem of libraries and frameworks that have been written to solve common requirements (such as mathematical calculations, image manipulation, cryptography, etc.). These are known as **packages** and the vast majority are open source and freely available for you to download and use. The process of searching for, installing and uninstalling packages is known as ***package management*** and there are several tools available that let you do this.

The most popular tool for installing Python packages is **pip**, which is included with all modern versions of Python. It provides the core features for finding, downloading, and installing packages from the Python Package Index (PyPI) and other Python package repositories, and is designed to be used from the command line / terminal.

It's pretty simple to install a package using pip:

```
$ pip install some-package-name
```

This will install the latest version of the package (assuming it exists on PyPI). However, it is generally considered best practice to specify precisely which version of a package you want to use for a given project, to ensure that everyone working on the project uses the same version and to guard against inconsistencies (say, for example, a new version of a package changes its behaviour slightly that would break how you are using it). You can specify a version when installing a package, for example:

```
$ pip install some-package-name==1.4
```

Version specifiers

It's possible to specify more flexible version requirements for a package, instead of an exact version.

This is particularly useful when packages follow **semantic versioning** (see the next section), since you can allow pip to use any *compatible* version of a package. What does compatible mean? It means any version where the package behaves in the same way in terms of how you use it. For example, different patch versions (e.g. `1.4.1` or `1.4.3`) guarantee compatibility.

In the example we've seen already, we used the == version specifier to specify an **exact** package version. The following version specifiers can also be used:

`~=`	Compatible version
`==`	Matching version
`!=`	Excluded version
`>=`	Equal or higher version
`<, >`	Lower or higher version (respectively)

The full list of version specifiers can be found here: https://www.python.org/dev/peps/pep-0440/#version-specifiers

Pip also understands more flexible version specifiers that can be used to define a minimum, maximum, or range of versions that you wish to use.

It is common for a software project to make use several or more packages as it grows in complexity. Rather than having to remember to install each of these packages one by one, pip allows you to declare them in a `requirements.txt` file in the project codebase and will install them all for you using the following command:

```
$ pip install -r requirements.txt
```

Semantic versioning

Why do packages have multiple versions? Like software applications, packages may be updated to add new features, change existing functionality or fix bugs. When the developer of a package releases an update to it (a new copy of the package that includes one or more changes), they release that update as a new **version** and assign it a unique identifier, so that each historical release has its own version identifier. That identifier can be any label the owners of the package choose, but typically they will follow a convention so that it's easy to determine which is the older or newer version when comparing two releases.

A common convention adopted by most packages is known as **semantic versioning**. This is a specific format of version identifier intended to indicate how big a change has been made to the package in a given release. An example of a semantic version identifier (also known as a version number) would be `"1.5.2"` . The semantic version format consists of three numbers separated by periods: `major.minor.patch` — where only one of those numbers is incremented in each new release. In the previous example, the major version number is 1, the minor version number is 5 and the patch version number is 2.

Major, minor and patch numbers are intended to convey the extent of the change included in a particular release. For example, incrementing the patch number (e.g. from `1.5.2` to `1.5.3`) would indicate that only bug fixes are contained in the release which do not change the way the package is used. Incrementing the minor version number (e.g. from `1.5.2` to `1.6.0`) might indicate that new functionality has been added, but that the way you use existing functionality remains unchanged (known as *backwards compatible changes*). Finally, incrementing the major

version number (e.g. from `1.5.2` to `2.0.0`) signals that more significant changes have been made to the package that may require you to change how you use it, known as *breaking changes*. Note that when a major or minor version number is incremented, the lesser version numbers are reset to zero, effectively "resetting the counter" on any lesser changes.

Why is semantic versioning useful? It allows developers to make informed decisions about whether to upgrade to a newer version of a package they are using in their application. If a package uses semantic versioning, then developers will know that a release of a new minor or patch version is generally safe to upgrade to, since it should not break how that package is used in their code — essentially, they do not need to change their own code to use the new version and it shouldn't break their application. A release of a new major version, however, would need to be reviewed more carefully before deciding whether to upgrade, to see if it contains any changes that would break how it is used in their code.

Package conflicts and virtual environments

By default, all Python packages are installed in a common location on your machine. However, this can lead to problems when working on multiple projects that require different versions of the same package. Sometimes applications will need a specific version of a package; for example, the application may require that a particular bug has been fixed, or the application may use an obsolete function of the library that is no longer supported in newer versions.

This means it may not be possible for your Python installation to meet the requirements of every project. If project A needs version 1.0 of a particular package, but project B needs version 2.0, then

Creating a virtual environment

To create a Python virtual environment within a code project directory, run the following command from a terminal:

```
$ python -m venv env
```

This will create a virtual environment in a subdirectory called `env`.

You then need to tell your terminal to use that environment for this project by ***activating*** the environment.

On Windows machines, you do this using the following command:

```
$ env\Scripts\activate.bat
```

On Linux or macOS, run:

```
$ env/bin/activate
```

the requirements are ***in conflict*** and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution is to create a Python virtual environment (https://docs.python.org/3/tutorial/venv.html). This is a self-contained directory within the project directory that contains a dedicated Python installation for a particular version of Python, plus any packages needed by that project. Different projects can have their own virtual environments with the specific versions of the packages they need, and each virtual environment is isolated from the others, so there are no conflicts.

Web application frameworks

Web applications are perhaps the most widespread form of computer software being developed and released today. Their popularity stems from the ubiquity of the World Wide Web, allowing businesses to deliver their applications to customers around the world without having to build their own global network infrastructure. All modern personal computers come with a web browser installed by default (and a reasonable range of well-supported alternative browsers that are available) so users don't need to install additional software before they can access a business's web app. Web browsers are also able to display rich and interactive content, thanks to advancements in HTML, CSS and JavaScript — key technologies that power the Web and which web applications can take advantage of.

Due to their dominance, we will be focusing on the development, deployment and maintenance of web applications during this course. However, all the philosophies and aspects of DevOps we will cover apply equally to mobile and desktop applications, too.

What is a web app?

Web applications (or more informally, *web apps*) are software programs that you interact with via a web browser. Examples include webmail, online banking and social media. They rely heavily on some of the key technologies underpinning the World Wide Web, including HTML and CSS to generate their user interface, HTTP requests and responses for communication with the browser, and JavaScript to enable dynamic functionality without needing to reload the whole page. Web apps have a client-server architecture, where the logic and content is provided by an application running on a server, but is displayed and runs in a user's web browser (the client).

So what's the difference between a website and a web application? Well, there isn't really a clearly defined boundary between the two, but generally speaking, websites consist of mainly *static* content (pages of information that do not depend on input from the user, such as news articles), whereas web apps have similar functionality to a desktop or mobile application. In reality, the division is rather blurry, and most modern websites will include at least some dynamic functionality. The term "dynamic web page" muddies the waters even further.

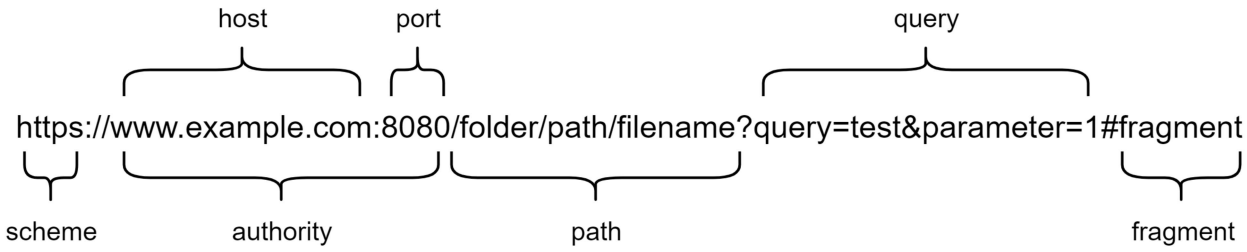
For the purposes of this course, we will use the term "web app" to mean any computer program that contains dynamic logic and deals with user-generated data, which users interact with via the World Wide Web (either through their browser or other client software).

Web frameworks (or web application frameworks) are common across many programming languages and are designed to take care of some of the common requirements that all web apps need to handle. Regardless of what sort of web app you're building, chances are it will need to be able to do the following:

- Accept HTTP requests from clients and perform logic specific to the request (e.g., return different pages based on the requested URL).
- Extract data from the request (e.g., details from a registration form)
- Generate dynamic HTML content (e.g., a web page showing a list of items for sale).
- Return HTTP responses containing HTML content (the page that the client web browser should display).

Web frameworks aim to provide this common functionality (and often a lot more, such as handling authentication, managing session cookies, etc.) and expose the request and response data to your app in a more convenient way, so that you can focus on building the logic that is specific to your app's features. Mature web frameworks also provide a more secure foundation for your app, often protecting against many common forms of attack that your web app would otherwise be vulnerable to.

Anatomy of a URL



A URL, or **Uniform Resource Locator**, is the location of a page, file, API or service on the internet. It has the following component parts, shown above:

- **Scheme:** both the format of the URL and the protocol to use; most commonly 'http' or 'https' for HTTP and HTTPS, respectively.

- **Host:** the hostname or IP address of the server to connect to, to fetch or use this resource.

- **Port:** an optional IP port number identifying the service or process to connect to; if this is omitted, use default values, such as 80 for HTTP and 443 for HTTPS.

- **Authority:** together the **host** and **port** are known as the **authority**. This may also include authentication in the form `username:password@host`

- **Path:** a logical path for the file or resource on the server, which need not correspond to a real filesystem path; components are separated by '/'

- **Query:** an optional sequence of key=value pair parameters to send with the request, separated by '&'s

- **Fragment:** an optional string not submitted to the remote server with the request; typically used to scroll to a specific point on the page, or identify a selected page or tab in a modern web application.

Where individual parts of the URL contain characters used as delimiters (separators) in URL syntax — e.g., `':'`, `'/'`, `'&'`, `'='`, `'#'` — these must be replaced (or *escaped*) using `%` and a two-digit hexadecimal value. For example, `'='` is represented as `'%3D'`.

Relative URLs are URLs that omit the scheme and authority, starting from the path component.

URLs are actually a subset of **Uniform Resource Indicators (URIs)** and the terms are often used interchangeably. URIs are defined by the IETF (Internet Engineering Task Force) document RFC 3986 (warning: that document doesn't make for light reading!).

Flask: a framework for Python web apps

Flask is a popular web framework written in Python. Actually, it's considered to be a *micro-framework* (as opposed to a *full-stack* framework) because it focuses on providing the minimal functionality needed, while allowing for extensibility by other packages to provide things like database connectivity, authentication, etc. This has the benefit of making it lightweight and flexible.

We'll be using the Flask web framework extensively in the project exercise for this course, so it's worth covering some of its key functionality in detail.

Anatomy of a basic Flask web app

Let's start with a concrete example: a Hello World web app, which has a single page that displays "Hello World" in your browser. First, create a new folder to contain the code for this example app and open a terminal from within that folder.

Before we can use the Flask framework, we need to install its package. You can install it from your terminal using pip:

```
$ pip install Flask
```

Once pip has finished installing Flask, create a new Python file in that folder called `app.py` and copy the following into it:

```
from flask import Flask

app = Flask(__name__)

@app.route('/hello')
def hello_world():
    return 'Hello World!'
```

We'll go through what each of these lines means shortly, but for now, save the file and run the following command in your terminal:

```
$ flask run
```

This launches Flask's built-in web server and you should see output similar to the following printed to your terminal:

```
 * Environment: production
   WARNING: This is a development
server. Do not use it in a production
deployment.
   Use a production WSGI server
instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/
(Press CTRL+C to quit)
```

Now open your web browser and go to http://127.0.0.1:5000/hello — you should see a page displaying "Hello World!".

Great! But how does that work and what's the Flask framework doing for us behind the scenes? When we execute the `flask run` command, it starts up a web server application that we installed as part of the Flask package. That server registers with your computer's networking system that it wants to accept any network requests sent to the IP address `127.0.0.1` (a special address meaning your own computer, also known as `localhost`) on port `5000`. Don't worry if you're unfamiliar with network ports and addresses, we will cover these in later modules. The key concept to understand is that a web server application reserves a specific *port* on the computer's networking system and then listens for any network requests that are sent to that port.

Different programs running on a computer can listen for network requests on different ports, but generally speaking, two programs cannot listen on the same port.

Now that the web server application has successfully registered itself on a network port and is listening for requests, let's walk through what happens when you enter the URL http://127.0.0.1:5000/hello into your web browser:

1. Your browser processes the URL you've entered and makes an HTTP request to `127.0.0.1` (i.e. your computer itself) on port `5000`.
2. Your computer's networking system sends the request to the Flask web server application, since it is registered (or *bound*) to that port.
3. The Flask web server receives the request and uses the app defined in `app.py` to determine how to handle the request.
4. The Flask framework extracts the **path** specified in the request URL (in this case, `/hello`) and checks to see if the app has registered how to handle that path.
5. Our app contains code instructing the Flask framework to call the function `hello_world()` when requests with the path `/hello` are received. Specifically, the code in `app.py` tells Flask to register a **route** for any requests with the path `/hello` and to call the associated function `hello_world()` whenever those requests are received.
6. Flask calls the app's `hello_world()` function, which returns the string "Hello World!".
7. The Flask framework then takes the string data returned from the function and creates an HTTP response containing that data.
8. The Flask web server sends that response back to your web browser.
9. Your web browser processes the response (in this case, containing plain text) and renders it in the browser window.

Phew! Putting that more concisely, your browser sends a request to your computer's networking system, the networking system sends it to the application registered on that port (the Flask web server), and the Flask web server calls the function registered by our app for that path, `hello_world()`. The data returned by that function is wrapped up in a network response and travels back the other way to your browser, which then displays it.

How does the code in `app.py` tell Flask how to handle requests? Going through line by line:

1. We import the `Flask` class from the `flask` package. Whenever you want to use a function or class from a package, you need to import it.
2. A new instance of the `Flask` class is created — this is our web app (note that `__name__` is a special variable in Python, which in this case is simply the name of the Python file without its extension, i.e. "app").

3. We register a route in our app, associating the function defined on the next line with the path `/hello`. This uses a Python language feature called a **decorator**, indicated by the `@` symbol, that wraps the `hello_world()` function in another function, `app.route()`, which is defined by Flask. The function is then called whenever requests matching the specified URL path are received by the Flask server.
4. Here we define the `hello_world()` function itself.
5. The function returns the string "Hello World", which Flask will then send back in a network response.

The above example shows the basic code structure of a Flask app and demonstrates how web apps respond to requests from the user, sent as HTTP requests and responses via the user's web browser. HTTP is a key technology that drives web applications and we will dive into more detail about the specifics of HTTP requests and responses in the next module.

But how do we go from a simple "Hello World" page, which doesn't allow for any user interaction, to a fully featured application that allows the user to perform actions, submit and modify data, and perform complex logic?

From a user's perspective, there are three key capabilities an app needs to support to achieve this:

- **Display a user interface:** the presentation of data, input fields and buttons or links to allow actions to be triggered.
- **Trigger an action:** for example, to view existing data, add or delete data, or search for results.
- **Submit complex data:** for example, user details during registration, details of a new product, etc.

The features provided by Flask to meet those requirements are called **request routing**, **HTML templating**, and **parsing request data**.

Users trigger actions by clicking buttons and links in their browser, which make HTTP requests to the web app. Request routing allows the web app to match those requests to associated actions and triggers the relevant app code. HTML templating allows the app to generate HTML dynamically based on app logic and the relevant data to be displayed. This HTML content is then sent in the response back to the web browser, which renders it and displays it to the user. Finally, users submit data by filling in HTML forms generated by the app, then clicking the form's submit button. This data is sent in the HTTP request from their browser and Flask processes (or *parses*) that request data so that the app can access and use it (to save it to a database, for example).

In the next few sections, we'll cover how routing, HTML templating and request parsing are achieved using the Flask framework.

Routing

Users navigate and interact with a web app by clicking on buttons and links displayed in their browser as part of the app's user interface. These clicks cause the browser to send HTTP requests to the web server, and the web app uses the **URL path** specified in the request to determine what action to perform, such as fetching specific entries from a database and returning their details, creating or updating an entry in a database, or returning some other generated content. This action is performed by a function defined within the app code. In web frameworks, functionality called **routing** acts as the "glue" that determines which function is called for a given URL path.

A **route** therefore defines what *action* is being requested, in some cases what *entity* that action should be performed on (for example, which entry to view out of a list of entries), and what *function* in the app code should be called to perform that action.

The action and the entity it should be performed on are defined by a URL path. For example, if you were building a library app, you might decide that any requests with the URL path `/books` should perform the action of returning a list of books available in the library. If a user wanted to view the details of a specific book (let's assume that book has a unique ID of '1' within the library's database), then you might choose to have requests with the URL path `/books/1` perform the action of returning details about that book — and more generally, paths of the form `/books/<id>` return details about the book with an ID matching `<id>`, where `<id>` could be any number.

In addition to the URL path, routes can be defined for different *types* of request, based on the **HTTP method** contained in the request (we'll discuss HTTP methods when we talk about HTTP in detail in the next module). For example, it's common to associate requests that specify the POST method with the action of creating a new entry, so in our library app we might choose to have requests with the POST method and the URL path `/books` perform the action of creating a new book entry in the library's database. Note that in this case, the path is the same as the one that should return the list of books, but the method distinguishes which of the two actions should be performed. Generally, if a route does not specify a method, the default is the GET method, which is the request method web browsers use when you click on a link in the browser.

If we have functions in our library app code that return details for all books (`get_books()`), return details for a specific book (`get_book(id)`) and create an entry for a new book (`add_book()`), we can now define the routes we would want in our app:

- Requests with the URL path `/books` should call the `get_books()` function.
- Requests with the URL path `/books/<id>` should call the `get_book(id)` function.
- Requests with the URL path `/books` ***and*** the POST method should call the `add_book()` function.

In Flask, you use the `route()` decorator to associate a function to a URL path (and optionally an HTTP method). Taking our library app example:

```
@app.route('/books')
def get_books():
    # Code to fetch all books from the database and return their details.

@app.route('/books/<id>')
def get_book(id):
    # Code to fetch the book entry with matching id from the database and
return its details.

@app.route('/books', methods=['POST'])
def add_book():
    # Code to create a new book entry in the database.
```

In the code above, each `@app.route(...)` line defines a route and associates it with the function defined on the next line. The `route()` function requires the URL path to be specified as a string argument and optionally accepts a list of HTTP methods to define the route (e.g., `methods=['POST']`, used on line 9).

When declaring the path for a route, Flask treats anything inside angle brackets `<...>` specially, and will extract the corresponding value from the path in an incoming request and pass it as an argument to the function. In the above example, this means that requests with URL paths `/books/1` and `/books/27` would cause Flask to call `get_book(1)` and `get_book(27)`, respectively. These are called **variable rules** in Flask and are used to determine what *entity* the action should be performed on. Variable rules are also capable of extracting more complex values from request paths and validating that those values match an expected format. You can read more about them in the Flask documentation.

Templates

The user interface for a web app is displayed in the user's web browser and is rendered from the HTML returned by the app in response to an HTTP request. The functions in the app code that are associated with routes therefore need to return HTML in order to render this interface. Web frameworks often provide functionality that makes it easier for web apps to generate the desired HTML, via the use of **templates**. In this section we'll see why this functionality makes developers' lives easier, and how HTML is used to create a user interface in practice.

When a request is for an action that returns dynamic data — for example, details about all the books in a library's database — then the associated function needs to return HTML content that is generated dynamically from that data. Taking our library app example, let's say we have some code that queries the library database and returns the following list of book entries, each of which is represented as a dictionary:

```
books = [
  { 'id': 1, 'title': 'Clean Code', 'authors': 'Robert C. Martin' },
  { 'id': 2, 'title': 'The DevOps Handbook', 'authors': 'Gene Kim, Jez Humble,
Patrick Debois, John Willis' },
  { 'id': 3, 'title': 'The Phoenix Project', 'authors': 'Gene Kim, Kevin Behr,
George Spafford' }
]
```

(This library happens to have a rather small and select collection of books!)

If we wanted to display these books as a list in the app's user interface, some minimal HTML content might look something like this:

```
<!doctype html>
<html>
  <head>
    <title>All Books</title>
  </head>
  <body>
    <ul>
      <li>Clean Code - Robert C. Martin</li>
      <li>The DevOps Handbook - Gene Kim, Jez Humble, Patrick Debois, John
Willis</li>
      <li>The Phoenix Project - Gene Kim, Kevin Behr, George Spafford</li>
    </ul>
  </body>
</html>
```

You would therefore want the `get_books()` function in your app code to generate this HTML using the list of book entries from the database and then return the HTML as a string, which Flask would then send to the browser as an HTTP response. It's possible to write code that would achieve this — perhaps using a for loop to iterate over the list of book entries and concatenating various strings. However, generating HTML using Python code is fiddly and potentially insecure. Therefore, most modern web frameworks include some form of ***template engine*** to make the process easier, and Flask is no exception.

Templates allow you to write files containing standard HTML, but use special keywords to add logic and include data from the app, which the template engine then converts into the necessary HTML. Flask uses the Jinja2 template engine and provides the **`render_template()`** function to render the HTML from a given template. That function takes the filename of the template as its first argument and any data to be passed to the template as optional keyword arguments.

Let's see what the `get_books()` function and the associated template would look like to produce the above HTML output:

```
@app.route('/books')
def get_books():
    books = get_all_books_from_db() # Some function that returns the list of
book entries from the database.
    return render_template('book_list.html', books=books)
```

On the last line, the `render_template()` function is used to generate HTML from a template file called `book_list.html`. The list of book entries contained in the `books` variable is passed into the function, so that it can be used when rendering the template. Finally, the HTML generated by that function is returned by the `get_books()` function itself.

Here's the `book_list.html` template, which uses the `books` variable that is passed in by the `render_template()` function:

```
<!doctype html>
<html>
  <head>
    <title>All Books</title>
  </head>
```

```
  <body>
    <ul>
      {% for book in books %}
        <li>{{ book.title }} - {{book.authors}}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

This looks largely similar to the HTML we had above, but there are some key differences:

- `{% ... %}` statements for flow control (lines 8 and 10) — in this case, these define the start and end of a **for loop** to iterate over the list of books that are stored in the `books` variable.
- `{{ ... }}` expressions (line 9) that evaluate a Python expression and populate the HTML with the result — in this case, the values of `book.title` and `book.author` for each element in the `books` list are inserted into the generated HTML.

In Python, we normally don't need to end for loops or other flow control blocks, since the interpreter is able to determine when the loop ends from the indentation in the file. This is not true for Jinja templates, so we need to explicitly end the loop using `{% endfor %}`. Jinja also allows you to use if statements and other flow control statements in your templates, for example:

```
{% if book.available %}
  This book is available.
{% else %}
  This book is currently on loan.
{% endif %}
```

In order for the user to be able to trigger an action within the app, we use a link element in the HTML. This will make a request to the URL path associated with that action when clicked. When adding buttons and links in HTML templates to trigger actions,

New to HTML?

If you're unfamiliar with HTML syntax, or just want to brush up, it's worth reading through the first few pages of this W3Schools Introduction.

We won't be using a lot of HTML in this course, but it's helpful to know what the basic structure of an HTML document looks like, and the syntax for some common elements, such as links, lists and forms.

you have to specify the URL path for the route associated with that action, using the `href` attribute for the link. For example:

```
<a href="/books/1">View Details</a>
```

This would create a "View Details" link in the rendered HTML, which would trigger a request to get the details for the book with ID '1' when clicked.

Using this, we can modify our template to add a "View Details" button for each book in the list:

```
<!doctype html>
<html>
  <head>
    <title>All Books</title>
  </head>
  <body>
    <ul>
      {% for book in books %}
        <li>
          {{ book.title }} - {{book.authors}}
          <a href="/books/{{ book.id }}">View Details</a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Here, we've added an anchor tag on line 11 to display a link next to each book entry. What does the `{{ book.id }}` syntax do within the anchor tag's `href` attribute?

Remember that in Jinja templates, any Python expression within `{{ ... }}` is evaluated by the template engine and inserted into the rendered HTML. As this line is inside the for loop, it will be repeated for each `book` entry in the `books` list. If each entry is a dictionary (as we're assuming in these examples), then `book.id` will access the value from the dictionary with the key `id`. Putting this together, if the `book` entry is the following dictionary:
`{'id': 2, 'title': 'The DevOps Handbook', 'authors': '...'}`
then the anchor tag rendered by the template will be: `<a href="/books/2">`

You can read more about the template syntax in the official Jinja2 template documentation.

Accessing request data

Web apps need to be able to process the data that a client sends to the server in an HTTP request — for example, when a user submits details in a registration form. In Flask this data is provided by the `request` object.

For example, to access login form data submitted in a POST request:

```
from flask import request

@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']
    if valid_login(username, password):
        return log_the_user_in(username)
    else:
        error = 'Invalid username/password'
        return render_template('login_error.html',
error=error)
```

For further details of the methods and attributes of the `request` object, you can read the Request documentation provided by Flask.

URL helper functions

Rather than writing these paths as hardcoded text in your templates, you can use the **`url_for()`** function, which takes the *name* of the function you want the path for as its argument.

In the example shown here, we could replace the hardcoded "/books/1" path with the `url_for()` function:

```
<a href="{{
  url_for('get_book', id=1)
}}">View Details</a>
```

When Flask generates the HTML from this template, it will call `url_for()` and replace it in the final HTML with the return value from that function ("/books/1").

There are several benefits to doing this:

- It's often more descriptive than hardcoding the path.
- Flask will throw an error if no route matches the specified function name.
- You can change the paths defined in your routes without having to remember to manually change all hardcoded URLs throughout your code.

Chapter 4

Python, continued

And now for something completely different

We've covered the fundamentals of Python programming in this module, but that's just the tip of the iceberg! Python has a lot more to offer, including many ways to manipulate the data types we've seen so far, to do useful and interesting things with them.

This section provides some example problems that will allow you to explore these topics in more detail and become more familiar with the core concepts.

In each example we give a brief introduction, describe the exercise, and provide links to a few relevant resources. However, you'll probably need to do a bit of investigating yourself to solve the exercise. This is a great way to become comfortable using online resources, such as the Python documentation and Stack Overflow.

Python, Monty Python

Guido van Rossum, the creator of Python, actually named the language after “Monty Python's Flying Circus”. If you are familiar with the content, you may recognise its humour in many of the examples and tutorials available in Python's documentation.

Mad Libs Generator

Let's start by looking more into the string manipulation methods that Python provides and gain some practice using lists. The goal is to write a Mad Libs phrase generator, which lets you enter a noun, verb, adjective and adverb, and then prints a random phrase that uses those words.

Your program will need to:

- Ask for input from the user.
- Select a random phrase from a list.
  - A hardcoded list of phrases within the program is fine — a great extension would be to read this list from a separate text file.
- Replace the noun, verb, adjective and adverb in that phrase with the words supplied by the user.

The following resources should help get you started:

- String methods available in Python
- Python's `random` module (**hint:** you'll need to import the module before you can use it in your code)

Stretch goal

Can you make the program more flexible, so that each randomly selected phrase could contain a different mix of ADJECTIVE, NOUN, ADVERB and VERB placeholders? The program will need to determine which word types are needed for that phrase and ask the user to provide them.

Bulk File Renaming

File organisation is a common – and often boring – admin task. For example, renaming a bunch of files to have a common prefix or suffix. Let's write a program to rename all files in a folder so that they are prefixed by the current date and time. The program should prompt the user for the path to the directory, then rename all files in that directory and print out a list of all the new filenames.

Since we'll need to apply the same change to a list files, this is a good opportunity to learn more about ***list comprehensions***.

Here are some resources to get you started:

- Python's `shutil` module (**hint:** you'll need to import the module before you can use its functions)
- List comprehensions

Stretch goal

Generalise your program to handle subdirectories of files within the directory specified by the user. How will you handle naming the subdirectories themselves?

Chapter 5

Version Control, continued

Diving further

Working with Version Control systems is core to many DevOps concepts and activities. We've touched on the basics in this module, but going a little further will allow you to work with even greater ease.

Tags

After working with git for a short while you'll probably want to learn about tags.

- These act as a mechanism for marking specific commits as important (a common use case is for indicating release numbers)
- They act a little bit like fixed branches; you can't commit to a tag, but you can push a tag and checkout a tag

Stashes

Stashes provide a way of quickly storing/saving your changes without making a commit.

- A common use case is when you need to urgently checkout another branch, but your current changes conflict with those in the branch.
    - In this situation you could
        1. Stash your current changes
        2. Checkout the other branch
        3. Do whatever you want on that branch
        4. Return to your original branch
        5. Pop the stash (i.e. restore your original changes)
        6. Continue with your work

Merge Tools

There are quite a few merge resolution tools out there. They can broadly be grouped into three categories:

- Dedicated tools like kdiff3
    - You can integrate this with the command line by declaring it as your merge tool in the global git config (assuming kdiff3 is on your PATH):
      `$ git config --global merge.tool kdiff3`
      and then running the following whenever you encounter a merge conflict
      `$ git mergetool`
- Tools included as part of a Git visualisation suite like GitKraken
- Tools included as part of an IDE like IntelliJ

Setting Up a Custom Editor

You can wire up your text editor of choice for handling processes like adding a complex commit message (e.g. with a long description) or for carrying out interactive rebases.

For example, if you wanted to set notepad++ as your default text editor you could run:

```
$ git config --global core.editor notepad++
```

Like with setting a merge tool, this needs to be on your path for this to work!

Other Customisation

There are many other config settings beyond setting your name, email and default tools. Check out this page for a full list.
You can view your configured config by running:

```
$ git config --global --edit # Global configuration
$ git config --edit # Repository specific config
```

This will open the config in your configured editor (on Windows the default is vim)

Private Repositories

We've mentioned GitHub as a popular platform for hosting repositories for free, but there restrictions on the number of collaborators for private repositories (a maximum of 3).

If you want to host repositories privately with many collaborators for free you may wanting to consider using GitLab.

Hooks

Hooks are an essential tool for setting up CI triggers efficiently. They will be covered in later modules but you might be interested in reading up on them in advance as they are another justification for using source control.