

Introduction to Python for Economists

Paul Wohlfarth
pwohlf01@mail.bbk.ac.uk

January 17, 2020



Contents

1	Session 1	4
1.1	What is Python?	4
1.2	An Overview of Python	4
1.2.1	Distributions	5
1.2.2	WinPython vs. Anaconda	5
1.3	Applications and Editors	9
1.3.1	Anaconda Navigator and Anaconda Prompt	10
1.3.2	Editors (Jupyter Notebook and Spyder)	13
1.4	Final Remarks	18
1.5	Online Documentation Referenced	19
2	Session 2	20
2.1	About these Slides	21
2.2	Overview	21
2.3	Variables, Datatypes, and Objects	21
2.4	Numeric Datatypes	22
2.5	Boolean Datatypes	22
2.6	Sequential Datatypes	22
2.7	Expressions, Operators, and Operands	22
2.8	Python Operators	23
2.9	Arythmetic Operators	23
2.10	Assignment Operators	23
2.11	Comparison Operators	24
2.12	Logical Operators	24
2.13	Identity Operators	24
2.14	Membership Operators	25
2.15	Set Operations	25
2.16	Iterations, Indexing, and Slicing	26
2.17	Compound (Control Flow) Statements	26
2.18	While statements	27
2.19	If Statements	27
2.20	For Statements	28
2.21	Further Compound Statements and Breaks	28
2.22	Nested Loops	28
2.23	Programme Design	29
2.24	Object Oriented Programming	29
2.25	Functions	29
2.26	Special Methods	29
2.27	Debugging	30
3	Session 3: Python Libraries	31
3.1	About Libraries	32
3.2	Working with Libraries	32
3.3	Data Analysis in Python	32
3.4	Pandas: DataFrames	33
3.5	Constructing DataFrames	33

3.6	Importing Data	34
3.7	Indexing with Pandas	34
3.8	Adding and Deleting Data	34
3.9	Export Data	35
3.10	Concatenating Data	36
3.11	Merging Data	37
3.12	Plotting Data	38
3.13	Formatting Styles	39
3.14	Plotting Multiple Objects	40
3.15	Plotting with Keyword Strings	41
3.16	Categorical Plots	42
3.17	More Visualisation	43
3.18	3D Plots	43
3.19	Animations with plotly express	44
3.20	Networks	45
3.21	and much more	47
3.22	More Libraries	47

1 Session 1

1.1 What is Python?

“Python is an interpreted, high-level, general-purpose programming language.”
Wikipedia

Interpreted: In one way or another, at some point, any programmed code needs to be translated into another programming language (eventually machine, i.e. binary, code). In Python, a build-in interpreter does this. This makes Python flexible and relatively easy but comes at the price of speed and stability.

High-level: Python uses a high level of abstraction from machine language. This makes Python “readable” and relatively accessible for a wide audience but sacrifices speed and efficiency.

General-purpose: Python is designed to be used in a wide variety of *application domains*, i.e. it can be used on any platform/ operating system.

Python is glue-code!

This very simple definition shows, what makes Python so attractive. It is flexible, so it can be used to tackle many problems on almost any device, and it is simple, so it won’t take much to learn it. Programmers would often use Python to quickly solve problems that would take much more effort in other languages. Python hence acts as glue between those languages. It is not the quickest or most stable language but it often does the job. For others, speed and stability mostly matters less, so that a language like Python is all they need. The amount of programming languages is vast and this course isn’t suitable to cover them all. Instead table 1 gives an overview of characteristics of the most common languages.

1.2 An Overview of Python

Python is very versatile and comes with a vast variety of tools, which often do the same things in slightly different ways. For beginners this can be confusing. So this part aims at giving some orientation.

Python effectively operates in two universes: One that deals with everything to do with setting up environments and installing packages. Packages (or libraries) are the main tools used to write programmes in Python. They need to be installed first. This is what package managers do. They

Language	Pros	Cons
Compiled languages (C, C++, Fortran)	Very fast!	Difficult to use; relatively inaccessible for non-programmers.
Matlab Script	Flexible and accessible (nice integrated editor) and offers commercial support	Expensive and restrictive for advanced applications.
Julia	Fast, yet simple code; connects to Python and C. → Use this when you need something faster than Python!	limited to numerical computing
Other scripting languages (Scilab, Octave, R, ...)	open-source and free (or cheaper than Matlab); contain very advanced features (statistics in R)	less flexible (fewer algorithms, some dedicated to specific domains)
Python	vast (and increasing) amount of libraries; very readable; free and open-source; very powerful environments (IPython, Spyder, Jupyter Notebooks, Pycharm, Visual Studio Code)	lacking some features that more specialised tools offer.

are typically installed with a chosen Python distribution. Python then offers a choice of editors to do the actual programming.

1.2.1 Distributions

Python comes in several alternative distributions (versions) to the official (CPython) distribution. The two most used ones are:

- [WinPython](#)
- [Anaconda](#)

The main reason why most people choose distributions other than CPython is because they come with a number of important packages that would otherwise have to be installed manually. A list of other (less popular) distributions is available [here](#).¹

1.2.2 WinPython vs. Anaconda

Both distributions come with most common packages, although the number of packages installed with Anaconda is probably unbeatable. The main difference is the package manager. WinPython uses the standard package-manager *pip* and Anaconda uses *conda*.

Practically, this means that Anaconda allows to set up different environments, which is useful to e.g. switch between using different Python releases (a common example, where this is needed is to run programmes that were written releases older than Python 3, as the newer releases come with small changes in the syntax). Changing this in long programmes can be very tedious!

¹see: <https://wiki.python.org/moin/PythonDistributions>

Another advantage is that conda tests dependencies with installed packages, which avoids conflicts whilst pip automatically installs all package dependencies irrespective of whether or not they conflict with previously installed packages.

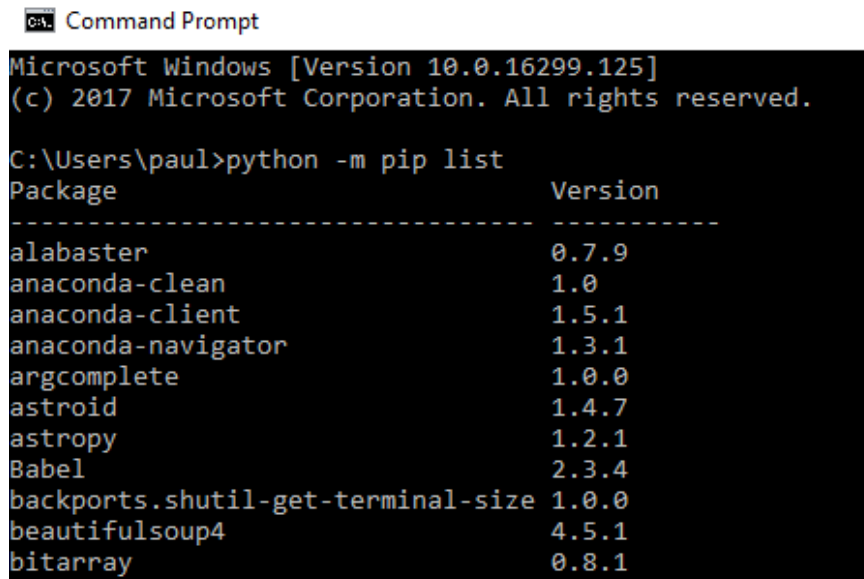
Managing packages with pip. We can simply run pip from the command line by either opening command prompt typing CMD into the search box. We now need to tell Windows that we would like to run Python commands using the module pip, so to call pip type

```
> python -m pip <argument>
```

where <argument> is the pip command to be executed. The most important one is to install packages. Type

```
> python -m pip list
```

to get a list of installed packages and respective versions:



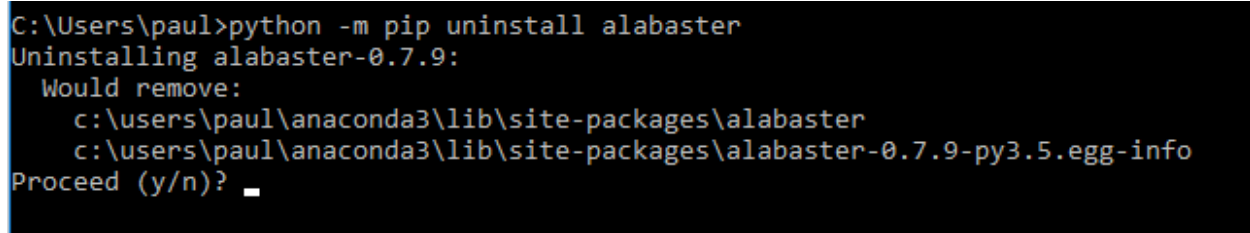
```
Microsoft Windows [Version 10.0.16299.125]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\paul>python -m pip list
Package                               Version
-----
alabaster                             0.7.9
anaconda-clean                        1.0
anaconda-client                       1.5.1
anaconda-navigator                    1.3.1
argcomplete                           1.0.0
astroid                               1.4.7
astropy                               1.2.1
Babel                                  2.3.4
backports.shutil-get-terminal-size    1.0.0
beautifulsoup4                        4.5.1
bitarray                              0.8.1
```

Type

```
> python -m pip uninstall alabaster
```

which gives the following output:



```
C:\Users\paul>python -m pip uninstall alabaster
Uninstalling alabaster-0.7.9:
  Would remove:
    c:\users\paul\anaconda3\lib\site-packages\alabaster
    c:\users\paul\anaconda3\lib\site-packages\alabaster-0.7.9-py3.5.egg-info
Proceed (y/n)?
```

Type y and press enter to confirm. This step has uninstalled the package alabaster. To install it again type

```
> python -m pip install alabaster
```

which gives the following output:

```
C:\Users\paul>python -m pip install alabaster
Collecting alabaster
  Downloading https://files.pythonhosted.org/packages/10/ad/00b090d23a222943eb0eda509720a404f531a439e803f6538f35136cae9e/alabaster-0.7.12-py2.py3-none-any.whl
Installing collected packages: alabaster
Successfully installed alabaster-0.7.12
```

To upgrade a package, type

```
> python -m pip install --upgrade alabaster
```

You can also upgrade pip in the same way, ie

```
> python -m pip install --upgrade pip
```

which gives:

```
C:\Users\paul>python -m pip install --upgrade pip
Collecting pip
  Downloading https://files.pythonhosted.org/packages/00/b6/9cfa56b4081ad13874b0c6f96af8ce16c1cb06bedf8e9164ce5551ec/pip-19.3.1-py2.py3-none-any.whl (1.4MB)
    |#####| 1.4MB 3.3MB/s
Installing collected packages: pip
  Found existing installation: pip 19.2.3
    Uninstalling pip-19.2.3:
      Successfully uninstalled pip-19.2.3
  Successfully installed pip-19.3.1

C:\Users\paul>python -m pip install --upgrade alabaster
Requirement already up-to-date: alabaster in c:\users\paul\anaconda3\lib\site-packages (0.7.12)

C:\Users\paul>
```

These are usually all commands needed to work with pip. The full reference is available [here](https://pip.pypa.io/en/stable/reference/).²

Managing Packages with conda. Package management using conda is very similar to pip, but since conda allows working in different environments, we need to call those as well. Both conda and pip can be accessed through their own command prompts. The syntax is the same, we simply don't have to call the Python modules anymore.

First, to verify that conda is installed, type

```
conda --version
```

Should this cause an error message, close and re-open the terminal window and verify that you are logged into the same user account on your machine that Anaconda was installed on (should

²See: <https://pip.pypa.io/en/stable/reference/>

you have multiple accounts).

To update conda, type

```
conda update conda
```

If a newer version of conda is available the response will be

```
Proceed ([y]/n)? y
```

Type y and enter to proceed.

To search for packages, type (replace <> by the package name)

```
conda search <package>
```

To install a new package in conda, using the conda command prompt type

```
conda install -n <environment> <package>
```

To install a package from a particular channel, type

```
conda install -c conda-forge rise
```

We will need this extension later.

Using a single hyphen allows abbreviating commands with the first letter. So here -n abbreviates --name.

New environments in conda are created using the create command:

```
conda create -n <environment> <python distribution>
```

e.g. to create a new environment "bunnies" with Python 3.4, type

```
conda create -n bunnies python=3.4
```

Not specifying this option results in the environment using the version of Python that is currently running.

To create some new environment snowflakes and install some package, say biopython on it, type

```
conda create -n snowflakes biopython
```

Switching between multiple environments is called activating the environment. Type

```
conda activate <environment>
```

or, for versions prior to conda 4.6

```
activate <environment>
```

for Windows and


```
source activate <environment>
```

for Linux and macOS.

To view a list of created environments, type

```
conda info -e
```

To see a list of packages installed on the active environment, type

```
conda list
```

and

```
conda list -n <environment>
```

for a list of packages installed on other environments.

conda offers a help function for commands. Type

```
conda create --help
```

to see the documentation for the command help.

These commands are what you need in most simple application. conda allows for much more complex package and environment management, which is usually not needed for simple application (i.e. not creating packages etc.). There is an excellent documentation on everything relating to conda [here](https://docs.conda.io/projects/conda/en/latest/user-guide/overview.html).³ A useful collection of the most important conda is available in this [cheat sheet](https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html).⁴

1.3 Applications and Editors

The preceding two sections have dealt with everything needed to set Python up. This section deals with what follows: Applications and editors used to write Python programmes. Anaconda installs nine tools by default. Not all of them are needed on an everyday basis. The most important apps are

- Anaconda Navigator
- Anaconda Prompt
- Jupyter Notebook
- Spyder

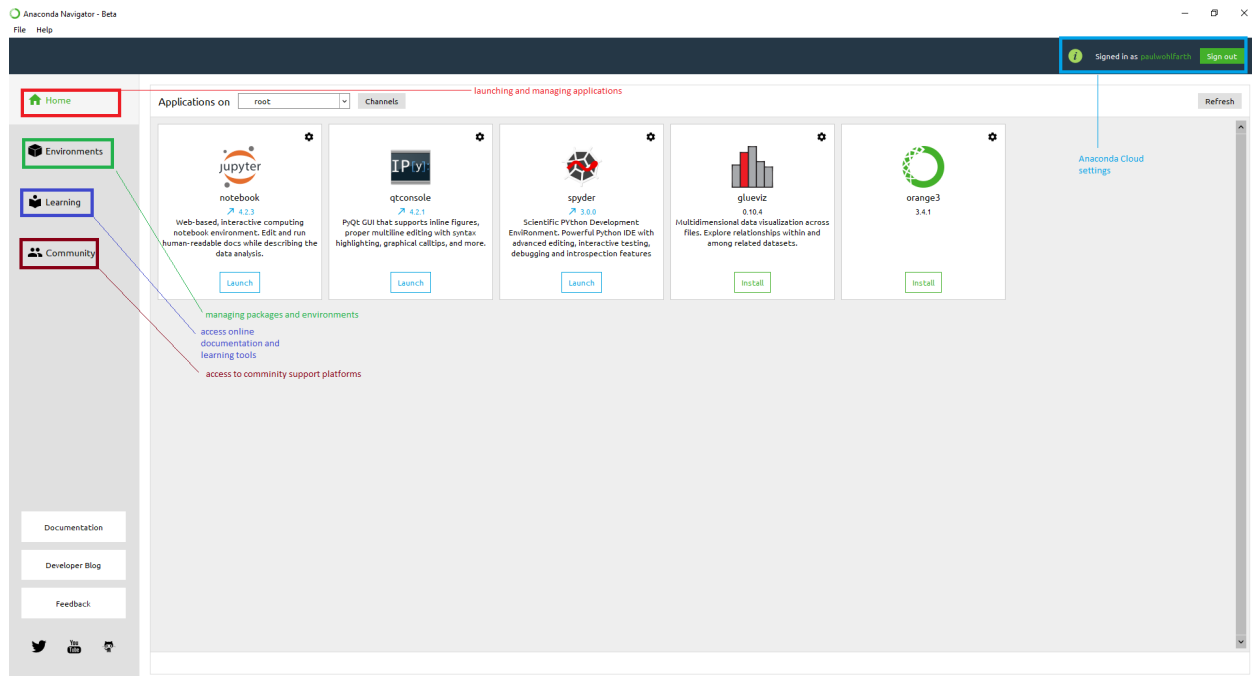
³See: <https://docs.conda.io/projects/conda/en/latest/user-guide/overview.html>

⁴See: <https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html>

1.3.1 Anaconda Navigator and Anaconda Prompt

Anaconda Prompt is the conda command prompt (or terminal). This is what we worked with in section 1.2 and can also be called from the Windows command prompt (or terminal for macOS and Linux).

Navigator is an extremely useful tool, particularly for those new to Python. It allows package and environment management without having to use terminal commands, gives quick access to documentation and online help communities and allows management of installed applications.



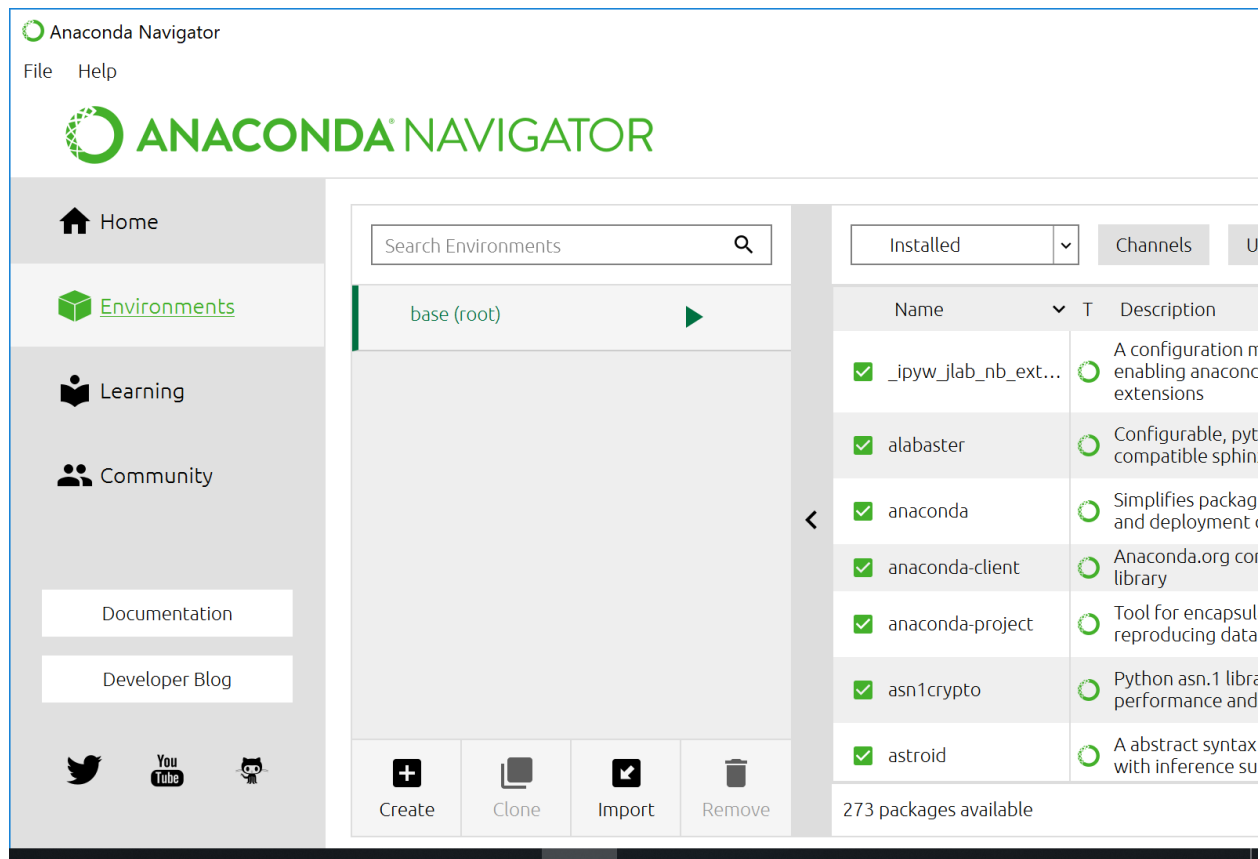
Home tab: Here, applications can be launched, installed, removed or updated. Currently, the following applications are available for Anaconda Navigator:

- JupyterLab
- Jupyter notebook
- Orange data visualisation
- Spyder IDE
- Glueviz multidimensional data visualisation
- R Studio IDE
- VS Code

Instructions on how to build own applications are available [here](#).⁵

⁵See: https://conda.io/docs/build_tutorials/app.html

Environments tab: This tab allows for environment and package management, without need for any command-line commands.

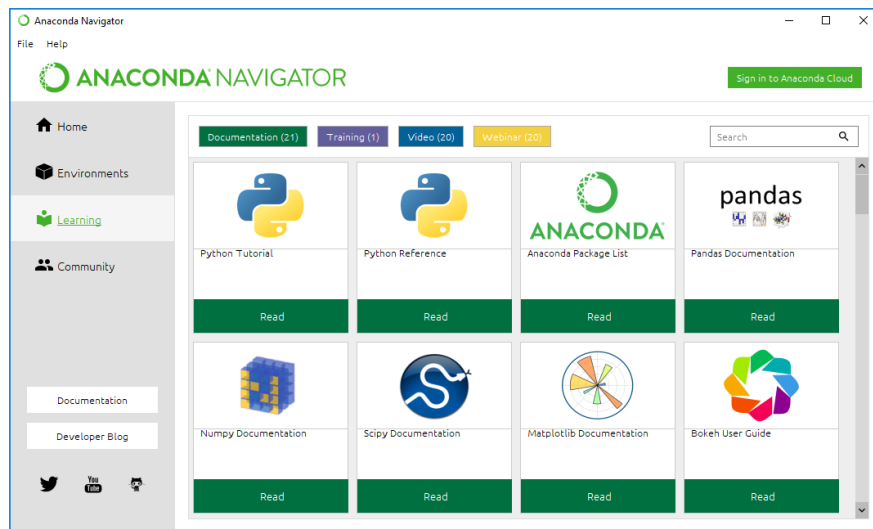


There are two main columns: Environments are listed on the left. An environment can be activated by a simple mouse-click. Further instructions on managing environments are available [here](https://docs.continuum.io/anaconda/navigator/tutorials/manage-environments/).⁶ The right column gives a list of all packages in the current environment. More information on package management, using Anaconda Navigator is available [here](https://docs.continuum.io/anaconda/navigator/tutorials/manage-packages/).⁷

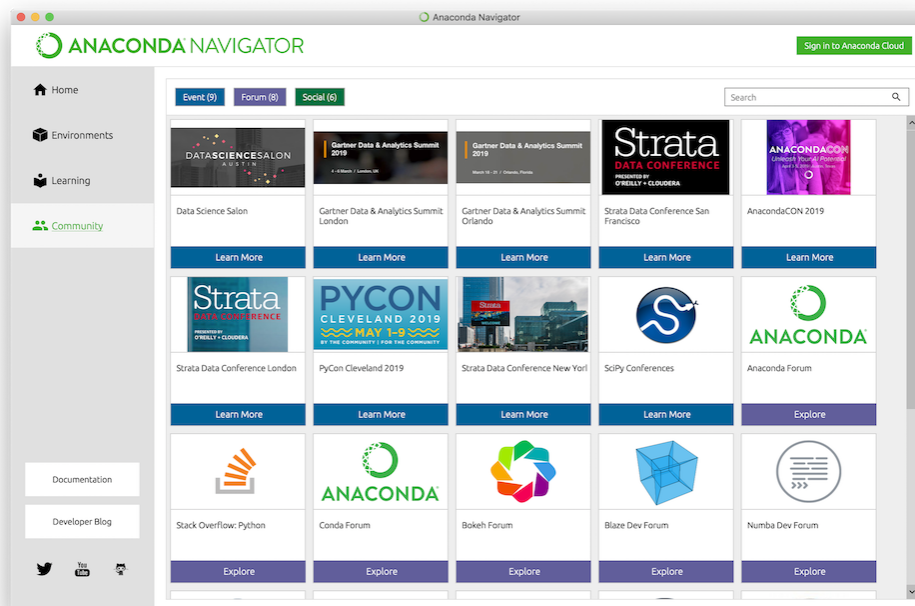
Learning tab: This tab gives a collection of learning and documentation reference resources, which are available for free online. Simply click on one of the buttons to open a resource in the browser.

⁶See: <https://docs.continuum.io/anaconda/navigator/tutorials/manage-environments/>

⁷See: <https://docs.continuum.io/anaconda/navigator/tutorials/manage-packages/>



Community tab: This tab offers further information about events and free support forums. Clicking on the buttons opens more information in a browser window.



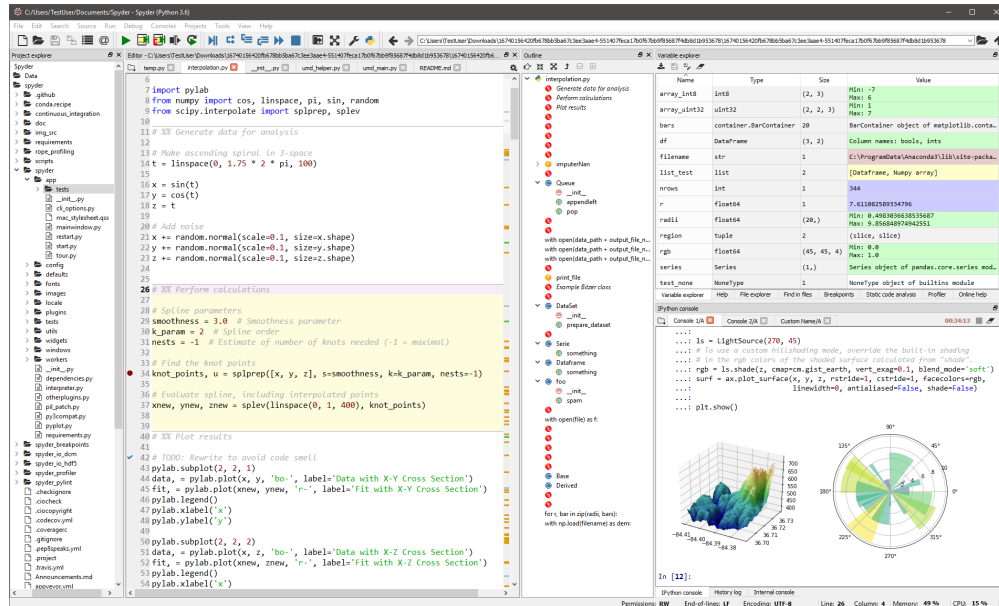
Further settings and Anaconda Cloud: Further settings are available in the **File** menu, in particular enabling offline mode, which allows limited functionality of Navigator offline.

Anaconda Cloud is a package management tool that allows accessing, storing and sharing public notebooks (more on this later) as well as conda and PyPI packages online. This is particularly useful for package development.

1.3.2 Editors (Jupyter Notebook and Spyder)

Anaconda offers two main choices for editors, i.e. tools to write Python programmes with, pre-installed: Jupyter Notebook and Spyder. They are both great tools that serve different purposes.

Spyder. This is a fully functional editor that is particularly useful for writing complex and long scientific programmes. It requires full installation and is normally used on desktop PCs or macOS/ Linux machines. This is a very brief overview of main functions in Spyder. Detailed documentation is available [here](#).



There are two main components of the Spyder interface: The left half gives a file browser (similar to Matlab) and an editor, which is where programmes are written.

The editor allows for multi-tabbing, so multiple programs can be opened in different tabs. The written text is colour coded for readability.

```
6
7 import pylab
8 from numpy import cos, linspace, pi, sin, random
9 from scipy.interpolate import splprep, splev
10
11 # %% Generate data for analysis
12
13 # Make ascending spiral in 3-space
14 t = linspace(0, 1.75 * 2 * pi, 100)
15
16 x = sin(t)
17 y = cos(t)
18 z = t
19
20 # Add noise
21 x += random.normal(scale=0.1, size=x.shape)
22 y += random.normal(scale=0.1, size=y.shape)
23 z += random.normal(scale=0.1, size=z.shape)
24
25
26 # %% Perform calculations
27
28 # Spline parameters
29 smoothness = 3.0 # Smoothness parameter
30 k_param = 2 # Spline order
31 nests = -1 # Estimate of number of knots needed (-1 = maximal)
32
33 # Find the knot points
34 knot_points, u = splprep([x, y, z], s=smoothness, k=k_param, nests=-1)
35
36 # Evaluate spline, including interpolated points
37 xnew, ynew, znew = splev(linspace(0, 1, 400), knot_points)
38
39
40 # %% Plot results
41
42 # TODO: Rewrite to avoid code smell
43 pylab.subplot(2, 2, 1)
44 data, = pylab.plot(x, y, 'bo-', label='Data with X-Y Cross Section')
45 fit, = pylab.plot(xnew, ynew, 'r-', label='Fit with X-Y Cross Section')
46 pylab.legend()
47 pylab.xlabel('x')
48 pylab.ylabel('y')
49
50 pylab.subplot(2, 2, 2)
51 data, = pylab.plot(x, z, 'bo-', label='Data with X-Z Cross Section')
52 fit, = pylab.plot(xnew, znew, 'r-', label='Fit with X-Z Cross Section')
53 pylab.legend()
```

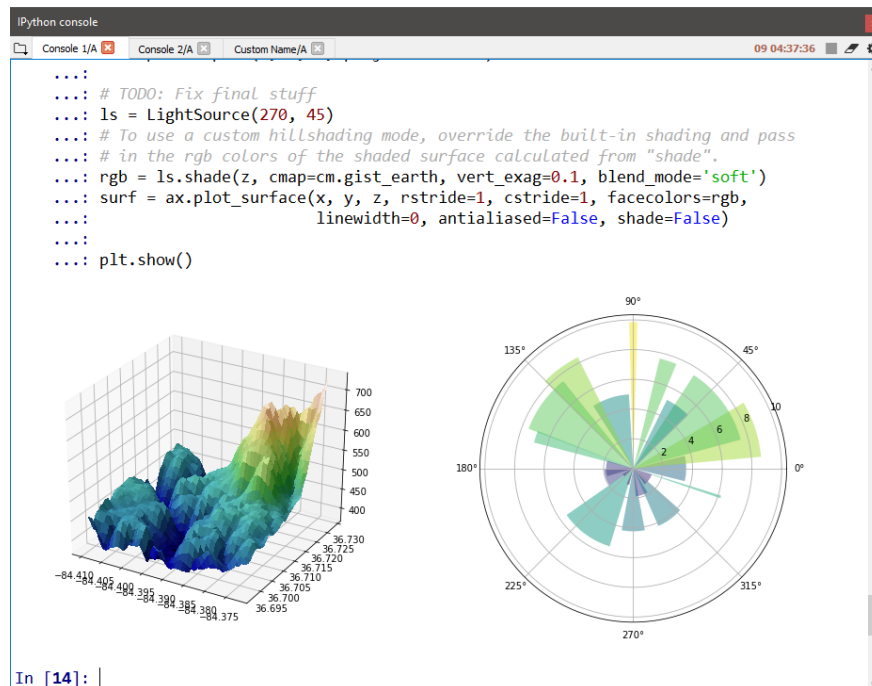
Lines are numbered and real-time code and style analysis (using pyflakes and pycodestyle) is marked with little exclamation mark symbols to the left of the line numbers and horizontal lines to the right. Hovering over the symbols with the cursor gives code analysis suggestions. Code errors are marked in red and style suggestions in yellow.

```

248
249 def read(filename, encoding='utf-8'):
250     """
251     Read text from file ('filename')
252     Return text and encoding
253     """
254     text, encoding = decode( open(filename, 'rb').read() )
255     return text, encoding
256
257 E201 whitespace after '(' encoding='utf-8'):
258 E202 whitespace before ('filename')
259 ')
260 Return lines and encoding
261 """
262 text, encoding = read(filename, encoding)
263 return text.split(os.linesep), encoding
264

```

The right bottom window contains the IPython Console. It allows execution, interaction and visualisation of Python commands. This is particularly useful for running and testing small pieces of code without disrupting a primary session.



On the top right side, Spyder features a variable explorer, which lists details on variables in the current IPython session. This is particularly useful for quickly looking up data types of variables used in programmes.

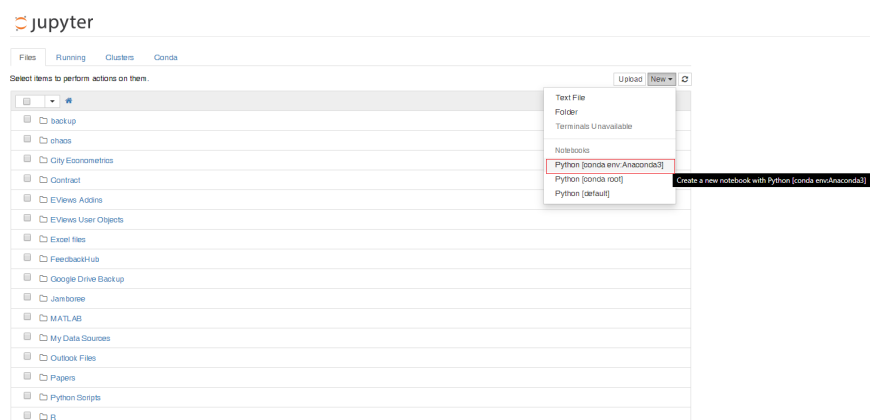
Variable explorer			
Name	Type	Size	Value
array_uint32	uint32	(2, 2, 3)	Min: 1 Max: 7
bars	container.BarContainer	20	BarContainer object of matplotlib.container module
df	DataFrame	(3, 2)	Column names: bools, ints
df_complex	DataFrame	(5, 1)	Column names: 0
filename	str	1	C:\ProgramData\Anaconda3\lib\site-packages\matplotlib\mpl-data\...
list_test	list	2	[Dataframe, Numpy array]
long_text	str	1	This is some very very very very long text! But Spyder can show...
nrows	int	1	344
r	float64	1	6.469949121504568
radii	float64	(20,)	Min: 0.031808170090177335 Max: 9.934459607320779
region	tuple	2	(slice, slice)
rgb	float64	(45, 45, 4)	Min: 0.0 Max: 1.0

Jupyter Notebooks. Jupyter offers an interactive alternative to fully equipped editors, such as Spyder. This is particularly useful for quickly experimenting with new ideas, as Jupyter has a fantastic build-in help function. Jupyter Notebooks are also a useful tool for writing code on devices other than PCs or macOS/Linux machines. Thanks to cloud-computing Python programmes can even be written on smartphones using Jupyter Notebooks!

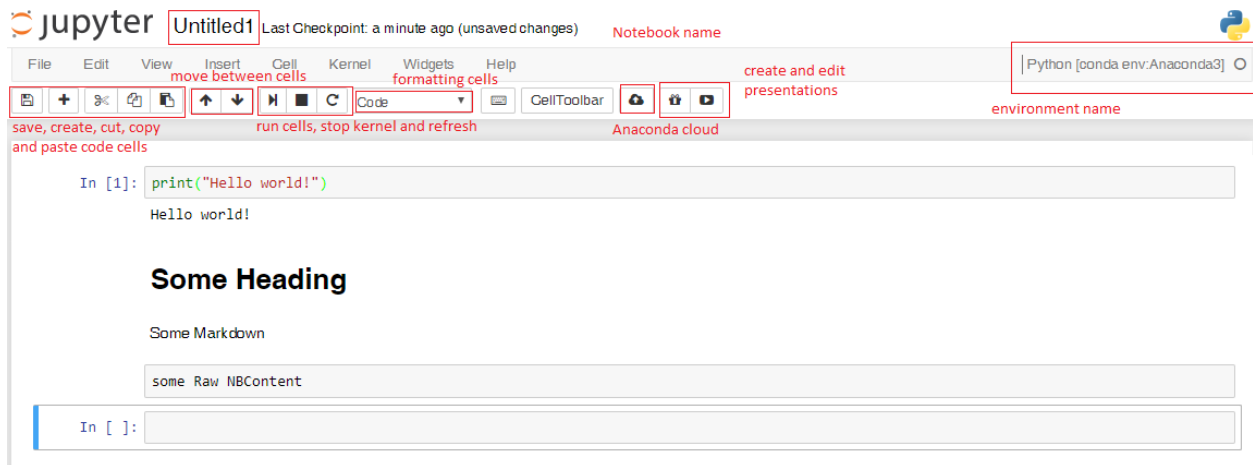
Jupyter Notebooks can be opened via either installed apps in the Anaconda folder or via prompt using the command

```
jupyter notebook
```

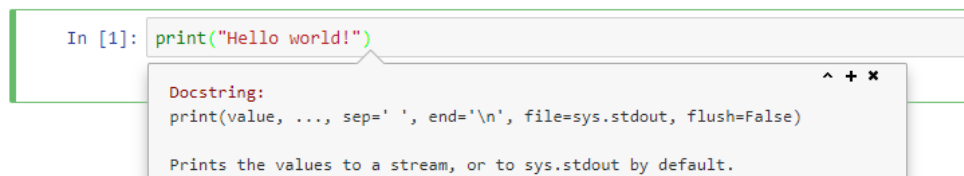
Jupyter opens in the file view, allowing to either open an existing programme file, called notebook, or to create a new notebook. To create a new notebook, click new and select the chosen environment, here Anaconda3, to work with.



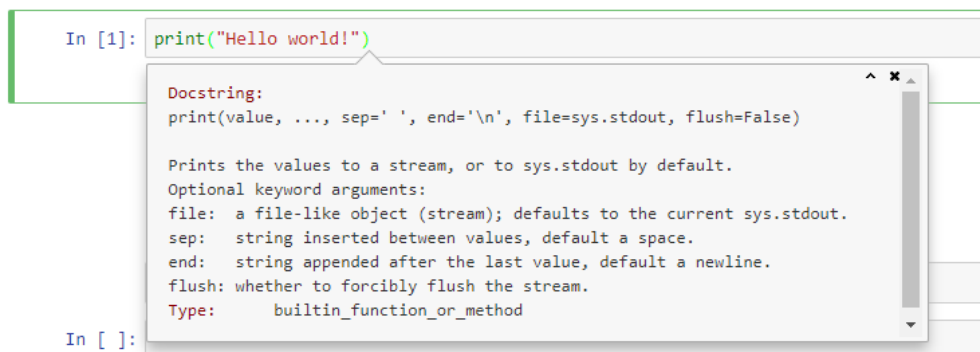
Programmes are written in code cells in Jupyter Notebook. One notebook could effectively be anything you'd want it to be, but often it's one programme or project. To execute a code cell press either Ctrl + enter or click run. You can also edit or format cells, create and edit presentations and save notebooks either locally or in Anaconda cloud. The figure below gives an overview of the layout for the most important functions. In the remaining parts of this workshop you will see examples of presentations written in Jupyter Notebook.



One useful feature in Jupyter notebooks is its build-in help function. This allows experimenting with code and makes Jupyter ideal for workshops like this one! To access the help function, click into an active cell, with code you'd like to evaluate and press shift+tab. This gives



To get more information repeat pressing shift + tab:



```
In [1]: print("Hello world!")
```

Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method

```
In [ ]:
```

```
In [1]: print("Hello world!")
```

Hello world!

Some Heading

Some Markdown

some Raw NBContent

```
In [ ]:
```

Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method

1.4 Final Remarks

This section covers everything you need to know to work with Python. It is not a complete manual, that's what the numerous documentation and reference manuals are for. These notes should be understood as allowing for an efficient and quick entry into the Python world. In this sense they provide the essentials alongside references for further information.

One important aspect that is left out in this workshop is cloud computing and remote access. This is deliberate as these are complex topics that can (and do) fill whole workshops alone.

As further information, I abstain from providing a reading list. There are, of course, several good textbooks available. But Python offers excellent resources for free that I find much more useful instead. In that sense these notes offer a map to navigate these resources. A list of documentation content referred to throughout this section is provided below.

1.5 Online Documentation Referenced

- <https://wiki.python.org/moin/PythonDistributions>
- <https://pip.pypa.io/en/stable/reference/>
- <https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html>
- <https://docs.conda.io/projects/conda/en/latest/user-guide/overview.html>
- <https://docs.continuum.io/anaconda/navigator/tutorials/manage-packages/>
- <https://docs.continuum.io/anaconda/navigator/tutorials/manage-environments/>
- <https://docs.spyder-ide.org/>

2 Session 2

2.1 About these Slides

These slides have been written as jupyter notebooks. This is very easy to do, as code cells can be defined as markdown cells like this one. Markdown cells also allow for Latex formulae, like this one:

$$c = \sqrt{a^2 + b^2}$$

and even embed images like this one:



There are many more options to present results in Jupyter Notebooks and I won't show them all now. Instead, I will use some functions in these slides and share the notebook files with you in the workshop. This way you will have a blueprint that you can use later.

I have used the RISE extension for these slides, that we have installed earlier. The documentation is available [here](https://rise.readthedocs.io/en/maint-5.6/).⁸

2.2 Overview

1. Datatypes, Variables, and Objects
2. Mathematical operations in Python
3. Indexing, lists and dictionaries
4. Compound statements
5. Programme design, object oriented programming, debugging

2.3 Variables, Datatypes, and Objects

An **object** can be a variable, a data-structure, a function, or a method. More technically, an object refers to an instance of a class.

A **variable** is simply a storage container. Something that carries data. Variables can be global (defined outside functions) or local (defined inside functions), they can be instance variables, class variables and more. But the definition we need is: something that carries data.

Variables have **datatypes** which classify data. Python supports the following datatypes:

- Numeric (numbers)
- Boolean (True/False)
- Sequential (text and ordered collections)
- Dictionaries (unordered collections)

⁸See: <https://rise.readthedocs.io/en/maint-5.6/>

2.4 Numeric Datatypes

Python supports the following numeric datatypes:

- Integer: Positive or negative whole numbers
- int: Plain integers
- long: Long integers (integers of infinite size)
- Float: Real number allowing for decimals; up to 17 digits in length
- Complex numbers

2.5 Boolean Datatypes

Data that can take the values True and False. **Important:** First letters are capitalised!

Booleans are used for simple logical tests with binary outcome (True or False).

```
# Example:  
my_string=("Hello world!")  
print(my_string.isnumeric())
```

2.6 Sequential Datatypes

Python supports the following sequential datatypes:

- string, str(): text
- lists, []: ordered collection of different datatypes, mutable
- tuples, (): ordered collection of different datatypes, immutable

Immutable means that tuples cannot be changed, once created.

maybe some example slide

2.7 Expressions, Operators, and Operands

Expressions perform specific actions, based on **operators** with one or two **operands**.

Example:

$$3 \times 5 = 15$$

In this expression, a multiplication (operator) is performed between the numbers 3 and 5 (operands) with the result 15.

Put simply: Operands are the numbers and operators the operations we perform on them.

2.8 Python Operators

Python supports operators that can be grouped into

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- (Bitwise operators)

2.9 Arithmetic Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

2.10 Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

2.11 Comparison Operators

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

2.12 Logical Operators

Operator	Description	Example
<code>and</code>	Returns True if both statements are true	<code>x < 5 and x < 10</code>
<code>or</code>	Returns True if one of the statements is true	<code>x < 5 or x < 4</code>
<code>not</code>	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

#Example:

```
x=3
x>0 and x<5
True
```

2.13 Identity Operators

Operator	Description	Example
<code>is</code>	Returns true if both variables are the same object	<code>x is y</code>
<code>is not</code>	Returns true if both variables are not the same object	<code>x is not y</code>

Example

```
x=3
y=3
x is y
True
```


2.14 Membership Operators

Operator	Description	Example
<code>in</code>	Returns True if a sequence with the specified value is present in the object	<code>x in y</code>
<code>not in</code>	Returns True if a sequence with the specified value is not present in the object	<code>x not in y</code>

Example

```
x=2
```

```
y=[1,2,3]
```

```
x in y
```

```
True
```

2.15 Set Operations

Operation	Equivalent	Result
<code>len(s)</code>		number of elements in set <i>s</i> (cardinality)
<code>s.issubset(t)</code>	<code>s <= t</code>	test whether every element in <i>s</i> is in <i>t</i>
<code>s.issuperset(t)</code>	<code>s >= t</code>	test whether every element in <i>t</i> is in <i>s</i>
<code>s.union(t)</code>	<code>s t</code>	new set with elements from both <i>s</i> and <i>t</i>
<code>s.intersection(t)</code>	<code>s & t</code>	new set with elements common to <i>s</i> and <i>t</i>
<code>s.difference(t)</code>	<code>s-t</code>	new set with elements in <i>s</i> but not in <i>t</i>
<code>s.symmetric_difference(t)</code>	<code>s^t</code>	new set with elements in either <i>s</i> or <i>t</i> but not both
<code>s.copy()</code>		new set with a shallow copy of <i>s</i>

#Example

```
A = {0, 2, 4, 6, 8};
```

```
B = {1, 2, 3, 4, 5};
```

```
C = {1,2,3,1,2,3};
```

Counting elements in A

```
lenA=len(A)
```

```
print(lenA)
```

Union of sets A and B

```
print(A | B)
```

```
D=A.union(B)
```

```
print(D)
```

#Eliminating duplicates:

```

C = list(set(C))
print(C)

5
{0, 1, 2, 3, 4, 5, 6, 8}
{0, 1, 2, 3, 4, 5, 6, 8}
[1, 2, 3]

```

2.16 Iterations, Indexing, and Slicing

Iterations are repeated processes to obtain a sequence of outcomes. **Iterators** are objects that can be iterated upon. Iterations allow for efficient programming that causes less errors.

Indexes refer to individual elements in **iterables** (most collections/containers) and **indexing** refers to ways to access individual elements. **Slicing** refers to obtaining subsets of iterables.

```

# Examples:
squares = [1, 4, 9, 16, 25]
index = squares[2] #Indexing: Access the third element in 'squares'
print(index)
slice1 = squares[2:4] #Slicing: Accessing the slice from the third to the forth
↪elements in 'squares'
print(slice1)
slice2 = squares[:-2] # Access all elements minus the last two
print(slice2)

9
[9, 16]
[1, 4, 9]

```

2.17 Compound (Control Flow) Statements

Iterations are implemented in control flow statements. There are generally 3 types of control flow statements in Python:

1. **while** statements
2. **if** statements
3. **for** statements

Note: Python uses indentation, i.e. the expression inside a control flow statement must be indented.

2.18 While statements

While statements iterate for as long as a condition is true.

*#Example: Fibonacci sequence (each element is the sum of the two preceeding ↵
↵elements)*

```
a, b = 0, 1
while a < 10:
    print(a)
    a, b = b, a+b
```

0

1

1

2

3

5

8

2.19 If Statements

An if statement executes an iteration based on a condition, which is given by the expression following **if**. Otherwise the program will execute the expression following **else**.

The iteration can test further conditions, using **elif**.

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

a is greater than b

2.20 For Statements

The for statement is used to iterate over the elements of a sequence or other objects.

```
for i in range(10):
    print(i)
    i = 5
```

2.21 Further Compound Statements and Breaks

There are other types of statements that are less commonly used for more advanced programmes. E.g. **try statements** for exception handling and **with statements** for wrappers (a programme that calls other programmes). We won't be covering those, but more information is available in the documentation [here](#).⁹

Control flow statements can sometimes lead to indefinite iterations. For this reason, and for debugging purposes it is useful to be able to stop them. This is what **break** statement does.

A **continue** statement continues with the next cycle of the nearest enclosing loop.

2.22 Nested Loops

Loops can be nested. A nested loop is a loop that exists inside another loop.

#Example for nested while loop (finding prime numbers up to 100)

```
i = 2
while(i < 30):
    j = 2
    while(j <= (i/j)):
        if not(i%j): break
        j = j + 1
    if (j > i/j) : print(i, " is prime")
    i = i + 1
```

```
print("Good bye!")
```

```
2  is prime
3  is prime
5  is prime
7  is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
Good bye!
```

⁹See: https://docs.python.org/3/reference/compound_stmts.html#while

2.23 Programme Design

Good design, i.e. planning of code is important, especially when programmes become large. Otherwise we create *Spaghetti code* (i.e. code with circular dependencies, lots of global variables, large nested loops) that complicate further and further, leading to conflicts down the line.

To avoid this, best practise should be followed. Python style guides (often build into editors) can be used for that, e.g. - pylint - pychecker - pep8

2.24 Object Oriented Programming

Object oriented programming uses the concept of inheritance, i.e. reusing methods and properties of objects, to obtain an efficient code structure. Common methods and properties are defined in classes.

Example: LandRover is an instance of the class *car*. **Parent Class** refers to a class that contains further classes, called **child class**.

Example: LandRover is an instance of the class *SUV*, which belongs to the parent class *car*. Child classes inherit attributes from parent classes.

2.25 Functions

A function is a part of code that only runs when it is called. Functions are defined using `def` followed by the function name and comma-separated arguments in brackets. Statements, that form the body of the function follow indented below.

```
def my_function(arg1, arg2, arg3)
>> Statement1
>> Statement2
```

2.26 Special Methods

There are special methods (enclosed by double underscores) in python that use reserved names. A lot of this is beyond the scope of this workshop (more info [here](#)).¹⁰

One special method name you will likely encounter is called *instantiation*. The function `__init__`, called a *constructor*, is used to construct an instance of the class. The default constructor is `__init__(self)`. This constructor only refers to the instance that is being created by the constructor itself.

One way to think about it is a table: We will first have to create an empty table, before we can populate it. Similarly, we will have to create an 'empty' class before we can use it.

¹⁰See: <https://docs.python.org/2/reference/datamodel.html#special-method-names>

```
#Example Classes:
class Person:
    def init(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")
x.printname()
```

2.27 Debugging

Debugging is the process of removing errors (bugs) from code. The challenge often is finding rather than removing the error.

Print statements. This is the simplest way to detect errors. If a statement that has been added to the code is printed, this indicates that the segment up to the print statement has no error.

Break points. A simple alternative to print statements that do the same job in a different way: stopping the code. If an error is raised, it has to be in the segment before the break.

Debuggers. These are built-in tools that highlight errors. Python's native debugger is [IDLE](#), another one is [pdb](#) Spyder uses [ipdb](#), but other tools are available.

3 Session 3: Python Libraries

3.1 About Libraries

Libraries are collections of resources that can be used. They are like toolboxes we can pick the tools from we need. Python has a large number of tools built-in, some of which we have already worked with. This is called the *Python Standard Library*.¹¹

But things get really interesting with all the other libraries around. Python is open-source, so the amount of resources is endless. Many of the most popular libraries are already installed with Anaconda (which is one of the reasons to choose it). But there are many more available.

It is impossible to cover all libraries in such a short workshop. So in this session we'll focus on the most important ones that come with Anaconda. These are

- **Pandas** along with **SciPy** and **NumPy** for data analysis and scientific programming, and
- **matplotlib** for visualisation

What this session will also do is giving resources and showing how to work with libraries. So ideally you'll be able to work with and explore more libraries yourself.

3.2 Working with Libraries

In Python libraries are technically *modules*. These need to be *imported*¹² to be used. simply write

```
import <module>
```

Modules are essentially a collection of scripts that contain functions. Often parts of modules can be called separately. For this type

```
from <module> import <part>
```

3.3 Data Analysis in Python

The main data structure in Python are **arrays**, which are essentially lists. They can be mutable or immutable. Since everything in Python is an object (even functions), arrays also support different datatypes, which is very powerful (but slow)!

The **NumPy** library allows working with multi-dimensional arrays. NumPy is written in low-level language (C and Fortran) for high-level mathematical functions. Having multi-dimensional arrays is very powerful (see matlab) as it allows avoiding slow loops, but NumPy arrays are homogeneous (i.e. of the same datatype), which makes them less flexible. Therefore Numpy is mainly used for mathematical operations, together with **SciPy**, which gives more maths-functionality.

Pandas is built on top of NumPy and has 2-dimensional arrays, called *dataframes*, as main data-structure. It supports heterogeneous datatypes and can be nested, which makes it powerful for data analysis.

¹¹See: <https://docs.python.org/3/library/>

¹²See: <https://docs.python.org/3/tutorial/modules.html>

3.4 Pandas: DataFrames

There are two data structures in Pandas

- Series are one dimensional, homogeneous arrays
- DataFrames are two-dimensional arrays with heterogeneous columns

DataFrames can be constructed from collections such as lists or dictionaries and from existing records, such as ndarray.

3.5 Constructing DataFrames

To construct DataFrames use `<my_df> = pd.DataFrame()`

```
# Example:
import pandas as pd

# lists
wds = ["these", "are", "some", "words"]
owds = ["words", "have", "datatype", "string"]
nbs = [1, 2, 3, 4]

# dictionary of lists
dict = {'words': wds, 'other words': owds, 'numbers': nbs}

df = pd.DataFrame(dict)

df
```

	words	other words	numbers
0	these	words	1
1	are	have	2
2	some	datatype	3
3	words	string	4

3.6 Importing Data

Pandas can import many file formats. The most important ones are excel spreadsheets (.csv and .xlsx files). To import a csv file use `read_csv()`

```
import pandas as pd
data = pd.read_csv('data.csv', parse_dates = ['col1'])
data

date col1 col2
0 01/01/20    1    a
1 02/01/20    2    b
2 03/01/20    3    c
3 04/01/20    4    d
```

3.7 Indexing with Pandas

Having imported data, we might want to work with only a subset of it. We can use *slicing* and *indexing* to call those subsets. `iloc` provides a useful tool for this.

```
data.iloc[0:1,0:2]

date col1
0 01/01/20    1
```

3.8 Adding and Deleting Data

We can add columns with `data.insert()`, rows with `data.loc()` and delete data with `data.drop()`.

```
# Example (don't execute everything at the same time)
data = data.drop(columns="newcol") ##drop column
data.insert(3, "newcol", [5, 6, 7, 8], False) ##add newcol
data = data.loc[4] = ['03/01/20', 9, 'e', 10] ##add row 4
data = data.drop(4) ##drop row 4

data

date col1 col2 newcol
0 01/01/20    1    a      5
1 02/01/20    2    b      6
2 03/01/20    3    c      7
3 04/01/20    4    d      8
```

3.9 Export Data

Export data as .csv file, simply use `pd.<dataframe>.to_csv(<filepath>)`. More options are available in the documentation and files can be exported in different formats.

```
#Example: Exporting dataframe 'data2' as csv  
data.to_csv('/Users/paulwohlfarth/Desktop/external/SPE/Session3/data2.csv',  
sep= ',')
```

3.10 Concatenating Data

Concatenation is simply adding data. In the following example we concatenate along the index axis (i.e. we add rows). For this we have three steps - we create the dataframes, and index rows uniquely, - we then create a list of the three dataframes, - we concatenate the dataframes using `pd.concat()`

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])

df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])

frames = [df1, df2, df3]
result = pd.concat(frames)

result
```

A	B	C	D	
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

3.11 Merging Data

Merging is very similar to concatenating. The difference is that DataFrames are joined based on keys and not that data is simply added.

We will only deal with the simplest merge function, that joins two dataframes based on one unique key, which is needed to uniquely identify rows or columns. The key can be either columns or indexes. There are generally four ways to merge:

- left (based on the first dataframe)
- right (based on the second dataframe)
- inner (default; intersection of dataframes)
- outer (union of dataframes)

Pandas also allow for more complex merging operations on multiple keys. This involves knowledge of relational algebra, which is beyond the scope of this workshop. More information on data merging with pandas is available [here](#).

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

left

  key  A  B
0  K0  A0 B0
1  K1  A1 B1
2  K2  A2 B2
3  K3  A3 B3

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})

right

  key  C  D
0  K0  C0 D0
1  K1  C1 D1
2  K2  C2 D2
3  K3  C3 D3

#Example result:
result = pd.merge(left, right, on='key', indicator=True)
result

  key  A  B  C  D _merge
0  K0  A0 B0 C0 D0  both
1  K1  A1 B1 C1 D1  both
2  K2  A2 B2 C2 D2  both
3  K3  A3 B3 C3 D3  both
```

3.12 Plotting Data

The main library for data visualisation in Python is **matplotlib**. It is the basis for most plots but many more specialised libraries exist for e.g. animated plots etc.

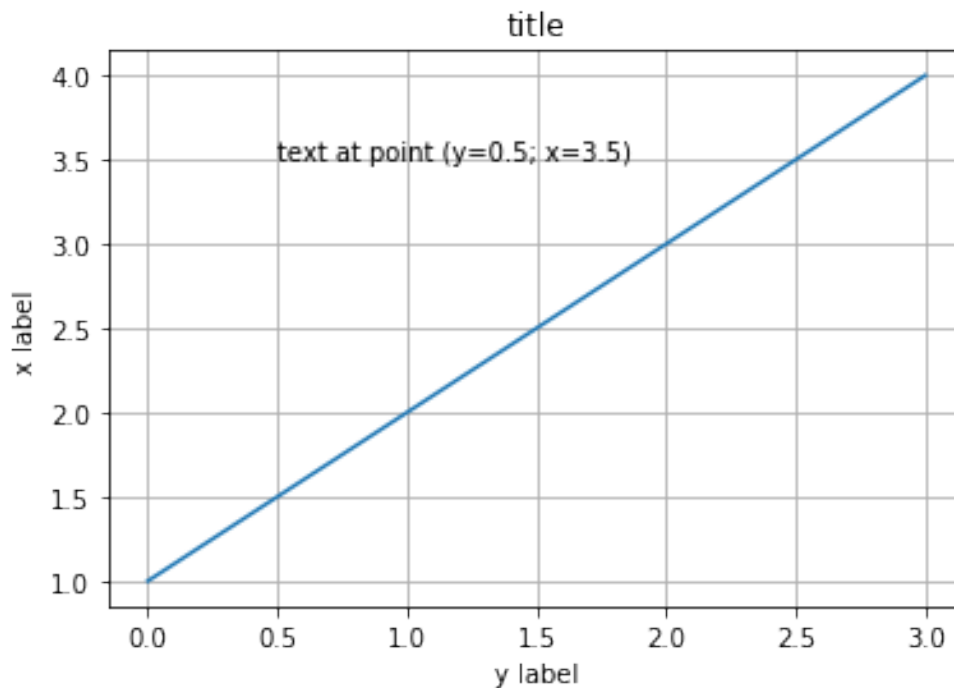
To plot data with matplotlib we use the `plot()` function. There are essentially three blocks of code that are used to design plots: 1. **Data:** This defines the data to be plotted and creates the actual plot object 2. **Formatting:** This allows adding titles, displayed text, and labels 3. **Output:** This calls the plot to be displayed. We can add a line to save the plot to a local folder as well.

```
# Example: Basic plot
import matplotlib.pyplot as plt

# Data
data = [1, 2, 3, 4] ## define data to be plotted
plt.plot(data) ## create the plot

#Formatting
plt.title('title') ## plot title
plt.ylabel('x label') ## label y-axis
plt.xlabel('y label') ## label x-axis
plt.text(0.5, 3.5, 'text at point (y=0.5; x=3.5)') ## adding text
plt.grid() ## adding grid lines

# Output
plt.show() ## show the plot
```



3.13 Formatting Styles

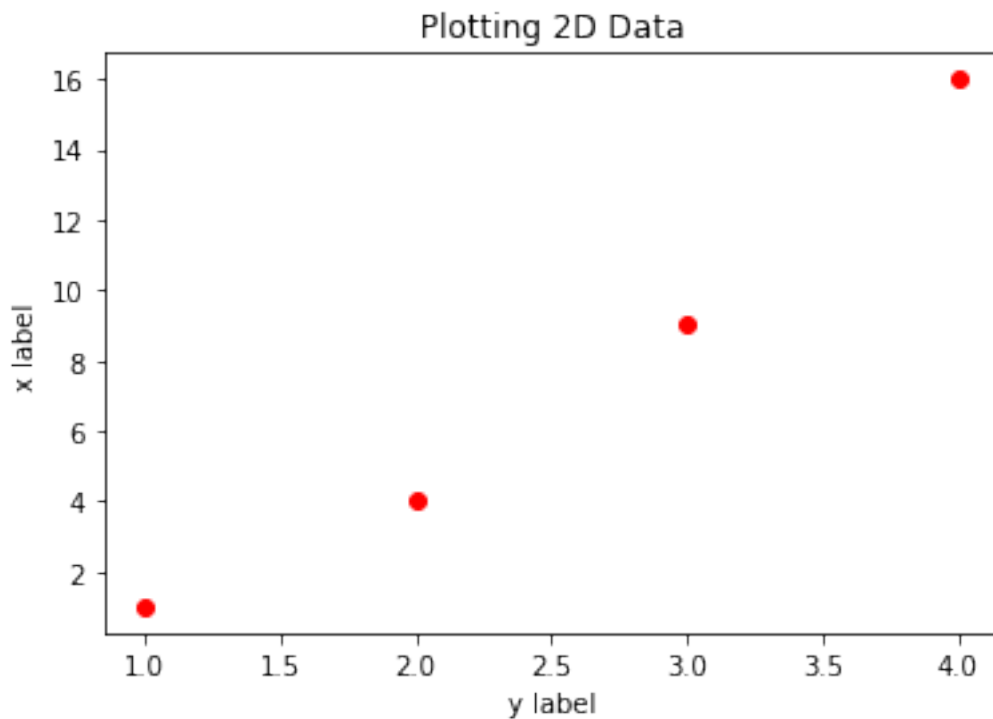
There are formatting options for line styles, markers and colours, that can be combined using the format string (third argument). A complete list of options is available [here](#) on the bottom of the page.

```
# Example: Basic plot
import matplotlib.pyplot as plt

# Data
xdata = [1, 2, 3, 4]
ydata = [1, 4, 9, 16]
# plt.plot(xdata, ydata)
plt.plot(xdata, ydata, 'ro') ## try: 'b', 'r', '--g', '^k:'

#Formatting
plt.title('Plotting 2D Data')
plt.ylabel('x label')
plt.xlabel('y label')

# Output
plt.show()
```



3.14 Plotting Multiple Objects

We can plot multiple objects (lists, functions,...) by simply adding them (including format string) to `plot()`.

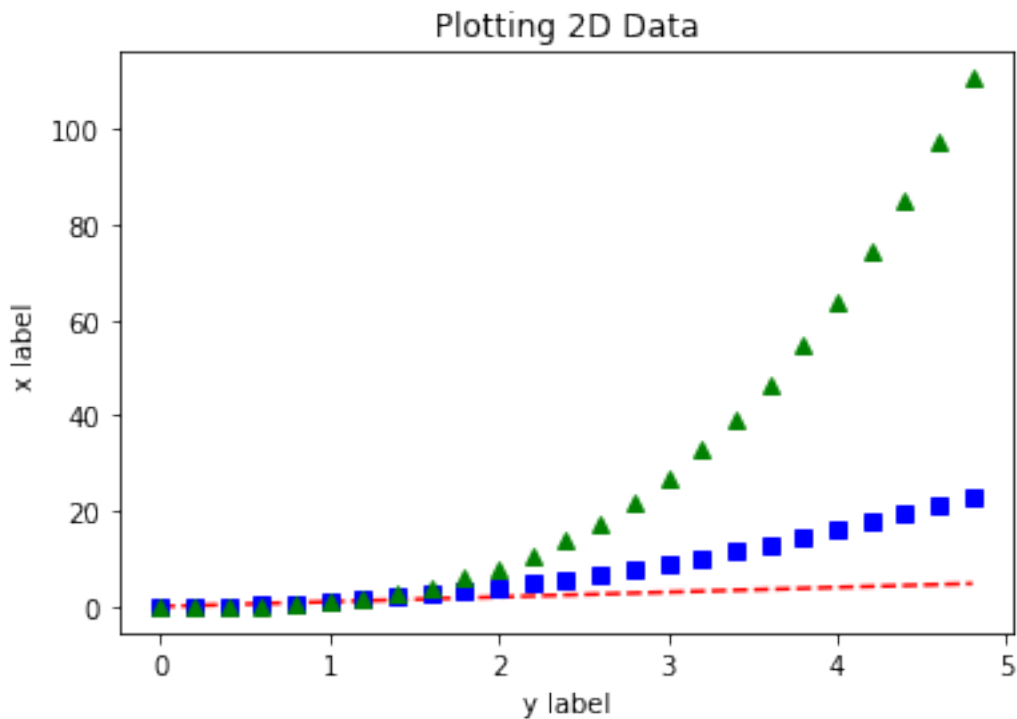
`arrange()` allows adjusting origin on the x-axis, scale on the y-axis, and spacing between markers.

```
import matplotlib.pyplot as plt

t = np.arange(0., 5., 0.2) ## adjust positioning and spacing
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^') ## plotting each series, followed by the format string

plt.title('Plotting 2D Data')
plt.ylabel('x label')
plt.xlabel('y label')

plt.show()
```



3.15 Plotting with Keyword Strings

In some cases (e.g. scatter plots) there are more plotting options, that can be accessed with **keyword strings**. In the following example we use scatter plots.

The syntax is `plot.scatter(var1, var2, c='colorstring', s='shapestring', options)`

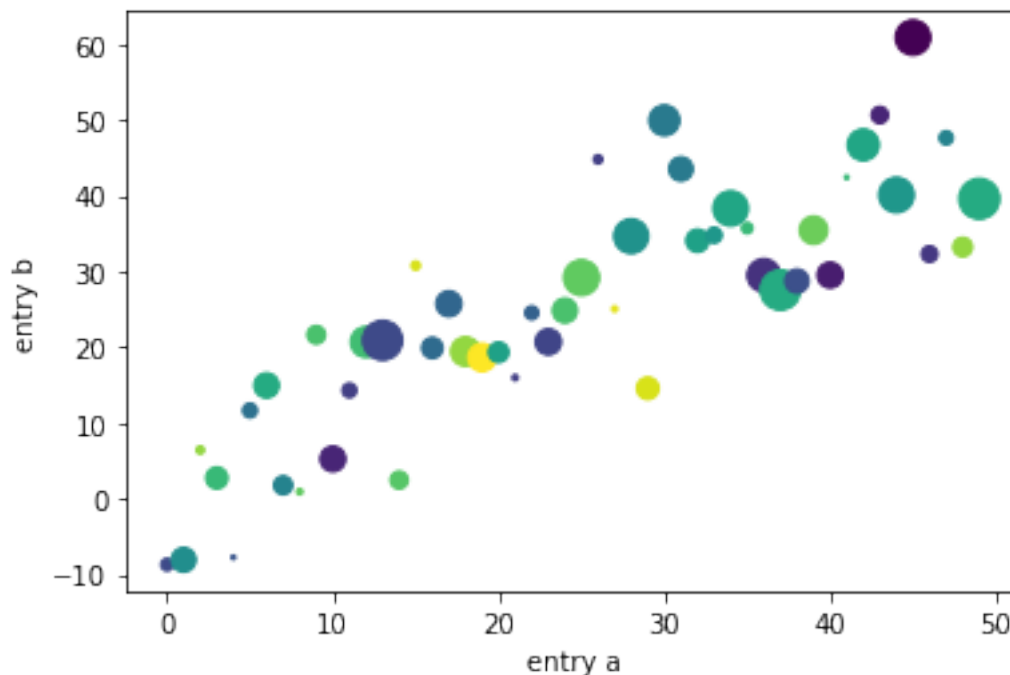
- var1, var2 data to be correlated
- c string value defining colouring
- s string value defining shape

The option `data=data` allows for keyword arguments, defined in the first code block. And `np.random.randint` and `np.random.randn` are random samples drawn from uniform and normal distributions.

```
#Example:
data = {'a': np.arange(50), ## creates 50 evenly spaced values
       'c': np.random.randint(0, 50, 50), ## assigning colours
       'd': np.random.randn(50)} ## assigning marker sizes
data['b'] = data['a'] + 10 * np.random.randn(50) ## defines second variable
↳ to correlate with
data['d'] = np.abs(data['d']) * 100 ## positioning

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')

plt.show()
```



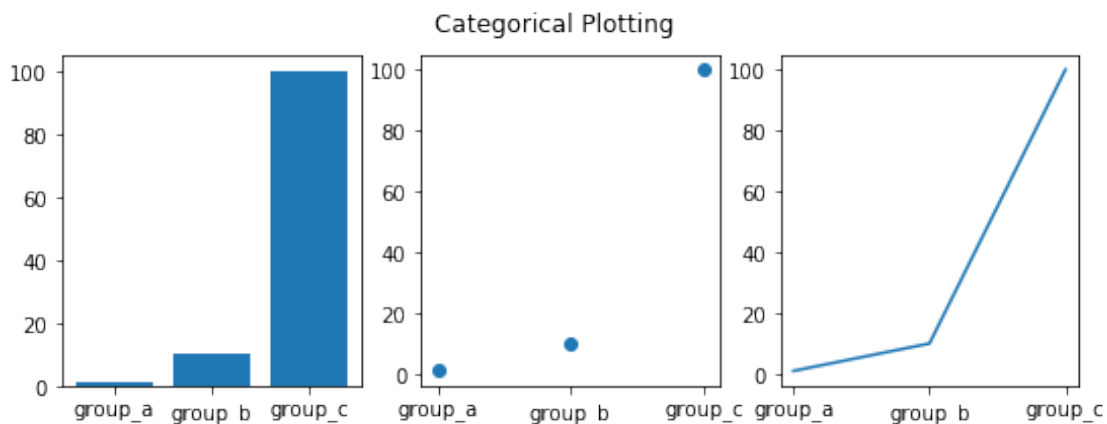
3.16 Categorical Plots

Data can be categorised into groups that can then be plotted. In the following example I've plotted three groups in three ways, creating three subplots:

```
#Example:
names = ['group_a', 'group_b', 'group_c'] ##names of data categories to be
↳ plotted
values = [1, 10, 100] # their values

plt.figure(figsize=(9, 3)) ## defining a figure object to plot subplots in
plt.subplot(131) ##first subplot
plt.bar(names, values) ## specify the plotting function
plt.subplot(132) ## second subplot
plt.scatter(names, values)
plt.subplot(133) ## third subplot
plt.plot(names, values)
plt.suptitle('Categorical Plotting')

plt.show()
```



3.17 More Visualisation

There are countless ways to do data visualisation in Python using matplotlib and other libraries. These are just the very basics.

I've collected some examples of more ideas, including - 3D plots - animated plots

I'm only showing the examples as this is much more than we can introduce in this short workshop. BUT: We are planning on organising a follow-on workshop on data-visualisation. Let us know, should you be interested!

3.18 3D Plots

```
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

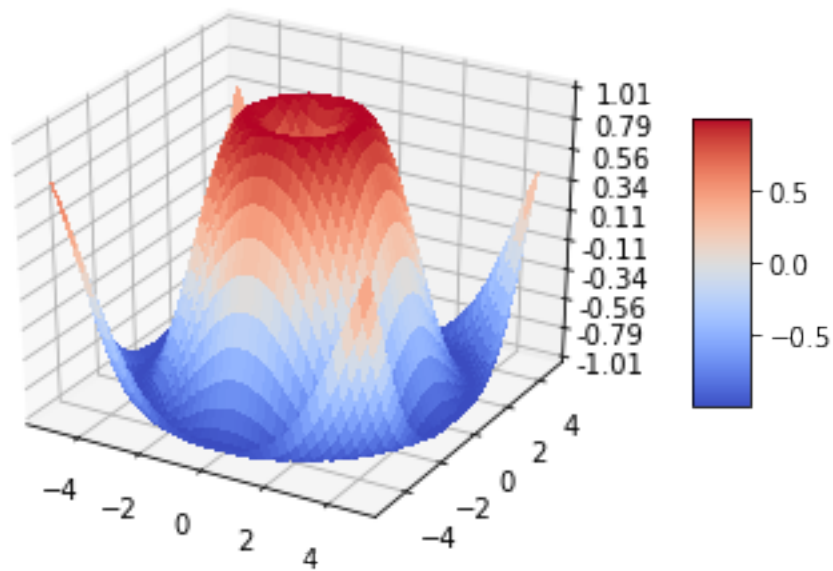
# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

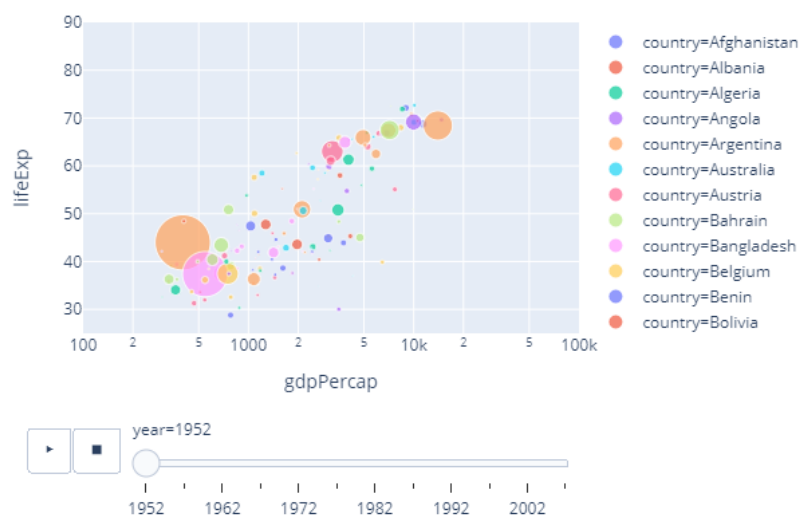
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```



3.19 Animations with plotly express

```
import plotly_express as px
px.scatter(px.data.gapminder(), x="gdpPercap", y="lifeExp",
animation_frame="year", animation_group="country",
size="pop", color="country", hover_name="country",
log_x = True,
size_max=45, range_x=[100,100000], range_y=[25,90])
```



3.20 Networks

```
import plotly.graph_objects as go

import networkx as nx

edge_x = []
edge_y = []
for edge in G.edges():
    x0, y0 = G.nodes[edge[0]]['pos']
    x1, y1 = G.nodes[edge[1]]['pos']
    edge_x.append(x0)
    edge_x.append(x1)
    edge_x.append(None)
    edge_y.append(y0)
    edge_y.append(y1)
    edge_y.append(None)

edge_trace = go.Scatter(
    x=edge_x, y=edge_y,
    line=dict(width=0.5, color='#888'),
    hoverinfo='none',
    mode='lines')

node_x = []
node_y = []
for node in G.nodes():
    x, y = G.nodes[node]['pos']
    node_x.append(x)
    node_y.append(y)

node_trace = go.Scatter(
    x=node_x, y=node_y,
    mode='markers',
    hoverinfo='text',
    marker=dict(
        showscale=True,
        # colorscale options
        #'Greys' | 'YlGnBu' | 'Greens' | 'YlOrRd' | 'Bluered' | 'RdBu' |
        #'Reds' | 'Blues' | 'Picnic' | 'Rainbow' | 'Portland' | 'Jet' |
        #'Hot' | 'Blackbody' | 'Earth' | 'Electric' | 'Viridis' |
        colorscale='YlGnBu',
        reversescale=True,
        color=[],
        size=10,
        colorbar=dict(
            thickness=15,
```

```

title='Node Connections',
xanchor='left',
titleside='right'
),
line_width=2))

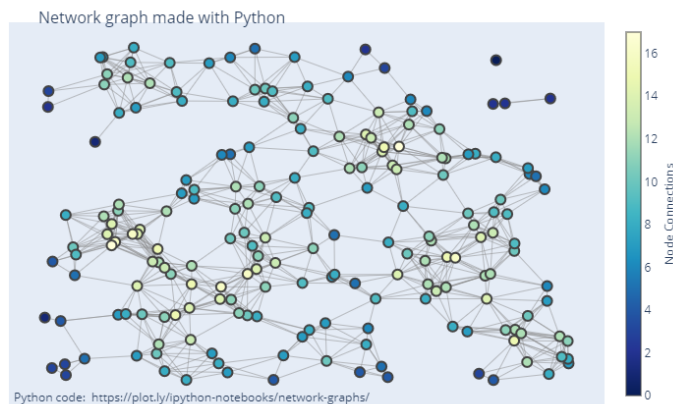
node_adjacencies = []
node_text = []
for node, adjacencies in enumerate(G.adjacency()):
    node_adjacencies.append(len(adjacencies[1]))
    node_text.append('# of connections: '+str(len(adjacencies[1])))

node_trace.marker.color = node_adjacencies
node_trace.text = node_text

fig = go.Figure(data=[edge_trace, node_trace],
    layout=go.Layout(
        title='<br>Network graph made with Python',
        titlefont_size=16,
        showlegend=False,
        hovermode='closest',
        margin=dict(b=20,l=5,r=5,t=40),
        annotations=[ dict(
            text="Python code: <a href='https://plot.ly/ipython-notebooks/network-graphs/'>
↳'> https://plot.ly/ipython-notebooks/network-graphs/</a>",
            showarrow=False,
            xref="paper", yref="paper",
            x=0.005, y=-0.002 ) ],
        xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
        yaxis=dict(showgrid=False, zeroline=False, showticklabels=False))
    )

fig.show()

```



3.21 and much more ...

3.22 More Libraries

A vast amount of other libraries are available, that we will not have time to cover in this workshop. Here are some useful ones

- [Beautiful Soup](#) for web scraping
- [statsmodels](#) for econometrics
- [pyfinance](#) for finance
- [pyfolio](#) for portfolio analysis

APIs

- [GoogleFinance](#) / [YahooFinance](#) for financial data
- [fred](#) wrapper for the FRED database (St Louis Fed)
- [Tweepy](#) for Twitter
- [PyTrends](#) for GoogleTrends and the [Official Google API](#)