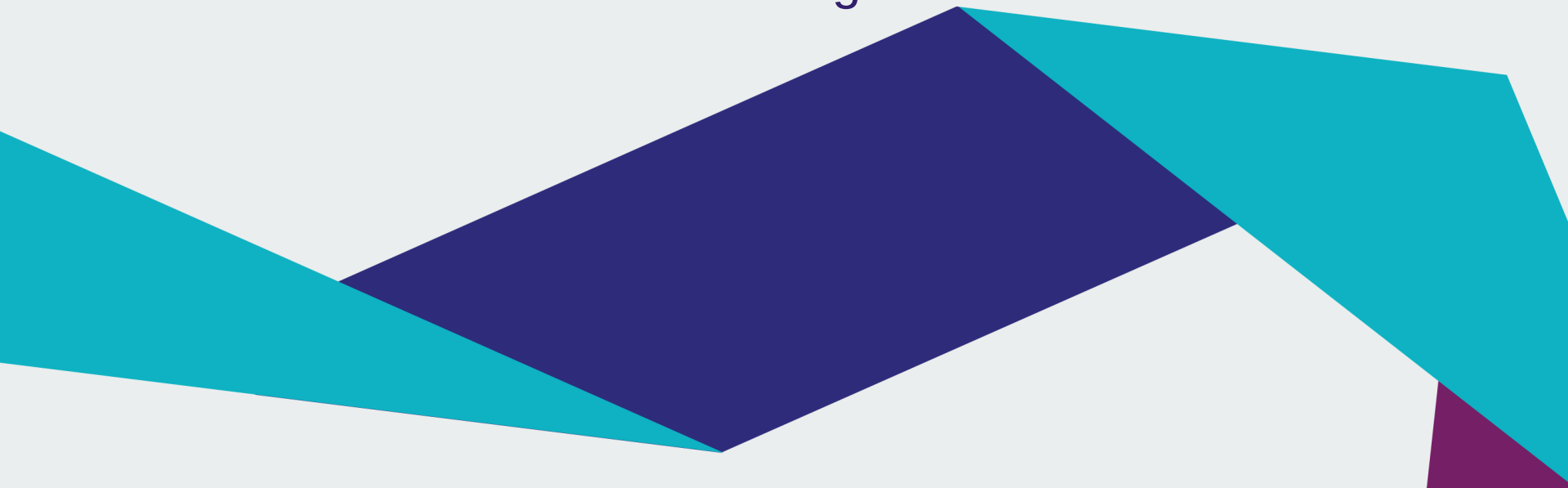
Imperial College London

Managing and Running Python Effectively

Chris Cooling

Graduate School Senior Teaching Fellow



Important Information on Marking your Attendance on Inkpath

I will show you a QR code at the end of the session allowing you to mark your attendance on Inkpath. Please do not mark your attendance until then.

If you are not a Postgraduate Research student and didn't book via Inkpath, your attendance will be marked on a separate database.

Expectations: Covid-safe teaching environments

- You are encouraged to wear a face covering indoors, especially in crowded, enclosed spaces, unless you are exempt. This includes **for Graduate School workshops**
- Cover your coughs and sneezes to reduce the spread of particles
- Respect people's wishes for extra space
- Use hand sanitiser where it is available to you

Learning Outcomes

1. **Describe** the terms “installation”, “environment”, “packages” and “kernel”
2. **Manage** Python installations and environments
3. **Utilise** Jupyter Notebooks and IDEs to create and run Python code
4. **Create** basic Python programs which use command line options

Python Installations

- A particular instance of a program installed on your computer
  - Python
  - Anaconda
- Will be a particular version of Python e.g. 3.9 or 2.7
  - Typing `python --version` in the terminal will give version number
- Contains
  - Python interpreter
  - Core Python functionality
  - Built-in modules

Python Installations

- Python
  - Install from the Python website
  - Can have multiple versions installed
  - “path” tells computer where to look for programs
    - Sometimes too many entries on path can lead to problems
- Anaconda
  - Can create from the environment tab
  - Click “Create”, give it a name and select a version

Python Environments

- An isolated installation of Python with associated installed packages
- Useful for managing projects with different dependencies
  - Different versions of Python sometimes required for different packages
  - Allows different environments for different projects
  - When starting a new project you might want to make use of functionality of new Python release

Environments in Anaconda

- Go to environments tab
- “Create” will create a new environment of a specific Python version
- “Clone” will create a duplicate of another environment
- “Backup” will save a specification of the current environment as a .yaml file
- “Import” will create an environment from a .yaml file
- “Remove” deletes an environment

Environments in Venv

- venv is a Python module
- `python3 -m venv .\Exercises\Example_environment\` will create a new environment in the selected directory
  - Use forward slashes instead of backslashes in Linux and Mac
  - Will use the same Python version as the currently active Python version

- In command prompt (**not** powerhshell)
- `.\Exercises\Example_environment\Scripts\activate.bat` will activate the virtual environment
- `deactivate` will deactivate it

- In terminal
- `./Exercises/Example_environment/bin/activate.sh` will activate the virtual environment
- `deactivate` will deactivate it

Exercise

- Anaconda
  - Create a new environment with a different Python version
  - Activate it
  - Open a new terminal in Anaconda
  - Check the Python version has changed
- Stand-alone Python
  - Create a new environment in `Exercises/Example_environment` of the course materials
  - Activate the new environment
  - Deactivate it

Definitions

- Script: A Python file designed to be run directly
- Module: A Python file containing definitions (e.g. classes, functions) designed to be imported into other modules or scripts but **not** run directly
  - Can be accessed with the `import` statement
- Package: A collection of modules
  - May contain nested directories and many files
  - Often contains an `__init__.py` file to tell Python it's a package
- Library: A (sometimes large) collection of code
  - e.g. matplotlib
  - Python standard library (contains modules like `math`) comes with a Python installation

Package Managers

- A program used to install packages and libraries
- Installs them in the current Python environment
- Packages and libraries will be available to all programs run using that Python environment
- Can access packages registered in a package index
- The most and convenient way to manage packages

Pip

- Package manager available in stand-alone Python installations and Anaconda
- Accesses packages from Python Package Index (PyPI)
- Type `pip3` in terminal to see list of commands (the `pip` command may also work, but is technically a Python 2 version)
- `pip3 install numpy` will install the module numpy
- `pip3 uninstall numpy` will uninstall the module numpy

Conda

- Available in Anaconda
- Accesses packages from Anaconda packages
- Type `conda` in an Anaconda terminal to see list of commands
- `conda install numpy` will install the module numpy
- `conda uninstall numpy` will uninstall the module numpy

Requirements File

- Contains a list of required modules (dependencies) and the versions required for a project
- Can be generated using `pip3 freeze > requirements.txt` or `conda list -e > requirements.txt`
  - Pip and Anaconda may not produce compatible file formats
  - Contains list of currently installed packages
  - Can choose any file name
  - Will be created in current directory
- Automatically generated version can contain a lot of packages as many environments will be created with a lot of packages pre-installed
- Can manually create a requirements file

Requirements File

- Can create a new Anaconda environment named `env_name` using the command `conda create -n env_name --file requirements.txt`
- Can install specified packages using pip in the current environment using `pip3 install – r requirements.txt`

Exercise

- Anaconda
  - In a current environment install the `numpy` package
  - Generate a requirements file named `requirements_anaconda.txt` in the `Exercises/Requirements` directory
  - Create a new environment using this requirements file Check the new environment has `numpy` installed
- Pip
  - In a current environment install the `numpy` package
  - Generate a requirements file named `requirements_pip.txt` in the `Exercises/Requirements` directory
  - Activate the virtual environment you created in the previous exercise
  - In the terminal write `python` then `import numpy` to verify numpy is installed
- Compare `requirements_anaconda.txt` and `requirements_pip.txt`
  - What differences can you see in their format?

Python Kernels

- An operating system process that runs Python code
- Separate to the front-end applications that you use to read/write code
- May run for a long time
- May be hosted locally or remotely
- Many applications can create a kernel

Python Commands in the Command Line

- Individual Python commands can be run directly in the command line, Anaconda terminal, etc
- Type `python` to create a kernel
- Type individual Python commands
- If using commands that require indentation (such as `for` or `if`) future commands will be taken as the indented block (indentation still required) until an empty line is entered
- Generally not a good option except for very short pieces of code

Jupyter Notebooks

- File based around a JSON file format
- Has file extension `.ipynb`
- Include sections representing Markdown code and Python code
- Normally presented in a browser window
- Creates a Python kernel
- Can be run in a variety of applications
  - Anaconda: run on local kernel
  - Colab: run on a kernel on Google's servers
- Special commands
  - Can include console commands (begin with "!")
  - Can include ipython magic commands (begin with "%")
- Can import from `.py` files

Examples/Notebooks/Sample.ipynb

Running Python within an IDE

- IDEs are application designed to help write and run code
  - You may need to install an extension to interpret Python
- Many IDEs have GUI buttons to run the current file
  - Will create a Python kernel
- Debugging
  - Many IDEs allow debugging
  - Runs the code but pauses at breakpoints
  - Can display values of variables
  - Useful for checking values without using lots of print statements and for seeing what path the interpreter takes through a code

Running Python Files From the Command Line

- Python files can be run from the command line
  - Linux/Mac terminal
  - Windows PowerShell/command prompt
  - Anaconda Terminal
  - Terminal in IDE can use these programs
- Type `python file_1.py` to run the file `file_1.py`

__name__

- When Python runs a file it will set the variable `__name__` inside the file's namespace before running code
- If this is the file directly run (such as from the command line) this will have the value `__main__`
- If the file is imported, it will be the name of the file
- Use if `__name__ == "__main__"` to cause code to run only if it's the code being directly run
  - Useful if you want to have some code which executes when it's the code directly run but not if its imported as a module

Common Command Line Arguments

- Some arguments can be provided to Python on the command line which cause it to behave differently
- Can invoke specific modules and features

Flags

- There are a number of "flags" that can be added on the command line
- "-" followed by one or more letters
- Often niche in usage and useful for more advanced users
- `-B` prevent Python from creating .pyc files when importing a module
- `-O` Assert statements and other non-essential statements ignored
- `-v` Verbose mode: provides extra output

unittest

- A module included with Python
- Automates running tests
- `python -m unittest test_script_name`
- `-m` invokes a module of the following name
- Can be used to check code and work and that it continues to work

Redirecting Output

- Python produces two types of output that can be redirected to files
- Redirection can be useful for storing or searching output
- stdout
  - Produced by print statements, etc
  - Can be redirected by writing `> filename` at the end of the command
- stderr
  - Produce by exceptions and messages from Python
  - Can be redirected by writing `2>filename` at the end of the command

Examples/Redirection

Exercise

- Write a `.py` file containing
  - A function which contains an `assert` statement which will fail
  - A `print` statement which will be executed
  - A piece of code which will only be run when the code is run directly. This should call the function.
- Run the `.py` file directly from the command line and redirect stdout and stderr to different files
- Run the .py file again but suppress the `assert` statement
- Create a Jupyter Notebook and import and call the function in the `.py` file

Command Line Arguments

- Command line arguments are the values which follow the name of the program on the command line
- Many programs accept arguments on the command line telling them exactly what to do
- Prevents having to edit the source code every time you want the code to consider a different case
- Makes the code much easier to give to other people to use
- When compiling, it allows the same executable to be used for different cases without recompiling

Command Line Arguments

- You will need to import the `sys` package
- `sys.argv` is a list containing each argument on the command line as an entry
  - Populated from the command line each time the code is run
- First value is the path to the file being run
- Subsequent values are the command line arguments
- Distinct entries are separated by spaces on the command line
- Each argument will be a string
- Can convert to other data types using functions such as `int()`, `float()`, `bool()`, etc

Input Files

- Some programs will require a lot of specification from the user
- Don’t try to provide all of this on the command line
- Instead, considered loading from a file
  - Easier to edit the data being passed in
  - Can preserve specific input to be used in the future
  - Can specify the path to the file as a command line argument

Exercise

- Write a script named `divider.py` designed to calculate and print the result when one value is divided by another
- The script should read arguments provided on the command line
  - The first is the numerator
  - The second is the denominator
- So, using the command `python divider.py 5.2 2` should cause the value `2.6` to be printed
- Consider what your code should do if
  - The number of arguments provided is not 2
  - The arguments cannot be converted to floats

Feedback

- Once you've completed this course, please provide feedback
  - The link is tinyurl.com/rcds2021-22
  - You should also have received an email with this link
  - This helps us improve the class for future students

Imperial College London

Kernels, Environments, Packages and Running Python