



# **BT furtHER**

## **Digital Intensive**

### **Programme**

**Module 2 - Python Fundamentals**

**Course curriculum**

2018- 2019

Prepared for BT by Chenhao Wu



# Contents

<b>Contents</b>	<b>1</b>
<b>Course overview</b>	<b>11</b>
<b>Installations</b>	<b>15</b>
Install Git	16
Install Python and Spyder	16
Install pip	16
Install database browser	17
Make Sure You've Got a GitHub Account	17
<b>Part 1 - Fundamentals</b>	<b>18</b>
Chapter 1 - Introduction to Python	18
Introducing Python	19
How we'll be using Python on this course	19
Experience coding through Python consoles	20
Creating and running your first Python script	21
Object oriented programming concept	22
Chapter 2 - Operations, Strings and Variables	24
Numbers in Python	24
Python 2 and python 3 operations	26
Names & variables	26
Introduce variables	26
How to use variable	27
Naming variables	27
Why meaningful names	27
Rules for naming variables	28
Variable practise	28
Strings in Python	29
Introduce strings	29
Double quote or single quote	29
String operation	30
Read error message	30
More explanation on string operations	30
String object methods	31
String formatting	31
Introduce string formatting	31
String formatting example	31
Different versions of string formatting	32

Comments	32
Chapter 3 - Functions & Importing Modules	35
Getting input from a user	35
Introducing functions	36
Function and method	36
Introducing function	36
Calling a function	38
Calling a function with indentation?	38
Arguments	38
The arguments (a form of inputs) of a function	39
Arguments example	39
More fun with functions	40
Must use variable names for your arguments	41
Mid-class challenge!	41
Revise functions	41
Your turn to write a similar function	42
Print result not equal to storage result	42
Returning stuff from functions	43
Return equivalent to save output in memory	43
Return the last operation result	44
Return None	45
Importing Python modules and functions	46
Separate function file and test file	46
Import beware	46
Import command can be used in several different ways	47
Preferred version	48
Shorten module name	48
Chapter 4 - Conditionals	50
Programmes and algorithms	50
Pseudocode	51
Introduce conditionals	51
More pseudocode tasks	51
Using “logic” in programming	52
Control structures	52
Boolean type and expressions	53
Logic and boolean type	53
Logic operators	53
Logic operations result in boolean type	54
Conditional statements (“if” statements)	56
How if-else conditions work	56
Else statements	57

Syntax of if-else statements	57
Reason for using an else statement	58
Three or more options -elif statements	58
Order is important	58
Chapter 5 - Object Oriented Programming	60
Object and classes	60
What is a class	60
Class and object	61
So everything has a class?	61
self	63
__init__	63
Fully-initialized object	64
Inheritance	64
One direction inheritance	64
Association (composition and aggregation)	66
Composition	66
Aggregation	68
Introduction to unified modeling language	68
UML with OOP	69
Visibility	69
Class relationships	70
Multiplicity numerical constraint	70
Public and Private	71
What problem does OO Programming solve?	71
Disadvantages of not using OOP	71
Object-Oriented Programming achieving modularity	72
Abstractions	72
OOP in real life applications	72
Inheritance and Real-world Programming Problems	72
Organising your Code via OOP	73
Chapter 6 - Command Line and Git with Python	75
Run Python using the command line	75
Steps to run a Python file with command line.	78
Navigating using the command line	78
Windows, Mac, Linux command line difference	79
Practise with bash shell	79
Creating new folder and files using bash	80
Shortcut and navigate tricks	80
Working with git in the command line	81
Setting up a new repository	82
Managing conflicts in git	85

Avoiding conflicts in git	86
<b>Part 2 - Design a Simple Programme</b>	<b>87</b>
Chapter 7- Debugging and Intro to Project Management	87
Use print function for debugging	88
How to check where is wrong?	88
Read error message and check/print variable accordingly	89
Your turn to solve the problem!	89
Datatype errors are very common	89
Do remember to remove the print debugging lines afterwards	89
Use breakpoints for debugging	90
Beware of line number and set breakpoints	90
Debug mode intro	91
Debug mode practise	91
Breakpoint summary and further resources	92
Best practice for coding	93
Think professionally	93
Different styles	93
A little bit about software project management	94
Planning what to work on.	94
Testing as you go along.	94
Keeping a list of broken stuff that needs fixing.	94
Github project management	95
How to design a program	95
Remember, it doesn't have to be perfect!	97
Chapter 8 - Mobile Data Bundle Purchase Programme	99
The mobile data bundle purchase programming task	99
First file, SimpleBundlePurchase.py	100
Second file, test_DataBundlePurchase.py	100
See if the printed out info as you expected?	101
Why None?	102
Recap on project design	102
Think through the problem logic	103
Writing steps based on the logic flow	103
Program structure	103
A first attempt in Python	104
Make a copy for your dev file	104
String or digit inputs	104
If-elif-else statement	105
A more realistic definition: using return	105
Test the code to see if it is as you expected	105

System needs to log/know what happened	106
Practise adding return in function	106
Return string logs in the program	107
Return two values in data bundle function	107
After adding returns, do test again	107
Breaking the task down, using functions as subroutines	108
Decompose functionalities	108
Introduce subroutines and its advantages	108
Example of using subroutine: passcode testing	109
Sub-function trick: combining boolean with conditionals	109
Three attempts input with a sub-subroutine	110
Mobile data bundle purchase function	110
<b>Part 3 - Compound Data Types</b>	<b>110</b>
Chapter 9 - Lists and Tuples	112
Intro to lists	112
Compound data types	112
Mutable and immutable arrays	112
First trial with a list	113
Index in list	113
Make changes to a list	114
Delete, overwrite and add items to a list	114
List operations	114
Slice of a list	115
Slicing is more 'permissive'	115
Sorting a list	116
Reverse sort	116
Introduction of tuples	117
Tuples are immutable	117
Advantages of tuples	118
Lambda function, used for a more complicated sort	119
Lambda is a notation for writing functions	119
Chapter 10 - Dictionaries	121
Intro to dictionaries	122
Dictionary is a mapping type data	123
Keys and values	123
Creating and using operations in dictionaries	123
Creating and assigning values to a dictionary.	123
Getting dictionary values through a key.	124
Overwriting dictionary values or doing operations with those values.	124
The 'format' for dictionaries.	124

Recap assess and update a value	125
Delete an item in a dictionary	125
Get keys and values from a dictionary.	125
What is the data type for keys and values?	126
Avoiding key errors	126
What are key errors?	126
How to avoid key errors	126
Sorting a dictionary by value	127
Experiment on small example before real task	128
Using lambda function	128
Return both key and values in sort	128
Sorting dictionary values in descending order	128
<b>Part4 - Iterations</b>	<b>129</b>
Chapter 11 - 'While Loop' and Guessing Number Game.	130
Loops as control structures	130
Two main loops	130
The while loop	131
While condition structure	132
Early exit and continuation	134
Do you have any questions about while loops?	134
'break' statement	135
'continue' statement	136
The guessing game	136
Let's create a game using while loop!	137
Chapter 12 - The 'For' Loop.	139
Introduction to 'for' loops	140
The counting loop	140
Also called iterative loops	141
A few helpful things about for loops	142
Using for loop to update items in a list	142
The 'for' loop vs the 'while' loop (intuitively)	142
The 'for' loop, used for simple iteration	143
Recap the list data type	143
Looping through a list	144
Looping through other types	144
Looping through a dictionary with sorted order	145
Search and operations inside a "for" loop	146
Loop list with index	147
Special cases counting loops: step size, range	147
Loop Control — with 'for' loops	148

Nested loops	149
A multiplication table	149
<b>Part 5 - More on OOP</b>	<b>151</b>
Chapter 13 - OOP Project	152
Revise object oriented programming	152
Person class example	152
The syntax of defining classes in python	153
The 'self' in class	153
Add conditionals in initialisation methods	154
Adding functionality	155
Defining classes with inheritance	157
Redefine __init__ in subclass	158
Smarter way to redefine __init__ attributes	158
Adding extra methods to subclasses	159
An object oriented programming project	160
Pre-project setting	160
Project intro	160
Pre-load file	160
Make your first moving figure	161
More obvious moving	161
Creating our first moving figure	162
Test MovingShape.py	163
Adding code in MovingShapes.py to move the figure	164
Adding random variation — velocities	165
Creating random movement	165
Move in positive and negative directions	166
Minimum and maximum start position	166
Adding random variation for the start positions	167
Hitting the wall	167
Diamonds vs. squares	167
Q/A Python modules and Classes	169
Python modules can contain Classes	169
Similarity	169
<b>Part 6 - Real World Applications</b>	<b>171</b>
Chapter 14 - Databases	171
Introducing databases	171
Model data in rows and columns	171
Why are databases necessary?	171
Relational database	172
Database third-party services	172



mySQL connector	172
Syntax to connect mySQL	173
SQLite	173
Creating a database table	173
Import and connect a database	173
Connect cursor	174
Create table	174
Uppercase commands	174
Database table	174
Inserting data into tables	175
Matching column names to insert row values	175
Routine command	175
Call the function to create table	175
Use database browser to check data table	176
Insert data using variables	176
Static data and stream data	176
Inserting data dynamically	177
Insert multiple rows with a for loop	177
No need for a sleep function in real insertion	177
Reading data from databases	178
The select statement	178
fetchall()	178
Data select exercises	178
Order by and group by	179
Preparation for next chapter	180
Chapter 15 - APIs	180
Introducing APIs	182
What you can do with API ?	182
Setting up your first API in Python	183
Confirmation email task	183
Getting ready to use the Mailgun API	183
Testing the code for your Mailgun API	185
Using an API to find out what the weather is today	187
HTTP status code	188
Unicode	189
Using JSON to report the weather in a user-friendly way	189
Preparation for next chapter	192
Chapter 16 - Flask, and Using Python and HTML Together	193
Making and running your first app with Flask	194
Import Flask	194
Routes & decorators	195

app.run()	196
Save Flask output in an app variable	196
run()	196
A little more about decorators	197
Setting up the file structure for your app	198
Serving your HTML using Flask	198
render_template	198
Using Jinja2 syntax in your HTML files	199
You're nearly ready to serve your first HTML file with Flask!	201
Getting user generated data from your webpage to Python	202
Debugging tips	204
Chapter 17 -Deploying Your Code	206
What it means to deploy code	207
Currently, where is your website?	207
Deploying your code to a web server	207
Cloud-based Platform	207
Setting up Heroku so it works with the command line	208
Getting your Github code ready to work with Heroku	209
Clone from git	209
Heroku folder structure	210
Create and deploy a simple app	210
Where's my website and how do I update it?	211
Pushing code from your local git repo to Heroku's git repo	211
Check running app	211
Getting your own project set up on Heroku	212
Why it's important to use git for version control	212
Self explore Flask with Heroku	213
<b>Appendix</b>	<b>213</b>
Python general	213
Good video tutorials	214
For part 1 - part 4.	214
Codingbat logic.	214
MIT Python courses.	214
More readings	214
Basic	214
Command Line	215
Code python online (no need install)	215
More Python practise and learn algorithms	215

# Course overview

The course aims to provide a basic overview of the technologies used, along with the tools and resources to discover more.

The focus of this course is learning the basics of how and why things work and to provide the basis to build upon in future courses.

Sessions will be as hands-on and practical as possible. The notes provided will give you and the students a good resource to read through and base your lessons on. Try to be as interactive as possible.

For a detailed overview, we will first install all required tools for this course and make sure everything works at the beginning of this course, we then let you get used to the Python console with some simple operations with a recap of the object oriented programming concept from your last module.

After the introduction chapter, you will be able to create your first Python script and run it without much hassle. In chapter 2, I will introduce more complex operations, simple string manipulation (which Python is great for!) and how to use variables, which is one of the most clever inventions in programming. In chapter 3, you will learn how to write a function and call it using script or even inside other functions, which is a very convenient way to reuse your code. You will also learn how to import other's functions and very useful libraries to make your project work. It is great to write your own code but at the same time it is also very important to be able to make use of already existing libraries and functions and understand them or even change them.

We then move on to chapter 4 to learn basic coding logic. Some people may argue that logic is more important to learn than coding. This is very true, for once you master the logic flow, you will be able to transfer between learning many different programming languages without much difficulty. Conditionals are also a great tool for teaching a machine to make decisions, all the 'big data', machine learning techniques would not be able to work without those 'if-else' statements. In Chapter 5

we will thoroughly revise Object Oriented Programming, how to define classes with inheritance and distinguish public and private attributes, which becomes very important when you write more complex programs. In chapter 6 we will do many exercises in using the command line to push your code to git, at the same time you may pick up some bash shell script languages as well.

After learning the very foundation of coding, we will move onto part 2 which includes learning how to debug and then how to design a semi-real life program based upon your clients' (instructors') needs. You will need to use all the materials you will have learned so far to achieve it, especially to think about how to test it to make sure it works while making sure the logic flow is correct.

You will soon find out that coding can be tough but very interesting and in part 3 Python has some very cool data types such as lists, tuples and dictionaries which are super useful when you write many functions in the near future. In part 4, you will have the chance to revise and apply part 3's data types by using loops, which are also very powerful tools to make machine learning and artificial intelligence possible. Overall, robots and machines can help humans do many repeated tasks which very much rely on those 'while' and 'for' loops.

Part 5 aims to let you to re-familiarise yourself with OOP while developing an OOP project using all the knowledge you have learnt. Part 6 will bring you into the web development scenario, which is a more advanced version compared to module 1 as some of the BT internal clients may ask you to do some web development related project in the near future, so those chapters will help you to build more ideas of how to match your clients' needs.

With different fonts, the text in this curriculum will draw your attention to different tips and information:

**Note and tasks, when you see any text in this font and format, it means there is a tip, note point or task that you need to do.**

Code will be in this font, all code in this module will be smaller than normal text.

**Future exploration:** and text in this colour font will not be covered in this course, but it is good to explore it more if you want to gain a deeper understanding.

The course curriculum overview is laid out below:

**Getting going: Install everything we need for module 2!**

- Installation

**Part 1: Fundamentals**

- Ch.1 - Introduction to Python
- Ch.2 - Operations, Strings and Variables
- Ch.3 - Functions & Importing modules
- Ch.4 - Conditionals
- Ch.5 - Object Oriented Programming
- Ch.6 - Command Line and Git with Python

**Part 2: Design a simple programme.**

- Ch.7 - Debugging
- Ch.8 - Mobile Data Bundle Purchase Program

**Part 3: Data Types.**

- Ch.9 - List
- Ch.10 - Dictionaries

**Part 4: Iterations**

- Ch.11- 'While loop' and Guessing Number Game Programme.
- Ch.12 - Iterations 2, 'For loop'.

**Part 5: More on OOP**

- Ch.13 - OOP project

**Part 6: Real world applications**

- Ch.14 - Database

- Ch.15 - APIs
- Ch.16 - Deploying Your Code
- Ch.17 - Web App Development

# Installations

There are a few things you'll need to set up before our first class, so that you have everything you need to get the most out of the module 2 Course. Chances are you'll have most of these things done already during module 1. All the installation instructions will be for Windows computers. If you wish to practise with your own computer system, please consult your instructor.

**If you have any trouble with these tasks, please make sure you contact an instructor as soon as possible, so they can help. You'll find their email addresses in your welcome/acceptance email from Code First: Girls.**

## Let's get started!

**Task: Finish reading this page. Compare similarities and differences between Python installation with web module 1 installation. Then turn to your partner and check whether you have both installed the Python course related software correctly.**

Things we need to do (we'll explain each of them in these notes):

1. Install Git
2. Install Python and it's editor
3. Install pip
4. Install a database browser
5. Make sure you've got a GitHub account

## Install Git

We need Git to serve as our default version control tool, it will also provide a nice bash terminal which will come in very handy during this course.

1. Download the Git for Windows [installer](#).
2. Run the installer and make sure you select the following options when needed:
  - a. **Keep using Git from the Windows command Prompt.**
  - b. **Checkout windows style, commit Unix-style line endings.**
  - c. **Use Windows' default console window.**
3. If your "HOME" environment variable is not set (or you do not know what this is) ask one of the instructors for help.

## Install Python and Spyder

Python does not come pre-installed with Windows systems.

1. The software packages we will use in this course are Python 3.7 and spyder editor. However, it can be installed together using an Anaconda package.
2. Head to the Anaconda main website ([here](#)) and download the latest version.

## Install pip

**pip** is a package management system for installing and managing software packages (libraries) written in Python. We'll be using it to install things like Flask and any other libraries we want to make use of during this course.

A library is a collection of pre-written code we want to re-use. As an example, we don't want to reinvent or re-write a web server from scratch for each of our web



projects, so instead we use a library to provide us with the functionality we want. Don't worry about what Flask is. We'll tell you all about it during the course.

**pip** comes with most of the newest Python distributions. First check if you have this installed, open the Git bash that you just installed and type the following command:

```
pip -V
```

You should see a legend stating the version of pip you have installed and the location of this package. If you do not have pip installed go to <https://pip.pypa.io/en/stable/installing/> - [do-i-need-to-install-pip](#) and download **get-pip.py**

Download [get-pip.py](#), being careful to save it as a .py file rather than .txt. Then, run it from the command prompt:

```
python get-pip.py
```

You possibly need an administrator command prompt to do this. Follow [Start a Command Prompt as an Administrator](#) (Microsoft TechNet).

## Install database browser

Follow the instructions on this website: <http://sqlitebrowser.org/>

## Make Sure You've Got a GitHub Account

You should have a GitHub account already, from the beginners course. If you don't have one yet, sign up for a free one [here](#).

# Part 1 - Fundamentals

## Chapter 1 - Introduction to Python

### Learning outcomes

- Install Anaconda Python with Spyder editor.
- Coding through console.
- Coding through file editor, (set up configurations).
- Understand the object oriented programming concept.

### Recap from web course

- Recap the installation section in the web development course.
- Thinking about the object oriented programming in module 1 and compare to the concept in this chapter.

### Preparation for today's session

In the pre-course work you were sent, there were instructions for how to install Python, pip, and an editor (Spyder). Please make sure you've done this before the first class, because we're going to dive right into the exercises. If you're having trouble installing anything, please email your course leader to let them know (see your welcome email for their email address).

### Please Note

Remember to use [GitHub](#) and [Stack Overflow](#), for searching for answers to any queries, those websites make it easier than ever to learn and contribute!

## Introducing Python

Python is a powerful, open-source and easy to learn programming language. Its design philosophy and syntax emphasises code readability and the ability to express concepts in fewer lines of code than would be possible in many other languages such as C++ or Java.

PEP 20 ([The Zen of Python](#)) summarises the language's core principles, which include:

- Beautiful is better than ugly
- Readability counts.
- Simple is better than complex.
- Complex is better than complicated.
- Explicit is better than implicit.

As you can see, it's an ideal language for first time programmers!

There are lots of things you can do with Python. You can take a look at examples from recent Code First Girls competition winners [here](#).

## How we'll be using Python on this course

Over the next 7 weeks, we'll be learning the fundamentals of Python, we then move to web app development using a simple web framework called Flask.

During the beginner's course you will have learned how to make HTML websites, to show information to your website's visitors. You may have even used a little bit of Javascript, to make things more interactive, perhaps by animating, showing or hiding information on a webpage.

On this course, you'll learn how to take things to the next level – you'll be making an interactive website that uses both Python and HTML to get information from your

website's visitors, do stuff with it using internet services and show the results to your visitors, either on your website or via email, Twitter, etc.

You might hear us using the terms “web application”, “web app” or “interactive website” – for the purposes of this course, they all mean the same thing.

## Experience coding through Python consoles

Begin by entering some arithmetic expressions, using the basic arithmetic operators:

+	plus
-	minus
*	multiply
/	divide

Open your Spyder editor and try the lines below in the Python console.

```
>>> 2 + 3
```

```
5
```

```
>>> 2.9 + 3.2
```

```
6.1
```

```
>>> (3.2 / 2.0) + 7
```

```
8.6
```

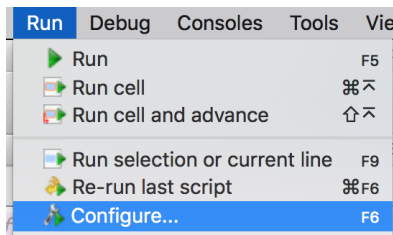
```
>>>
```

Try using different mixtures of the arithmetic operators, with and without brackets, to fix the scope of the operations. For cases without brackets, see if you correctly anticipate the operator scopes that Python assumes. Operations will be covered in more detail in Chapter 2. For now, you can try entering some more expressions, to test your understanding.

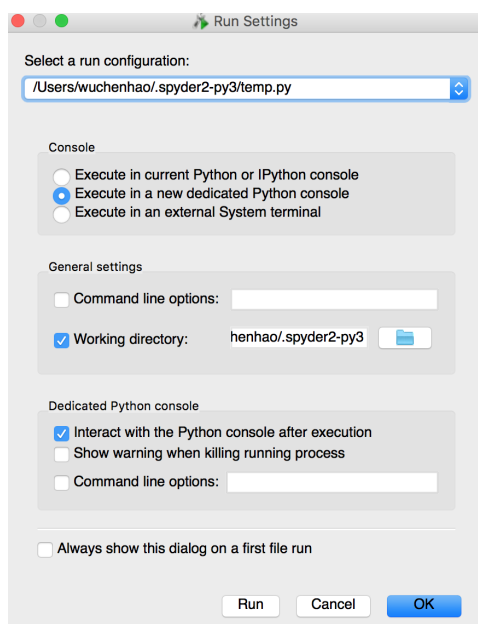
## Creating and running your first Python script

First, let's do some configuration setting. -- You will need to do this every time you use a new PC with Spyder.

Open your code editor (Spyder), click the **Run** section and click **Configure...** as below:



After clicking **Configure...**, a 'Run Settings' window will pop out as in the picture below:



In the **Console** section, select 'Execute in a new dedicated Python console', and in the **Dedicated Python console** section, select 'Interact with Python console after execution', then click **OK**.

After the configuration, let's create a new file. In this file, let's just type one thing:

```
print ('Hello, World!')
```

It doesn't matter if you use single or double quotation marks, as long as you use them together, in pairs. Now save your file – let's call it **hello.py** (the .py extension means we're dealing with a Python file). It's important that you **do not name any of your folders for this course "Python"** – things won't work if you do!

Now you have your brand new Python file saved, let's run it! Just click: 

Each line you type in your code editor is a single Python “statement” – a small piece of code that Python can evaluate to either produce a result or to do something. Python programs are simply long lists of statements spread across one or more (sometimes thousands!) of files. Python reads and performs each of these statements one after another.

**Be warned!** — indentation is very important in Python (as we will soon see in class). If you precede your expression with even one stray space character, the Python shell will complain of an Indentation Error and refuse to execute your code.

Also, remember to save your Python file every time you make some changes!!!

There is also a real time online Python editor. If you haven't been able to get Python working with Spyder, ask an instructor how to get started with [repl.it](https://repl.it), which lets you write and run Python code from your browser.

## Object oriented programming concept

Object Oriented Programming (OOP) has become the dominant programming paradigm over the last 20 years. It was developed to make it easier to create and/or modify large, complex software systems. You will come across it in Python in chapter 5 and there will be an OOP project recap section in chapter 13. You will be encouraged to use OOP throughout module 3.

In Object Oriented Programming, the focus is on creating objects to mimic real life (e.g. objects that define cats or objects that define rabbits), which contain both data (e.g. age, name and colour) and functionality (e.g. making sound or running speed). Here the objects are data types that you can define by yourself (define the class to

say what the object is and define functions to show how to use the object and how someone can interact with it). However, let's learn some basic data types in Python first, then later you may create your own object type.

## Chapter 2 - Operations, Strings and Variables

### Learning outcomes

- Do basic maths operations with integer and floating point numbers.
- Do basic string manipulation.
- Create and manipulate variables.
- Print numbers, strings and formatted text.
- Use comments in your code.

### Recap from last session

- Which editor are we using for this Python course?
- What do you need to check before running the code and why?
- Did you remember to save your file every time you made a change?
- Which web browser allows you to edit and run Python in real time?

### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain what strings, variables and comments are in Python! Once you've done that, explain how to do variable operations with different formatting!**

### Numbers in Python

As we tried in chapter 1, you can do maths in the Python console! This time, let's add some code to our hello.py file to solve some maths problems. We can print the results just like we printed Hello World, except because **we're using numbers, we don't need to use quotation marks.**



Here are the different mathematical operators you can use in Python:

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

### Task 1

Use your Python file to print answers in your command line for the following maths problems:

5 - 6

8 \* 9

6 / 2

5 / 2

5.0 / 2

5 % 2

2 \* (10 + 3)

2 \*\* 4

Don't forget to put each statement on a new line. For example, if you want to print the answer for 1 + 1, it's as simple as writing `print (1 + 1)` in your file, saving it and running it.

It's up to you whether you want to leave spaces between numbers and operators. In Python, it doesn't matter – but like quotation marks, it's important to be consistent.

Are the answers what you expected to see?

Don't forget to put `()` around the info you would like to print. E.g. `print(2**4)`

## Python 2 and python 3 operations

Python 2 will no longer be supported after 2020 so we are using Python version 3, which can take care of most cases, but you may encounter Python 2 occasionally.

However, if you used Python version 2 in other systems or the online browser editor, you might have seen a few results you didn't expect in the task above.

In those cases if you give Python integers (whole numbers), it will do integer division. For example,  $5 / 2$  gives an answer of 2, because that's the largest whole number of times you can remove 2 from 5.

We can experiment with this by typing `int(5/2)` or `5//2` in Python 3.

If you give Python decimal numbers (called "floating point numbers" or "floats" in many programming languages) it will do normal division. Any number with a decimal point is considered to be a float.

You might remember from your school days that a remainder is just what's left over from your whole number division. We can use `%` ("modulo") to get the remainder of an expression. So,  $5 \% 2$ , gives the answer 1.

For a few more examples you can try, have a look at the homework at the end of these notes.

In different organisations or company sites, their software package version can be different, so we need to adapt when working with them. There are also drawbacks for using older versions, which we will cover in some chapters of Python module 3 after learning the module 2 fundamentals.

## Names & variables

### Introduce variables

Enter some statements that use variables to store values. This is particularly helpful when you write code where you may need to adjust values afterwards. When using variables you can change a variable once rather than change the value everywhere in your programme.

You can also give meaningful names to those variables to help you remember the use of those values.

## How to use variable

We use “=” to assign a value to a variable (whose name appears to the left of =). Elsewhere, when a variable is evaluated (e.g. when it appears in an arithmetic expression), its value is accessed and used at that position:

```
>>> x = 12.5
>>> x
12.5
>>> y = 3 * x + 2
>>> y
39.5
>>> x = x + 1.1
>>> x
13.6
```

All the variables you declared here are just in this section, and the program will not remember it afterwards.

## Naming variables

Most of the time, it is better to use meaningful names for variables, rather than simple names such as x. For example, we might use a variable named balance to store our phone credit as a floating point number or a variable message to store a string that is to be printed as a message.

## Why meaningful names

Using **meaningful variable names** makes your code much more **readable and understandable**, which is a key feature of **good programming style**. It not only helps someone else to understand your code (e.g. your work partner or manager), it also helps you understand your own code, which is vital if you’re trying to figure out why it doesn’t work or if it does something that you’re not expecting.

## Rules for naming variables

Note that variable names can contain **letters, digits and underscores** (i.e. \_), but **must begin with a letter**. Note also that there are some special words — Python’s

keywords — that cannot be used as variable names, e.g. you cannot name a variable as 'print'. Do a web search for these terms and bring examples into class.

## Variable practise

Programming becomes a lot more powerful when you're able to give values names. A name is just something you can use to refer to a value in the future.

In Python you create a name by using the assignment operator = .

**Task 2, write the below code into your Python editor and run it:**

```
age = 5
```

You can change the value associated with a name at any point. The new value doesn't even have to be the same type as the old one. Here we change the value associated with age from an integer (number) to a string (text):

```
age = "almost three"
```

**Note** the 'age' variable is now updated to "almost three" rather than the number 5.

In Python it's conventional for variable names to start with a lowercase letter. If you want to use multiple words in a name, you can separate them with an underscore, like this:

```
a_longer_name = "hello, CFG!"
```

You can print a variable's value like this:

```
print (age)
print (a_longer_name)
```

Try printing age when you assign it different values e.g. "almost three" and 5 and see what will happens...

## Strings in Python

### Introduce strings

In the Python introduction chapter we printed some text. In most programming languages, values like this are called “strings” (or ‘str’), because they’re formed from a string of individual characters.

### Double quote or single quote

Try entering some strings, which are just character sequences between paired quotation marks. Python doesn’t mind if you use " or ' for this, as long as you have the same quotation mark at both the start and end of the string, but pick one and be consistent! E.g. --try these in your Python console:

```
>>> "Hello World!"
'Hello World!'
>>> 'Time for tea.'
'Time for tea.'
>>> "these quotes are not paired"
SyntaxError: EOL while scanning string literal
```

**If you run `type("almost eight")`, do you see what happens?**

### String operation

This time let’s try running code with an edited file. Python can do some clever things with strings. Let’s see what happens when we run our file with this:

```
print ('hello' + 'world')
```

As you see here, you can combine strings using +, which appends the second one to the first.

### Task 3

Let's add some code to our **hello.py** file to do some cool stuff with strings.

```
print ("Bob " * 3)
print ("Bob" + 3)
print ("hello".upper())
print ("GOODBYE".lower())
print ("the lord of the rings".title())
```

Are the answers what you expected to see?

### Read error message

Sometimes when you type things in a Python file and run it, you'll see an error message instead of a result. **Turns out that you can't add a string to an integer.** Have another read of the error message that was given out. Can you figure out what it's saying? When something goes wrong, Python tries to be as helpful as it can.

### More explanation on string operations

Some of the arithmetic operators, specifically `+` and `*`, have additional special meanings when used with strings. For example, if `s1` and `s2` are variables storing strings, we can evaluate expressions such as `s1+s2` and `s1 * 10`. Investigate the effects of these operations with the Python interpreter. Also, investigate precedence for these operators, e.g. by entering: `s1+s2*10`.

### String object methods

`.upper()`, `.lower()`, and `.title()` are called "methods". Attaching a method to a string tells Python to do some processing on the string – in this case, it converts all of the characters to uppercase, lowercase, or title case.

For object oriented programming, as mentioned in chapter 1, a string is an example of an object. When using a method that can only be used by a particular object, we must connect that method with that object by prefixing it with a `'.'` just like

with `.upper()` and `.lower()`. You can find out more about string methods in the Python documentation [here](#).

## String formatting

### Introduce string formatting

String formatting is a way of taking one or more variables and putting them inside a string, using placeholders for the values. This is a very useful technique that is not only limited to Python (we will mention it again in SQL and web security).

There are a few ways you can do this in Python, but not all of them work in different versions of Python, so on this course we use `{}` because it's the most reliable.

### String formatting example

Let's use an example of a 5 year old who likes to paint. We need two variables to hold this information:

```
age = 5
like = "painting"
```

So, we've got some variables, but how do we print this information in a sentence that looks nice? There are a few different ways we can do this:

```
age_description = "My age is {} and I like {}".format(age, like)
```

Or we could also do it this way:

```
age_description = "My age is {0} and I like {1}".format(age, like)
```

These will both give you the same result:

```
'My age is 5 and I like painting.'
```

## Task 4 Have a try by yourself with the examples above.

### Different versions of string formatting

Sometimes, when you're doing exercises or looking at code online, you might see string formatting done with % instead of {}. This is from older versions of Python. If you see %, it just means this:

%s → {}

%d → {}

%r → {}

If you're wondering what the letters stand for, "s" is for string, "d" is for decimal, and "r" is used to format a string in a particular way (you can read more about %r [here](#)). It's best to use {} to make sure your code works across all versions of Python.

## Comments

In Python, any part of a line that comes after a # is ignored. This is useful when you're writing complicated programs and working with others, because it means you can write short comments in plain English to help others to follow your code and understand what it does.

Here are some comments in action:

```
greeting = "Hello World!"           #This creates a variable
greeting.upper()                    #This converts the string to uppercase
print (greeting * 3)                #This prints the string 3 times
```

**Remember:** any line preceded by # is a comment — it is ignored by Python.

**Comments can be very useful or confusing. Please have a read of the comment chapter in Clean Code, chat with your classmates and instructor about your findings.**



## Homework

Add code to your Python file to print the answers to these expressions in your command line:

```
10 / 3
10 % 3
```

Create a new Python file with the following statements in it, then run it. Were the things that were printed what you expected to see?

```
a = 1
a = a + 1
print (a)
b = "hello"
print (b)
c = b.title()
print (b)
print (c)
d = "hello"
e = d.title()
print (d)
print (e)
name = "Dave"
f = "Hello {0}!".format(name)
print (f)
name = "Sarah"
print (f)
print (f * 5)
```

It might seem obvious, but it's worth pointing out that = is an “assignment operator”. This means “set the name on the left equal to the value on the right”. It isn't the same equals as you see in maths!

This means that strings are a little bit different. String formatting happens when you write it down. So, when you first write `f = "Hello {0}!".format(name)` Python

immediately looks up name and bakes it straight into the string called f. Setting name to something different later on won't change f.

1. Work your way through exercises 1 to 10 of [Learn Python The Hard Way](#). If you saw a warning on not using Python 2, just ignore it as we will use Python 3+ in this course.

## **Extra Homework (optional)**

1. Continue to try some challenges in codingbat Python <https://codingbat.com/python/String-1>

## Chapter 3 - Functions & Importing Modules

### Learning outcomes

- Get input from a user.
- Understand what is a function.
- Understand function syntax and arguments.
- Write a function (also known as a “method”).

### Recap from last session

- What is a string?
- What is a variable?
- What is a formatted string?
- What is a comment?

### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how functions work! Once you've done that, explain how to create a function by yourself!**

### Getting input from a user

From last week's session we now know how to create variables and how to print them, but what if we don't know what the value of our variable should be, because we need to get it from a user? By using the method `input()` we can prompt a user for information.

### Task 1

Create a new file called **my\_name.py**. Copy and paste the code below into the file and run it in your command line:

```
print ("What's your name?")
name = input()
print ("Hello {}".format(name))
```

Cool, right? Now we can get information from a user and use Python to do stuff with it.

Also, as shown above, the `input()` function takes a single (optional) argument, which is a string. When called, the function prints this string as a 'prompt', and then reads in whatever text the user types up to the first line return. This input string is returned and we can assign it to a variable.

## Introducing functions

### Function and method

During this course you might hear the words “method” and “function”. We use them interchangeably, because they’re essentially the same thing but different programming languages gravitate towards one name or the other.

### Introducing function

A function is a block of code with a name assigned to it. Functions are really handy, because you only have to write them out in full once, then you can use them again and again anywhere in your program by “calling” them (more on what that means in a bit).

The concept is a little bit like a paragraph. If you think back to English class in school, a paragraph is “a distinct section of a piece of writing, usually dealing with a

single theme and indicated by a new line, indentation, or numbering” (Google Dictionary).

In Python, a function is a really similar idea – it’s a distinct block of our code that deals with a single task/theme, and is formatted in a particular way. Just like a paragraph, a function can be one statement or many statements long.

Here’s an example of a really simple function:

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

Any time you write a function in Python, it’ll have the following things:

- **def** at the very beginning, so that Python knows the indented code below is part of a function (you’re “defining” your function below this line).
- A unique name, with no spaces in it. It’s important that you don’t have two functions with the exact same name in your code. You also don’t want to name your function “function”, “sum” or “python” (or later on in this course “email”, “mailgun” or “tweepy”). If you name your function after a Python library or package or built-in function, Python will get really confused and your code won’t run.
- You’ll also need to put brackets and a colon right after the name.

Notice how, after the first line of code in a function, the other lines are **indented**. In Python, this is very important – it’s how the code interpreter in your computer knows what’s part of your function and what isn’t.

If you directly copy and paste code from this material, the indentation may not work, as the formatting is different (or some intended mistakes :-p). Therefore it is down to you to sort out and it is really a good practise for you to organize your own version of examples/exercises code from this criculumn.

**PLEASE NOTE: be warned! — getting indentation right is crucial to successful Python programming. You should create indentation by using the TAB key in Spyder. You should never attempt to indent code ‘manually’ by using spaces.**

## Calling a function

If you watched the pre-course material, you will know that a function by itself doesn't actually do anything. When you run your code, it just passes it.

You have to "call" it by its name to run the code inside of it. This is because there may be many functions in a library/Python file, so you will only use the ones you "call".

Calling a function is really easy! All you have to do is type the name of your function with () after it, like this: `hello_world()`. **Have a try by yourself!**

### Task 2

Create a new file called **my\_first\_function.py** and write a function that prints your name in the command line, instead of Hello World!

Add another line of code so that your function also prints the answer to  $2 + 2$ .

### Calling a function with indentation?

Notice that the function call is NOT indented, because it's not part of the function definition!

However, we will learn some cases to calling a function inside another function definition later. Things will become much more exciting then! So make sure you understand everything and get practised at this stage.

## Arguments

We've all had those moments when our computer says "no" to our code and we say "but my code looks good! Why isn't it working?!" – but that's not the kind of argument we're talking about here!

## The arguments (a form of inputs) of a function

Every function you write in Python will have () after its name. Sometimes, you might want or need to put some values/variables/parameters between those brackets, so that they can be used inside your function. The values inside those brackets are called “arguments”. Sometimes you might see them referred to as “args”. Depending on the function you’re using, you might have no arguments, one argument or multiple arguments. If you’re writing your own function, you get to define this!

Also, the arguments are just one form of input of your function, If the arguments type and the function’s functionality do not match, of course your computer will complain! Also, read the difference between argument and input in [stackoverflow](#).

### Arguments example

Let’s use the built-in Python function `range()` as an example. We don’t have to write the code for it because the lovely people who came up with Python have done that for us already. From Python’s documentation, we know that `range()` can take 1, 2, or 3 arguments (you can read more about this function and how it works [here](#)).

#### Task 3

Create a new file called **my\_arguments.py**, paste the code below into it, and run it. What do you think `range()` is doing when it's given 1, 2, and 3 arguments?

```
print (range (10))           #one argument
print (range (1,10))         #two arguments
print (range (1,10,2))       #three arguments
```

When we write a function, we can design it to have a fixed number of arguments first. Once you feel like you’re very good at that, you can then explore more about flexible argument setting.

## More fun with functions

**Task, try to write a function that can add two numbers together and prints out the answer. You can name it as `add_two_numbers()`. Think about the steps for how you going to write it.**

After writing the function, remember, **it doesn't actually DO anything until we call the function** using the statement `add_two_numbers()`.

What do you think about the following `add_two_numbers()` function?

```
def add_two_numbers():
    number1 = 1
    number2 = 2
    answer = number1 + number2
    print("{} plus {} is {}".format(number1, number2, answer))
```

```
add_two_numbers()
```

Here's another function that adds two numbers together. Notice how this function takes two arguments: `number1` and `number2`.

```
def add_two_numbers_from_args(number1, number2): #These can only be variable names
    answer = number1 + number2
    print("{} plus {} is {}".format(number1, number2, answer))
```

```
add_two_numbers_from_args(5,10)    #These can be variable names or actual numbers
```

### Task 3

1. Write the code immediately above into your **my\_arguments.py** file and run it.
2. What happens when you change the values of the arguments you used when you call the function?
3. What do you think happens if you don't put 2 arguments in when you call the function `add_two_numbers_from_args`?



## Must use variable names for your arguments

You must use variable names for your arguments when you're defining your function (you can't put the actual values of the numbers or strings there). This is so you can **re-use your functions**, because the actual values for the arguments are given when you call the function, just like you see in `add_two_numbers_from_args(5,10)`. Python is smart and knows that when you call that function, the first value inside the bracket is `number1` and the second value is `number2`.

## Mid-class challenge!

### Revise functions

Defining a function is a good way of creating conveniently reusable chunks of functionality. Let's combine the knowledge we just learned from defining functions, calling functions and arguments. Copy the `convert_distance` function below into a new file and save it.

```
def convert_distance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    print ("Converting distance in miles to kilometers:")  
    print ("Distance in miles:", miles)  
    print ("Distance in kilometers:", kilometers)
```

Here, the first line starts with the keyword `def`, indicating a function definition. What follows this keyword indicates that the function has the name `convert_distance` and takes a single argument.

When the function is called, the argument value provided is assigned to the variable `miles` (indicated in the brackets of the definition's first line). Note how the body of the definition is indented relative to the first line (which is not indented). This indentation is crucial — it signals that the indented lines belong to the body of the definition.

If you run this code, the interpreter will actually just read and remember the function definitions. We can then call the function as follows, i.e. using the function's name with the required argument value supplied in brackets.

```
>>> convert_distance(44)
Converting distance in miles to kilometers:
Distance in miles: 44
Distance in kilometers: 70.4
>>> convert_distance(122.5)
Converting distance in miles to kilometers:
Distance in miles: 122.5
Distance in kilometers: 196.0
```

## Your turn to write a similar function

### Task 4

**Task: Create a function definition version of your temperature conversion program in a new file (i.e. with a modified file name) and test that it works in the interpreter.**

**Tip:** When you receive a task like this, what questions will you ask yourself to refine the requirement?

The refined requirement is asking us to write a function that can automatically convert temperature from centigrade to fahrenheit and kelvin!

**More tips:**

$\text{fahrenheit} = \text{centigrade} * 9.0 / 5.0 + 32$

$\text{kelvin} = \text{centigrade} + 273.15$

## Print result not equal to storage result

For now, we just print out the result that we want from a function. However, print is not a equal to save the function's result.

**Note, print() just presents the result in the console rather than saving it to the computer, which means that although we saw the result, the computer has not store it and we will not be able to re-use it.**

Print() is important when you want to check or would like the user to see the result immediately without using computer storage. In the next section we will learn how to make your function return its values.

## Returning stuff from functions

### Return equivalent to save output in memory

A common mistake when learning Python is to confuse printing a value with returning the value. You can modify your function so that it returns the value it computes. As mentioned previously, only a function that returns a value the computer will store the function result in memory. You can then assign the return value to a variable for further calculations.

You can use functions to do things like crunch numbers or manipulate data and then “return” information from your function so that you can use it somewhere else in your program.

Functions can return a number, a string, a list or even another function (check out the recursive function in the MIT course’s [video6](#)). Today, we’re going to learn how to return a simple number value from your function.

Let’s have a look at the code below:

```
def add_two_numbers_and_return_value():
    number1 = 1
    number2 = 2
    answer = number1 + number2           # answer = 3
    return answer                        # 3

returned_value = add_two_numbers_and_return_value()

print (returned_value)

print(returned_value-5)
```

The variable names `answer`, `number1` and `number2` only exist inside this function. So, even though you’re returning `answer` from your function, it’s only the *value* associated with the variable that’s being returned (i.e. 3) not the actual variable `answer = 3`.

In order to be able to use this value we just returned, we need to assign a new variable name to it so we can do stuff with it. That’s what we’re doing with

`returned_value = add_two_numbers_and_return_value()`. Notice how we can then do stuff with `returned_value` after that, like print it.

**Try with the `def add_two_numbers_from_args(number1, number2):` version as well.**

**You also can change back to `print(answer)` to see if `print(returned_value-5)` worked or not.**

Similarly, in the mid-class challenge, you can return the function result as well.

In this example we define a function to accept an argument and return a value:

```
def convertDistance(miles):  
    kilometers = (miles * 8.0) / 5.0  
    return kilometers
```

We can then assign the returned value to some variable, e.g.

```
>>> k = convertDistance(10)  
>>> k  
16.0  
>>>k+30  
  
????? try yourself
```

## Return the last operation result

If you're returning something, the return statement should be the last line you want to run in your function. **Once the Python interpreter reaches that line, it won't execute anything in the function after it** (it'll go back to your main program and run the rest of the statements you have there).

Hence, it makes no sense to have code such as:

```
def myfunction():  
    return 1  
    return 2
```

This is because the second return command would never be reached.

**It's better you have a try with some examples you used previously to help you remember it.**

Sometimes, we may use return for the specific purpose of terminating function execution, which may not be the last line but it will not continue the code after it.

## Return None

### Task 5

What do you think the returned value will be if you don't have a return line in your function? Copy the code from `add_two_numbers_and_return_value()` above into your `my_arguments.py` file and run the file. Then remove only the line return answer and see what happens.

**Python functions always return a value**, even if there's no explicit return command. We can use the return on its own (i.e. without a return value being specified). In that case, it just returns None (the special Python 'null' value) by default.

## Best practice tips for writing functions (clean code tips)

- Try to give your functions descriptive names so it's easier to understand what they do just by looking at the name (you can also use comments to help with this).
- Try not to write giant functions! It's best to keep each function related to a specific task – this makes them easier to reuse.

Sometimes it helps to sketch out on paper what you're trying to do before you start coding.

We can also use many functions that are written in other files, which we will cover in the next section about how to import code!

## Importing Python modules and functions

The usual way to access pre-existing code — either code you’ve written yourself, or code in a Python library module — is to import it. For example, there is a Python module `math` providing many goodies, such as a variable `pi` defined with quite an accurate value of  $\pi$ . If we import this Python module with the statement `import math`, we would need to refer to this variable with the ‘long name’ `math.pi`. The alternative import statement `from math import *` brings everything (\*) in with a ‘local name’, i.e. so we could refer to the variable as just `pi`.

You can try to run `import math` in Python console then type `pi` and `math.pi`;

Then try `from math import *` then type `pi` and `math.pi` to see the difference.

### Separate function file and test file

Create a new file `test.py`, which imports from the code file containing your temperature conversion function definition and then includes a series of test calls to this function.

In here we can treat your function file as a Python module and only run tests or save results through the test file.

Note that, when importing from a local code file with a name such as `mycode.py`, the “.py” is dropped, i.e. we would use an import statement such as `from mycode import *`.

### Import beware

#### **Beware:**

Import statements appear at the top of the code file, which indicates that the imported material is assumed for use in the code that follows.

For best practise, **NEVER** put import statements anywhere else in your code. **NEVER** put import statement inside a function definition.

## Import command can be used in several different ways

- Simple import statement: `import pylab`

Import the contents of Python module pylab. But MUST prefix the Python module name to access its functions/values, e.g.

```
>>> import pylab
```

```
>>> pylab.sin(2.2)
```

```
0.80849640381959009
```

```
>>> pylab.pi
```

```
3.141592653589793
```

- Alternative form: `from pylab import *`

Imports everything (\*) from pylab Python module but now DON'T prefix the Python module name to use, e.g. refer to `sin`, `plot` & `pi` functions/values directly.

- Variant form:

```
from pylab import sin, cos, pi
```

Imports ONLY named items from pylab Python module, WITHOUT the need to prefix the Python module name. Imports ONLY named items from module, WITHOUT the need to prefix Python module name. Which means you can directly use `sin`, `cos`, `pi` in your Python code.

## Preferred version

**Note that this last version is preferred over \* version as it is considered better programming style.** Spyder even gives error-like messages for the \* version. Other reasons are:

- The Python module may contain thousands of definitions, most of which you won't use.
- This form means you are being explicit about what your code requires.

**However**, there can be a problem, different modules may use the same name to define different functions. e.g. One `cos` function might just return cosine of an angle in radians vs. another that might compute cosine of an angle between two vectors.

Importing as above (`from ...`), cannot tell definitions apart.

In practice, the definition loaded later will overwrite the one loaded earlier. This issue is avoided in the approach where you prefix the Python module name to use imported functions/values.

**Therefore, you need to always be aware of which module's function is being used.**

### Shorten module name

- ✓ Import form: `import pylab as pl`

This allows you to provide a shorthand name for the Python module (some have long names) then, use the shorthand name as the prefix to access the item, thus avoiding the name-clash problem, e.g.

```
>>> import pylab as pl
```

```
>>> import math as m
```

```
>>> pl.pi
```

```
3.141592653589793
```

```
>>> m.pi
```

```
3.141592653589793
```

```
>>> pl.pi == m.pi
```

```
True
```



>>>

## Homework

1. Review the examples from today's session to make sure you understand them.  
If you get stuck, don't be shy about asking an instructor for help!
2. Work your way through exercises 18 to 21 of [learnpythonthehardway/ex18](https://learnpythonthehardway.org/ex18).
3. Read Python official documents on Python module and package:  
[modules.html#packages](https://docs.python.org/3/tutorial/modules.html#packages)

## Chapter 4 - Conditionals

### Learning outcomes

- Use logic and Boolean operators.
- Write a conditional (“if/else”) statement.

### Recap from last session

- How to write a function?
- How to call a function?
- What are the arguments in a function?
- How to return function results?
- How to import code and functions?

### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how ‘if’, ‘elif’ and ‘else’ works! Once you’ve done that, explain how to test a statement with conditionals and Boolean type!**

### Programmes and algorithms

First, let’s familiarise ourselves with some concepts in computing and programming.

What are programmes and algorithms?

A computer programme is a set of instructions that tell a computer how to carry out a task. The instructions are written in a special formal language - a programming language.

Therefore the coding procedures are the most detailed requirements to ask computer to do what you want for a task.

In order to solve a programming problem, we need a step-by-step specification of the actions that must be taken to compute the result - this specification is called an algorithm.

An algorithm should be:

- Precise and unambiguous.
- Correct, i.e. finish and deliver the correct result.
- Efficient, but this depends on the task.

## Pseudocode

The same algorithm can be implemented in different languages or even stated in (pseudo) English. In this case, the algorithm idea is more general than a specific piece of code.

Here is an example of an algorithm being expressed in English-like 'pseudocode' -

Task: making a cup of (instant) coffee:

1. Fill kettle.
2. Boil kettle.
3. Put spoon of coffee in cup.
4. Fill cup (nearly) with water from kettle.
5. Add a dash of milk.

## Introduce conditionals

This 'algorithm' is just a simple fixed sequence of actions. You can handle more complex tasks by allowing **conditionals: actions that happen only under certain conditions** and loops: (groups of) actions that repeat until the result is achieved. We will cover conditions in this chapter and loops in later chapters.

## More pseudocode tasks

**Task 1: As you now understand what programmes and algorithms are, let's design a programme for supermarket shopping. Try to think about the flow by yourself first.**

1. Get a trolley.
2. While there are items on shopping list:
  - 2.1 Read first item on shopping list.
  - 2.2 Get that item from shelf.

- 2.3 Put item in trolley.
- 2.4 Cross item off shopping list.
3. Pay at checkout.

### Pseudocode with conditionals

**Task 2: how about supermarket shopping on a budget!! Think by yourself first before reading below :-)**

1. Get a trolley.
2. While there are items on shopping list:
  - 2.1 Read first item on shopping list.
  - 2.2 Get that item from shelf.
  - 2.3 IF item costs less than £3.
    - 2.3.1 Put item in trolley.
  - 2.4 ELSE
    - 2.4.1 Put item back on shelf.
  - 2.5 Cross item off shopping list.
3. Pay at checkout.

Can you tell the difference between tasks 1 and 2? Which one do you think is more realistic to real life applications? Let's learn the Control Structures to write those programs in code!

## Using “logic” in programming

Again, let's familiarise ourselves with some concepts first:

### Control structures

The way that program execution moves from one statement to the next is called the flow of control within a program. The major control structures are sequence, selection and repetition:

- Sequence: simply does one statement after the next.
- Selection: the flow of control is determined by a simple decision.
- Repetition: executes a statement or block of statements more than once.

## Boolean type and expressions

In order to make decision, we will need to learn Boolean first. We have seen Python basic types such as integer and float. A further basic type is boolean. It has only two values: True and False.

- A boolean expression is one that evaluates to True or False.
- The ‘decision’ of a selection structure is typically formulated as a boolean expression.
- Simple boolean expressions commonly involve a comparison operator.

## Logic and boolean type

Logic is an important concept in computer programming. Just like a “Choose Your Own Adventure” game, we can write programs that can take a specific action based on whether a condition is satisfied (“True”) or not satisfied (“False”). An example we use every day is signing into a service. If both your username and password match what’s in the service’s database when you’re signing in, then you get logged in; but if the username and password do not match then it means the condition is not true, so you’re not allowed access. In a Choose Your Own Adventure game, you might ask a player to choose between Door #1 and Door #2 and what happens next in the game is based on which door the player chooses.

## Logic operators

Here’s a list of logic operators and how they “evaluate”:

OPERATOR	WHAT IT CHECKS
and	Are both/all conditions true
or	Is at least one condition true
not	(self explanatory)
!=	not equal to
<b>==</b>	<b>equal to</b>
>	greater than

Just like in maths class, if you have things inside of brackets to evaluate, deal with the things inside the brackets first:

STATEMENT TO EVALUATE	ANSWER
<code>1 == 1</code>	True
<code>1 == 0</code>	False
<code>(1 == 1) and (1 == 0)</code>	(True) and (False) → False

Try the code below in a Python console and see if you get similar results or not:

```
>>> "this" == 'this'
True
>>> 3 >= 4
False
>>> 3 >= 2
True
>>> 5 != 3
True
>>> 5 != 'some string'
True
>>>
```

**Beware:** it's easy to use “==” in place of “=”, and vice versa; it's a very common coding error. Do you remember what “=” is used for?

**Short revision: You can form more complex conditions by using the boolean operators: and, or and not.**

**Given boolean expressions E1, E2 then:**

- E1 and E2 are True if both E1 and E2 are True, and False otherwise.
- E1 or E2 are True if either E1 or E2 are True, and False otherwise.
- not E1 is True if E1 is False, and True otherwise.

Example: testing for teenagers!

```
>>> age = 15
>>> isaTeen = age >= 13 and age <= 19 #set a variable in certain range
>>> isaTeen
True
>>> age = 22
>>> isaTeen = age >= 13 and age <= 19
>>> isaTeen
False
```

## Conditional statements (“if” statements)

We talked about logical operators a bit earlier and how they’re useful because we can use them to write programs that can take a specific action based on whether a condition is satisfied or not.

These are selection control structures achieved by use of if-else constructions.

You can think of each conditional statement as a different “branch” or “path” of code that you can follow – a bit like a fork in the road, you have to choose one path to follow because you can’t be on both at the same time.

### How if-else conditions work

If you have a conditional statement that evaluates to True, then Python will execute the block of code that immediately follows the if statement. Where the statement evaluates to False, Python will skip that block of code. Just like a function or a for loop, the code you want to run when a condition is True must be indented.

### Task 3

Write the code below into a new file called **numbers.py** and run the file.

```
number = input("Enter a number between 1 and 10: ")
number = int(number)  #Converts the input string to an integer

if number > 10:
    print ("Too high!")

if number <= 0:
    print ("Too low!")
```

1. What happens if you enter a number that's greater than 10?
2. What happens if you enter a number that's less than 0?
3. What happens if you enter a number between 1 and 10?
4. What happens if you remove the line with `number = int(number)`?

If none of your conditions are met, Python will simply carry on through the rest of your code. In the example above, we didn't write any code for the situation where the user enters a number between 1 and 10, so you didn't see any sort of helpful message or feedback in your terminal.

Notice how we used `int()` to convert the user input into a number. `input()` treats all input as a string.

## Else statements

An else statement is like a catch-all for when none of your conditions are satisfied. That means **you can only have one else statement**, and **it can't exist on its own without an if statement**. If you think you need more else statements, you actually need more if statements!



## Syntax of if-else statements

The basic form of if and else is shown here:

```
if CONDITION:
    CODE-BLOCK-1

else:
    CODE-BLOCK-2
```

### Task 4

Using the code from the previous task:

1. Add an if statement that prints a message if you enter a number between 1 and 10.
2. Instead of the if statement you just wrote, now use an else statement to do the same thing.  
(This is a bit of a trick question!)

## Reason for using an else statement

Using else statements isn't mandatory, but they're handy for debugging because they can help to catch unexpected behaviour in a program. An example of omitting the else is shown here:

```
if altitude < 100:
    print("Warning!")

print("Time to bail out.")
```

## Three or more options -elif statements

You can also chain a series of cases, using the keyword elif,

### Task 5, try the code below:

```
if age < 13:
    print('child')

elif age < 18:
    print('teen')
```

```
elif age < 65:  
    print('adult')  
  
else:  
    print('pensioner')
```

Although this would work equally well if each elif was simply another if instead.

## Order is important

**Note that in the above example, the order of the cases matters: - reordering them would give incorrect behavior.**

Consider what would happen with code where the cases were reordered:

```
if age < 65:  
    print('adult')  
  
elif age < 18:  
    print('teen')  
  
elif age < 13:  
    print('child')  
  
else:  
    print('pensioner')
```

Can you work out how the above code would behave incorrectly?

## Homework

1. Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!
2. Complete exercises 11, 12, 27, 29, 30 and 31 on [Learn Python The Hard Way](#). Don't worry about doing the study drills that ask you to write things out 50 times or upside down – the point of the homework is to help reinforce what you've learned in class, so use your judgement on what's most helpful for your learning journey.

3. Try the string challenges through code bat Python [Warmup-1](#) and [Warmup-2](#), you may come across some concepts that we haven't covered yet, e.g. arrays. You can skip them for now and come back later.

## Chapter 5 - Object Oriented Programming

### Learning outcomes

- Understand what object oriented programming is.
- Understand what inheritance and association are.
- Learn the differences between public and private attributes.

### Recap from last session

- What is a program and what is an algorithm?
- What is Boolean type?
- List some logic operations.
- How to use logic operations to test statements with Boolean type?
- What is an if statement used for?
- How to do multiple options with logic?

### Object and classes

#### What is a class

Similar to the function-based term `def`, class concerns the definition of things. While `def` is used when defining functions, `class` is used for defining classes. A class is simply a logical grouping of data and functions (functions are often referred to as methods when defined within a class).

How to decide this "logical grouping"? A class can contain any data you choose, and any functions (methods) you want, but rather than putting random things together under the name "class", it's better to **create classes with a logical connection between things.**

## Class and object

Classes are a modeling technique, so basically a way of designing programs. When you design and implement your system using this technique, it is called Object-Oriented Programming. Also, while "classes" and "objects" are words that are often used interchangeably, they are different and understanding the difference is the key to understanding what they are and how they work.

### So everything has a class?

**One way to think about classes is as blueprints for creating objects.** Defining a Customer class using the class keyword, doesn't create customers. Instead, it creates a blueprint or an instruction manual for constructing "customer" objects.

#### Task 1 try the following example code:

```
class Customer(object):

    """A customer of ABC Bank with a checking account. Customers have the following properties:

    Attributes:

        name: A string representing the customer's name.

        balance: A float tracking the current balance of the customer's account.

    """

    def __init__(self, name, balance=0.0):

        """Return a Customer object whose name is *name* and starting balance is *balance*."""

        self.name = name

        self.balance = balance

    def withdraw(self, amount):

        """Return the balance remaining after withdrawing *amount* dollars."""

        if amount > self.balance:

            raise RuntimeError('Amount greater than available balance.')
```

```
self.balance -= amount  
  
return self.balance
```

```
def deposit(self, amount):  
  
    """Return the balance remaining after depositing *amount* dollars."""  
  
    self.balance += amount  
  
    return self.balance
```

**Notice, there is nothing that marks the end of the class block. Python knows the end of a class by the indentation of the code!**

The class `Customer(object)` line does not create a new customer. Again, just because we've defined a `Customer` doesn't mean we've created one; we've just stated the way to create a `Customer` object. To create one, we must call the class's `__init__` method with the proper number of arguments (minus `self`, which we'll get to in a moment).

To create `Customer` objects using the blueprint we have now defined, call the class name like you would call a function:

```
jason = Customer('Jason Taylor', 1000.0)
```

This line means "use the `Customer` blueprint to create a new object, which will be called `jason`."

The new `jason` object, is the realized version of the `Customer` class, known as an instance of that class. Before we called `Customer()`, no `Customer` object existed.

```
>>jason.balance  
  
1000.0  
  
>>>jason.name  
  
'Jason Taylor'
```

We can, if we wish, create as many `Customer` objects as we want (think about an excel spreadsheet). But there is only one `Customer` class, regardless of how many

instances of that class are created (similar to a spreadsheet, where no matter how many rows you create, it doesn't change the number of columns).

## self

So, you may have noticed that there is a `self` parameter in all of the Customer classes. What's that? It's how the instance refers to itself. Put another way, a method like `withdraw` defines the instructions for withdrawing money from some abstract customer account. Calling:

```
>>>jason.withdraw(100.0) # as jason is a object, you can use .function() to run any function only work  
for this type of object
```

```
900.0
```

```
>>>jason.balance #check balance after withdraw
```

```
900.0
```

puts those instructions to use on the `jason` instance.

So when we say `def withdraw(self, amount):`, we're saying, "here's how you withdraw money from a Customer object (which we'll refer to as `self`) and a monetary figure (which we'll call `amount`). `self` is the instance of the Customer that `withdraw` is being called on. That's not me making analogies either, `jason.withdraw(100.0)` is just shorthand for `Customer.withdraw(jason, 100.0)`, which is also a perfectly valid (if not often seen) line of code.

## \_\_init\_\_

`self` may make sense for other methods, but what about when used in `__init__`? When we call `__init__`, we're in the process of creating an object, so how can there already be a `self` to refer to? Python allows us to extend the `self` pattern to when objects are constructed as well, even though it doesn't exactly fit. Just imagine that `jason = Customer('Jason Taylor', 1000.0)` is the same as calling `jason = Customer(jason, 'Jason Taylor', 1000.0)`; the input '`jason`' instance, which is called by the Customer class when constructing the object, is also part of the result.

This is why when we call `__init__`, we initialize objects by saying things like `self.name = name`. Remember, since `self` is the instance, this is the same as saying `jason.name = name`, which is the same as `jason.name = 'Jason Taylor'`. Likewise, `self.balance = balance` is the same as `jason.balance = 1000.0`. After these two lines, we consider the Customer object "initialized" and ready for use.

### Fully-initialized object

Be careful what you `__init__`. After `__init__` has finished, the caller can rightly assume that the object is fully ready to use. Meaning that after `jason = Customer('Jason Taylor', 1000.0)`, we can start making deposit and withdraw calls on jason; that jason is a **fully-initialized** object.

In some books people refer to `__init__` as an object **constructor**, however, there is a debate about that in [stackoverflow](https://stackoverflow.com/questions/1136650/what-is-the-difference-between-__init__-and-__new__), so let's just treat it as an initialiser.

## Inheritance

In OOP, you can also create subclasses that create objects in different categories. This is called inheritance.

In inheritance, the subclass is a category of the superclass, so we create subclasses based on the superclass. A subclass (e.g. Dog class) can inherit all possible methods and attributes from the superclass (e.g. Animal class).

### One direction inheritance

Inheritance does not allow the superclass to use any functionalities created in the subclasses, but inheritance extends the class, so we can add more attributes and behaviour to the inherited class.

For example, we have superclass Animal that has a method eat, and the subclasses Dog and Cat can not only inherit the eat method but also have their own bark and meow methods.

```
class Animal():
    def eat(self):
        print('yum')

class Dog(Animal):
    def bark(self):
```



```
        print('Woof!')
class Cat(Animal):
    def meow(self):
        print('Meow')
```

You can test the class by creating objects as in the code below in your Python console:

```
>>> Snoopy = Dog()
>>> Snoopy.bark() #subclass method
Woof!
>>> Snoopy.eat() #superclass method
Yum
```

Please note, we have not set an `__init__` method for this class so we can create objects directly without giving any input parameters. You may also notice, inside the `subclass()`, we need to put the superclass name to clarify the link between subclass and superclass.

## **Task 2, create your own cat object using the code above.**

For another example, a superclass Robot has a method move, and the subclasses CleanRobot and CookRobot not only inherit the move method from the superclass, but also have their own clean and cook methods.

```
class Robot():
    def move(self):
        print('...move move move...')
class CleanRobot(Robot):
    def clean(self):
        print('I vacuum dust')
class CookRobot(Robot):
    def cook(self):
        Print ('I make rice')
```

You can test the class by typing the below code into your console:

```
>>> Nana = CleanRobot()
```

```
>>>Nana.clean()
I vacuum dust
>>>Nana.move()
...move move move...
```

### Task 3, also try your own cooking robot object using CookRobot class :-)

We have seen many advantages to inheritance here, such as the fact that you don't need to repeat much code in the new classes, as they are inherited from the superclass. You can find many more of those examples in the Python library.

However, what if you needed to create a robot that can clean, move, bark and play games? Composition can solve this problem.

## Association (composition and aggregation)

There is not only the inheritance relationship but also association relationship between different classes. In fact, you may find that you use association more than inheritance in real work scenarios.

### Composition

One type of association is composition, for example a subclass (e.g. Body class) composed of one or more instances of a superclass (e.g. Human class). In composition, we can think of the superclass as a container and the subclasses as the contents inside the container. So if the container (superclass) has been deleted, then the contents of the container (the subclasses) no longer exist.

Therefore let's consider how to create a robot that can clean, move, bark and play games?

### Task 4 Let's organise some code first:

```
class Dog(Animal):
    def bark(self):
        print("Woof!")
```

```
class Robot():
    def move(self):
        print('...move move move...')
class CleanRobot(Robot):
    def clean(self):
        print('I vacuum dust')
class SuperRobot():
    def __init__(self):
        self.o1 = Robot()
        self.o2 = Dog()
        self.o3 = CleanRobot()
    def playGame(self):
        print('alphago game')
    def move(self):
        return self.o1.move()
    def bark(self):
        return self.o2.bark()
    def clean(self):
        return self.o3.clean()
```

The idea is that as we are including objects which have come from other classes, we still need to define the methods, but these can just refer to the object.method that is based in the other classes. Test this by typing the below code into your console:

```
machineDog = SuperRobot()
machineDog.move()
...move move move...
machineDog.bark()
Woof
```

In general, composition is better to use in design models than inheritance, because it is selective about which methods the new class gains. This is why many people believe that you should use composition when you can and only use inheritance when you must, because as soon as you write inheritance from superclasses, you have no choice but to inherit everything from the superclass which may result in lots of constraints e.g. name clashes or inheriting characteristics that are not required or

desirable. [Read more example](#) to practise how to convert inheritance to composition.

## Aggregation

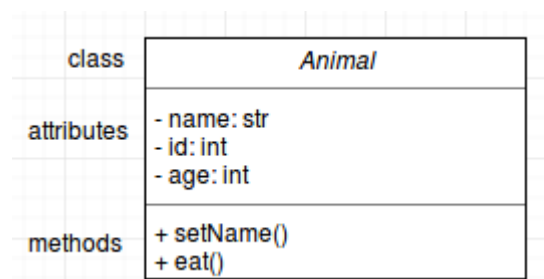
The other type of association is aggregation, which can be considered a weaker form of composition. With aggregation, once the container (superclass eg. Team) has been deleted, all of the contents (subclasses eg. players) can still exist without the container. We will not cover aggregation in this course, but you can find out more information on [stackoverflow](#) to compare composition with aggregation.

## Introduction to unified modeling language

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system. People can use UML in many different ways with different tasks. It is also a great tool for object oriented programming task designs.

### UML with OOP

Let's watch a [video](#) about how to apply UML in OOP first. By following the video's instructions, can you draw the UMLs for the example shown in the video? (Hand draw or simple ppt/drawing tools is fine, no need to use lucidchart.)



**Task 1: Animal is the super class (abstraction) with attributes: name, id and age (think about what the data types are for them?); there are 2-3 subclasses**

**e.g. Tortoise, Otter, etc. What is the relationship between those subclasses and the superclass?**

## Visibility

Also, think about the attribute and function visibilities while drawing the charts.

+ public:

Can be accessed by other classes.

- private:

Cannot be accessed by any other class or subclass.

# protected:

Can only be accessed by the same class or its subclasses.

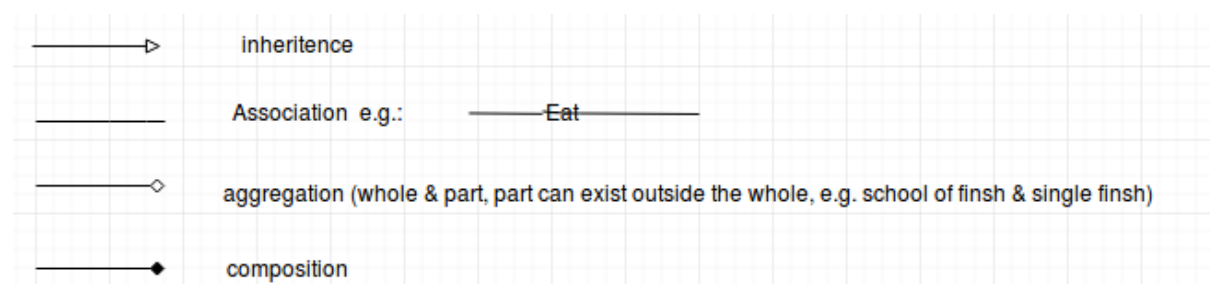
~ package/default (rarely used):

Can be used by any other class as long as in the same package.

**Although attributes are usually private - and methods are usually public +, think about why that is?**

The reason that we keep our variables/attributes private, is because we don't want anyone else to depend on them. We want to have the freedom to change their type or implementation on a whim or impulse, without them affecting anything else outside.

## Class relationships



**Task 3: Now try to draw the composition example (child object wouldn't exist without a parent). The example is visitor centre as the superclass, with lobby and bathroom as subclasses. Apart from the composition relationship, what else will you add in this UML case?**

Find more inheritance, composition and aggregation examples [here](#) and see if you can draw the UML for them accordingly!

### Multiplicity numerical constraint

Have you guessed correctly? Yes, you need to add the numerical constraint.

0...1:	0 to 1 (optional)
n :	specific number
0...*:	0 to many
1...*:	1 to many
m...n:	specific range
1:	1 and only 1

**Think about why this is important. In the meantime, try to draw the UML for all the above examples from this chapter.**

## Public and Private

As you can see from the previous examples, the variables in the superclass are public variables which any subclass has access to. However, we can also rewrite those public variables inside a subclass to make them private. Initially, using the private attribute helps to distinguish between different functionalities, if many people (or one person coding over different times) have used the same name for multiple public variables then the program can soon get confused. The private attribute is also very important for security and you can read more about that aspect [here](#) and [here](#).

## What problem does OO Programming solve?

### Disadvantages of not using OOP

In simpler 'imperative' (non-OO) programming, the program may be one "long" list of commands. You might group smaller sections of statements into functions / subroutines and it is common for data to be 'global' (i.e. accessible from any part of the program) hence, any statement/function might modify any piece of data.

As a result of that, it is considered to be very difficult to build large programs with this approach as it allows bugs to have far-reaching consequences and they may be very hard to track down.

A large program may also require several programmers working on different parts of code. When one programmer's code makes changes to data, the second programmer may not anticipate this. As a result, the second programmer's code now crashes or appears to work but produces incorrect results.

### Object-Oriented Programming achieving modularity

The solution to this problem is through modularity; by breaking the code down into suitably-sized chunks or modules. This limits the interactions between different components.

Object-Oriented Programming is the most successful approach to achieving modularity. The modules are realised as Classes, which group functionality / data-handling together. **External code should not interfere with the 'inner workings' of a class, but instead should only access/modify data stored by a class via methods that the class provides.**

### Abstractions

These methods together constitute an interface for external code to use the class. In theory, I should be able to change the inner workings of my class, without affecting the external code that uses my class. i.e. Provided the interface stays the same and the code delivers the same functionality. Some people refer to this as abstraction.

In industrial work, OOP is all about abstractions, by hiding implementation, it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation. We don't want to expose the details of our data. We want to express our data in abstract terms.

## OOP in real life applications

### Inheritance and Real-world Programming Problems

Here's an example. Say that a company wants to keep records of its employees (past and present) but, the company has various kinds of employee, such as:

- employee (standard, full-time)
- employee (standard, part-time)
- employee (executive)
- employee (retired)
- employee (zero-hours contract worker)

Records of different employee types will have much in common — both for the information stored (e.g. name, D.O.B., address, etc) and actions required (e.g. function to update address, etc).

But records of different employee types will also have differences — again for both information and actions. An executive may need additional fields for “type of company car”, “expenses quota”, etc, while different National Insurance contribution calculations may be needed for “zero-hour” vs. “full-time” employees.

Therefore you could create a superclass for Employee Record, to capture the common aspects and subclasses for different employee types — so that different characteristics can be handled, without redundancy.

**Task 5 (homework), try to create your own employee super class and different subclasses with different specified methods.**



## Organising your Code via OOP

Your programming task may not be complicated enough to need inheritance, but even so, OOP Classes are an excellent way to organise your code. Without OOP, you may have a large collection of functions, meaning that you must always check what is the right data to provide as the input and must collect and store results of the function calls, etc.

It is often much tidier to collect them together, as a Class initialisation method can define (and initialise) all attributes needed to store the data to be handled, then provide as many methods as needed to operate on the data, with results often being stored back into the object. Conceptually, this is a much cleaner way to work.

## Chapter 6 - Command Line and Git with Python

### Learning outcomes

- Navigate through folders using basic command line commands.
- Create a new folder and file using your command line.
- Use basic git commands in the command line to commit code.
- Manage and avoid conflicts in git.

### Let's get started!

Today's session should be completed in groups or pairs, so find a buddy or several to team up with! We're going to review command line and git basics, to help you get more comfortable with using them for your projects; then you'll have the rest of the session to go over things you've learned on the course so far and ask any questions you might have.

### Run Python using the command line

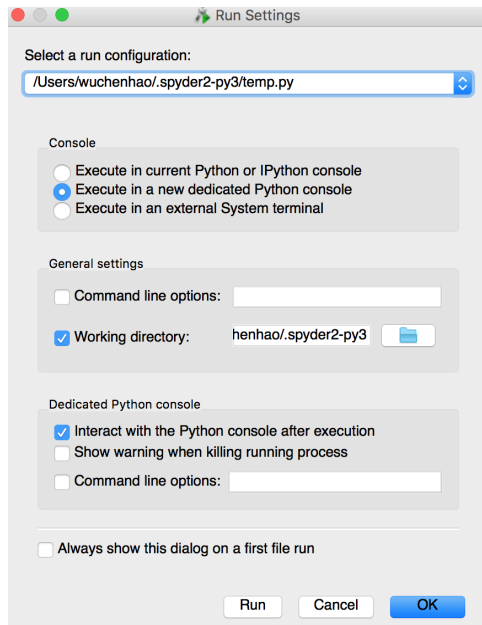
Previously we have tested code in the Python console or run Python scripts through Spyder. In this chapter, let's run Python through the command line, which will be a very useful tool when you build big programmes and need to push to git frequently.

It might seem easier at first to use GitHub Desktop to manage all your git needs, but it really is better to use the command line instead. Believe it or not, it can actually save you time and when you need to troubleshoot or do more advanced things with git, it's much easier to do them in the command line.

Using command line to run Python is also a great way to package your code in another script. E.g. in bash and tc shell languages which you may encounter in your working environment.

So here are the ways to run Python through the command line:

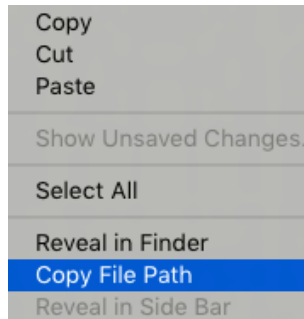
First, find the path of your Python files. Remember the configure setting window when you first ran Python?



You can find the path through the working directory section above. If you use other editors e.g. Atom, Notepad etc please read below:

MAC / Linux users	Windows users
-------------------	---------------

To begin, right click anywhere inside the file you just created and pick the Copy File Path option:



Next, you'll need to open your command line. You'll remember this from the pre-course notes. Note: this is your Terminal app.

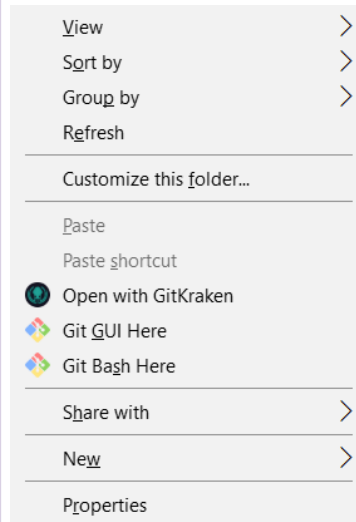
Once the command line is open, type `python` (always in lowercase) followed by a space and then paste the file path you just copied, inside quotation marks. The quotation marks make sure that everything works properly if you have spaces in any of your folder names.

What you type will look something like this:

```
python "/users/andreas/cfg-python-work/
hello.py"
```

Now all you need to do is hit **enter** and you should see Hello, World! printed in your command line window.

Find the location of the file you just created (use your file Explorer) and right click anywhere in there. Now click on **Git Bash Here**. A terminal will open at the file location.



Once the terminal is open type:

```
Python hello.py
```

Now all you need to do is hit **enter** and you should see Hello, World! printed in your command line window.

## Steps to run a Python file with command line.

As you complete tasks in this course, it's handy to remember what steps to follow to run a file:

1. Make changes to your Python file (don't forget to save them).
2. Make sure the file path is copied (or use the up arrow in your command line to display the last command you ran).
3. In your command line, repeat the rest of the steps from the Hello World exercise in chapter 1 to run your file (i.e. after pasting the file path you just copied, inside quotation marks, hit enter).
4. Then run the code by typing: `Python code_file_name.py`.

E.g. `python hello.py`

## Navigating using the command line

So far you've been running files from your command line, but you've been copying and pasting the file path each time to do that. With a few simple commands, you can find your way around your files using your command line in no time! -- we do this through bash shell command.

Now we will cover how to:

- Change directories (change the folder you're in) using `cd`.
- List the contents of a directory using `ls`.
- Print the file path to your current directory using `pwd`.
- Create a new folder `mkdir`.

The command line is a powerful thing. You can use it to tell your computer to do almost anything! We're showing you some basic commands today, but if you want to learn more outside of class, make sure you understand what a command is supposed to do first before you try it out, especially if it involves changing or deleting

stuff. If you accidentally ask your computer to erase all of your data, it will do so, and then you'll be stuck!

## **Windows, Mac, Linux command line difference**

The command line interface (CLI) that comes with Windows is called Command Prompt. The CLI that comes with Mac is called Terminal. They both basically serve the same purpose, but, because the Windows operating system is built fundamentally differently from a Mac's, the commands you use to do something often aren't the same across both Command Prompt and Terminal. For example, in Command Prompt, if you want to list the contents of the folder you're in, you'd use the command `dir`, whereas in Terminal, you'd use `ls`.

## **Practise with bash shell**

For this reason, if you've got a Windows computer we will be using the Git bash command line tool you installed earlier.

## Task

Work through the following exercises with your buddy/group:

There are a few simple commands you can use to find out where you are in your computer's file system.

1. Print the file path to your current working directory (i.e. the folder you're in) by typing `pwd` and hitting enter.
2. List the contents of the directory you're in by typing `ls` and hitting enter.
3. Can you tell what's a file and what's a folder?

You can navigate to a different folder using the change directory command `cd`.

1. Do you see your Documents folder in the list?
2. Navigate to it using `cd <folder_name>`. If the name has a space in it, like "My Documents" then use the format `cd "<folder_name>"` instead. E.g. `cd "My Documents"`.

List the contents of your Documents folder using `ls`.

## Creating new folder and files using bash

You now know the basic commands you need to find your way around! Let's use a few commands to make a new folder, create a new file and open the new file.

1. First, navigate to your Python course folder. If you already know the file path for it, you can use it in the format `cd <folder_name1>/<folder_name2>` to get there more quickly.
2. When you get to your project folder, create a new folder in it called **my\_new\_folder**, using the command `mkdir my_new_folder`.
3. Type `ls`. Do you see your new folder in the list?
4. Navigate to your new folder using the command `cd my_new_folder`.
5. Type `ls`. Notice how the folder is empty because we haven't put anything in it yet.
6. Create a new Python file called **my\_new\_python\_file.py** using the command `touch my_new_python_file.py`. You can check that it's been created using `ls` to list the contents of **my\_new\_folder**

7. Open your new file using the command `open my_new_python_file.py`. Don't forget to include the file extension `.py` after the name!
8. When you want to close the file you just opened, you don't have to do anything in command line; just close it like you usually do.

If you want to rename or delete anything, for now, it's best to do that in Explorer (Windows) or Finder (Mac) like you usually would, so you don't accidentally delete something important.

### Shortcut and navigate tricks

You can navigate "up" (backwards) through your file system too.

1. Check where you are using `pwd`.
2. Type `cd ..` and hit enter. What happened? What directory are you in now?
3. Now type `cd` (without any dots). What happened? What directory are you in now?

In the first example, you moved up one level to the "parent" folder. In the second example, you were taken all the way back to your "root" folder where we started this exercise (usually this is `C:/User/`).

### Task, give this a go:

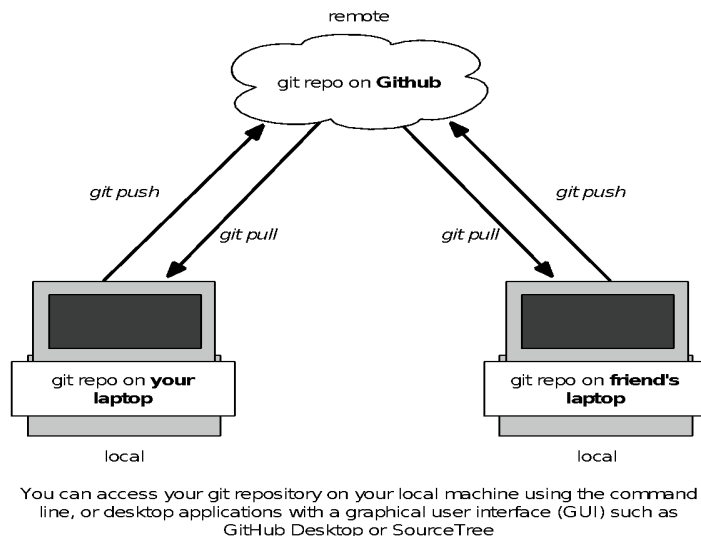
1. Press the "up" arrow on your keyboard (don't hit enter though). What do you see?
2. Press it again a few times. Notice anything helpful?

Instead of having to type the same thing over and over again, you can use the up arrow to save time running commands, for example, when you're running your Python file over and over again to test it.



## Working with git in the command line

You'll remember from module 1 that git is used to track your project files over time and makes it possible for different people to contribute code to the same project files. With git, you can see what changes have been made over time and go back to a previously working version of your code if something goes wrong.



You could use GitHub Desktop for all your git needs, but now that you know how to find your way around your files and folders using the command line, we can start to do some git from the command line instead.

Even if using git is still fresh in your mind, it's important that you don't rush ahead with these exercises. We'll be switching between GitHub and your command line and things need to be done in a really specific way, so please follow what your instructors say step by step.

Also, it might be handy to get a GitHub commands cheat-sheet for when you cannot remember all the instructions: [git-cheat-sheet-education.pdf](#)

## Setting up a new repository

A git **repository** is the thing that holds all of your project files and folders. You'll need to set one up for your course project. While you can do this on your computer and

link it with your GitHub account, today we'll create the repository in GitHub and "clone" it to your computer instead.

1. Open your browser and log into your account on [GitHub](#).
2. Click on the tab called "Repositories".
3. Click on the button that says "New".
4. Give your new repository a name and then click "Create Repository". Don't worry about the other fields for now.

GitHub now shows us a page with a bunch of different options for uploading our code. We don't want to upload any code yet; we just want to make sure we've got a repository for our project on our computer that's linked with the one we just made on GitHub.

1. Open up your command line.
2. Navigate to your Python course folder using `cd`.
3. Create a copy of your new GitHub repository on your laptop, using the command `git clone https://github.com/YOUR_REPOSITORY_NAME`. This creates a new folder for your project, which contains your git repository. It's also where you'll put all your project files. You can check that the folder was created by using `ls` in your command line.

**Note:** if you ever wonder whether the directory is a git repository or not, type `git status` on the command line/terminal (in the directory). If this is a repository you will see the status of it, otherwise git will show a message saying that location has not been initialized as a repository.

Now we're going to do a quick refresher on committing and pushing a file to our repository. Remember, you need to be in the folder for your project when you do this, because that's where your repository is. We're already in the right place, so we don't need to worry about this today.

1. Make a new file using the command `touch <filename>`. Name the file whatever you want but make sure it's a Python file (a file that ends with `.py`).

2. Open your new file using `open <filename>`.
3. Add the line of code `print "Hello World!"` and save the file.

Before we can commit or push any changes, we have to "add" them to a "staging area". A staging area is just like a waiting room at a train station. Passengers there haven't boarded the train yet - they still have the option of committing to their train journey, or changing their mind and not getting on the train.

1. Type `git status` to see what's in your staging area. Notice how your file is red. That's because git sees it's a change, but it hasn't been added to the staging area yet.
2. You can add files to the staging area in your command line using `git add`. This adds all of your changes, including new files, deletions, and modified files. To add a single file, use the command `git add <filename>` instead.
3. Type `git status` again. See how your file is now green?
4. Now that you've added a file to the staging area, you can take the next step and **commit** it, like this: `git commit -m"put a little message here"`. The message you include is important, because it reminds you and your team members what the change you made was about. Keep in mind that this only commits (saves) the change to your local machine. GitHub has no idea these changes even exist!
5. To make sure your changes are stored in the cloud on GitHub, you need to **push** your changes to GitHub's server. You can do this using the command `git push -u origin master`.
6. Head back to your repository on GitHub.com. Do you see the files you just pushed?

Just like it's good to test your work in smaller bits as you go along, it's best practice to commit frequently, when you've got something smaller working. This way, if something goes wrong, you don't have to start over from the beginning because you forgot to commit your code.

## Managing conflicts in git

Chances are, as you work on your project, a few team members will make different changes to the same file. Sometimes these changes will conflict with each other, and someone will get a message in their command line saying they can't push their code because it's not up to date. This is called a **merge conflict**, because it means they can't merge their changes with the master code branch.

When this happens, git will also add some markers in your file to tell you where the conflicts are. The person who tries to sync last will be the one that has to deal with the merge conflict.

Merge conflicts are annoying, but you can fix them. Here's how:

1. Open the problem file in your text editor
2. Merge conflicts have lines around them that look like this <<<<<<<<<. Your document may have a few lines like that, depending on where the conflicts are. You might see ===== as well.

```
<<<<<<< HEAD:config/environment.rb
# this is my new version
foo = Bar.new
=====
# it conflicts with this old one
foo = Baz.new
>>>>>>> Decided to use a softer route t
end
```

3. Choose the lines of code you want to keep, editing them if you need to.
4. Once you've got the code the way you want it, remove the <<<<<<<<< bits you see and save your file
5. Use `git add <filename>` to add the updated file to the staging area

## 6. Commit and push the change

# Avoiding conflicts in git

You can avoid merge conflicts by making sure you're using the most up date code from GitHub, and regularly pushing code/submitting **pull requests**.

Remember that committing code just commits it to your local team. Your team members won't see anything until you push it to GitHub. A pull request, is a request you make when you're proposing your changes to your team's code, and requesting that someone pull in your contribution to GitHub.

1. Get the latest code from your team's repository on GitHub by using `git pull origin master` to update your local repository
2. Make the changes you want to make to your code
3. Add any changes to the staging area, then commit them to your computer's repository
4. Push your changes to your team's repository on GitHub
5. Git push - sync upload the changes back to github so others can see them

You can read more about merge conflicts [here](#) and [here](#) if you're interested!

## Part 2 - Design a Simple Programme

### Chapter 7- Debugging and Intro to Project Management

#### Learning outcomes

- Learn to use the print function for debugging.
- Learn to use breakpoints for debugging.
- Start using best practice as you develop your projects.
- Combine this habit with project management skills.
- Training to have a good step-by-step design and testing habit while coding.

#### Recap from last session

- How to run Python through the command line?
- What bash shell commands you have learned to navigate directories through the command line?
- What are the steps for when you use git through the command line?

#### Let's get started!

Meeting an error report is very common during coding, and the way to find and solve errors in code is called [debugging](#). This word was actually created by a famous female computer scientist [Admiral Grace Hopper](#) in the 1940s. In this chapter you will learn different ways to debug and use tools to understand code better. This chapter will also introduce some project management skills and ways to help you build step-by-step logical ways to think when you design a project.

## Use print function for debugging

We have learned about the print function earlier, which is a great way to interact with your code and check results. It is also a great tool for debugging, especially for beginners. Whenever you set variables or write functions to make changes, it is always good to use print to test and check understanding.

**Task 1 try to run the code below in your script or Python console, Mmm.. if you directly use the user input as a number to do a operation, what will happen?**

```
userInput = input('please give a number ')
result = userInput - 2
```

Yes! You get an error:

```
File "/Users/wuchenhao/Documents/ch7_printDebug.py", line 3, in
<module>
    result = userInput - 2
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

### How to check where is wrong?

When you read user input and use it in a function, you may print it out first to check if you used the correct argument to refer to the user input. You can also print the 'data type' of the users' input.

**How about this time running a print to check the data type of the user input?**

**Try the code below:**

```
userInput = input('please give a number ')
print(type(userInput))
```

If I give the number 5 as the user input, you can see that the result in the console is:

```
please give a number 5
<class 'str'>
```

### Read error message and check/print variable accordingly

Where Python is very good is that it tells you exactly which line caused the error (this is true for simple code, but for much more complex code you may need to find a way to trace back and use the step in and out method below, but the error message will

still give some clue. E.g. a variable name). Therefore it is very important to patiently read the error message whenever you meet one, they are a great help!

### **Your turn to solve the problem!**

**Think about why is the data type a 'str' -- a string, not a 'int' -- integer? Try to find out how to cast one datatype to another in Python.**

**Now, have you found a way to solve the error? Try to solve it yourself and ask your instructor if you get stuck.**

### **Datatype errors are very common**

The above example is very common when people develop software, as lots of errors are due to using incorrect data types.

In daily coding work, you can redefine the variable as many times as you like. When you learn about using 'while' and 'for' loops, you can set variables to update by themselves. However in more complex settings, or when you revisit code you wrote months ago, or when you try to understand code from others, it is easy to get lost. Therefore using print is a great way to check understanding, clear out doubts and avoid mistakes as well.

### **Do remember to remove the print debugging lines afterwards**

Also it would be a good idea to remember to remove the debugging print statements once you're done debugging! - this is for security reasons. You don't want anyone outside to see your code printed details! We will mention this more in module 3. :-)

## **Use breakpoints for debugging**

Apart from using the print function, you also can use breakpoints and 'step in', 'step out' to check the code flow and correct errors.



## Beware of line number and set breakpoints

In Spyder, double click next to the line number of your code to set up a breakpoint.

### Task 2, write this nested function in your editor:

```
userInput = input('please give a number ')
```

```
def simpleOperation(userInput):
```

```
    intInput = int(userInput)
```

```
    result = intInput - 2
```

```
    return result
```

```
def nestedOperation(result):
```

```
    result = simpleOperation(userInput)
```

```
    result2 = result * 2
```

```
    return result2
```

```
result = simpleOperation(userInput)
```

```
result2 = nestedOperation(result)
```

```
print(result2)
```

If you put the breakpoint at line 21: `result = simpleOperation(userInput)`, a red circle should appear to show that you have set up the breakpoint successfully:

```
● 21 result = simpleOperation(userInput)
```

## Debug mode intro

After setting up a breakpoint, you can run your code in debug mode, for which you do not use the run button, but the debugging buttons as shown on the toolbar in Spyder,




**The first button** is for you to start running your code **until the break point**.

**The second button** allows you to **run your code line-by-line** until the breakpoint.

**The third one** is for stepping into the sections (class and functions) that you would like to dig into more and the fourth button is for you to step out when you feel that the error is not related to the current section.

**The fourth button** is for you to go to the next breakpoint (if you have setup multiple ones) and **the last, square shaped button** is for you to exit debugging mode and go back to normal coding mode.

## Debug mode practise

I would suggest that you try the second debugging button  to test it out. You will probably get similar results to mine:

```
In [13]: debugfile('/Users/wuchenhao/Documents/ch7_breakPDebug.py',
wdir='/Users/wuchenhao/Documents')
None
> /Users/wuchenhao/Documents/ch7_breakPDebug.py(1)<module>()
----> 1 userInput = input('please give a number ')
      2
      3 def simpleOperation(userInput):
      4
      5     intInput = int(userInput)

ipdb>
please give a number 7
None
> /Users/wuchenhao/Documents/ch7_breakPDebug.py(21)<module>()
      18     return result2
      19
      20
1--> 21 result = simpleOperation(userInput)
      22 result2 = nestedOperation(result)
```

Where it will interact with you every time you press this button and tell you which line it has just run.

**Have a play with all those buttons and chat with your partner about your code line by line and your findings.**

## Breakpoint summary and further resources

Breakpoints are very useful tools while checking complex code and you can run the code until the line that you want it to stop at. Therefore, it helps you to confirm which parts of the code do not have problems and which parts of code are related to the error.

Here is a very good [video](#) to explain how to use breakpoints to debug in Spyder, and a step by step [wiki](#) that explains it with some examples. You can also ask questions in stackoverflow or Github as an example in [here](#).

## Best practice for coding

### Think professionally

Though you're just starting to learn Python, it's still good practice to think like a **professional** product or software developer as you progress through this course!

"Best practice" isn't just about tracking and squashing bugs – it's also about:

- **How you lay out your code.**
- **How you manage your projects.**
- **How you work with others!**

Professional software developers almost always have to work with others, whether those people are teammates, clients or a colleague they're **pair programming** with (plus peer reviewing!).

### Different styles

Every developer has their own preferred style of writing code. For example:

- How they use **spaces between lines and blocks of code.**
- How they **use comments.**

- Whether to use **single or double quotation** marks.

But perhaps the **most important** thing to remember is to try to **keep your code neat, organised and easy to read**. This is especially important when you're working in a team!

(Just thinking about when people play music together in a band. You will not only have your own part to organise and to let yourself stand out, but you also need to adjust your piece with pitch, volume and pausing time based on others in the band!)

## A little bit about software project management

Learning to **interpret** and **deal with the errors** is an important part of learning to become a programmer. So is **keeping track** of the code you're working on, **any bugs** you've found and **any new features** you want to develop.

**Planning what to work on.**

- Every **software program** you see has been **built line by line and function by function**.
- Whether you're working on a new feature or a bug, it helps to **have a plan** so that each person on your team **knows what they're working on**.
- Breaking a **big project into smaller bits** will help make sure you don't get overwhelmed.
- Thinking about the **logic flow and writing some 'pseudocode'** is always good before you start writing actual code.

**Testing as you go along.**

- You **won't know if your code actually does** what you're expecting it to **unless you test it!**
- Broken stuff can teach you a lot. It's better to **test smaller bits** of your code more often, than a big chunk all at once.
- **Think about the input and output** for each function you write and **test with various input** to check if you have correct output.

- Have a read about [unit tests](#) which we will cover more in Python module 3, but you can consider how to use them now as you are just starting to learn how to write a program.

### Keeping a list of broken stuff that needs fixing.

- Every day, software developers find broken things in their code and things that could be improved. This is **totally normal** and where a **lot of learning happens!**
- Some bugs might be **showstoppers** you really need to fix and others might be **minor inconveniences** you can work around. Either way, it helps to **have a list so you know which things you want to fix first!**
  - List them then prioritise, don't get so stuck into one small problem that you forget the important ones!
- Remember to use [Stack Overflow](#), think about **what is the best way to describe your questions** and you may find an exact answer for the broken stuff you do not know how to solve.
- **Logging all problems you have encountered** is a very good practise for asking a correct question later on!

### Github project management

There are tools you can use to help manage your projects. You might be surprised to hear that GitHub isn't just for storing your git repository in the cloud – it has some great project management tools too, like:

- Using “cards” to capture ideas, tasks, bugs. etc.
- Assigning tasks to team members.
- Commenting on cards and issues.
- Tracking progress of tasks.

**We will not cover this in the course, but feel free to explore!**

**In module 4 there will be more project management tools that you will learn to use, but do feel free to check with your mentor on what project management tool they currently use.**

## How to design a program

We will design a whole program in the next chapter, but before that we need to consider what are the essentials to building a program:

1. When writing a larger program, it is good to **proceed incrementally**.

**To save and test (i.e. run) your code each time when you make a significant change.**

Doing so makes it easier to identify and resolve errors at each stage. This is much easier than trying to write your code in one go and then discovering that you have a large number of errors to fix. You can use [unit tests](#) during this process.

2. **Plan** ahead and **observe** what you expect.

**When your code is only partly written, you might find it useful to put print statements in place of code blocks that are not yet written.**

This can allow you to run your incomplete code, so as to observe whether execution proceeds as you expect, even though some of the code is not yet in place. For example, if you're writing a **conditional if-else structure**, you might start by **putting a print statement for the else code block**, allowing you to get on with writing and testing the if code block.

3. Study the error message.

**If Python prints an error message when you test your code, study the error message.**

It may help you discover the problem in your code, e.g. pointing out a syntax error.

4. Print statements

**Print statements can be used in various ways to help you understand why your code is not working as you intended:**

- For example, you can print out a value computed as part of some larger task as a way to check that it has been **computed correctly**.

- You can add print statements that signal whether the **IF or ELSE** case of a conditional has been followed **as you expected**. If the wrong case is followed, you may have specified the condition incorrectly or incorrectly calculated one of the values being tested.
- Print statements can also help you find the **source of an error** when you are having **trouble locating it**.
  - e.g. if your code exits in a way you don't expect and you can't see why. In this case, it may be useful to **add print statements at various points, printing strings such as `"*POINT-1*"`, `"*POINT-2*"`, etc.**
  - If there are, say, **four such statements**, but only the **first two messages get printed**, this suggests the **error is located in the part of your code between the second and third print statements**.

## 5. Indentation

**Finally, don't forget the critical importance of correct indentation in Python programming.**

Spyder will help you achieve correct indentation. Pressing **TAB** or **DEL** in the 'indentation zone', will cause the **cursor to move in or out one level** of indentation.

## Remember, it doesn't have to be perfect!

You've learned a lot in a short time – it's ok if not everything works as expected!

If you've had trouble getting your code pushed to git or your code hasn't been cooperating, there's still a lot you can show to the class. Your instructors are here for guidance and support, so please let them know if you have any questions.

Remember, nobody can do everything on Day 1.

**Having a list of things you'd like to add or fix** gives you something to pursue after the class!

Computer scientists often work late hours as computers never need to eat or sleep, which is a very bad habit. It is better if you can build a better and healthier working style from the beginning, **keeping everything logged and well managed**, which will **benefit you in the long run!**



## Chapter 8 - Mobile Data Bundle Purchase Programme

### Learning outcomes

Design a mobile data bundle purchase system which includes the functionalities of checking the passcode, getting a credit balance and purchasing a data bundle.

### Recap from last session

- What is your understanding about coding project management?
- What are the essentials when building a big programme?
- How to use the print function for debugging?

### Let's get started!

The aim of this chapter is to give you experience of using conditional statements in Python and in structuring code through the use of functions. You will write a program which simulates some of the behaviour of a mobile data bundle purchase system.

This chapter requires you to write a reasonable amount of code, that you must produce yourself from scratch. *For this project, tips will be in this colour.*

When you get stuck, recap the tips from the previous chapter.

### The mobile data bundle purchase programming task

The envisaged scenario is how to design a phone company's mobile data bundle purchase system. *We imagine that the system reads the user's input and uses it to access key information from the phone company's central computer, namely: (i) the*

phone user's **true passcode** and (ii) their **current phone credit**. The system **then calls the code** that you will write, which **checks** that the **user** knows the correct **passcode**, and **if so, then provides** mobile data bundle purchase services to the user.

### First file, SimpleBundlePurchase.py

```
def DataBundlePurchase(truePasscode, balance):
```

```
    Print ('not yet created')
```

The first file you need to create (SimpleBundlePurchase.py) contains a 'dummy' (i.e. empty) definition of the DataBundlePurchase function, which consists of a single print statement (which prints a message that the function has not yet been defined).

**It is your task to complete this function definition. Patiently read through all the requirements and make a plan before start coding! Recap how to design a program and project management in last chapter :-)**

### Second file, test\_DataBundlePurchase.py

In a real world scenario, we would **ask the customer to enter their password** before seeing their balance. At the same time we retrieve the real password from a database to check if they match or not. If the password is correct, we then retrieve the customer's information (e.g. balance) from the database, and start to use this information as a parameter in Python coding. We will cover the retrieval of information from a database in later chapters, so currently we can just treat the program as if it had already retrieved the **real password (e.g. '2345')** and **balance (e.g.22.00)**.

```
from SimpleBundlePurchase import DataBundlePurchase
```

```
# Test calls to program:
```

```
print 'TEST-EXAMPLE 1'
```

```
result = DataBundlePurchase('1234',34.55) #database input, you will still need to check user pin
```

```
print '-----\nRESULT:', result

print '-' * 50, '\n'

print 'TEST-EXAMPLE 2'

result = DataBundlePurchase('2345',22.00)

print '-----\nRESULT:', result

print '-' * 50, '\n'

print 'TEST-EXAMPLE 3'

result = DataBundlePurchase('3456',17.55)

print '-----\nRESULT:', result

print '-' * 50, '\n'
```

The second file you need to create (test\_DataBundlePurchase.py) contains some test cases.

**Think about if you run the test code, what will be printed out in the console?**

**See if the printed out info as you expected?**

If you run this file (by pressing F5), it first **imports the DataBundlePurchase function** from the first file and then **calls it with different input parameters**, i.e. **specifying different passcodes and different phone credit balances**.

**Try to understand the info which then gets printed to the console, which is also shown below.**

TEST-EXAMPLE 1

<CASHPOINT FUNCTION: not yet defined>

-----

RESULT: None

-----

The first call to the function in that file is given as:

```
result = DataBundlePurchase('1234',34.55)
```

### Why None?

Note how the result 'returned' by the function call is here assigned to the variable `result`, so that it can subsequently be printed out (reused in the next line of the test file). This doesn't make much sense at this stage, as the function is not yet written to return a result (that is why return None!).

**A function that does not specifically return a result instead returns None** (which is a special null value in Python) and it is this value that is printed when the test file is run.

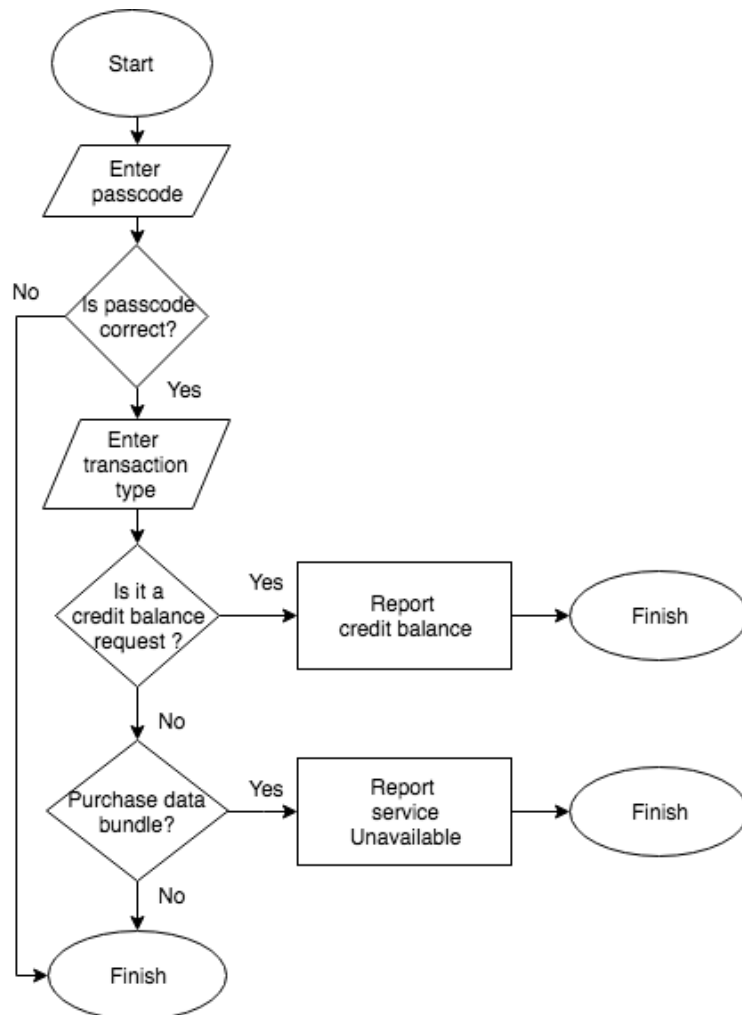
### Recap on project design

Carefully read through the next three sessions. Before you start coding, make a plan:

- 1 function should only do 1 thing. Make sure you design the smallest functionality for each function as possible.
- How many functions are you going to write to get the whole program to work?
- What are each functions' input and output?
- How do the functions interact with each other? (e.g. one function's output can be another function's variable/input).

## Think through the problem logic

### Writing steps based on the logic flow



**Look again at the flowchart. Using this information, you can write down the logical steps that your code must follow, when the `DataBundlePurchase` function that you are defining is called.**

### Program structure

You may notice we put the data bundle function on hold for now and design the whole program structure first:

1. Ask the user to input their passcode.

2. Check whether the passcode value entered matches the true passcode (i.e. the value given in the function call, such as '1234' above). If they match, then continue as below, otherwise print a suitable message and finish.
3. Ask the user to choose their transaction by printing a numbered list of options and reading the value entered by the user.
4. If the input is 1 (for credit balance request), print the credit balance information and exit.
5. If the input is 2 (for purchase data bundle), then (for now) just print a message that says the service is unavailable and exit for now.
6. If the input is something else, then print a suitable message and exit.

## A first attempt in Python

### Make a copy for your dev file

Next, have a go at writing a first definition of the `DataBundlePurchase` function, completing the dummy definition given in the file `SimpleBundlePurchase.py`. First, save a copy of the file with a modified name (say `SimpleBundlePurchase_v1.py`) and develop your code in this new file. You will need to modify the testing file also, so that it imports from the new file. Some hints:

### String or digit inputs

Where the user is required to provide input (e.g. their passcode, transaction choice or bundle choice), you can use the built-in `input()` and `int()` functions (as in the earlier chapters).

**NOTE** that the test file call to the `DataBundlePurchase` function specifies the correct passcode as a `STRING` (of digits), rather than as a number (i.e. an integer), suggesting that `input()` should be used for entering passcodes. This choice was made to avoid problems that could otherwise arise for passcodes beginning with '0'. (e.g. for number/int input, computer will treat 007 as 7 ).

## If-elif-else statement

Where there is a choice of how to proceed, e.g. what to do depending on whether the passcode values match, you can use an if-else conditional statement. Where there are two options (like when choosing the transaction type), but need to think if there are other options or error input that you may need to handle, so you might use an if-elif-else statement.

Test your code thoroughly, to ensure that it behaves as you expect.

**Explain to the people sitting next to you about your design (or code) line by line to see if they have any suggestions or any differences in their design.**

## A more realistic definition: using return

**Take a short break, think about what is missing in your code.**

Before you proceed, you should again save your code with a different name and work in the new file (e.g. SimpleBundlePurchase\_v2.py), however, submit the latest version at the end.

## Test the code to see if it is as you expected

If you have written it well, your first definition of the DataBundlePurchase function may display a reasonable version of the visible behaviour we might expect, i.e. of the pattern of interaction of a user with the mobile data bundle purchase system.

## System needs to log/know what happened

However, it would not be much use in a real mobile data bundle purchase system, as for this purpose, the function would need to return information to the mobile data bundle purchase system that called it, so that the mobile data bundle purchase system would know how to proceed. For example, if the user requested to purchase

a data bundle, the mobile data bundle purchase system would need to know how much phone credit to deduct before completing the purchase. If the passcode was entered incorrectly, then the mobile data bundle purchase system should know to prevent access to the user's account, etc.

## Practise adding return in function

To add this functionality to our code, we can use **return statements**. As we saw in previous chapters, a return statement in a function that has the form “return <value>”. When executed, it causes the function's execution to terminate at this point, with the specified value being returned.

**Task 1, Before making significant changes to a large program, it is always good to exercise with smaller tasks. Let's practise return with conditionals here:**

For example, the following function tests if a number is positive (greater than or equal to 0) and returns (or gives back) the value True if it is, or otherwise gives back the value False if it is not.

**Think about how you are going to code it before reading below :-)**

```
def is_positive(n):  
    if n >= 0:  
        return True  
    Else:  
        return False
```

The value returned might then be assigned to a variable, as in the following:

```
>>> is_positive(3)  
  
True  
  
>>> is_positive(-2)  
  
False
```



```
>>> result = is_positive(5)
```

```
>>> result
```

```
True
```

## Return string logs in the program

Extend your code by adding return statements so that it returns, to the system that has called it, a value that provides the system with the information it needs to proceed. For some cases, this returned value can be a single string, such as "passcode-error" or "credit-balance-request".

## Return two values in data bundle function

For the case of a data bundle purchase, however, the information must specify both that a data bundle purchase was requested, but also the amount to be deducted from the phone credit balance. This can be handled by returning the two pieces of information as a pair, i.e. with a return statement such as:

```
return ("bundle-purchase",amount)
```

## After adding returns, do test again

Recall how the code in the test file (`test_DataBundlePurchase.py`) tries to collect the value returned by the function call, so that it can be printed. Hence, you should now see the results that are returned by your code being printed when the test file is run.

# Breaking the task down, using functions as subroutines

## Decompose functionalities

Again, before you go any further, save your code with a different name (`SimpleBundlePurchase_v3.py`) and work in the new file.

The code you have written so far is hopefully fairly readable, but if the functionality is extended much further, it could easily become long-winded and hard to read. For example, a real mobile data bundle purchase system **might allow three attempts at entering the passcode before refusing to continue.**

It might allow you to **conduct several transactions in one visit to the mobile data bundle purchase system.** If behaviours were achieved by further adding and embedding conditionals, then our function definition could soon be very long indeed.

## Introduce subroutines and its advantages

Many programming languages address this problem by allowing users to specify **named chunks of code**, known as **subroutines**. A subroutine has the **advantage** of being **reusable in different parts of the program** (whilst being specified only once itself) and also — by fulfilling a conceptually coherent subtask — **making the higher level code that calls it much easier to read**. In Python, this idea of subroutines is realised by defining functions.

Develop your program by **defining sub-functions to package up some of the required functionality**. The aim is to give the overall program more complex behaviour, **without making the top-level DataBundlePurchase function itself much more complicated.**

**Think about in which part of the program design you can apply subroutine functions? Have you written any very big functions? Can you write several smaller functions and call them back in the top-level function instead?**

## Example of using subroutine: passcode testing

Define a function **check\_passcode** to cover the passcode checking part of the task. The function will ask the user to input their passcode and compare this to the true passcode, printing an error message if it is wrong, etc.

## Sub-function trick: combining boolean with conditionals

The function might **return a boolean value**, i.e. returning True if the check succeeds and False otherwise. In that case, a **call to this function can appear as the condition of the relevant if-else statement in the DataBundlePurchase function.** as in the following (**where the "..." represent sections of code that you have not filled out**):

```
def DataBundlePurchase(truePasscode,balance): # top-level function

    if check_passcode(truePasscode):

        ...

        ... # Case where passcode check succeeds

    else:

        ... # Case where passcode check fails

def check_passcode(truePasscode): #subroutine function

    ...

    ... # Code asking user to input their passcode

    ... # Returns True or False, depending on success of check
```

Observe how this approach simplifies the definition of the top-level DataBundlePurchase function by **delegating some of the work to the check\_passcode function, which performs a conceptually coherent subtask.**

## Three attempts input with a sub-subroutine

Next, **extend the check\_passcode function to allow the user three attempts at entering their passcode before final rejection.** This can be done by **modifying only the definition of the check\_passcode function, i.e. without complicating the top-level DataBundlePurchase function.**

## Mobile data bundle purchase function

Define a function to provide reasonable mobile data bundle purchase functionality.

This should require:

- The user to confirm their choice of phone number twice in succession and checks that the same value was entered both times.
- The credit amount required for the purchase should be restricted to not exceed the current credit balance.
- The credit amount to top up should be a multiple of 5, (via **another subroutine**).
- You also can setup a maximum top up amount as a restriction at the beginning of the function.

## Homework

1. After completing the program, think about the flow of the whole program and chat with your team members about the importance of testing throughout designing and building the program.
2. Peer review: while developing your code, in some off coding time explain your code to your partner (line by line if needed).
3. Submit your code to git and pull the others' code to have a look if you missed any part or if you can do better in any part.

## Part 3 - Compound Data Types

### Chapter 9 - Lists and Tuples

#### Learning outcomes

- Use index in a list.
- Make changes to a list.
- Understand the difference between list and tuples
- Sort a list.

#### Recap from last session

- Think and reflect on the whole experience of designing a program.
- Why is writing tests so important even before starting to write the program?
- Why is having a clear logic flow so important before starting to write the program?

#### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how lists works! Once you've done that, explain how to change the list by using different list related methods!**

#### Intro to lists

##### Compound data types

Compound data types means data types made out of other data types. You will see what I mean in the following two chapters.

## Mutable and immutable arrays

Mutable means able to change, and immutable means fixed, unchanging over time or unable to be changed.

List is a mutable type of arrays, which means you can make changes to the items in the list. We will also learn the immutable type of array: tuple, in Python in later sessions.

## First trial with a list

List items can be accessed in terms of their position in the list and their positions are identified by their index. You can access an item in a list by using the list's name and the item's index number, try this on your Python console:

```
>>>my_favourite_fruits = ['apple', 'orange', 'banana']  
  
print (my_favourite_fruits[0])
```

## Index in list

Lists are inherently ordered. In the world of programming, the first item in a list is always at position 0 (its “index” position). This means that [0] is the first item in the list, [1] is the second item, and so on. List items can be accessed by their position in the list, with index values from 0 to (n – 1) for a list of length n.

### Task 1, try the code below in your Python console:

```
>>> x = ['this', 55, 'that', my_favourite_fruits] #list can contain mix type of data!  
>>> x[0]  
'this'  
>>> x[1]  
55  
>>> x[2]  
'That'
```

**What about the result for x[3]? Find out by yourself! Also, try x[3][0]?**

**How about x[4]**

```
>>> x[4]
```

Traceback (most recent call last):

```
File "<pyshell#106>", line 1, in <module>
    x[4]
IndexError: list index out of range
>>>
```

The last one (`x[4]`) in the above example attempts to access a non-existent position, so gives an error as the list `x=['this', 55, 'that, my_favourite_fruits]` only contains 4 items, making the index range 0-3.

## Make changes to a list

### Delete, overwrite and add items to a list

As mentioned above, a list can be changed: individual values in the list can be **overwritten**, the list can be **extended** (by **appending**) and items in the list can be deleted.

#### Task 2, try the code below in the Python console:

```
>>>x.remove(my_favourite_fruits) #delete a item in list
>>>x
['this', 55, 'that']
```

```
>>> x[1] = 'and' #update list item value by using index of the item in the current list
>>> x
['this', 'and', 'that']
```

```
>>> x.append('again')
>>> x
['this', 'and', 'that', 'again']
```

**Can you tell how to overwrite a list?**

**Can you tell which function you should use to extend a list?**

**There are also del and pop functions used to delete a item in a list, find out how to use them by yourself online.**

## List operations

We also can use + to compute the concatenation of two lists:

```
>>> x = ['the', 'cat', 'sat']
>>> y = ['on', 'the', 'mat']
>>> z = x + y
>>> z
['the', 'cat', 'sat', 'on', 'the', 'mat']
```

**Tasks: How about trying other operations e.g. -, \* and seeing what happens!**

**Try set(x+y) and guess what it does.**

There are many operations with list that we will not cover at the moment, but do explore by yourself, try them out and chat with the people sitting next to you about your findings.

## Slice of a list

You can also take a slice of a list, using two indices: [i:j]. The slice starts with the item at index i, plus items up to (**but not including**) the item at index j.

**Before slicing a list, can you tell what the index values are for each item in the above list?**

```
>>> x = ['this', 'and', 'that', 'once', 'again']
```

Now write the code below:

```
>>> x[1:4] #include item in position 1-3, not 4
['and', 'that', 'once']
```

**Task 3: Experiment with other index numbers and see what happens.**

### Slicing is more 'permissive'

Slicing is more 'permissive' than access-by-index. As we saw, accessing a non-existent position by index gives an error, whereas a **slice that ranges beyond actual positions, just gives what is available.**

```
>>> z
['the', 'cat', 'sat', 'on', 'the', 'mat']
>>> z[3:10]
```



```
['on', 'the', 'mat']  
>>> z[8:10]  
[]
```

## Sorting a list

You will often want to sort values into some order. e.g. numbers into ascending / descending order or strings (such as words) into alphabetical order.

Python provides two functions for the sorting of lists: with the `sorted()` general function which returns a sorted copy of a list; and `.sort()` list-class specific method, can make change of the original list. Hence it sorts the list “in place”.

### Task 4: try the code below in your Python console:

```
>>> x = [7,11,3,9,2]  
>>> y = sorted(x)  
  
>>>y  
[2, 3, 7, 9, 11]  
>>> x  
[7, 11, 3, 9, 2]  
  
>>> x.sort()  
  
>>> x  
[2, 3, 7, 9, 11]  
>>>
```

**Chat with the people sitting next to you and tell each other the difference between the `sorted()` and `.sort()` functions that sort a list.**

### Reverse sort

By default, sorting puts numbers into ascending order and strings into the standard alphabetical order (uppercase before lower-case).

However we can change this default behaviour, using keyword arguments. e.g. we can **reverse** the standard sort order as follows:

```
>>> x = [7,11,3,9,2]
>>> sorted(x)
[2, 3, 7, 9, 11]
>>> sorted(x,reverse=True)
[11, 9, 7, 3, 2]
```

## Introduction of tuples

Apart from lists, Python has an alternative sequence type: the tuple.

### Tuples are immutable

Tuples are written with **round brackets**, e.g. ('this', 55, '33', 00) and, like a Python list, is ordered and allows access by index **but cannot be changed**, i.e.:

**Task 5, try all the examples below to see if you get the same result or not.**

- Cannot delete an item inside a tuple:

List example 1:

```
>>>a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>del a[-1]
>>>a
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Tuple example 1:

```
>>>b = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>del b[-1]
```

Traceback (most recent call last):

```
File "<ipython-input-21-e85c0322a53b>", line 1, in <module>
    del b[-1]
```

TypeError: 'tuple' object doesn't support item deletion

- Cannot assign a new value to a position in an existing tuple.

List example 2:

```
>>>a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>a[0] = 50
>>>a
Out[25]: [50, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Tuple example 2:

```
>>>b = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
>>>b[0] = 50
```

Traceback (most recent call last):

File "<ipython-input-27-b8228a43e1dc>", line 1, in <module>

```
b[0] = 50
```

TypeError: 'tuple' object does not support item assignment

- Cannot append to an existing tuple.

List example 3:

```
>>> a.append('z')
```

```
>>>a
```

```
Out[29]: [50, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'z']
```

Tuple example 3:

```
>>>b.append('z')
```

Traceback (most recent call last):

File "<ipython-input-22-02ad6e6dcf82>", line 1, in <module>

```
b.append('z')
```

AttributeError: 'tuple' object has no attribute 'append'

**Task, think about whether any other data types that we have covered are mutable or immutable? E.g str, int etc.**

**At some point you are going to design your own data type (object) so you may need to think about whether they are mutable or immutable in your design!**

### Advantages of tuples

So why bother with tuples? They are **more memory efficient**, which can be a key factor in some projects, but that's not a big concern for what we are doing currently.

## Lambda function, used for a more complicated sort

**Lambda is a notation for writing functions**

Is `'x2 + 1'` a single value, or a function? The expression `λx.x2 + 1` unambiguously denotes that it is a function.

In Python:

- `lambda x:(x * x) + 1` : means give me one input (`x`) and I'll give you back the result `x*x + 1`.
- `lambda s:s[1]` : means that given the item `s`, it shall compute/return `s[1]`, which only makes sense if either: `s` is a sequence, so `s[1]` is its 2nd element, or if `s` is a dictionary, so `s[1]` looks up value for key 1.

An example of when lambda could be useful is when **sorting a list of three-items (tuples) by their second value**. By default, sorting using the first value is as shown here:

```
>>> x2 = [('a', 3, z), ('c', 1, y), ('b', 5, x)]
>>> sorted(x2)
[('a', 3, ['the', 'cat', 'sat', 'on', 'the', 'mat']),
 ('b', 5, [7, 11, 3, 9, 2]),
 ('c', 1, [2, 3, 7, 9, 11])]
```

So to **sort by the second value** instead, we use the `key` keyword argument and a lambda expression. The **lambda function looks up the second item of the tuple** and then sorting is done based on these alternative values.

```
>>> x2 = [('a', 3, z), ('c', 1, y), ('b', 5, x)]
>>> sorted(x2, key=lambda s:s[1])
[('c', 1, [2, 3, 7, 9, 11]),
 ('a', 3, ['the', 'cat', 'sat', 'on', 'the', 'mat']),
 ('b', 5, [7, 11, 3, 9, 2])]
```

This should return a negative/0/positive value depending on whether the first argument is considered smaller than/same as/bigger than the second.

### Task 6: If you try `s[2]` what will happen?

How about changing:

```
z[0] =22, z[1] =5
```

### then updating x2:

```
x2 = [('a', 3, z), ('c', 1, y), ('b', 5, x)]
```

### Now try the below code to see if the result is as you expected?

```
sorted(x2, key=lambda s:s[2])
```

### How about?

```
sorted(x2, key=lambda s:s[2][1])
```

**Task 7: Create your own version of a list of tuples and sort them by ascending and descending orders, then sort them by second values using lambda function -- try starting with simple examples.**

## Homework

1. Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!
2. Complete exercises 34 and 38 on [Learn Python The Hard Way](#).
3. Try the string challenges from code bat Python [List-1](#), you may come across some concepts that we haven't covered yet e.g. loops. You can skip them for now and come back to them later.

## Further learning

During a career in programming, you may find the List object used a lot in coding interview questions related to data structures. This [video](#) will let you know more about Lists.

Also, it is a good idea to check the Python library about this chapter: [datastructures](#).

## Chapter 10 - Dictionaries

### Learning outcomes

- Understand the dictionary data type in Python.
- Understand keys and values in dictionaries.
- How to make changes to a dictionary.
- How to use dictionaries in Python.

### Recap from last session

- What is a list in Python? Why is it different to the traditional concept of arrays?
- What is the index of a list?
- How to make changes to a list?
- How to take a slice of a list?
- How to sort a list in reverse order?
- What is a tuple? How to use lambda functions to sort by the 2nd values in a list of tuples?

### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how dictionaries work! Once you've done that, explain how to make use of keys and values and how to sort a dictionary!**

### Intro to dictionaries

We have seen several compound types: i.e. lists, tuples and strings, which group together multiple elements. These are all sequence types, i.e. are inherently ordered.

Another form of compound type is the Python dictionary, which is inherently NOT an ordered type, instead it is a mapping type.

## Dictionary is a mapping type data

In many other programming languages you may come across hash tables, which are a very powerful and efficient data type. Dictionaries in Python are implemented using hash tables or hash mapping. It is an array whose indexes are obtained using a hash function on the keys.

## Keys and values

The essential purpose of a dictionary is very simple: to associate keys and values. **Keys** can be strings (e.g. "bill"), or numbers (e.g. 55), or even tuples (e.g. the tuple ('bill',"bryson")), but not ordinary lists.

The stored **values** may be any Python value: strings and numbers are common cases. **A key can be paired with only one value at a time**, i.e. if **we assign a new value to a key, its previous value is lost**, this is similar to when using an index to overwrite an item in a list in the previous chapter. You can treat the key as an 'index' of its value.

## Creating and using operations in dictionaries

Start by trying out some simple operations with the Python interpreter. You can create an empty dictionary by assigning the value '{ }'. This is illustrated in the example below, which also shows cases of assigning a value to a new key and of assigning a new value to an existing key.

**Task 1 Study the example thoroughly and be sure you understand each step.**

**Creating and assigning values to a dictionary.**

```
>>> salary = {}  
>>> salary['al'] = 20000  
>>> salary['bo'] = 50000
```

```
>>> salary[7] = ('Joke', 'Chen', 'Bond') # key and value can be any data type with any numbers of them. Beware, numbers can not start with 00X e.g. 007
```

```
>>> salary
```

```
{'bo': 50000, 'al': 20000, 7: ('Joke', 'Chen', 'Bond')}
```

### Getting dictionary values through a key.

```
>>> salary['bo']
```

```
50000
```

### Overwriting dictionary values or doing operations with those values.

```
>>> salary['bo'] = 55000
```

```
>>> salary['al'] += 2000
```

```
>>> salary
```

```
{'bo': 55000, 'al': 22000}
```

### So is the dictionary data type mutable or immutable?

**Task 2 : try to create a dictionary of your own, to store phone numbers for a few people you know.**

### The 'format' for dictionaries.

Here is how to prepopulate a dictionary with some name:number pairs. Using a comma to separate each pair:

```
>>> tel = { 'alf':111, 'bobby':222, 'calvin':333 }
```

```
>>> tel
```

```
{'alf': 111, 'calvin': 333, 'bobby': 222}
```

**Task 3 try to look up / update values of the tel dictionary above.**



## Recap assess and update a value

```
>>> tel['bobby']      # access a value
222
>>> tel['bobby'] = 555 # update a value
>>> tel

{'alf': 111, 'bobby': 555, 'calvin': 333}
```

We have learned how to access a value and how to update a value, have you tried other operations?

## Delete an item in a dictionary

**How to delete a key value pair in a dictionary? Try to search for the answers by yourself and chat with the people sitting next to you about your findings.**

```
>>> del tel['bobby'] # delete entry with given key -- same with delete item in a list. Del function is
not object oriented

>>> tel

{'alf': 111, 'calvin': 333}
```

## Get keys and values from a dictionary.

**Task 4, write the code below and see what happens:**

```
>>> tel.keys() # get list of keys (non-standard)

dict_keys(['alf', 'calvin'])

>>> tel.values()

dict_values([111, 333])
```

## What is the data type for keys and values?

**Task 5: retrieve the first key from a dictionary, think about how you going to do it?**

```
tel.keys()[0]
```

TypeError: 'dict\_keys' object does not support indexing

**Mmm, think about what you need to check?**

```
>>>type(tel.values())
```

dict\_values

**How to cast between data types?**

```
>>>list(tel.keys())[0]
```

'Alf'

**How to retrieve the first value in the dictionary then?**

## Avoiding key errors

**What are key errors?**

Now try looking up a key that is not in the dictionary. This gives an error, as shown below. Such errors (which will crash your code) is a major issue in using dictionaries.

```
>>> tel['dave']
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

tel['dave']

KeyError: 'dave'

**Think about how you can avoid such errors?**

**How to avoid key errors**

We can check if a key is present in a dictionary with a **test in the form of “KEY in DICT”**, which returns True or False, as shown below. Thus, we can **avoid key look-up errors** by checking the look-up step within a conditional that has such a test.

```
k = 'eric'
if k in tel:

    print(k, ':', tel[k])

Else:

    print(k, 'not found!')
```

**Task 6, Keep a note on this, and do a revision on the conditionals and Boolean type which are great tools for doing tests.**

## Sorting a dictionary by value

An example of using dictionaries in applications is that you may use dictionaries to store numeric values associated with keys.

Think about the scenario: A company have a list of metals (keys) with the density (v1) of different metals, the share prices (v2) of companies or how often each possible outcome occurred in a series of experiments (v3).

You may want to handle a dictionary in a manner ordered with respect to the values: e.g. print metals in descending order of density or sort companies by share price (v2), so that you can identify the “top ten” companies.

**Recap about how to sort items in a list in ascending / descending order and how to use lambda functions to sort items.**

### Experiment on small example before real task

We can get keys and values as a list, but how can we sort them within a dictionary?

```
>>> counts = {'a': 3, 'c': 1, 'b': 5}
>>> labels = list(counts.keys())
>>> labels
['a', 'c', 'b']
```

## Using lambda function

You can use the lambda function to return keys' values to the dictionary, e.g.

```
>>> labels.sort(key=lambda v:counts[v]) --check example
>>> labels    # puts labels into ascending order of count
['c', 'a', 'b']
```

Remember key 'c' has the smallest value '1' in the dictionary and key 'b' has the largest value '5' in the dictionary.

Can you do the sort by using function sorted()?

```
>>> sorted(matel_labels, key=lambda k:counts[k])
```

## Return both key and values in sort

```
>>>sorted(counts.items(), key=lambda kv: kv[1])
[('c', 1), ('a', 3), ('b', 5)]
```

In this case, the program treats the key and value as a tuple data type, key in position 0 and value in position 1. Therefore it can be sort by value, kv[1].

## Sorting dictionary values in descending order

Here is an example of using lambda, this time for printing metals in descending order of density:

```
>>> densities = {'iron':7.8, 'gold':19.3, 'zinc':7.13, 'lead':11.4}
>>> metals = list(densities.keys())
>>> metals
['zinc', 'gold', 'iron', 'lead']
>>> metals.sort(reverse=True,key=lambda m:densities[m]) # key: dict[key]
>>> metals
['gold', 'lead', 'iron', 'zinc']
```

Also the version for returning both keys and their values:

```
sorted(densities.items(), key=lambda kv: kv[1],reverse=True)
[('gold', 19.3), ('lead', 11.4), ('iron', 7.8), ('zinc', 7.13)]
```

### Task 7, Summarise the sorting code in task.

### Task 8, Back to the original problem, think about what if the dictionary

contains three values: density (v1) of different metals, the share prices (v2) of companies, and how often each possible outcome occurred in a series of experiments (v3). How you going to sort the dictionary's 2nd values in descending order?

Please sort using both methods above: returning only the keys, and returning keys with their values.

## Homework

Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!

Try to complete the challenges in [codingbat](#), see if you can get all stars!

## Part4 - Iterations

### Chapter 11 - 'While Loop' and Guessing Number Game.

#### Learning outcomes

- Learn about loops as a control structure.
- Learn about the 'while' loop and use it in an application.
- Learn about using 'break' and 'continue' to exit a 'while' loop.

#### Recap from last session

- What are the keys and values in a dictionary?
- How to make changes to a dictionary?
- What are key errors?
- How to sort a dictionary by its values?

#### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how 'while' loops work! Once you've done that, explain how to make use of a while loop in a guessing number game!**

#### Loops as control structures

Part 4 gives you a chance to use loop constructs in Python and will illustrate how programs can be used to solve some **maths and engineering problems**.

First, recall the major control structure types: sequence, selection & repetition. In this chapter we will learn about the ones for repetition, which are used for: **executing a**

**statement or block of statements more than once.** This means using programming language to allow repetition by the provision of looping constructs.

## Two main loops

- **Conditional loops** (“while”), where a loop repeats until **a certain condition is met.** e.g. Successive approximations to solve an equation. We will learn about the ‘while’ loop in this chapter.
- **Counting loops** (“for”), where a loop **repeats a preset number of times.** e.g. Changing the brightness of each pixel in an image. We will learn about the ‘for’ loop in the next chapter.

## The while loop

A key construct in Python (and many languages) is while. The while loop has an associated condition. Here, the loop continues to repeat so long as the condition is satisfied (i.e. returns True):

while CONDITION:

CODE-BLOCK

This can be used in the informal pseudo-code supermarket shopping example.

1. Get a trolley
2. While there are items on shopping list
  - 2.1 Read first item on shopping list
  - 2.2 Get that item from shelf
  - 2.3 Put item in trolley
  - 2.4 Cross item off shopping list
3. Pay at checkout

## While condition structure

Here, “there are items on shopping list” **equates to a test, that evaluates to True or False**. In Python, the while construct has the form:

```
while CONDITION:
```

CODE-BLOCK

**Task 1: to print a series of values produced when doing repeated division (by 2) of some initial value.**

**Make a plan, write pseudo-code with the while structure above. Try to do the code by yourself before seeing the result.**

**(Think about the difference between your design and the code below. Try out the code below and try to understand and type the code based on your understanding rather than just copy-and-pasting).**

```
x = 33

while x >= 1:

    print(x, ': ', end='')

    x = x / 2

print(x)

>>>

33 : 16.5 : 8.25 : 4.125 : 2.0625 : 1.03125 : 0.515625
```

**Task 2: A function for computing triangular numbers, which for a input integer n, returns the sum of values from n down to 1, i.e.  $n + (n-1) + \dots + 2 + 1$ .**

**Make a plan, write pseudo-code as comments first.**

**Think about how to program this little task first then look at the code below, ask the instructor if you have any questions.**



```
def triangular(n):  
    trinum = 0  
  
    while n > 0:  
        trinum = trinum + n  
        n = n - 1  
  
    return trinum
```

```
>>> triangular(1)
```

```
1
```

```
>>> triangular(3)
```

```
6
```

```
>>> triangular(5)
```

```
15
```

Task 3: Loop to determine if student marks are a pass/fail/first:

**Make a plan, e.g. mark  $\geq 70$  is first class, mark  $\geq 40$  is pass and mark  $< 40$  is fail. Write some pseudo-code.**

**Note in this program you can use user inputs and use conditional logic to interact!**

```
mark = 1  
  
while mark > 0:  
    mark = input('Enter mark: ')  
  
    mark = int(mark)  
  
    print("Mark is", mark, end="")  
  
    if mark  $\geq 70$ :  
        print(" - first class!")
```

```
elif mark >= 40:

    print(" - that's a pass")

else:

    print(" - oh dear, that's a fail")
```

Enter mark: 77

Mark is 77 - first class!

Enter mark: 44

Mark is 44 - that's a pass

Enter mark: 0

Mark is 0

Oh dear - that's a fail

>>>

**Think how to change the above program to make it 80+ as first class, 60+ as pass?**

**Can you change the whole program and make use of the logic for other applications? Have a chat with your partner and be creative!**

## Early exit and continuation

**Do you have any questions about while loops?**

What happens if the condition is false before the loop begins? Is the code-block executed once or not at all? What happens if the condition is never going to become false, e.g. if the test was  $1 < 100$ ? (NOTE: if needs be, you can interrupt the run by clicking the “terminate” icon (a small red square) at the top right-hand side of the console window.)

```
i = 55
```

```
while i > 10:  
  
    print i  
  
    i = i * 0.8
```

Python provides special commands: `break` and `continue` to modify the normal flow of a loop.

### **‘break’ statement**

A ‘break’ statement in a loop **immediately terminates the current iteration and ends the loop overall.**

This can be very useful. E.g. task 4: in an application, for printing a greeting for name entered until the user enters ‘done’. (Tip: Use of while True loop will run indefinitely).

**Think about how you design this code before starting.**

```
while True:  
  
    name = input('Enter name: ')  
  
    if name == 'done':  
  
        break  
  
    print('Hello', name)
```

Enter name: Bill

Hello Bill

Enter name: done

**Think how to use break in the previous enter marks program or for your own program?**

## 'continue' statement

A 'continue' statement in a loop immediately **terminates the current iteration and starts the next iteration**. An example of the form follows:

```
while CONDITION-1:

    if CONDITION-2:

        STATEMENTS-1

        continue

    STATEMENTS-2
```

But the same result can often be achieved without `continue`. The preceding code can be restated more clearly like this, without using `continue`:

```
while CONDITION-1:

    if CONDITION-2:

        STATEMENTS-1

    else:

        STATEMENTS-2
```

`break` and `continue` should be used with care. Their use/overuse is often symptomatic of bad programming style, as the **same result is often better achieved by the use of conditionals instead**, which can better express the intended logic of the task.

## The guessing game

Let's create a game using while loop!

The green colour text gives some clues for you to design the game.

**Task 5: recall how to import a Python module and function?**

Type and save the code below into a file e.g. `guessing_game.py`. This code contains the skeleton of a function definition for a guessing game. The code **imports a function `randint`** (from the `random` module), which is used to generate a random integer, e.g. a call `randint(1,20)` returns a random integer in the range of 1 to 20 (including 1 and 20). The function has parameters for the **number of guesses allowed** and the **range from which the code picks a number to be guessed**.

```
from random import randint

def guess(attempts,range):

    number = randint(1,range)

    print ("Welcome! Can you guess my secret number?")

    # YOUR CODE GOES HERE!

    print ("END-OF-GAME: thanks for playing!")
```

**Try different numbers inside `randint()` first and see what happen.**

Ok, after saving the file, **Task 6 is to complete the definition of the guess function**, so that it behaves as illustrated in the dialogue below, i.e. telling the player how many guesses they have left, reading in their guess, telling them if their guess is right and if not, whether it is too low or too high. Your code will use a while-loop to make sure the player is only given the allowed number of attempts.

NOTE: a tricky issue is handling the case where the player guesses correctly while further attempts remain, as here the loop must **terminate early**. Since this is a function, a return statement could be used to end the entire function call, but then the final “GAME OVER” statement of the definition would not be reached. Instead, you can use the special loop-control command `break`, which causes the loop to end, with execution proceeding to the statements that follow the loop. An example of the game program in the console output can look like the result below, find out if your code gives the same response:

```
>>> guess(3,10)
```

```
Welcome! Can you guess my secret number?
```

You have 3 guesses remaining

Make a guess: 4

No - too low!

You have 2 guesses remaining

Make a guess: 7

Well done! You got it right.

GAME OVER: thanks for playing. Bye.

```
>>> guess(3,20)
```

Welcome! Can you guess my secret number?

You have 3 guesses remaining

Make a guess: 12

No - too low!

You have 2 guesses remaining

Make a guess: 16

No - too high!

You have 1 guesses remaining

Make a guess: 14

No - too low!

No more guesses - bad luck!

GAME OVER: thanks for playing. Bye.

**Challenges, think about --as you test your definition, you might ponder the following questions:**

**(1) Is there an optimal strategy to follow, i.e. that gives the best chance of success?**

**(2) Is there a number of guesses that can guarantee success for a given range size (one that is less than the range size itself)?**

## Homework

Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!

Complete exercises 33 on [Learn Python The Hard Way](#).

## Chapter 12 - The 'For' Loop.

### Learning outcomes

- Write iterative statements with the 'for' loop.
- The difference between 'while' and 'for' loops.
- Use the 'for' loop to loop through different data types.

### Recap from last session

- What are the two main loops in the repetition category of control structures?
- What is a 'while' loop?
- What is the danger of using a 'while' loop?
- How to exit a loop early?

### Let's get started!

**Task: Finish reading this chapter. Then turn to your partner and explain how 'for' loops work! Once you've done that, think about how to use nested "for" loops to do a digital clock?**

### Introduction to 'for' loops

Recall that repetition is a category of control structures: executing a statement or block of statements more than once. In this chapter, we are going to cover the other type of loop, the **counting loop** ('for').



## The counting loop

The 'for' loop construct is widely used to implement counting loops. In most languages, the **for loop has an explicit loop variable**, whose value counts in fixed steps from an initial value to a final value. Once the final value has been reached, the loop stops. e.g. variable `i`, counting 0, 1, ..., 9. A loop variable may be used within the loop, such as in an index, to access successive elements of an array.

## Also called iterative loops

'For' loops are also called iterative loops, because they iterate (run through their code repeatedly) until there's nothing left to iterate through.

This concept makes more sense if we go back to what we learned about **lists** earlier.

Imagine you're going to a coding party with some friends from class and you need to go to the supermarket to get supplies. Keeping with the Python theme of your party, you bring your laptop with you, make a digital "shopping cart" in Python and add stuff to it using `.append()`, which you read about in [Python's documentation](#).

Before you get to the checkout counter, you want to print the contents of your cart, to make sure you have everything. Here's how you'd do that in Python, using a for loop:

```
my_shopping_cart = ["cake", "plates", "plastic forks", "juice", "cups"]

for item in my_shopping_cart:
    print item
```

### Task 1, try to run this for loop with the list in the example above.

The only condition the Python **interpreter cares about in your for loop is whether there's anything left to iterate through**. Since you've only got 5 items in your shopping cart, your loop will only run 5 times. Python's clever that way.

## A few helpful things about for loops

- The **code that's part of your for loop must be indented**, just like when you write a function. Just like a function, make sure you also don't forget the **colon ':'** at the end of the first line!
- for loops don't return anything (as they're not functions).
- This is the format that you need to use when you write a for loop:

```
for x in name_of_thing_to_iterate_through:  
    print (x)
```

The `x` can be called anything – you could even call it “`x`”, “`bananas`”, “`count`” or “`item`” – **it's basically just a placeholder name**. But the name of the thing you're iterating through is important – it's the name of your list, so that Python knows what to iterate through.

## Using for loop to update items in a list

You can do lots of things with for loops, including adding things to or deleting things from lists (remember lists are mutable), which we'll cover soon. But, let's do a short comparison between “for” and “while” loops first.

## The 'for' loop vs the 'while' loop (intuitively)

Remember the 'while' loop used for that informal pseudo-code supermarket shopping example:

1. Get a trolley
2. While there are items on shopping list
  - 2.1 Read first item on shopping list
  - 2.2 Get that item from shelf
  - 2.3 Put item in trolley
  - 2.4 Cross item off shopping list

### 3. Pay at checkout

The corresponding for loop pseudocode is simpler:

1. Get a trolley
2. For (each) item on shopping list
  - 2.1 Get item from shelf
  - 2.2 Put item in trolley
3. Pay at checkout

## The 'for' loop, used for simple iteration

### Recap the list data type

Recall from the list chapter, that Python lists are ordered collections of items, that can be of mixed types, e.g. as in ["some string", 3.55, "this", 2] which contains both strings and numbers. The empty list (i.e. which contains no items) is []. Listed items can be accessed by their position in the list, with index values from 0 to (n – 1) for a list of length n. As the example below shows, a list can be changed, with values at existing positions being overwritten or with a list being made shorter or longer, as in this example (where append is used to 'grow' a list).

```
>>> x = ['aa', 'bb', 'cc']
```

```
>>> x[1] = 33
```

```
>>> x
```

```
['aa', 33, 'cc']
```

```
>>> x.append('dd')
```

```
>>> x
```

```
['aa', 33, 'cc', 'dd']
```

## Looping through a list

Often, ‘for’ loops are used together with a list. Such loops will repeat as many times as there are items in the list, for example **assigning variables to each member of the list in turn as its value**. **Task 2, experiment with the code below:**

```
>>> values = [875, 23, 451]
```

```
>>> for val in values:
```

```
    print('---> '+str(val))
```

```
--> 875
```

```
--> 23
```

```
--> 451
```

```
>>> for val in values:
```

```
    print('---> '+str(val+50))
```

**Task 3, create a list by yourself and use a ‘for’ loop to print each item.** Such as:

```
>>> values = ['this', 55, 'that']
```

```
>>> for item in values:
```

```
    print('***', item)
```

```
    *** this
```

```
    *** 55
```

```
    *** that
```

## Looping through other types

The “for” loop can also similarly iterate over other types, such as tuples and strings.

**Task 4, write the code below in your Spyder console:**

```
>>> for char in "Yes":
```

```
    print(char)
```

```
Y
```

```
e
```

```
s
```

More generally, various types demonstrate iterable behaviour and can appear in a for loop. **Task 5, create a tuple by yourself and use a 'for' loop to print each item.**

### Looping through a dictionary with sorted order

Next explore what happens if we do a simple iteration over a dictionary, e.g. a loop in the form "for VAR in DICT:". Try such a loop with your dictionary and print the values assigned to VAR as the loop runs, so you can see what is assigned,

**Task 6, create a salary dictionary with the info below. Think about how to sort your dictionary and print key:value pairs.**

```
al = 20000
bo = 50000
ced = 1500
```

**Not sure how to do the above task? If stuck, try to understand the below example first, then do the task again!**

### Task 7, Create a dictionary first:

```
>>> densities = {'iron':7.8, 'gold':19.3, 'zinc':7.13, 'lead':11.4}
```

Let's revise the dictionary data type by checking the keys of the dictionary and saving them into a list:

```
>>> metals = list(densities.keys())
>>> metals
['zinc', 'gold', 'iron', 'lead']
```

Do you remember how to sort the dictionary keys by their values, in reverse order?

```
>>> metals.sort(reverse=True,key=lambda m:densities[m])
>>> metals
['gold', 'lead', 'iron', 'zinc']
```

Now, let's print the sorted key:value pairs and revise print formatting (check any info

that appears new to you, then chat with the people next to you to check your understanding):

```
>>> for metal in metals:
    print('{0:>8} = {1:5.1f}'.format(metal,densities[metal]))
gold = 19.3
lead = 11.4
iron = 7.9
zinc = 7.1
```

### Search and operations inside a “for” loop

Apart from printing items inside a data type, “for” loops are also a great tool for many tasks for accessing items in those data types:

**Task 8: Think about how to use ‘if’ and ‘else’ conditional logic and the ‘for’ loop to scan a list to search for/print a particular value. E.g. only print metals that have a density value > 8.**

**Task 9: How about using ‘for’ loops to compute the sum of values in a list of numbers? Try the code below in a file. Can you convert it into a SumValues function with a returned sum value?**

```
values = [3, 12, 9]
total = 0
for val in values:
    total += val
print('TOTAL:' + str(total))
```

Remember `total += val` means the same as `total = total + val`.

Remember to cast the data types into one type when not using format printing,

```
>>> def sumValues(l):
    sumV = 0
    for val in l:
        sumV +=val
    return sumV
```

```
>>> print ( sumValues(values))
```

### Loop list with index

**Do you remember what len() and range() functions do? Try to experiment using them with different lists.**

range(3) – gives values 0, 1, 2 , range(n) – gives values 0, ..., (n-1). E.g.

```
>>> range(3)
```

```
[0, 1, 2]
```

**Task 10, experiment with the len() function with a list by yourself and explain to your partner what it does, then try the code below.**

```
>>> for i in range(len(values)):
```

```
    values[i] = values[i] * 2
```

```
>>> values
```

```
[1750, 46, 902]
```

**Can you write the above code without index (directly doing operations with list items)? Which style is better?**

### Special cases counting loops: step size, range

Remember you can modify the behaviour of range by adding extra arguments, which can be useful for 'special cases' of counting loops. Examples:

range with 1 argument: arg gives end value.

e.g. range(5) gives: 0, 1, 2, 3, 4

range with 2 arguments: args give start and end values.

e.g. range(2,5) gives: 2, 3, 4

range with 3 arguments: args give start, end and step values.

e.g. range(3,10,2) gives: 3, 5, 7, 9 •

With step size 2, the loop item will also jump for each two numbers:

```
>>> for i in range(3,10,2):
```

```
...     print(i)
...
3
5
7
9
```

**Task 11, try to generate different lists by using different parameters in the range function and print out the items in the list using a ‘for’ loop.**

### Loop Control — with ‘for’ loops

Loop control commands break and continue also work within for loops for modifying the normal flow of a loop.

Remember:

- A break statement in a loop immediately terminates the current iteration and ends the loop overall.
- A continue statement in a loop immediately terminates the current iteration and starts the next iteration. For a for loop, this causes the loop to move onto the next item of iteration.

In applications, for example, you might use a for loop to scan for a sought item, then use break to end the scan if it is found. After all, once you’ve found it, there is no need to read the rest of the file.

**Task 12, scan a list of numbers to check if any are greater than 100 and stop if you find one that is:**

```
nums = [2,3,65,23,123,432,3]
for n in nums:
    if n > 100:
        print('found:', n)
        break
```



## Nested loops

One loop can contain another loop, which is referred to as nested loops.

```
outer_vals = [1, 2, 3]
inner_vals = ['A', 'B', 'C']
for oval in outer_vals:
    for ival in inner_vals:
        print(oval, ival)
```

Be careful when you place the print, make sure which level of loop item you would like to print out. **Task 13, experiment with the code above. It is printing items from the inner loop and the inner loop runs to completion for each iteration of the outer loop.**

Therefore the above code produces this output:

```
1 A
1 B
1 C
2 A
2 B
2 C
3 A
3 B
3 C
```

### A multiplication table

Here is code to print a small multiplication table

```
for i in range(1,7):
    for j in range(1,11):
        print('{0:>3}'.format(i * j), end=" ")
    print('\n')
```

The inner loop generates a single row of the table and the outer loop causes multiple rows to be printed.

### Task 14, can you extend the table to a 10\*10 one? What other applications can you think of using nested loops?

#### Challenge! A sorting algorithm.

Nested loops have many uses. For example, if you wanted to sort a list of values into ascending order, there are various sorting algorithms available.

We'll look at the method called Bubble Sort. This method moves along a list, comparing adjacent values and swapping them if they are out of order. This is likened to moving a small window (or 'bubble') along the list, comparing values in that window and swapping them if needed.

Example: sorting list: [4, 3, 6, 5, 2, 1]

#### **Bubble passes over list for the first time:**

As the bubble moves, the highest value seen so far is carried along. At the end of the first pass, the list is not yet in order, but the highest value has been moved to its final position — its correct place.

#### **Bubble passes over list for the second time:**

Second highest value will be carried along to its correct position.

#### **After N passes (where N = length of list):**

All values have been carried to their correct positions — the list is now sorted.

If doing this in Python... first make a single pass of the bubble:

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)
for i in range(N-1):
    if values[i] > values[i+1]:
        tmp = values[i]
        values[i] = values[i+1]
        values[i+1] = tmp
```

Why does i here range up to N-1, rather than N? Because otherwise, accessing values at position i+1 will cause an **index-out-of-bounds error**. The bubble passing

over must be repeated until the list is sorted. Nest previous 'bubble pass' loops within another, to repeat it N times:

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)
for j in range(N):
    for i in range(N-1):
        if values[i] > values[i+1]:
            tmp = values[i]
            values[i] = values[i+1]
            values[i+1] = tmp
```

The preceding version works, but we can improve it by avoiding some unnecessary work. We only need to run the outer loop N-1 times, as once N-1 items are correctly in place, so also must the final one already be in place.

After j runs of the inner loop, j final items are correct, so the bubble can stop its pass earlier — there no need to look at these items as they are already in the right places.

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)
for j in range(N-1):
    for i in range(N-1-j):
        if values[i] > values[i+1]:
            tmp = values[i]
            values[i] = values[i+1]
            values[i+1] = tmp
```

## Homework

Review the examples from today's session to make sure you understand them. If you get stuck, don't be shy about asking an instructor for help!

Try to complete the challenges in [codingbat](#), see if you can get all stars!

## Part 5 - More on OOP

### Chapter 13 - OOP Project

#### Learning outcomes

- Know how to define a class and create an instance.
- Be able to correctly use 'self' in methods.
- Complete the OOP project.

#### Revise object oriented programming

As we have learned the Python fundamentals, let's combine this knowledge and do a cool project. However, let's revise OOP first as we will use it hugely in this project and your future coding structures.

The class is a fundamental building block in Python. It underpins not only many popular programs and libraries, but the Python standard library too.

Let's revise the OOP with a person class example:

##### Person class example

**What kind of attributes would you like to set up in `__init__`? What methods would you want to have in the superclass? - As generic as possible.**

E.g. A Person class might have attributes (variables) for name, age, height, address, tel.no., job, etc and have methods (functions) to update address, update job status, work out if they are an adult or child, work out if they pay full fare on the bus, etc.

There might be many objects of the Person class, each representing a different person with different specific data, but all store similar information and behave similarly.

So Person would be a class and person1, person2 & person3 would be objects in that class.

## The syntax of defining classes in python

The definition opens with the keyword class and then the class name.

Most classes needs an initialisation method, which is called when an instance is created and has the 'special' name: `__init__`. This **establishes the attributes**, i.e. **variables belonging to objects in that class**.

```
class Person:

    def __init__(self):

        self.name = None

        self.age = 'not defined'

        self.isMale = None
```

## The 'self' in class

Note the use of the special variable 'self' here. It is the instance's way of referring to itself, e.g. self.age above means "the age attribute of this instance".

The Person class with this initialisation method can create an object (i.e. instance) of this class as follows:

```
>>> p1 = Person()

>>> p1.age

'not defined'
```

Here, the call to `Person()` creates a new instance of the Person class, so the `__init__` method is called automatically, to initialise the object, which is assigned to p1. Then the statement `p1.age` accesses p1's age attribute directly, i.e. that value is accessed, and printed by the interpreter.

## Add conditionals in initialisation methods

More generally, **initialisation methods can have parameters**, which can be used to set the initial values of attributes. **Task 1, try the extended version below, note the parameters that you need to give as input of the `__init__` method:**

```
class Person:

    def __init__(self,name,age,gender):

        self.name = name

        self.age = age

        if gender == 'm':

            self.isMale = True

        elif gender == 'f':

            self.isMale = False

        else:

            print("Gender not recognised!")
```

**Think about how you are going to create objects e.g. p1, p2 now?**

An example of creating an instance:

```
>>> p1 = Person('John',44,'m')
```

```
>>> p1.name
```

```
'John'
```

```
>>> p1.isMale
```

```
True
```

## Adding functionality

We can define (more) methods under the Person class. Think about what generic method you would like to add?

The reason to set generic methods here is because it's subclasses can benefit from those generic methods.

```
class Person:

    def __init__(self,name,age,gender):

        ...

    def greetingInformal(self):

        print('Hi', self.name)

    def greetingFormal(self):

        if self.isMale:

            print('Welcome, Mr', self.name)

        else:

            print('Welcome, Ms', self.name)
```

As before, 'self' is used to refer to this instance and allows access to the data of the instance.

**Task 2, try the code above to define (more) functions / methods.**

```
>>> p1 = Person('Harry',12,'m')
```

```
>>> p2 = Person('Jean',12,'f')
```

```
>>> p1.greetingInformal()
```

```
Hi Harry
```

```
>>> p1.greetingFormal()
```

```
Welcome, Mr Harry
```

```
>>> p2.greetingFormal()
```

```
Welcome, Ms Jean
```

Here, both method calls use the instance data (name) and show behaviour conditioned to that data (gender).

### Task 3, add another method, a greeting filter. Please explore more by yourself!

```
class Person:

    ...

    def greetingAgeBased(self):

        if self.age < 18:

            print('Welcome, young', self.name)

        elif self.age > 60:

            print('Welcome, venerable', self.name)

        else:

            self.greetingFormal()
```

### How to test the above method? Try it out yourself :-)

Note that **here the 'else' case calls another method of this instance**, through the use of self in the method call, i.e. in `self.greetingFormal()`. Compare this to the method's definition (given earlier) that the definition specifies a single parameter (argument) 'self', but that argument is not provided in the above method call. Instead, it is implicit in the "self. " prefix.

Here are some example calls:

```
>>> p1 = Person('Harry',12,'m')

>>> p2 = Person('Amali,88,'f')

>>> p3 = Person('Richard',50,'m')

>>> p1.greetingAgeBased()
```



Welcome, young Harry

```
>>> p2.greetingAgeBased()
```

Welcome, venerable Minerva

```
>>> p3.greetingAgeBased()
```

Welcome, Mr Sirius

## Defining classes with inheritance

### Task 4, let's create a subclass for the Person class.

```
class Wizard(Person):  
  
    def greetingFormal(self):  
  
        print('Welcome, Mr', self.name, end=' ' )  
  
        print('- you\'re a fine wizard!')
```

It starts essentially as a copy of the Person class, but you can then overwrite methods of the Person class to modify behaviour and add new methods to add new behaviours.

The subclass inherits from the superclass, so the Wizard class inherits from the Person class. The subclass may override some features of the superclass and may add some additional features to the superclass.

Let's compare the difference between superclass and subclass `greetingFormal` method with objects.

For a generic person, we use the person class:

```
>>> p1 = Person('Harry',12,'m')  
  
>>> p1.greetingFormal()  
  
Welcome, Mr Harry
```

However, if you use the subclass:

```
>>> p1 = Wizard('Harry',12,'m')
```

```
>>> p1.greetingFormal()
```

```
Welcome, Mr Harry - you're a fine wizard!
```

## Redefine `__init__` in subclass

Subclasses inherit the `__init__` method of the superclass by default. As we can overwrite some methods in the super class, we also can choose to redefine the `__init__` method, e.g.:

```
class Wizard(Person):

    def __init__(self,name,age):

        self.name = name

        self.age = age

        self.isMale = True

    ...
```

## Smarter way to redefine `__init__` attributes

The new definition overwrites the old definition in the same way as methods get overwritten. But some of the work done by this `__init__` method **is the same as** the work done by the superclass `__init__` method, meaning that there is some redundancy. Compare the Wizard class's `__init__` method above to the Person class below to see how they overlap:

```
class Person:

    def __init__(self,name,age,gender):

        self.name = name

        self.age = age

        if gender == 'm':
```

```
self.isMale = True

elif gender == 'f':

    self.isMale = False

else:

    print("Gender not recognised!")
```

It may be that we want to change only a part of the superclass initialisation behaviour. In that case, we can **invoke the superclass `__init__` method directly**. It would perform its usual initialisation of the current data bundle, then the subclass could have additional commands to modify the initialisation. **Task 5, use example of a new `__init__` method definition for the Wizard class:**

```
class Wizard(Person):

    def __init__(self,name,age,gender):

        Person.__init__(self,name,age,'m')

    ....
```

This definition invokes the superclass (Person) `__init__` method with call:

```
Person.__init__(self,name,age,'m')
```

Note how the special variable `self` is explicitly passed as the first argument.

### Adding extra methods to subclasses

**Task 6, you can also add completely new methods to add functional behaviour that doesn't exist in the superclass.**

```
class Wizard(Person):

    ...

    def spell(self):
```

```
print('Expelliarmus!')
```

Here, instances of the Wizard class have a spell method, whereas instances of the Person class have no such method.

**Test it by yourself and show the result to the people sitting next to you!**

## An object oriented programming project

### Pre-project setting

Before this task, I'd recommend revising the OOP chapter and watching a very good video about OOP [here](#) and also to make sure you fully understand the concept and syntax of OOP.

For this task, it is vital that you use an appropriately **configured** shell: select F6 for the configuration options, then choose both “**Execute in a new dedicated Python interpreter**” and “**Interact with the Python interpreter after execution**”.

### Project intro

This session aims to provide an opportunity to practice using the object oriented approach to programming in Python. This will be done through a task that involves **animating simple shapes**, which will allow you to practice using both classes and objects, along with inheritance.

### Pre-load file

First, ask the **pre-setting files**: **Shape.py** from your instructor and save in a sensible location on your PC. e.g. **Python\_coding\_course/ch13/**

The code in the file **Shapes.py** provides some quite basic graphics capability. For this task you **don't need to fully understand how this piece of code works, just how to call it for now.**

## Make your first moving figure

**Task 7\_1, to begin, create a file name like ch13\_load\_shapes.py** (writriting a single line of code does **imports the Shapes Python module locally**) and run it. To try out the module, enter the following commands into the interpreter window:

```
frame = Frame()

s1 = Shape('square',100)

s1.goto(200,100)
```

The first command creates an instance of the Frame class and opens a graphics window containing an outer 'frame' line, marked with min/max x and y coordinate values (as defined in class Frame \_\_init\_\_).

The second command creates an object of the Shape class, assigning it to the variable s1, thus causing a square figure to appear in the window. The figure appears with its central point positioned at coordinate position (0,0).

The figure's shape is set by the **first argument**, in this case 'square', (other possible values include 'diamond' and 'circle' -- check the Shape class in the file). The **second argument** sets the figure's diameter.

The third command calls the figure's goto method, moving it to the specified (x, y) coordinates. That's all you need to know about how the Shapes Python module works. You can now close the window by calling `frame.close()` or by causing the shell to exit.

## More obvious moving

The above is sufficient for simple animation, **for an extension of task 7\_1, by iteratively making a figure goto new locations as movement.** For example, if we initialise variables x and y to 0, try entering the following code to animate our sample square:

```
>>> for i in range(100):

    s1.goto(x,y)
```

```
x = x + 5
```

```
y = x + 5
```

## Creating our first moving figure

**Task 7\_2, define classes for moving shapes.** Create a file called `MovingShapes.py`. Write the skeleton of `MovingShapes.py` as below, which includes some basic code:

```
from Shapes import *

from pylab import random as r

class MovingShape:

    def __init__(self,frame,shape,diameter):

        self.shape = shape

        self.diameter = diameter

        self.figure = Shape(shape,diameter)

    def goto(self,x,y):

        self.figure.goto(x,y)

    def moveTick(self):

        pass

#####

class Square(MovingShape):

    def __init__(self,frame,diameter):

        MovingShape.__init__(self,frame,'square',diameter)

#####

class Diamond(MovingShape):
```

```
def __init__(self,frame,diameter):

    MovingShape.__init__(self,frame,'diamond',diameter)

#####

class Circle(MovingShape):

    def __init__(self,frame,diameter):

        MovingShape.__init__(self,frame,'circle',diameter)
```

Study this code to make sure that you understand it.

Note that the `MovingShape` class has an **incomplete definition**. Its initialisation method currently only creates variables to store the object for the figure created (an instance of the `Shape` class), plus its associated details. Instances also need to **store information for their current location and rate of movement** so that the information can be iteratively updated. There is a `goto` method, which works simply by calling the figure's `goto` method. There is also a `moveTick` method, which is meant to update a `MovingShape` instance's position with each tick of the clock, but this currently has an empty definition (this consists of a single pass command, a special null command, that doesn't perform any action).

The file also contains currently **incomplete definitions** of the classes: `Square`, `Circle` and `Diamond`, which all inherit from `MovingShape`, each having an `__init__` method that invokes the superclass's `__init__` method, instantiating its shape parameter appropriately, as 'square', etc.

### Test MovingShape.py

**For testing Task 7\_2, as the `load_shape` file we created earlier, for this project we need to create a test files called `test_one_shape.py`.**

E.g. The code in the file `test_one_shape.py` can be:

```
from MovingShapes import *
```

```
frame = Frame()

shape1 = Square(frame,100)

for i in range(100):

    shape1.moveTick()
```

This file tests the code in `MovingShapes.py`, by creating an instance of `Square` and attempting to move it, which, given the `moveTick` method's null definition, currently has no effect .

(Note, again, that it is vital to use an appropriately configured shell, as specified above.)

### Adding code in `MovingShapes.py` to move the figure

**Task 7\_3, follow instructions below, see if your code can achieve the requirements described:**

Begin by adding some instance variables (i.e. having names of the form `self.x`) to the `__init__` method of the `MovingShape` class, to store an object's position as `x` and `y` coordinates. Initialise the starting position as `(0, 0)`, as that's where the figure initially appears when added.

Also add instance variables to specify the rate of movement in `x` and `y` directions — you might use `self.dx` (for 'delta-x') and `self.dy` for the latter attributes. You can try out different initialisation values for these variables, but values around 10 are a good place to start.

Finally, to achieve animation, we must complete the definition of `moveTick`. This must first update the `x` and `y` position variables by adding the corresponding position change values (e.g. adding `self.dx` to `self.x`, etc). It should then call the `goto` method, so that the figure moves to the new location. If everything is correct, then running the test file `test_one_shape.py` should open a graphics window, in which there is a square which travels across the graphics window.



## Adding random variation — velocities

**Task 7\_4, create a 2nd test file test\_multi\_shapes.py, type and study the code below:**

```
from MovingShapes import *

frame = Frame()

numshapes = 3

shapes = []

for i in range(numshapes):

    shapes.append(Square(frame,100))

for i in range(100):

    for shape in shapes:

        shape.moveTick()
```

This test file attempts to animate several shapes at the same time. Observe that it creates several instances of the Square class — how many is determined by the variable numshapes. These instances are stored in a list (shapes).

When the second ‘animation’ loop of the code runs, there is an embedded loop that calls the moveTick method of each Shape object. The results, however, are not very interesting, as the shapes all start at the same position and have the same speed/direction, so all travel in unison.

## Creating random movement

MovingShapes.py imports the function random from pylab, renaming it as r. Hence, a call r() returns a random value between 0 and 1 (try typing r() into your console to see what it does), with which we introduce variation into the movement vectors.

Thus, as 10 has been a reasonable value for self.dx so far, we might now assign a value such as “5 + 10 \* r()”, which will be a random value between 5 and 15.

Implement this idea for both the x and y velocity components. Run the test file several times to see the variations introduced.

### Move in positive and negative directions

Although the shapes should no longer travel together, we've introduced only limited variation, as all values are positive (only go in the positive direction). The test "`r() < 0.5`" returns True 50% of the time, so it can be used as a conditional to randomly to flip individual velocity values from positive to negative. **So for task 7\_5, try that out.**

### Minimum and maximum start position

Figures should ideally start at random locations, preferably not overlapping the line frame of the graphics window. Note that when we create a MovingShape instance, frame is provided as a parameter (re-read Shapes.py). A Frame object has attributes `frame.width` and `frame.height` (i.e. the frame's dimensions). **In task 7\_6, you will use this info to computing a suitable random start location for the figure.**

Consider first the x component of a shape's (x, y) position. To **avoid the figure overlapping the border**, there is a minimum x value that should ever be assigned, which is determined by its diameter d. To take the case of a Square, this value should not be less than d/2. Likewise, the maximum value for x should not be more than `frame.width` minus d/2. Similar reasoning produces min and max values for y. Compute these values and assign them to instance variables (`self.minx`, `self.maxx`, etc), as they will be useful again later on. **Tip, you may want to do some part of this as a subroutine function and decompose the complexity.**

### Adding random variation for the start positions

When assigning a random position to a shape, a suitable x value is one that lies randomly between `self.minx` and `self.maxx`, which might be computed as shown immediately below. Implement this idea to assign random positions to objects and test that it works as expected.

```
self.minx + r() * (self.maxx - self.minx)
```

## Hitting the wall

We can now look at adding some more interesting behaviour to figures. We can treat the border in the window as a wall against which figures bounce. The 'physics' here is simple. Consider a square which is moving upwards to the right, which hits the right hand wall. On bouncing, it would continue to move upwards, but now away from the wall. Assuming no energy is lost, we can achieve this result simply by reversing the x velocity component (i.e. multiplying by -1). The same is true for a figure bouncing off the left wall. For a figure hitting the top or bottom wall, we need only reverse the y velocity component.

**For this bouncing task (task 7\_8), the natural place to include code for this behaviour is in the definition of the moveTick method.** Thus, after we've updated the position of the figure, we can check if the new position is one that counts as 'hitting the wall'. For the x component, this is true if the new x position (say self.x) is less than the minimum x position, or is more than the maximum x position (meaning when 'hit the wall'), which are values we computed in the previous section and stored as variables self.minx, self.maxx. Implement this idea by adding conditionals to the moveTick method and test that your code behaves correctly.

## Diamonds vs. squares

It's easy to see that the behaviour we've created for squares is not correct for diamonds. This is because the tilted orientation of a diamond makes its width and height greater than that of a square with the same specified diameter, so the practical diameter d2 is such that  $d2 = 2 \cdot d1$ . Hence, our calculation of min/max x, y values is incorrect for diamonds, with the consequence that the 'initial positions' allowed for diamonds where their corners overlap the boundary, which is also true of the point at which diamonds 'bounce'.

For this reason we need different figures to have different behaviour, specifically in the calculation of min/max x and y values. We could do this in various ways, but since the calculation done previously for this works for both squares and circles, let's

keep this as the default in the MovingShape class, but **have the diamond subclass override it (overwrite the method under diamond class)**. To do this, take the segment of code that calculates the min/max values in the `__init__` method of the MovingShape class and put it into a separate method, which is then called within the `__init__` method in place of the original code segment. If you then add an alternative definition of this method to the diamond subclass, which does the different calculation required for diamonds, this will serve to override the superclass definition for diamonds. For squares and circles, the superclass definition will still be used.

If you have done and understand this part, update the test file as below to see the combination of all three shapes inside the frame. This will be your task 7\_9.

```
from MovingShapes_task8 import *

frame = Frame()

numshapes = 3

shapes = []

size = 60

for i in range(numshapes):

    shapes.append(Square(frame,size))

    shapes.append(Diamond(frame,size))

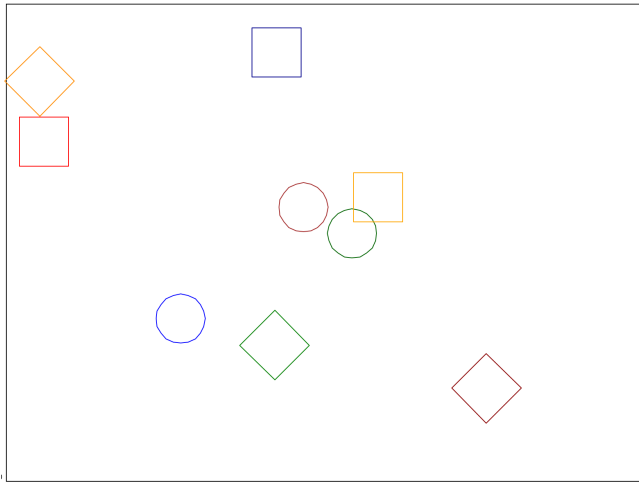
    shapes.append(Circle(frame,size))

for i in range(300):

    for shape in shapes:

        shape.moveTick()

frame.close()
```



## Q/A Python modules and Classes

### Python modules can contain Classes

There may be some confusion between Python modules and Python Classes as people can mention importing a Python module or importing a Python Class.

Classes are just a part of Python code related to OOP, whereas Python modules are files that contain Python code (which can include OOP structure code). This means Python modules can contain Python Classes. Sometimes a Python module consists of a single Python Class, in this case a Python module may seem synonymous with a Python Class.

### Similarity

For Python modules and classes, they both use 'dot' notation to execute methods that are inside them.

For Classes or OOP, you have to do it this way, We have seen many examples for OOP above. Here is an example for a module: Let's import the numpy Python module (numpy is a big Python module or library which contains many Classes):

```
import numpy
numpy.mean([1,2,3])
2.0
```

And if you directly use the mean method, you get an error.

```
mean([1,2,3])
Traceback (most recent call last):
  File "<ipython-input-29-d10ac0b6465b>", line 1, in <module>
```

```
mean([1,2,3])
```

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

Therefore if you import the numpy module, you will need to run the mean method by using `numpy.mean`. However, you do not have to use this 'dot' notation if you use a different importing process, which is to import only the variable/method/class directly from the module. Using the same example:

```
from numpy import mean.
```

```
mean([1,2,3])
```

```
2.0
```

In this case, you will not get an error, as you have only imported the method (not the full module) so do not require 'dot' notation.

I hope this is clear now. You can also watch this [video](#) to see some more examples.

## Part 6 - Real World Applications

### Chapter 14 - Databases

#### Learning outcomes

- Introduce mySQL and SQLite.
- Learn how to create a simple database table.
- How to select data from a database.

#### Introducing databases

A database is an organized collection of data, stored and accessed electronically. Database designers typically organize the data to model aspects of reality in a way that supports processes that require information.

##### Model data in rows and columns

These model data as rows and columns in a series of tables, also the vast majority use SQL for writing and querying data.

SQL combines the roles of data definition, data manipulation and querying in a single language. It was one of the first commercial languages for the relational model, although it departs in some respects from the relational model as described by Codd (for example, the rows and columns of a table can be ordered). SQL became a standard of the American National Standards Institute (ANSI) in 1986 and of the International Organization for Standardization (ISO) in 1987. A very good example of how to use SQL in Python is [here](#), which you can read after reading this chapter.

##### Why are databases necessary?

At a high level, web applications store data and present it to users in a useful way. For example, Google stores data about roads and provides directions to get from one location to another by driving through the Maps application. Driving directions are possible because the data is stored in a structured format.

Databases make structured storage reliable and fast. They also give you a mental framework for how the data should be saved and retrieved instead of having to figure out what to do with the data every time you build a new application.

## Relational database

The database storage abstraction most commonly used in Python web development is sets of relational tables, we will cover more about this in module 3.

Just for a heads up, the relational databases store data in a series of tables. Interconnections between the tables are specified as *foreign keys*. A foreign key is a unique reference from one row in a relational table to another row in a table, which can be the same table but is most commonly a different table.

## Database third-party services

Numerous companies run scalable database servers as a hosted service. Hosted databases can often provide automated backups and recovery, tightened security configurations and easy vertical scaling, depending on the provider.

- Amazon Relational Database Service (RDS) provides pre-configured MySQL and PostgreSQL instances. The instances can be scaled to larger or smaller configurations based on storage and performance needs.
- Google Cloud SQL is a service with managed, backed up, replicated and auto-patched MySQL instances. Cloud SQL integrates with Google App Engine but can be used independently as well.
- BitCan provides both MySQL and MongoDB hosted databases with extensive backup services.

## mysql.connector

### Syntax to connect mysql

E.g. In order to connect to a database you will need to use code similar to what is below:

```
import mysql.connector
```



```
mydb = mysql.connector.connect(  
    host="localhost",  
    user="yourusername",  
    passwd="yourpassword"  
)  
  
mycursor = mydb.cursor()  
mycursor.execute("CREATE DATABASE mydatabase")
```

You will probably use the above version when you do real work.

## SQLite

In this chapter we will learn a light version of MySQL as we don't have a real database to experiment with (you will need to know a server host, username and password to connect a database with MySQL, but we cannot access those at the moment).

SQLite allows us to create a database locally and experiment on it. In most cases, SQL and SQLite use the same commands.

## Creating a database table

Let's start the journey of using databases! Open your Spyder editor and create a file named `ch14_createTable.py`.

### Import and connect a database

We first need to import the SQLite toolkit `import sqlite3`

As shown in the MySQL example above, the very first step to using a database is to connect to it using the `connect()` function. As we are going to create the database, we can just name a database file with `.db` as the input of the `connect()` function at this stage. e.g.

```
conn = sqlite3.connect('task1.db')
```

## Connect cursor

We then need to link our database with cursor, where cursor is a low level pointer specifically used in databases to find positions of data in each data table. The function is very simple to use:

```
c = conn.cursor()
```

## Create table

After the pre-setup, we can now create a table by using the **execute** function. As you will see, this function can be used in many ways.

As we don't have to create a table everytime we run the code, let's create a function about it and only call it when you need to:

```
def create_table():  
    c.execute('CREATE TABLE IF NOT EXISTS stuffToBuild(unix REAL , datestamp TEXT, keyword TEXT, value REAL)')
```

Here 'CREATE TABLE IF NOT EXISTS' is the command we use to create and check table existence. This is SQL language rather than Python, but within the `execute()` function, we can put a SQL command as an input.

## Uppercase commands

Please note, that SQL commands don't have to be in uppercase. SQL is a case insensitive language, but it is useful to put them in uppercase so that we can more easily distinguish which are SQL related commands and which are table and column names that we give for the database.

## Database table

So, 'stuffToBuild' is the database table name, while 'unix', 'datestamp', 'keyword' and 'value' are the column names in the database table. You can define any names you like, so we put them in lowercase. 'REAL' and 'TEXT' are the data types and format in each column, which are also SQL language, therefore we write it in uppercase.

## Inserting data into tables

Well, the table we created is empty with no real data but only a data frame. Therefore we need to put data into the table by using `'INSERT INTO'` SQL command. The function below gives you an example of how to insert data:

```
def data_entry():  
    c.execute("INSERT INTO stuffToBuild VALUES(142233222, '2018-01-01', 'python', 5)")  
    conn.commit()  
    c.close()  
    conn.close()
```

### Matching column names to insert row values

In the previous section we created 4 columns in the database (`'unix'`, `'datestamp'`, `'keyword'` and `'value'`) so we will need to insert 4 values into the table, one for each column.

### Routine command

Similar to git command, after inserting (pushing) values into a database, you will need to run the commit command.

If you complete the process, you will also need to close the table and database with:

```
c.close()  
conn.close()
```

### Call the function to create table

Now, let's run the `create_table` and `data_entry` function to see what will happen:

```
create_table()  
data_entry()
```

### Use database browser to check data table

In your folder, a `task1.db` file should appear, which is the local database we just created. You cannot open it with the normal editor, so must use a database browser. If you don't already have a database browser installed in your PC, you can download one [here](#).

Once you open the database file, you can see the table you're just created with the value we just inserted.

## Insert data using variables

We have learned how to create a database and insert data into the database, however in real life applications, databases are usually huge and all the inserted data is automatically added rather than hard coded. However, how to insert huge amounts of data automatically? Variables are one of the best inventions in coding and we will use them here as well.

Save-as your last file as ch14\_insertWithVariable.py.

Let's first import some modules and functions to our script, so as to automatically generate data:

```
import sqlite3
import time
import datetime
import random
```

### Static data and stream data

The `time.time()` function can generate time since unix system start, which we will use as our time data for one column.

**Note, data can be static (pool of data) or dynamic (stream of data), the time data we used in here is dynamic data which will change as the time changes.**

Many databases, e.g. customer history data, will be the static.

### Inserting data dynamically

Now we can write a new function to insert data, setting up variables first. Have a read of the code below and see how much you understand:

```
def dynamic_data_entry():
    unix = time.time()
    date = str(datetime.datetime.fromtimestamp(unix).strftime('%Y-%m-%d %H:%M:%S'))
```

```
keyword = 'Python'
value = random.randrange(0, 10)

c.execute('INSERT INTO stuffToBuild(unix, datestamp, keyword,value) VALUES (?, ?, ?, ?)', (unix,
date, keyword,value))

conn.commit()
```

### Insert multiple rows with a for loop

Then run this code by using the commands below and see the database file change with your database browser:

```
for i in range(10):
    dynamic_data_entry()
    time.sleep(1)

c.close()
conn.close()
```

### No need for a sleep function in real insertion

The use of `time.sleep(1)` is just to slow down the process. If we didn't use this then the time data inserted into our database table may not change enough (computers just work so fast and may insert into multiple rows in 1 millisecond!), leading to lots of too similar values. Usually you don't have to add a `sleep()` function when you insert data.

**Experiment with any parts you are not sure about in the Python console and ask the instructor to explain it to you.**

## Reading data from databases

Now that we have a database with data in it, we can read data from the database and use Python to build tools to analyse it.

### The select statement

One of the advantages of using databases is that they will not use too much computer memory as we **only 'select' what we want to use**. The **SELECT**

**command in SQL is used to only read the data columns based on our choice** (similar to: from xx import xx), whereas SELECT \* means to SELECT all the columns in the data table.

Usually, after nominating the selected data columns, there is a **FROM** command that points to **which data table** we should select from. This is because one database can contain multiple database tables.

A **WHERE** clause is used for **further filtering** the data **inside each column**.

**Task 3, please try the code below in your Python script file:**

```
def read_from_db_all():  
    c.execute('SELECT * FROM stuffToBuild WHERE value =8 ')  
    for row in c.fetchall():  
        print(row)
```

\* means everything! So it select everything from the table.

### fetchall()

The `fetchall()` function is similar to the pull function in git. After nominating which column to select, you need to use this function to produce the action.

### Data select exercises

**Extend task 3, have a try with the code below and try to change the \* to different columns, also try to change the WHERE clause to filter different rows of data.**

```
def read_from_db2():  
    c.execute('SELECT * FROM stuffToBuild WHERE value =8 and unix > 1534855733 and unix < 1534855741 ')  
    for row in c.fetchall():  
        print(row[0])
```

After fetching the data, you then can append any data you want into a list or dictionary etc and make use of it.

### Order by and group by

When you meet more complicated databases, there are other ways to filter data. We will not cover everything here, but for good examples of using GROUP BY or

ORDER BY see [here](#). Remember to check stackoverflow whenever you meet any difficulties.

## Homework

1. Read [Writing better SQL](#) which is a short code styling guide to make your queries easier to read.
2. There are also some good SQL language lessons and tutorials to be found [here](#).
3. Let's build a database table for the Phonebook project. The columns we need to add in the table are:
  - First Name
  - Last names
  - Address Line 1
  - City
  - Postcode
  - Telephone Number
  - Business Category
  - X Coordinate
  - Y Coordinate

## Preparation for next chapter

There's a lot to cover in next week's session, so please make sure you complete the prep work before class:

1. Sign up for a free [Mailgun account](#). Mailgun is a service that developers can use to programmatically send emails (i.e. send emails automatically based on parameters you write in your code). You can send up to 10,000 emails for free every month, so no need to worry about any costs!
2. After you set up your free Mailgun account, don't forget to head over to your email inbox and make sure you activate your account. After you click the activation button/link, you might need to enter your mobile number to verify

that you're a genuine person and not someone trying to create loads of free accounts to spam people with.

3. As another anti-spam measure, you'll need to add your email address to Mailgun's "authorized recipient" list. Head back to your Mailgun account and click [here](#) to add/invite a new "authorized recipient". Add your own email address when prompted. Once you've done that, you'll need to verify your email address again, to say you're happy to get emails. Once you've done that, you should be all set to start using Mailgun!
4. Also make sure to add another friend's (or two) email address(es) to your authorized recipient list as we will be sending some emails next week. (Preferably one of your Python course fellows.)
5. Use pip to install the **requests** library (your pre-course notes explain how to use pip to install things). This library is different from the request functionality that we imported from the Flask library we installed last week.
6. Sign up for a free [Open Weather Map account](#) and register to get a free API key. We'll be using it to do cool stuff with weather information!



## Chapter 15 - APIs

### Learning outcomes

- Use an API to trigger an email to be sent and get weather information.
- Get JSON formatted data from a payload response.

### Introducing APIs

API stands for Application Programmer Interface. It's an interface between your Python code and services like Twitter, Facebook, etc. that makes it possible for you to get data from those services for use in your application.



### What you can do with API ?

Using APIs, you can “programmatically” access data from services like Twitter. This means that with your Python code, you can get your application to talk to Twitter via an API, so that you can get data from Twitter that you can then do stuff with in your

Python app (e.g, showing the last 10 tweets containing a particular hashtag). There are loads of things you can do with APIs. You can read more about some of them [here](#).

## Setting up your first API in Python

Today, we're going to get information from a user. Let's take this scenario – your friend is launching an amazing, innovative new product and has asked you to build a landing page for their website, so people can go there to register their interest and be kept up to date about when the product launches, along with other important information.

### Confirmation email task

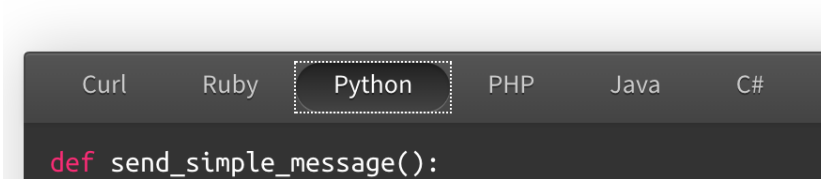
You'll probably want to send a confirmation email to people who've signed up to get that all-important information about your friend's amazing new product! You can make this happen automatically, using Mailgun's API, which we'll be having some fun with today.

## Getting ready to use the Mailgun API

To get started, follow these steps:

1. Log into your [Mailgun account](#).
2. Head to the Sending Email section of the Quickstart Guide by clicking [here](#).
3. We need to include some Python code to connect to the Mailgun server. Near the top of your page, you'll see a menu bar with different programming languages listed. Make sure you pick "Python" from the available tabs.

## Try Now



Then, scroll down to the heading “send via API” and you’ll see a block of Python code that looks like the picture below.

### Send via API

Run this:

```
def send_simple_message():
    return requests.post(
        "https://api.mailgun.net/v3/YOUR_DOMAIN_NAME/messages",
        auth=("api", "YOUR_API_KEY"),
        data={"from": "Excited User <mailgun@YOUR_DOMAIN_NAME>",
              "to": ["bar@example.com", "YOU@YOUR_DOMAIN_NAME"],
              "subject": "Hello",
              "text": "Testing some Mailgun awesomness!"})
```

4. Copy and paste this code into a new file, and save the file as a .py file. E.g. `ch15_send-email.py` or any other .py file name. However, please make sure you don’t call that file “email”, “send\_simple\_message” or “mailgun”.
5. Before we can programmatically send emails, we need to amend a few placeholders in the code above, most importantly:

- 1) **YOUR\_DOMAIN\_NAME**

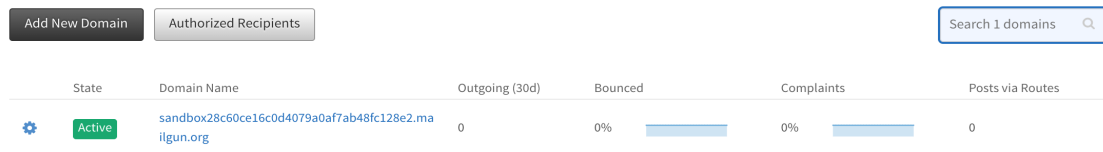
Since we’re just testing with this account, Mailgun gives us a domain name we can use instead of having to get our own (more on this in a bit).

- 2) **YOUR\_API\_KEY**

An API key **allows a service to identify you as a "customer" of their API**. Services often ask you to register for one so they can give you a private key that allows them to monitor your account for any unusual usage patterns (like thousands of spam emails suddenly being sent). This means we need to get

an API key and domain name from Mailgun, so that the service knows it's our app communicating with their server.

- 3) To get your Mailgun API key, go to your [Mailgun Dashboard](#) and scroll down to the section called Domains:



Add New Domain		Authorized Recipients		Search 1 domains		
State	Domain Name	Outgoing (30d)	Bounced	Complaints	Posts via Routes	
Active	sandbox28c60ce16c0d4079a0af7ab48fc128e2.mailgun.org	0	0%	0%	0	

Under the Domain Name column, you'll see something that starts with "sandbox". Click on this and you'll be taken to a new page with all the information you need.

On this new page you'll see an API key and domain name (that long URL that starts with sandbox). Update the Python code above that you pasted into your new Python file, with the API key and domain name provided by Mailgun.

Notice for the data variable in the code, there are several fields listed. "from":, "to":, "subject": and "text": are also things that you can update. For now, go ahead and change the "to": field to your email address (the one you added to the authorised recipients list, if not already there), then save your file.

## Testing the code for your Mailgun API

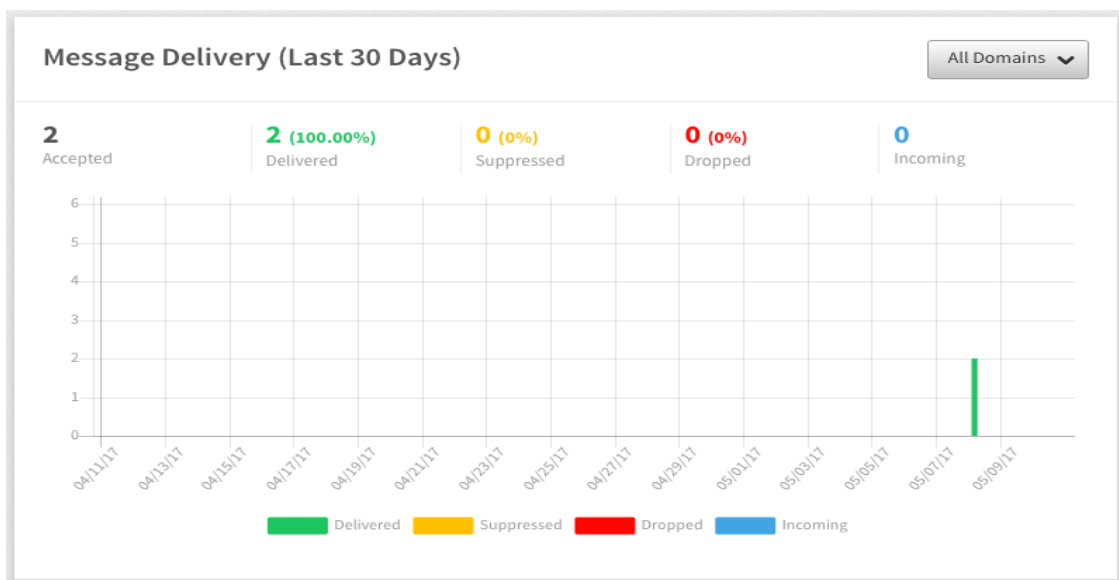
You're nearly ready to test your code!

To test whether your Mailgun API code works, you'll need to run the file you just created and look at the Mailgun dashboard to see whether everything worked. Before you run the file though, you need to add `import requests` at the very beginning (to import the requests library) and `send_simple_message()` at the end of it to call the function.

## Task 1

Run your file. What information do you see in the Mailgun logs? You might need to click on the “Refresh” button. If you see a response code that looks like 2xx (such as 250), that means everything went well as far as Mailgun knows.

You can check your [Mailgun dashboard](#) to see whether an email has been sent or not. Emails that fail to send are marked as “dropped” and highlighted in red. You can see the details of the failure(s) by clicking on Open Logs. The logs show a list of all email send attempts and whether they were successful or not, along with a representative error message if they failed.



Keep in mind that because **your Mailgun account is a test account**, you’ll need to make sure whichever email address you submit above has been added to your “authorised recipients” list in Mailgun, otherwise the email won’t send to that address.

## Account

Success: Invited recipient.

Settings Security **Authorized Recipients** Upgrade

### Authorized Recipients

Invite New Recipient

Delete Selected

<input type="checkbox"/>	State	Email
<input type="checkbox"/>	Unverified	chenh

## Using an API to find out what the weather is today

Now, we're going to play with [OpenWeatherMap's API](#), which enables us to query the weather for a given city and use the data in our web app!

### Task 2

- Let's start off by creating a new file called **weather.py** and adding the following code to it:

```
import requests
```

```
endpoint = "http://api.openweathermap.org/data/2.5/weather"
```

```
payload = {"q": "London,UK", "units":"metric", "appid":"YOUR_APP_ID"}
```

```
response = requests.get(endpoint, params=payload)
```

```
Print (response.url)
```

```
print (response.status_code)
```

```
print (response.headers["content-type"])
```

In the code above, we're using Python's **requests** library to make a request for information about the weather using the Open Weather Map API.

Notice how we don't use the word "from" before "import". This is because 'requests' is a library that comes built-in with Python.

You'll see a couple of new words here: **endpoint** and **payload**. '**endpoint**' is just another word for destination or location for the data we want to fetch, while '**payload**' is just another word for cargo (i.e. all the information we want to send to the endpoint, in order for the server to know what parameters to include in the response).

What data do you think is being sent in the payload above, to the endpoint? Notice the use of quotation marks for the endpoint and payload parameters.

The documentation for the service you're using will tell you what sort of options you can include in your payload, and what the endpoint's URL is.

## HTTP status code

As you may already know, HTTP stands for Hypertext Transfer Protocol – it's the set of rules that govern how clients and servers talk to each other. 200 is an **HTTP status code**, just like 404 (page not found) and 500 (internal server error). HTTP status codes provide useful information you can use when troubleshooting.

## Task

1. Just like the Mailgun exercise, there's some placeholder information here that we need to update. `YOUR_APP_ID` is a placeholder that, for testing purposes, you can update with your own API key (which you requested from OpenWeatherMap as part of the prep work for this session).

2. As you can see in the code above, we can print the information we get back in the response.e.g.

```
print (response.url)
```

```
http://api.openweathermap.org/data/2.5/weather?q=London%20UK&units=metric&appid=0e4b40375a943eac6113b2445aebc8c2
```

```
print (response.status_code)
```

```
200
```

```
print (response.headers["content-type"])
```

```
application/json; charset=utf-8
```

- `response.url` is the actual URL that requests use to get a response for you. If you copy and paste the URL that is printed to your command line into your browser, what do you see?
  - `response.status_code` is the HTTP status code of the response. In HTTP, when a request succeeds, we expect to see the message '200 OK'. You don't need to include it for your code to work, but it can be helpful to print information like this in your command line window, so you can see what's happening when your code runs.
  - Likewise, you can use `response.headers["content-type"]` to print the content type of the response (as defined in the HTTP header) in your command line.
3. So now that you've got a whole bunch of information back from the server, you're probably wondering how you can access just the data you want from the response! You do this using `response.text`. Try it out for yourself, by adding `print (response.text)` at the end of your file and running it.

Recap with debugging tips, you'll see that printing variables to your command line/console is a really handy way to check that you're getting the results you expect and, if you're not, it will help you find where the problem might be!



## Unicode

In some response you might find some unusual characters. These strings we get back in the response are called **unicode** strings. Unicode strings can include characters from other alphabets (e.g. Ж, ऋ, پش) as well as emojis, which you don't see in regular text strings. Unicode encoding of strings is often used in web applications, because they may have to handle text input from users all around the world. For example, think of how many countries Facebook is in and how many millions of users the service has who don't speak any English!

## Using JSON to report the weather in a user-friendly way

Ok, so, now we know how to get an API response; but it's not really in a nice format, is it? Wouldn't it be nice to print a sentence about the weather instead of just the raw output of the API?

We're in luck, because we can do this using a data interchange format called **JSON**. JSON notation is easy for humans to understand and easy for machines to work with. It's basically a collection of name/value pairs that you can use to store and parse (pick out) information. It looks something like this:

```
{ "users": [
  {
    "firstName": "Ray",
    "lastName": "Villalobos",
    "joined": {
      "month": "January",
      "day": 12,
      "year": 2012
    }
  },
  {
    "firstName": "John",
    "lastName": "Jones",
    "joined": {
      "month": "April",
      "day": 28,
      "year": 2010
    }
  }
]
}
```

Notice something? It is quite similar to a Python dictionary where each item has a key.

When you pasted the API response URL in your browser, what you saw was the response in its raw format. To get something that's easier to read, we first need to convert the response to JSON format, then store it in a variable so that we can do stuff with the JSON data. We can do this as follows:

```
data = response.json()
```

Remember how we used methods like `.upper()` and `.title()`? Just like those converted your strings into uppercase format or a title format, `.json()` takes your response and converts it into JSON format. You can check the contents of the data variable by printing it to your command line.

You'll see from OpenWeatherMap's documentation that to access the value stored in JSON for temperature, we can use this notation: `data['main']['temp']`, try to understand it as a nested dictionary, which means for the big `data` dictionary, choose key `'main'`; as the matching value of `data['main']` is also a dictionary (the inner dictionary):

```
>>>data['main']
```

```
{'temp': 13.28,
```

```
'pressure': 1015,
```

```
'humidity': 58,  
'temp_min': 12,  
'temp_max': 14}
```

We then choose key `'temp'` for the `data['main']` dictionary as `data['main']['temp']` and the matching value will be `13.28`.

### Task 3

1. Update your **weather.py** file so that it converts the response to JSON format, prints the temperature in London and also prints the response in its original (non-JSON) format. Your code should look like this:

```
import requests  
  
endpoint = "http://api.openweathermap.org/data/2.5/weather"  
payload = {"q": "London,UK", "units":"metric", "appid":"YOUR_APP_ID"}  
  
response = requests.get(endpoint, params=payload)  
data = response.json()  
  
print (data['main'])  
print (response.url)  
print (response.status_code)  
print (response.headers["content-type"])  
print (response.text)
```

2. Update your Python file so that it prints out a nice statement to your command line about the weather:

```
temperature = data["main"]["temp"]  
name = data["name"]  
weather = data["weather"][0]["main"]  
print (u"It's {}C in {}, and the sky is {}".format(temperature, name, weather))
```

## Homework

1. Have a look at [Mailgun's API Reference for Sending Messages](#). Can you figure out how to:
  - Attach a file to an email?
  - Set a specific delivery time for an email (so that the email is not sent immediately, but after a fixed delay or at a specific time)?

## Preparation for next chapter

We'll be using a framework called **Flask** next week. You'll need to install it on your laptop using pip, before you can start using it. Your pre-course notes explain how to install things using pip.

## Extra Homework (optional)

1. Learn Python The Hard Way exercises [38](#) and [39](#).
2. With your new knowledge about APIs and JSON, can you figure out how to use [Giphy's API](#) to get the first GIF returned from doing a search for a user's favourite animal?

# Chapter 16 - Flask, and Using Python and HTML Together

## Learning outcomes

- Write and run a simple Flask web app.
- Use route paths and decorators along with functions to do different things on different “pages” of your web app.
- Pass variables between Python and HTML using `render_template()`

## Recap from last session

- What are APIs used for?
- What's an API key?
- What's JSON, and why might we want to use it?

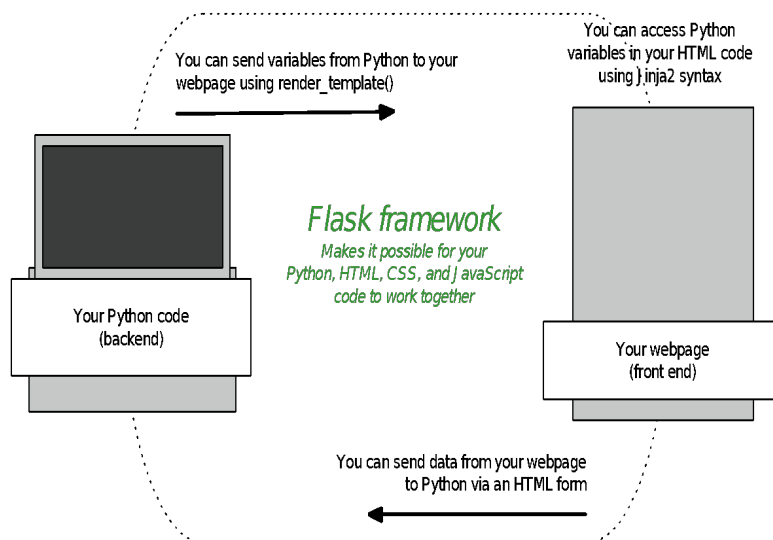
## Let's get started!

In order to get your Python code working together with your HTML code so that you have an interactive website, you need to use a framework for building lightweight web applications.

A **web framework** is a set of code that's already been written, used to automate tasks – such as the way your Python code, HTML code, web browser and server work together. In other words, it makes sure your server knows how to take your Python code and HTML code and make them work together to display your website.

You may have heard of different frameworks that can be used for web apps, like Django, Pyramid, Flask, etc. In this course, we will use Flask – it's commonly used and has [good documentation](#).

Here's a simple illustration of how Python, Flask, Jinja2 and HTML interact with each other:



You'll see the terms "interactive website", "app", "web app", "web application", etc. on this course – we basically use them interchangeably.

## Making and running your first app with Flask

Make sure you have installed: `pip install Flask`; `pip install virtualenv`. Now that we'll be using Flask, you'll notice a few small differences when you write your code, as well as when you run it.

In your Python file, you'll be adding a bit of code so we can make use of the libraries that come with Flask. You'll also notice different information in your command line when you run your file.

## Task 1

1. Make a new file called **hello.py** and paste in the code below. Remember, don't call your files "python" or "flask" – things won't work if you do! Don't worry about running it yet. We'll talk you through what the code does first.

```
from flask import Flask
app = Flask("MyApp")

@app.route("/")
def hello():
    return "Hello World"

app.run(debug=True)
```

## Import Flask

Recall importing from the function chapter? In order to make use of Flask's libraries, and also the libraries/modules that come built-in with Python, we have to "import" the ones we want to use. That's why, at the beginning of this file, you see `from flask import Flask` this is Python's way of making different modules and libraries available for us to use in our app. Notice one is capitalised and the other one isn't (flask is the file/library name and Flask is the function/class name inside the library).

Since you've already installed Flask, you can go ahead and import it. You'll notice if you try to import a Python module or library that you haven't already installed, you'll get an error.

## Routes & decorators

You may be wondering what `@app.route` stands for and why we're using it. This is what's known as a decorator.

The concept of decorators exists in most programming languages; decorators allow you to **execute code and functions at a particular location**. The syntax `@app.route`, comes from Flask, and tells the server to do some things internally for us so that when your browser sends a request into Python, Python and your server know what to do with it. The inner workings of this are beyond the scope of this course, so don't worry about it for now.

A decorator always starts with the **@ sign** and goes on the line right **before a function is defined**. In simple terms, you can think of "route" being like a "route to somewhere" or "path to a place" or "at this location". Remember, a **decorator isn't the same as a function** – it tells you the path in your web app where you want to run your functions.

## app.run()

### Save Flask output in an app variable

In the code above, you'll notice some new code we haven't used before: `app = Flask("MyApp")`, where Flask is the function, "MyApp" is the function input, the returned result of the function will be saved in variable app. (Basically says "I have a Flask app called "MyApp" and to make things easy to write, I'll call it app").

### run()

At the very, very end of your file, `app.run(debug=True)` makes sure that Flask runs your app. Here run() is a method that only can be used with the app object (the Flask function's output)

Using `debug=True` doesn't actually debug anything, it simply provides you with extra information that you can use to debug your code yourself. For example, instead of just getting a blank page when there's a problem, and going crazy trying to figure out where the problem is, when this bit of code is included, Python will helpfully tell you (in your browser) where the problem is. If the problem is actually with your Python code, it will also include the line(s) that caused the problem, a standard name for the error type, etc.



**Task 1\_2**

1. Run your **hello.py** app from your command line `python hello.py`
2. Navigate to `localhost:5000` or <http://127.0.0.1:5000/> in your web browser (localhost:5000 is your computer's local server rather than the internet) – what do you expect to see?
3. Edit **hello.py** so that it returns something other than “Hello World!”
4. Refresh the page in your browser and see what happens.
5. In your browser, type in a URL like `localhost:5000/idontexist` - what happens?

Now that you're running a web app and not just a simple Python file, when you make changes to your .py file, you need to stop and restart your server in the command line (using `control-c`), otherwise your changes won't be picked up.

## A little more about decorators

In the beginners' course you learned about relative links, where “relative” means relative to the website's “root” or index page.

This concept is really helpful to think about when you start building your web app/ interactive website. For instance, have a look at BBC's [website](#). Even though the examples in the table below are of static webpages, you might want your interactive website to do different things at different locations on your website.

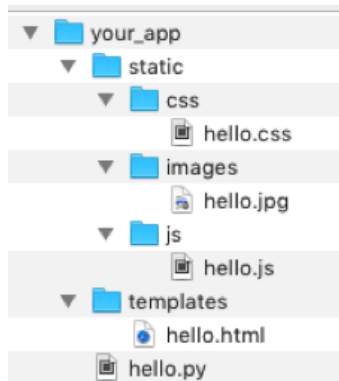
Decorators are used to specify the locations on your site where you want things to happen. In the task above, when you typed in a URL that didn't have a decorator in your Python file, you got an error. Flask only knows what to do at that URL if you tell it what you want it to do in Python using a decorator.

URL	RELATIVE PATH	EXAMPLE DECORATOR
bbc.co.uk	/	<code>@app.route("/")</code>
bbc.co.uk/news	/news	<code>@app.route("/news")</code>
bbc.co.uk/news/technology	/news/technology	<code>@app.route("/news/technology")</code>

Now change your code from `@app.route("/")` to `@app.route("/bio")`, and run your code. Open your web browser with url: <http://127.0.0.1:5000/bio> to see what happened.

## Setting up the file structure for your app

Flask looks in specific places for all the files that make up your web app. This means there's a file structure that you need to use so that everything works properly:



Notice the different types of files in each folder, like the .html file in the **templates** folder. This is very important! If you don't use this file structure, Flask won't know where to find your files, so things won't work.

## Serving your HTML using Flask

So far, we haven't actually served any HTML pages to our browser. We've just printed some plain text there from Python, without any formatting.

### **render\_template**

The Flask framework enables us to "serve" our HTML pages by using a function called `render_template` in our Python code. We can pass arguments to `render_template()` to tell Flask what HTML page to display (in Flask a template means a HTML file) and to indicate any variables in Python that we want to pass along to our HTML file.

### Task 3

The `render_template` function is part of the `render_template` library, so you'll need to import that library before you can use it, by updating the first line in your `hello.py` file, so that it looks like this:

```
from flask import Flask, render_template
app = Flask("MyApp")
@app.route("/")
def hello():
    return "Hello World"

app.run(debug=True)
```

## Using Jinja2 syntax in your HTML files

One of the libraries Flask uses is called [Jinja2](#). Jinja2 is a templating language that enables you to access values in your HTML file that you pass across from your Python code, so that you can display these values on your web page or use them to evaluate conditional statements (for example, to show or hide something on your web page).

Jinja2 is really powerful, but beyond the scope of this course. We'll show you a few of the basics, but feel free to read more about it [here](#) and make use of it if you like.

### Task 3

1. Create a new folder called **templates**.
2. Below is some HTML code that uses Jinja2 syntax. Copy this code into a new file and save that file in your templates folder as **hello.html**:

```
<!doctype html>
<html>
  <head>
    <title>My Flask App Page</title>
  </head>
  <body>
    {% if name %}
      <p>Hello {{name}}!</p>
    {% else %}
      <p>Hello anonymous person!</p>
    {% endif %}
  </body>
</html>
```

Notice a few new things that we haven't seen before in HTML files:

- **name** is a variable that's been passed across from Python. In Jinja2 syntax, variables go inside of double curly brackets (more on this in a bit).
- There are if and else statements. HTML is used for the layout of web pages, which means if and else statements don't actually exist in HTML; but we can use Jinja2 to get around that. Conditional statements and for loops go inside curly brackets with percentage signs. You can read more about syntax of Jinja2 for if statements [here](#) and for for statements [here](#).

We're not using CSS or Javascript today, but here's an example of how you link to CSS in your HTML file using Jinja2. For a file called main.css:

```
<link rel="stylesheet" href="{{ url_for("static", filename="css/main.css") }}">
```

Of course, you can still link to a file that's hosted online, like in this example:

```
<link rel="stylesheet" href="//code.jquery.com/ui/1.11.4/themes/smoothness/jquery-ui.css">
```

## You're nearly ready to serve your first HTML file with Flask!

You can use Flask to take parameters from a URL. This means you can take what a user enters in the URL field in their browser and use that information to do stuff in Python.

Have a look at this code:

```
from flask import Flask, render_template, request

app = Flask("MyApp")

@app.route("/")
def hello():
    return "Hello World"

@app.route("/<name>")
def hello_someone(name):
    return render_template("hello.html", name=name.title())

app.run(debug=True)
```

### Task 4

1. Update your **hello.py** file with this code, so that it looks like the example above.

```
@app.route("/<name>")
def hello_someone(name):
    return render_template("hello.html", name=name.title())
```

In your command line, when you ran your hello.py file `python hello.py`, you might have noticed the change in <http://127.0.0.1:5000/>, <http://127.0.0.1:5000/name>. **Try to type**

your own name after 5000/, rather than the 'name' variable, and see what happens!

## Getting user generated data from your webpage to Python

Don't worry if the relationships between Python, Flask, your webpage and Jinja2 doesn't all make sense yet – as we continue through the course (and you run more code) things will become clearer!

We've talked about how you can display HTML in your browser and how you can send data from Python to a HTML template using `render_template()`.

You also can ask a visitor to your website to give you information that you can use, for example, to sign them up to your mailing list, get them to log into your service, to recommend recipes or theatre shows based on their interests, to record the results from their scientific experiment... the possibilities are endless!

You'll remember from the beginners course that you can request information from a user by using a [HTML form](#). In Python, that's also how we'll request information.

Email

Password

Radios

- ☒ Option one is this and that—be sure to include why it's great
- ☐ Option two can be something else and selecting it will deselect option one
- ☐ Option three is disabled

Checkbox ☐ Check me out

### Super simplified explanation of GET and POST requests

In the world of HTTP, there are many different types of requests you can make. For this course, we'll be concentrating on just two of them: **GET** and **POST**. You'll see in your Python and HTML code that GET and POST are request "methods" – method here just means the way that the request needs to be handled/sent (i.e. they're not

functions).

GET is the default method in both Python and HTML for getting information from the server. The data from a GET request is shown in the URL in your browser. For example, `@app.route("/news")` is actually a GET request, just like typing **bbc.co.uk/news** into your browser and hitting enter. Both of these things are saying *"hey server, get me this specific webpage"*.

You can submit a form using GET or POST. If you submit a form using GET, you'll see the submitted form data in your URL bar in your browser.

Sometimes that doesn't matter (if you're searching for a bookshelf on IKEA's website, you might want to bookmark the results page). But there are other times, like if a visitor is submitting personal information through a form, when you need that information to be sent privately.

The POST method is used to send data to the server so that it can be processed. It's like saying *"hey server, here's some data for you to do stuff with"*. In HTML, you can only make a POST request using a form. Information sent using the POST method doesn't show in the URL bar, so it's more private.

You can read more about the differences between GET and POST [here](#). And if you're really interested, Mozilla also has some neat walkthroughs [here](#).

## Task 5

Let's go back to the **hello.html** file you saved in your templates folder and add some more code to it so we can get information from a user. We'll keep it really simple for now, so just ask for their email address.

1. Copy and paste the code below, just before the closing tag in your HTML file:

```
<p>Let's keep in touch!</p>

<div id="contact-form">
    <form method="post" action="/signup">
        <label for="email">Email address:</label>
        <input type="email" id="email" name="email" required="required">
        <input type="submit" value="Submit" id="submit-button">
    </form>
</div>
```

2. We need to specify what happens after the form is submitted, using action in the `<form>` tag. The value for the action attribute is the URL the form will be submitted to, relative to the absolute URL you're at (in our case localhost:5000). This means we need to add a decorator and some code in our **hello.py** file that says what to do when the browser goes to /signup.

3. Copy and paste the code below into your **hello.py** file.

```
@app.route("/signup", methods=["POST"])
def sign_up():
    form_data = request.form
    print (form_data["email"])
    return "All OK"
```

4. There is one more thing we need to do! You might have noticed that we're now accessing a variable called request, and are wondering where it came from. This is actually a part of Flask, so we need to import it to use it. So, the very top of your **hello.py** file needs to be updated to look like this:

```
from flask import Flask, render_template, request
```

5. Ok, now you're ready to run your file! Once you've stopped and restarted your server, head



Remember, when using a form, you need to specify the request type you're using in the HTML form's opening tag, as well as in Python. Notice that in HTML, the GET and POST methods are typed in lowercase, whereas in Python, they're in uppercase.

## Debugging tips

Part of being a software developer is playing detective. When something doesn't go quite the way you expect, look for clues in your command line and your browser window. All the information you see might seem a bit intimidating at first, but you'll soon find it's more friendly than it looks and really helpful!

Python can tell you the line(s) it's having a problem with and even a bit about what the problem is. Sometimes it's as simple as forgetting to put an indent or a quotation mark or a colon somewhere.

If the answer isn't as clear as that, you can do some more sleuthing online to find out more about what the problem actually is and how to solve it. 99% of the time you'll find the answer to your problem through a Google search that takes you to Stack Overflow, W3 Schools or someone's blog. You'd be amazed at how many really experienced developers do the same thing when they need to figure out a problem!

## Homework

1. Review what you've learned so far. If you have any questions, ask your fellow classmates or an instructor.
2. You can use HTML forms to get all sorts of information from visitors to your website, including text, email, numbers, dates and more. You're not limited to just using text fields either – you can use input types like radio buttons, tick boxes, etc.

Add another one or two input fields of your choice to your HTML form, and update your Python code to do something with that information (for example, display it back to your user). If you want to get a bit creative with input fields, have a look at the [W3 Schools HTML form tutorial](#) and the [W3 Schools input type tutorial](#) for inspiration.

## Preparation for next chapter

1. Go to [heroku.com](https://heroku.com) and sign up for a free account. Make sure you follow the instructions that Heroku emails you to validate it.
2. Check if you have **setuptools**, **virtualenv** and **postgres** installed already by opening your command line and typing **pip list**. If you don't see these packages in the list that displays, you can install each of them using pip

## Extra Homework (optional)

1. Add some more code to your Python file so that visiting [localhost:5000/hi/](http://localhost:5000/hi/) returns “Hi” and visiting [localhost:5000/bye/](http://localhost:5000/bye/) returns “Goodbye”.

## Chapter 17 -Deploying Your Code

### Learning outcomes

- Understand the purpose of platforms like Heroku.
- Understand the importance of version control and testing.

### What it means to deploy code

#### Currently, where is your website?

So far, you've been running your website "locally", using your own computer as a server so that your code can display your website in your browser. This means your website is only available on your own machine and the rest of the world can't see it.

#### Deploying your code to a web server

In order to show off your fantastic work to the world, you'll need to "deploy" your code to a web server so that when people type in your website's URL, there's a server somewhere that knows what to do with that request and how to run your code so that people can see your website.

You could use your own computer as a web server, but its processors can only handle so much. For example, if your website had 100,000 visitors a day, you'd need a computer that was powerful enough to handle all those requests coming in (directing people to the right pages, showing them the right images, crunching numbers, executing queries, etc.). You also wouldn't want your website to go offline every time your laptop goes offline! Fortunately, there are services we can use that take care of all these headaches for us. For this course, we're using one called Heroku.

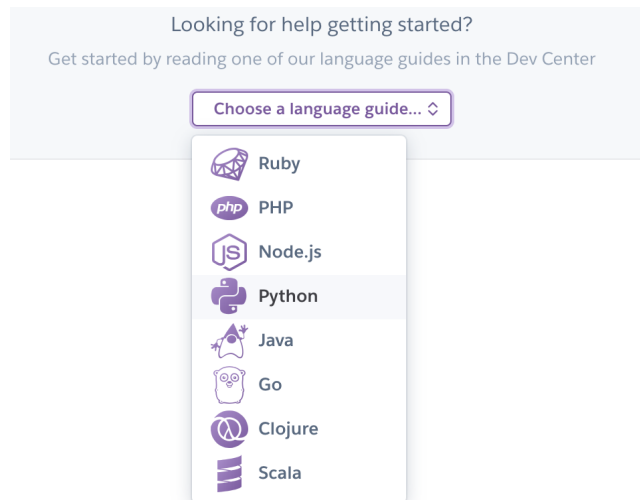
## Cloud-based Platform

Heroku is a cloud-based Platform as a Service (or “PaaS”) that people can use to “deploy” their code online to get their web apps up and running. Unlike your laptop, cloud-based platforms are always on and easily scalable – so if your web app becomes massively popular, you can upgrade your computing power to handle all that extra traffic without having to buy another actual computer.

## Setting up Heroku so it works with the command line

Today we’ll be using the command line, Heroku and some simple git to deploy some test code and get a simple web app “live” on the internet! Before doing this for your own project, we’ll work through Heroku’s demo together. We’re not following the exact setup instructions Heroku gives, so please follow what your instructors say step by step. Don’t get tempted to rush ahead!

1. Log into your [Heroku account](#).
2. Click **Python** from the options that show, then you’ll automatically be taken to another page to get started.



3. You can ignore what the introduction says, because our setup is a little bit different. Simply click the button that says “**I’m ready to start**” and you’ll be taken to a new page.
4. When you see the button below, click the arrow on the right side to select your operating system and download the “**Heroku Command Line Interface**”.

### Windows

Download the appropriate installer for  
your Windows installation:

64-bit installer

Look for the installer in your Downloads folder (or wherever downloaded files usually end up on your computer) then open the installer and follow the instructions. Installing this allows your command line to understand what you mean when you use the word **heroku** in your commands and also makes it possible for you to push your code from your computer to Heroku’s servers (you’ll hear more about this later). If you have a Windows laptop, choose the "Recommended" installation. It might take a little time to install.

Once you’ve installed the Heroku CLI, open your command line and type **heroku login**. You’ll need to use your Heroku account details here.

When you’ve finished the step above, go back to your web browser, scroll down a bit, and click the button saying you’ve installed the Heroku CLI.

I have installed the Heroku CLI

## Getting your Github code ready to work with Heroku

We'll be using [Heroku's demo](#) to see how to use git and Heroku to deploy a simple web app.

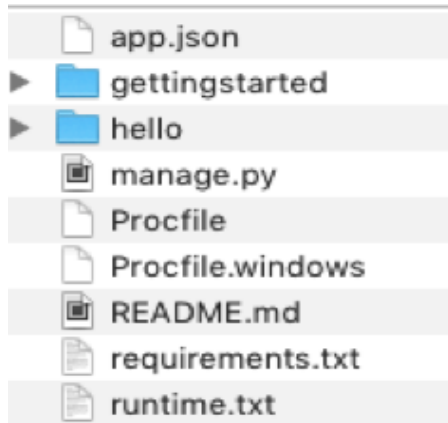
### Clone from git

We'll be "cloning" some files from a GitHub repository that already exists, so before we do that, we want to make sure we know where we're putting these files on our computer. Use the command line to navigate to wherever you've got your Code First Girls Python work. For example (yours will look different, depending on your file path): `cd Documents/Code_First_Girls/Python_Stuff/Heroku_Demo`

1. Now that we know where we'll be putting the Heroku files from GitHub, we'll use `git clone` to create a local copy on our computer. Type this into your command line: `git clone https://github.com/heroku/python-getting-started.git`
2. Now we'll make a folder to put these project files into (and navigate to that folder at the same time) by typing: `cd python-getting-started`

### Heroku folder structure

If you navigate in Explorer (Windows), Finder (Mac) or your command line to the **python-getting-started** folder you just created, you'll see you've now got a copy of the files from GitHub, on your local machine:



## Create and deploy a simple app

You'll need to create and deploy your simple app. You don't have to write any code for this demo, because we've got all the files cloned from GitHub and Heroku already knows what to do with them. In your command line, type: `heroku create`

## Where's my website and how do I update it?

When you create a web app, Heroku assigns you a random name/URL for it.

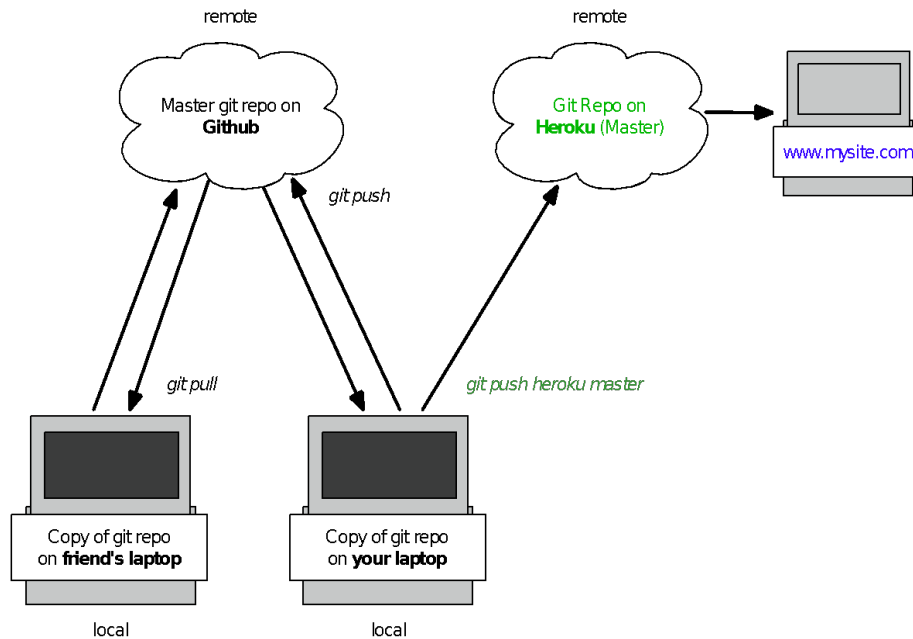
1. Notice the **light blue text** below – copy that URL into your web browser... what do you see?

```
Creating app... done, ● gentle-inlet-97745  
https://gentle-inlet-97745.herokuapp.com/ | https://git.heroku.com/gentle-inlet-97745.git
```

Notice the **green text above** – when you create an app with Heroku, Heroku makes an additional remote git repo and associates it with your local git repository. **It's the repo in green that you push code to in order to make it live.**

## Pushing code from your local git repo to Heroku's git repo

Remember, the only way to update your “live” website files is by pushing code from your local git repo to Heroku's git repo (see the picture below). This means we're not done yet, because we haven't pushed any code to the Heroku Master.



In your command line, type `git push heroku master`. What do you see now, when you navigate to your web app's URL?

### Check running app

Have a read through the text at that command line, it should be something looks like as [here](#).

The application is now deployed, and you can check how many app are running by type below in the command line:

```
heroku ps:scale web=1
```

If you would like to visit the app at the URL, there is a shortcut through command line, you can open the website as follows:

```
heroku open
```

## Getting your own project set up on Heroku

Be sure to have a look through the rest of the [Getting Started documentation](#) to get your project set up. If you see “Django” mentioned, don't worry about that – we're using Flask instead.



Since your website only has one URL, if you working in a team, only one person on your team needs to do the creating and deploying bits on Heroku for your project. However, if you want more than one team member to be able to push code to your live site (your Heroku remote repo), each person needs to be given access privileges for your app. You can set that up by clicking on the Heroku logo on the top left of the page, then clicking on your app, then the **Access** tab and then **Add collaborator**.

## Why it's important to use git for version control

Whether you're working on a project alone, or in a group, it's really important to use git for version control. There are lots of benefits:

- If you've been committing code regularly, you won't lose all your work if something happens to your computer (because you've got a copy of it in your git repository in the cloud).
- Having your work on Github helps you be a part of the coding community (and can even help you get your first job as a developer!) because you can show off your code for others to see and contribute to other people's public projects.
- It helps keep your team organised and working on the right bits, because every person in your team can see a master copy of the code, any code branches you have, as well as commits and comments that might have important information (like, if a commit fixed a bug).

Remember, as shown in the picture earlier in these notes, anything you push to Heroku is "live" (available for the world to see), so it's really, really important that you only push code to Heroku that comes from the Master branch of your project's git repository.

If you've followed best practice and tested your code locally on your laptop before committing it to your team's Master code branch, then your code should work as expected when it's deployed on Heroku, because it's just a copy of the code you've already tested.

## **Self explore Flask with Heroku**

Here is a detailed blog for [flask-deployment-on-heroku](#). Please read through and experiment with flask app with Heroku.

# Appendix

## Python general

Overview of Python programming applications: [here](#), and very simple ways to learn Python [videos](#).

## Good video tutorials

**For part 1 - part 4.**

There is a very good series of Python beginner course videos [here](#).

**Codingbat logic.**

If you take the codingbat practises and get confused, you may want to check the channel [here](#), for while some of the explanations are in Java, the logic will be similar to the Python problems.

**MIT Python courses.**

There is also a great MIT Python course which is open to the public for free. It is a more academic style rather than being taught by developers, but there are good foundation building techniques to let you understand the meanings step by step. You can find all the course videos [here](#), and you can enroll for free, see the course link [here](#).

## More readings

**Basic**

Python official documents: [docs.python.org](https://docs.python.org)

Two python GCSE courses in youtube: [channel 1](#) , [channel 2](#)

Get Programming Learn to code with Python: [/books/get-programming](#) & [in-motion](#)

Introduction to Computation and Programming Using Python: [pdf](#)

## Command Line

1. Linux Command Line and Shell Scripting Bible, 3rd Edition: [safaribooksonline](#).

## Code python online (no need install)

<http://www.pythontutor.com/>

<https://www.pythonanywhere.com>

## More Python practise and learn algorithms

<https://codingbat.com/python>

MIT courseware 600.1-6

Leetcode

Find personal project:

- [brilliant .org](#)
- <https://www.kaggle.com/>
- [Project Euler](#)