How to Use Jupyter Notebook in 2020: A Beginner's Tutorial

dataquest.io/blog/jupyter-notebook-tutorial/

August 24, 2020

Published: August 24, 2020

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

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.

(In fact, this article was written as a Jupyter Notebook! It's published here in read-only form, but this is a good example of how versatile notebooks can be. In fact, most of our programming tutorials and even our Python courses were created using Jupyter Notebooks).

Example Data Analysis in a Jupyter Notebook

First, we will walk through setup and a sample analysis to answer a real-life question. This will demonstrate how the flow of a notebook makes data science tasks more intuitive for us as we work, and for others once it's time to share our work.

So, let's say you're a data analyst and you've been tasked with finding out how the profits of the largest companies in the US changed historically. You find a data set of Fortune 500 companies spanning over 50 years since the list's first publication in 1955, put together from Fortune's public archive. We've gone ahead and created a CSV of the data you can use here.

As we shall demonstrate, Jupyter Notebooks are perfectly suited for this investigation. First, let's go ahead and install Jupyter.

Installation

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

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, and Matplotlib, though the full 1000+ list is exhaustive.

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:

1. Download the latest version of Anaconda for Python 3.8.
2. 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:

```
pip3 install jupyter
```

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.

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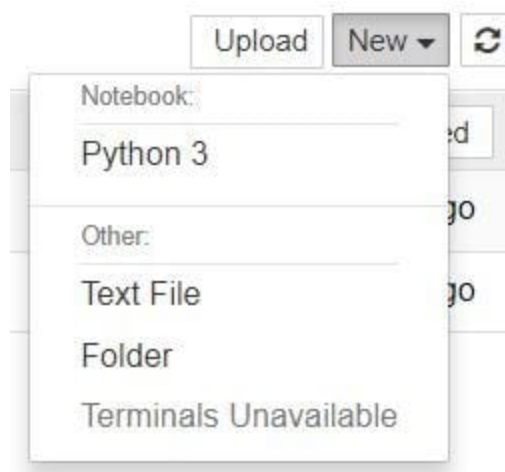
Hey presto, here we are! Your first Jupyter Notebook will open in new tab — each notebook uses its own tab because you can open multiple notebooks simultaneously.

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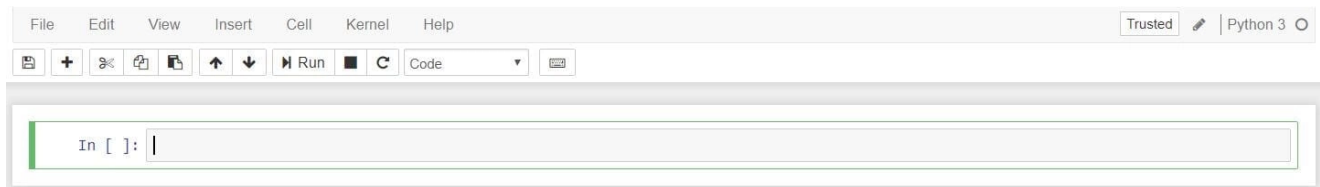
There are two fairly prominent terms that you should notice, which are probably new to you: *cells* and *kernels* are key both to understanding Jupyter and to what makes it more than just a word processor. Fortunately, these concepts are not difficult to understand.

- 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

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 `▶ 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:

```
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 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.

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.

Wow, that looks like an exponential, but it's got some huge dips. They must correspond to the early 1990s recession and the dot-com bubble. It's pretty interesting to see that in the data. But how come profits recovered to even higher levels post each recession?

That adds another side to the story. Revenues were not as badly hit — that's some great accounting work from the finance departments.

Sharing Your Notebooks

When people talk about sharing their notebooks, there are generally two paradigms they may be considering.

Most often, individuals share the end-result of their work, much like this article itself, which means sharing non-interactive, pre-rendered versions of their notebooks. However, it is also possible to collaborate on notebooks with the aid of version control systems such as Git or online platforms like Google Colab.

Before You Share

A shared notebook will appear exactly in the state it was in when you export or save it, including the output of any code cells. Therefore, to ensure that your notebook is share-ready, so to speak, there are a few steps you should take before sharing:

1. Click "Cell > All Output > Clear"
2. Click "Kernel > Restart & Run All"
3. Wait for your code cells to finish executing and check ran as expected

This will ensure your notebooks don't contain intermediary output, have a stale state, and execute in order at the time of sharing.

Exporting Your Notebooks

Jupyter has built-in support for exporting to HTML and PDF as well as several other formats, which you can find from the menu under "File > Download As."

If you wish to share your notebooks with a small private group, this functionality may well be all you need. Indeed, as many researchers in academic institutions are given some public or internal webspace, and because you can export a notebook to an HTML file, Jupyter Notebooks can be an especially convenient way for researchers to share their results with their peers.

But if sharing exported files doesn't cut it for you, there are also some immensely popular methods of sharing `.ipynb` files more directly on the web.
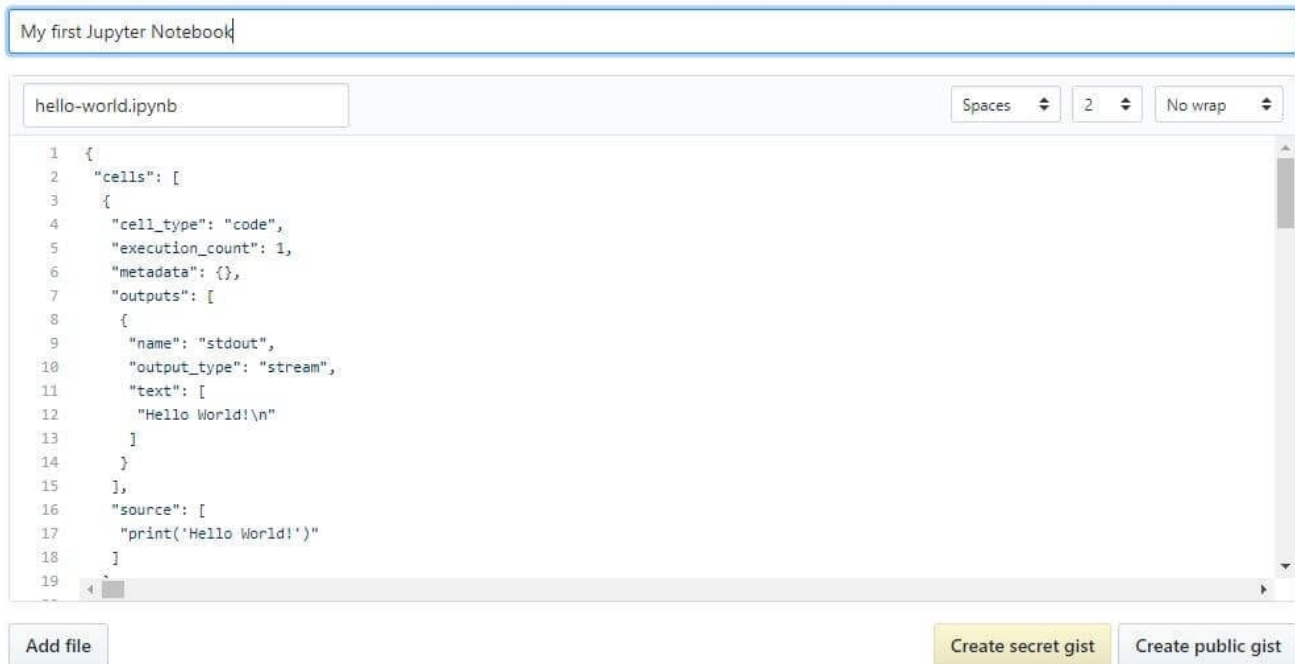
GitHub

With the number of public notebooks on GitHub exceeding 1.8 million by early 2018, it is surely the most popular independent platform for sharing Jupyter projects with the world. GitHub has integrated support for rendering `.ipynb` files directly both in repositories and gists on its website. If you aren't already aware, GitHub is a code hosting platform for version control and collaboration for repositories created with Git. You'll need an account to use their services, but standard accounts are free.

Once you have a GitHub account, the easiest way to share a notebook on GitHub doesn't actually require Git at all. Since 2008, GitHub has provided its Gist service for hosting and sharing code snippets, which each get their own repository. To share a notebook using Gists:

1. Sign in and navigate to gist.github.com.
2. Open your `.ipynb` file in a text editor, select all and copy the JSON inside.
3. Paste the notebook JSON into the gist.
4. Give your Gist a filename, remembering to add `.iypnb` or this will not work.
5. Click either "Create secret gist" or "Create public gist."

This should look something like the following:

If you created a public Gist, you will now be able to share its URL with anyone, and others will be able to fork and clone your work.

Creating your own Git repository and sharing this on GitHub is beyond the scope of this tutorial, but GitHub provides plenty of guides for you to get started on your own.

An extra tip for those using git is to add an exception to your `.gitignore` for those hidden `.ipynb_checkpoints` directories Jupyter creates, so as not to commit checkpoint files unnecessarily to your repo.

Nbviewer

Having grown to render hundreds of thousands of notebooks every week by 2015, NBViewer is the most popular notebook renderer on the web. If you already have somewhere to host your Jupyter Notebooks online, be it GitHub or elsewhere, NBViewer will render your notebook and provide a shareable URL along with it. Provided as a free service as part of Project Jupyter, it is available at nbviewer.jupyter.org.

Initially developed before GitHub's Jupyter Notebook integration, NBViewer allows anyone to enter a URL, Gist ID, or GitHub username/repo/file and it will render the notebook as a webpage. A Gist's ID is the unique number at the end of its URL; for example, the string of characters after the last backslash in `https://gist.github.com/username/50896401c23e0bf417e89cd57e89e1de`. If you enter a GitHub username or username/repo, you will see a minimal file browser that lets you explore a user's repos and their contents.

The URL NBViewer displays when displaying a notebook is a constant based on the URL of the notebook it is rendering, so you can share this with anyone and it will work as long as the original files remain online — NBViewer doesn't cache files for very long.

If you don't like Nbviewer, there are other similar options — here's a thread with a few to consider from our community.

Extras: Jupyter Notebook Extensions

We've already covered everything you need to get rolling in Jupyter Notebooks.

What Are Extensions?

Extensions are precisely what they sound like — additional features that extend Jupyter Notebooks's functionality. While a base Jupyter Notebook can do an awful lot, extensions offer some additional features that may help with specific workflows, or that simply improve the user experience.

For example, one extension called "Table of Contents" generates a table of contents for your notebook, to make large notebooks easier to visualize and navigate around.

Another one, called Variable Inspector, will show you the value, type, size, and shape of every variable in your notebook for easy quick reference and debugging.

Another, called ExecuteTime, lets you know when and for how long each cell ran — this can be particularly convenient if you're trying to speed up a snippet of your code.

These are just the tip of the iceberg; there are many extensions available.

Where Can You Get Extensions?

To get the extensions, you need to install Nbextensions. You can do this using pip and the command line. If you have Anaconda, it may be better to do this through Anaconda Prompt rather than the regular command line.

Close Jupyter Notebooks, open Anaconda Prompt, and run the following command: `pip install jupyter_contrib_nbextensions && jupyter contrib nbextension install`.

Once you've done that, start up a notebook and you should seen an Nbextensions tab. Clicking this tab will show you a list of available extensions. Simply tick the boxes for the extensions you want to enable, and you're off to the races!

Installing Extensions

Once Nbextensions itself has been installed, there's no need for additional installation of each extension. However, if you've already installed Nbextensons but aren't seeing the tab, you're not alone. This thread on Github details some common issues and solutions.

Final Thoughts

Starting from scratch, we have come to grips with the natural workflow of Jupyter Notebooks, delved into IPython's more advanced features, and finally learned how to share our work with friends, colleagues, and the world. And we accomplished all this from a notebook itself!

It should be clear how notebooks promote a productive working experience by reducing context switching and emulating a natural development of thoughts during a project. The power of using Jupyter Notebooks should also be evident, and we covered plenty of leads to get you started exploring more advanced features in your own projects.

If you'd like further inspiration for your own Notebooks, Jupyter has put together a gallery of interesting Jupyter Notebooks that you may find helpful and the Nbviewer homepage links to some really fancy examples of quality notebooks.

More Great Jupyter Notebooks Resources

- Advanced Jupyter Notebooks Tutorial – Now that you've mastered the basics, become a Jupyter Notebooks pro with this advanced tutorial!
- 28 Jupyter Notebooks Tips, Tricks, and Shortcuts – Make yourself into a power user and increase your efficiency with these tips and tricks!
- Guided Project – Install and Learn Jupyter Notebooks – Give yourself a great foundation working with Jupyter Notebooks by working through this interactive guided project that'll get you set up and teach you the ropes.

About the author

Benjamin Pryke

Python and web developer with a background in computer science and machine learning. Co-founder of FinTech firm Machina Capital. Part-time gymnast and digital bohemian.