Brief introduction..

Data intelligence for the evaluation of new treatments for breast cancer

Quest for a better method to estimate the true impact of implementing innovative breast cancer drugs on the Scottish NHS Secondary Care budgets

Robert Nagy (Year-3 PhD student)

The project is funded by the Breast Cancer Institute

Drive & aspirations

Ultimate goal(s):

1. Maximise my potential via personal & professional development

2. Give back to the community – volunteer for helping out and instructing at various data-related skills-development events

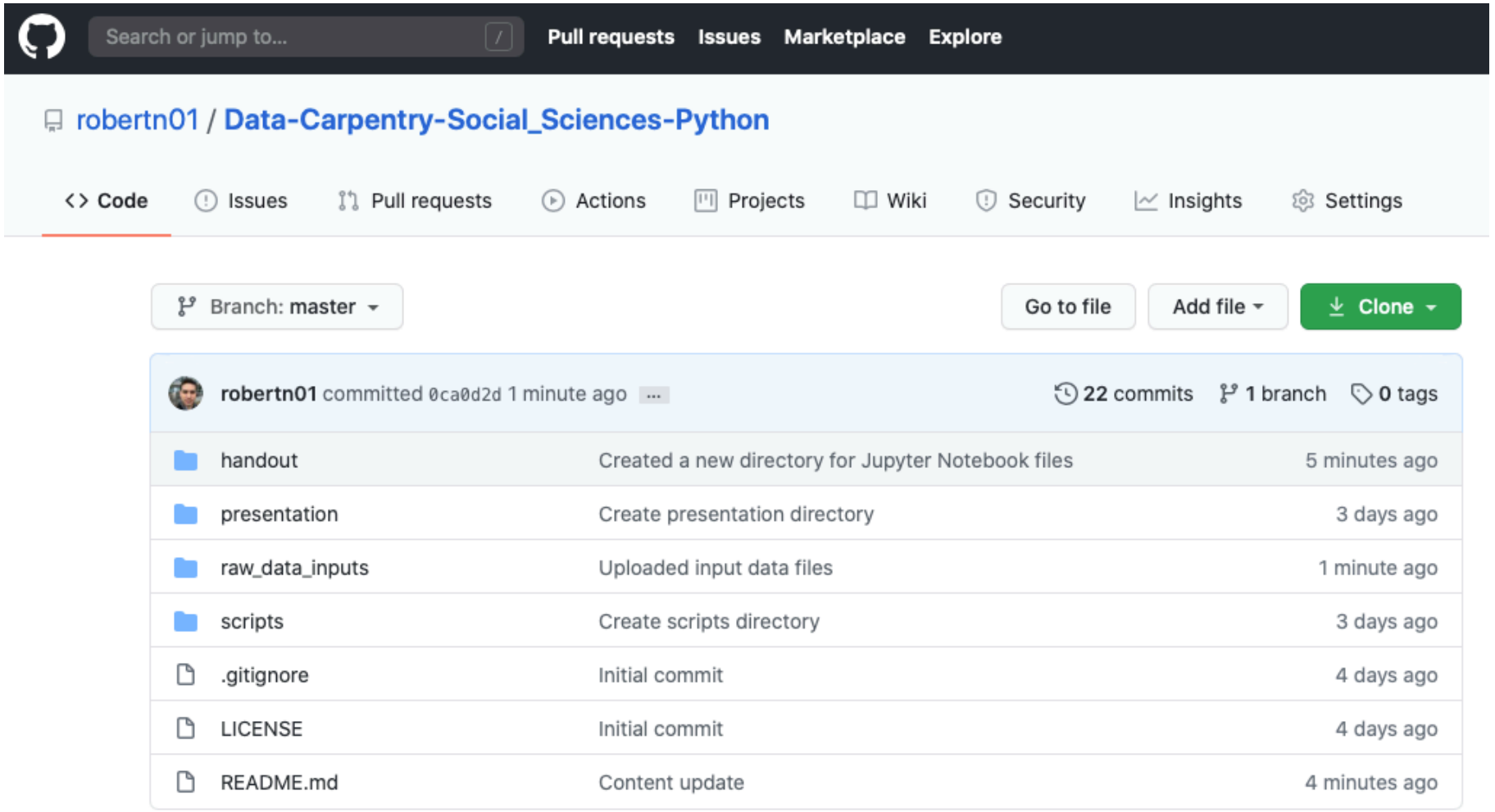
3. Being a pioneer & contribute to data-driven solutions development in a collaborative setting

Before we start… < TROUBLESHOOTING > Python/ Anaconda install & setup

https://swcarpentry.github.io/python-novice-gapminder/setup/

https://github.com/robertn01/Data-Carpentry-Social_Sciences-Python/tree/master

Before we start… < TROUBLESHOOTING > #1 Get the slides



URL: https://github.com/robertn01/Data-Carpentry-Social_Sciences-Python

Before we start… < TROUBLESHOOTING > Python 3.x install & setup

If you already have Python/ Anaconda/ JupyterLab

→ **Check for UPDATES**;

Otherwise… →

https://www.python.org/about/gettingstarted/

Before we start… < TROUBLESHOOTING > Anaconda install & setup

Individual Edition

Your data science toolkit

With over 20 million users worldwide, the open-source Individual Edition (Distribution) is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for solo practitioners, it is the toolkit that equips you to work with thousands of open-source packages and libraries.

Download

ANACONDA.

https://www.anaconda.com/products/individual

Anaconda Installers

Windows	MacOS	Linux
Python 3.7	Python 3.7	Python 3.7
64-Bit Graphical Installer (466 MB)	64-Bit Graphical Installer (442 MB)	64-Bit (x86) Installer (522 MB)
32-Bit Graphical Installer (423 MB)	64-Bit Command Line Installer (430 MB)	64-Bit (Power8 and Power9) Installer (276 MB)

Before we start… < TROUBLESHOOTING > Anaconda/ JupyterLab

***JupyterLab** is a built-in module of **Anaconda Navigator** and is a next-generation web-based user interface for Project Jupyter.*

https://jupyterlab.readthedocs.io/en/stable/getting_started/overview.html

Starting JupyterLab:

https://jupyterlab.readthedocs.io/en/stable/getting_started/starting.html

Installation separately from Anaconda:

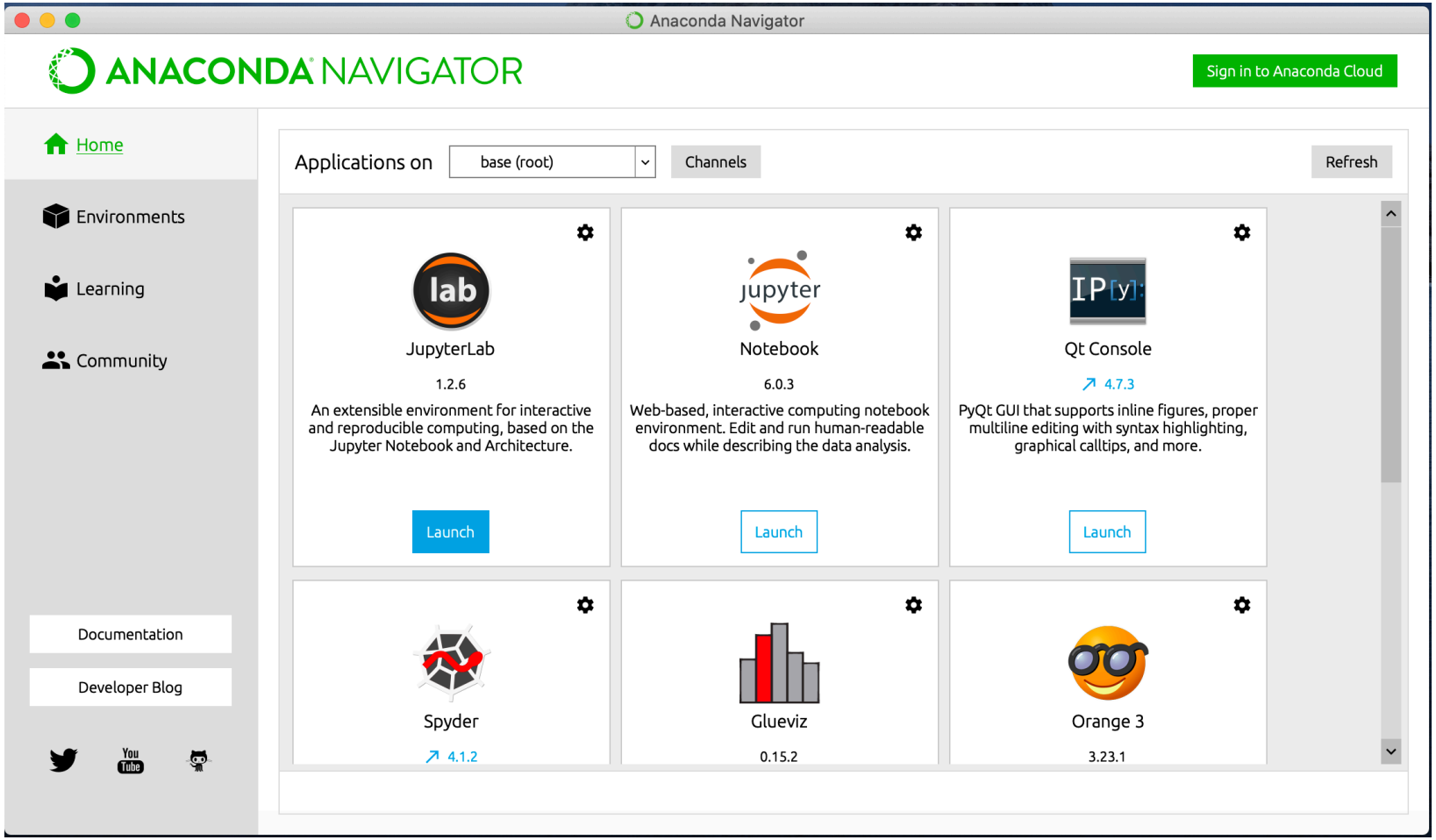
- https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html

OR via JupyterHub

- https://jupyterlab.readthedocs.io/en/stable/user/jupyterhub.html#jupyterhub

Before we start… < TROUBLESHOOTING > Anaconda/ JupyterLab

[*illustration*: launching Anaconda-Navigator on a Mac]



Download and unzip the data to a preferred working directory

https://swcarpentry.github.io/python-novice-gapminder/files/python-novice-gapminder-data.zip
[compressed files]

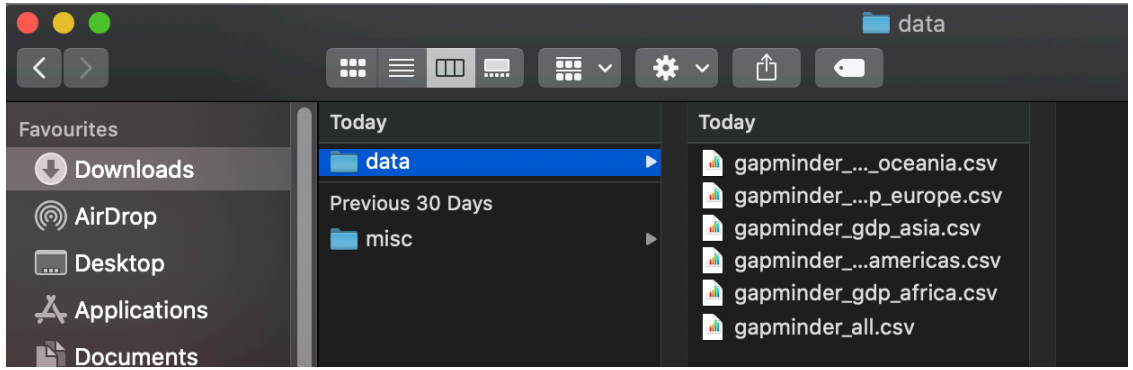
https://github.com/robertn01/Data-Carpentry-Social_Sciences-Python [direct download]

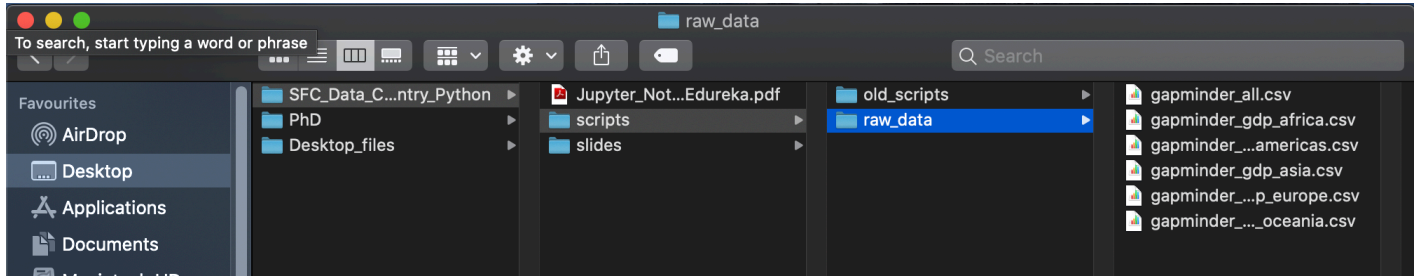
Brief

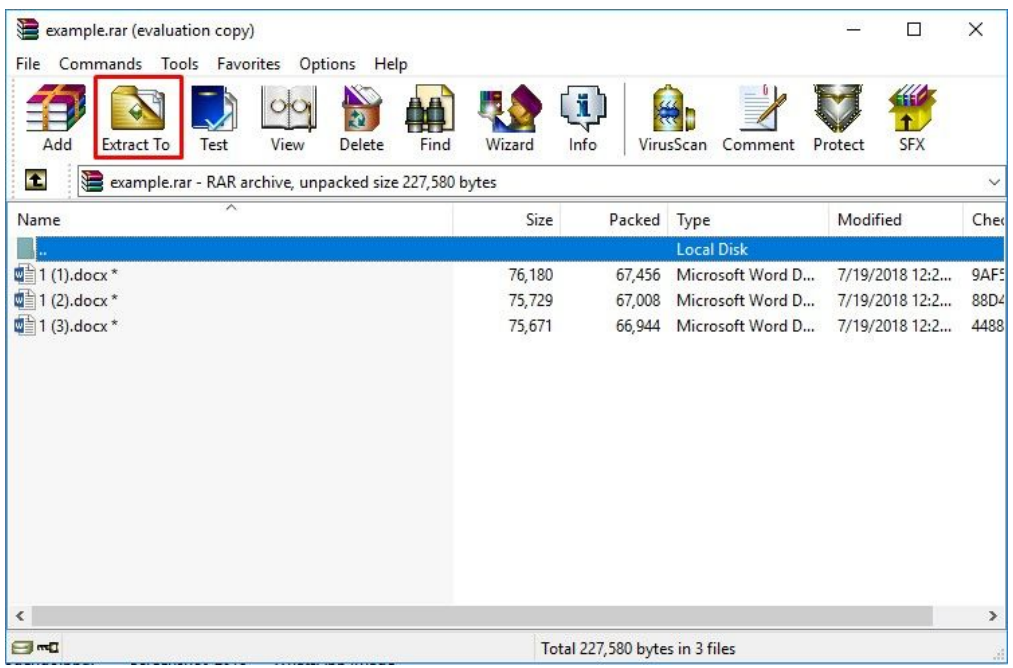
"Gapminder's stated mission is *'Fighting devastating ignorance with fact-based worldviews everyone can understand*."

[Source: https://en.wikipedia.org/wiki/Gapminder_Foundation]

Before we start… < TROUBLESHOOTING > Get the data ready…



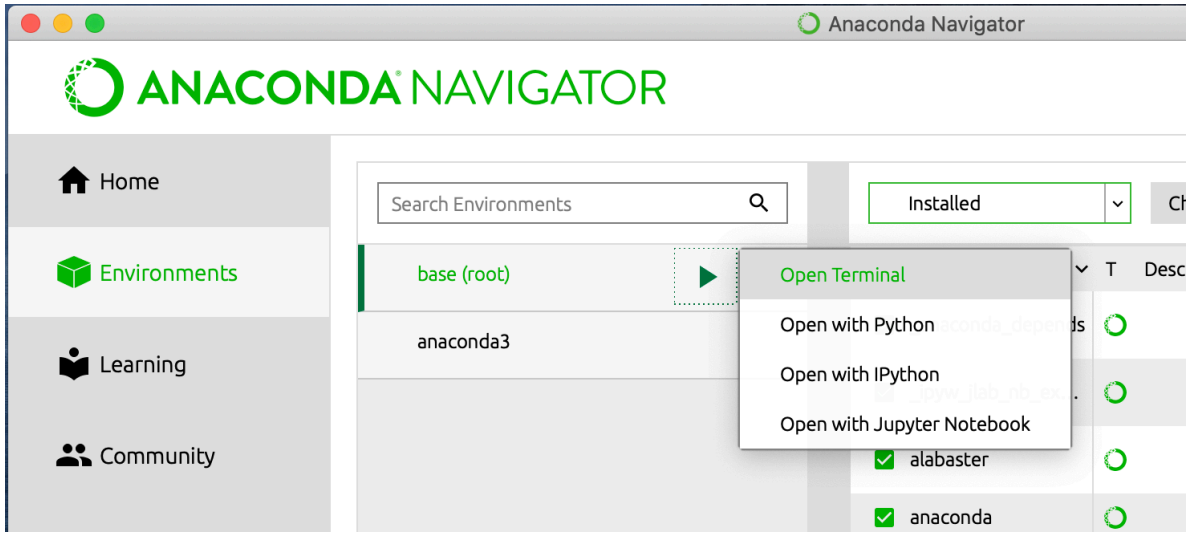


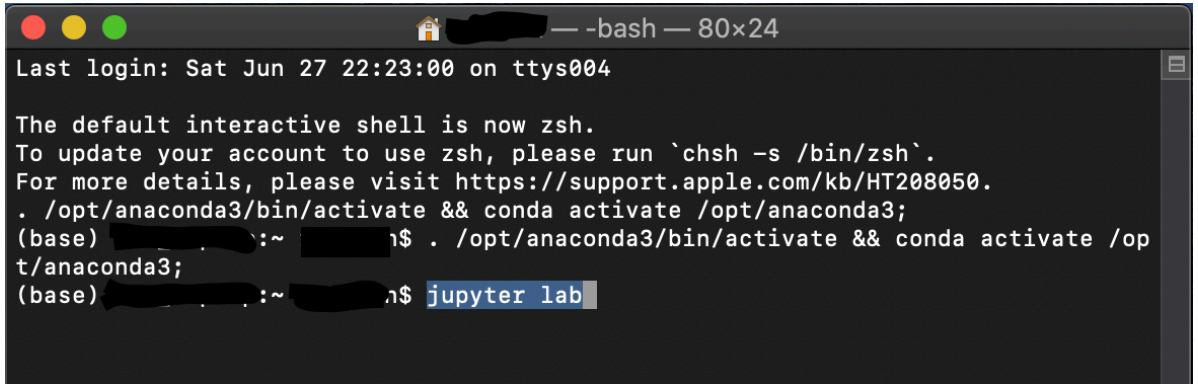


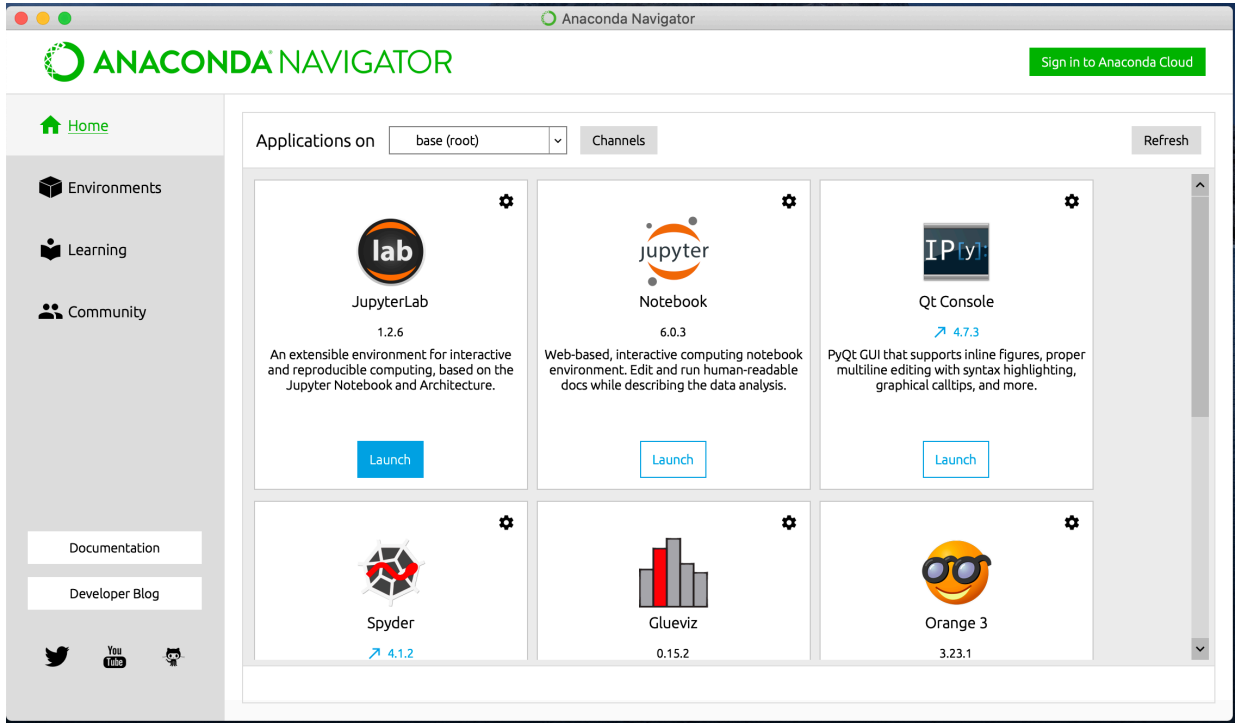
1. Download compressed files from online repository
2. Decompress (e.g. unzip) file(s) into preferred working directory

Before we start… < TROUBLESHOOTING > launch JupyterLab

[*illustration*: launching JupyterLab on a Mac* – *Left*: from Anaconda Terminal; *Right*: from Anaconda GUI]

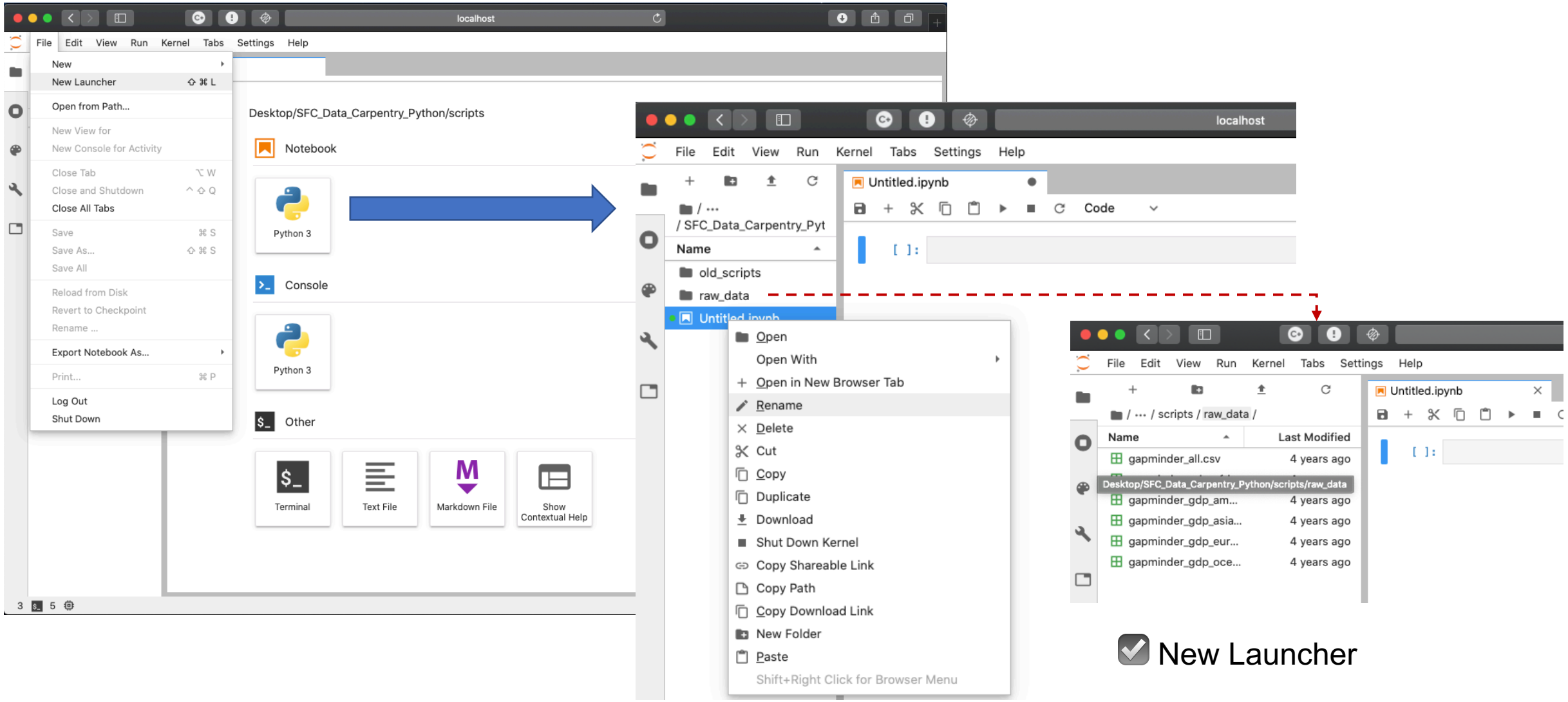






[*Similar procedure for Windows users]

Before we start… < TROUBLESHOOTING > launch JupyterLab



- [x] New Launcher
- [x] Data files pre-loaded

Before we start... Some important keyboard shortcuts/ cheat sheets

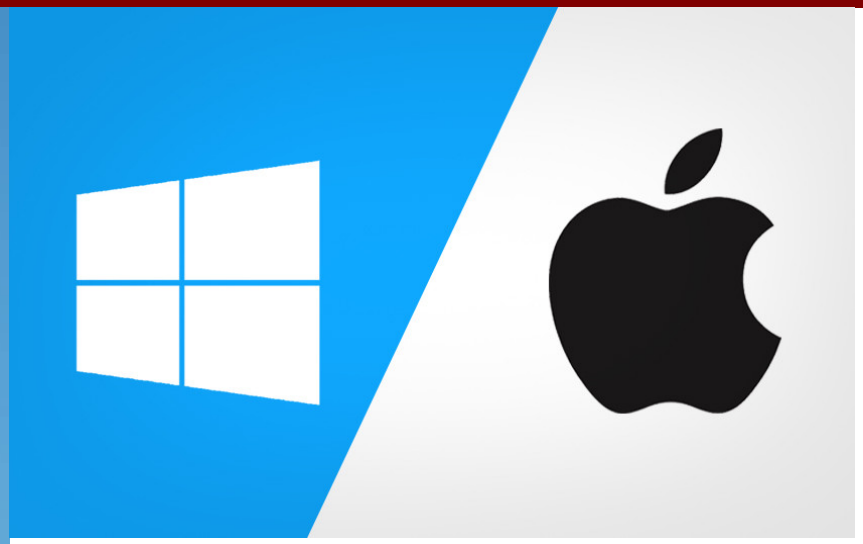
Jupyter Notebook Keyboard Shortcuts

https://cheatography.com/weidadeyue/cheat-sheets/jupyter-notebook/pdf_bw/

Jupyter Notebook Cheatsheet

https://www.edureka.co/blog/wp-content/uploads/2018/10/Jupyter_Notebook_CheatSheet_Edureka.pdf

Before we start… **Please be aware..**



Anaconda-Navigator/ JupyterLab could behave slightly differently when running on different OS (e.g. Windows vs UNIX [e.g. Mac or Linux] machines) and even on the same OS different Python software and library versions can ‘surprise’ us.

DATA CARPENTRY

[*alpha* version]

DAY 1
Welcome!

Scottish Funding Council - Data Skills Workforce Development

Data Carpentry for Digital Humanities Curriculum

Introduction to good data practices using **Python/ JupyterLab**

Demonstrator: Robert Nagy

(PhD student)

The University of Edinburgh – Cancer Research UK Edinburgh Centre
+ Western General Hospital - Edinburgh Cancer Centre
+ Edinburgh BioQuarter – Edinburgh Health Economics Team

[**web:** https://www.ed.ac.uk/profile/robert-nagy]

robert.nagy@ed.ac.uk

Outline for Python sessions & Schedule for the rest of the Workshop

Episodes

- ***1. Running and Quitting***
    - *How can I run Python programs?*
- ***2. Variables and Assignment***
    - *How can I store data in programs?*
- ***3. Data Types and Type Conversion***
    - *What kinds of data do programs store?*
    - *How can I convert one type to another?*
- ***4. Built-in Functions and Help***
    - *How can I use built-in functions?*
    - *How can I find out what they do?*
    - *What kind of errors can occur in programs?*
- ***6. Libraries***
    - *How can I use software that other people have written?*
    - *How can I find out what that software does?*
- ***7. Reading Tabular Data into DataFrames***
    - How can I read tabular data?

Day 1 (Tuesday)

14:30-14:45 *Coffee break 2*
14:45-15:45 Python
15:45-16:00 *Coffee break 3*
16:00-17:00 Python

Day 2 (Wednesday)

9:00-10:30 Python
10:30-11:00 *Coffee break 1*
11:00-12:30 Python
12:30-13:30 *Lunch break*
13:30-15:00 Python
15:00-15:30 *Coffee break 2*
15:30 -17:00 Python

Useful URLs

Lessons

https://swcarpentry.github.io/python-novice-gapminder/

Etherpad (AKA the 'Pad') – ALL the essential URLs are there!

https://pad.carpentries.org/2020-07-01-sfc-online

Download the datasets

https://swcarpentry.github.io/python-novice-gapminder/files/python-novice-gapminder-data.zip

Slides, Python scripts → Navigate to my GitHub page..

https://github.com/robertn01/Data-Carpentry-Social_Sciences-Python

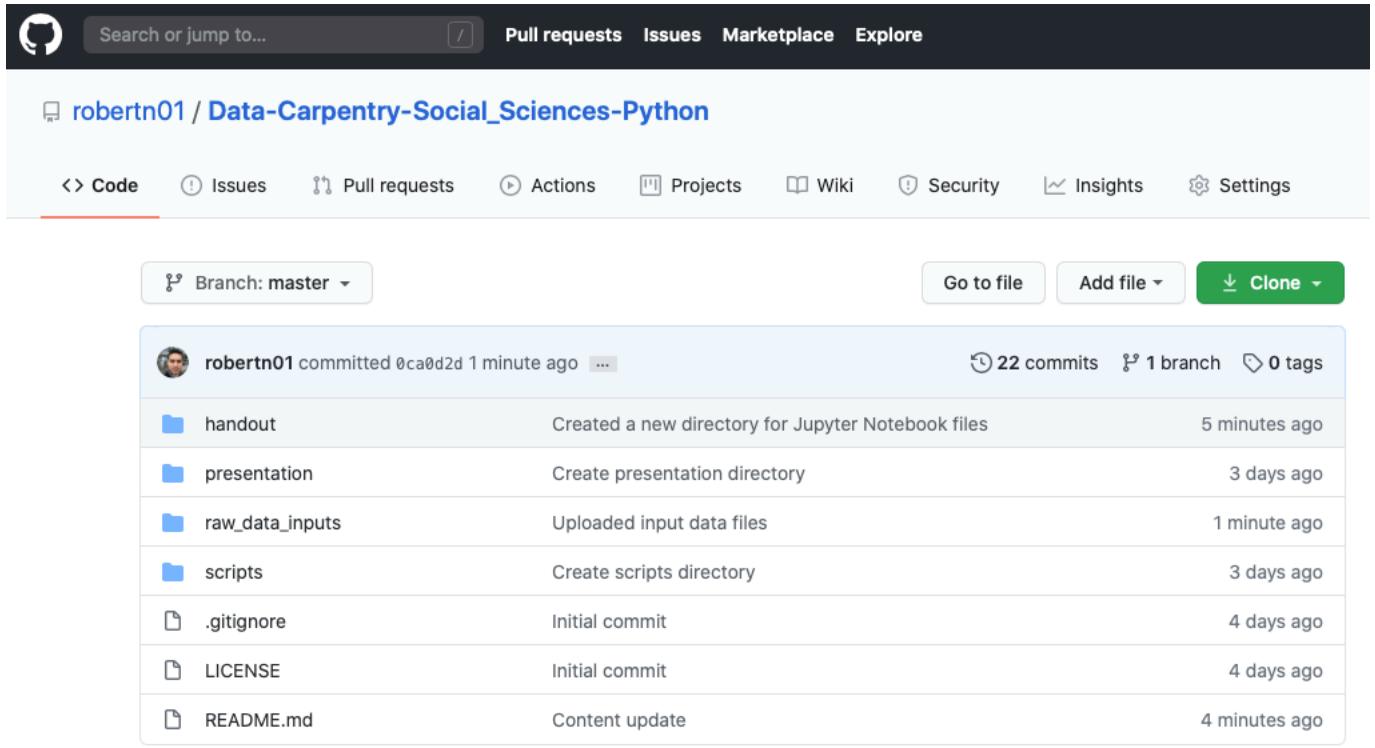
Data literacy and skills

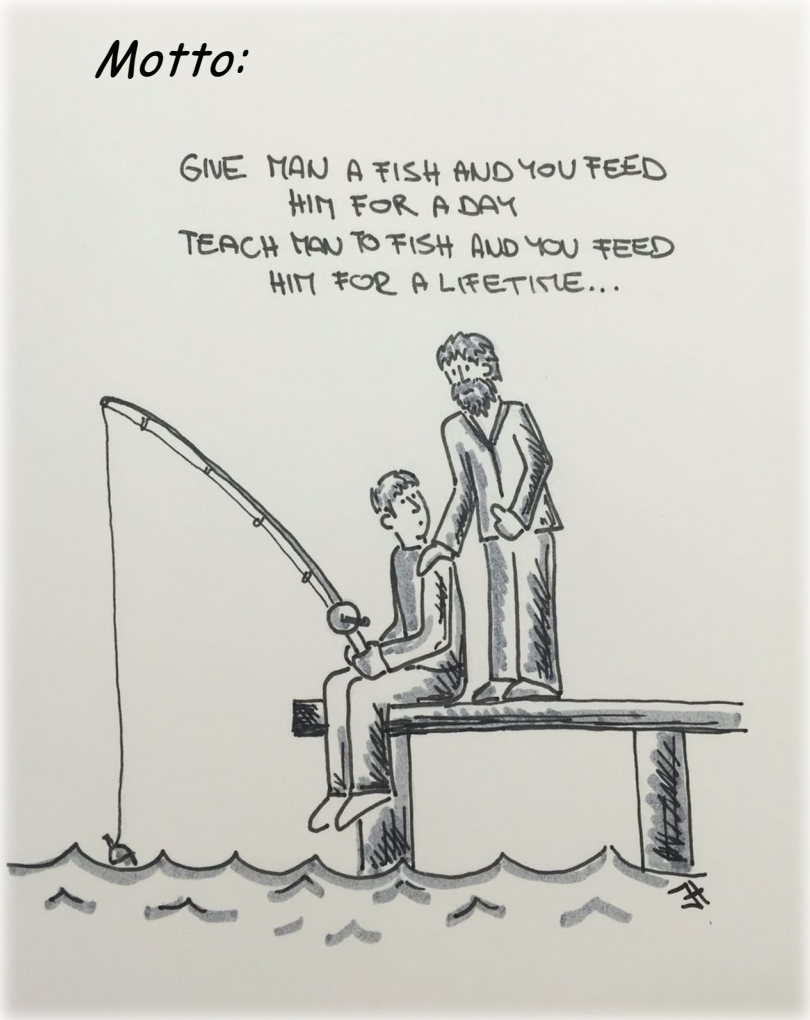
"Data literacy is the ability to read, work with, analyse, and argue with data.

Much like literacy as a general concept, data literacy focuses on the competencies involved in working with data."

[Source: *Wikipedia*]s

Presentation slides and handouts.. available via GitHub

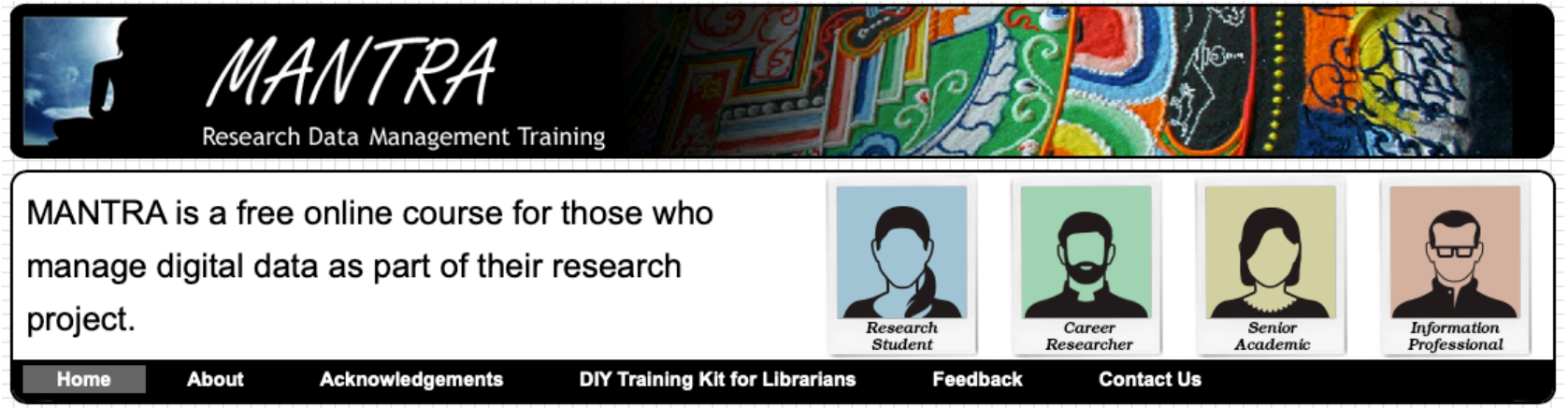




In your web browser navigate to:

https://github.com/robertn01/Data-Carpentry-Social_Sciences-Python

Fundamental concepts: '*data*' and '*research data*'



https://mantra.edina.ac.uk

What are data?

So what do we mean when we talk about data and research data?

- Data and information are closely related concepts. Data provide the foundation from which knowledge is derived, whereby data become information in the process of being analysed.

- Data are most commonly understood at the level at which measures were originally collected, and may be qualitative or quantitative, factual or non-factual, numerical, textual or audio-visual.

Fundamental concepts: '*data*' and '*research data*' [*TL:DR*]

What are research data?

Research data are collected, observed or generated for the purpose of analysis, to produce and validate original research results.

Primary data are data that are collected for a particular purpose (e.g. to answer particular research questions). Secondary data are data collected for one purpose that are made available for reuse by others for a different purpose.

Research data are collected across a range of disciplines using a variety of research approaches and methods.

Further, the word data means different things to different people in different contexts. For example, different disciplines may adopt discipline-specific language around the subject research data.

Some people refer to everything digital as data. Others refer to both analogue and digital materials as data. For the purposes of this course, research data will refer to data created in a digital form (born digital) or converted to a digital form (digitised).

In summary, research data refers to whatever is necessary to verify or reproduce research findings, or to gain a richer understanding of them.

Fundamental concepts: '*data*' and '*research data*'

Research records

In addition to research data, during a research project you will also need to think about how to manage the ***accompanying research records*** *- that is administrative materials and supporting documentation - both during and beyond the life of a project. For instance:*

- *Correspondence (electronic mail and paper-based correspondence)*
- *Grant applications*
- *Ethics applications*
- *Technical reports*
- *Technical appendices*
- *Research reports*
- *Research publications*
- *Signed consent forms*
- *Social media communications (blogs, wikis, tweets, etc.)*

Fundamental concepts: '*data*' and '*research data*'

Research data can be generated for different purposes and through different processes in a multitude of digital formats.

The following classification, compiled by the *Research Information Network*, gives one way to categorise research data according to the different contexts in which they are created:

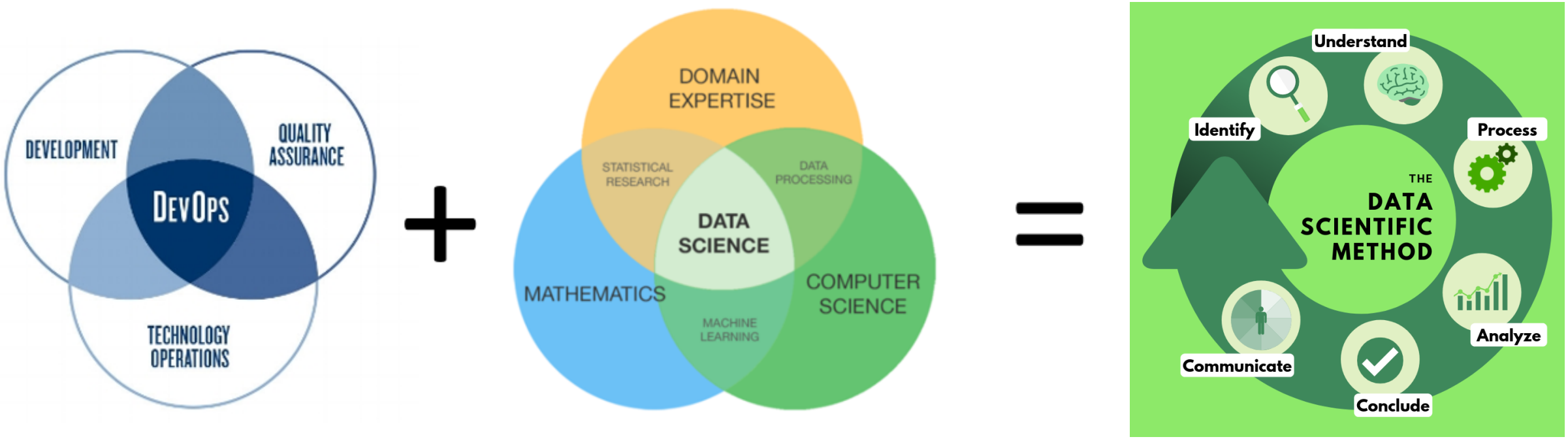
- **Observational**: data captured in real time, usually unique and irreplaceable.
- **Experimental**: data from experimental results, e.g. from lab equipment, often reproducible, but can be expensive.
- **Simulation**: data generated from test models where model and metadata may be more important than output data from the model.
- **Derived or compiled**: resulting from processing or combining 'raw' data, often reproducible but expensive.
- **Reference or canonical**: a (static or organic) conglomeration or collection of smaller (peer reviewed) datasets, most probably published and curated.

Fundamental concepts: *What is 'Data Science'?*

Data Science …

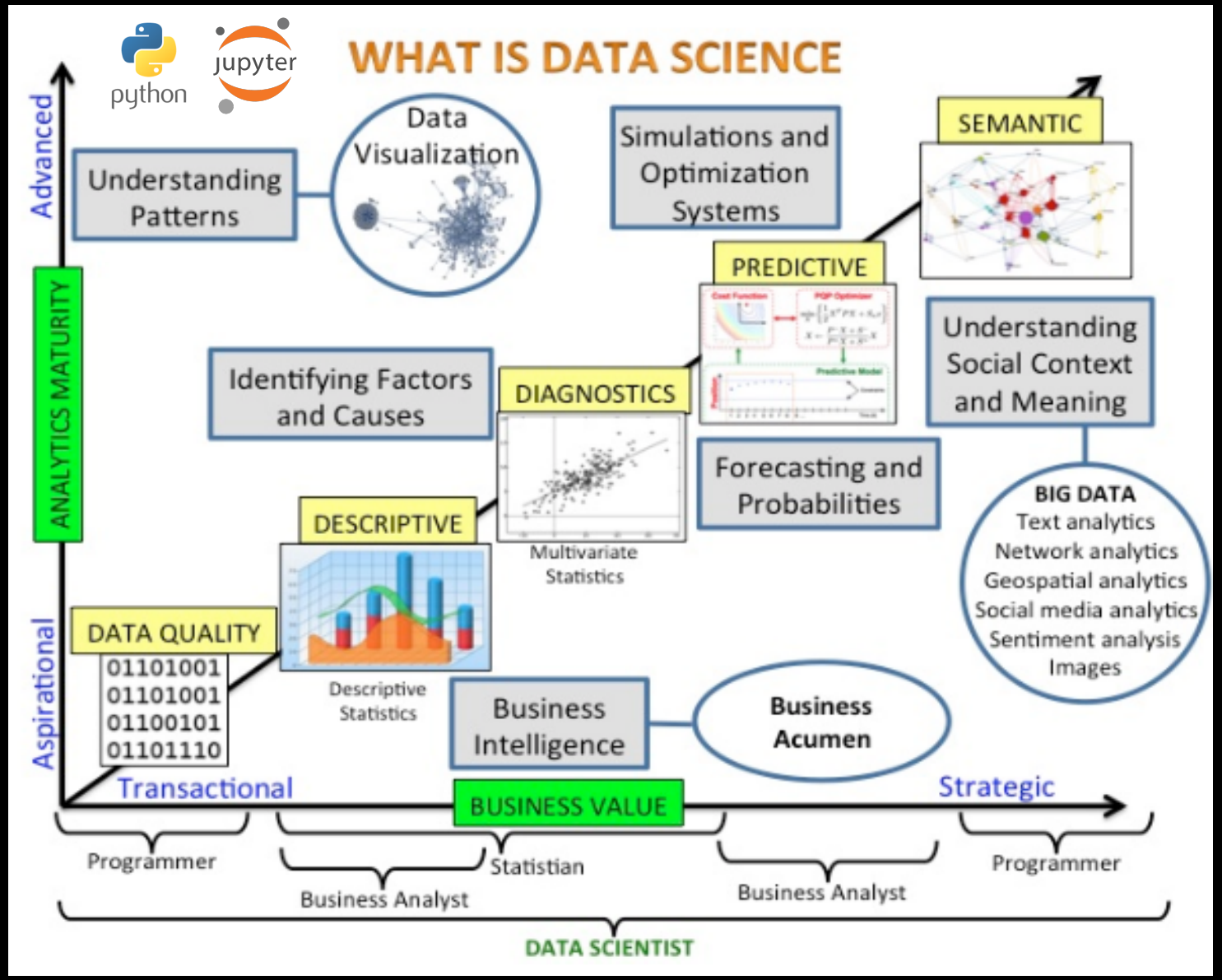
… a **process** starting with formulating question that can be ***answered with data***, and **iteratively collecting**, **cleaning**, **analysing** and **modelling** it, while **communicating** the answers to the relevant audience along this iteration ….

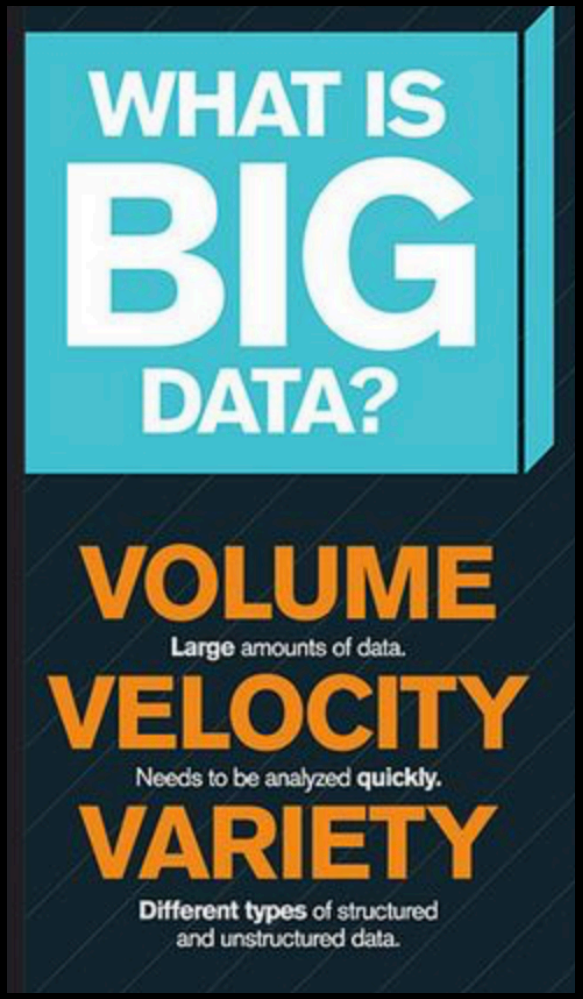
‘Ontology of Data Science’ (the boarder perspective..)



[source: https://towardsdatascience.com]

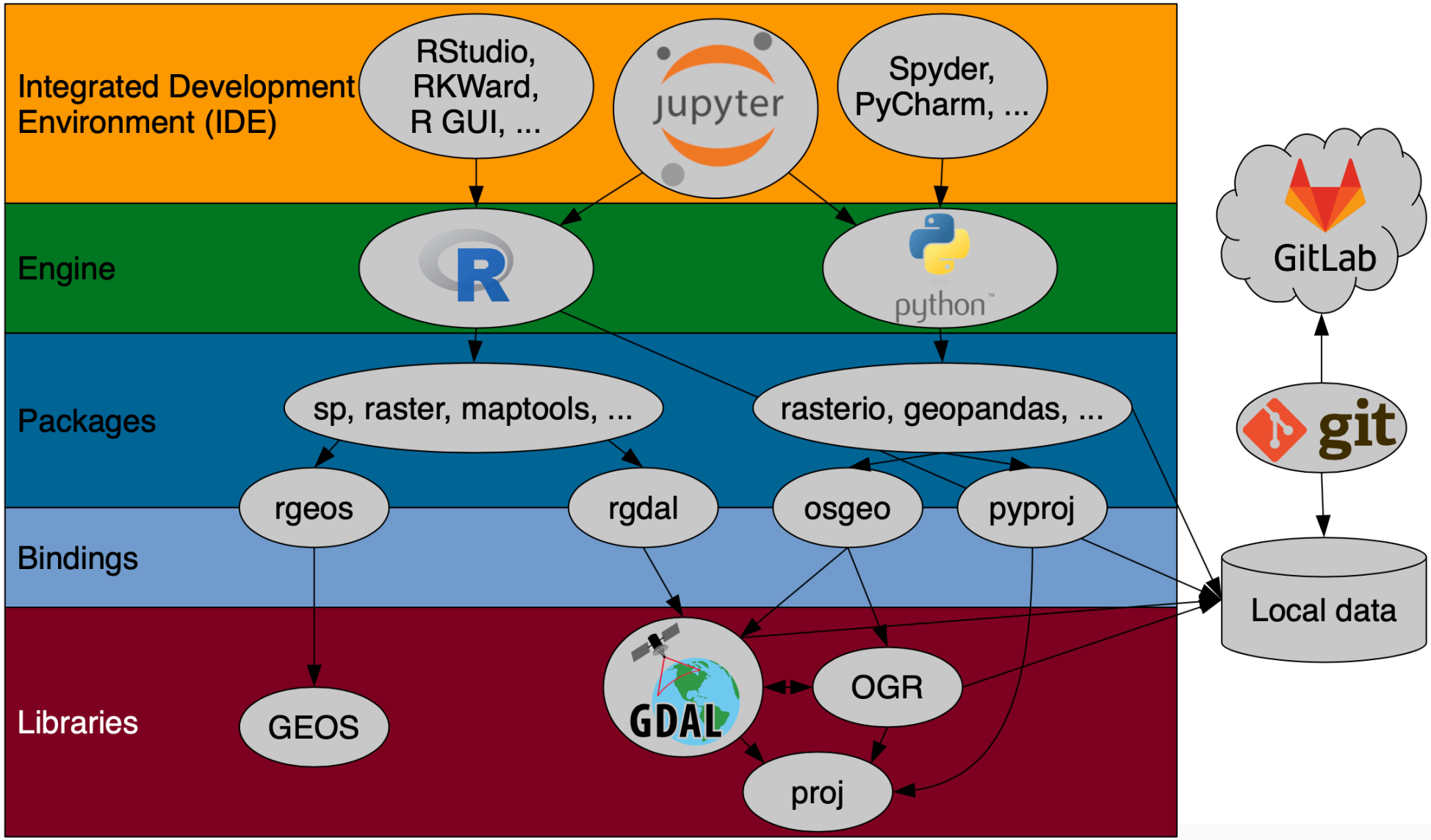
[**Keywords**: formal ontology; hermeneutic cycle; epistemology; knowledge/ data/ information re-cycling]





[Source: https://www.datasciencecentral.com/profiles/blogs/data-science-summarized-in-one-picture]

Systemic overview [an illustrative example]



The Python Code Style Guide & Beginner's Guide

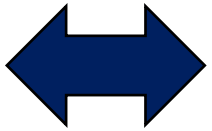
PEP 8 -- Style Guide for Python Code

https://www.python.org/dev/peps/pep-0008/

https://wiki.python.org/moin/BeginnersGuide

Why Python …?

A complex decision: guarantee, reliability, standardisation, price, flexibility, …etc. considerations



Python can deliver the whole Data Science workflow

- Raw data collection, preparation and formatting (e.g. Spreadsheets)
- Data cleansing and management (e.g. OpenRefine)
- Data wrangling, cleansing, analysis, visualization and publishing (e.g. R, Python, SQL)

Cheat sheets.. :)

Cheat sheets

- http://get.treasuredata.com/rs/714-XIJ-402/images/TD_Jupyter%20Notebook%20Cheatsheet_V1%281%29%20%281%29.pdf
- https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Jupyter_Notebook_Cheat_Sheet.pdf
- https://www.dataquest.io/blog/pandas-cheat-sheet/
- https://www.dataquest.io/blog/python-cheat-sheet/

…etc.

For now, everyone should have a fully operational ***Anaconda Navigator*** / ***JupyterLab*** environment on board and having Python 3.x installed.

Episode #1 – '**Running & Quitting**'

https://swcarpentry.github.io/python-novice-gapminder/01-run-quit/index.html

Learning Objectives

- Launch the *JupyterLab* server.
- Create a new *Python script*.
- Create a *Jupyter Notebook*.
- Understand the difference between a plain Python script and a Jupyter Notebook file.
- Create and run Python cells in a notebook.
- Create *Markdown cells* in a notebook.
- Shutdown the JupyterLab server.

Episode #1 – We'll going to use **JupyterLab..**

JupyterLab is an application with a web-based user interface (*graphical user interface AKA GUI*) from Project Jupyter that enables one to work with documents and activities such as Jupyter notebooks, text editors, terminals, and even custom components in a flexible, integrated, and extensible manner.

JupyterLab requires a reasonably up-to-date browser (ideally a current version of Chrome, Safari, or Firefox); Internet Explorer versions 9 and below are *not* supported.

JupyterLab is included as part of the Anaconda Python distribution (Check *Anaconda-Navigator*).

JupyterLab runs locally on your machine and does not require an internet connection by default.

- *The JupyterLab server sends messages to your web browser.*
- *The JupyterLab server [running on local machine] does the work and the web browser renders the result.*
- *You will type code into the browser and see the result when the web page talks to the JupyterLab server.*

Episode #1 – Starting with **JupyterLab..**

Mac OS X

To start the JupyterLab server you will need to access the command line through the Terminal. There are two ways to open Terminal on Mac.

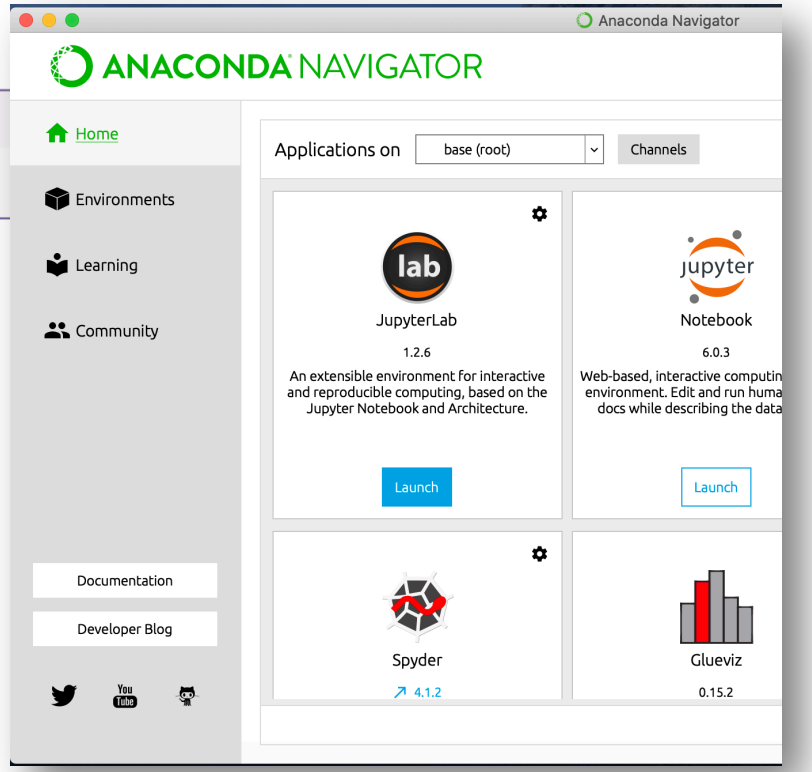
1. In your Applications folder, open Utilities and double-click on Terminal
2. Press Command + spacebar to launch Spotlight. Type `Terminal` and then double-click the search result or hit Enter

After you have launched Terminal, type the command to launch the JupyterLab server.

Bash

```
$ jupyter lab
```

OR use the GUI



Windows Users

To start the JupyterLab server you will need to access the open Anaconda Prompt.

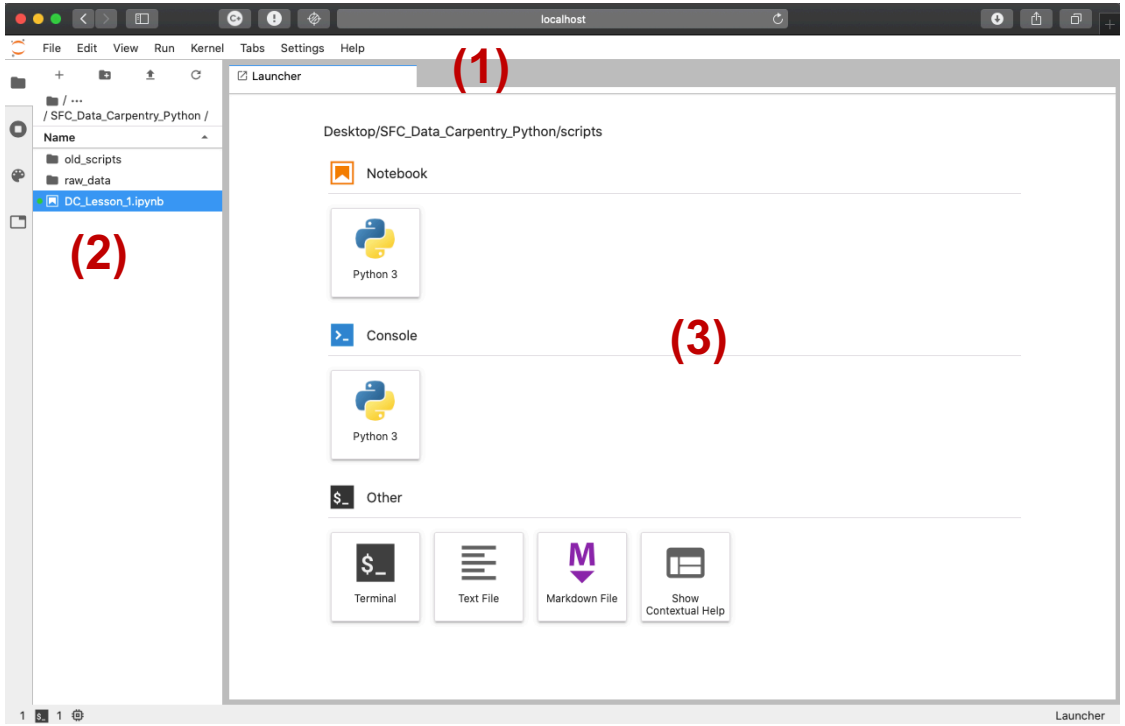
Press Windows Logo Key and search for `Anaconda Prompt`, click the result or press enter.

After you have launched the Anaconda Prompt, type the command:

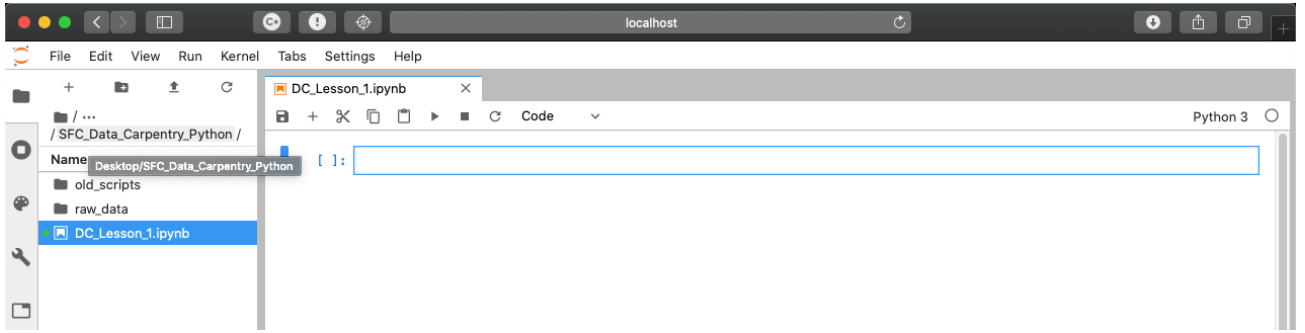
Bash

```
$ jupyter lab
```

Lesson #1 – The **JupyterLab** *user interface (GUI)*..



(3) Drag a tab to the centre of a tab panel to move the tab to the panel. Subdivide a tab panel by dragging a tab to the left, right, top, or bottom of the panel. The work area has a single current activity. The tab for the current activity is marked with a coloured top border (blue by default).



JupyterLab has many features found in traditional integrated development environments (IDEs) but is focused on providing flexible building blocks for interactive, exploratory computing.

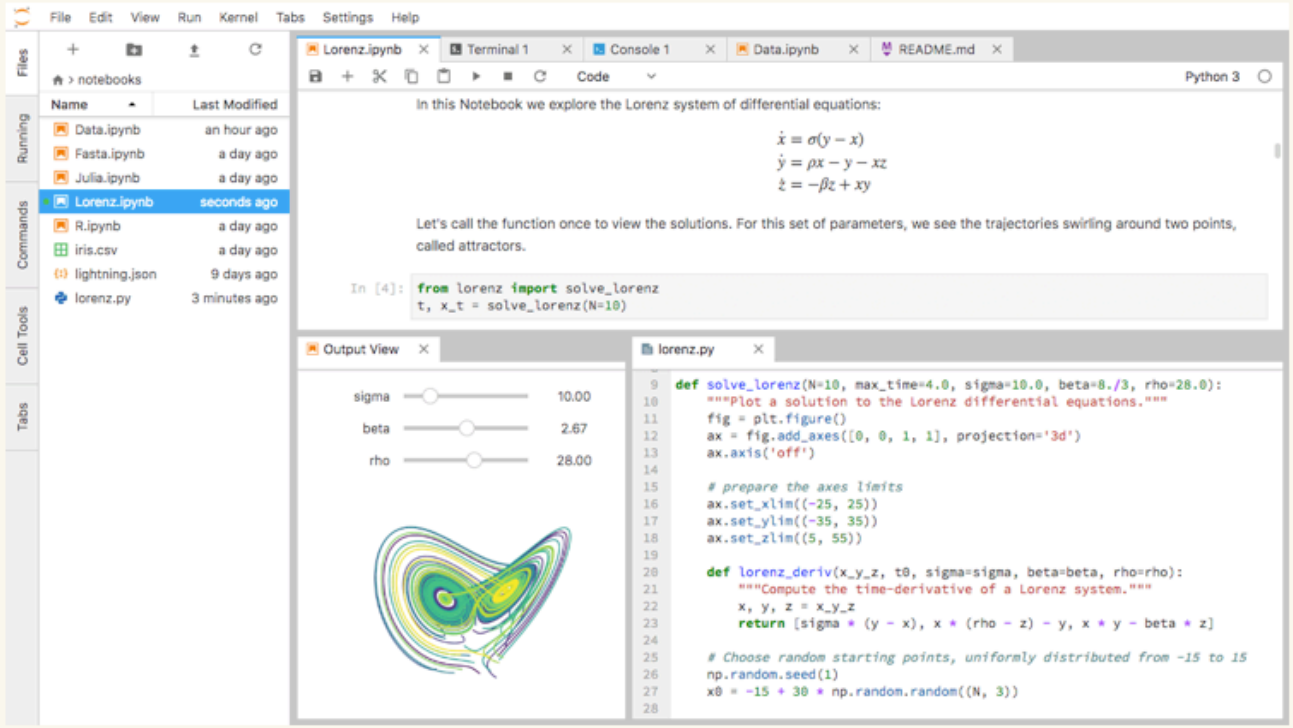
The JupyterLab Interface consists of

- the **Menu Bar (1)**,
- a collapsible **Left Side Bar (2)**,
- and the **Main Work Area (3)** - which contains tabs of documents and activities.

Episode #1 – **JupyterLab – Finger Exercise 1.1** (~ 1 minute)

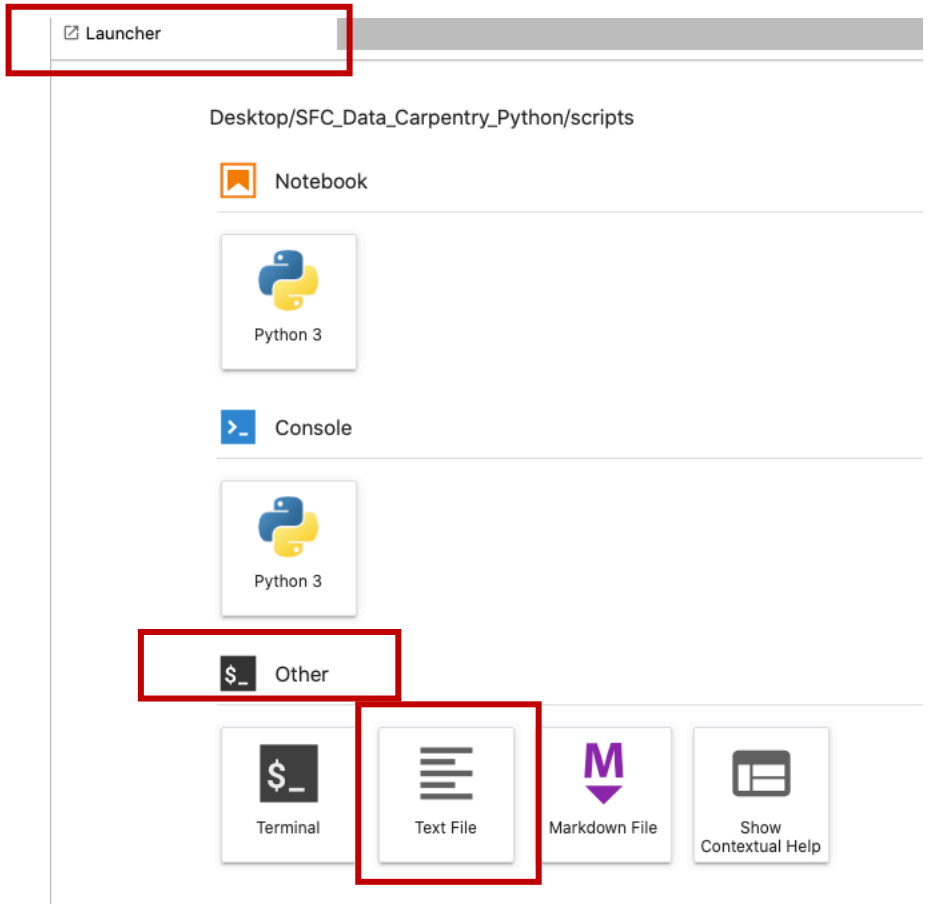
Arranging Documents into Panels of Tabs

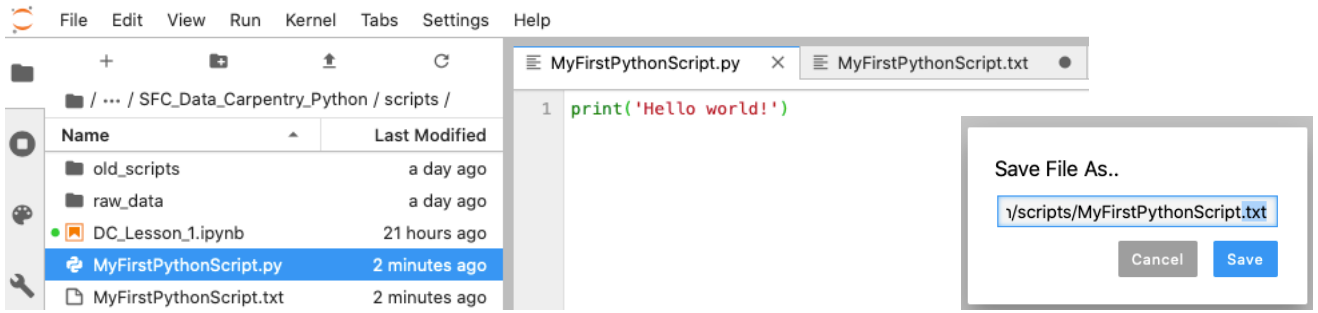
In the JupyterLab Main Work Area you can arrange documents into panels of tabs. Here is an example from the official documentation.



First, create a text file, Python console, and terminal window and arrange them into three panels in the main work area. Next, create a notebook, terminal window, and text file and arrange them into three panels in the main work area. Finally, create your own combination of panels and tabs. What combination of panels and tabs do you think will be most useful for your workflow?

Episode #1 – Creating a *Python* script (*file_name.py*) in **JupyterLab**





• **To start writing a new Python script** click the ***Text File*** icon under the *Other* header in the **Launcher tab** of the **Main Work Area**.

- You can also create a new plain text file by selecting the *New -> Text File* from the *File* menu in the Menu Bar.

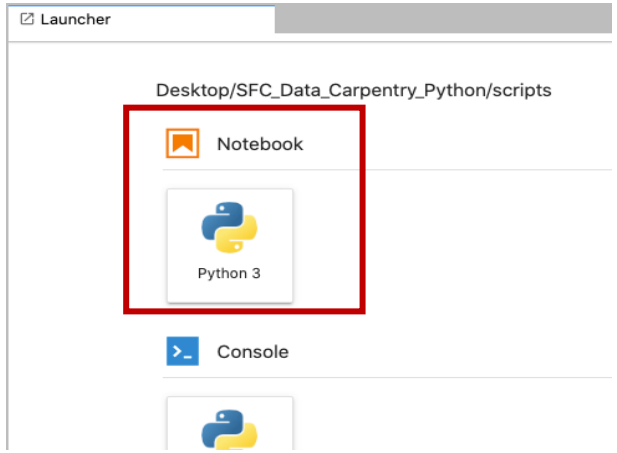
• **To convert this plain text (.txt) file to a Python program (.py)**, select the *Save File As* action from the *File* menu in the **Menu Bar** and give your new text file a name that ends with the *.py* extension (*replace .txt extension*).

- The *.py* extension lets everyone (including the operating system) know that this text file is a *Python* program.

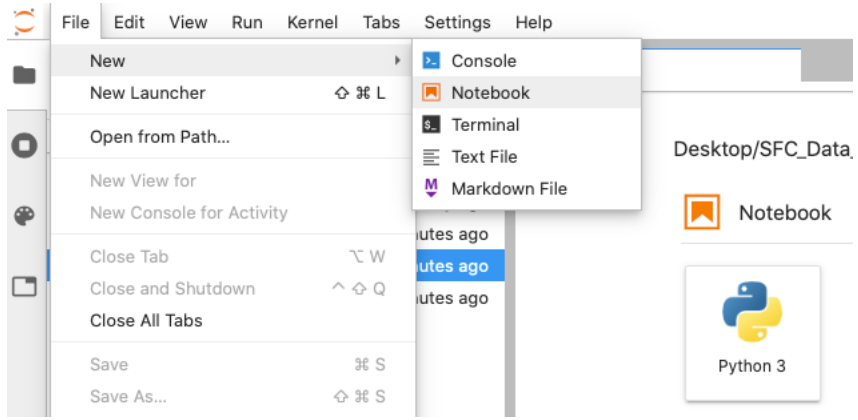
- This is convention, not a requirement.

Episode #1 – Creating a *Jupyter Notebook (file_name.ipynb)* in **JupyterLab**

To open a new *Jupyter Notebook* click the **Python 3** icon under the *Notebook* header in the Launcher tab in the main work area.



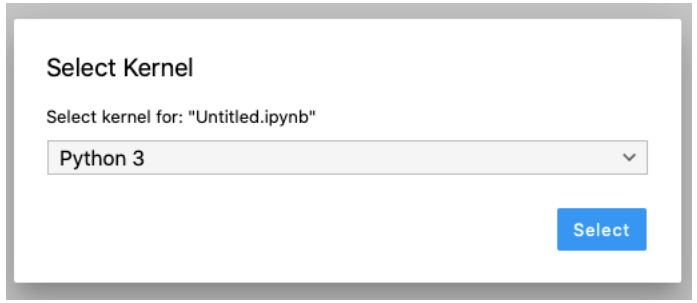
OR: You can also create a new notebook by selecting *New* -> *Notebook* from the *File menu* in the **Menu Bar**.



• Notebook files have the extension *.ipynb* to distinguish them from plain-text Python (*.py*) programs.

• Notebooks can be exported as Python scripts that can be run from the command line.

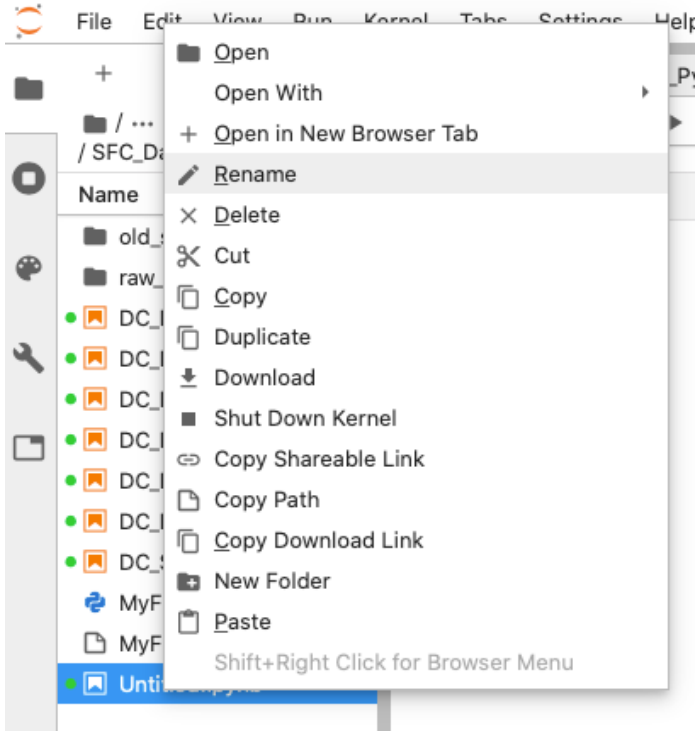
•The notebook file is stored in a JSON file format that allows Jupyter to combine code, text and images.



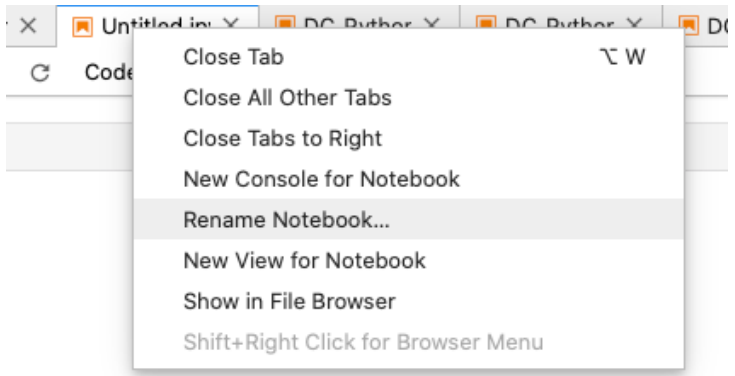
[Read more: https://jupyterlab.readthedocs.io/en/stable/user/notebook.html]

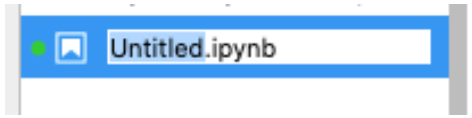
Episode #1 – Creating a *Jupyter Notebook (file_name.ipynb)* in **JupyterLab**

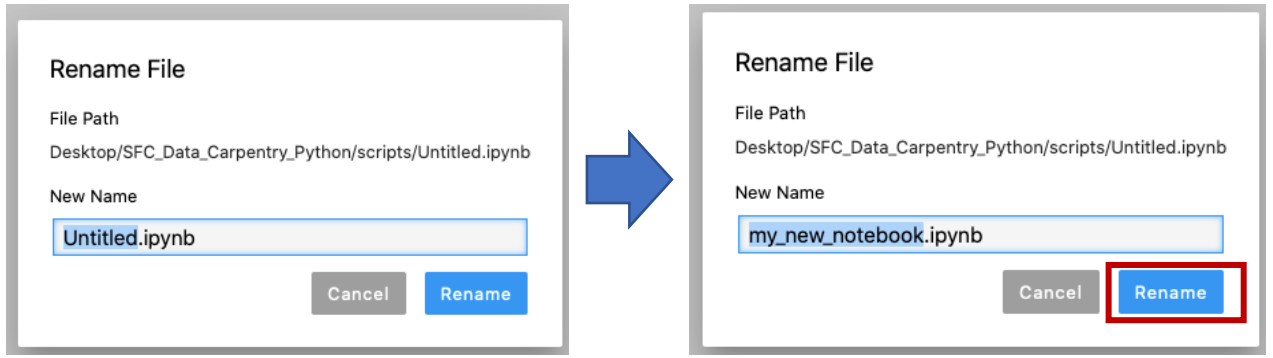
Name your newly created Jupyter Notebook *(.ipynb)* file by right click on the file's name and select *Rename*..



OR







For now, everyone should have a fully functional ***JupyterLab*** running in a web browser and a ***Jupyter Notebook file (.ipynb)*** open and ready-to-go

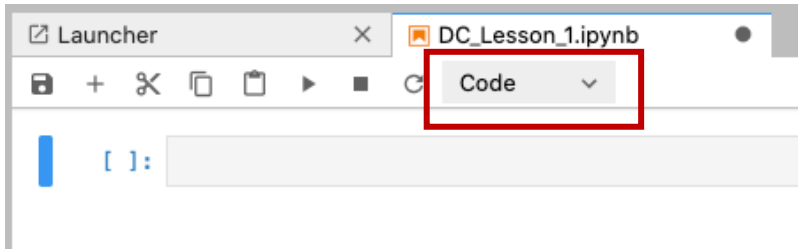
Episode #1 – Editing and running Python scripts in *Jupyter Notebook*

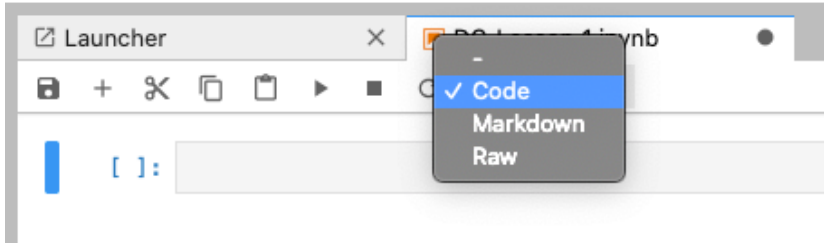
While it's common to write Python scripts using a *Text editor*, we are going to use the Jupyter Notebook for the remainder of this workshop.

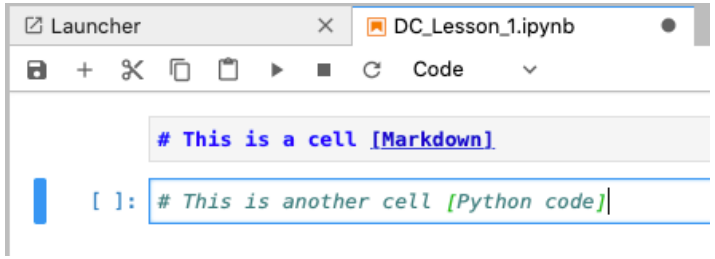
- **Some key advantages**:
    - You can easily type, edit, and copy and paste blocks of code.
    - Tab complete allows you to easily access the names of things you are using and learn more about them.
    - It allows you to annotate your code with links, different sized text, bullets, etc. to make it more accessible to you and your collaborators.
    - It allows you to display figures next to the code that produces them to tell a complete story of the analysis.

Episode #1 – Editing and running Python scripts in *Jupyter Notebook* - ***Cells***

Each notebook contains one or more cells that contain code, text, or images.





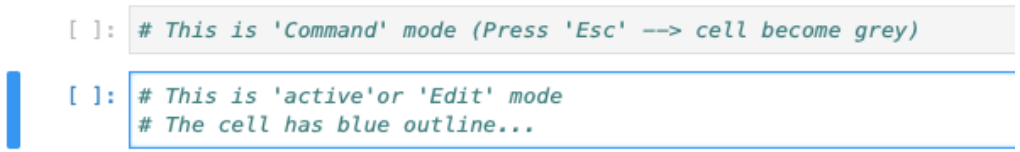


Jupyter mixes code and text in different types of blocks, called cells. We often use the term “code” to mean “the source code of software written in a language such as Python”.

A “code cell” in a Notebook is a cell that contains software; a “text cell” is one that contains ordinary prose/ narrative written for human beings.

Episode #1 – Python scripts in *Jupyter Notebook* – ***Command & Edit***

- If you press `Esc` and `Return` alternately, the outer border of your code cell will change from gray to blue.
- These are the **Command** (gray) and **Edit** (blue) modes of your notebook.
- Command mode allows you to edit notebook-level features, and Edit mode changes the content of cells.
- When in Command mode (esc/gray),
  - The `b` key will make a new cell below the currently selected cell.
  - The `a` key will make one above.
  - The `x` key will delete the current cell.
  - The `z` key will undo your last cell operation (which could be a deletion, creation, etc).
- All actions can be done using the menus, but there are lots of keyboard shortcuts to speed things up.

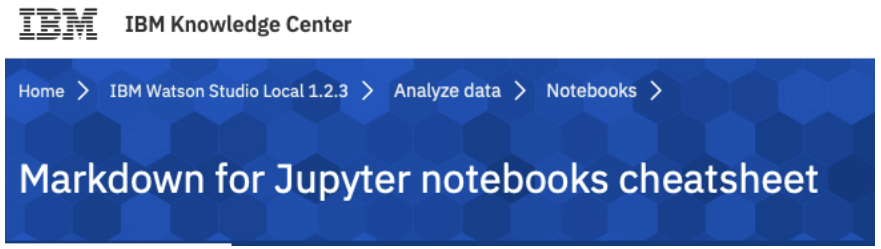


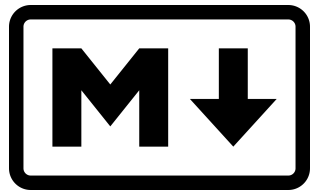
Use the keyboard and mouse to select and edit cells.

- Pressing the `Return` key turns the border blue and engages Edit mode, which allows you to type within the cell.
- Because we want to be able to write many lines of code in a single cell, pressing the `Return` key when in Edit mode (blue) moves the cursor to the next line in the cell just like in a text editor.
- We need some other way to tell the Notebook we want to run what's in the cell.
- Pressing `Shift` + `Return` together will execute the contents of the cell.
- Notice that the `Return` and `Shift` keys on the right of the keyboard are right next to each other.

Episode #1 – *Jupyter Notebook* – ***'Markdown'* for well-formatted documentation**

- Notebooks can also render Markdown.
  - A simple plain-text format for writing lists, links, and other things that might go into a web page.
  - Equivalently, a subset of HTML that looks like what you'd send in an old-fashioned email.
- Turn the current cell into a Markdown cell by entering the Command mode (Esc/gray) and press the M key.
- `In [ ]:` will disappear to show it is no longer a code cell and you will be able to write in Markdown.
- Turn the current cell into a Code cell by entering the Command mode (Esc/gray) and press the y key.





https://www.ibm.com/support/knowledgecenter/SSHGWL_1.2.3/analyze-data/markd-jupyter.html

https://daringfireball.net/projects/markdown/

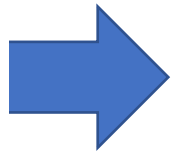
https://medium.com/analytics-vidhya/the-ultimate-markdown-guide-for-jupyter-notebook-d5e5abf728fd

Episode #1 – *Jupyter Notebook* – ***'Markdown'*** **for well-formatted documentation**

[Execute cell: *Shift* + *Enter*]

Marking down text in aJupyter Notebook

```
# Main heading

## Subheading

* List 1 of items with indentation
    * Item 1
    * Item 2

        * Point A
        * Point B

* List 2 with numbered item
    1. Item one
    2. Item two

1. First point

2. Second point

#### Emphasis

Use the following code to emphasize text:

Bold text: __string__ or **string**

Italic text: _string_ or *string*

Italic AND bold (combined): ***string***
```



Marking down text in aJupyter Notebook

Main heading

Subheading

- List 1 of items with indentation
  - Item 1
  - Item 2
    - Point A
    - Point B
- List 2 with numbered item
  1. Item one
  2. Item two

1. First point
2. Second point

Emphasis

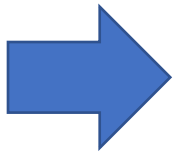
Use the following code to emphasize text:

Bold text: **string** or **string**

Italic text: *string* or *string*

Italic AND bold (combined): ***string***

```
[Create links](http://software-carpentry.org) with `[...](...)`.

Or use [named links][data_carpentry]
```



Create links with `[...](...)`.

Or use [named links][data_carpentry]

[ ]:

Episode #1 – *Jupyter Notebook* – ***'Markdown'* for well-formatted documentation**

Further example: using *arithmetic operations*..

```
[4]: # This is a conventional comment.. Cell is in 'Code' mode

     7 * 3
     2 + 1
```

```
[4]: 3
```

```
## Cell is in 'Markdown' mode

7 * 3

2 + 1
```

Cell is in 'Markdown' mode

7 * 3

2 + 1

```
[ ]:
```

Further example: *Marking down a Math equation*..

```
Standard Markdown (such as we're using for these notes) won't render
equations, but the Notebook will. Create a new Markdown cell and enter
the following:

$\sum_{i=1}^{N} 2^{-i} \approx 1$
```

Standard Markdown (such as we're using for these notes) won't render equations, but the Notebook will. Create a new Markdown cell and enter the following:

$$\sum_{i=1}^{N} 2^{-i} \approx 1$$

The notebook shows the equation as it would be rendered from **LaTeX equation syntax**. The dollar sign, $, is used to tell Markdown that the text in between is a LaTeX equation. If you're not familiar with LaTeX, underscore, _, is used for subscripts and circumflex, ^, is used for superscripts. A pair of curly braces, { and }, is used to group text together so that the statement i=1 becomes the subscript and N becomes the superscript. Similarly, -i is in curly braces to make the whole statement the superscript for 2. \sum and \approx are LaTeX commands for "sum over" and "approximate" symbols.

Episode #1 – **JupyterLab – Finger Exercise 1.2** **(~ 2 minutes)**

Creating Lists in Markdown

Create a nested list in a Markdown cell in a notebook that looks like this:

1. Get funding.
2. Do work.
    - Design experiment.
    - Collect data.
    - Analyze.
3. Write up.
4. Publish.

Episode #1 – *Closing* ***JupyterLab***

From the **Menu Bar** select the "*File"* menu and the choose *"Quit*" at the bottom of the dropdown menu.

You will be prompted to confirm that you wish to shutdown the ***JupyterLab server*** (*don't forget to save your work*!).

Click "*Confirm*" to shutdown the JupyterLab server.

Episode #1 – *Community*

Python's community is vast; diverse & aims to grow; Python is Open.

Great software is supported by great people, and Python is no exception. Our user base is enthusiastic and dedicated to spreading use of the language far and wide. Our community can help support the beginner, the expert, and adds to the ever-increasing open-source knowledgebase.

https://www.python.org/community/

stackoverflow

https://stackoverflow.com

Episode #1 – **Getting involved.. Where to look? (plenty of sources out there!)**

THE CARPENTRIES

https://carpentries.org/community/

software carpentry — Teaching basic lab skills for research computing

ReproducibiliTea

"We are a grassroots journal club initiative that helps young researchers create local Open Science journal clubs at their universities to discuss diverse issues, papers and ideas about improving science, reproducibility and the Open Science movement."

https://reproducibilitea.org

…etc

Episode #1 – **Getting involved.. Where to look? (plenty of sources out there!)**

Explore exciting projects! ☺

https://towardsdatascience.com

Towards Data Science

https://www.kaggle.com/

Publish your Open Source Software projects!

The Journal of Open Source Software

The Journal of Open Source Software is a **developer friendly**, open access journal for research software packages.

Committed to publishing quality research software with zero article processing charges or subscription fees.

https://joss.theoj.org

Episode #1 – **Recommended readings**: a few hand-picked articles

Article Source: **Good enough practices in scientific computing**
Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, et al. (2017) Good enough practices in scientific computing. PLOS Computational Biology 13(6): e1005510.https://doi.org/10.1371/journal.pcbi.1005510

Wilkinson, M., Dumontier, M., Aalbersberg, I. *et al.* **The FAIR Guiding Principles for scientific data management and stewardship**. *Sci Data* 3, 160018 (2016). https://doi.org/10.1038/sdata.2016.18

Article Source: **Best Practices for Scientific Computing**
Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014) **Best Practices for Scientific Computing**. PLOS Biology 12(1): e1001745.https://doi.org/10.1371/journal.pbio.1001745

Wickham H. **Tidy Data**. *Journal of Statistical Software* 59(10); (2014). 10.18637/jss.v059.i10

Any questions for Episode #1 – ‘Running and Quitting’?

Summary >> We’ve talked about..

The Notebook has Command and Edit modes.
Use the keyboard and mouse to select and edit cells.
Python scripts are plain text files.
Use the Jupyter Notebook for editing and running Python.
The Notebook will turn Markdown into pretty-printed documentation.
Markdown does most of what HTML does.

Episode #2 – '**Variables & Assignment**'

https://swcarpentry.github.io/python-novice-gapminder/02-variables/index.html

Learning Objectives

- Assign values (data or piece of information) to a named variable (an object, in general) .
- Correctly trace value changes in programs that use scalar assignment.
- Indexing of a character string – filter individual characters / slicing to get a substring.

Episode #2 – ‘**Variables & Assignment**’ - ***Variable object to store values***

Variables are named objects to store information (data).

• In Python the ‘**=**‘ symbol assigns the value on the right to the name on the left.

• The variable is created and stored at a specific dedicated memory address when a value is assigned to it.

• Here, Python assigns an age (*integer numeric data*) to a variable named **age** and a name in quotes (a *character string*) to a variable named as **first_name**.

[8]:
```
# We can execute multiple assignment operations in a single cell

age = 42

first_name = 'John'
```

Episode #2 – ‘**Variables & Assignment**’ - ***Variable object to store values***

Naming conventions for variables:

- can **only** contain letters, digits, and underscore _ (typically used to separate words in long variable names)
- cannot start with a digit
- are **case sensitive** (age, Age and AGE are three different variables)

• Variable names that start with underscores like **__this_is_another_variable** have a special meaning so we won’t do that until we understand the convention.

Episode #2 – ‘**Variables & Assignment**’ – ***Print out variable (object) content***

• Python has a *built-in function* called **print()** that prints things as text (*character stream*).

• Call the function (i.e., tell Python to run it) by using its name. Functions (or methods) are also named objects and the user can refer to them by their names.

• Provide values to the function (i.e., the things to print) in parentheses.

• To add a string to the printout, wrap the string in single or double quotes.

• The values passed to the function are called **arguments (or parameters).**

• **print()** automatically puts a single space between items to separate them.

• And wraps around to a new line at the end.

```
[13]: # prints only the most recent (last) variable's value

      age

      first_name

[13]: 'John'

[14]: age

[14]: 42

[11]: (age)

[11]: 42

[15]: first_name

[15]: 'John'

[12]: (first_name)

[12]: 'John'

[16]: print(age)

      42

[20]: # Can 'concatenate' the stored values and combine with further user-defined data to display
      # a desired character stream, e.g.:

      print(first_name, 'is', age, 'years old.')

      John is 42 years old.
```

Episode #2 – ‘Variables & Assignment’ – *Print out variable (object) content*

Variables must be created before they are used.

If a variable doesn’t exist yet, or if the name has been mis-spelled, Python reports an **error.**

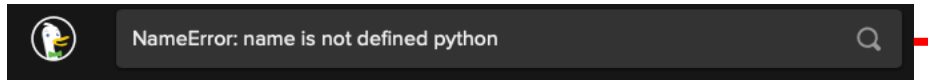
Python

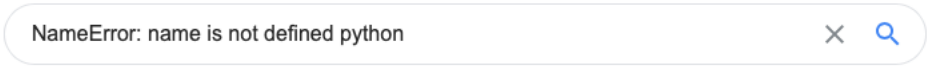
```
print(last_name)
```

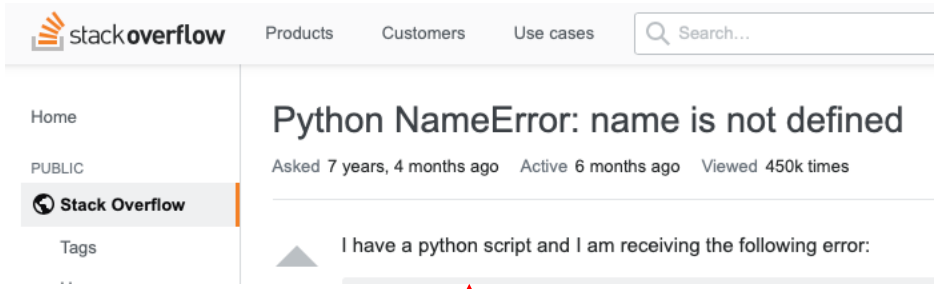
Error

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-c1fbb4e96102> in <module>()
----> 1 print(last_name)

NameError: name 'last_name' is not defined
```

Need help?







That is very likely a solution is already out there.. ☺

Episode #2 – 'Variables & Assignment' – Print out variable (object) content

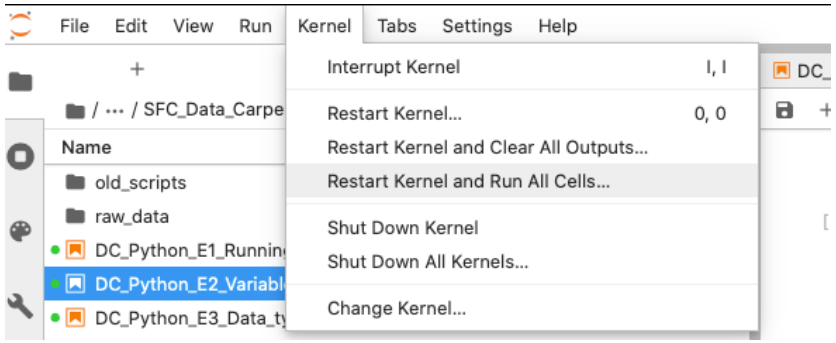
Variables Persist Between Cells

Be aware that it is the *order* of execution of cells that is important in a Jupyter Notebook, not the order in which they appear.

Python will remember *all* the code that was run previously, including any variables you have defined, irrespective of the order in the notebook. Therefore if you define variables lower down the notebook and then (re)run cells further up, those defined further down will still be present.

To prevent confusion, it can be helpful to use the *Kernel -> Restart & Run All* option which clears the interpreter and runs everything from a clean slate going top to bottom.

```
: print(my_variable)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-22-f999b8cc1cdb> in <module>
----> 1 print(my_variable)

NameError: name 'my_variable' is not defined

: my_variable = 1
```



Episode #2 – ‘Variables & Assignment’ – Pre-defined variables can be utilised..

Variables can be used in calculations

We can use variables in calculations (e.g. arithmetic operations) just as if they were values, but now the stored values (or data) can be addressed or referred by its given name.

```
# an object named 'age' is already in the memory, storing a specific value
# We can overwrite or update its value when executing a new assignment

age = age + 3
```

```
print(first_name,"'s age in three years:", age)
```

```
John 's age in three years: 45
```

Episode #2 – ‘Variables & Assignment’ – Strings, indexes, slicing ..

• The (*alphanumeric*) characters in a string are ordered. Think of a stream of concatenated characters.

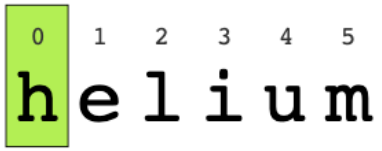
• We can treat a string as a list of characters

• Each position in the string (first character, second character, etc.) is given a number. This number is called an **index** or sometimes a subscript.

• Indices are numbered from 0 (when indexing from left to right). Otherwise, indexing starts with ‘-1’.

• Use the position’s index in square brackets to get the character at that pos

```
print(atom_name[0])
```




```
[31]: print(atom_name[6]) # indexing starts from zero

      # If we would retreive/ print the n-th character in a stri
      # we would need to address the character on the (n-1)-th i
      # print(my_character_string_variable[n-1])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-31-21c13545827c> in <module>
----> 1 print(atom_name[6]) # indexing starts from zero
      2
      3 # If we would retreive/ print the n-th character in a string
      4 # we would need to address the character on the (n-1)-th index
      5 # print(my_character_string_variable[n-1])

IndexError: string index out of range
```

Episode #2 – 'Variables & Assignment' – Strings, indexes, slicing ..

If we are going to assign a numeric string (e.g. an integer type value) to our variable / object and would use indexing..
Let's see what happens!

[32]:
```
numeric_string = 123
```

[33]:
```
# Now, say, aiming for printing out the second digit (or quasi-character):
print(numeric_string[1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-07610b28eb23> in <module>
      1 # Now, say, aiming for printing out the second digit (or quasi-character):
----> 2 print(numeric_string[1])

TypeError: 'int' object is not subscriptable
```

[ ]:
```
# We need to make a variable (or data) type conversion and transform our numeric stream to actual character
# string by using Python's built-in str() function. Otherwise, we may wrap quotation marks around
# our number (digit) stream that would let Python evaluate the object as a character string.
```

[34]:
```
numeric_string = str(123)
```

[35]:
```
numeric_string
```

[35]:
```
'123'
```

[36]:
```
numeric_string_2 = '123'
```

[37]:
```
numeric_string_2
```

[37]:
```
'123'
```

[38]:
```
print(numeric_string[1])
```

```
2
```

[39]:
```
print(numeric_string_2[1])
```

```
2
```

Episode #2 – ‘**Variables & Assignment’ – Strings, indexes, slicing ..**

• A subset of a string is called a **substring**. A substring can be as short as a single character.

• An item in a list is called an element. Whenever we treat a string as if it were a list, the string’s elements are its individual characters.

• A slice is a part of a string (or, more generally, any list-like thing).

• We take a slice by using **[start : stop]**, where *start* is replaced with the index of the first element we want and *stop is replaced with the index of the element just after the last element we want*. (character under ‘stop’ index won’t be included in your substring)

• Mathematically (*closed-open interval symbols*), you might say that a slice selects **[start : stop)**.

•The difference between stop and start is the slice’s length.

• Taking a slice does not change the contents of the original string.

Episode #2 – **JupyterLab – Finger Exercise 2.1** **(~ 1 minute)**

Slicing practice

What does the following program print?

Python

```
atom_name = 'carbon'
print('atom_name[1:3] is:', atom_name[1:3])
```

Episode #2 – **JupyterLab – Finger Exercise 2.2** **(~ 3 minutes)**

Slicing concepts

1. What does `thing[low:high]` do?
2. What does `thing[low:]` (without a value after the colon) do?
3. What does `thing[:high]` (without a value before the colon) do?
4. What does `thing[:]` (just a colon) do?
5. What does `thing[number:some-negative-number]` do?
6. What happens when you choose a `high` value which is out of range? (i.e., try `atom_name[0:15]`)

Episode #2 – **JupyterLab – Finger Exercise 2.2** **(~ 3 minutes)**

What does *my_string [low : high]* do?

Let's use our *atom_name* example.

```
atom_name[1:3]
```

```
'el'
```

What does *my_string [low :]* do?

```
atom_name[1:]
```

```
'elium'
```

What does *my_string [: high]* do?

```
atom_name[:3]
```

```
'hel'
```

What does *my_string [:]* do?

```
atom_name[:]
```

```
'helium'
```

What does *my_string [1 : -2]* do?

```
atom_name[1:-2]
```

```
'eli'
```

Episode #2 – 'Variables & Assignment' – Strings, indexes, slicing ..

Slicing example:

```
print(atom_name[0:3]) # Will prin out the first 3 characters but not the 4th
```

```
hel
```

Use the built-in function **len()** to find the length of a string

Use the built-in function len to find the length of a string

```
[ ]: print(len('helium'))
```

```
[ ]: print(len('helium'),'=', len(atom_name))
```

NB: The order of command execution is linear/ sequential (cell-by-cell)!

Nested function and order of execution

Below is a stylised illustration of a *nested function*:

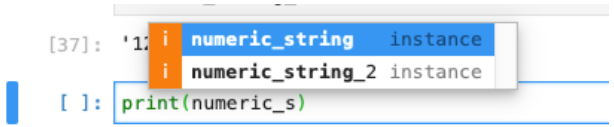
function_name_2 (function_name_1 (argument_or_parameter))

The order of execution:

1. top-down
2. right-to-left
3. inside-out

In thie above pseudo-code example, function_name_1 will execute its pre-defined operation first on 'argument' (this function should have an output or a *return* value), then function_name_2 will be executed taking the return value of function_name_1 as its argument (or parameter).

Hit *Tab* to autocomple pre-defined objects' names (e.g. variables, functions or methods)



Episode #2 – ‘Variables & Assignment’ – Assignment is sequential/ linear…

Variables only change value when something is assigned to them

- If we make one cell in a spreadsheet depend on another, and update the latter, the former updates automatically.
- This does not happen in programming languages.

Let's consider the following example:

first = 1

*second = 5 * first*

first = 2

print('first is', first, 'and second is', second)

[2]:
```
first = 1
second = 5 * first
```

[3]:
```
first = 2
print('first is', first, 'and second is', second)
```

```
first is 2 and second is 5
```

- The computer reads the value of *first* when doing the multiplication, creates a new value, and assigns it to *second*.
- After that, *second* does not remember where it came from.

Again, execution of operations are linear-sequential as we saw in earlier examples, as well.

Episode #2 – **JupyterLab – Finger Exercise 2.3** **(~ 3 minutes)**

Swapping Values

Fill the table showing the values of the variables in this program *after* each statement is executed.

Python

```
# Command  # Value of x   # Value of y   # Value of swap #
x = 1.0    #              #              #               #
y = 3.0    #              #              #               #
swap = x   #              #              #               #
x = y      #              #              #               #
y = swap   #              #              #               #
```

Hint: the order of command execution is linear/ sequential (cell-by-cell)!

```
# Step 1 - Result after executing the first line:

# Command  # Value of x   # Value of y   # Value of swap #
x = 1.0    # 1.0          # not defined  # not defined    #
```

Episode #2 – **JupyterLab – Finger Exercise 2.4** **(~ 1 minutes)**

Predicting Values

What is the final value of `position` in the program below? (Try to predict the value without running the program, then check your prediction.)

Python

```
initial = 'left'
position = initial
initial = 'right'
```

Hint: the order of command execution is linear/ sequential (cell-by-cell)!

Episode #2 – ‘Variables & Assignment’ – Strings, indexes, slicing ..

Further good-to-know practices:

1. **Unsure about what arguments the given function need to have?**

E.g. let's consider a buil-in function, **abs()** as an example - this function does as its name indicates: Return the absolute value of a number. The argument may be an integer or a floating point number.

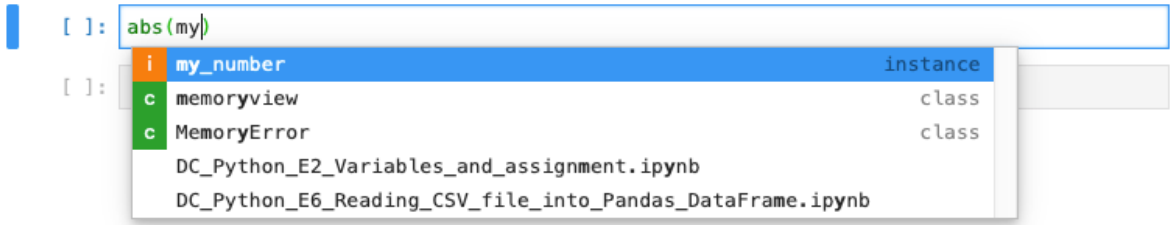
move the cursor (click between) the parenthesis, then hit *tab*. This will return with a drop-down list of possible pre-defined objects as arguments (on the top ranked the most important ones).

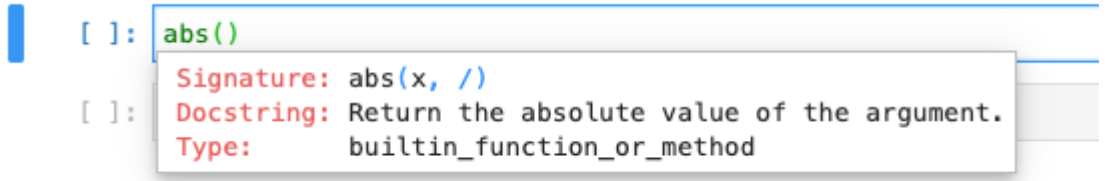
2. **Unsure about what the function is doing?**

By using the same example, move the cursor right before the first parenthesis, then hit **Shift** + **tab** combination. A tiny window will pop-up that describes the function's brief (e.g. brief documentation and its operational mechanism 'under the hood').

```
# Let's try the above in this cell:
# First, define a new object (a variable) that stores a negative integer number:
my_number = -13
```

```
[12]: abs(my_number)
[12]: 13
```





Episode #2 – ‘Variables & Assignment’ – Strings, indexes, slicing ..

Python is case-sensitive

```
[13]: Name = 'Alice'
      name = 'Bob'

[14]: print(Name, 'and', name)

      Alice and Bob
```

• Python thinks that *upper-* and *lower-case* letters are different, so *Name* and *name* are different variables.

• There are conventions for using upper-case letters at the start of variable names so we will use lower-case letters for now.

Use meaningful (intuitive/ self-explaining) variable names

• Python doesn’t care what you call variables as long as they obey the rules (alphanumeric characters and the underscore).

• Use meaningful variable names to help other people understand what the program does.

• The most important “other person” is your future self. (*E.g. think about scenarios when you are tired*..)

```python
flabadab = 42
ewr_422_yY = 'Ahmed'
print(ewr_422_yY, 'is', flabadab, 'years old')
```

Any questions for Episode #2 – ‘Variables and Assignment’?

Summary >> We’ve talked about..

Use variables to store values.
Use print to display values.
Variables persist between cells.
Variables must be created before they are used.
Variables can be used in calculations.
Use an index to get a single character from a string.
Use a slice to get a substring.
Use the built-in function len() to find the length of a string.
Python is case-sensitive.
Use meaningful variable names.

Any questions or comments?

Stretch your legs & enjoy your..

/

robert.nagy@ed.ac.uk

Episode #3 – ‘Data Types and Type Conversion’

https://swcarpentry.github.io/python-novice-gapminder/03-types-conversion/index.html

Learning Objectives

- Explain key differences between integers and floating point numbers.
- Explain key differences between numbers and character strings.
- Use built-in functions to convert between integers, floating point numbers, and strings.

Episode #3 – ‘Data Types and Type Conversion’

For every variable object: has a specific type of data value (or information) stored in it.

- Every data value stored in a program memory has a specific type.
- **Integer** (*int*): represents positive or negative whole numbers like 3 or -512.
- **Floating point number** (*float*): represents real numbers like 3.14159 or -2.5.
- **Character string** (usually called “*string*”, *str*): text.
    - Written in either single quotes or double quotes (as long as they match).
    - The quote marks aren’t printed when the string is displayed.

Episode #3 – ‘Data Types and Type Conversion’ – Use the *type()* function …

Use the built-in function **type()** to find the type of a variable or object (what type if information has been stored in it)

```
# For example:

print(type(42))
```

```
<class 'int'>
```

```
type(42)
```

```
int
```

```
print(type('character'))
```

```
<class 'str'>
```

```
type('character')
```

```
str
```

```
a = 10
b = 'myString'
```

```
type(a)
```

```
int
```

```
print(type(a))
```

```
<class 'int'>
```

```
type(b)
```

```
str
```

```
print(type(b))
```

```
<class 'str'>
```

Episode #3 – ‘Data Types and Type Conversion’ – Use the *type()* function …

Types control what **operations** (or functions / methods) can be performed (*executed*) on a given value / object, value / object pair.

```
5 - 3
```

```
2
```

```
print(5 - 3)
```

```
2
```

```
(5 - 3)
```

```
2
```

```
'hello' - 'h'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-51fc02de0216> in <module>
----> 1 'hello' - 'h'

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

- *Addition* (‘**+**’) and *multiplication* (‘*****’), on the other hand, are valid operations on both **numeric** and **str** values or objects.
  - “Adding” character strings concatenates them.
  - Multiplying a character string by an integer N creates a new string that consists of that character string repeated N times. (*NB: Multiplication is a repeated addition*)

```
[26]: first_name = 'John'
      last_name = ' Doe' # NB: there is a white space at the beginning
```

```
[27]: full_name = first_name + last_name
```

```
[28]: full_name
```

```
[28]: 'John Doe'
```

```
[31]: full_name_2 = 'Alice' + ' ' + 'Smith'
```

```
[32]: full_name_2
```

```
[32]: 'Alice Smith'
```

```
[57]: 'first name' + ' last name'
```

```
[57]: 'first name last name'
```

```
[59]: print('First name' + ' and' + ' last name')
```

```
First name and last name
```

Episode #3 – ‘Data Types and Type Conversion’ – Use the *type()* function …

Types control what **operations** (or functions / methods) can be performed (*executed*) on a given value / object, value / object pair.

```
[52]: # Apply multiplication ('*') operation on character string ('str') values
      # or variable objects:

      string_object_1 = 'Apple '
      string_object_2 = "Banana "

[43]: full_name = full_name + ' ' # Add white space to the end

[44]: 'Pear ' * 10

[44]: 'Pear Pear Pear Pear Pear Pear Pear Pear Pear Pear '

[47]: c = "pear "

[48]: 10 * c

[48]: 'pear pear pear pear pear pear pear pear pear pear '

[49]: c * 10

[49]: 'pear pear pear pear pear pear pear pear pear pear '

[53]: print(5 * string_object_1, 5 * string_object_2)

      Apple Apple Apple Apple Apple  Banana Banana Banana Banana Banana

[55]: print(5 * string_object_1 + 5 * string_object_2) # in case of strings, comma ','
      # can be replaced by '+' to concatenate two streams of strings

      Apple Apple Apple Apple Apple Banana Banana Banana Banana Banana
```

Episode #3 – ‘Data Types and Type Conversion’ – Use the *length()* function …

Length can be checked in case of string (character) values but not in case of numeric objects

- The built-in function ***len()*** counts the number of characters in a string.

```
[83]: # full_name = full_name.rstrip(' ') # rstrip removes e.g. white spaces from the

[85]: full_name

[85]: 'John Doe'

[87]: len(full_name) # 8-character-long; NB: white spaces count as well!

[87]: 8

[88]: len('abc')

[88]: 3

[89]: len(a) # remember, earlier we've assigned numeric 10 to object named as 'a'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-89-1a2e6ec5f1e3> in <module>
----> 1 len(a)

TypeError: object of type 'int' has no len()

[90]: len(42)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-90-de5840b53763> in <module>
----> 1 len(42)

TypeError: object of type 'int' has no len()
```

Episode #3 – 'Data Types and Type Conversion' – Converting data types..

Data type conversions

Data objects/ values should be converted in order to let Python implement 'arithmetic operations' on them. For example:

- Cannot add numbers and strings - *Mind about data type homogeneity within scope of the operation*!

```
[91]: print(1 + '2') # numeric values can be represented as both character strings as well as numeric objects

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-91-013270d67d3d> in <module>
----> 1 print(1 + '2')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
[95]: print(a, 2)

10 2
```

```
[92]: print (a + 2)

12
```

```
[96]: print (a + '2')

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-66e9743c2381> in <module>
----> 1 print (a + '2')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
[97]: print("The participant's age is: " + '42')

The participant's age is: 42
```

```
[98]: print("The participant's age is: " + 42)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-98-4a9f2dab4121> in <module>
----> 1 print("The participant's age is: " + 42)

TypeError: can only concatenate str (not "int") to str
```

Episode #3 – ‘Data Types and Type Conversion’ – Converting data types..

Let's convert data types to preserve homogeneity

Here, we are going to implement **explicit data type conversion**.

[99]:
```
print(1 + int('2'))
```

```
3
```

[100]:
```
print(5 + a)
```

```
15
```

[107]:
```
print(str(1) + '2') # str + str --> This IS concatenation
# print(str(1) + ' ' + '2')
```

```
12
```

[108]:
```
print('2' + str(a))
```

```
210
```

Episode #3 – ‘Data Types and Type Conversion’ – *Mixing numeric subtypes*..

Can mix integers and floats as both are numeric subtypes and can be easily transformed to one another

- *Python 3* automatically converts integers to floats as needed. (Integer division in Python 2 will return an integer, the floor of the division).

[109]:
```
print('half is ', 1/2.0)
```
```
half is  0.5
```

[110]:
```
print('three sqared is ', 3.0 ** 2)
```
```
three sqared is  9.0
```

[116]:
```
print('a third is ', 1/3)
```
```
a third is  0.3333333333333333
```

[117]:
```
print('a third is ', 1.0/3)
```
```
a third is  0.3333333333333333
```

[118]:
```
print('a third is ', 1/3.0)
```
```
a third is  0.3333333333333333
```

[119]:
```
print('a third is ', 1.0/3.0)
```
```
a third is  0.3333333333333333
```

Episode #3 – ‘Data Types and Type Conversion’ – *Mixing numeric subtypes*..

Can use floor division - implements arithmetic division and rounds to the closest integer. The returning result's *data type* or *format* is dependent on the operands' data types (e.g. if either is float then the result will be printed out in a float format).

[120]:
```
print('a third is ', 1//3.0)
```
```
a third is  0.0
```

[121]:
```
print('a third is ', 1//3)
```
```
a third is  0
```

[ ]:
```

```

Episode #3 – ‘Data Types and Type Conversion’ – *Further considerations*..

1. **Fractions** AKA *floating point number* representation (*data type:* ***float***)

- E.g. check data type of the value 3.14

```
[122]: type(3.14)
```

```
[122]: float
```

2. **Automatic (AKA Implicit) type conversion**: *float* is 'higher order' than *int* (by default, Python want to preserve the granularity opr resolution..)

- E.g. check data type of returning value of the following operation: 3.14 + 3

Read more

```
[123]: type(3.14 + 3)
```

```
[123]: float
```

```
[142]: d = 5.3 + 1
```

```
[143]: d
```

```
[143]: 6.3
```

```
[147]: type(d)
```

```
[147]: float
```

Episode #3 – ‘Data Types and Type Conversion’ – *Further considerations*..

3. **Division types**

- In Python 3, the '**//**' operator performs **integer (*whole-number*) floor division**, the '**/**' operator performs **floating-point division**, and the '**%**' (*or modulo*) operator calculates and **returns with the remainder from integer division**.
- For illustration, consider the following examples:
  - *print('5 // 3:', 5//3)*
  - *print('5 / 3:', 5/3)*
  - *print('5 % 3:', 5%3)*

```
[151]: print('5 // 3:', 5//3)
```

```
5 // 3: 1
```

```
[152]: print('5 / 3:', 5/3)
```

```
5 / 3: 1.6666666666666667
```

```
[153]: print('5 % 3:', 5%3)
```

```
5 % 3: 2
```

Episode #3 – ‘Data Types and Type Conversion’ – *Further considerations*..

4. **Type conversion: trings to numbers**

- Where reasonable, **float()** will convert a *string* to a *floating point number*, and **int()** will convert a *floating point number* to an *integer*.
- Look at the following examples:
  - *print("string to float: ", float("3.14"))*
  - *print("float to int: ", int(3.14))*
  - *print("string to int: ", int("3.14"))*

NB: the last one will throw an error - try to comprehend, why..

[155]:
```
print("string to float: ", float("3.14"))
```
```
string to float:  3.14
```

[156]:
```
print("float to int: ", int(3.14))
```
```
float to int:  3
```

[157]:
```
print("string to int: ", int("3.14"))
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-157-96f3c399861f> in <module>
----> 1 print("string to int: ", int("3.14"))

ValueError: invalid literal for int() with base 10: '3.14'
```

- If the type conversion does not make sense...
  - *print("string to float:", float("Hello world!"))*

[158]:
```
print("string to float:", float("Hello world!"))
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-158-8b7db3f305e2> in <module>
----> 1 print("string to float:", float("Hello world!"))

ValueError: could not convert string to float: 'Hello world!'
```

Episode #3 – **JupyterLab – Finger Exercise 3.1** **(~ 3 minutes)**

Choose a Type

What type of value (integer, floating point number, or character string) would you use to represent each of the following? Try to come up with more than one good answer for each problem. For example, in # 1, when would counting days with a floating point variable make more sense than using an integer?

1. Number of days since the start of the year.
2. Time elapsed from the start of the year until now in days.
3. Serial number of a piece of lab equipment.
4. A lab specimen's age
5. Current population of a city.
6. Average population of a city over time.

Episode #3 – **JupyterLab – Finger Exercise 3.2** **(~ 3 minutes)**

Arithmetic with Different Types

Which of the following will return the floating point number `2.0`? Note: there may be more than one right answer.

Python

```
first = 1.0
second = "1"
third = "1.1"
```

1. `first + float(second)`
2. `float(second) + float(third)`
3. `first + int(third)`
4. `first + int(float(third))`
5. `int(first) + int(float(third))`
6. `2.0 * second`

Any Qs for Episode #3 – ‘Data types and type conversion’?

Summary >> We’ve talked about..

Every value has a type.
Used the built-in function type() to find the type of a value.
Types control what operations can be done on values.
Strings can be added and multiplied.
Strings have a length (but numbers don’t).
Must convert numbers to strings or vice versa when operating on them.
Can mix integers and floats freely in operations.

Episode #4 – ‘Built-in Functions & Help’

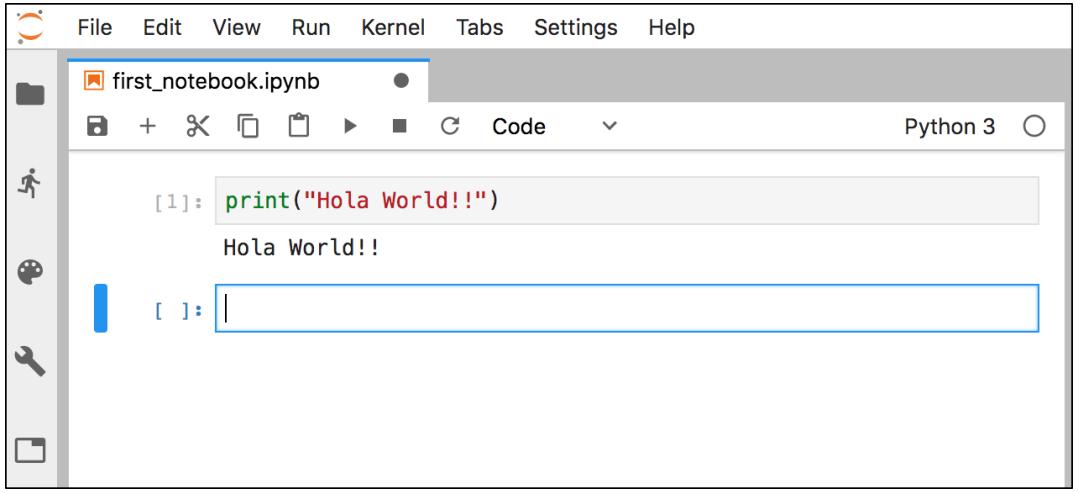
https://swcarpentry.github.io/python-novice-gapminder/04-built-in/index.html

Learning Objectives

- Explain the purpose of functions.
- Correctly call built-in *Python* functions.
- Correctly nest calls to built-in functions.
- Use *help()* to display documentation for built-in functions.
- Correctly describe situations in which *SyntaxError* and *NameError* occur.

Episode #4 – ‘Built-in Functions & Help’

Python script is colour coded. This is to help the user to reduce typing mistakes and inform about what object s/he is looking at. When the user calling a built-in Python function the name of the function will turn green. Consider the *print() function* as an example.

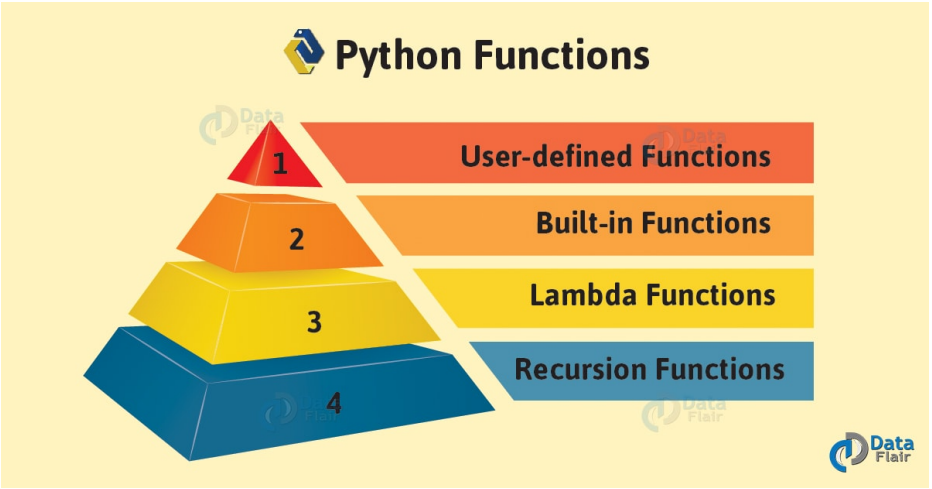


Episode #4 – '[Built-in Functions & Help]' – *List of Python 3.x built-in functions*

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	



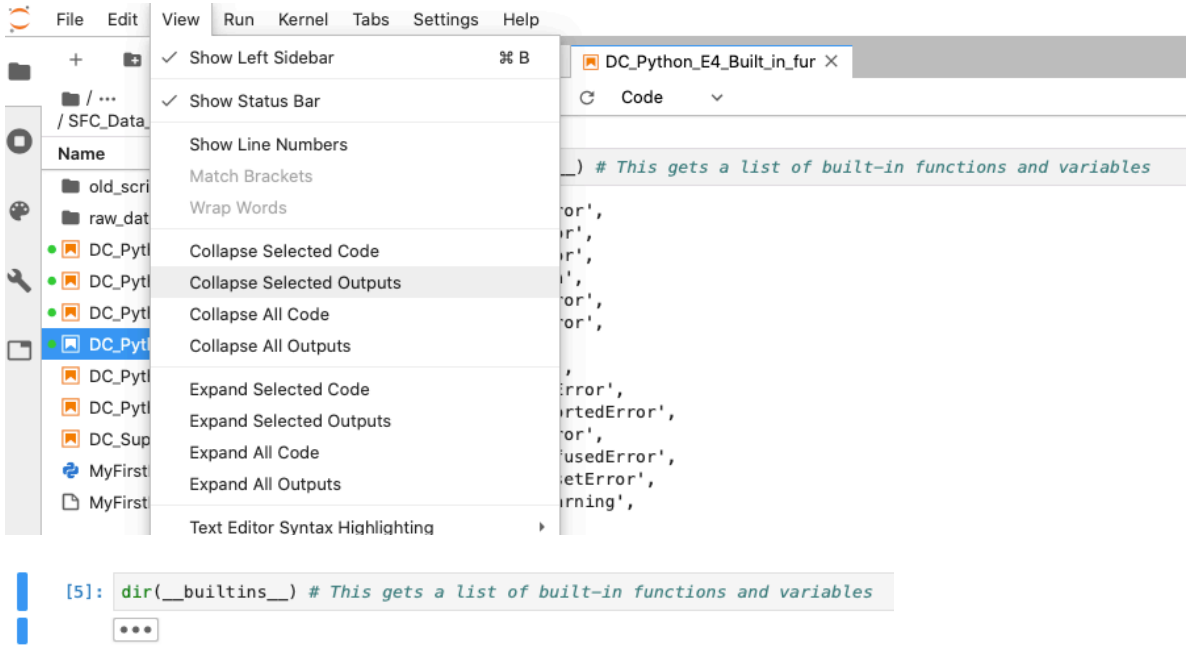
https://docs.python.org/3/library/functions.html

Episode #4 – '[Built-in Functions & Help]'

```
[5]: dir(__builtins__) # This gets a list of built-in functions and variables
```

```
[5]: ['ArithmeticError',
      'AssertionError',
      'AttributeError',
      'BaseException',
      'BlockingIOError',
      'BrokenPipeError',
      'BufferError',
      'BytesWarning',
      'ChildProcessError',
      'ConnectionAbortedError',
      'ConnectionError',
      'ConnectionRefusedError',
      'ConnectionResetError',
      'DeprecationWarning',
      'EOFError',
      'Ellipsis',
      'EnvironmentError',
      'Exception',
      'False',
      'FileExistsError',
      'FileNotFoundError',
      'FloatingPointError',
      'FutureWarning',
      'GeneratorExit',
      'IOError',
      'ImportError',
      'ImportWarning',
      'IndentationError',
      'IndexError',
      'InterruptedError',
      'IsADirectoryError',
      'KeyError',
      'KeyboardInterrupt',
      'LookupError',
      'MemoryError',
      'ModuleNotFoundError',
      'NameError',
      'None',
      'NotADirectoryError',
      'NotImplemented',
      'NotImplementedError',
      'OSError',
      'OverflowError'
```

We can **collapse / expand** the *selected cell* regardless if it is a code/ markdown cell or Python code output. To **collapse** either clicking on the **blue vertical bar** on the left or navigate to *Menu Bar > View > [the user can chose as s/he will proceed..].* To **expand,** either click on the 3 horizontal dots (. . .) / **blue vertical bar** or navigate to *Main Menu bar > View > [select the appropriate 'Expand…' option]*



Episode #4 – '[Built-in Functions & Help]'

Commonly-used built-in functions: **min()**, **max()** and **round()**

- Use **min()** to find the smallest *atomic element*.
- Use **max()** to find the largest value (*or atomic element*) of one or more values.
- Both work on character strings as well as numbers.
  - "Larger" and "smaller" use (0-9, A-Z, a-z) to compare letters.
  - Use the following illustrative examples:
    - *print(max(1, 2, 3))*
    - *print(min('a', 'A', '0'))*

[5]:
```
dir(__builtins__) # This gets a list of built-in functions and variables
```
```
...
```

[6]:
```
print(max(1, 2, 3))
```
```
3
```

[7]:
```
print(min('a', 'A', '0'))
```
```
0
```

Episode #4 – '**Built-in Functions & Help**' – *Use comments to document code..*

Use *Comments* to add documentation to Python code

- In Python, anything written after a *hash* ('#') will be interpreted as a comment and won't be executed
  - E.g. *# This is a comment.*
- Brief description e.g. in a function's body ''' This is a docstring: e.g. a description what this function is doing (...) '''
  - E.g. *''' Write an arbitrary few-line alphanumeric textual stream (...)'''*

[1]:
```
# This line won't be executed by Python (...)
# In this way can add brief comments/ explanations what a specific bit of code is doing
# Or what sort of information have been stored in specific variable or data objects
```

[10]:
```
# Example - only for illustration

def my_fun(a):

    '''This is a so-called 'docstring' that can be used for providing a brief
    description of what
    this function is actually doing.
    Please red more [arbitrary web address]: https://www.python.org '''

    print('The input parameter was ','"', a,'"')
```

[3]:
```
help(my_fun) # Will print the user-defined '''docstring''' too
```

```
Help on function my_fun in module __main__:

my_fun(a)
    This is a so-called 'docstring' that can be used for providing a brief
    description of what
    this function is actually doing.
    Please red more [arbitrary web address]: https://www.python.org
```

[11]:
```
my_fun('This')
```

```
The input parameter was  " This "
```

Episode #4 – '**Built-in Functions & Help**' – *Function arguments (or parameters)*

A function may take zero or more arguments (*or parameters*)

We have seen some functions (*or methods*) already — now let's take a closer look.

An *argument (or parameter)* is a value passed into a function.

- **len()** takes exactly one argument.
- **int()**, **str()**, and **float()** create a *new value* from an existing one.
- **print()** takes zero or more.
  - **print()** with no arguments prints a blank line.

NB: Must always use parentheses, even if they're empty, so that Python knows a function is being called.

```
]: print('before')
   print()
   print('after')
```

```
before

after
```

Functions may only work for certain (*combinations of*) arguments

- **max()** and **min()** must be given *at least one argument*.
  - *"Largest of the empty set"* is a 'meaningless' query. (Or rather it's more philosophical)
  - And they must be given argument(s) for which comparison can be deemed *valid*.

```
[12]: # For example:

      min(1, 'b')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-d0b9e34f3b2c> in <module>
      1 # For example:
      2
----> 3 min(1, 'b')

TypeError: '<' not supported between instances of 'str' and 'int'
```

```
[13]: max('a', 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-21e9ff9e60d2> in <module>
----> 1 max('a', 2)

TypeError: '>' not supported between instances of 'int' and 'str'
```

```
[ ]:
```

Episode #4 – '**Built-in Functions & Help**' - *Optional vs mandatory arguments..*

Functions (*or methods*) may have *optional arguments* which often have pre-defined default values

- For instance, consider the **round()** function will round off a *floating-point number*.
- By default, this function rounds its operand to zero decimal places.

```
[15]: round(3.14159)
```

```
[15]: 3
```

- Although, the user can specify the values of optional arguments to return with an output in a desired format.
- Remain with the above example, let's specify the number of decimal places we would like to see on the outout:
  - Syntax (pseudo-code): *round (input_floating_point_number, number_of_deimal_places)*

```
[16]: round(3.14159, 2)
```

```
[16]: 3.14
```

```
[20]: my_float_number = 3.14159
      round(my_float_number, 4)
```

```
[20]: 3.1416
```

Episode #4 – '**Built-in Functions & Help**' – *Every function returns with 'sg' ..*

That is fair to claim, in Python every function (*or method*) returns with something (*despite there is nothing printed to the output*)

- Every function call produces some result.
- E.g. a valid *empty or latent* return value in Python is **None**.

[28]:
```
result = print('example') # printing out to the screen [the function call/ execution itself], per se, has no specific return value
print('result of print is', result)
```

```
example
result of print is None
```

Episode #4 – '**Built-in Functions & Help**' - *Optional vs mandatory arguments..*

A function that requires at least one argument (*or parameter*) would be deemed that argument as mandatory

And would throw an error if the 'compulsory' parameter remain unspecified. Remaining with the above example:

- Executing *round()* with no input parameter would return with a *TypeError ('missing argument')*

```
[21]: round()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-776eeab594c2> in <module>
----> 1 round()

TypeError: round() missing required argument 'number' (pos 1)
```

The error message is quite explicit, speaks for itself..

Episode #4 – '**Built-in Functions & Help**' – *Interpret error messages..*

Python reports a **syntax error ('SyntaxError')** when it can't understand the *program code (or source)*

It won't even try to run the program if it can't be parsed.

[22]:
```
# Forgot to close the quote marks around the string.

name = 'Feng
```

```
  File "<ipython-input-22-8994afc7b074>", line 3
    name = 'Feng
                ^
SyntaxError: EOL while scanning string literal
```

[24]:
```
# An extra '=' in the assignment.
age = = 52
```

```
  File "<ipython-input-24-ccc3df3cf902>", line 2
    age = = 52
          ^
SyntaxError: invalid syntax
```

[25]:
```
print("hello world"
```

```
  File "<ipython-input-25-fe69f65f3ba9>", line 1
    print("hello world"
                       ^
SyntaxError: unexpected EOF while parsing
```

Regarding the above error message, let's dissect and try to comprehend..

- The message indicates a problem on first line of the input cell ("line 1").
- In this case the *"ipython-input"* section of the file name tells us that we are working with input into *IPython*, the *Python interpreter used by the Jupyter Notebook*.
- The *-25-* part of the filename indicates that the error occurred in *cell number 25* (cell numbers indicated in front of every cell on the left '[25]:') of our Notebook.
- Next is the problematic line of code, indicating the problem's location with a ^ pointer.

Episode #4 – '**Built-in Functions & Help**' – *Interpreting error messages..*

Python reports a runtime error (multiple types) when something goes wrong while a program is being executed

Example below:

[26]:
```
age = 42
remaining = 100 - aege # mis-spelled 'age'
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-0803865f33b2> in <module>
      1 age = 42
----> 2 remaining = 100 - aege # mis-spelled 'age'

NameError: name 'aege' is not defined
```

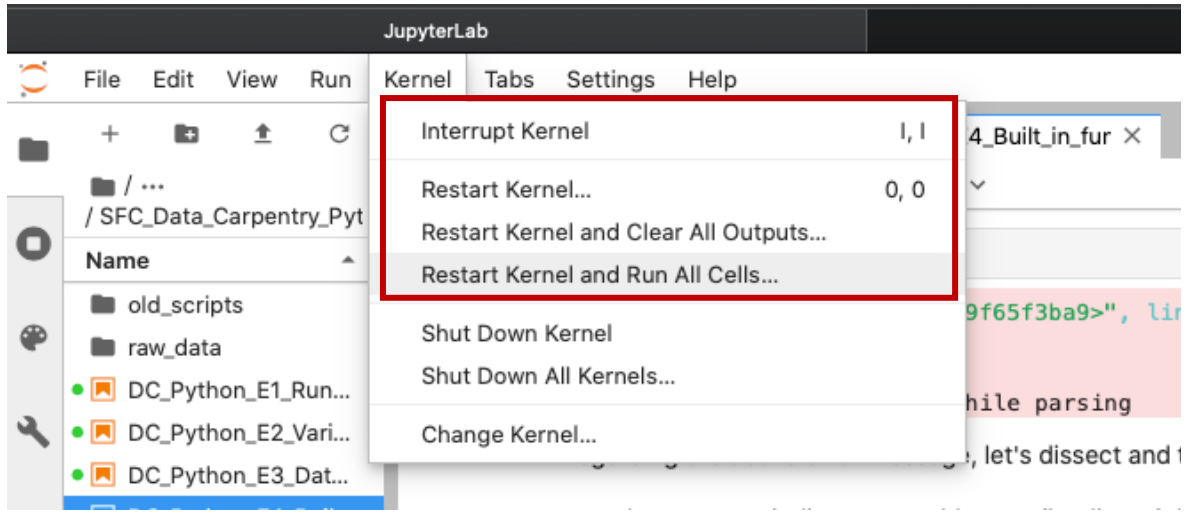
What to do?

- Fix syntax error(s) by proof-reading the source and runtime error(s) by *tracing the execution process*.

[ ]:

Episode #4 – '**Built-in Functions & Help**' – *Interpreting error messages..*

Handling a frozen cell issue in Jupyter Notebook.. (usually indicated by an asterisk in square brackets on the far left before the specific cell or cells: '**[*]**')



- If ***restarting the 'kernel' and 're-running' the cells*** won't resolve the issue, that is *recommended to restart the web browser* (*including the Anaconda-Navigator*).

- *If the problem is still present/ onset, restart your computer..*

Otherwise, the issue needs deeper technical investigation.. !

Episode #4 – '**Built-in Functions & Help**' – *Looking for help ..*

1. Use the built-in function **help()** to get help for a spceific function

Every built-in function has online/ offline documentation.

[29]:
```
help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

Episode #4 – '**Built-in Functions & Help**' – *Looking for help ..*

2. In Jupyter Notebook there are further explicit ways to get help about a given function

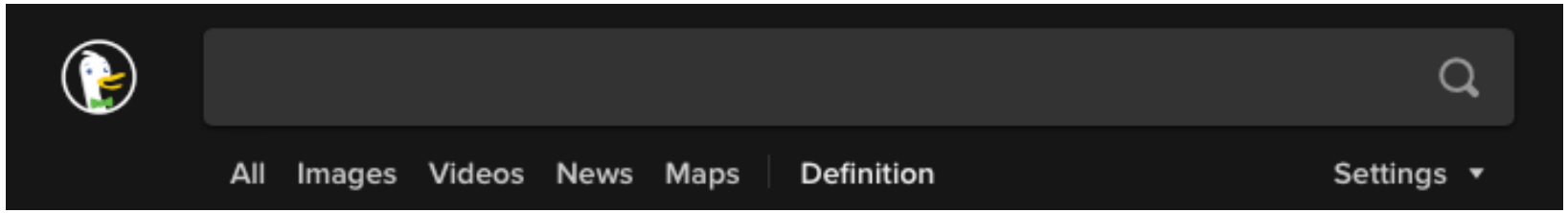
- Place the cursor anywhere in the function invocation (i.e. the function's name or its parameters), hold down **shift**, and **press tab**.
- Or type a specific function's name with a question mark after it (e.g. **round?** - *NB: without parentheses!*).

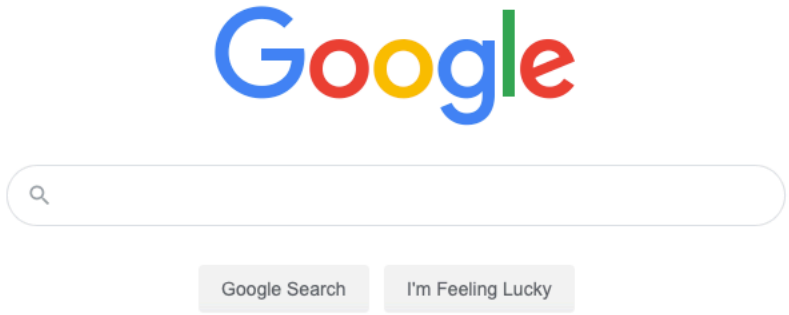
[31]:
```
round()
```

```
Signature: round(number, ndigits=None)
Docstring:
Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None.  Otherwise
the return value has the same type as the number.  ndigits may be negative.
Type:      builtin_function_or_method
```

Episode #4 – '**Built-in Functions & Help**' – *Looking for help ..*

Otherwise…





Episode #4 – **JupyterLab – Finger Exercise 4.1** **(~ 3 minutes)**

Order of function call execution for nested functions AND order of execution of arithmetic operations..

What Happens When

1. Explain in simple terms the order of operations in the following program: when does the addition happen, when does the subtraction happen, when is each function called, etc.
2. What is the final value of `radiance`?

Python

```
radiance = 1.0
radiance = max(2.1, 2.0 + min(radiance, 1.1 * radiance - 0.5))
```

Episode #4 – **JupyterLab – Finger Exercise 4.2** **(~ 4 minutes)**

Spot the Difference

1. Predict what each of the `print` statements in the program below will print.
2. Does `max(len(rich), poor)` run or produce an error message? If it runs, does its result make any sense?

Python

```
easy_string = "abc"
print(max(easy_string))
rich = "gold"
poor = "tin"
print(max(rich, poor))
print(max(len(rich), len(poor)))
```

Episode #4 – **JupyterLab – Finger Exercise 4.3** **(~ 2 minutes)**

Why Not?

Why don't `max` and `min` return `None` when they are given no arguments?

Any Qs for Episode #4 – ‘Built-in Functions & Help’?

Summary >> We’ve talked about..

Use comments to add documentation to programs.
A function may take zero or more arguments.
Commonly-used built-in functions include max(), min(), and round().
Functions may only work for certain (combinations of) arguments.
Functions may have default values for some arguments.
Seek for help
Every function returns something.
Python reports a syntax error when it can’t understand the source of a program.
Python reports a runtime error when something goes wrong while a program is executing.

Any questions or comments?

Well done! See you ALL tomorrow morning at 9 am!

robert.nagy@ed.ac.uk

Have your word! Please provide your feedback on *Etherpad after the end of the Python session*!