Lecture 5.1

Using Modules

Topics 5.1 and 5.2 Intended Learning Outcomes

- By the end of the week you should be able to:
  - Import modules and access the definitions that they contain,
  - Access documentation for modules in the Python Standard Library, and
  - Download and use third party packages from PyPI-Python Package Index, such as Pandas and Matplotlib.

Lecture Overview

1. Importing Modules
2. Example Program: Projectile Flight Time
3. JSON Serialisation

Importing Modules

Modules

- Until now we have relied on **built-in functions** and our **own** code to write programs.
- Python provides a vast collection of additional pre-made definitions (**functions**, **classes**, etc).
  - These definitions are organised into modules.
- To **use a module** we must **import** it.
- We've actually imported a couple of modules already:
  - The **os** module (for **file system** operations).
  - The **datetime** module (for **date** objects).

What is a Module?

Modules

A Module in Python is simply a file that involves a collection of variables, functions and classes. A module can also include run-able code. If we have two files in the same folder we can call them modules.

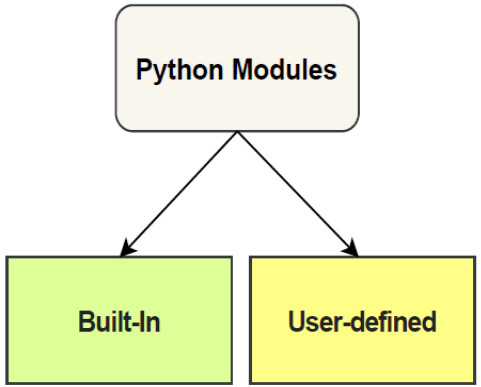
Modules are very helpful in:

- organising large code (variables, functions and classes).
- making existing codes available to re-use and easy to find them.
- implementing shared services or data

Modules

Modules can be classified into **two** types as follows:

1. **Built-In** Modules: available in Python by default.
2. **User-Defined** Modules: defined and written by programmers (ourselves).



Modules

How to use existing Modules?

- We use `import` keyword to execute and access to Modules items (variables, functions and classes).
  - **import moduleName**. moduleName.function().
- We can also use both `import` and `from` keywords.
  - **from moduleName import function**. function().

Module information

Python **help()** function can be used to get the documentation of specified **module**, class, function, variables, etc.

The Python Standard Library

Examples of Python Built-in modules are:

```
>>> help('modules')
IPython             base64              imp                 scrolledlist
PIL                 bdb                 importlib           seaborn
PyQt5               beampy              inspect             search
__future__          binascii            io                  searchbase
__main__            binhex              iomenu              searchengine
_abc                bisect              ipaddress           secrets
_ast                bleach              ipython_genutils    select
_asyncio            browser             itertools           selectors
_bisect             builtins            jedi                setuptools
_blake2             bz2                 joblib              shelve
_bootlocale         cProfile            json                shlex
_bz2                calendar            keras               shutil
_codecs             calltip             keyword             sidebar
_codecs_cn          calltip_w           kiwisolver          signal
_codecs_hk          cgi                 lib2to3             site
_codecs_iso2022     cgitb               linecache           six
_codecs_jp          chardet             locale              sklearn
_codecs_kr          chunk               logging             smtpd
_codecs_tw          cmath               lxml                smtplib
_collections        cmd                 lzma                sndhdr
_collections_abc    code                macosx              socket
_compat_pickle      codecontext         mailbox             socketserver
_compression        codecs              mailcap             sqlite3
_contextvars        codeop              mainmenu            squeezer
_csv                collections         markdown            sre_compile
_ctypes             colorama            marshal             sre_constants
_ctypes_test        colorizer           math                sre_parse
_datetime           colorsys            matplotlib          ssl
_decimal            compileall          matplotlib_inline   stackviewer
_dummy_thread       concurrent          mimetypes           stat
_elementtree        config              mmap                statistics
_functools          config_key          modulefinder        statusbar
_hashlib            configdialog        msilib              storemagic
_heapq              configparser        msvcrt              string
_imp                contextlib          multicall           stringprep
_io                 contextvars         multiprocessing     struct
_json               copy                netrc               subprocess
_locale             copyreg             networkx            sunau
_lsprof             crypt               nntplib             symbol
_lzma               csv                 nt                  sympyprinting
_markupbase         ctypes              ntpath              symtable
```

Module

- Importing a module is simply a way of getting access to the definitions (variables, functions and classes) inside it.

- For example, let's consider the math module.
  - Somewhere on our **computer** there is a file called math.py which contains **mathematical** definitions.

The `math` Module

- The documentation for the math module is available at https://docs.python.org/3/library/math.html.
  - This web page describes all of the functions, constants, and so forth available to us in the math module.

Functions in the `math` Module

- The math module mostly consists of **mathematical functions**, e.g., *log, sine, cosine, sqrt*, ..etc.

```
>>> math.ceil(1.1)  # Round 1.1 up to the nearest integer.
2
>>> math.log(10)  # Take the natural logarithm of 10.
2.302585092994046
```

Constants in the `math` Module

- The math module also contains some useful mathematical **constants** such as *Pi* and *Euler's number*

```
>>> math.pi  # π
3.141592653589793


>>> math.e  # Euler's number
2.718281828459045
```

The `math` Module

- One of the **definitions** in the math module is a **function** for calculating **square roots**, **`sqrt`**.

- So how do we use this function in our **own** program?

Import Statements

- If we try to use the ***sqrt*** function without **importing** it, Python won't *recognise* it.

```
>>> sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined


>>> math.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

Import Statements

- To gain access to the math module and the definitions within, we must use the import keyword to *import* the **math** module.
- A module only has to imported **once** per script file/interpreter session.
- It is good practice to write all import statements at the **top** of your script files.

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.sqrt(36)
6.0
```

- Note that we specify both the module **name** and function **name** when **calling the sqrt function**.

The `random` Module

Example: Import **random** built-in Python module.

Example (random module - range(start, stop, step))

```
>>> import random
>>> print("Random number: ", random.random())
Random number:  0.07300231837766846

>>> print("Random integer is", random.randint(0, 5), end='')
Random integer is 5
>>> print("Random integer is", random.randrange(2, 8, 2), end='')
Random integer is 6
>>> print ("Random float is", random.random() * 100)
Random float is 69.75257284697204
```

Example (random module - import)

```
>>> import random
>>> city_list = ['Melb', 'Sydney', 'ADL', 'BNE', 'Perth']
>>> print("Random city from the list:", random.choice(city_list))
Random city from the list: ADL
>>> print("Random cities from the list:", random.choices(city_list, k=2))
Random cities from the list: ['Melb', 'Sydney']
```

The random Module

Example: Import **random** built-in Python module.

Example (random module - seed)

```
>>> import random
>>> random.seed(1234)
>>> [random.random() for _ in range(5)]
[0.9664535356921388, 0.4407325991753527, 0.007491470058587191, 0.9109759624491242, 0.939268997363764]
>>> random.seed(1234)
>>> [random.random() for _ in range(5)]
[0.9664535356921388, 0.4407325991753527, 0.007491470058587191, 0.9109759624491242, 0.939268997363764]
>>> random.seed(1235)
>>> [random.random() for _ in range(5)]
[0.9085506848193617, 0.4247091824969128, 0.8418417915109482, 0.47000231140228577, 0.1491641026296857]
```

Example (random module - shuffle)

```
>>> import random
>>> mylist =['Melb', 'Sydney', 'ADL', 'BNE', 'Perth']
>>> random.shuffle(mylist)
>>> print(mylist)
['Sydney', 'Perth', 'Melb', 'BNE', 'ADL']

>>> def myfunction():
...     return 0.1
...
>>> random.shuffle(mylist, myfunction)
>>> print(mylist)
['Perth', 'Melb', 'BNE', 'ADL', 'Sydney']
```

The Python Standard Library

- The modules that come with Python make up what is known as the Python Standard Library.
  - The math, datetime, random, and os modules are all part of the Python Standard Library.
  - These modules are available to all Python programs.
- For full details on the Python Standard Library, check out https://docs.python.org/3/library/index.html.

Check Your Understanding

Q. There is a module in the Python Standard Library called shutil which contains a function called rmtree which takes a directory path as its argument. When called, it deletes the directory and its contents.

How would you import and call this function to delete a directory called "old_data"?

Check Your Understanding

Q. There is a module in the Python Standard Library called shutil which contains a function called rmtree which takes a directory path as its argument. When called, it deletes the directory and its contents.

How would you import and call this function to delete a directory called "old_data"?

A.

```
import shutil
shutil.rmtree('old_data')
```

- The first line imports the shutil module so that we can use the definitions it contains.
- The second line calls rmtree from the module, passing 'old_data' as its argument.

Example Program: Projectile Flight Time

Task Definition

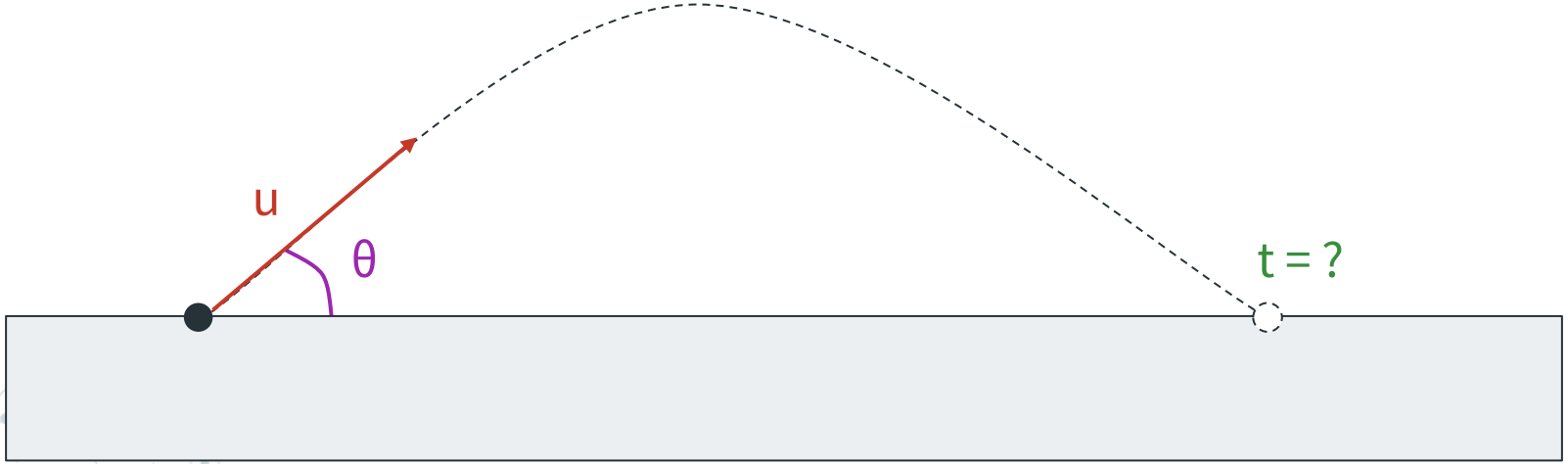
Write a program which, given the launch angle in degrees, θ, and initial speed in metres per second, u, of a projectile, calculates the flight time in seconds, t, according to the following formula:

$$t = \frac{2u\sin(\theta)}{g}$$

Assume that acceleration due to gravity is ***g*** *= 9.81 m/s^2.*

Task Definition

- We are calculating the time that it takes for the launched projectile to hit the ground.



Identifying Inputs and Outputs

- Inputs:
  - Launch angle, θ (in degrees).
  - Initial speed, u (in metres per second).
- Outputs:
  - Flight time, t (in seconds).
- Example:
  - $\theta = 60°$, $u = 9$ m/s → $t = 1.59$ s

Coding the Solution

- You should be comfortable coding the input and output portions of the program.

```
# Input.
theta_deg = float(input('Initial angle (degrees): '))
u = float(input('Initial speed (m/s): '))

# Processing.
# TODO: Calculate `t` here...

# Output.
print(f'Flight time (s): {t:.2f}')
```

Coding the Solution

- All that's left now is to write the Python code implementing the mathematical formula.
- The `sin` function is provided by the math module.
- The rest of the formula can be implemented using **standard** numeric operators.

$$t = \frac{2u\sin(\theta)}{g}$$

First Coding Attempt

- Our first attempt at flight_time.py:

```
import math

# Input.
theta_deg = float(input('Initial angle (degrees): '))
u = float(input('Initial speed (m/s): '))

# Processing.
g = 9.81
t = (2 * u * math.sin(theta_deg)) / g

# Output.
print(f'Flight time (s): {t:.2f}')
```

$$t = \frac{2u\sin(\theta)}{g}$$

First Coding Attempt

- When we somehow get a **negative** flight time!
  - Clearly there is something **wrong**, we were expecting **1.59 s**.

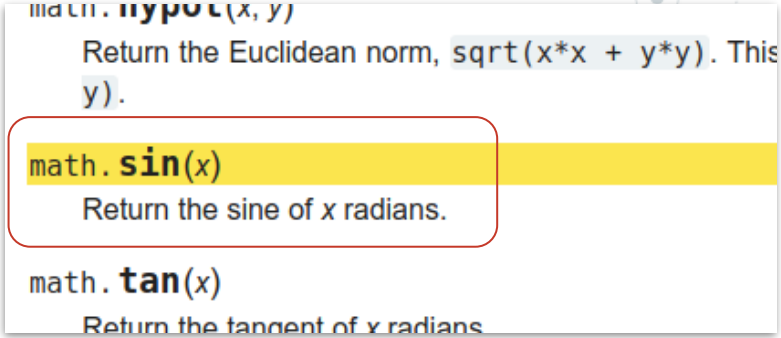
- Since everything else is correct, there must be something wrong with the way that we are using the sin function.

```
$ python flight_time.py
Initial angle (degrees): 60
Initial speed (m/s): 9

Flight time (s): -0.56
```

Fixing the Problem

- If you read the documentation for the **math** module, you will notice that the math.sin function **expects** an **angle** in radians.

- Therefore we must **convert** our angle from degrees to **radians**.
- There is **another function** in the math module which does this: **math.radians**.



Source: https://docs.python.org/3/library/math.html#math.sin

Working Solution

$$t = \frac{2u\sin(\theta)}{g}$$

```
import math

# Input.
theta_deg = float(input('Initial angle (degrees): '))
u = float(input('Initial speed (m/s): '))

# Processing.
g = 9.81

theta_rad = math.radians(theta_deg) #convert degrees to radians

t = (2 * u * math.sin(theta_rad)) / g

# Output.
print(f'Flight time (s): {t:.2f}')
```

Working Solution

- The output is now as expected, hooray!

- This example program **highlights** how important it is to **understand** the functions that you use.
  - Don't just import code and **cross** your fingers---**read** the documentation!

```
$ python flight_time.py
Initial angle (degrees): 60
Initial speed (m/s): 9

Flight time (s): 1.59
```

JSON Serialisation

Serialisation and Deserialisation

- Serialisation is the process of taking an object and converting it into a format that can be stored in a file.
  - Useful for **saving** data.

- Deserialisation is the reverse process.
  - Useful for **loading** data.

- Technically the code from an earlier lecture for reading and writing a list to a text file line-by-line was a rudimentary form of serialisation/deserialisation.

Serialisation and Deserialisation

- Technically the code from an earlier lecture for reading and writing a list to a text file line-by-line was a rudimentary form of serialisation/deserialisation.

- However, **this code only worked with lists of floats**.

```
# File: sort_houses.py
prices = []
for line in open('house_prices.txt'):
    price = float(line)
    prices.append(price)

prices.sort()

out_file = open('sorted_prices.txt', 'w')
for price in prices:
    out_file.write(f'{price:.2f}\n')
out_file.close()
```

The `json` Module

- The `json` module provides functions for serialising and deserialising data according to the JSON (JavaScript Object Notation) format.

- Makes it easy to save data from your program and load it again later.

- JSON is a known format which makes **interchange** of data **between programs easier** (unlike custom-designed formats).

The JSON Format

- A JSON file typically has a **.json** file extension and is a **text** file.
  - Can be viewed and edited in text editors like Notepad and Visual Studio Code.
- Supports a variety of **primitive** data types (integers, floats, booleans, strings) as well as some data structures (dictionaries, lists).
- An **alternative** to other file formats for storing data, like **CSV** (comma separated values) files.

The JSON Format

- JSON files can be easily read/written by humans and computers alike.
- Although the representation used in JSON **resembles** literals in Python, it is **not** Python code (the syntax is different).

Example: car.json

```
{
    "make": "Nissan",
    "model": "Skyline GT-R",
    "colour": "Gunmetal",
    "manual": true,
    "kilometres": 127832,
    "owners": [
        "Rachel",
        "Tommy",
        "Samantha"
    ]
}
```

Writing to a JSON File

- A Python object can be serialised in JSON format and written to a text file using the `dump` function from the **json module**.

- To use this function, **provide** the object to serialise as the **first** argument, and the **file** object to write to as the **second** argument.

Writing to a JSON File

- The program below creates a JSON file called people.json representing the people dict.

```
import json

people = [
    {'name': 'John Doe', 'age': 43, 'is_employee': True},
    {'name': 'Jane Doe', 'age': 44, 'is_employee': False}
]
out_file = open('people.json', 'w')
json.dump(people, out_file)
out_file.close()
```

Writing to a JSON File.

- If we look at the people.json file produced by the program, we can see that it contains all of our data.

- Although the file content resembles Python code, **it is not Python code**.
  - Notice that booleans are in all lowercase, for example.

```
[{"name": "John Doe", "age": 43,
"is_employee": true}, {"name":
"Jane Doe", "age": 44,
"is_employee": false}]
```

people.json

Reading from a JSON File

- A JSON file can be read as a Python object using the `load` function from the json module.

- To use this function, provide the **file** object to read from as its argument.
  - The function will return the **deserialised** data as a Python object.

Reading from a JSON File

```
# File: read_json.py
import json

in_file = open('people.json', 'r')
people = json.load(in_file)
in_file.close()

for person in people:
    print(person)
```

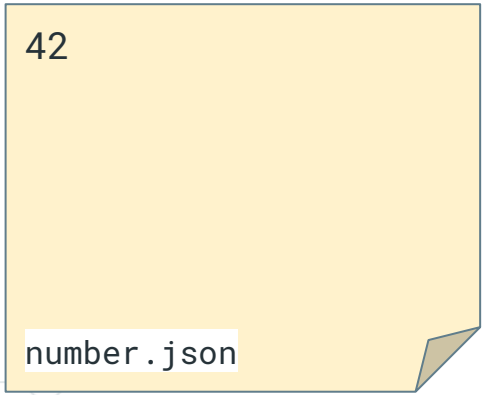
```
$ python read_json.py
{'name': 'John Doe', 'age': 43, 'is_employee': True}
{'name': 'Jane Doe', 'age': 44, 'is_employee': False}
```

Reading from a JSON File

- In the previous example, the object returned by json.load was a **list** (of **dictionaries**).
- The **type** of the object will depend on the **contents** of the JSON file.
- It is perfectly valid for a JSON file to contain a single dictionary, or even just a single number.

Reading from a JSON File

```
>>> import json
>>> x = json.load(open('number.json'))
>>> type(x)
<class 'int'>
>>> x
42
```



- In this case, the type of the object returned by json.load is int.

Limitations of JSON

- The JSON format has some **limitations**:
  - Dictionary **keys** are always stored as **strings**.
  - Other data structures (e.g. sets) and **objects** from other **classes** (e.g. dates, custom classes) **can't** be **directly** serialised as JSON.
- If you need to store an arbitrary object in a JSON file, consider **converting** it to a **dictionary** first.
  - e.g. Date objects could be converted into dicts like `{'year': 2022, 'month': 5, 'day': 3}`.

Check Your Understanding

Q. Wilbert wants to use the contents of a JSON file to define a constant in his Python program. His plan is to copy-paste the JSON into his program and assign it to a variable. Will this work?

Check Your Understanding

Q. Wilbert wants to use the contents of a JSON file to define a constant in his Python program. His plan is to copy-paste the JSON into his program and assign it to a variable. Will this work?

A. In general, no.

- The Python interpreter will try to interpret the copy-pasted JSON as Python code.
- But JSON has different syntax to Python code.
- Wilbert should either use json.load to deserialise the file's contents, or manually rewrite the JSON as Python.

Lecture 5.2

Third-Party Modules

Lecture Overview

1. Installing Packages
2. Data Analysis with Pandas
3. Plotting with Matplotlib

Installing Packages

Packages

- A package is a collection of modules that you can use in your programs.
  - This allows package creators to **group** related functionality into separate modules.
  - Each module must be **imported separately**, even if they are from the same package.

- Anyone can create a package.

PyPI

- The largest repository of third-party Python packages is PyPI (the Python Package Index).
  - https://pypi.org/
- There are packages available for almost any purpose you can imagine!

- Finding a package that does part of what your program needs to do is not "lazy".
  - There is no point in reinventing the wheel when someone has solved a task well already.

That Said...

- This a subject about learning Python!
- **Therefore you are not to use third-party packages in assessment tasks unless specifically instructed to.**
- But in terms of writing your own programs for your own problems---go nuts!
  - I use all kinds of Python packages in my own projects.

Python's Package Installer

- Packages can be installed from PyPI using a program that comes with Python called `pip` - *package installer for Python*

- Generally you will first search for a relevant package using the PyPI website or Google, and then invoke `pip` to install it.

Example: Python's Package Installer

- Say you are writing an application for processing medical imaging data stored in the DICOM format.
- After a bit of googling, you discover a Python package which handles the nitty-gritty of reading DICOM data called "pydicom".
- To install this package, you would run `pip` like so:

```
$ pip install pydicom
```

Python's Package Installer

- To see a list of all Python packages that you have installed, use the "list" subcommand.

```
$ pip list
```

- To uninstall a package from your computer, use the "uninstall" subcommand.

```
$ pip uninstall pydicom
```

Package Documentation

- Unfortunately there is no central source of documentation for all Python packages on PyPI.

- To find documentation, the best approach is to use a general purpose search engine.
  - e.g. google for "pydicom documentation".

Package Documentation

- Well-written documentation for a package will usually include:
  - A user guide,
  - Examples, and
  - Technical descriptions of the modules contained in the package (sometimes called an "API reference").
- This will give you a good idea of which modules you need to import and how to use them.

Data Analysis with Pandas

Pandas

- According to the official Pandas website, Pandas is "a fast, powerful, flexible and easy to use open source data analysis and manipulation tool".

- Pandas allows you to gain the kinds of data insights you might expect from graphical spreadsheet applications like Microsoft Excel.
  - It's much more flexible and powerful though!

Installing and Importing Pandas

- The Pandas package can be installed from PyPI using the following command:

```
$ pip install pandas
```

- Once installed, the main `pandas` module can be imported in your programs like normal.

```
import pandas
```

Example: Melbourne Weather

- Pandas is an extremely comprehensive package and we only have time to scratch the surface.

- We'll be analysing weather data stored in a CSV (comma separated value) file as a demonstration.
  - Real values for Melbourne during May 2020.

Example: Melbourne Weather

```
date,min_temperature,max_temperature,rainfall,max_wind_direction,max_wind_speed,max_w
2020-05-1,7.6,13,8,NW,46,10:02,9,80,,WSW,9,1001.7,12.4,62,,W,17,999.5
2020-05-2,8.3,14.2,1.4,WNW,44,11:18,10.3,81,,W,15,1005.8,13.4,66,,SSW,15,1007.7
2020-05-3,9.8,15,1.8,SSW,30,12:12,11.3,72,,W,6,1023.3,13.8,63,,SSW,6,1024.1
2020-05-4,9.4,14.8,0.2,WSW,19,04:24,10.1,97,,W,6,1029.6,13.8,78,,SSW,7,1028.2
2020-05-5,8,19.5,0.4,N,31,12:52,10.2,100,,NNE,13,1030.5,18.9,51,,NNE,17,1026.7
2020-05-6,10.2,20.1,0,N,48,14:09,15.2,64,,NNW,19,1026.1,19.1,56,,N,26,1021.6
2020-05-7,15.1,21.1,0,NNW,59,11:04,17.9,52,,N,17,1014.8,20.5,51,,N,24,1008.8
2020-05-8,12.6,19.6,0,NNW,41,14:08,14.8,70,,NNW,15,1013.2,18.7,53,,NNW,22,1011.1
2020-05-9,12.6,14,0.6,SW,39,14:23,12.6,81,,W,17,1007.2,12.4,75,,WSW,9,1009.5
2020-05-10,7.2,14.5,7,NW,28,01:53,9.9,65,,NW,7,1023.1,13.6,59,,W,9,1023.5
2020-05-11,7.8,17.1,0.6,NNW,22,10:13,11.1,74,,NNE,9,1028.8,16,57,,N,11,1026.4
2020-05-12,10.3,14.8,0.2,N,41,12:02,12.1,69,,N,17,1024,14.4,66,,N,19,1020.9
2020-05-13,10.4,14.5,1.4,SSW,30,14:09,10.5,90,,WSW,7,1025.2,13.8,54,,SSW,13,1025.3
2020-05-14,6.6,14.7,2.8,SSW,20,14:58,7.9,89,,NNE,9,1030.7,13.6,53,,SSW,11,1029.3
2020-05-15,4.5,15.3,0,SSW,19,14:17,6.7,86,,NE,9,1032.3,14.1,58,,S,7,1029.3
2020-05-16,4.5,18,0,NNW,35,11:17,7.9,87,,NNE,7,1031.2,17.5,46,,N,15,1028.9
2020-05-17,5.1,18.5,0,NNE,19,00:49,7.1,91,,NNE,9,1033.1,16.6,44,,S,7,1030.4
2020-05-18,5.5,19.4,0,N,33,15:04,8.1,87,,NNE,7,1031.6,18.7,51,,N,19,1027.5
2020-05-19,8.1,17.7,0,NNW,33,12:51,13.6,69,,N,13,1024.4,17.5,57,,N,19,1019
2020-05-20,10.3,15.8,15.6,NNW,56,01:27,12.5,83,,NNW,11,1013.1,15.3,43,,NW,17,1012
```

Data source: [Bureau of Meteorology]

Example: Melbourne Weather

- There are data columns describing
  - Temperature,
  - Rainfall,
  - Wind,
  - Cloudiness,
  - Humidity,
  - Pressure,
  - etc.

DataFrames

- The data structure that Pandas uses for representing data is the DataFrame.
- A DataFrame is a table with rows and columns.
- Typically the **columns** represent different **attributes** of the data and the **rows** represent different **items**.
- For our data:
  - The **columns** are date, min_temperature, etc.
  - The **rows** are different days.

Reading from a CSV File

- To load the contents of a CSV file as a DataFrame object, use the `read_csv` function provided by Pandas.

- Pandas will automatically **open** the file, **deserialise** its contents, and **instantiate** a DataFrame object containing all of the data.

```
>>> df = pandas.read_csv('bom_melbourne_weather.csv')
```

Inspecting a DataFrame

- You can grab the first five rows of a DataFrame using the head method to take a sneak peek at the data.

```
>>> df.head()
        date  min_temperature  ...  afternoon_wind_speed  afternoon_pressure
0  2020-05-1              7.6  ...                    17               999.5
1  2020-05-2              8.3  ...                    15              1007.7
2  2020-05-3              9.8  ...                     6              1024.1
3  2020-05-4              9.4  ...                     7              1028.2
4  2020-05-5              8.0  ...                    17              1026.7

[5 rows x 19 columns]
```

Not all columns will be shown in the output

Listing Column Names

- The full list of column names can be accessed using the `columns` DataFrame attribute.

```
>>> list(df.columns)
['date', 'min_temperature', 'max_temperature', 'rainfall',
➥ 'max_wind_direction', 'max_wind_speed', 'max_wind_time',
➥ 'morning_temperature', 'morning_humidity', 'morning_cloudiness',
➥ 'morning_wind_direction', 'morning_wind_speed', 'morning_pressure',
➥ 'afternoon_temperature', 'afternoon_humidity',
➥ 'afternoon_cloudiness', 'afternoon_wind_direction',
➥ 'afternoon_wind_speed', 'afternoon_pressure']
```

Number of Rows

- The len built-in function can be used to count the number of items (rows) in the DataFrame.

```
>>> len(df)
31
```

- So our DataFrame has 31 rows of data (which makes sense since there are 31 days in May).

Filtering Rows

- We can write conditions to use for **filtering rows** in a DataFrame.
- These conditions are written similarly to regular **boolean** expressions.

Example: Filtering Wind Direction

```
>>> northerly_days = df[df['max_wind_direction'] == ‘N’ or 'max_wind_direction'] ==
‘n']]
>>> northerly_days
          date  min_temperature  ...  afternoon_wind_speed  afternoon_pressure
4    2020-05-5              8.0  ...                    17              1026.7
5    2020-05-6             10.2  ...                    26              1021.6
11  2020-05-12             10.3  ...                    19              1020.9
17  2020-05-18              5.5  ...                    19              1027.5
25  2020-05-26              5.4  ...                    15              1021.6
26  2020-05-27              5.9  ...                    17              1018.1
27  2020-05-28             12.1  ...                     9              1021.5
29  2020-05-30              8.1  ...                    28              1015.8
30  2020-05-31             12.4  ...                    15              1009.8

[9 rows x 19 columns]
```

Example: Filtering Wind Direction

```
>>> df['max_wind_direction'].head()
0     NW
1    WNW
2    SSW
3    WSW
4      N
Name: max_wind_direction, dtype: object
```

- Here we can see the value of the max_wind_direction column for the first few rows.

```
>>> (df['max_wind_direction'] == 'N').head()
0    False
1    False
2    False
3    False
4     True
Name: max_wind_direction, dtype: bool
```

- Applying a comparison operator to a DataFrame column will return multiple boolean results (one for each row).

Example: Filtering Wind Direction

```
>>> northerly_days = df[df['max_wind_direction'] == 'N']
```

- The collection of booleans can then be used to **index** the original DataFrame.
  - Returns a new DataFrame **containing** only the rows which correspond to **True** in the condition.

- The end result: northerly_days contains only the **rows** from **df** where the value of column **'max_wind_direction**' is '**N**' (indicating north in this dataset).

Selecting Columns

- We can **create** a new DataFrame from an **existing** one, **keeping only some** of the **columns** from the original.

- Say, for example, that we want to keep only the '**date**' and **'max_wind_speed' columns** from our **northerly wind days**.

Example: Selecting Columns

```
>>> northerly_days[['date', 'max_wind_speed']]
          date  max_wind_speed
4    2020-05-5              31
5    2020-05-6              48
11  2020-05-12              41
17  2020-05-18              33
25  2020-05-26              26
26  2020-05-27              39
27  2020-05-28              20
29  2020-05-30              56
30  2020-05-31              44
```

- We specify a list of column names to keep, in this case ['date', 'max_wind_speed'].

A Moment of Reflection

- So in **only 2 lines** of code we can find the maximum wind speed for all days when the greatest wind gust was northerly.

- Pandas is powerful!

```
>>> northerly_days = df[df['max_wind_direction'] == 'N']
>>> northerly_days[['date', 'max_wind_speed']]
```

Example: Comparing Columns

- You can also **compare** values from two different columns.

- Here's how we would select all days where the morning temperature was higher than the afternoon.

```
>>> df[df['morning_temperature'] > df['afternoon_temperature']]
        date  min_temperature  ...  afternoon_wind_speed  afternoon_pressure
8  2020-05-9             12.6  ...                     9              1009.5

[1 rows x 19 columns]
```

Aggregating Data

- Pandas also makes aggregating data extremely easy.
- We can calculate summary statistics without writing any loops!

Example: Aggregation

Average daily rainfall

```
>>> df['rainfall'].mean()
2.1548387096774193
```

Highest afternoon wind speed

```
>>> df['afternoon_wind_speed'].max()
28
```

Selecting a Row by Aggregating

- What if we want to find the date of the day with highest afternoon wind speed?
- First we find the **index** of the row with maximal afternoon wind speed (the **idxmax** method).
- Then we use that **index** to **fetch** the entire row (the **iloc** method).

```
>>> windy_day_index = df['afternoon_wind_speed'].idxmax()
>>> windy_day = df.iloc[windy_day_index]
>>> windy_day['date']
'2020-05-30'
```

Aggregating Groups

- It's also possible to group the data according to one column's value, and then aggregate per group.

- For example, here's how we would find the average wind speed for each wind direction.

```
>>> df.groupby('max_wind_direction')['max_wind_speed'].mean()
max_wind_direction
ESE    31.000000
N      37.555556
NNE    19.000000
NNW    39.857143
NW     37.000000
…
```

Aggregating Groups

- You can also aggregate multiple columns at once.

- Here's how we would calculate the mean value of each numeric column, grouped by wind direction:

```
>>> df.groupby('max_wind_direction').mean()
                     min_temperature  ...  afternoon_pressure
max_wind_direction                    ...
ESE                         5.500000  ...         1024.100000
N                           8.655556  ...         1020.388889
NNE                         5.100000  ...         1030.400000
NNW                         9.371429  ...         1018.728571
NW                          7.400000  ...         1011.500000
…
```

Adding Columns

- The syntax for adding a column to a DataFrame is similar to adding an item to a **dictionary**.

- You can calculate values for the new column based on **existing** column values.

Example: Monthly Rainfall Percentage

```
>>> total_rainfall = df['rainfall'].sum()
>>> total_rainfall
66.8
>>> df['rainfall_percentage'] = (df['rainfall'] / total_rainfall) * 100
>>> df.head()
        date  min_temperature  ...  afternoon_pressure  rainfall_percentage
0  2020-05-1              7.6  ...               999.5            11.976048
1  2020-05-2              8.3  ...              1007.7             2.095808
2  2020-05-3              9.8  ...              1024.1             2.694611
3  2020-05-4              9.4  ...              1028.2             0.299401
4  2020-05-5              8.0  ...              1026.7             0.598802

[5 rows x 20 columns]
```

- Here we are adding a new column which shows the percentage of the month's rainfall in each row.

Writing to a CSV File

- Suppose that after **processing** we want to save our new DataFrame to a **CSV** file.

- This can be done using the `to_csv` DataFrame method.
  - This method takes the name of the file to write to as its **first** argument.

Example: Writing to a CSV File

```
>>> cold_days = df[df['max_temperature'] < 14]
>>> cold_days_simple = cold_days[['date', 'min_temperature', 'max_temperature']]
>>> cold_days_simple.to_csv('cold_days.csv', index=False)
```

- The `index=False` part tells Pandas not to save the row indices to the file.
  - This is an example of a named argument.

Example: Writing to a CSV File

```
>>> cold_days = df[df['max_temperature'] < 14]
>>> cold_days_simple = cold_days[['date', 'min_temperature', 'max_temperature']]
>>> cold_days_simple.to_csv('cold_days.csv', index=False)
```

```
date,min_temperature,max_temperature
2020-05-1,7.6,13.0
2020-05-21,9.4,13.7
2020-05-22,9.1,13.6
2020-05-23,9.4,13.5
2020-05-24,10.9,13.8
```

cold_days.csv

Plotting with Matplotlib

Matplotlib

- Matplotlib is a Python package for plotting graphs.
- It is useful for visualising data.
- Like Pandas, Matplotlib is an expansive package with lots of functionality.
  - We will cover some useful basics.
  - More information is available at matplotlib.org.

Installing Matplotlib

- The Matplotlib package can be installed from PyPI using the following command:

```
$ pip install matplotlib
```

Importing Matplotlib

- The main module that you will need to import when using Matplotlib is `matplotlib.pyplot`.
  - This is quite a long name to type!

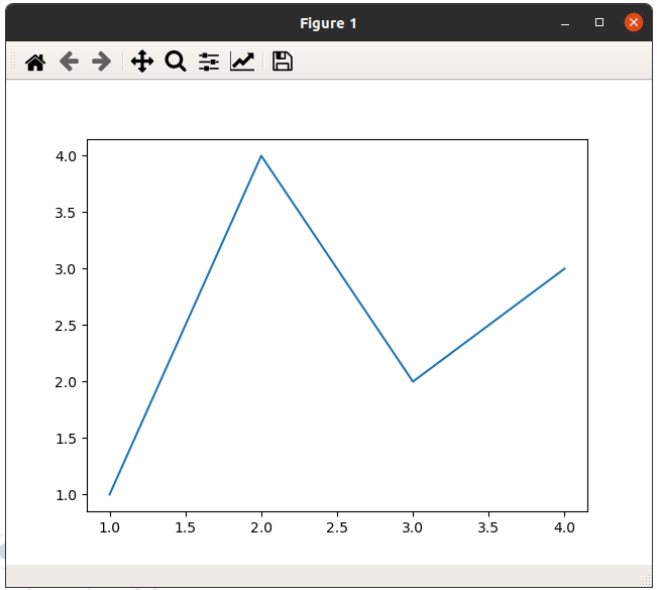
- Fortunately, you can tell Python to give the module a different name in your program by adding an as clause to your **import** statement.

```
>>> import matplotlib.pyplot as plt
```

We now refer to the module as `plt` in our program

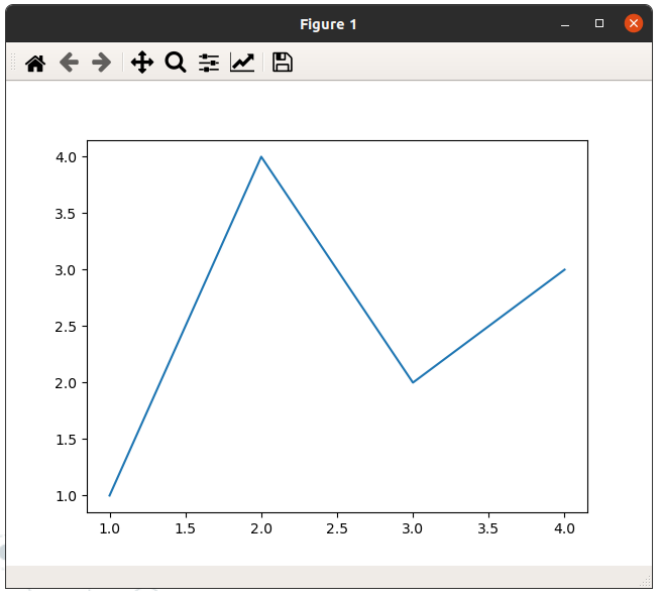
Plotting a Line Graph

```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```



Plotting a Line Graph

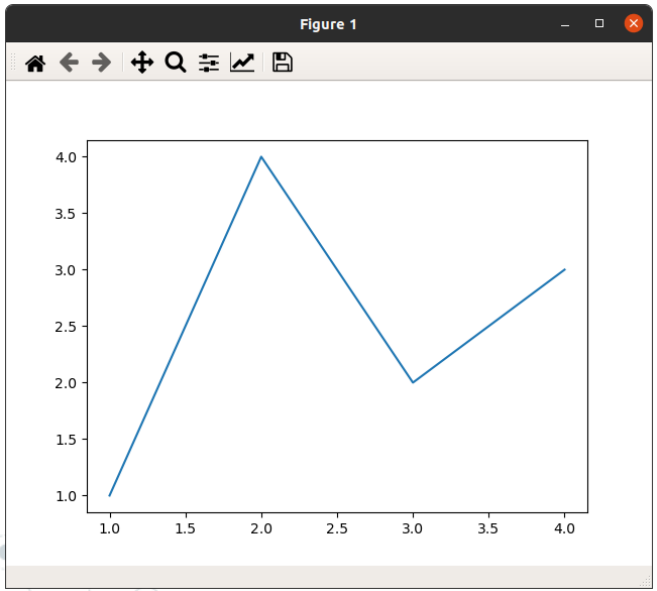
```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```



- The `subplots` function returns two new objects:
  - A new figure (fig).
  - A set of x-y axes in that figure (ax).
- The figure object represents the overall figure, whereas the axes object represents a plot of data.

Plotting a Line Graph

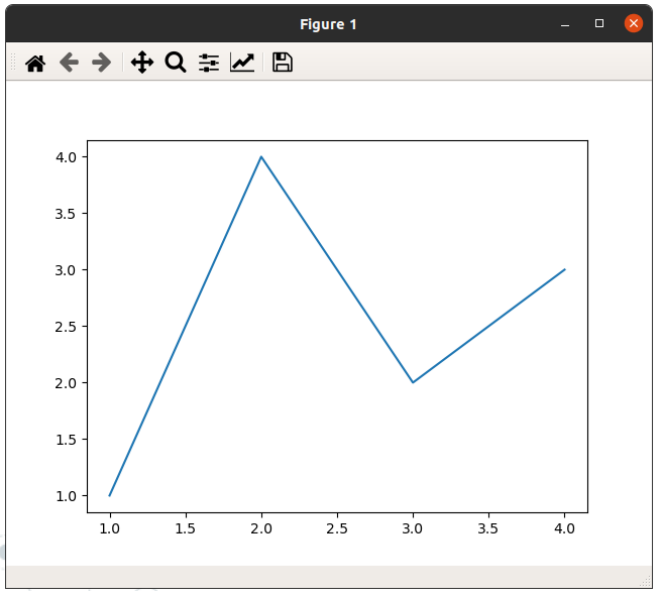
```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```



- The `plot` axes method accepts the x-values and y-values to plot.
- The points will be represented as a line graph.

Plotting a Line Graph

```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```



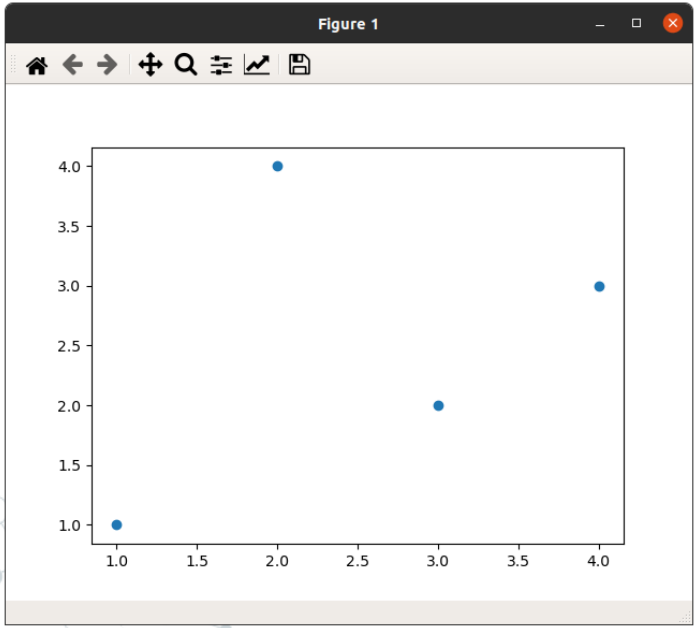
- Finally, the show function displays the figure in a graphical window.
- The program pauses execution until the user closes the window.

Plotting Other Kinds of Graphs

- Other types of graphs can be plotted in a similar way by replacing the `plot` method call with an alternative.
  - `bar` creates a bar graph.
  - `scatter` creates a scatter plot.

Example: Scatter Plot

```
>>> fig, ax = plt.subplots()
>>> ax.scatter([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```



- A scatter plot draws points without lines connecting them.

```
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
>>> plt.show()
```

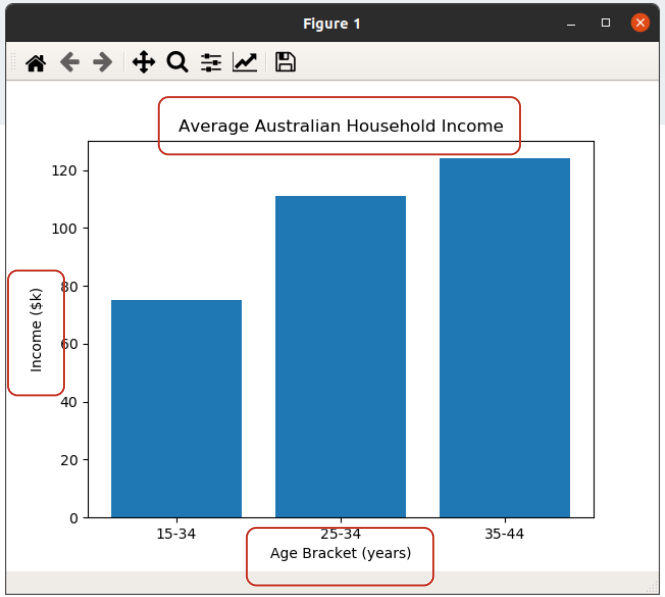
Labelling Axes

- It's good practice to label your graphs.

- Matplotlib provides axes methods to add labels.
  - `set_title` sets the overall title.
  - `set_xlabel` sets the x-axis title.
  - `set_ylabel` sets the y-axis title.

Example: Labelling Axes

```
>>> fig, ax = plt.subplots()
>>> ax.bar(['15-34', '25-34', '35-44'], [75, 111, 124])
>>> ax.set_title('Average Australian Household Income')
>>> ax.set_xlabel('Age Bracket (years)')
>>> ax.set_ylabel('Income ($k)')
>>> plt.show()
```

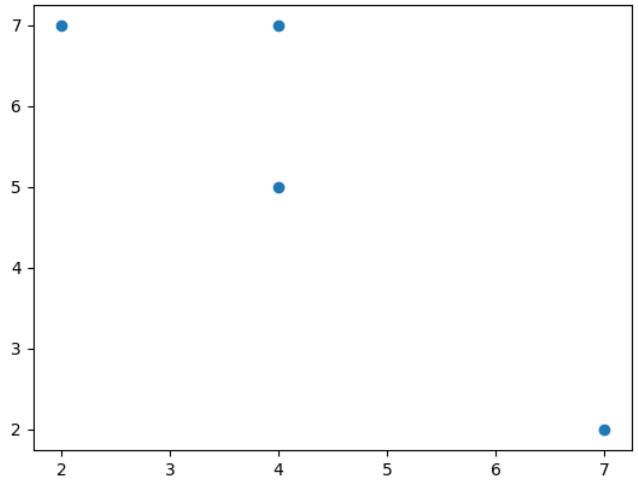


Saving a Figure as an Image

- Instead of pausing the program to show a graph, you can instead save the graph as an image file.
- To do this, use the `savefig` figure method instead of `plt.show()`.
- Supply the desired file path of the output image as an argument.

Example: Saving a Figure as an Image

```
>>> fig, ax = plt.subplots()
>>> ax.scatter([4, 7, 2, 4], [7, 2, 7, 5])
>>> plt.savefig('mygraph.png')
```



- This program will create an image file called "mygraph.png".
- The program will not pause or pop up a window.

Using Matplotlib with Pandas

- **Pandas** and **Matplotlib** are great companions 
  - You can manipulate data with Pandas and then visualise the result.

- Let's look at an example which uses Pandas and Matplotlib to plot the daily Melbourne rainfall.

```
import pandas
import matplotlib.pyplot as plt

# Read the weather data into a DataFrame object.
df = pandas.read_csv('bom_melbourne_weather.csv')
# Create a figure and a set of axes.
fig, ax = plt.subplots()
# Plot dates (x-axis) against rainfall (y-axis) as a bar chart.
ax.bar(df['date'], df['rainfall'])
# Label the axes.
ax.set_title('Daily Melbourne rainfall for May 2020')
ax.set_xlabel('Date')
ax.set_ylabel('Rainfall (mm)')
# Rotate the x-axis labels sideways.
ax.xaxis.set_tick_params(rotation=90)
# Adjust the layout of the figure so that everything fits nicely.
fig.tight_layout()
# Show the figure and pause the program.
plt.show()
```

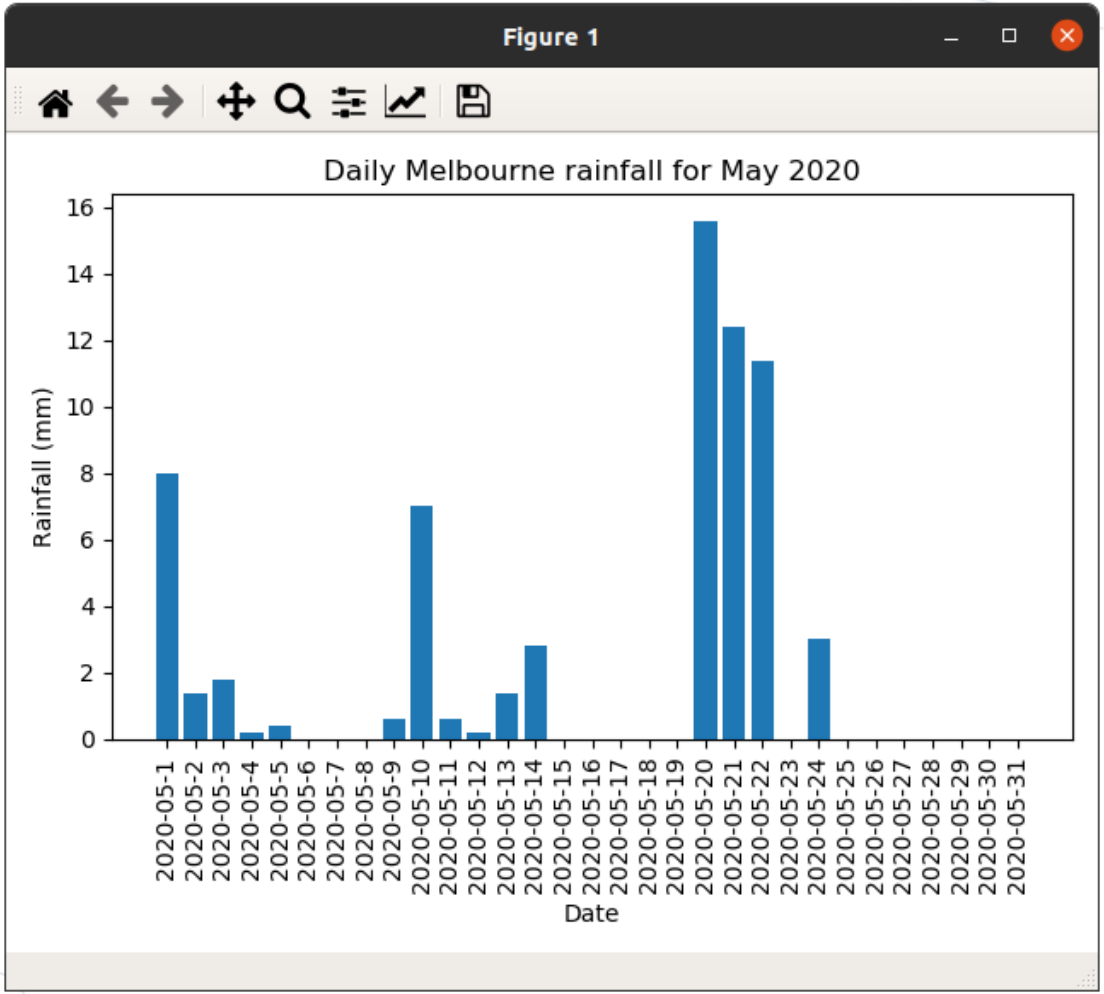


Figure 1
Daily Melbourne rainfall for May 2020
Rainfall (mm)
0
2
4
6
8
10
12
14
16
2020-05-1
2020-05-2
2020-05-3
2020-05-4
2020-05-5
2020-05-6
2020-05-7
2020-05-8
2020-05-9
2020-05-10
2020-05-11
2020-05-12
2020-05-13
2020-05-14
2020-05-15
2020-05-16
2020-05-17
2020-05-18
2020-05-19
2020-05-20
2020-05-21
2020-05-22
2020-05-23
2020-05-24
2020-05-25
2020-05-26
2020-05-27
2020-05-28
2020-05-29
2020-05-30
2020-05-31
Date

Plotting Multiple Data Series

- `plot` and `scatter` can be called more than once on the same axes object.
  - This will plot multiple data series on the same set of axes.
  - Great for creating a visual comparison of data.

Plotting Multiple Data Series

- In the plot/scatter method call, you can use the following named arguments:
  - `color` to specify the colour of a series.
  - `label` to specify the name of a series.

- The `legend` axes method places a legend in the figure (which will display the series names).

Example: Daily Temperature

- Let's look at an example which combines these features.

- We will plot the daily minimum and maximum temperature for Melbourne during May 2020.

```
import pandas
import matplotlib.pyplot as plt

# Read the weather data into a DataFrame object.
df = pandas.read_csv('bom_melbourne_weather.csv')
# Create a figure and a set of axes.
fig, ax = plt.subplots()
# Plot dates (x-axis) against temperature (y-axis) as a line graph.
ax.plot(df['date'], df['min_temperature'], color='blue', label='Minimum')
ax.plot(df['date'], df['max_temperature'], color='red', label='Maximum')
# Label the axes.
ax.set_title('Daily Melbourne Temperature for May 2020')
ax.set_xlabel('Date')
ax.set_ylabel('Temperature (degrees celsius)')
# Show a legend.
ax.legend()
# Rotate the x-axis labels sideways.
ax.xaxis.set_tick_params(rotation=90)
# Adjust the layout of the figure so that everything fits nicely.
fig.tight_layout()
# Show the figure and pause the program.
plt.show()
```

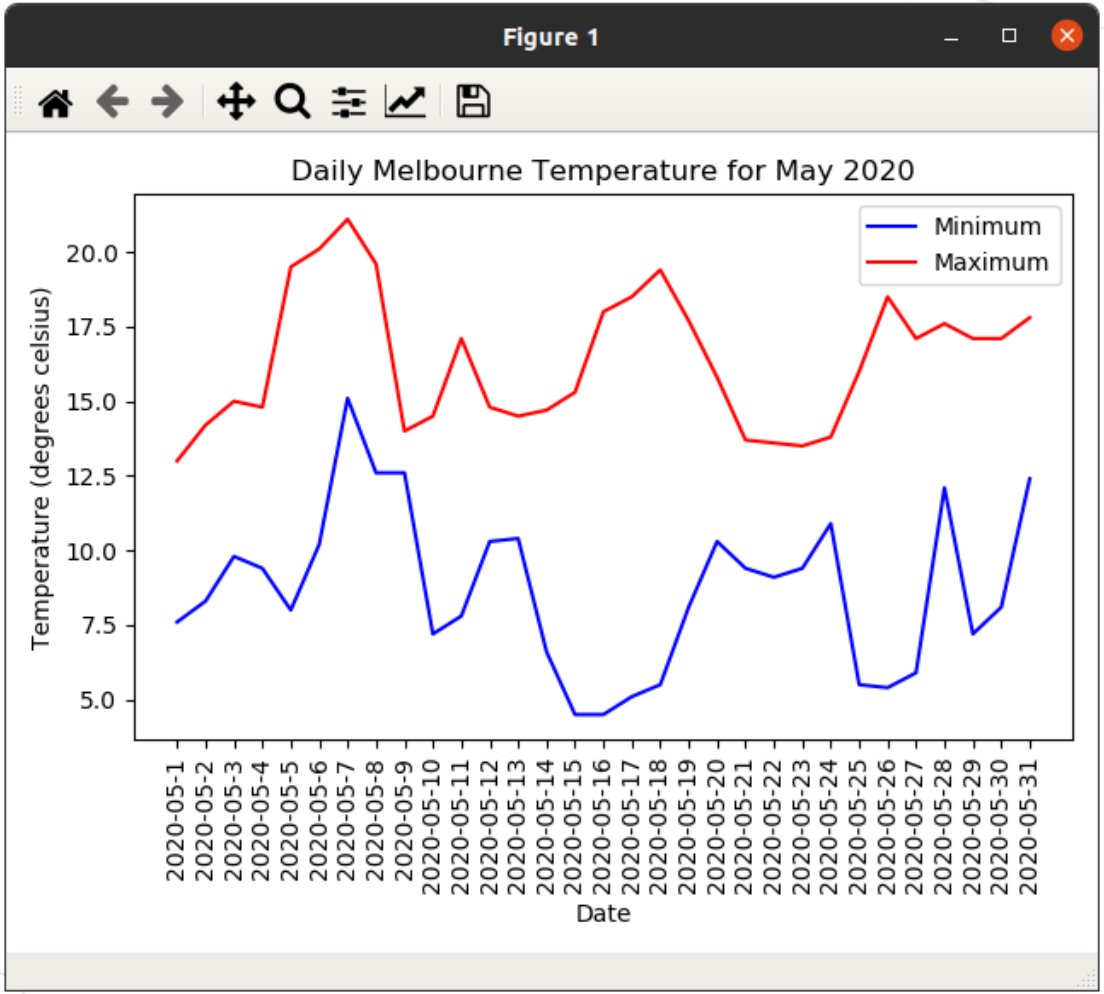


Figure 1
Daily Melbourne Temperature for May 2020
Minimum
Maximum
Temperature (degrees celsius)
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2020-05-1
2020-05-2
2020-05-3
2020-05-4
2020-05-5
2020-05-6
2020-05-7
2020-05-8
2020-05-9
2020-05-10
2020-05-11
2020-05-12
2020-05-13
2020-05-14
2020-05-15
2020-05-16
2020-05-17
2020-05-18
2020-05-19
2020-05-20
2020-05-21
2020-05-22
2020-05-23
2020-05-24
2020-05-25
2020-05-26
2020-05-27
2020-05-28
2020-05-29
2020-05-30
2020-05-31
Date

Lecture 5.3

Structuring Code

Topics 5.3 and 5.4 Intended Learning Outcomes

- By the end of the week you should be able to:
  - Apply modular design techniques when writing programs,
  - Follow good practices for formatting your source code, and
  - Use comments and docstrings to add informative explanations to code that you write.

Lecture Overview

1. Modular Programming
2. Creating Modules
3. Formatting Code

Modular Programming

Modular Programming

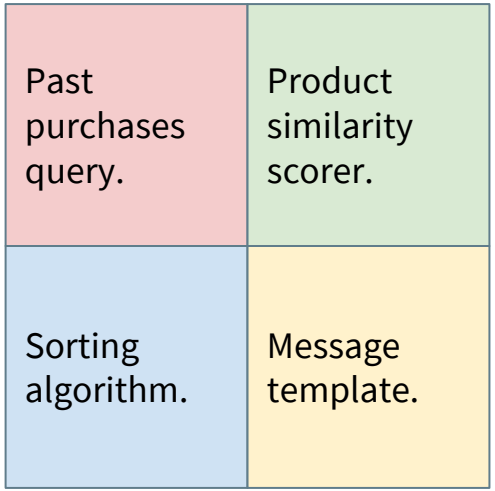
- Modular programming is the practice of writing code which separates functionality into self-contained **chunks**.
- When we talk about modular programming we are **not specifically** talking about **Python modules**.
  - Although modules do play a part in structuring modular Python programs.
- Modularity can be achieved on many **levels**, from writing **reusable** functions to **custom** classes.

Modular Programming

- Think about **dividing** the overall program into **sub-programs**.
- Each **sub-program** is implemented as a **separate chunk** in your code (e.g. **functions**, **classes**, **modules**).
- This separation makes it easier to reason about the program as a whole.
  - Can solve each **sub-problem** separately.
  - Can think about how the solutions to the sub-problems **interact**.

Related product recommender.

vs.



Thinking in terms of modular programming.

Modular Programming

- We've actually already touched upon modular programming when introducing **functions** and **classes**.
  - Recall DRY (don't repeat yourself).

- However, it's worth emphasising as modular programming is vital to writing well-structured programs.

Example Program: Allowed Luggage

Task definition

*Write a program which checks whether a particular combination of two luggage items is allowed on the plane. The user enters the **dimensions** and **weights** of two **rectangular** luggage items. The luggage is allowed if the smaller item (by volume) is no larger than 8000 cm^3, the larger item is no larger than 12000 cm^3, and neither item is heavier than 7 kg.*

Example Program: Allowed Luggage

- First let's consider a solution which **does not** follow modular design practices.
  - This solution has a high amount of repetition.
  - Different parts of the program are entangled, making it harder to modify in the future.
  - Overall the program is difficult to read.

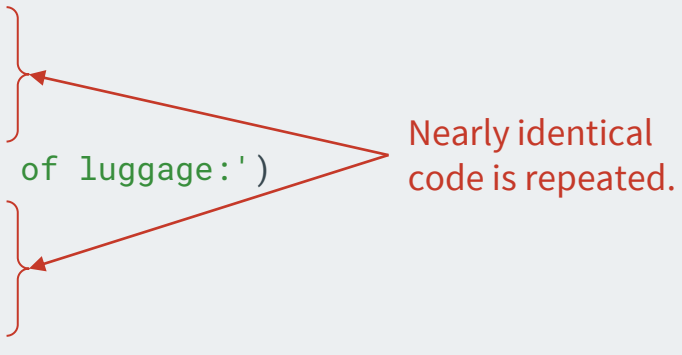
- Let's look at the code for this bad solution. It works, but it's not well-written.

Example: Non-Modular Code (Part 1 of 4)

```
# Can you see how repetitive the code is for accepting user input?
print('Enter details for the first item of luggage:')
width1 = float(input('Width (cm): '))
height1 = float(input('Height (cm): '))
depth1 = float(input('Depth (cm): '))
weight1 = float(input('Weight (kg): ‘))

print('Enter details for the second item of luggage:')
width2 = float(input('Width (cm): '))
height2 = float(input('Height (cm): '))
depth2 = float(input('Depth (cm): '))
weight2 = float(input('Weight (kg): '))
```

Example: Non-Modular Code (Part 1 of 4)

```
# Can you see how repetitive the code is for accepting user input?
print('Enter details for the first item of luggage:')
width1 = float(input('Width (cm): '))
height1 = float(input('Height (cm): '))
depth1 = float(input('Depth (cm): '))
weight1 = float(input('Weight (kg): '))
print('Enter details for the second item of luggage:')
width2 = float(input('Width (cm): '))
height2 = float(input('Height (cm): '))
depth2 = float(input('Depth (cm): '))
weight2 = float(input('Weight (kg): '))
```



Example: Non-Modular Code (Part 2 of 4)

```
# Each luggage item is represented using two variables (one for volume
# and one for weight). Can you see how this makes the code less
# manageable, and could be the source of hard-to-spot errors?
volume1 = width1 * height1 * depth1

volume2 = width2 * height2 * depth2
```

- We've repeated the volume calculation.
- But our problems are just beginning!
  - There are two possibilities---the first luggage item is "small" and the second is "large", or vice versa.

Example: Non-Modular Code (Part 3 of 4)

```
if volume2 > volume1:
    print('Small luggage item:')
    small_allowed = True
    if volume1 > 8000:
        print('Too big!')
        small_allowed = False
    if weight1 > 7:
        print('Too heavy!')
        small_allowed = False
    if small_allowed:
        print('Allowed.')
    print('Large luggage item:')
    large_allowed = True
    if volume2 > 12000:
        print('Too big!')
        large_allowed = False
    if weight2 > 7:
        print('Too heavy!')
        large_allowed = False
    if large_allowed:
        print('Allowed.')
```

Here item 1 is the **small** luggage item.

Example: Non-Modular Code (Part 4 of 4)

```
else:
    print('Small luggage item:')
    small_allowed = True
    if volume2 > 8000:
        print('Too big!')
        small_allowed = False
    if weight2 > 7:
        print('Too heavy!')
        small_allowed = False
    if small_allowed:
        print('Allowed.')
    print('Large luggage item:')
    large_allowed = True
    if volume1 > 12000:
        print('Too big!')
        large_allowed = False
    if weight1 > 7:
        print('Too heavy!')
        large_allowed = False
    if large_allowed:
        print('Allowed.')
```

Here item 1 is the **large** luggage item.

Here's Some Python Code For You

```
non_modular_solution == 'trash'
```

(just kidding)

(kinda)

Taking A Modular Approach

- Let's revisit the problem with a modular programming approach.
- We'll design a solution which is not only correct, but is also easy to read and maintain.

Defining Constants

- Firstly we will define constants for important values.
- This allows us to refer to quantities by name, conveying meaning.
- In the future may want to replace these (e.g. read values from a file).

```
MAX_SMALL_VOLUME = 8000
MAX_SMALL_WEIGHT = 7
MAX_LARGE_VOLUME = 12000
MAX_LARGE_WEIGHT = 7
```

- Now we can refer to `MAX_SMALL_VOLUME` instead of 8000, etc.
- Compare the readability:
  - `volume2 > 8000`
  - `volume2 > MAX_SMALL_VOLUME`

Defining a `LuggageItem` Class

- We can define a class which groups the volume and weight values together into a single object.
- This means that instead of volume1 (float) and weight1 (float), we could just have luggage1 (LuggageItem).

```
class LuggageItem:
    def __init__(self, volume, weight):
        self.volume = volume
        self.weight = weight
```

Adding Logic as a Method

```
class LuggageItem:
    def __init__(self, volume, weight):
        self.volume = volume
        self.weight = weight

    def print_allowed_status(self, max_volume, max_weight):
        allowed = True
        if self.volume > max_volume:
            print('Too big!')
            allowed = False
        if self.weight > max_weight:
            print('Too heavy!')
            allowed = False
        if allowed:
            print('Allowed.')
```

Now all LuggageItem objects know how to print whether they are allowed.

Adding a Reusable User Input Function

- We know that we need to ask the user to input luggage details twice.
- This is a great opportunity to write a reusable function.
- The function takes no arguments, and asks the user to input details.
- The details are used to instantiate a LuggageItem object.

```
def input_rectangular_luggage():
    width = float(input('Width (cm): '))
    height = float(input('Height (cm): '))
    depth = float(input('Depth (cm): '))
    weight = float(input('Weight (kg): '))
    volume = width * height * depth
    return LuggageItem(volume, weight)
```

Putting Everything Together

- Each of the definitions we've made thus far are relatively simple and easy to debug.
  - The sub-problems are easier to solve than the problem as a whole.
  - The overall logic of our program becomes simpler also.

- Let's see how our definitions can be leveraged to solve the problem as a whole.

Putting Everything Together

```
print('Enter details for the first item of luggage:')
luggage1 = input_rectangular_luggage()
print('Enter details for the second item of luggage:')
luggage2 = input_rectangular_luggage()
if luggage2.volume > luggage1.volume:
    small_item = luggage1
    large_item = luggage2
else:
    small_item = luggage2
    large_item = luggage1
print('Small luggage item:')
small_item.print_allowed_status(MAX_SMALL_VOLUME, MAX_SMALL_WEIGHT)
print('Large luggage item:')
large_item.print_allowed_status(MAX_LARGE_VOLUME, MAX_LARGE_WEIGHT)
```

Putting Everything Together

```
print('Enter details for the first item of luggage:')
luggage1 = input_rectangular_luggage()
print('Enter details for the second item of luggage:')
luggage2 = input_rectangular_luggage()
if luggage2.volume > luggage1.volume:
    small_item = luggage1
    large_item = luggage2
else:
    small_item = luggage2
    large_item = luggage1
print('Small luggage item:')
small_item.print_allowed_status(MAX_SMALL_VOLUME, MAX_SMALL_WEIGHT)
print('Large luggage item:')
large_item.print_allowed_status(MAX_LARGE_VOLUME, MAX_LARGE_WEIGHT)
```

Here we are using new variables to indicate which item is the small item.

To Infinity (and Beyond)

- The advantages of modular programming become more prominent as programs grow.

- Let's say that we need to expand our program to allow "deluxe" passengers with a third luggage item.
  - We can reuse our user input function.
  - We can leverage the convenience of the LuggageItem class.

Creating Modules

Writing Modules

- We have seen how to use existing Python modules by importing them to access the definitions within.

- We can **write** our own modules to group related definitions.

Writing Modules

- To create a module, all we need to do is **put** the **definitions** which make up our module together in a **Python source file** (a .py file).
- The module can be imported using its file **name**, **excluding** the .py part.
  - e.g. my_module.py → `import my_module`
- The module **name** must follow variable naming **rules**.
  - e.g. my_module.py is allowed, but $module.py is not.

Example: Rectangle Module

```
# File: rectangle.py
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    def calculate_perimeter(self):
        return 2 * self.width + 2 * self.height

def input_rectangle():
    width = float(input('Width: '))
    height = float(input('Height: '))
    return Rectangle(width, height)
```

Example: Rectangle Module

- Now we have a module for all of our rectangle needs.
- Let's say that we want to create a program for calculating the amount of wood required to build a picture frame.

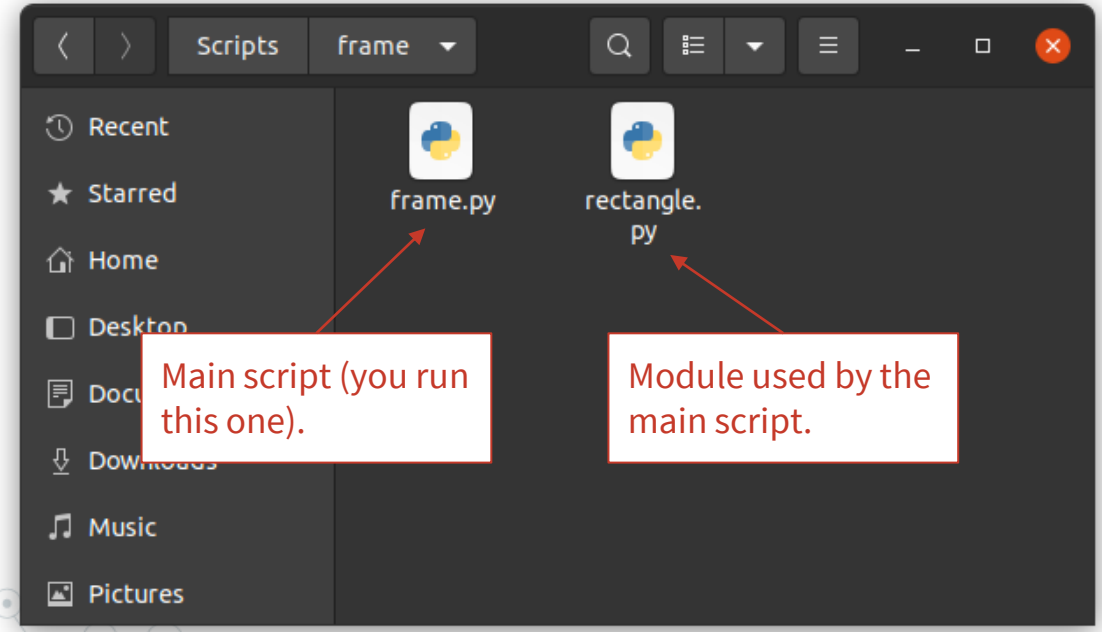
```
# File: frame.py
import rectangle

print('Enter frame dimensions (in cm):')
frame = rectangle.input_rectangle()
wood = frame.calculate_perimeter()
print(f'{wood} cm of wood required.')
```

Modules and File Locations

- Python will only be able to find a module if it is in one of the **locations** that Python is configured to look for modules in.
  - One of those locations is the **directory** containing the script being **run**.

- For this reason, you might find it convenient to place your **main** script and all of the **modules** you write for it **together** in the same **directory**.

Modules and File Locations



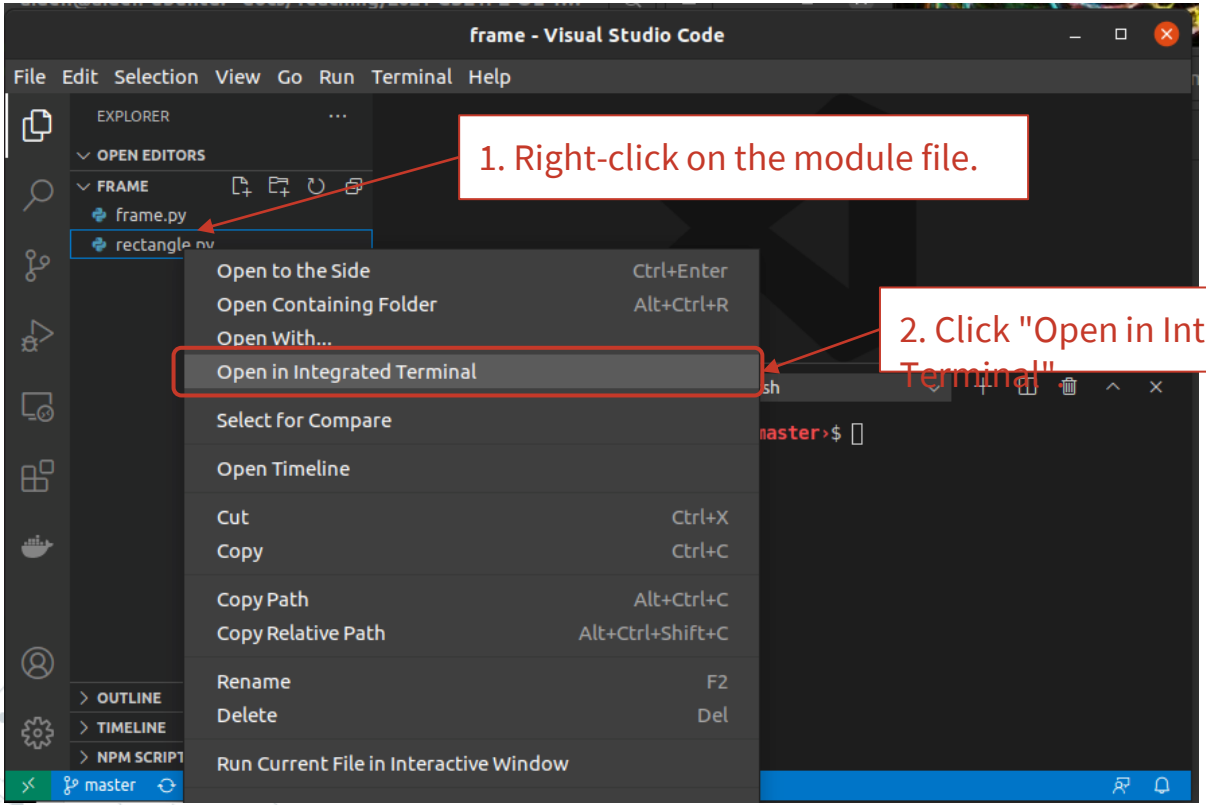
Both Python source code files are placed in the same directory, so Python knows where to look when you import your module.

Importing Your Module from an Interactive Session

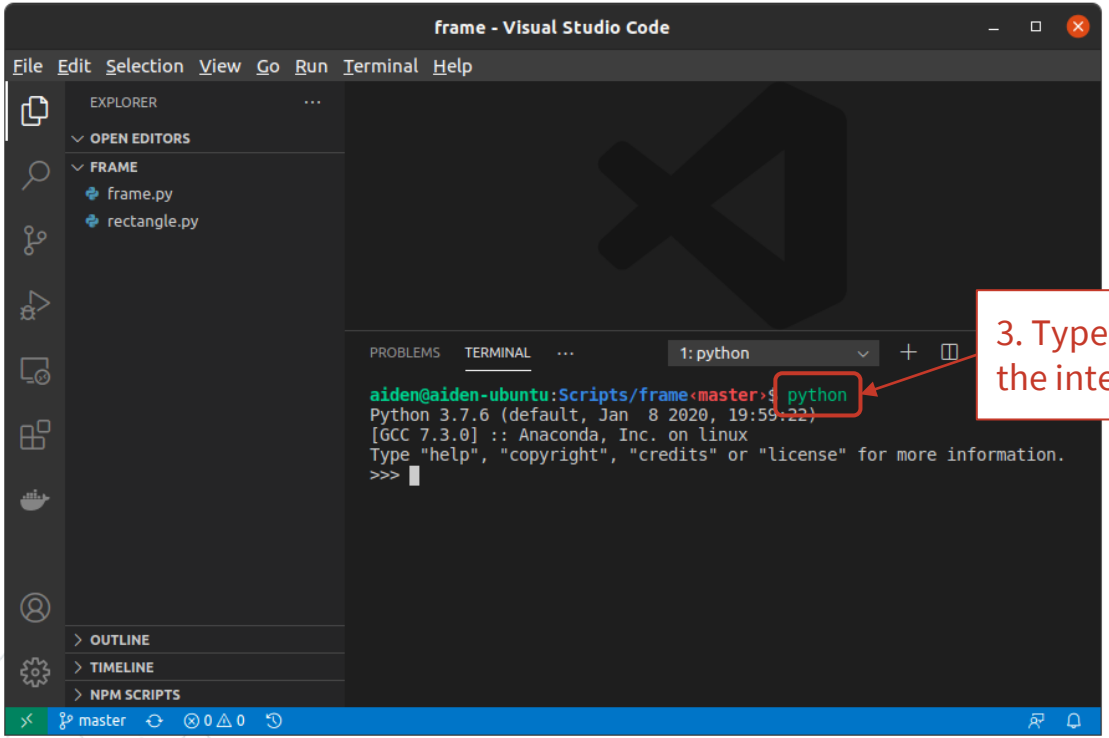
- Similarly, if you want to use a module that you wrote from an interactive interpreter session, you should **start** the interpreter in the **same directory** as your module file.

- You can use the "**Open in Integrated Terminal**" option in Visual Studio Code to do this.

Importing Your Module from an Interactive Session



Importing Your Module from an Interactive Session



3. Type "python" and hit Enter to start the interactive session.

ModuleNotFoundError

- If Python can't find a module to be imported, it will raise a ModuleNotFoundError.

```
>>> import doesntexist
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'doesntexist'
```

ModuleNotFoundError

- If you get a ModuleNotFoundError but feel like you shouldn't, check the following:
  - Make sure that the module **name** matches the file name of the module you are trying to import.
  - Make sure that your module name follows variable naming **rules**.
  - Make sure that the module file is **located** in a place Python can find it (e.g. the same directory).

Code Execution in Imported Modules

- When you import a module, **all of the code inside** is **executed**, and the top-level definitions are made available.

- This means that if, for example, your module contains a **bare** print statement, that print statement will be **executed** when the module is **imported**!

Code Execution in Imported Modules

```
# File: spookymodule.py
print('Boo!')
```

```
>>> import spookymodule
Boo!
>>> import spookymodule
```

- Notice how importing `spookymodule` results in "Boo!" being printed.
- This only happens the first time that the module is imported.
  - Python remembers when you have already imported a module.

Code Execution in Imported Modules

- It is rare that you actually **want** code to execute when your module is **imported**.

- Hence you should **restrict** code in a module to definitions only:
  - **Constants**,
  - **Classes**, and
  - **Functions**.

Check Your Understanding

Q. If you create a module file called "mymodule.py" as below, what code would you write in a separate script file (in the same directory) to display MESSAGE to the user?

```
# File: mymodule.py
MESSAGE = 'Hello from mymodule!'
```

Check Your Understanding

Q. If you create a module file called "mymodule.py" as below, what code would you write in a separate script file (in the same directory) to display MESSAGE to the user?

```
# File: mymodule.py
MESSAGE = 'Hello from mymodule!'
```

A.

```
import mymodule
print(mymodule.MESSAGE)
```

The Main Module

- Every .py file **becomes** a Python **module** when loaded by the Python interpreter.
- This means that scripts which you run using the `python` command are also considered **modules**.
  - The script that you run in this way is a special kind of module called the **main module**.
- A Python program can involve **many** modules, but only **one** can ever be the **main module**.

The Main Module

- Say you have a Python script in my_program.py.
- You run it using the command shown.
- In this case my_program is the main module.
- Any modules imported by my_program are just **regular** modules.

```
$ python my_program.py
```

Example: Cuber

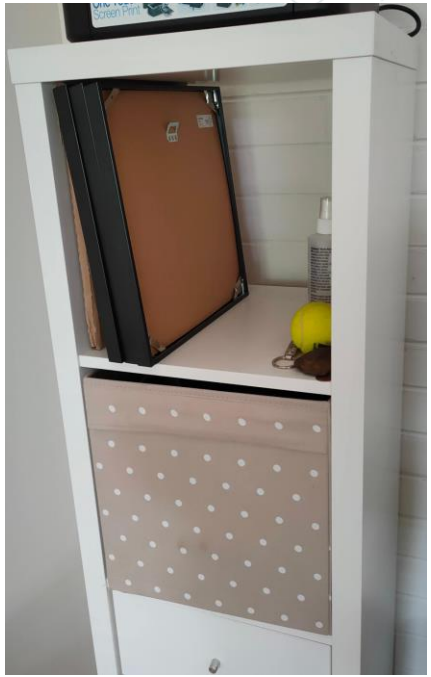
- Let's say that we write a simple program for cubing a number input by the user.
- Note that in this scenario, `cuber` is the main module.

```
# File: cuber.py
def cube(x):
    return x ** 3

x = float(input('Enter a number to cube: '))
print(f'The result is {cube(x)}')
```

```
$ python cuber.py
Enter a number to cube: 5.5
The result is 166.375
```

Example: Cuber

- Now consider the situation where we want to use the **cube function** from another script.
- We can **import** the **cuber module** to gain access to the **cube** function.
- Here's an example program which calculates the total **capacity** of a piece of furniture with multiple **cube-shaped compartments**.



Example: Cuber

```
# File: capacity.py
import cuber

compartments = int(input('Enter the number of compartments: '))
width = float(input('Enter the width of a compartment: '))
volume = compartments * cuber.cube(width)
print(f'The total capacity is {volume:.4f}')
```

- Let's give this program a run to see what happens.

Example: Cuber

```
# File: capacity.py
import cuber

compartments = int(input('Enter the number of compartments: '))
width = float(input('Enter the width of a compartment: '))
volume = compartments * cuber.cube(width)
print(f'The total capacity is {volume:.4f}')
```

```
$ python capacity.py
Enter a number to cube: 3
The result is 27.0
Enter the number of compartments: 5
Enter the width of a compartment: 3
The total capacity is 135.0000
```

Example: Cuber

```
# File: capacity.py
import cuber

compartments = int(input('Enter the number of compartments: '))
width = float(input('Enter the width of a compartment: '))
volume = compartments * cuber.cube(width)
print(f'The total capacity is {volume:.4f}')
```

```
$ python capacity.py
Enter a number to cube: 3
The result is 27.0
Enter the number of compartments: 5
Enter the width of a compartment: 3
The total capacity is 135.0000
```

Importing cuber has caused additional code to execute!

```
# File: cuber.py
def cube(x):
    return x ** 3

x = float(input('Enter a number to cube: '))
print(f'The result is {cube(x)}')
```

Example: Cuber

- In that scenario, `capacity` was the main module.
- There was an undesirable side-effect of code inside cuber running.

- Can we have cuber.py run like a **complete** program when it is the main module, **but skip** the input and print statements when it is **imported**, as in capacity.py?
  - Yes we can!

Main Module Guard

- A main module guard is an **if statement** which checks to see whether the current module is the main module.
  - If a module is **imported**, the statements in the **block associated** with the **main** module guard will **not** be executed.

Example: Cuber

```
# File: cuber.py
def cube(x):
    return x ** 3

if __name__ == '__main__':
    x = float(input('Enter a number to cube: '))
    print(f'The result is {cube(x)}')
```

- The main module guard (highlighted) is satisfied only for the main module.
- This means that the input and print statements are **only** ever **executed** when cuber is the **main module**.

Example: Cuber

```
$ python cuber.py
Enter a number to cube: 5.5
The result is 166.375
```

- The main module is cuber.
- __name__ == '__main__' evaluates to **True** in cuber.py.

```
$ python capacity.py
Enter the number of compartments: 5
Enter the width of a compartment: 3
The total capacity is 135.0000
```

- The main module is capacity.
- __name__ == '__main__' evaluates to **False** in cuber.py.

Using Main Module Guards

- It is good practice to use a main module guard around any code which would only be run when a module is the main module.
  - This is the case even if you do not have immediate plans to import the module elsewhere.

Formatting Code

Formatting Source Code

- There are certain conventions most Python developers follow when formatting code.
- Having conventions makes it easier to **switch** from project to project.
  - The style will be similar, making the code **easier** to read.
- It is possible to have syntactically correct Python code that is **formatted** poorly, and we would like to avoid this.

Example: Poor Formatting

```
dx = float(input(                  'Enter horizontal distance: '))
import      math
def hypotenuse(a, b): return math.sqrt(a** 2+b **2)
dy = float(
input(
'Enter vertical distance: '
))
TOLERANCE = 1.1
distance =hypotenuse(dx, dy) * TOLERANCE
print(f'Distance (with tolerance): {distance  :.2f}' )
```

- Hopefully none of you lost your breakfast looking at this code!

Example: Nice Formatting

```
import math

TOLERANCE = 1.1

def hypotenuse(a, b):
    return math.sqrt(a**2 + b**2)

dx = float(input('Enter horizontal distance: '))
dy = float(input('Enter vertical distance: '))
distance = hypotenuse(dx, dy) * TOLERANCE
print(f'Distance (with tolerance): {distance:.2f}')
```

- You can see all imports and constant definitions at a glance.
- The main processing steps are clearly laid out.

Python Style Guidelines

- As with most things, there is an element of the old adage "beauty is in the eye of the beholder".
- That said, here are some basic **guidelines** which will help you to get started writing **beautiful** code.
- You will also find that most existing Python code **follows** these guidelines.

Guideline #1: Import Statements

- Write all import statements at the **top** of the file.
- This makes it easy to tell at a glance which modules are used in a file.
- It also prevents you from accidentally using a module before it is imported.

Guideline #2: Constant Definitions

- Define all constants **after** any import statements, but **before** other **definitions**.
- This makes it easy to edit constant values if necessary.
- You can also see all constants at a glance.

Guideline #3: Indentation

- Use groups of 4 **spaces** for indentation.
- This takes the thinking out of remaining consistent with indentation.
- Maintains maximum **compatibility** with most other projects (although you will encounter the odd project which uses **different** indentation).

Guideline #4: Blank Lines

- Separate function/class definitions with **blank** lines.
- This makes it easier to see where one definition ends and the next begins.

Guideline #5: Avoid Long Lines

- As a rule of thumb, it is best to avoid lines **longer** than 100 characters.
- The main reason for this is that having to **scroll** horizontally to see everything is annoying!
- There are a few reasons why a line of code might be long, so let's look at a few and how to solve them.

Long Expressions

- If the line is long due to a long expression, you can use a **backslash** (\) to split the line.

- You can only split wherever you could normally put a space character.

```
value = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
```

```
value = 1 + 2 + 3 + 4 + \
  5 + 6 + 7 + 8 + 9
```

Long Strings

- If a line is long due to a long string, you can split the string by closing the quotes, adding a **backslash**, and **opening** the quotes again on the next line.

```
long_message = 'This is a really long message, and it makes the line long'
```

```
long_message = 'This is a really long message, ' \
  'and it makes the line long'
```

Many Function Arguments

- If the line is long due to a function call with many arguments, consider specifying each argument on a separate line.

```
my_func('first argument', 22222, True, 'more argument action', 555.5555)
```

```
my_func(
    'first argument',
    22222,
    True,
    'more argument action',
    555.5555,
)
```

Lecture 5.4

Documenting Code

Lecture Overview

1. Effective Commenting
2. Docstrings
3. README Files

Effective Commenting

The Importance of Comments

- Writing clear, self-explanatory code a very good practice.
- However, sometimes code is complex and there's nothing that you can do about it!
- For these situations, comments are invaluable for documenting how that section of code works.
  - You can explain, in plain language, what the code does.

Who Are Comments For?

- Comments are useful for:
  - Other **programmers** that you are working with.
  - A **future** you (you might understand how the code works now, but you might not remember in a few months).

- A few **minutes** spent writing **comments** now can save **hours** of frustration later down the line.

Comment Everything?

- So comments are important, but be careful!
- Bad comments can:
  - Introduce needless clutter.
  - Be downright misleading.
- Only include comments when they help in understanding the code.

Source: The meme graveyard

Example: Bad Comments

```
avg_weight = total_weight / n_items
# Check whether the temperature is high.
if avg_weight < 500:
    # Add one to the variable x
    x = x + 1
# Calculate average weight
# Old McDonald had a farm, ee-ai-ee-ai-oh!
```

That's not what the code does!

Thanks Captain Obvious!

Comment is in the wrong location.

???

- These comments aren't helping anyone.
- So how do we avoid writing bad comments?

Guideline #1: Avoid Restating Code

- Work under the assumption that the reader understands the fundamentals of programming.
  - i.e. imagine that they have passed this subject.
- Restating what code is obviously doing has no benefit, and just adds clutter.
- Instead, focus on the logic of the code.
  - What does the code mean in the context of your particular application?

Guideline #1: Avoid Restating Code

Bad:

```
# Assign True to `ready` if 2 goes
# into `p` with no remainder, False
# otherwise.
ready = p % 2 == 0
```

- Adds no additional information.

Better:

```
# The game can be started when
# there's an even number of players.
ready = p % 2 == 0
```

- Indicates the purpose of the code.

Guideline #2: Position Comments Appropriately

- Comments should be written near the code that they refer to.
- If the comment is on its **own line**, it is assumed that it refers to the **code** that comes **immediately after** it.
- If the comment is on the **same** line as code, it is assumed that it **refers** to that **line of code**.
- Writing comments that are **disconnected** from the code being explained is confusing.

Guideline #2: Position Comments Appropriately

Bad:

```
a = (w * h) * 1.2

if a > 100:
    print('Item is too large to fit')
# Calculate the area with 20% tolerance.
```

- The comment is disconnected from the code.

Better:

```
# Calculate the area with 20% tolerance.
a = (w * h) * 1.2

if a > 100:
    print('Item is too large to fit')
```

- The comment refers to the following line.

Guideline #3: Keep Comments Updated

- If you change the behaviour of code, make sure that you update the comments too!
- Outdated comments do more **harm** than good, since they can be very misleading.
- You might want to keep an **explanation** of how the code used to work if you think that would be useful.

Guideline #3: Keep Comments Updated

Bad:

```
# A goal is worth 6 points.
score = score + 9
```

- Is the code correct, or the comment? This is confusing.

Better:

```
# A goal is worth 9 points.
# Previously they were worth
# 6 points but the rules
# changed in 2019.
score = score + 9
```

- Here the comment provides information relevant to people familiar with the older code.

The Golden Rule of Commenting

- **Will adding the comment clarify the purpose of the code?**
  - If the answer to the above question is "yes", include the comment.
- Don't get overly stressed about whether the comment is too long, or whether you have enough comments, or even if a comment is strictly necessary.
- Use your discretion and strive for readable code.

Check Your Understanding

Q. Is the following an example of a good or bad comment to include in a program's source code?

```
a = b ** 2  # Square the value of `b` and assign it to `a`.
```

Check Your Understanding

Q. Is the following an example of a good or bad comment to include in a program's source code?

A. A bad comment.

The positioning is correct and it's up-to-date, but it just restates the code. Either omit the comment, or use it to explain *why* b is being squared.

```
a = b ** 2  # Square the value of `b` and assign it to `a`.
```

Docstrings

What is a Docstring?

- A docstring is "a string literal that occurs as the **first statement** in a module, function, class, or method definition" (Python Enhancement Proposal 257).

- A docstring provides a **description** of the definition it is attached to.
  - Written in plain human language.
  - Written by programmers for programmers, just like **comments**.

Writing Docstrings

- It is conventional to write docstrings using triple double quotes, `"""like this"""`, since they often span multiple lines.
- Here's an example of a function with a docstring:

```
def hypotenuse(a, b):
    """Calculate the hypotenuse of a right-angled triangle."""
    return math.sqrt(a ** 2, b ** 2)
```

This fellow here is the docstring.

Python Tracks Docstrings

- The **advantage** of using a **docstring** instead of **comments** is that Python **keeps track of them**.
- This means that **tools** can **access** docstrings to better help you as a programmer.
- You can **access** the docstring on any Python object using the `__doc__` **attribute**.

```
>>> print(hypotenuse.__doc__)
Calculate the hypotenuse of a right-angled triangle.
```

The `help` function

- You can access useful information about a module, function, class, or method definition using the `help` built-in function.
  - Especially useful in interactive sessions.
- The information shown by `help` includes the docstring and shows details of the definition (e.g. parameter names).
- Press "Q" on your keyboard to **q**uit the help. Q

The help function

```
>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in
module math:

sqrt(x, /)
    Return the square root of x.
```

Don't worry too much about this forward slash, it means that the parameters before it can't be specified using keyword arguments (so sqrt(25) works but sqrt(x=25) doesn't).

- In this example, the programmer who wrote the sqrt function included a docstring:
  - "Return the square root of x."
- As you can see, the docstring is shown as part of the help function output.

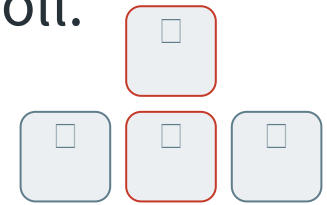
The `help` function

- Naturally, the help function also works for our own functions and docstrings.

```
>>> help(hypotenuse)
Help on function hypotenuse in module __main__:

hypotenuse(a, b)
    Calculate the hypotenuse of a right-angled triangle.
```

Getting Help for a Module

- The `help` function can also be applied to an entire module.
- This is useful for finding out which definitions are contained in a module.
- The output can be quite long.
  - Use the up and down cursor keys on your keyboard to scroll.



```
>>> help(math)
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.7/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)

...
```

README Files

Documentation Outside Python Files

- Larger programs consist of dozens of Python source code files.
- Comments and docstrings are written inside Python source code files.
- It is unreasonable to expect that a user of your software will read all of the source code to figure out what it does and how to run it.
- Hence there is a need to provide some kind of overview outside of Python source files.

README Files

- It is conventional to include a README file in a project directory (not just in Python but for all software).
- The README is simply a text file which introduces and explains the software contained in the directory.
- Typically a README file will be called `README`, `README.txt`, `README.md` or something similar.
- README files are written in plain English (not in Python code).

README Files

- If you want others to use your software, or help develop it, include a README!
- When you give someone your software, typically the first thing that they will do is look for a README.
  - And you should be looking for READMEs in unfamiliar projects too!
- There are no hard and fast rules for what a README should contain but there are some common elements.

Common README Elements

- The name and purpose of the software.
- How to setup the software.
  - Inform the reader of which 3rd party packages are required.
- How to run the software.
  - Indicate which .py file to run when there are multiple modules.
- Authorship, copyright, and licensing details.
  - Give credit to yourself and anyone else who helped create the software.

Writing a README

- Since a README is simply a form of written communication from the human developer of a piece of software to another human being who has received the software, there are no precise rules to follow.
- You can include any information that you think might be useful.
- Avoid using programs like Microsoft Word to create the README---plain text files are much more accessible.

Writing a README

- Start by creating a text file (e.g. `README.txt`).
  - You could use Visual Studio Code for this.
- It's a good idea to begin the README with the name of your software and a short description of what it does.
- Afterwards, you can proceed to explain how to setup and run the software.

README.txt

```
# TaxPro5000

TaxPro5000 is a program for calculating income tax.

Written with love by John Doe, for anyone to use free of charge!

## Setting Up

TaxPro5000 requires the following 3rd party Python packages, which can be
installed using pip:

* pandas
* matplotlib

## Usage

To calculate income tax, run `calc_tax.py` and enter your pre-tax income when
prompted.

To plot your historical tax payments on a graph, run `graph_tax.py` and enter
the file name containing past tax payments when prompted.
```

In-The-Wild Examples

- Matplotlib's README.rst
- Pandas' README.md

Next Lecture We Will...

- Learn about Algorithm Design Strategies.

Thanks for your attention!

The slides and lecture recording will be made available on LMS.

PPT Acknowledgement: Dr Aiden Nibali, CS&IT LTU.