

Machine Learning with Python

Lab 1

Intro to Jupyter Notebook

1	LAB SETUP	1
2	STARTING JUPYTER NOTEBOOK.....	1
3	CREATING AN EMPTY WORKBOOK	3
4	BASIC CELL OPERATIONS	4
5	SAVING AND CHECKPOINTING.....	9
6	KEYBOARD SHORTCUTS.....	10
7	MARKDOWN	11
8	WORKING WITH THE KERNEL	13
9	MAGIC COMMANDS	16
10	SHARING YOUR NOTEBOOK VIA GOOGLE COLAB	17

1 Lab setup

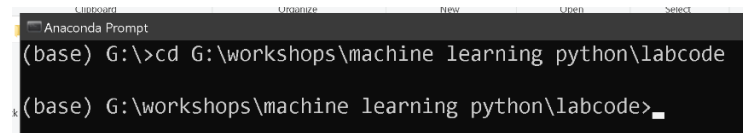
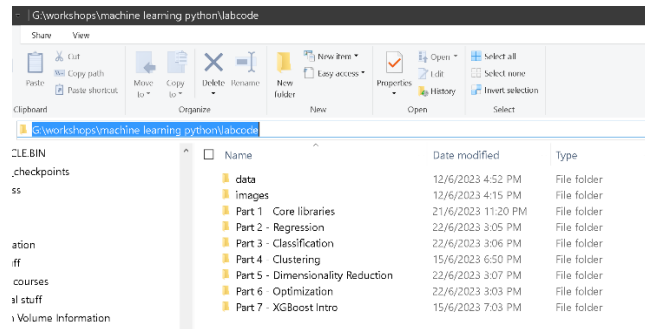
Make sure you have the following items installed (using either Anaconda or Miniconda / Miniforge)

- Python
- Jupyter Notebook
- Relevant data processing and visualization libraries (pandas, matplotlib, seaborn)
- Relevant machine learning libraries (scikit-learn, xgboost)
- A suitable text editor (Notepad ++)

2 Starting Jupyter notebook

Jupyter Notebook is a powerful and very popular tool for interactively developing and presenting data science and machine learning / deep learning projects. It combines code, visualizations, narrative text, and other rich media into a single document. At its core, a notebook is a document that blends code and its output seamlessly. It allows you to run code, display the results, and add explanations, formulas, and charts all in one place. This makes your work more transparent, understandable, and reproducible. They have become an essential part of the data science workflow in companies and organizations worldwide. They enable data scientists to explore data, test hypotheses, and share insights efficiently.

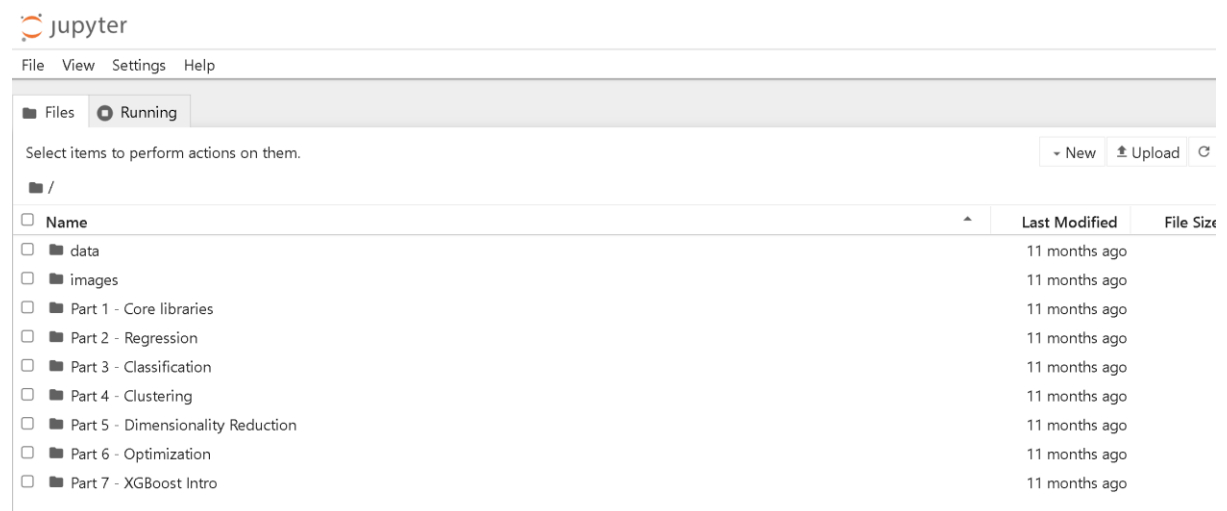
Open an Anaconda prompt / Miniforge prompt (depending on which installation approach you used), and then navigate into the folder `labcode` from the zip file that you extracted for this workshop using the `cd` command. You can copy and paste from the address bar in your Windows explorer to get the correct path following the `cd` command.



Open a new browser tab (use Chrome or Firefox) and then start Jupyter notebook by typing into the prompt:

```
jupyter notebook
```

You should see the main dashboard of the notebook in your browser tab, which shows the files and subfolders in the folder that the notebook was opened in (`labcode`):



```

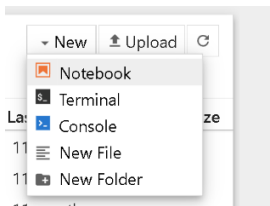
Anaconda Prompt: jupyter notebook
(base) G:\workshops\machine learning python\labcode>
(base) G:\workshops\machine learning python\labcode>
(base) G:\workshops\machine learning python\labcode>jupyter notebook
[I 2024-06-02 11:24:19.686 ServerApp] Package notebook took 0.0000s to import
[I 2024-06-02 11:24:19.896 ServerApp] Package aext_assistant took 0.2089s to import
[I 2024-06-02 11:24:19.898 ServerApp] Package aext_core took 0.0023s to import
[I 2024-06-02 11:24:19.913 ServerApp] ***** ENVIRONMENT Environment.PRODUCTION *****
[I 2024-06-02 11:24:19.916 ServerApp] ***** ENVIRONMENT Environment.PRODUCTION *****
[I 2024-06-02 11:24:19.918 ServerApp] Package aext_panels took 0.0197s to import
[I 2024-06-02 11:24:19.921 ServerApp] Package aext_share_notebook took 0.0023s to import
[I 2024-06-02 11:24:19.937 ServerApp] Package jupyter_lsp took 0.0159s to import
[W 2024-06-02 11:24:19.937 ServerApp] A `_jupyter_server_extension_points` function was not found in jupyter_lsp. Instead, a `_jupyter_server_extension_paths` function was found and will be used for now. This function name will be deprecated in future releases of Jupyter Server.

```

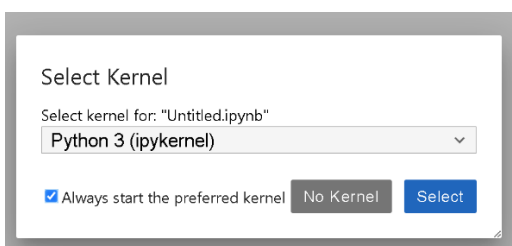
Jupyter's Notebooks and dashboard are web apps, and Jupyter starts up a local Python server to serve these apps to your web browser, making it essentially platform-independent. The prompt window currently shows a variety of messages related to the starting up of the Jupyter notebook web app. You can later scan the latest messages in this window in order to perform trouble shooting / debugging if the notebook behaves in an unexpected manner.

3 Creating an empty workbook

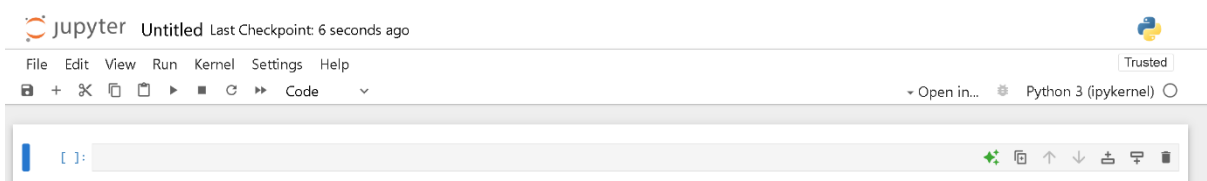
To create a new empty notebook, select the New Notebook option from the New drop down menu option in the upper right corner (notice that you also have options to create a New empty file or New folder in the current folder that the Notebook app is running in).



You may be prompted to select a Kernel. Select the Always start the preferred kernel and click Select.

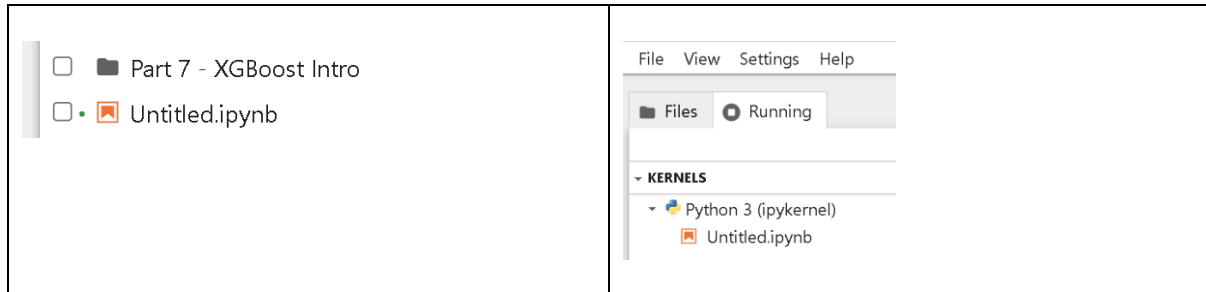


The new empty Jupyter notebook opens in a new browser tab, each notebook uses its own tab while allows you to open and work with multiple notebooks simultaneously in your browser.



If you switch back to the main Jupyter dashboard, you will see the default name of the new notebook (Untitled.ipynb) and the Running tab should show that this notebook is currently active and running using the default Python 3 ipykernel.

A kernel is a computational engine that executes the code contained in a notebook document - Jupyter's default kernel is Python 3, but you can also install kernels for other programming languages.



Each time you create a new notebook, a new .ipynb file will be created. This is a text file that describes the contents of your notebook in a format called JSON. Each cell and its contents, including image attachments that have been converted into strings of text, is listed therein along with some metadata.

You can open up this .ipynb file directly from Windows explorer using Notepad++ to verify this. We will almost always modify the contents of this file using the Notebook interface in Jupyter, rather than directly using a text editor such as Notepad++

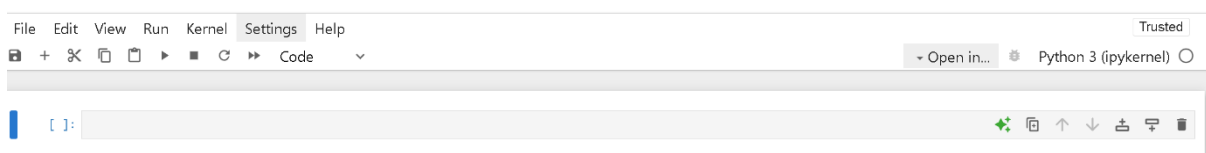
```
{
  "cells": [
    {
      "cell_type": "code",
      "execution_count": null,
      "id": "74372932-b072-4c53-be7a-fb9203d379ef",
      "metadata": {},
      "outputs": [],
      "source": []
    }
  ],
}
```

4 Basic cell operations

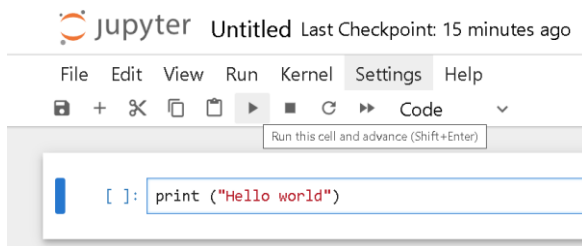
A notebook body consists of multiple cells. There are two main cell types:

- A code cell contains code to be executed in the kernel. When the code is run, the notebook displays the output below the code cell that generated it.
- A Markdown cell contains text formatted using Markdown and displays its output in-place when the Markdown cell is run.

The first cell in a new notebook is always a code cell.



Type a print command for some random text into this cell and click the Run button to execute it. You can use Shift + Enter as a shortcut for the Run button.



Once the cell is executed, the output from its execution is displayed immediately below it and the label on the left is now numbered with 1. A new cell is now generated automatically below, which is now the active cell.

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

```
[ ]:
```

Go ahead and type a few more random statements, including assigning value to a variable and printing the content of that variable in each of these consecutive new cells and execute them.

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

```
[2]: print ("This is cool")
      This is cool
```

```
[3]: print ("I like Python")
      I like Python
```

```
[4]: x = 5
```

```
[5]: print ("value of x is ", x)
      value of x is 5
```

Each code cell can contain one or more statements (including loops, conditional statements, function definitions, etc). Executing a code cell will execute all the code in it, but not the code in any other cell.

So the cell is a self-contained block of code that you want to execute, and allows you to execute certain parts of your program independently of other parts. This feature is of course different from the sequential execution from top to bottom in a normal Python script, and is very useful in data science projects where we often only want to execute certain key parts of the project (rather than the entire project; which may require a long time to complete execution).

Notice that each of the cells have a labelled number next to them after they have been executed. This number indicates the sequence or order of execution of the particular code in the cell with regards to all cells in the notebook.

NOTE: Depending on the particular version of Notebook you are using and how it was installed, the numbering might be odd consecutive numbers: 1, 3, 5, 7 instead of sequential 1, 2, 3, 4. The actual number itself is not so important, we use the order of the numbers to decide which the sequence of code cell execution. So for e.g. code cell 4 was executed after code cell 2, or code cell 7 was executed after code cell 5, etc.

Go up to the first 2 cells and execute them again. Notice that their sequence execution number now changes accordingly.

```
[6]: print ("Hello world")
      Hello world

[7]: print ("This is cool")
      This is cool

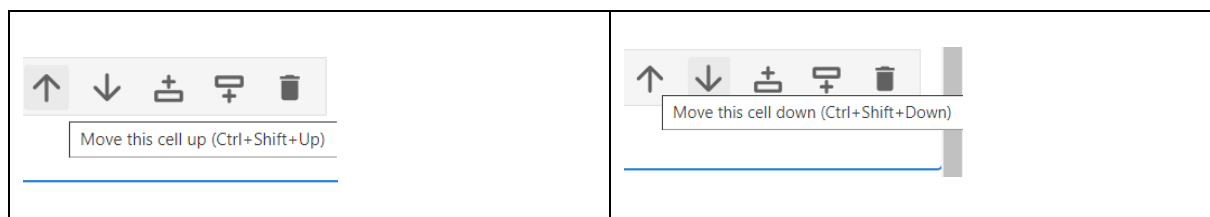
[3]: print ("I like Python")
      I like Python

[4]: x = 5
```

The icon options to the right corner of each code cell gives you a chance to perform an action on any cell.

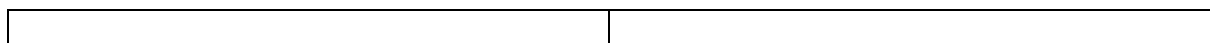


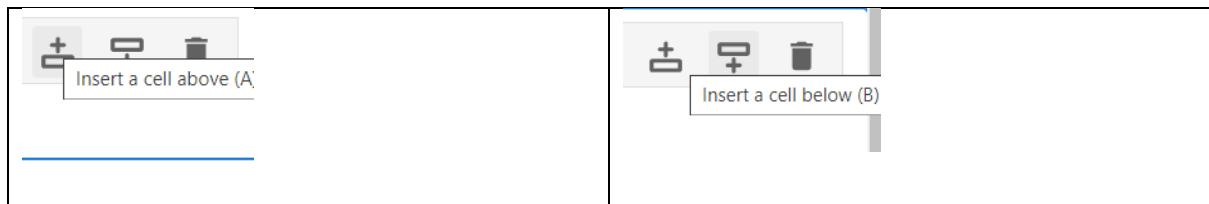
For e.g. you can use the up and down arrow keys to move the current cell up and down through the existing sequence of cells.



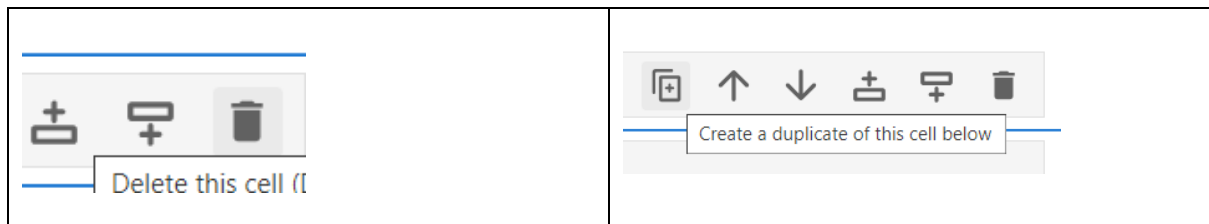
Notice that the label number does not change regardless of the current new position of the cell. This is because the label number is NOT RELATED to positioning, instead it tells us the sequence in which that particular code cell was executed with regards to the overall order execution of all the other code cells. So its perfectly fine to reposition cells so that you have a code cell with a very large number positioned right at the top of the notebook, and another code cell with a very low number positioned right at the bottom of the notebook.

You can also add new cells below and above the current cell, and then execute those cells in the usual way. Try and practice this for yourself with some random print statements for the new cells.

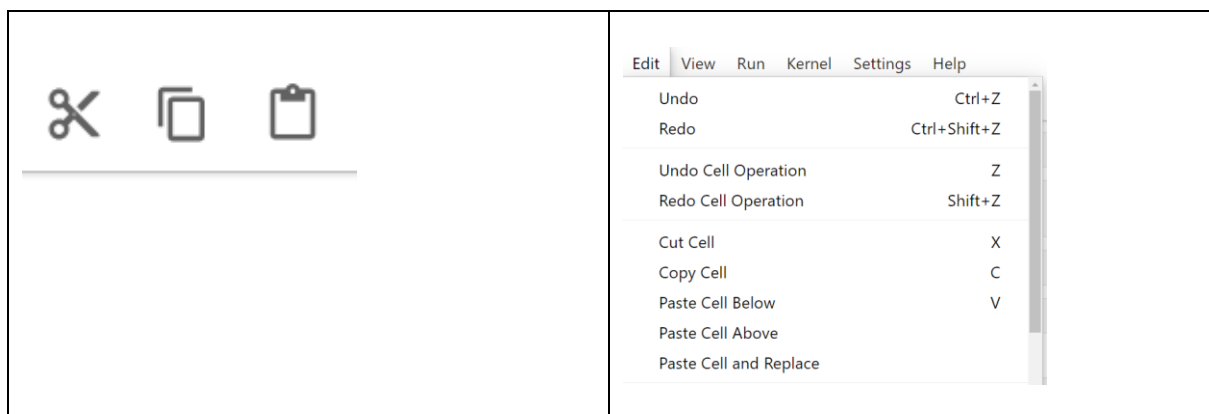




You can also create a duplicate of the current cell immediately below and also delete the current cell. Try and practice this for yourself.



You can also cut, copy and paste cells to various positions in the notebook using the various options in the Edit menu or the short cut options.



You can also split content in an existing cell or merge content from multiple cells into a single cell. Type the following 4 single statements into a single cell:

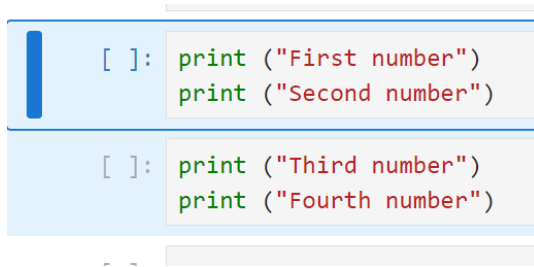
```
print ("First number")
print ("Second number")
print ("Third number")
print ("Fourth number")
```

Now position your cursor at the place where you wish to split the content (say at the end of the 2nd statement), and then select Edit -> Split cell. You should now see the content being split into 2 separate cells.

```
[ ]: print ("First number")
      print ("Second number")
```

```
[ ]: print ("Third number")
      print ("Fourth number")
```

Likewise, we can now merge back two or more separate cells to get a single cell. Select one of the cells and then use Shift + up/down arrow key to select the next consecutive cell either below or above it.



Then select Edit -> Merge Selected Cells. If you are only merging two consecutive cells (rather than a sequence of many cells), there is a shorter option Edit -> Merge Cell Above / Merge Cell Below -> without the need to select cells beforehand.

Insert a new cell anywhere in the notebook and place these statements within it (which will cause execution to pause for about 5 seconds).

```
import time
time.sleep(5)
```

Execute this cell. Notice that the cell label number changes to * temporarily while waiting for the cell to complete execution. When it does, the cell label will hold an appropriate number in the current sequencing.

```
[*]: import time
time.sleep(5)
```

Try and define a simple function in a cell somewhere close to the bottom of the worksheet and execute it:

```
def add(a, b):
    sum = a + b
    return sum
```

Then call that function in a cell somewhere above its definition, and execute that cell:

```
result = add(3, 5)
print("The result is ", result)
```

Notice that this works perfectly fine since the function definition exists (was executed) before it was called – and we can determine this sequence of execution based on the cell label relative numbering. It doesn't matter the position of the code cell containing the function definition and function calling is relative to each other.


```
[14]: result = add(3, 5)
      print ("The result is ", result)
```

The result is 8

```
[9]: y = 10
```

```
[5]: print ("value of x is ", x)
```

value of x is 5

```
[12]: def add(a, b):
      sum = a + b
      return sum
```

This is the reason for the importance of the cell label numbers and why Notebook is different from a standard IDE (like Spyder or Pycharm), where code is generally executed in the order in which it appears: from top to bottom.

To remove all the cell label numbers, you can select Edit -> Clear outputs of all cells.

However, this does not reset the number sequencing back to 1. If you randomly pick another cell to execute, the label number will continue with the next number in the original sequence.

Also clearing outputs does not remove variable values that have already been assigned or functions that have already been declared. Notice that you will still be able to execute the cells that print variables or call functions without executing the cells that declare those variables or define the functions first.

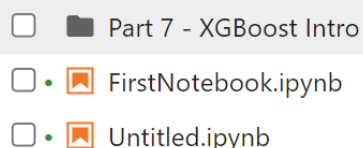
To completely reset number sequencing and remove all existing variable values and function declaration, you will need to reset the kernel (see later)

5 Saving and checkpointing

All newly generated notebooks have a default name (untitled1, untitled2, etc). You can override this default name with your own name when you save with File -> Save Notebook As

Give your first notebook a suitable name, for e.g. `FirstNotebook`

Notice that in the main dashboard tab you will be able to see a new file corresponding to this chosen name, plus another file with the original name Untitled. Subsequently, you can just use Ctrl+S to save your file in the future.

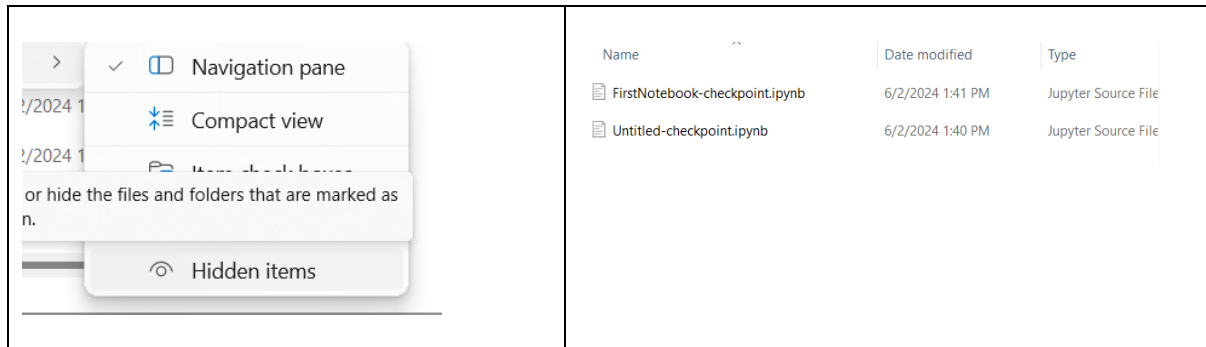


```

☐ Part 7 - XGBoost Intro
☐ • FirstNotebook.ipynb
☐ • Untitled.ipynb
  
```

Whenever you start working with a new notebook, Jupyter will automatically save the contents of the notebook periodically (typically every 120 seconds) even if you don't explicitly save the file. This automatic background saving is called checkpointing and the checkpointed files are stored within a

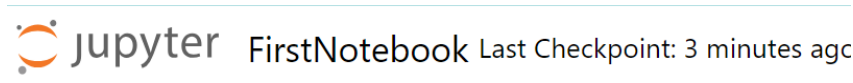
hidden subdirectory of your save location called `.ipynb_checkpoints` and is also a `.ipynb` file. You can verify this from Windows explorer by ensuring that the option View Hidden Items is selected:



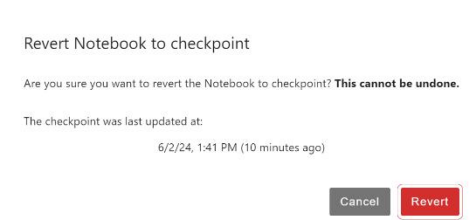
There are therefore 2 saved versions of the notebook on hard disk: the last saved content (performed with Ctrl – S) stored in the file `xxxx.ipynb` and the most recent checkpoint after the last save, stored in `xxxx-checkpoint.ipynb`. This checkpoint enables you to recover your unsaved work in the event of an unexpected issue.

Every time you save content explicitly with Ctrl – S, this will also save the current content as well as create a new checkpoint simultaneously with the current content: thereby synchronizing the content of `xxxx.ipynb` and `xxxx-checkpoint.ipynb`

The header of the current notebook will show when the last checkpoint was created



If you wish to revert the content of your notebook back to the most recent checkpoint in the event of an unexpected issue or error, use File -> Revert Notebook to checkpoint. Note that this revert operation cannot be undone (it will permanently overwrite the current notebook content) and you will be warned accordingly.



6 Keyboard shortcuts

The current active or highlighted cell can be in one of two modes:

- Edit mode - cursor is visible in the editor area of the cell. The editor area border is highlighted in blue
- Command mode - cursor is not in the editor area of the cell. The area

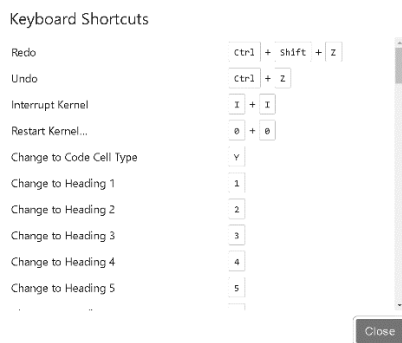
To switch between these two modes, either click in the editor area of the cell or outside of it. You can also use Esc and Enter to toggle between these two modes.

Keyboard shortcuts are a very popular aspect of the Jupyter environment because they facilitate a faster cell-based workflow. They require a cell to be in command mode.

Once in command mode, these are some of the more well known shortcuts:

- Scroll up and down the cells with Up and Down keys.
- Press A or B to insert a new cell above or below the active cell.
- D + D (D twice) will delete the active cell.
- Z will undo the latest cell operation (deletion, etc)

You can see the complete list of Keyboard shortcuts with Help -> Show Keyboard shortcuts

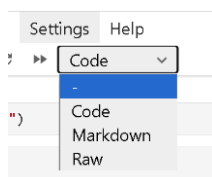


7 Markdown

Markdown is a lightweight, easy to learn markup language for formatting plain text and is used for annotations that you add to the notebook to elaborate on the content of the code cells.

All newly created / inserted cells in the notebook are by default code cells.
Insert a new code cell anywhere in the notebook.

To toggle this new cell between a Markdown / Code cell, you can either use the drop down option at the top or the M (change to markdown) / Y (Change to code) keyboard shortcut in command mode.



A Markdown cell does not have a label number next to it. This is how we distinguish it from a code cell.



Insert the following text below into the Markdown cell. It contains some common examples of Markdown syntax formatting.

```
# This is a level 1 heading

## This is a level 2 heading

### This is a level 3 heading

This is some plain text that forms a paragraph.
Add emphasis via bold and bold, or italic and italic.

Paragraphs must be separated by an empty line.
```

Execute the cell to see the effect of the Markdown formatting.

Insert another new Markdown cell and insert the following text into that cell to see some more common examples of Markdown syntax formatting.

```
We can have unordered lists, which are bulleted using asterisks

* cat
* dog
* mouse

We can also have ordered lists

1. Malaysia
2. Singapore

It is possible to have hyperlinks
[Visit this exciting website](https://www.cnn.com)

To show code blocks, place them within triple backticks:
```
while number < 5 :
 print("Thank you")
 number = number + 1
```
```

Execute the cell to see the effect of the Markdown formatting.

You can double click the generated Markdown to further edit it. However, making changes to Markdown directly inside the Notebook can sometimes result in funny errors, so best is to do it in a separate text editor like Notepad++.

For more references on Markdown:

<https://www.datacamp.com/tutorial/markdown-in-jupyter-notebook>

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

8 Working with the kernel

All notebooks are created with a kernel (the computation engine) to support execution of code within the code cells. The default kernel for Jupyter notebook is Python 3, but you can install new kernels for other languages (C++, Java, C#), so you can run code in these languages using the Notebook style of execution.

The kernel's state persists over time and between cells — it pertains to the document as a whole and not individual cells. So if you import a library or define a function in one cell and execute that cell, the library or function is persisted into the kernel and can subsequently be accessed by any other code cell.

We saw this earlier that regardless of the positioning of the code cells, as long as you have executed the definition of a function, you can now access the function in any other code cell:

```
[16]: result = add(3, 5)
      print ("The result is ", result)
```

The result is 8

```
[12]: x = 5
```





```
[13]: print ("Value of x is ", x)
```

Value of x is 5

```
[15]: def add(a, b):
      sum = a + b
      return sum
```

The kernel is available from the Jupyter server app that is actively running in background with messages from it coming out in the prompt window.

This means if you accidentally close all the browser tab for any of your notebooks, you can still return to the main dashboard and double click on the notebook file name to return to it in a new browser tab.

- ☐  Part 7 - XGBoost Intro
- ☒  FirstNotebook.ipynb
- ☐  Untitled.ipynb
- ☒  Untitled1.ipynb

If you accidentally close the main dashboard as well, you can access it again by typing:

<http://localhost:8888/tree>

in the same browser that opened it in.

If you are using a different browser instead, you will need to copy and use the URL shown in the prompt window:

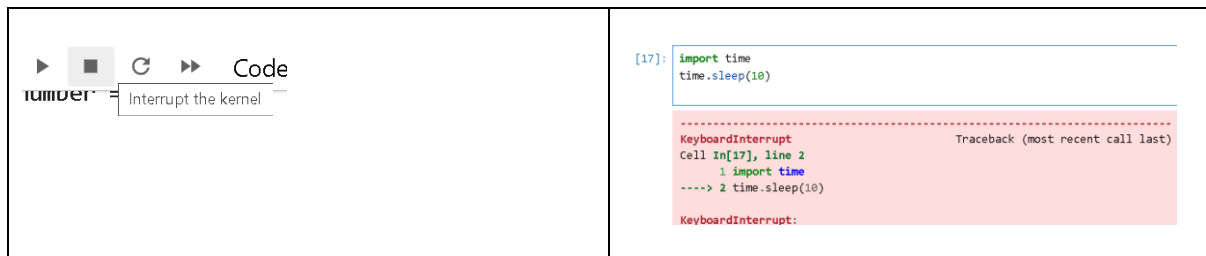
```
action).
[C 2024-06-02 14:42:02.915 ServerApp]

To access the server, open this file in a browser:
file:///C:/Users/User/AppData/Roaming/jupyter/runtime/jpserver-14580-open.html
Or copy and paste one of these URLs:
http://localhost:8888/tree?token=cc0c2eb2ee42194441fba0f98b8dcfa0f1d19ba836a0b2ae
http://127.0.0.1:8888/tree?token=cc0c2eb2ee42194441fba0f98b8dcfa0f1d19ba836a0b2ae
```

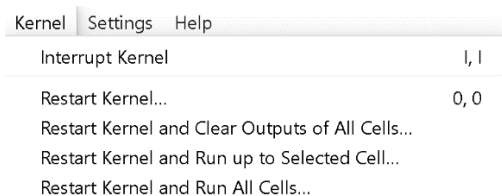
You can stop the kernel anytime if a particular code cell execution is taking too long. Type this into a new cell and execute it to simulate a situation of excessively long code cell execution:

```
import time
time.sleep(10)
```

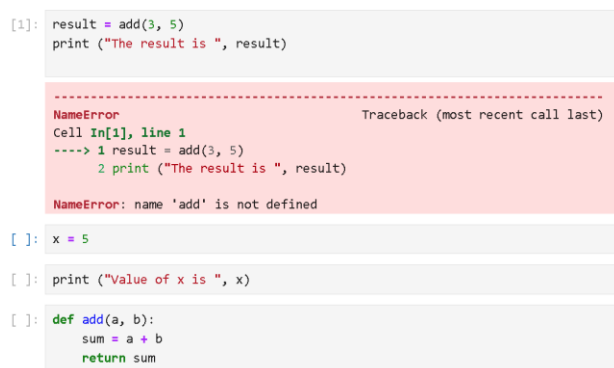
You can then use the interrupt button to terminate execution, which will produce output indicating this appropriately.



You can also restart the kernel and perform a variety of other associated actions at the same time from the main menu.



When you restart the kernel, the cell execution label numbering will be reset to count again from 1. At the same time, all library imports and function definitions will be erased. This means if you try to execute a cell that references a function before the function definition is recreated again, you will get an error, for e.g.



You can also shut down the kernel for a particular set of notebooks if that kernel execution is consuming too much system resources:

Kernel -> Shutdown kernel

The notebook header will indicate appropriately:



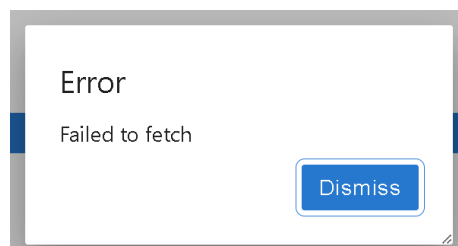
To restart the kernel, you can click on any cell and execute it.

You can shutdown the kernel and also the Jupyter notebook webapp server with File -> Shutdown. You will be able to see the webapp server has terminated in the prompt window.

```
[I 2024-06-02 15:51:16.667 ServerApp] Kernel shutdown: 8ec205fe-2141-48b5-aeed-6d511b51eb5b
[I 2024-06-02 15:51:16.669 ServerApp] Kernel shutdown: 39078e33-4932-4e31-bab8-d935f37841c2
[I 2024-06-02 15:51:16.994 ServerApp] Shutting down on /api/shutdown request.
[I 2024-06-02 15:51:16.994 ServerApp] Shutting down 6 extensions

(base) G:\temp\templabcode>
(base) G:\temp\templabcode>
```

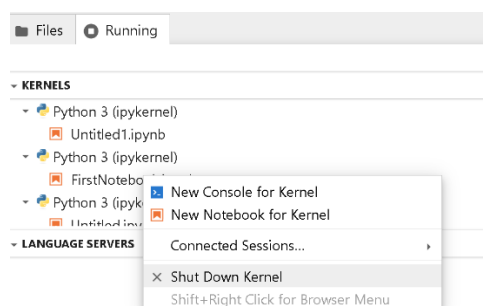
With the server down, the main dashboard will no longer be operational and if you try to issue commands like creating a new notebook from the dashboard, you will get an error message:



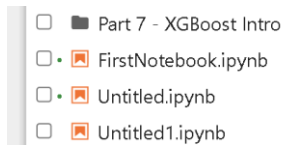
In this case, you will need to restart the Jupyter notebook webapp server from the prompt by typing into the prompt:

```
jupyter notebook
```

The Running tab of the main dashboard shows all the actively running kernels for all the active notebooks. You can shutdown the kernels for individual notebooks or shutdown all the actively running kernels from here:



The notebooks with actively running kernels are also shown with green dots next to their file names in the main dashboard.



If a kernel for a notebook is shutdown from the main dashboard (rather than from inside that notebook itself), you can no longer revive or restart the kernel from that notebook. Instead, you will need to close that notebook tab in the browser and reopen that notebook from the main dashboard by double clicking on its filename.

9 Magic commands

Magic commands are special commands that can help you with running and analyzing data in your notebook. They add a special functionality that is not straight forward to achieve with python code or Jupyter notebook interface.

There are two kinds of magics:

- a) Line magics operate on a single line of a code cell
- b) Cell magics operate on the entire code cell in which they are called

If automagic is on (which is default in newer versions of notebook), you can run a magic simply by typing it on its own line in a code cell, and running the cell. If it is off, you will need to put % before line magics and %% before cell magics to use them. However, its usually best to use the % and %% to ensure that users understand that they are magic commands.

Create a new code cell and type into it and run:

```
%lsmagic
```

You should see a list of all line and cell magics available.

To get more detailed info on what each of them are supposed to do, type:

```
%quickref
```

You can get more info on the cell and line magic commands by appending ? to the end of the individual magic command, for e.g.

```
%pwd?
```

Try typing all these line magics that perform basic directory operations (check current directory, create new directory, move into new directory, delete existing directory, etc) into new code cells:

```
pwd
ls
ldir
mkdir coolstuff
```



```
ls
cd coolstuff
pwd
cd ..
pwd
rmdir coolstuff
ldir
```

Some more examples of common cell and line magics

<https://www.aboutdatablog.com/post/top-8-magic-commands-in-jupyter-notebook>

<https://medium.com/@marc.bolle/learn-these-15-magic-commands-in-jupyter-notebook-to-save-time-a864ca9b15c7>

Some particularly useful examples are `%who` and `%pinfo` to get more information about the various variables and objects declared and registered in the kernel.

10 Sharing your notebook via Google Colab

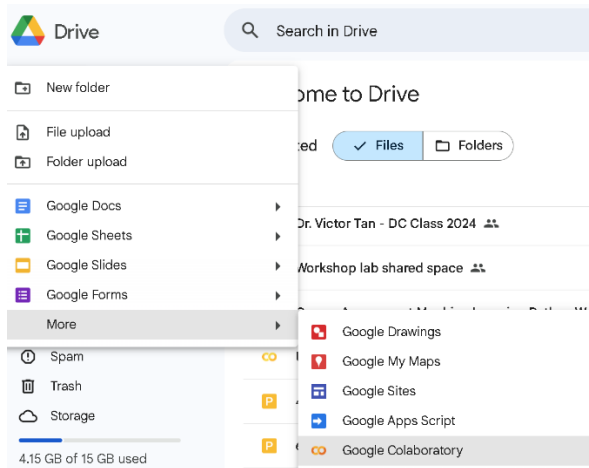
There are many ways to share your Jupyter notebook with others: via Google Colab or GitHub Gist are the most popular approaches. Alternatively, you can use any other standard cloud file sharing utility such as Google Drive, Dropbox, etc

Google Colaboratory, commonly known as Google Colab, is a cloud-based Jupyter notebook environment that provides a platform for writing and executing Python code through your browser. It's especially popular in the data science and machine learning communities.

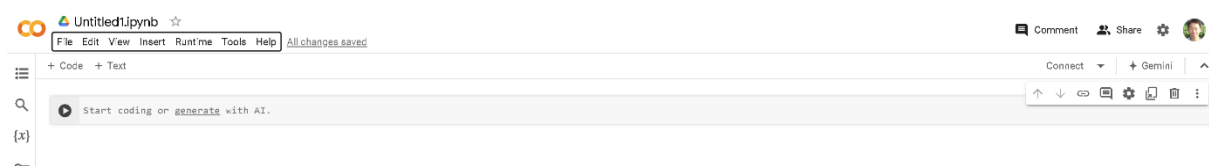
Google Colab offers free access to limited graphics processing unit (GPU) and TPUs (tensor processing units) capabilities, and you can pay more for increased capabilities. This is particularly useful for training deep learning models on the cloud, which required these increased resources.

Google Colab inherits Google Docs' collaborative features. You can share your notebooks, have others comment on them, and even edit them in real-time - an excellent feature for team projects or teaching. Google Colab comes pre-installed with popular Python libraries like TensorFlow, PyTorch, and Keras. This convenience allows you to jump right into coding without worrying about installing and updating libraries.

In Google Drive of your Google Account, select new Google Colaboratory:



The UI of the new Colab notebook is similar to Jupyter notebook:

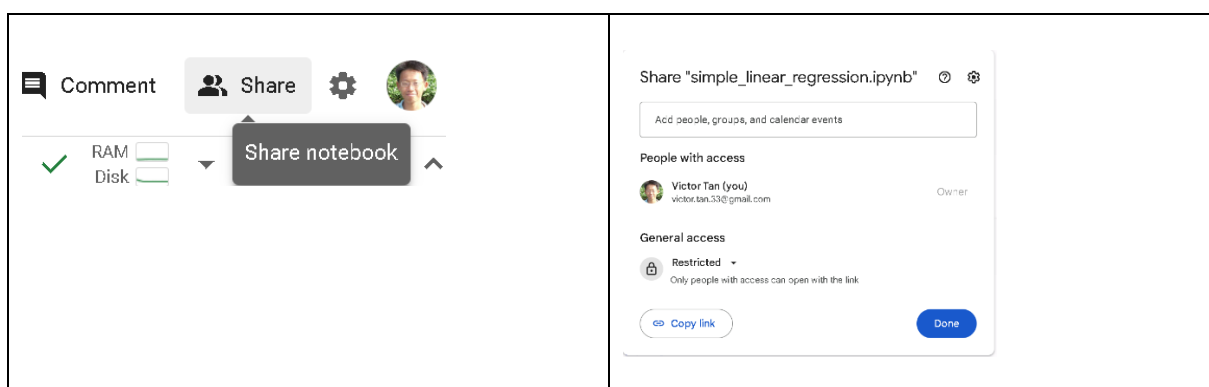


Select File -> Upload a file, and drag and drop your Jupyter ipynb file into the dialog box.

You can now edit and run the cells in the Colab notebook exactly as you have done with your Jupyter notebook.

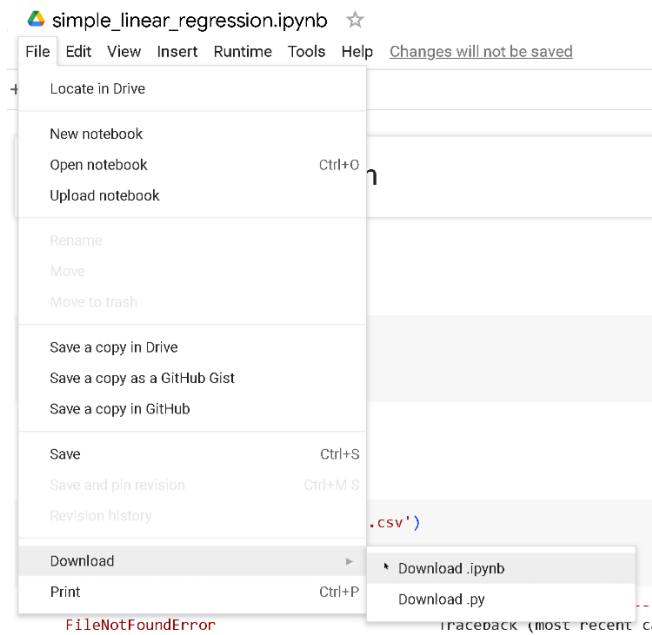
NOTE: For the rest of the workshop, **DO NOT run the Jupyter notebooks in Colab** as the free cloud GPU resources allocated for each Colab notebook and will not match the computational resources available on your own machine. Using higher capability resources from Colab is possible only with a paid subscription.

To share the notebook with someone else, select the Share option and configure the sharing options appropriately.



Then copy the link, and distribute to the individuals you want to share the notebook with.

Set the access rights to anyone with the link, copy the link into an alternative browser and you should be able to view the notebook and subsequently download it.



You can also share Jupyter notebooks via

<https://www.geeksforgeeks.org/how-to-export-and-share-jupyter-nootbooks/>