How to use Jupyter Notebook[1]

What is Jupyter Notebook?

The Jupyter Notebook is an incredibly powerful tool for interactively developing and presenting data science projects. This article will walk you through how to use Jupyter Notebooks for data science projects and how to set it up on your local machine.

First, though: **what is a "notebook"?**

A notebook integrates code and its output into a single document that combines visualizations, narrative text, mathematical equations, and other rich media. In other words: it's a single document where you can run code, display the output, and also add explanations, formulas, charts, and make your work more transparent, understandable, repeatable, and shareable.

Using Notebooks is now a major part of the data science workflow at companies across the globe. If your goal is to work with data, using a Notebook will speed up your workflow and make it easier to communicate and share your results.

Best of all, as part of the open source Project Jupyter, Jupyter Notebooks are completely free. You can download the software on its own, or as part of the Anaconda data science toolkit.

Although it is possible to use many different programming languages in Jupyter Notebooks, this article will focus on Python, as it is the most common use case. (Among R users, R Studio tends to be a more popular choice).

How to follow this tutorial?

To get the most out of this tutorial you should be familiar with programming — Python and pandas specifically. That said, if you have experience with another language, the Python in this article shouldn't be too cryptic, and will still help you get Jupyter Notebooks set up locally.

Jupyter Notebooks can also act as a flexible platform for getting to grips with pandas and even Python, as will become apparent in this tutorial.

We will:

- Cover the basics of installing Jupyter and creating your first notebook
- Delve deeper and learn all the important terminology
- Explore how easily notebooks can be shared and published online.

Installation

The easiest way for a beginner to get started with Jupyter Notebooks is by installing Anaconda.

[1] Reference: https://www.dataquest.io/blog/jupyter-notebook-tutorial/

Anaconda is the most widely used Python distribution for data science and comes pre-loaded with all the most popular libraries and tools.

Some of the biggest Python libraries included in Anaconda include NumPy, pandas, Matplotlib, etc.

Anaconda thus lets us hit the ground running with a fully stocked data science workshop without the hassle of managing countless installations or worrying about dependencies and OS-specific (read: Windows-specific) installation issues.

To get Anaconda, simply:

- Download the latest version of Anaconda for Python 3.8.
- Install Anaconda by following the instructions on the download page and/or in the executable.

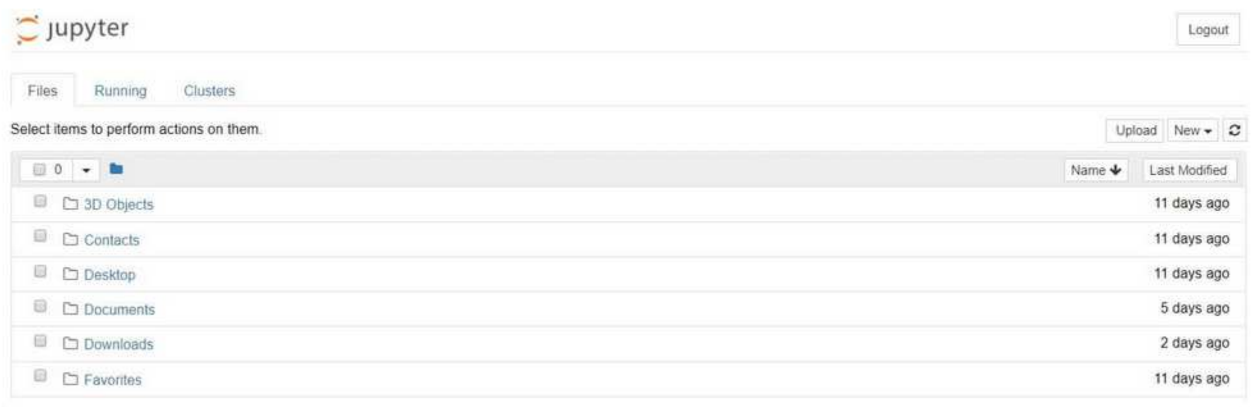
If you are a more advanced user with Python already installed and prefer to manage your packages manually, you can just use pip:

Creating your first notebook

In this section, we're going to learn to run and save notebooks, familiarize ourselves with their structure, and understand the interface. We'll become intimate with some core terminology that will steer you towards a practical understanding of how to use Jupyter Notebooks by yourself and set us up for the next section, which walks through an example data analysis and brings everything we learn here to life.

Running Jupyter

On Windows, you can run Jupyter via the shortcut Anaconda adds to your start menu, which will open a new tab in your default web browser that should look something like the following screenshot.



Select items to perform actions on them.

3D Objects	11 days ago
Contacts	11 days ago
Desktop	11 days ago
Documents	5 days ago
Downloads	2 days ago
Favorites	11 days ago

This isn't a notebook just yet, but don't panic! There's not much to it. This is the Notebook Dashboard, specifically designed for managing your Jupyter Notebooks. Think of it as the launchpad for exploring, editing and creating your notebooks.

Be aware that the dashboard will give you access only to the files and sub-folders contained within Jupyter's start-up directory (i.e., where Jupyter or Anaconda is installed). However, the start-up directory can be changed.

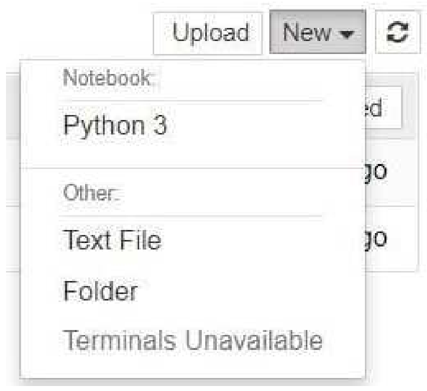
It is also possible to start the dashboard on any system via the command prompt (or terminal on Unix systems) by entering the command jupyter notebook; in this case, the current working directory will be the start-up directory.

With Jupyter Notebook open in your browser, you may have noticed that the URL for the dashboard is something like https://localhost:8888/tree. Localhost is not a website, but indicates that the content is being served from your *local* machine: your own computer.

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 and opening the door to easier sharing on the web.

(If you don't understand this yet, don't worry — the important point is just that although Jupyter Notebooks opens in your browser, it's being hosted and run on your local machine. Your notebooks aren't actually on the web until you decide to share them.)

The dashboard's interface is mostly self-explanatory — though we will come back to it briefly later. So what are we waiting for? Browse to the folder in which you would like to create your first notebook, click the "New" drop-down button in the top-right and select "Python 3":



Hey presto, here we are! Your first Jupyter Notebook will open in new tab — each notebook uses its own tab because you can open multiple notebooks simultaneously.

If you switch back to the dashboard, you will see the new file Untitled.ipynb and you should see some green text that tells you your notebook is running.

What is an ipynb File?

The short answer: each .ipynb file is one notebook, so each time you create a new notebook, a new .ipynb file will be created.

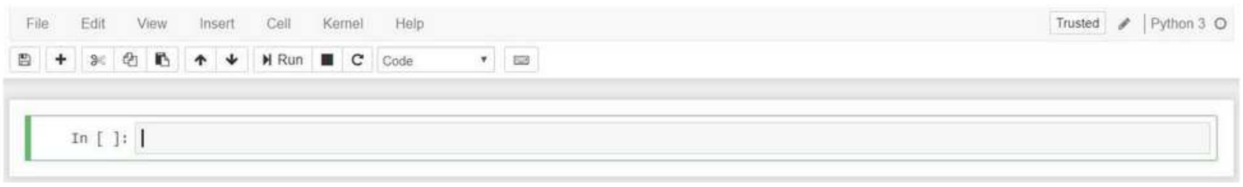
The longer answer: Each .ipynb file is a text file that describes the contents of your notebook in a format called JSON. Each cell and its contents, including image attachments that have been converted into strings of text, is listed therein along with some metadata.

You can edit this yourself — if you know what you are doing! — by selecting "Edit > Edit Notebook Metadata" from the menu bar in the notebook. You can also view the contents of your notebook files by selecting "Edit" from the controls on the dashboard

However, the key word there is *can.* In most cases, there's no reason you should ever need to edit your notebook metadata manually.

The Notebook interface

Now that you have an open notebook in front of you, its interface will hopefully not look entirely alien. After all, Jupyter is essentially just an advanced word processor.



Why not take a look around? Check out the menus to get a feel for it, especially take a few moments to scroll down the list of commands in the command palette, which is the small button with the keyboard icon (or Ctrl + Shift + P).

There are two fairly prominent terms that you should notice, which are probably new to you: *cells* and *kernels* are key both to understanding Jupyter and to what makes it more than just a word processor. Fortunately, these concepts are not difficult to understand.

- A **kernel** is a "computational engine" that executes the code contained in a notebook document.
- A **cell** is a container for text to be displayed in the notebook or code to be executed by the notebook's kernel.

Cells

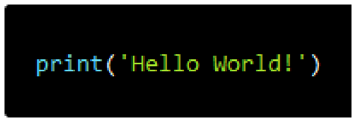
Cells form the body of a notebook. In the screenshot of a new notebook in the section above, that box with the green outline is an empty cell. There are two main cell types that we will cover:

- 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.

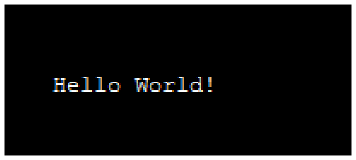
The first cell in a new notebook is always a code cell.

Let’s test it out with a classic hello world example: Type print('Hello World!') into the cell and click the run button Run in the toolbar above or press Ctrl + Enter.

The result should look like this:



```
print('Hello World!')
```



```
Hello World!
```

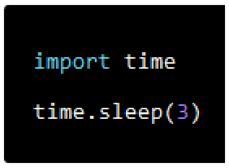
When we run the cell, its output is displayed below and the label to its left will have changed from In [ ] to In [1].

The output of a code cell also forms part of the document, which is why you can see it in this article. You can always tell the difference between code and Markdown cells because code cells have that label on the left and Markdown cells do not.

The “In” part of the label is simply short for “Input,” while the label number indicates *when* the cell was executed on the kernel — in this case the cell was executed first.

Run the cell again and the label will change to In [2] because now the cell was the second to be run on the kernel. It will become clearer why this is so useful later on when we take a closer look at kernels.

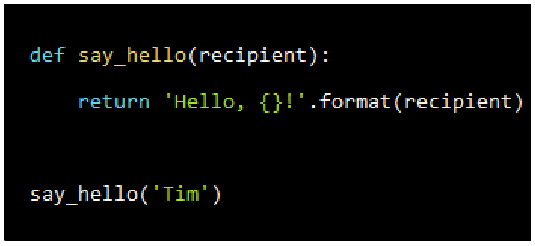
From the menu bar, click *Insert* and select *Insert Cell Below* to create a new code cell underneath your first and try out the following code to see what happens. Do you notice anything different?



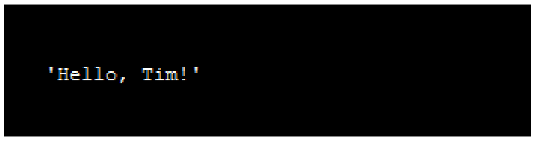
```
import time

time.sleep(3)
```

This cell doesn’t produce any output, but it does take three seconds to execute. Notice how Jupyter signifies when the cell is currently running by changing its label to In [*].

In general, the output of a cell comes from any text data specifically printed during the cell's execution, as well as the value of the last line in the cell, be it a lone variable, a function call, or something else. For example:



```python
def say_hello(recipient):
    return 'Hello, {}!'.format(recipient)

say_hello('Tim')
```



```
'Hello, Tim!'
```

You'll find yourself using this almost constantly in your own projects, and we'll see more of it later on.

Keyboard short cuts

One final thing you may have observed when running your cells is that their border turns blue, whereas it was green while you were editing. In a Jupyter Notebook, there is always one "active" cell highlighted with a border whose color denotes its current mode:

- **Green outline** — cell is in "edit mode"
- **Blue outline** — cell is in "command mode"

So what can we do to a cell when it's in command mode? So far, we have seen how to run a cell with Ctrl + Enter, but there are plenty of other commands we can use. The best way to use them is with keyboard shortcuts

Keyboard shortcuts are a very popular aspect of the Jupyter environment because they facilitate a speedy cell-based workflow. Many of these are actions you can carry out on the active cell when it's in command mode.

Below, you'll find a list of some of Jupyter's keyboard shortcuts. You don't need to memorize them all immediately, but this list should give you a good idea of what's possible.

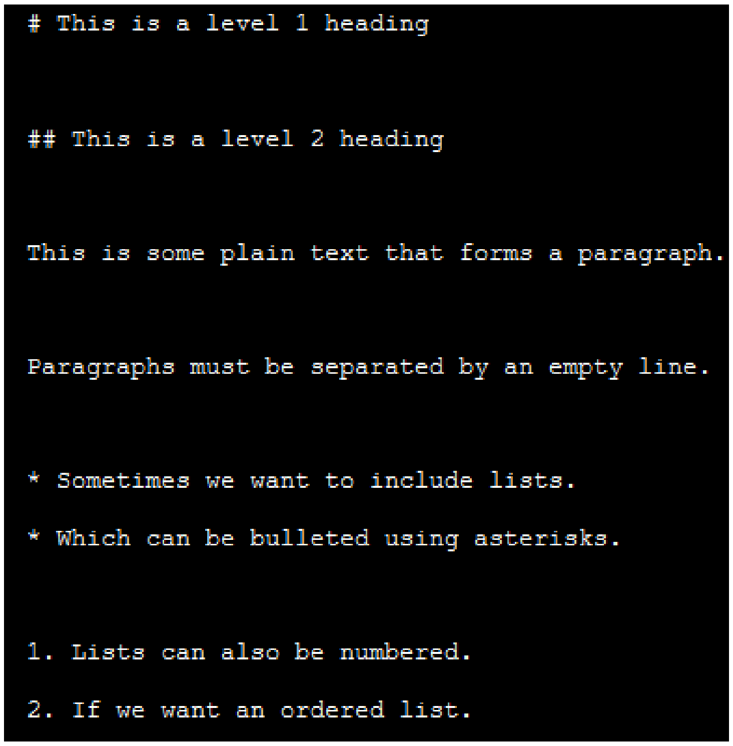
- Toggle between edit and command mode with Esc and Enter, respectively.
- Once in command mode:
  - Scroll up and down your cells with your Up and Down keys.
  - Press A or B to insert a new cell above or below the active cell.
  - M will transform the active cell to a Markdown cell.
  - Y will set the active cell to a code cell.
  - D + D (D twice) will delete the active cell.
  - Z will undo cell deletion.
  - Hold Shift and press Up or Down to select multiple cells at once. With multiple cells selected, Shift + M will merge your selection.
- Ctrl + Shift + -, in edit mode, will split the active cell at the cursor.
- You can also click and Shift + Click in the margin to the left of your cells to select them.

Go ahead and try these out in your own notebook. Once you're ready, create a new Markdown cell and we'll learn how to format the text in our notebooks.

Markdown

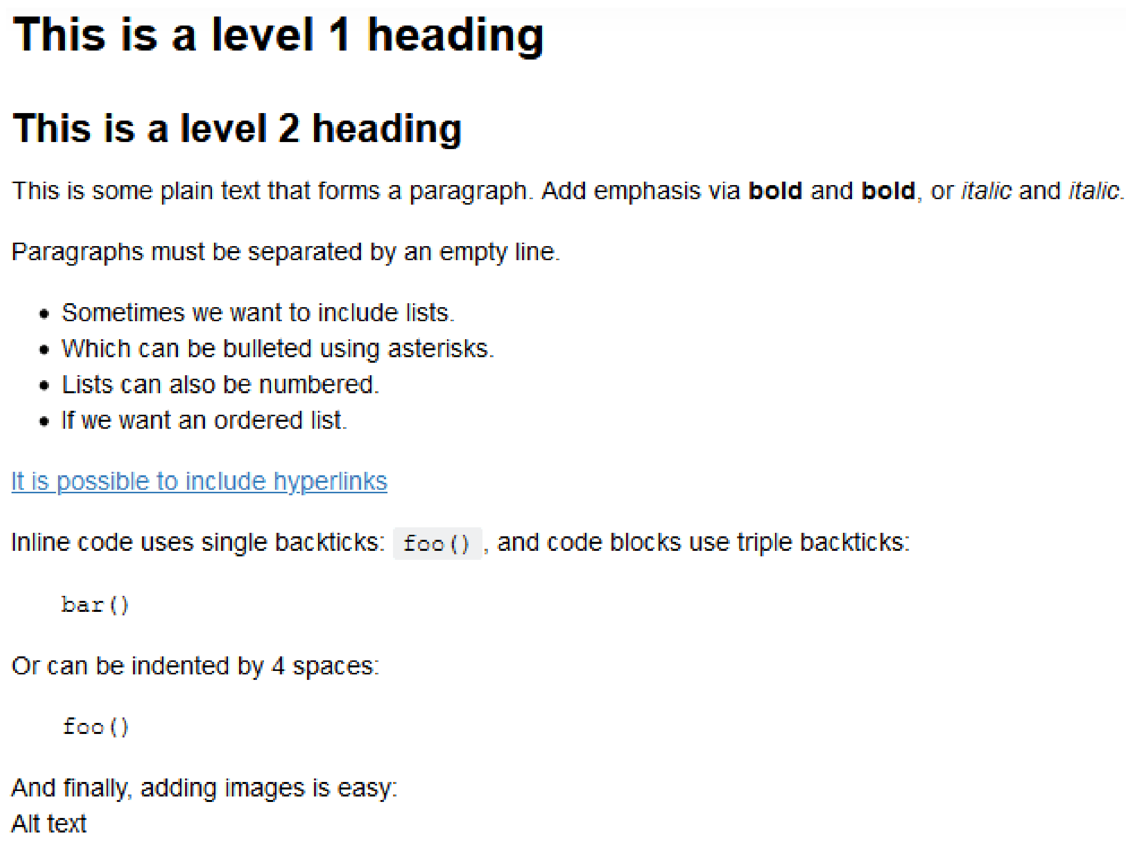
Markdown is a lightweight, easy to learn markup language for formatting plain text. Its syntax has a one-to-one correspondence with HTML tags, so some prior knowledge here would be helpful but is definitely not a prerequisite.

Remember that this article was written in a Jupyter notebook, so all of the narrative text and images you have seen so far were achieved writing in Markdown. Let's cover the basics with a quick example:



```
# This is a level 1 heading


## This is a level 2 heading


This is some plain text that forms a paragraph.


Paragraphs must be separated by an empty line.


* Sometimes we want to include lists.

* Which can be bulleted using asterisks.


1. Lists can also be numbered.

2. If we want an ordered list.
```

Here's how that Markdown would look once you run the cell to render it:


This is a level 1 heading

This is a level 2 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.

- Sometimes we want to include lists.
- Which can be bulleted using asterisks.
- Lists can also be numbered.
- If we want an ordered list.

It is possible to include hyperlinks

Inline code uses single backticks: `foo()`, and code blocks use triple backticks:

```
bar()
```

Or can be indented by 4 spaces:

```
foo()
```

And finally, adding images is easy:
Alt text

When attaching images, you have three options:

- Use a URL to an image on the web.
- Use a local URL to an image that you will be keeping alongside your notebook, such as in the same git repo.
- Add an attachment via "Edit > Insert Image"; this will convert the image into a string and store it inside your notebook .ipynb file. Note that this will make your .ipynb file much larger!

There is plenty more to Markdown, especially around hyperlinking, and it's also possible to simply include plain HTML. Once you find yourself pushing the limits of the basics above, you can refer to the official guide from Markdown's creator, John Gruber, on his website.

Kernels

Behind every notebook runs a kernel. When you run a code cell, that code is executed within the kernel. Any output is returned back to the cell to be displayed. The kernel's state persists over time and between cells — it pertains to the document as a whole and not individual cells.

For example, if you import libraries or declare variables in one cell, they will be available in another. Let's try this out to get a feel for it. First, we'll import a Python package and define a function:

```python
import numpy as np

def square(x):

    return x * x
```

Once we've executed the cell above, we can reference `np` and `square` in any other cell.

```python
x = np.random.randint(1, 10)

y = square(x)

print('%d squared is %d' % (x, y))
```

```
 1 squared is 1
```

This will work regardless of the order of the cells in your notebook. As long as a cell has been run, any variables you declared or libraries you imported will be available in other cells.

You can try it yourself, let's print out our variables again.

```python
print('Is %d squared %d?' % (x, y))
```

```
 Is 1 squared 1?
```

No surprises here! But what happens if we change the value of `y`?

```python
y = 10

print('Is %d squared is %d?' % (x, y))
```

If we run the cell above, what do you think would happen?

We will get an output like: Is 4 squared 10?. This is because once we've run the y = 10 code cell, y is no longer equal to the square of x in the kernel.

Most of the time when you create a notebook, the flow will be top-to-bottom. But it's common to go back to make changes. When we do need to make changes to an earlier cell, the order of execution we can see on the left of each cell, such as In [6], can help us diagnose problems by seeing what order the cells have run in.

And if we ever wish to reset things, there are several incredibly useful options from the Kernel menu:

- Restart: restarts the kernel, thus clearing all the variables etc that were defined.
- Restart & Clear Output: same as above but will also wipe the output displayed below your code cells.
- Restart & Run All: same as above but will also run all your cells in order from first to last.

If your kernel is ever stuck on a computation and you wish to stop it, you can choose the Interrupt option.

Choosing a Kernel

You may have noticed that Jupyter gives you the option to change kernel, and in fact there are many different options to choose from. Back when you created a new notebook from the dashboard by selecting a Python version, you were actually choosing which kernel to use.

There kernels for different versions of Python, and also for over 100 languages including Java, C, and even Fortran. Data scientists may be particularly interested in the kernels for R and Julia, as well as both imatlab and the Calysto MATLAB Kernel for Matlab.

The SoS kernel provides multi-language support within a single notebook.

Each kernel has its own installation instructions, but will likely require you to run some commands on your computer.

Example Analysis

Now we've looked at *what* a Jupyter Notebook is, it's time to look at *how* they're used in practice, which should give us clearer understanding of *why* they are so popular.

It's finally time to get started with that Fortune 500 data set mentioned earlier.

It's worth noting that everyone will develop their own preferences and style, but the general principles still apply. You can follow along with this section in your own notebook if you wish, or use this as a guide to creating your own approach.

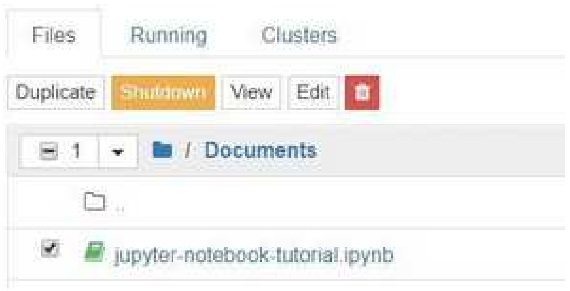
Naming the Notebook

Before you start writing your project, you'll probably want to give it a meaningful name. file name Untitled in the upper left of the screen to enter a new file name, and hit the Save icon (which looks like a floppy disk) below it to save.

Note that closing the notebook tab in your browser will **not** "close" your notebook in the way closing a document in a traditional application will. The notebook's kernel will continue to run in the background and needs to be shut down before it is truly "closed" — though this is pretty handy if you accidentally close your tab or browser!

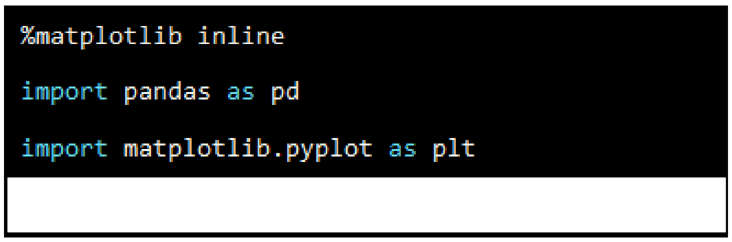
If the kernel is shut down, you can close the tab without worrying about whether it is still running or not.

The easiest way to do this is to select "File > Close and Halt" from the notebook menu. However, you can also shutdown the kernel either by going to "Kernel > Shutdown" from within the notebook app or by selecting the notebook in the dashboard and clicking "Shutdown" (see image below).

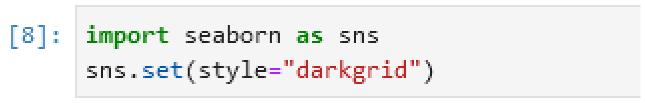


Setup

It's common to start off with a code cell specifically for imports and setup, so that if you choose to add or change anything, you can simply edit and re-run the cell without causing any side-effects.



```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
```

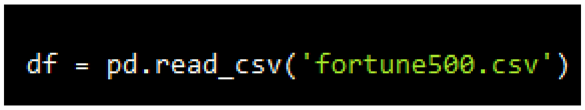


```
[8]: import seaborn as sns
     sns.set(style="darkgrid")
```

We'll import pandas to work with our data, Matplotlib to plot charts, and Seaborn to make our charts prettier. It's also common to import NumPy but in this case, pandas imports it for us.

That first line isn’t a Python command, but uses something called a line magic to instruct Jupyter to capture Matplotlib plots and render them in the cell output. We'll talk a bit more about line magics later, and they're also covered in our advanced Jupyter Notebooks tutorial.

For now, let’s go ahead and load our data.



```
df = pd.read_csv('fortune500.csv')
```

It’s sensible to also do this in a single cell, in case we need to reload it at any point.

Save and Checkpoint

Now we’ve got started, it’s best practice to save regularly. Pressing Ctrl + S will save our notebook by calling the “Save and Checkpoint” command, but what is this checkpoint thing?

Every time we create a new notebook, a checkpoint file is created along with the notebook file. It is located within a hidden subdirectory of your save location called .ipynb_checkpoints and is also a .ipynb file.

By default, Jupyter will autosave your notebook every 120 seconds to this checkpoint file without altering your primary notebook file. When you “Save and Checkpoint,” both the notebook and checkpoint files are updated. Hence, the checkpoint enables you to recover your unsaved work in the event of an unexpected issue.

You can revert to the checkpoint from the menu via “File > Revert to Checkpoint.”

Investigating the dataset

Now we’re really rolling! Our notebook is safely saved and we’ve loaded our data set `df` into the most-used pandas data structure, which is called a `DataFrame` and basically looks like a table. Below are the commands for querying the dataframe:

df.head() – show the first five rows in the dataframe.

df.tail() – show the last five rows in the dataframe.

len(df) – show the total number of rows in the dataframe.

df.columns = [‘’, ‘’, …, ‘’] – rename columns.

df.dtypes – show the data types.