Programming Level-up

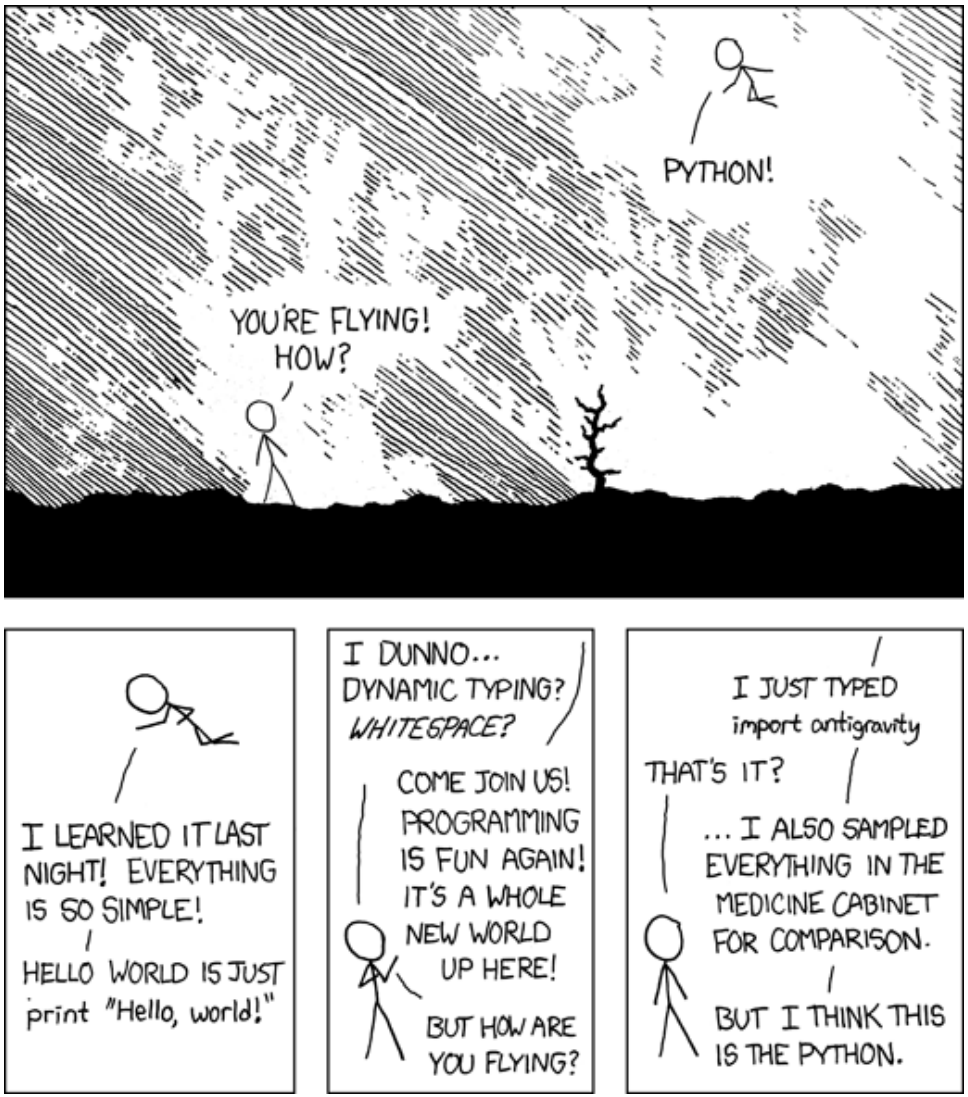
Lecture 3 - Modules & Development Environments

Jay Morgan

20th September 2022

Outline

Importing in python



https://xkcd.com/353/

The basic structure of importing

Modules or packages are other *scripts or programs* that can be imported into other scripts. This definition is very general, but we shall see how flexible importing in Python can be.

The basic syntax of importing is:

```
1 import <package_name>
2
3 <package_name>.<function/class/variable/etc>
```

If we import <package_name> using this syntax, we always have to use the dot . syntax to refer to something within this package.

The basic structure of importing

Let's take a look at a very basic example.

```
4  import math
5
6  radius = 6.4  # cm
7  circum = 2 * math.pi * radius
```

In this example, we are importing the built-in `math` package. This package contains a bunch of useful functions and variables. We're not going to take a look at them here, as we're focusing on importing, but you can see we're referring to a variable called `pi` to calculate the circumference of a circle.

Importing specific items

If we didn't always want to specify the package name when we only want to use something specific from a package, we can directly import that something.

```
8  from <package_name> import <function/class/variable/etc>
9
10 <function/class/variable/etc>
```

As you can see, we're using the `from ... import ...` syntax.

```
11 from math import pi
12
13 circumference = 2 * pi * radius
```

Don't do this!

When using `from ... import ...`, there is a wildcard `*` that we could use. You may sometimes see this style of importing when looking at documentation online:

```
14 from <package_name> import *
15
16 <function/class/variable/etc>
```

However, this can create many problems with reading your program code

Don't do this!

Which module does `my_function()` originate? Are there are common names between the two? Which would be used?

```
17 from my_module import *
18 from my_second_module import *
19
20 my_function()
```

Alias

When importing, we can optionally create an alias to a symbol. Here we're creating an alias to the existing `pi` in `math`.

```
21 from math import pi as decilious_pi
22
23 circumference = 2 * delicious_pi * radius
```

There are some very common conventions of aliasing very highly used packages that we will definitely revisit in another lecture!

```
24 import numpy as np
25 import pandas as pd
26 import matplotlib.pyplot as plt
```

Importing local libraries

let's consider a hypothetical local directory:

```
27 main.py
28 src/
29  |-- my_module.py
30  |-- module_1/
31         |-- cats.py
32         |-- dogs.py
```

If we wanted to import something from `my_module.py` we would do:

```
33 from src.my_module import MyAwesomeClass
34
35 my_class = MyAwesomeclass()
```

Importing local libraries


```
main.py
src/
 |-- my_module.py
 |-- module_1/
        |-- cats.py
        |-- dogs.py
```

Here is another example for increased nesting of directories:

```
from src.module_1 import cats
from src.module_1.dogs import Dog

cat = cats.Cat()
dog = Dog()
```

Quick exercise – imports

- Create a directory to store your scripts
- In this directory, create a file called `main.py`.
- Create a sub-directory called `src`. In `src` create another file called `library.py`.
- In `library.py` create a class (that doesn't do anything right now) called `Database`.
- In `main.py`, create an instance of `Database`.

Shortcuts with __init__.py

Let's say you often import Cat and Dog. We can use a file called __init__.py to help us and make the imports shorter. This fill gets executed when its module is imported.

```
main.py
src/
 |-- my_module.py
 |-- module_1/
        |-- __init__.py
        |-- cats.py
        |-- dogs.py
```

In __init__.py:

```
from cats import Cat
from dogs import Dog
```

In main.py:

What is __main__?

Consider a file with the following:

```
57  x = 2
58  y = 1
59  z = x + y
60
61  class MyAwesomeClass:
62      ...
```

If we import this file in another script, `x`, `y`, and `z` will be computed. In this very simple case this will have very little impact. But what if the computation of these takes a very long time?

What is __main__?

Here we are wrapping any global computations into a appropriate functions. This prevents the global variables being computed as soon as the script is imported.

Now, if we wanted to compute x, y, and z if this script is run, we could use:

```
63 if __name__ == "__main__":
64     # do something
```

Anything within the scope of the `if` function will only be run if the current file is the script that is being run directly (i.e. `python <the-file>.py`). If the script is being imported, the statements within this if scope will not be run.

What is __main__?

So if we wanted to run `compute()` if this file is being run directly, we would write:

```
65  def compute():
66      x = 2
67      y = 1
68      z = x + y
69
70  class MyAwesomeClass:
71      ...
72
73  if __name__ == "__main__":
74      compute()
75      # we can of course use MyAwesomeClass as well
76      my_class = MyAwesomeClass()
77      my_class.do_something()
```

Current working directory

The folder in which you run Python will be the *current working directory (CWD)*. We can print this value with the `os.getcwd()` function, or change the directory with `os.chdir(...)`. Its important to know what your CWD is as all relative paths (paths that do not start with a '/') will be relative to your CWD.

```
78  import os
79
80  print(os.getcwd())
81  os.chdir("../")
82  print(os.getcwd())
83  os.chdir("week-3")
```

```
Results:
# => [...]/Programming Level-up/week-3
# => [...]/Programming Level-up
```

I've replaced the full path printed by Python with `[...]` so you can see the differences in the paths!

Listing directories

Continuing with our usage of the `os` package, we can use the `listdir` function to list all files within a directory.

```
4 print(os.listdir())
5 print(os.listdir("images/"))
```

```
Results:
# => ['images', '__pycache__', 'lecture.pdf', 'lecture.tex', 'data', 'test_file_1.
# => ['legend-2.png', 'fig-size.png', 'basic.png', 'subplots.png', 'python.png', '
```

This returns a list of files and directory relative to your current working directory. Notice how from this list you cannot tell if something is a file or directory (though the filename does provide some hint).

Testing for files or directories

In the previous example we saw that the items returned by `listdir` does not specify if the item is a file or directory. However, `os` provides an `isfile` function in the `path` submodule to test if the argument is a file, else it will be a directory.

```
4  for path in os.listdir():
5      print(f"{path} => is file: {os.path.isfile(path)}")
```

```
Results:
# => images => is file: False
# => __pycache__ => is file: False
# => lecture.pdf => is file: True
# => lecture.tex => is file: True
# => data => is file: False
# => test_file_1.py => is file: True
# => lecture.org => is file: True
# => _minted-lecture => is file: False
# => test_file_2.py => is file: True
```

Using wildcards

If we wanted to get all files within a directory, we could use the `glob` function from the `glob` package. `glob` allows us to use the `*` wildcard. E.g. `*.png` will list all files that end with `.png`. `test-*` will list all files that start with `test-*`.

```
11  from glob import glob
12
13  for fn in glob("images/*"):
14      print(fn)
```

```
Results:
# => images/legend-2.png
# => images/fig-size.png
# => images/basic.png
# => images/subplots.png
# => images/python.png
# => images/pycharm01.png
# => images/installing-scikit-learn.png
# => images/pycharm02.png
# => images/PyCharm_Icon.png
# => images/axis.png
# => images/legend.png
# => images/complex-pycharm.jpg
```

Pathlib – a newer way

`pathlib` is a somewhat recent addition to the Python standard library which makes working with files a little easier. Firstly, we can create a `Path` object, allowing us to concatenate paths with the `/`. Instead of using the `glob` module, a `Path` object has a `glob` class method.

```
14 from pathlib import Path
15
16 data_dir = Path("data")
17 processed_data = data_dir / "processed"
18
19 data_files = processed_data.glob("*.txt")
20
21 for data_file in data_files:
22     print(data_file)
```

```
Results:
# => data/processed/data-2.txt
# => data/processed/data.txt
```

Pathlib – convenient functions

`pathlib` allows us to easily decompose a path into different components. Take for example getting the filename of a path with `.name`.

```
4  from pathlib import Path
5
6  some_file = Path("data/processed/data.txt")
7
8  print(some_file.parts)  # get component parts
9  print(some_file.parents[0])  # list of parent dirs
10 print(some_file.name)    # only filename
11 print(some_file.suffix) # extension
```

```
Results:
# => ('data', 'processed', 'data.txt')
# => data/processed
# => data.txt
# => .txt
```

Converting Path into a string

As `pathlib` is a recent addition to Python, some functions/classes are expecting a `str` representation of the path, not a `Path` object. Therefore, you may want to use the `str` function to convert a `Path` object to a string.

```
6 str(Path("data/"))
```

```
Results:
# => 'data'
```

Quick exercise – locating files

- In the same directory of scripts you created in the last exercise, create another directory called `data`.
- In data, create 3 text files, calling them `<book_name>.txt`.
- These each text file should contain the information from table below in the format:

```
Name: <book_name>
Author: <author>
Release Year: <release_year>
```

Title	Author	Release Date
Moby Dick	Herman Melville	1851
A Study in Scarlet	Sir Arthur Conan Doyle	1887
Frankenstein	Mary Shelley	1818
Hitchhikers Guide to the Galaxy	Douglas Adams	1979

- From `main.py`, print out all of the text files in the directory.

Reading files

To read a file, we must first open it with the `open` function. This returns a file stream to which we can call the `read()` class method.

You should always make sure to call the `close()` class method on this stream to close the file.

`read()` reads the entire contents of the file and places it into a string.

```
4  open_file = open(str(Path("data") / "processed" / "data.txt"))
5  contents_of_file = open_file.read()
6  open_file.close()  # should always happen!
7  print(contents_of_file)
```

```
Results:
# => this is some data
# => on another line
```

Reading files – lines or entire file?

While `read` works for the last example, you may want to read files in different ways. Luckily there are a number of methods you could use.

```
4  open_file.read()        # read entire file
5  open_file.readline()    # read a single line
6  open_file.readline(5)   # read 5 lines
7  open_file.readlines()   # returns all lines as a list
8
9  for line in open_file:  # read one line at a time
10     do_something(line)
```

Reading files

It can be a pain to remember to use the `.close()` every time you open a file. In Python, we can use `open()` as a context with the `with` keyword. This context will handle the closing of the file as soon as the scope is exited.

The syntax for opening a file is as follows:

```
11 with open("data/processed/data.txt", "r") as open_file:
12     contents = open_file.read()
13
14 # the file is automatically closed at this point
15
16 print(contents)
```

```
Results:
# => this is some data
# => on another line
```

Writing files

The syntax for writing a file is similar to reading a file. The main difference is the use `"w"` instead of `"r"` in the second argument of `open`. Also, instead of `read()`, we use `write()`.

```
4  data = ["this is some data", "on another line", "with another
   ↪  line"]
5  new_filename = "data/processed/new-data.txt"
6
7  with open(new_filename, "w") as open_file:
8      for line in data:
9          open_file.write(line + "\n")
10
11 with open(new_filename, "r") as open_file:
12     new_contents = open_file.read()
13
14 print(new_contents)
```

```
Results:
# => this is some data
# => on another line
# => with another line
```

Appending to files

Every time we write to a file, the entire contents is deleted and replaced. If we want to just append to the file instead, we use "a".

```
5  data = ["this is some appended data"]
6  new_filename = "data/processed/new-data.txt"
7
8  with open(new_filename, "a") as open_file:
9      for line in data:
10         open_file.write(line + "\n")
11
12 with open(new_filename, "r") as open_file:
13     new_contents = open_file.read()
14
15 print(new_contents)
```

Results:

```
# => this is some data
# => on another line
# => with another line
# => this is some appended data
```

Quick exercise – reading/writing files

- Using the same text files from the previous exercise, we will want to be able to read each text file, and parse the information contained in the file.
- The output of reading each of the text files should be a list of dictionaries, like we have seen in previous lectures.
- We will go through a sample solution together once you've had the chance to try it for yourself.

Reading CSV files – builtin

When working with common file types, Python has built-in modules to make the process a little easier. Take, for example, reading and writing a CSV file. Here we are importing the csv module and in the context of reading the file, we are creating a CSV reader object. When reading, every line of the CSV file is returned as a list, thus an entire CSV file is a list of lists.

```
6  import csv  # built-in library
7
8  data_path = "data/processed/data.csv"
9
10 # read a csv
11 with open(data_path, "r") as csv_file:
12     csv_reader = csv.reader(csv_file, delimiter=",")
13     for line in csv_reader:
14         print(line)
```

```
Results:
# => ['name', 'id', 'age']
# => ['jane', '01', '35']
# => ['james', '02', '50']
```

Writing a CSV file – builtin

Writing a CSV file is similar except we are creating a CSV writer object, and are using `writerow` instead.

```
5  # write a csv file
6  new_data_file = "data/processed/new-data.csv"
7  new_data = [["name", "age", "height"], ["jane", "35", "6"]]
8
9  with open(new_data_file, "w") as csv_file:
10     csv_writer = csv.writer(csv_file, delimiter=",")
11     for row in new_data:
12         csv_writer.writerow(row)
```

Quick exercise – reading/writing CSV files

- Given the parsed data from the previous exercise, write a new CSV file in the `data` directory.
- This CSV file should contain the headings: name, author, $\text{release}_{\text{data}}$.
- The data in the CSV file should be the 3 books with data in the correct columns.
- Test that you can read this same CSV file in python.

Read JSON files – builtin

Like CSV, json is a common format for storing data. Python includes a package called `json` that enables us to read/write to json files with ease.

Let's first tackle the process of reading:

```
13  import json
14
15  json_file_path = "data/processed/data.json"
16
17  # read a json file
18  with open(json_file_path, "r") as json_file:
19      data = json.load(json_file)
20      print(data)
21      print(data.keys())
22      print(data["names"])
```

Results:

```
# => {'names': ['jane', 'james'], 'ages': [35, 50]}
# => dict_keys(['names', 'ages'])
# => ['jane', 'james']
```

Write JSON files – builtin

While we used `json.load` to read the file, we use `json.dump` to write the data to a json file.

```
5  new_data = {"names": ["someone-new"], "ages": ["NA"]}
6
7  # write a json file
8  with open("data/processed/new-data.json", "w") as json_file:
9      json.dump(new_data, json_file)
10
11 with open("data/processed/new-data.json", "r") as json_file:
12     print(json.load(json_file))
```

```
Results:
# => {'names': ['someone-new'], 'ages': ['NA']}
```

Introduction

When working on projects, we may want to use external packages that other people have written. There are tools in Python to install these packages. However, we may want to use specific versions, again these tools help us to manage these dependencies between different packages and these versions of packages.

Virtual Environments

When installing packages, by default, the packages are going to be installed into the system-level Python. This can be a problem, for example, if you're working on multiple projects that require different versions of packages.

Virtual environments are 'containerised' versions of Python that can be created for each different project you're working on.

We will take a look at package management and virtual environments in Python.

What is Anaconda?

- Distribution of Python and R designed for scientific computing.
- We're going to focus on `Conda`, a package manager in the Anaconda ecosystem.
- Helps with package management and deployment.
- Create virtual environments to install packages to avoid conflicts with other projects

Installing Anaconda

We're going to install miniconda (a minimal installation of anaconda). `https://docs.conda.io/en/latest/miniconda.html`

The steps to install Miniconda are roughly:

- Download Miniconda3 Linux 64-bit
- Save the file to the disk
- Open up a terminal and run the following commands:

```
3 chmod +x <miniconda-file>.sh
4 ./<miniconda-file>.sh
```

Follow the installation instructions (most of the time the defaults are sensible).

Working with Anaconda – creating an environment

Conda is a command line tool to manage environments. We're going to highlight some of the most used commands. But for the full list of management, you can use the instructions at: `https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html`

If you're creating a brand new environment, use:

```
5 conda create --name <name-of-env>
```

This will prompt you to confirm you want to create a new environment, whereupon you enter either a `y` or `n`. If `y` your new environment will be created, but start using the environment, you will first have to activate it.

Working with Anaconda – activating an environment

Once you've created a new environment, you can activate it. This is as simple as:

```
6 conda activate <name-of-env>
```

You will notice that your command line prompt has changed from `(base)` to `(<name-of-env>)`. And whenever you start a new terminal it will always be `(base)`.

Working with Anaconda – de-activating an environment

To deactivate an environment, just use:

```
7 conda deactivate
```

or:

```
8 conda activate base
```

Working with Anaconda – installing using conda

Let's say we want to install a package, say `scikit-learn` (if we're doing some data processing or machine learning). To install this package in conda, use:

```
9 conda install scikit-learn
```

Conda will then check what packages are needed for `scikit-learn` to work, and figure out if anything needs to be upgraded/downgraded to match the required dependencies of other packages.

When Conda has finalised what packages need to change, it will tell you these changes and ask to confirm. If everything seems okay type `y`, and enter.

`scikit-learn` is a package in the anaconda repository. For a list of packages, you can use: `https://anaconda.org/anaconda/repo`

Working with Anaconda – package versions


```
10 conda install <package-name>=<version-number>
```

Installing a specific version of Python

If we wanted to, we could also change the python version being used in the virtual environment.

```
11 conda install python=3.9
```

This will try to install Python version 3.9 providing that the packages you already have installed support it.

Working with Anaconda – conda-forge and other repositories

Let's say that the package is not within the basic anaconda repository. You can specify another repository or channel using the `-c` flag.

```
12 conda install -c <channel> <package>
```

For example, PyTorch (`https://pytorch.org/`) uses their own channel:

```
13 conda install -c pytorch pytorch
```

Working with Anaconda – exporting an environment

We will want to share our research and work with others. To allow others to use the exact same packages and especially the versions of packages we're using, we want to export a snapshot of our environment. Conda includes an export command to do just this:

```
14  conda env export --no-builds > environment.yml
```

Here we exporting our currently activated environment to a file called `environment.yml` (common convention) file. I am using the `--no-builds` flag to improve compatibility with other operating systems such as Mac OS.

Working with Anaconda – creating environment from existing

To create an environment from an existing environment.yml file, you can use the following command:

```
15 conda env create -f environment.yml
```

This will create an environment with the same name and install the same versions of the packages.

Deleting an Environment

At later points in our project life-cycle – we have finished our project and we don't want to have the environment installed anymore (besides we already have the `environment.yml` to recreate it from if we need to!).

We can remove an environment using:

```
16 conda env remove --name <name-of-env>
```

This will remove the environment from Anaconda.

Cleaning up

If you use Anaconda for a long time, you may start to see that a lot of memory is being used, this is because for every version of the package you install, a download of that package is cached to disk. Having these caches can make reinstalling these packages quicker as you won't need to download the package again. But if you're running out of hard drive space, cleaning up these cached downloads is an instant space saver:

```
17 conda clean --all
```

This command will clean up the cache files for all environments, but doesn't necessarily affect what's already installed in the environments – so nothing should be *broken* by running this command.

What is Pip?

Pip is another package installer for python. If you're reading documentation online about how to install a certain Python package, the documentation will normally refer to pip.

Pip, like conda, uses a package repository to locate packages. For pip it is called Pypi (`https://pypi.org`)

We're going to take a look at the most commonly used commands with pip.

Installing packages with pip

If you want to install a package, its as simple as `pip install`.

```
18 pip install <package-name>
```

Installing specific versions

Sometimes, though, you will want to install a specific package version. For this use '==<version-number>' after the name of the package.

```
19 pip install <package-name>==<version-number>
```

Upgrade packages with pip

If you want upgrade/install the package to the latest version, use the `--upgrade` flag.

```
20 pip install <package-name> --upgrade
```

Export requirements file

Like exporting with conda, pip also includes a method to capture the currently installed environment. In pip, this is called `freeze`.

The common convention is to call the file `requirements.txt`.

```
21 pip freeze > requirements.txt
```

Installing multiple packages from a requirements file

If we want to recreate the environment, we can install multiple packages with specific versions from a requirements file with:

```
22 pip install -f requirements.txt
```

Anaconda handles both conda and pip

Conda encompasses pip, which means that when you create a virtual environment with conda, it can also include pip. So I would recommend using conda to create the virtual environment and to install packages when you can. But if the package is only available via pip, then it will be okay to install it using pip as well. When you export the environment with conda, it will specify what is installed with pip and what is installed via conda.

```
23 conda env create -f environment.yml
```

When the environment is re-created with conda, it will install the packages from the correct places, whether that is conda or pip.

PyCharm

So far we have been using a very basic text editor. This editor is only providing us with *syntax highlighting* (the colouring of keywords, etc) and helping with indentation.

PyCharm is not a text editor. PyCharm is an Integrated Development Environment (IDE). An IDE is a fully fledged environment for programming in a specific programming language and offers a suite of features that makes programming in a particular language (Python in this case), a lot easier.

Some of the features of an IDE are typically:

- Debugging support with breakpoints and variable inspection.
- Prompts and auto-completion with documentation support.
- Build tools to run and test programs in various configurations.

We will use PyCharm for the rest of this course.

PyCharm – installing

Using Ubuntu snaps:

```
24 snap install pycharm-community --classic
```

Or we can download an archive with the executable. The steps to run goes something like:

```
25 tar xvf pycharm-community-<version>.tar.gz
26 bash pycharm-community-<version>/bin/pycharm.sh
```

PyCharm – using PyCharm

We shall take a look at the following:

- Creating a new project.
- Specifying the conda environment.
- Creating build/run instructions.
- Adding new files/folders.
- Debugging with breakpoints.

What is a Jupyter notebook?

Jupyter notebooks are environments where code is split into cells, where each cell can be executed independently and immediate results can be inspected.

Notebooks can be very useful for data science projects and exploratory work where the process cannot be clearly defined (and therefore cannot be immediately programmed).

Installing Jupyter

We first need to install Jupyter. In you conda environment type:

```
27 conda install jupyter
28 # or pip install jupyter
```

Starting the server

With Jupyter installed, we can now start the notebook server using:

```
29 jupyter notebook
```

A new browser window will appear. This is the Jupyter interface.

If you want to stop the server, press Ctrl+c in the terminal window.

Using the interface

We shall take a look at the following:

- Creating a new notebook
- Different cell types
- Executing code cells
- Markdown cells
- Exporting to a different format
- How the notebook gets stored

Markdown 101

We will revisit markdown in a later lecture, but since we're using notebooks, some of the cells can be of a type markdown. In these cells, we can style the text using markdown syntax.

`https://www.markdownguide.org/basic-syntax/`

A slightly better environment – jupyterlab

The notebook environment is fine, but there exists another package called jupyter-lab that enhances the environment to include a separate file browser, etc.

```
30 conda install jupyterlab -c conda-forge
31
32 jupyter-lab
```

A sense of style

Now that we have looked at syntax you will need to create Python projects, I want to take a minute to talk about the style of writing Python code.

This style can help you create projects that can be maintained and understood by others but also yourself.

Python itself also advocates for an adherence to a particular style of writing Python code with the PEP8 style guide: `https://www.python.org/dev/peps/pep-0008/`. Though, I will talk through some of the most important ones, in my opinion.

Meaningful names

What does this code do?

```
33  def f(l):
34      x = 0
35      y = 0
36      for i in l:
37          x += i
38          y += 1
39      return x / y
40
41  a = range(100)
42  r = f(a)
```

Meaningful names

What about this one?

```
43  def compute_average(list_of_data):
44      sum = 0
45      num_elements = 0
46      for element in list_of_data:
47          sum += element
48          num_elements += 1
49      return sum / num_elements
50
51  dataset = range(100)
52  average_value = compute_average(dataset)
```

They are both the same code, but the second version is a lot more readable and understandable because we have used meaningful names for things!

Use builtins where possible

Don't re-invent the wheel. Try to use Python's built-in functions/classes if they exist, they will normally be quicker and more accurate than what you could make in Python itself. For example:

```
53 dataset = range(100)
54 average_value = sum(dataset) / len(dataset)
```

or maybe even:

```
55 import numpy as np
56 dataset = range(100)
57 average_value = np.mean(dataset)
```

Use docstrings and comments


```
58 def compute_average(list_of_data, exclude=None):
59     """
60     Compute and return the average value of an iterable list.
61     This average excludes any value if specified by exclude
62
63     params:
64     - list_of_data: data for which the average is computed
65     - exclude: numeric value of values that should not be taken
66       into account
67
68     returns:
69     The computed average, possibly excluding a value.
70     """
71     sum = 0
72     num_elements = 0
73     for element in list_of_data:
74         if exclude is not None and element == exclude:
75             continue  # skip this element
76         sum += element
77         num_elements += 1
78     return sum / num_elements
```

Using agreed upon casing

- `snake_casing` for functions and variables
- Classes should use `CamelCasing`

```
79 def this_if_a_function(data_x, data_y):
80
81
82 class BookEntry:
```

Use type-annotations if possible

Type annotations can helper your editor (such as PyCharm) find potential issues in your code. If you use type annotations, the editor can spot types that are not compatible. For example, a string being used with a division.

https://docs.python.org/3/library/typing.html
https://realpython.com/python-type-checking/

```
83 def compute_average(list_of_data: list[int],
84                       exclude: Optional[int] = None) -> float:
85     ...
```

Organise your imports

Make the distinction between standard library imports, externally installed imports, and your own custom imports.

```
86  # internal imports
87  import os
88  from math import pi
89
90  # external imports
91  import numpy as np
92  import pandas as pd
93  import matplotlib.pyplot as plt
94
95  # custom imports
96  from src.my_module import DAGs
```

Functions should do one thing only

Do one thing and do it well. Docstrings can help you understand what your function is doing, especially if you use the word 'and' in the docstring, you might want to think about breaking your single function into many parts.

Functions as re-usability

If you find yourself doing something over and over, a function call help consolidate duplication and potentially reduce the chance of getting things wrong.

Be wary of God classes

God classes/God object is a class that is doing too many things or 'knows' about too much. When designing a class, remember that like a function, in general, it should manage one thing or concept.

Documentation

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes! – PEP 8 Style Guide

- Ensure that comments are correct.
- Don't over document (i.e. if something is self explanatory, then comments will distract rather than inform). An example from PEP 8:

```
97  x = x + 1                 # Increment x
98  x = x + 1                 # Compensate for border
```

- Document what you think will be difficult to understand without some prior knowledge, such as why a particular decision was made to do something a certain way. Don't explain, educate the reader.

Perform testing!

Make sure to write tests, for example, using `unittest` (`https://docs.python.org/3/library/unittest.html`). Writing tests can help find source of bugs/mistakes in your code, and if you change something in the future, you want to make sure that it still works. Writing tests can automate the process of testing your code.