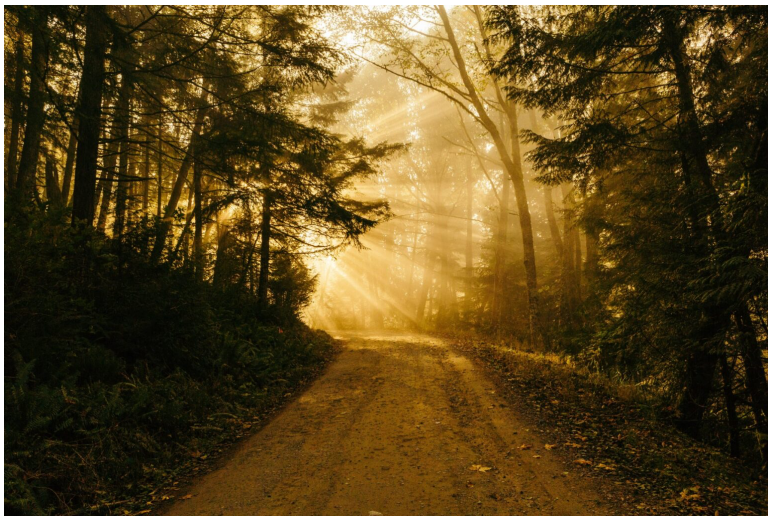
DATA SCIENCE

The Journey from Jupyter to Programmer: A Quick-Start Guide

Explore the real benefits of ditching the notebook

Lucy Dickinson

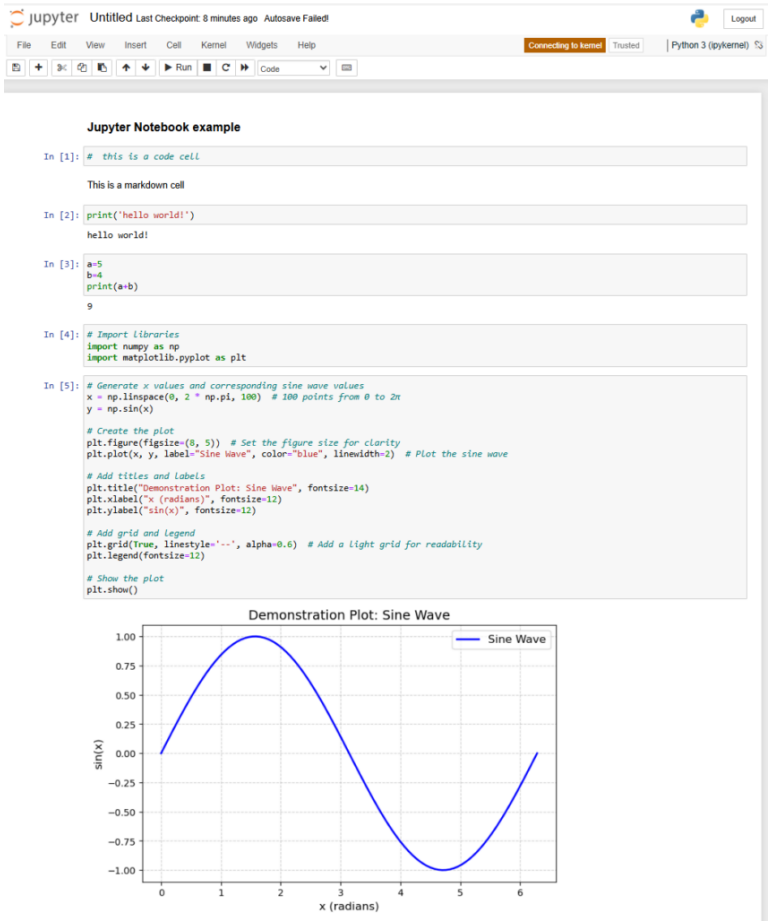
Jun 4, 2025 17 min read



Most Data Scientists, myself

included, start their coding journey using a Jupyter Notebook. These files have the extension .ipynb, which stands for **Interactive Python Notebook**. As the extension name suggests, it has an intuitive and interactive user interface. The notebook is broken down into ‘cells’ or small blocks of separated code or markdown (text) language. Outputs are displayed underneath each cell once the code within that cell has been executed. This promotes a flexible and interactive environment for coders to build their coding skills and start working on data science projects.

A typical example of a Jupyter Notebook is below:



Example of a Jupyter Notebook with code cells, markdown cells and a sample visualisation.

This all sounds great. And don't get me wrong, for use cases such as conducting solo research or exploratory data analysis (EDA), Jupyter Notebooks ***are*** great. The issues arise when you ask the following questions:

- How do you turn a Jupyter Notebook into code that can be leveraged by a business?

- Can you collaborate with other developers on the same project using a version control system?
- How can you deploy code to a production environment?

Pretty soon, the limitations of exclusively using Jupyter Notebooks within a commercial context will start to cause problems. It's simply not designed for these purposes. The general solution is to organise code in a modular fashion.

By the end of this article, you should have a clear understanding of how to structure a small data science project as a Python program and appreciate the advantages of transitioning to a programming approach. You can check out an example template to supplement this article in my github here.

Disclaimer

The contents of this article are based on my experience of migrating away from solely using Jupyter Notebooks to write code. Do notebooks still have a purpose? Yes. Are there alternative ways to organise and execute code beyond the methods I discuss in this article? Yes.

I wanted to share this information to help anyone wanting to make the move away from notebooks and towards writing scripts and programs. If I've missed any features of Jupyter Notebooks that mitigate the limitations I've mentioned, please drop a comment!

Let's get back to it.

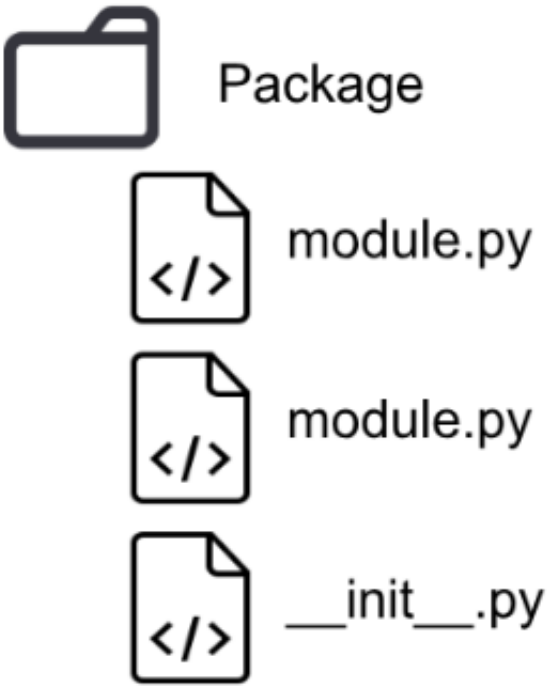
• • •

Programming: what's the big deal?

For the purpose of this article, I'll be focusing on the Python

programming language as this is the language I use for data science projects. Structuring code as a Python program unlocks a range of functionalities that are difficult to achieve when working exclusively within a Jupyter Notebook. These benefits include collaboration, versatility and portability – you're simply able to do more with your code. I'll explain these benefits further down – stay with me a little longer!

Python programs are typically organised into modules and packages. A module is a python script (files with a .py extension) that contains python code which can be imported into other files. A package is a directory that contains python modules. I'll discuss the purpose of the file `__init__.py` later in the article.



Schematic of package and module structure in a data science project

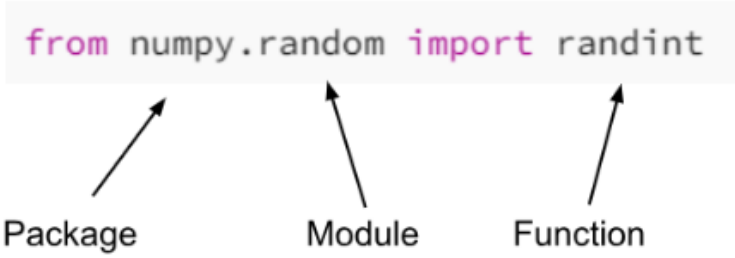
Anytime you import a python library into your code, such as built-in libraries like `os` or third-party libraries like `pandas`, you are interacting with a python program that's been organised into a package and modules.

For example, let's say you want to use the randint function from numpy. This function allows you to generate a random integer based on specified parameters. You might

write:

```
from numpy.random import randint
```

Let's annotate that import statement to show what you're actually importing.



In this instance, `numpy` is a *package*; `random` is a *module* and `randint` is a *function*.

So, it turns out you probably interact with python programs on a regular basis. This poses the question, what does the journey look like towards becoming a python programmer?

The great transition: where do you even start?

The trick to building a functional

python program is all in the file structure and organisation. It sounds boring but it plays a super important part in setting yourself up for success!

Let me use an analogy to explain: every house has a drawer that has just about everything in it; tools, elastic bands, medicine, your hopes and dreams, the lot. There's no rhyme or reason, it's a dumping ground of just about everything. Think of this as a Jupyter Notebook. This one file typically contains all stages of a project, from importing data, exploring what the data looks like, visualising trends, extracting features, training a model etc. For a project that's destined to be deployed on a production system or co-developed with colleagues, it's going to cause chaos. What's needed is some organisation, to put all the tools in one compartment, the medicine in another and so on.

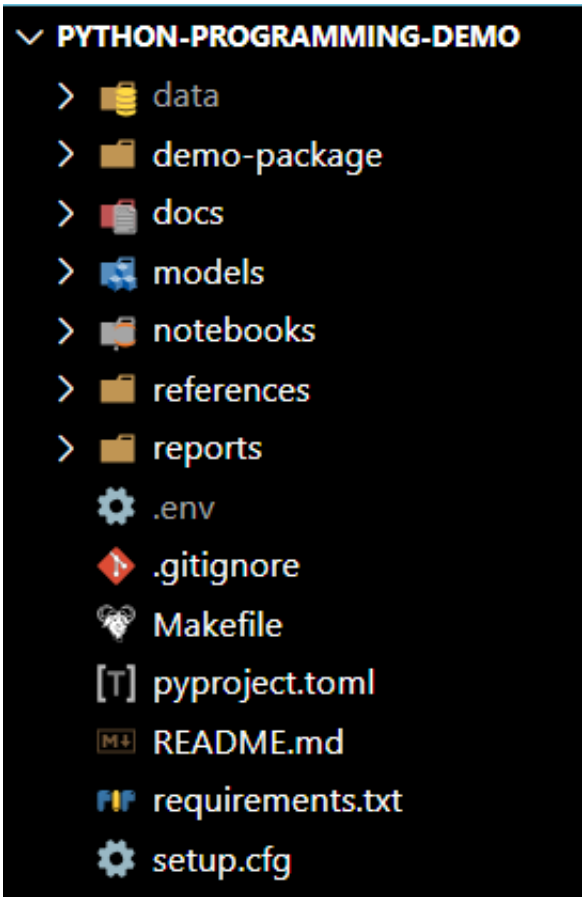
A great way to do that with code is

to use a project template. One that I use frequently is the Cookie Cutter Data Science template. You can create a whole directory for your project with all the relevant files needed to do just about anything in a few simple operations in a terminal window – see the link above for information on how to install and run Cookie Cutter.

Below are some of the key features of the project template:

- **package** or **src** directory — directory for python scripts/modules, equipped with examples to get you started
- **readme.md** — file to describe usage, setup and how to run the package
- **docs** directory — containing files that enable seamless autodocumentation
- **Makefile** — for writing OS ambivalent bespoke run commands

- **pyproject.toml/requirements.txt** — for dependency management



Project template created by the Cookie Cutter Data Science package.

Top tip. Make sure to keep Cookie Cutter up to date. With every release, new features are added according to the ever-evolving data science universe. I've learnt quite a few things from exploring a new file or feature in the template!

Alternatively, you can use other templates to build your project such as that provided by Poetry. Poetry is a package manager which you can use to generate a project template that's more lightweight than Cookie Cutter.

The best way to interact with your project is through an **IDE (Integrated Development Environment)**. This software, such as Visual Studio Code (VS Code) or PyCharm, encompass a variety of features and processes that enable you to code, test, debug and package your work efficiently. My personal preference is VS Code!

From cells to scripts: let's get coding

Now that we have a development environment and a nicely structured project template, how exactly do you write code in a python script if

you've only ever coded in a Jupyter Notebook? To answer that question, let's first consider a few industry-standard coding Best Practices.

- **Modular** — follow the software engineering philosophy of 'Single Responsibility Principle'. All code should be encapsulated in functions, with each function performing a single task. The Zen of Python states: 'Simple is better than complex'.
- **Readable** — if code is readable, then there's a good chance it will be maintainable. Ensure the code is full of docstrings and comments!
- **Stylish** — format code in a consistent and clear way. The PEP 8 guidelines are designed for this purpose to advise how code should be presented. You can install autoformatters such as Black in an IDE so that code is automatically formatted in

compliance with PEP 8 each time the python script is saved. For example, the right level of indentation and spacing will be applied so you don't even have to think about it!

- **Versatile** — if code is encapsulated into functions or classes, these can be reused throughout a project.

For a deeper dive into coding best practice, this article is a fantastic overview of principles to adhere to as a Data Scientist, be sure to check it out!

With those best practices in mind, let's go back to the question: how do you write code in a python script?

Module structure

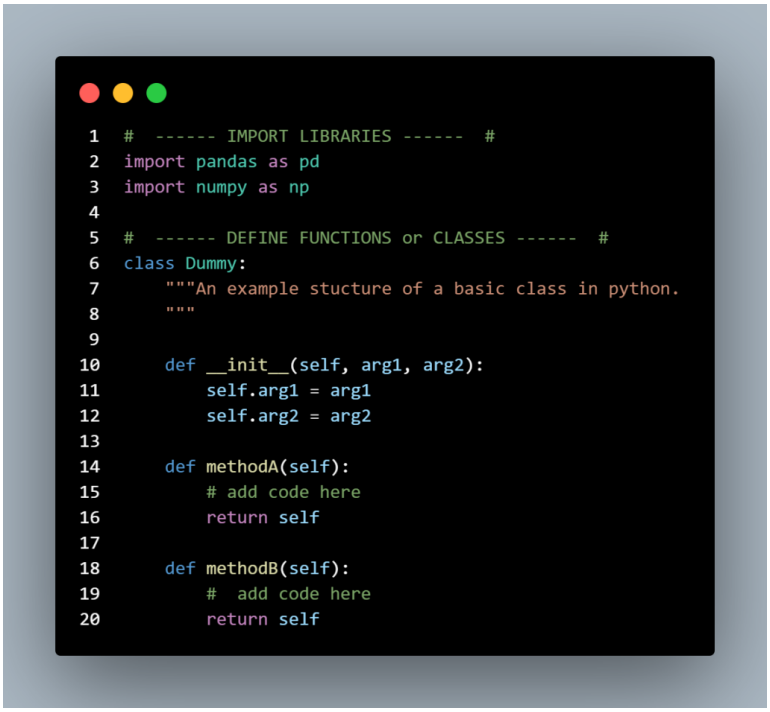
First, separate the different stages of your notebook or project into

different python files. And make sure to name them according to the task. For example, you might have the following scripts in a typical machine learning package: `data.py`, `preprocess.py`, `features.py`, `train.py`, `predict.py`, `evaluate.py` etc. Depending on your project structure, these would sit within the `package` or `src` directory.

Within each script, code should be organised or 'encapsulated' into a classes and/or functions. A **function** is a reusable block of code that performs a single, well-defined task. A **class** is a blueprint for creating an object, with its own set of attributes (variables) and methods (functions). Encapsulating code in this manner permits reusability and avoids duplication, thus keeping code concise.

A script might only need one function if the task is simple. For example, a data loading module (e.g. `data.py`) may only contain a

single function ‘load_data’ which loads data from a csv file into a `pandas` DataFrame. Other scripts, such as a data processing module (e.g. `preprocess.py`) will inherently involve more tasks and hence requires more functions or a class to encapsulate these tasks.



```python
#  ------ IMPORT LIBRARIES ------  #
import pandas as pd
import numpy as np

#  ------ DEFINE FUNCTIONS or CLASSES ------  #
class Dummy:
    """An example stucture of a basic class in python.
    """

    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def methodA(self):
        # add code here
        return self

    def methodB(self):
        #  add code here
        return self
```

Example template of a typical module in a data science project.

Top tip. Transitioning from Jupyter Notebooks to scripts may take some time and everyone’s personal journey will look different. Some Data Scientists I

know write code as python scripts straight away and don't touch a notebook. Personally, I use a notebook for EDA, I then encapsulate the code into functions or classes before porting to a script. Do whatever feels right for you.

There are a few tools that can help with the transition. 1) In VS Code, you can select one or more lines, right click and select Run Python > Run Selection/Line in Python Terminal. This is similar to running a cell in Jupyter Notebook. 2) You can convert a notebook to a python script by clicking File > Download as > Python (.py). I wouldn't recommend that approach with large notebooks for fear of creating monster scripts, but the option is there!

The '__main__' event

At this point, we've established that

code should be encapsulated into functions and stored within clearly named scripts. The next logical question is, how can you tie all these scripts together so code gets executed in the right order?

The answer is to import these scripts into a single-entry point and execute the code in one place. Within the context of developing a simple project, this entry point is typically a script named `main.py` (but can be called anything). At the top of `main.py`, just as you would import necessary built-in packages or third-party packages from PyPI, you'll import your own modules or specific classes/functions from modules. Any classes or functions defined in these modules will be available to use by the script they've been imported into.

To do this, the package directory within your project needs to contain a `__init__.py` file, which is typically left blank for simple projects. This

file tells the python interpreter to treat the directory as a package, meaning that any files with a .py extension get treated as modules and can therefore be imported into other files.

The structure of `main.py` is project dependent, but it will generally be dictated by the necessary order of code execution. For a typical machine learning project, you would first need to use the load_data function from the module `data.py`. You then might instantiate the preprocessor class that is imported from the module `preprocess.py` and apply a variety of class methods to the preprocessor object. You would then move onto feature engineering and so on until you have the whole workflow written out. This workflow would typically be contained or referenced within a conditional statement at the bottom of `main.py`.

Wait..... who mentioned anything about a conditional statement? The

conditional statement is as follows:

```
if __name__ == '__main__':
    #  add code here
```

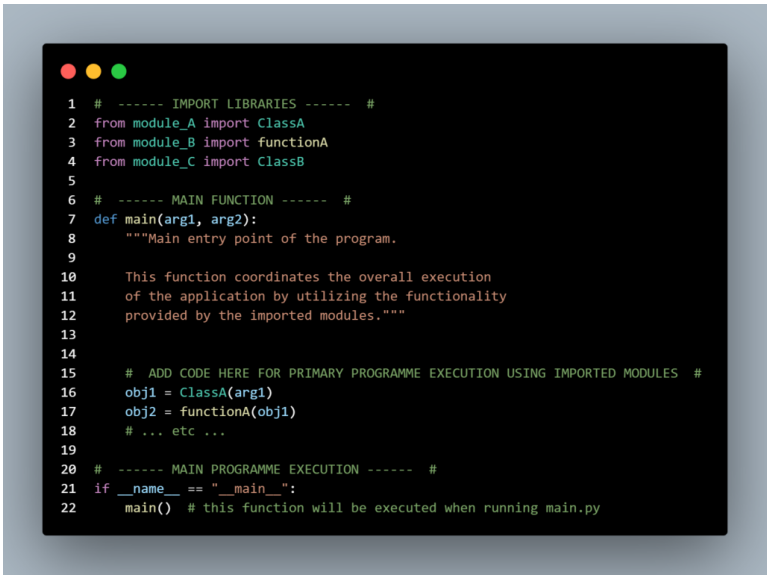
`__name__` is a special python variable that can have two different values depending on how the script is run:

- If the script is run directly in terminal, the interpreter assigns the `__name__` variable the value `'__main__'`. Because the statement `if '__name__=='__main__':` is true, any code that sits within this statement is executed.
- If the script is run as an imported module, the interpreter assigns the name of the module as a string to the `__name__` variable. Because the statement if `if '__name__=='__main__':` is false, the contents of this statement is not executed.

Some more information on this can

be found here.

Given this process, you'll need to reference the master function within the `if '__name__=='__main__':` conditional statement so that it is executed when `main.py` is run. Alternatively, you can place the code underneath `if '__name__=='__main__':` to achieve the same outcome.



```python
#  ------ IMPORT LIBRARIES ------  #
from module_A import ClassA
from module_B import functionA
from module_C import ClassB

#  ------ MAIN FUNCTION ------  #
def main(arg1, arg2):
    """Main entry point of the program.

    This function coordinates the overall execution
    of the application by utilizing the functionality
    provided by the imported modules."""


    #  ADD CODE HERE FOR PRIMARY PROGRAMME EXECUTION USING IMPORTED MODULES  #
    obj1 = ClassA(arg1)
    obj2 = functionA(obj1)
    # ... etc ...

#  ------ MAIN PROGRAMME EXECUTION ------  #
if __name__ == "__main__":
    main()  # this function will be executed when running main.py
```

Example template of main.py, which serves as the main entry point to the program

`main.py` (or any python script) can be executed in terminal using the following syntax:

```
python3 main.py
```

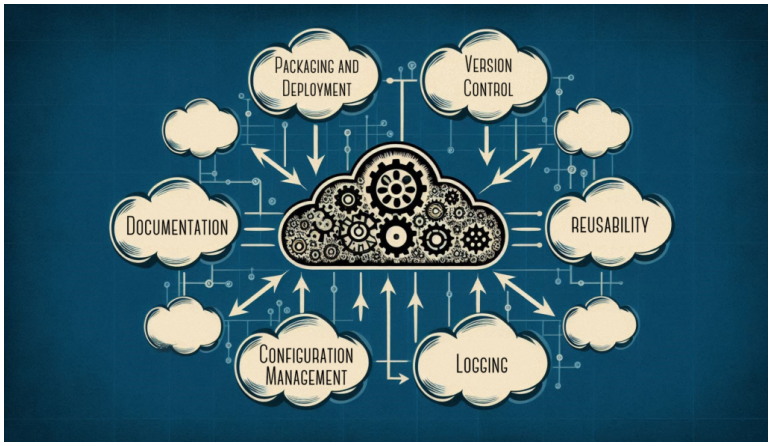
Upon running `main.py`, code will be executed from all the imported modules in the specified order. This is the same as clicking the 'run all' button on a Jupyter Notebook where each cell is executed in sequential order. The difference now is that the code is organised into individual scripts in a logical manner and encapsulated within classes and functions.

You can also add **CLI (command-line interface) arguments** to your code using tools such as argparse and typer, allowing you to toggle specific variables when running `main.py` in the terminal. This provides a great deal of flexibility during code execution.

So we've now reached the best part. The pièce de résistance. The real reasons why, beyond having fantastically organised and readable code, you should go to the effort of Programming.

The end game: what's the point of programming?

Let's walk through some of the key benefits of moving beyond Jupyter Notebooks and transitioning to writing Python scripts instead.



Visualisation of the key benefits to programming. Image generated by author.

- **Packaging & distribution** — you can package and distribute your python program so it can be shared, installed and run on another computer. Package managers such as pip, poetry or conda can be used to install the package, just as you would

install packages from PyPI, such as `pandas` or `numpy`. The trick to successfully distributing your package is to ensure that the dependencies are managed correctly, which is where the files `pyproject.toml` or `requirements.txt` come in. Some useful resources can be found here and here.

- **Deployment** — whilst there are multiple methods and platforms to deploy code, using a modular approach will put you in good stead to get your code production ready. Tools such as Docker enable the deployment of programs or applications in isolated environments called containers, which can be easily managed through CI/CD (continuous integration & deployment) pipelines. It's worth noting that while Jupyter Notebooks can be deployed using JupyterLab, this approach lacks the flexibility

and scalability of adopting a modular, script-based workflow.

- **Version control** — moving away from Jupyter Notebooks opens up the wonderful worlds of version control and collaboration. Version control systems such as Git are very much industry standard and offer a wealth of benefits, providing you use them appropriately! Follow the motto 'incremental changes are key' and ensure that you make small, regular commits with logical commit messages in imperative language whenever you make functional changes whilst developing. This will make it far easier to keep track of changes and test code. Here is a super useful guide to using git as a data scientist.

Fun fact. It's generally discouraged to commit Jupyter Notebooks to version control

systems as it is difficult to track changes!

- **(Auto)Documentation** — we all know that documenting code increases its readability thus helping the reader understand what the code is doing. It's considered best practice to add docstrings to functions and classes within python scripts. What's really cool is that we can use these docstrings to build an index of formatted documentation of your whole project in the form of html files. Tools such as Sphinx enable you to do this in a quick and easy way. You can read my previous article which takes you through this process step by step.
- **Reusability** — adopting a modular approach promotes the reuse of code. There are many common tasks within data science projects, such as

cleansing data or scaling features. There's little point in reinventing the wheel, so if you can reuse functions or classes with minor modification from previous projects, as long as there are no confidentiality restrictions, then save yourself that time! You might have a `utils.py` or `classes.py` module which contains ambivalent code that can be used across modules.

- **Configuration management** — whilst this is possible with a Jupyter Notebook, it is common practice to use configuration management for a python program. Configuration management refers to organising and managing a project's parameters and variables in a centralised way. Instead of defining variables throughout the code, they are stored in a file that sits within the project directory. This

means that you do not need to interrogate the code to change a parameter. An overview of this can be found here.

Note. If you use a YAML file (.yml) for configuration, this requires the python package `yaml`. Make sure to install the pyyaml package (not 'yaml') using `pip install pyyaml`. Forgetting this can lead to "package not found" errors—I've made this mistake, maybe more than once..

- **Logging**—using loggers within a python program enables you to easily track code execution, provide debugging information and monitor a program or application. Whilst this functionality is possible within a Jupyter Notebook, it is generally considered overkill and is fulfilled with the print() statement instead. By using python's logger module, you can format a logging object to your

liking. It has five different messaging levels (info, debug, warning, error, critical) relative to the severity of the events being logger. You can include logging messages throughout the code to provide insight into code execution, which can be printed to terminal and/or written to a file. You can learn more about logging here.

When are Jupyter Notebooks useful?

As I eluded at the beginning of this article, Jupyter Notebooks still have their place in data science projects. Their easy-to-use interface makes them great for exploratory and interactive tasks. Two key use cases are listed below:

- Conducting exploratory data analysis on a dataset during the initial stages of a project.
- Creating an interactive resource or report to demonstrate

analytical findings. Note there are plenty of tools out there that you can use in this nature, but a Jupyter Notebook can also do the trick.

Final thoughts

Thanks for sticking with me to the very end! I hope this discussion has been insightful and has shed some light on how and why to start programming. As with most things in Data Science, there isn't a single 'correct' way to solve a problem, but a considered multi-faceted approach depending on the task at hand.

Shout out to my colleague and fellow data scientist Hannah Alexander for reviewing this article 🙂

Thanks for reading!

WRITTEN BY

Lucy Dickinson

See all from Lucy Dickinson

Topics:

Best Practices

Editors Pick

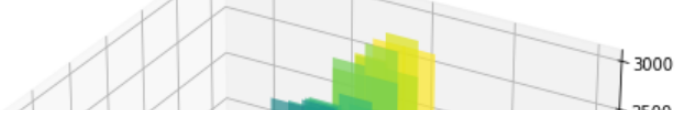
Jupyter Notebook

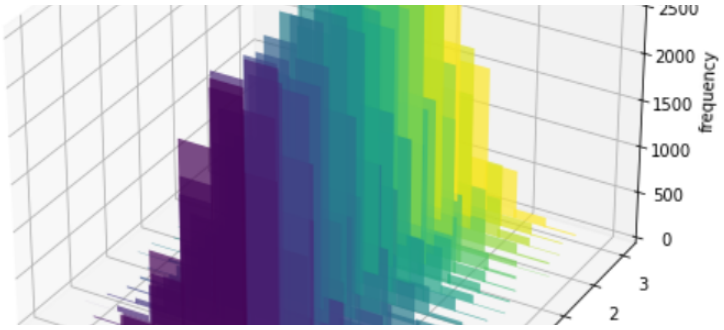
Programming

Python

Share this article:

Related Articles





DATA SCIENCE

Must-Know in Statistics: The Bivariate Normal Projection Explained

Derivation and practical examples of this powerful concept

Luigi Battistoni

August 14, 2024 7 min read

DATA SCIENCE

Optimizing Marketing Campaigns with Budgeted Multi-Armed Bandits

With demos, our new solution, and a video

Vadim Arzamasov

August 16, 2024 10 min read

ANALYTICS

Methods for Modelling Customer Lifetime Value: The Good Stuff and the Gotchas

Part three of a comprehensive, practical guide to CLV techniques and real-world use-cases

Katherine Munro

November 17, 2023 12 min read

DATA SCIENCE

Squashing the Average: A Dive into Penalized Quantile Regression for Python

How to build penalized quantile regression models (with code!)

Álvaro Méndez Civieta

August 16, 2024 5 min read

DATA SCIENCE

The Math Behind Keras 3 Optimizers: Deep Understanding and Application

This is a bit different from what the books say.

Peng Qian

August 17, 2024 9 min read

DATA ENGINEERING

Feature Engineering with Microsoft Fabric and Dataflow Gen2

Fabric Madness part 3

Roger Noble

April 15, 2024 13 min read

DATA SCIENCE

Done is Better Than Perfect

How to be more pragmatic as a Data Scientist, and why it matters for your...

Torsten Walbaum

July 30, 2024 11 min read

Your home for data science and Al. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Subscribe to Our Newsletter