Table of Contents

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.

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.

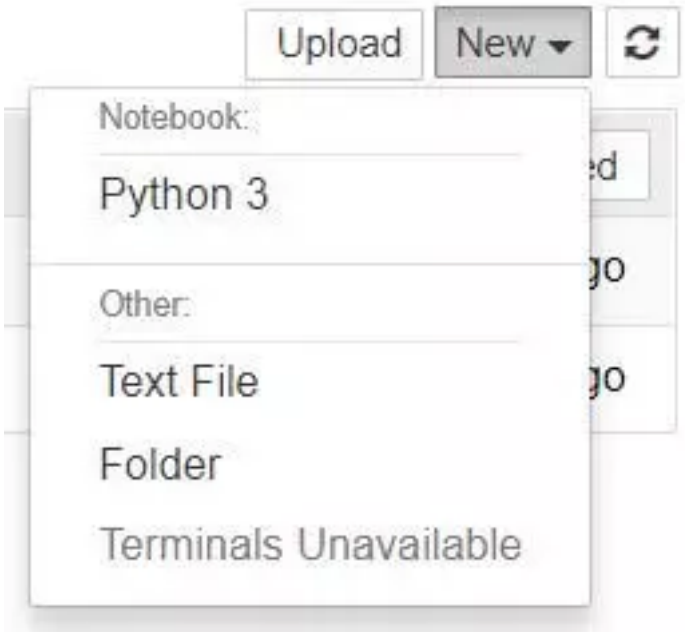
It is also possible to start the dashboard on any system via the command prompt (or terminal on Unix systems) by entering the command jupyter notebook; in this case, the current working directory will be the start-up directory.

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.

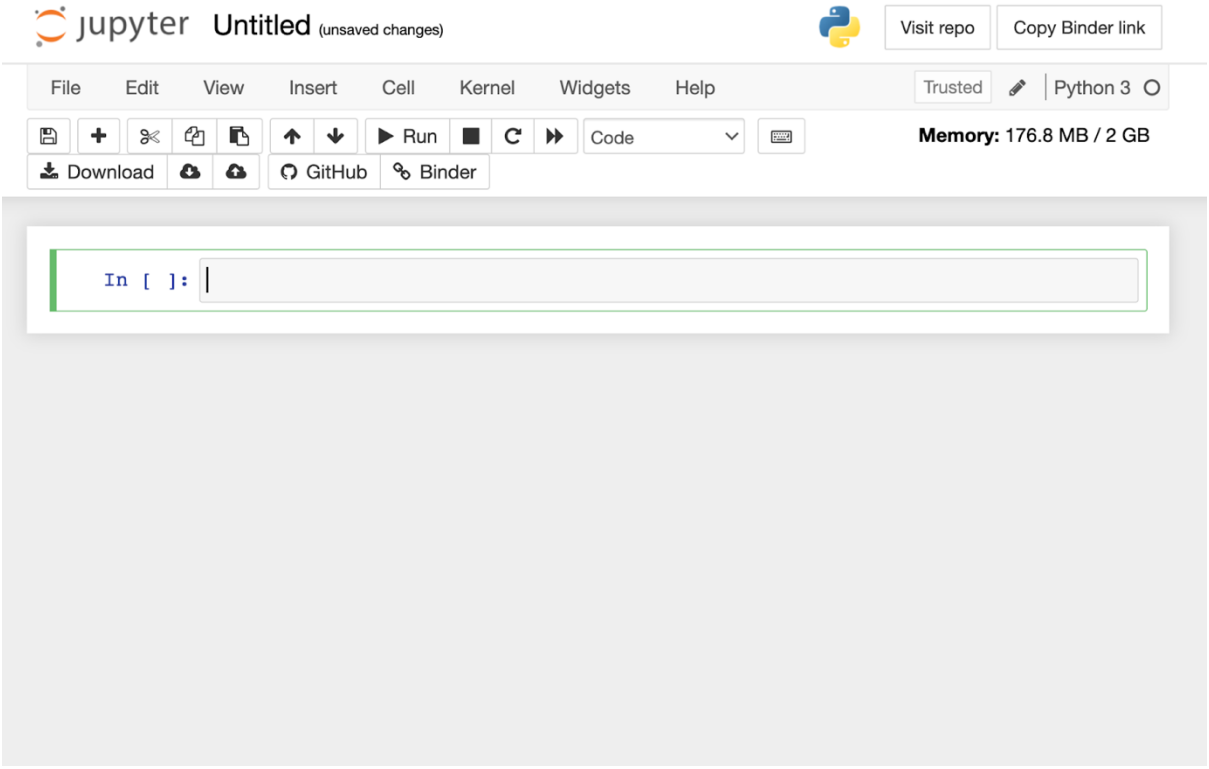
For Mac:

Open anaconda navigator using (command + spacebar)

Launch Jupyter Notebook.

Creating Your First Notebook

It’s time to create one of our first notebook documents.We’ll select “File”, then “New Notebook”.
this



Watch Video

Watch you tube video about jupyter notebook

https://www.youtube.com/watch?v=3jZYC9rGrNg

Notes:

Headings

- # for the titles
- ## for the main headings
- ### for the subheadings
- #### for the smaller subheadings
- ##### for the italic subheadings

Emphasis

- `__string__` or `**string**` for bold text
- `_string_` or `*string*` for italic text

Monospace fonts

- A back single quotation mark ` on both sides to get monospace fonts

Line breaks

- `<br>` wherever you want a line break, as the notebook sometimes doesn`t give you the required line break where you want it

Indenting

- `>` to indent the text
- `>>` for further indenting it, and so on

Bullets and numbering

- A single dash, i.e. - followed by two spaces to make bullet points
- A number and a dot followed by a space, i.e. `1.` to make numbered lists

Colouring

- `<font color = blue, yellow, red, pink, green, etc.> String </font>` to give your font any colour that you want

LaTeX Equations

- `$` on both the sides of the text to write LaTeX equations

Command mode shortcuts

- `Esc`: To go into command mode
- `Enter`: To go back to edit mode
- `M`: To convert a cell to a markdown cell
- `Y`: To convert a cell back to a code cell
- `A`: To insert a new cell above
- `B`: To insert a new cell below
- `D + D`: To delete cell
- `Z`: Undo the last operation

- `F`: To find and replace on your code
- `Shift + Up/Down`: To select multiple cells
- `Space`: Scroll notebook downwards
- `Shift + Space`: Scroll notebook upwards

Edit mode shortcuts

- `Shift + Enter`: To execute the code in the current cell and go to the next cell
- `Alt + Enter`: To execute the code in the current cell and insert new cell below
- `Shift + Tab`: To get a brief documentation of the object that you have just typed in the coding cell
- `Ctrl + Shift + -`: To split the cell at the cursor
- `Shift + M`: To merge selected cells

- Jupyter Notebook Mac shortcuts

Menus

In the above image, we can see a list of menus on our screen. Let's discuss what each of these menus does.

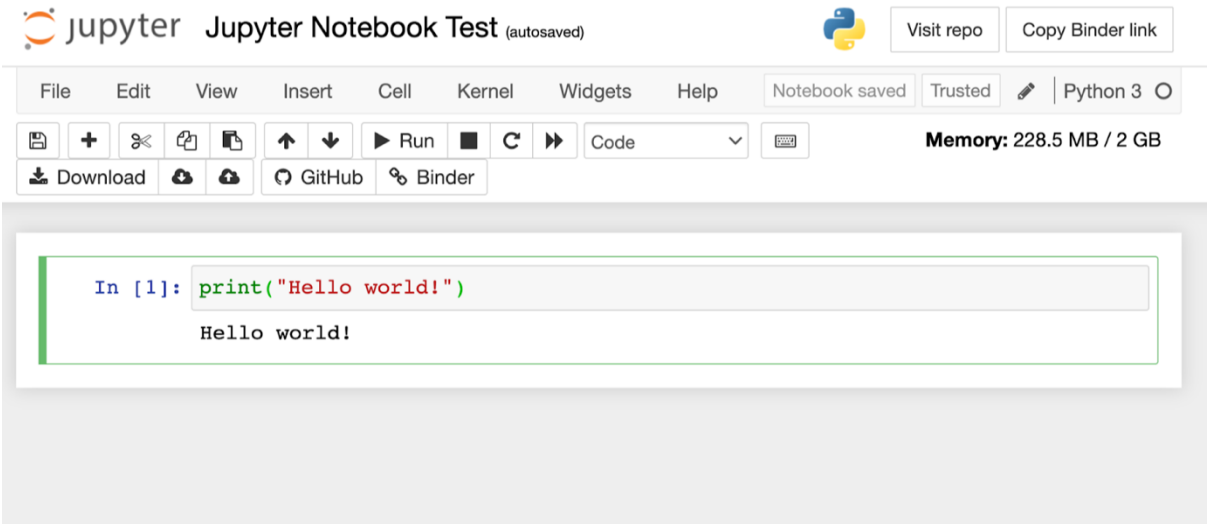
- **File**: Create new notebooks or open existing notebooks
- **Note**: We can use "File" to rename our notebooks. "File" also gives us the options to "Save and Checkpoint" and "Revert to Checkpoint". These options allow us to save our progress at different checkpoints and return back to them if we want to.
- **Edit**: Manipulate our cells
- **View**: Toggle headers, toolbars, and line numbers
- **Insert**: Insert cells above or below the selected cell
- **Cell**: Run our cells
- **Kernel**: Control the kernel
- **Widgets**: Save and clear widget state
- **Help**: Learn about Jupyter Notebook

Running cells

When we first create a new Jupyter Notebook, the **first cell defaults to using code and the kernel we selected at the beginning**. Since we started with Python 3, we can run Python code in our cells. Let's check it out! We can follow these steps:

1. Enter "print("Hello World!") into the first cell
2. Select the cell
3. Select "Run"

Here's what our notebook should look like now:



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:

```
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.

Add content to your notebook

Let's practice adding some content to our Jupyter Notebook. In this section, we'll discuss Jupyter cell types, including code and Markdown.

Jupyter cell types

The main cell types we use are **code cells and Markdown cells**. We practiced using code cells when we wrote our Hello World!, so let's take a look at the Markdown cell type.

We can use the "Insert" menu to insert a cell below our current one. Now, we can change our output type from "Code" to "Markdown". We won't spend much time discussing Markdown and its formatting, but let's practice creating headers and making lists.

Creating headers

To create a header in Markdown, all we need to do is use the pound # sign. One pound sign makes a heading 1, two pounds signs makes a heading 2, and so on.

Jupyter Notebook previews the headers for us:

```
# Heading 1
## Heading 2
### Heading
#### Heading 4
```

```
# Heading 1
## Heading 2
### Heading
#### Heading 4
```

When we select “Run”, the cell will format nicely like this:

Heading 1

Heading 2

Heading 3

Heading 4

Heading 1

Heading 2

Heading 3

Heading 4

Making lists

We can create bullet points in Markdown using single asterisks in place of bullets. Here’s what it’ll look like before we select “Run”:

```
* Item 1
* Item 2
* Item 3
```

```
* Item 1
* Item 2
* Item 3
```

Here’s what it looks like after we select “Run”:

- Item 1
- Item 2
- Item 3

- Item 1
- Item 2
- Item 3

Quiz

This is fun quiz link to play around the concepts.

https://quizizz.com/join/quiz/6396a426867650001d6249ca/start

==========

Additional Information:

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.

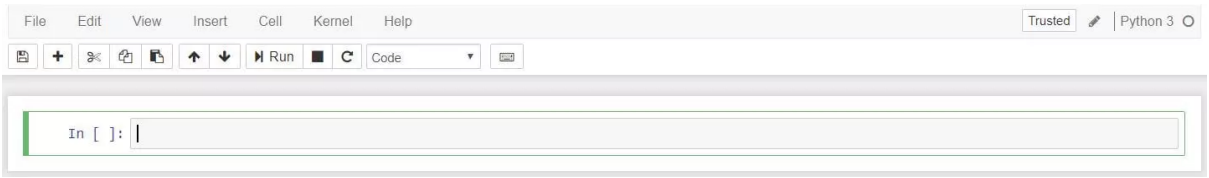
You can edit this yourself — if you know what you are doing! — by selecting “Edit > Edit Notebook Metadata” from the menu bar in the notebook. You can also view the contents of your notebook files by selecting “Edit” from the controls on the dashboard

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

We’ll return to kernels a little later, but first let’s come to grips with 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 

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.

Keyboard Shortcuts

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. 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.

1. Lists can also be numbered.
2. If we want an ordered list.

[It is possible to include hyperlinks](https://www.example.com)

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](https://www.example.com/image.jpg)
```

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

(Note that the alt text for the image is displayed here because we didn't actually use a valid image URL in our example)

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:

```
In [4]: 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.

```
In [6]: 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.

```
In [7]: print('Is %d squared %d?' % (x, y))
```

Is 1 squared 1?
No surprises here! But what happens if we change the value of y?

```
In [8]: 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.