

CHAPTER 3

Tools for Data Science Developers

“Data! Data Data! I can’t make bricks without clay!”

—Sir Arthur Conan Doyle

How do we manage data and models? What are the tools we can use to make ourselves more efficient and agile in data science? In this chapter, we will deep dive into the tools and technology that you will depend on daily as an MLOps engineer.

AI tools can make you more productive. With the release of GPT3 in June 2020, the large language model and “brains” behind the ChatGPT app, and in March of 2023, GPT4, the first multimodal large language model capable of understanding both text and images was released. Data scientists will increasingly use AI tools to write code.

The growth is exponential, and although it cannot predict very far into the future what specific tools will be available, it is certain that basic tools like code version control systems, data version control, code editors, and notebooks will continue to be used in some form or another in data science, and what’s important is to have a solid foundation in the basics.

You will understand version control, data version control, and specific python packages used at various stages of the spiral MLOps lifecycle. You should be comfortable enough at the end of this chapter to complete the

titular MLOps toolkit lab work where you'll build a cookie cutter MLOps template you can apply to accelerate your projects and be able to install a wide range of MLOps packages like MLFlow and Pandas to support various stages of the MLOps lifecycle.

Data and Code Version Control Systems

Data science is a collaborative activity. When you are first learning data science you might spend most of your time alone, exploring data sets you choose and applying whatever models perform best on your data set.

In the real world you typically work on a team of data scientists, and even if you are the sole individual contributor on your team, you still likely report results to stakeholders, product managers, business analysts, and others and are responsible for handling changes.

All of these changes in a business impact data science as they result in changes in downstream feature engineering libraries and training scripts. You then need a way to share code snippets and get feedback in order to iterate on results and keep track of different versions of training scripts, code, and notebooks. Are there any tools to manage this change in data, code, and models? The answer is version control.

What Is Version Control?

Version control is a software tool used to manage changes in source code. The tool keeps track of changes you make to the source code and previous versions and allows you to roll back to a previous version, prevent lost work, and pinpoint where exactly in the code base a particular line was changed. If you use it properly, you read the change log and understand the entire history of your project. Git, a distributed version control system (as opposed to centralized version control), is a standard for data science teams.

What Is Git?

As we mentioned, Git is a standard tool for version control. Git works basically by taking a snapshot of each file in your directory and storing this information in an index. Git is also a distributed version control system (as opposed to a central version control like TFS) which means it supports collaboration among data scientists and developers. Each developer can store the entire history of the project locally, and because Git only uses deltas, when you are ready to commit changes, you can push them to the remote Git server, effectively publishing your changes.

Git Internals

Git uses commands. There are several Git commands you should be aware of and some special terminology like “repos” which refers to a collection of files that are source controlled. If you are unfamiliar with the concept of repos, you could think of it like a kind of directory where your source code lives.

In practice, when working on a data science project as in MLOps role, you will probably use a source control tool like Sourcetree since productivity is important, and also once you know the basics of the commands, it gets very repetitive to type each time. Tools like Sourcetree abstract these details away from you. You may be wondering why a tool like Sourcetree could help data scientists when you can use the Git command. As we will see in the next section, Git does provide low level commands for interacting with Git repositories, but Sourcetree is a GUI tool, and since typing the same command over and over again takes time, using a GUI tool will make you a more productive developer.

Plumbing and Porcelain: Understanding Git Terminology

Porcelain commands refer to high level Git commands that you will use often as part of your workflow. Plumbing is what Git does behind the scenes. An example of a porcelain command is the Git status to check for changes in your working directory.

Ref: A ref is essentially a pointer to a commit. A pointer is how Git represents branches internally.

Branch: A branch is similar to a ref in that it's a pointer to a commit. You can create a new branch using the command given in Listing 3-1.

Listing 3-1. Git command to create a new branch

```
git branch <branch>
```

Head: Some documentation makes this seem really complicated but it is not. In Git, there was a design choice that only one branch can be checked out at a time, and this had to be stored in some reference or pointer. (If you don't know what a reference or a pointer is, read the Git documentation¹).

How Git Stores Snapshots Internally

Git assigns a special name to this pointer called HEAD. There can only be one HEAD at a time and it points to the current branch. This is the reason why you might hear HEAD referred to as the “active” branch.

You might be wondering, how is this “pointer” physically stored on a computer. Well, it turns out the pointer is not a memory address but a file. This file stores the information that the HEAD is the current branch

¹The Git documentation covers topics including references and pointers: <https://git-scm.com/book/en/v2/Git-Internals-Git-References>

(remember the definition of a branch from earlier). There is a physical location on the computer in the `.git/HEAD` directory where this file is located and you can open it up in a text editor (such as Notepad++) and read its contents for yourself to understand how Git stores information internally.

Don't worry if this seems complicated, as it will be much easier in the lab work and begin to make sense when you use it and see the purpose of Git for yourself.

Sourcetree for the Data Scientist

We recommend using Sourcetree, a free open source GUI based tool. If you are a professional software developer, you can try Kraken which has some additional features but requires a license. There are two steps for using Sourcetree:

You can download Sourcetree at sourcetreeapp.com. You need to agree to terms and conditions and then download the app (Figure 3-1):

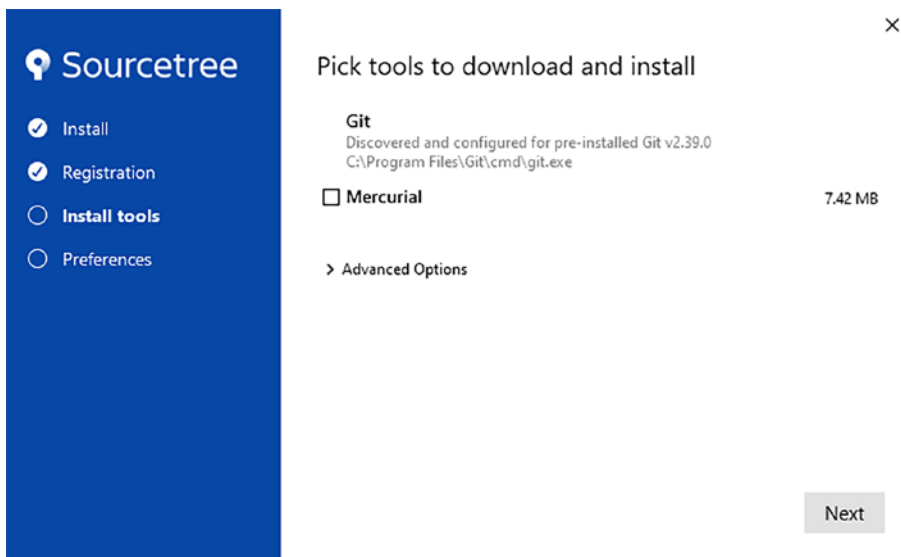


Figure 3-1. Sourcetree GUI tool for interacting with Git repositories

Step 1: Clone a remote repository. Figure 3-2 shows the GUI interface for cloning a repository in Sourcetree.

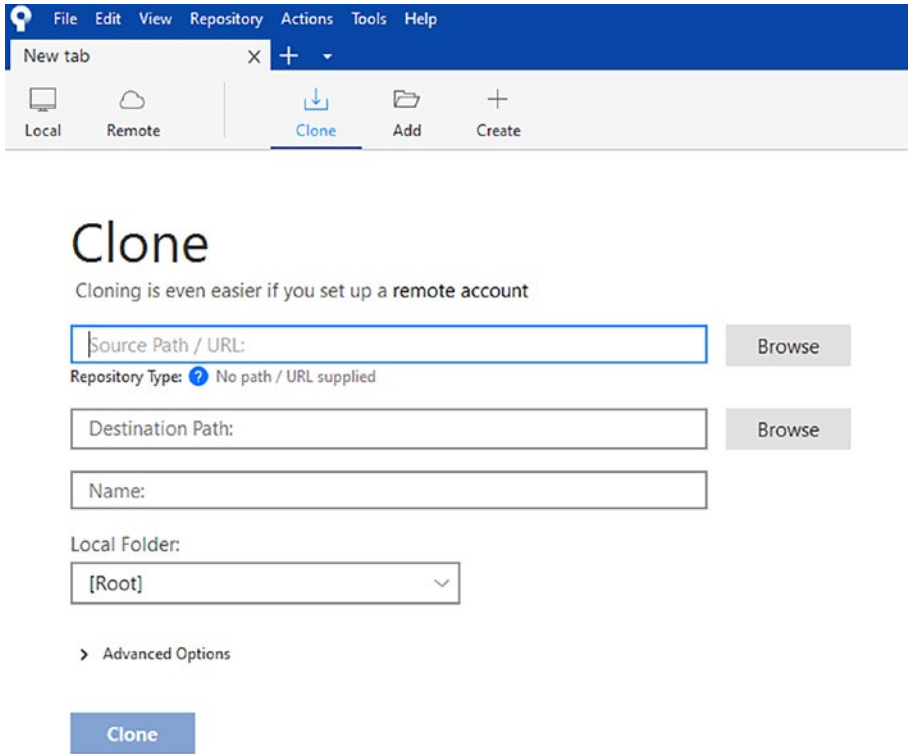


Figure 3-2. Cloning a Git repository using a GUI

Step 2: If you use a private repo, you'll need to configure your ssh key. Make sure to click SSH not HTTPS as shown in Figure 3-3.

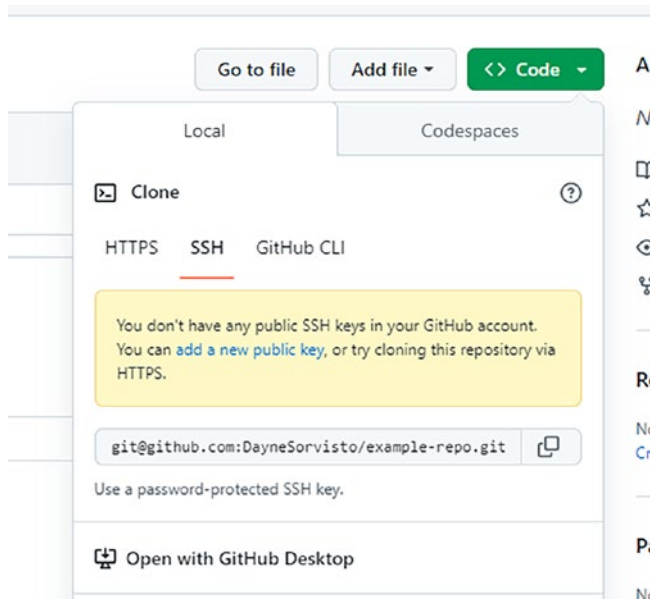


Figure 3-3. Copying the SSH path to your GitHub repo

Branching Strategy for Data Science Teams

If you are on a team of at least five developers, you may have to consider a branching strategy. This matters less if you are on a small team or alone because as a data scientist you may be OK to rely on a single main branch, but with more than five developers, you may consider setting up a second branch.

If you want to learn about more complex branching strategies beyond feature branches, you can read about Git Flow. Usually different branching strategies are chosen in consideration of a software release schedule in collaboration with other teams depending on the size of your organization among other factors. Figure 3-4 shows how to create a new branch from the Sourcetree GUI.

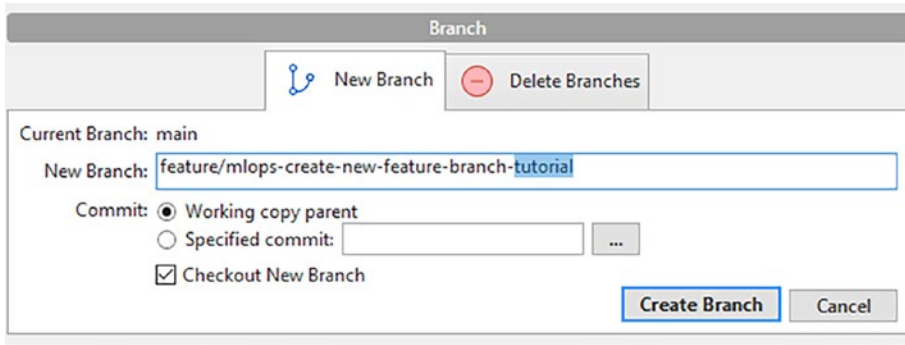


Figure 3-4. Creating a new branch from the Sourcetree GUI

Creating Pull Requests

Pull requests are a great tool for code reviews and should be adopted by data science teams. Typically the main branch is a stable branch, and prior to merging changes into main, you should have a peer review your changes. Ideally, a data scientist on your team that is familiar with Git would be designed as the release manager and would coordinate this with the team, but the process can be done informally. Figure 3-5 shows how to create a pull request.

Benefits of pull requests for data scientists include the following:

- Opportunity to review changes and learn new data science techniques.
- Catch mistakes and bugs before they are committed to main branch, increasing code quality metrics.

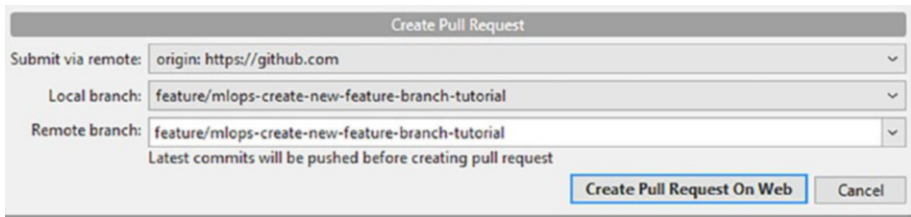


Figure 3-5. Creating a pull request

Do I Need to Use Source Control?

You might wonder if all of this is necessary or do you even need source control. But what are the consequences of not using it? You should use source control for the following reasons

- You are part of a team of data scientists sharing code and collaborating and need to coordinate changes through a remote branch.
- You want to version your data so you can train models on a previous version of data.
- You are a data scientist that does not want to lose their work on their local system and wants something more reliable than a Jupyter Notebook’s autosave functionality.
- You need a way to save different snapshots of your data or code.
- You want a log or paper trail of your work in case something breaks (you can use the “Blame” feature to pinpoint the author of a change).

Version Control for Data

We've talked about code version control, but as we've mentioned, MLOps involves code, data, and models. While we use tools like Git for code version control, data version control exists and can be applied to both data and models (which are usually serialized in a binary format).

The standard package is called DVC (you can guess this stands for data version control). DVC works on top of Git, and many of the commands and terminology are similar. For example, the `dvc init` command is used to initialize data version control in your repo. In the lab, you'll work through some basic `dvc` commands for data and model version control.

Git and DVC Lab

In this lab (Figure 3-6), you will gain some hands-on experience with both Git for interacting with Git repositories and DVC for versioning data. Fortunately, many of the Git commands you will learn in the lab are very similar to the DVC commands you will learn later. However, throughout the lab, you should keep in mind the distinct purpose of each tool and where you might want to use each in your own workflow.

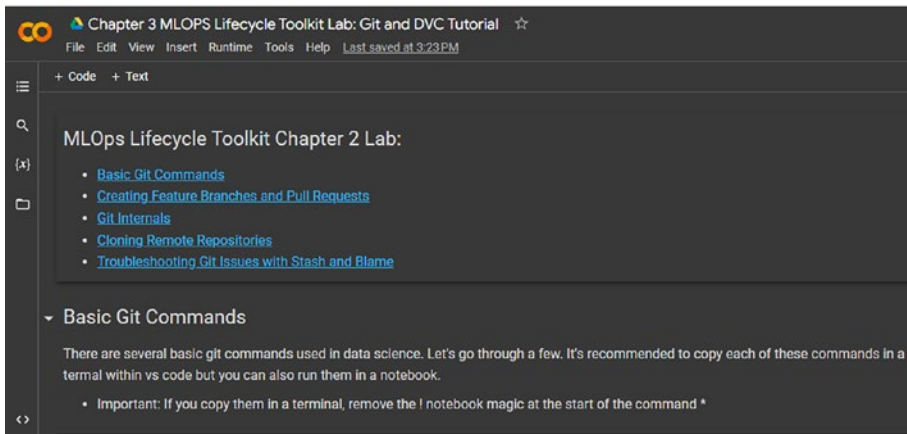


Figure 3-6. GIT and data version control (DVC) lab

Before proceeding to the next section on code editors, complete the version control lab titled: `Chapter_3_MLOPS_Lifecycle_Toolkit_Lab_Git_and_Dvc`

Step 1. Open `Chapter_3_MLOPS_Lifecycle_Toolkit_Lab_Git_and_Dvc.ipynb` and read the instructions.

Step 2. Copy paste the commands in the notebook into a terminal, and get familiar with each command and what it does; you can use the `-h` flag to see what each command does (e.g., `git status -h`).

Step 3. Sign up for a GitHub account by following instructions in the lab.

Model Development and Training

So we've covered version control systems for both code and data but how about the tools we use to edit our code and develop models? You may be using a tool like Spyder or a Jupyter notebook to edit your code, and surely like most developers, this is your favorite editor. I don't want to change your mind, but it's worth knowing the range of code editors available in data science and when and why you might want to consider using an editor like VS Code over Spyder.

Spyder

Spyder is a free and open scientific environment for data science. It was first released in 2009 and is available cross-platform (Windows, Linux, and MacOS) through Anaconda. It provides the following features and several more:

- An editor includes both syntax highlighting and code completion features as well as introspection.
- View and modify environment variables from UI.
- A Help pane able to render rich text documentation for classes and functions.

Visual Studio Code

You can launch vs. code using the code command as shown in Figure 3-7.

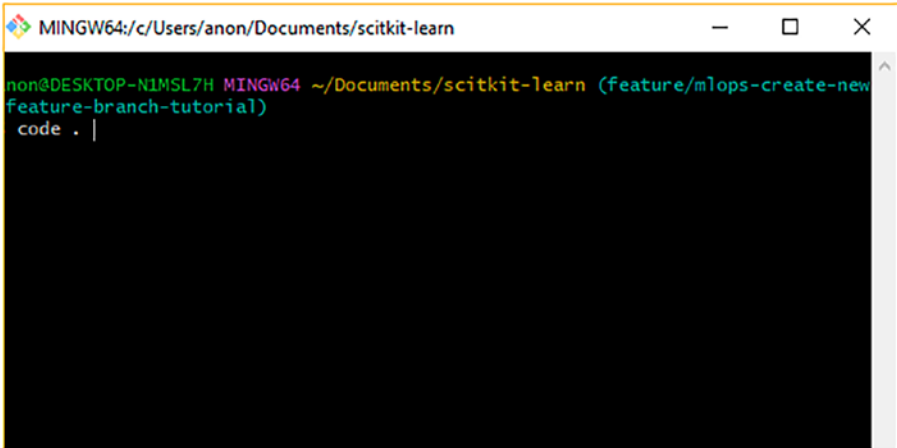


Figure 3-7. Shortcut for launching Visual Studio Code editor from a terminal

I'd suggest customizing the layout but at least including the Activity Bar as shown in Figure 3-8.

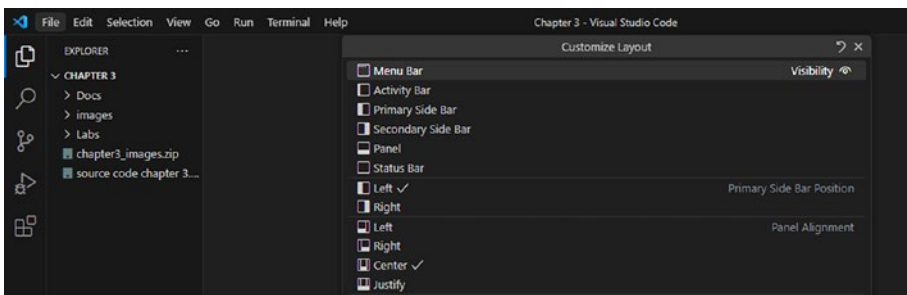


Figure 3-8. The Activity Bar in Visual Studio Code editor

Visual Studio Code is a source control editor from Microsoft based on the electron framework and is available for MacOS, Windows, and Linux distributions. The tool includes debugging, source control management, syntax highlighting, and intelligent code completion and operates by using extensions to add additional functionality. It is much more of a tool for large software projects and includes many extensions that allow you to interact with cloud infrastructure, databases, and services.

For example, there is an extension for Azure that allows accessing resources in the cloud. If you need to format your code, you could install one of several profile extensions or specific packages like `black`² or `autopep8`³. You search for these extensions in the activity bar and can access functionality in extensions using the keyboard shortcut **CTRL + SHIFT + P** to access the palette. We recommend at minimum you install the Microsoft Python extension or the Python Extension Package which includes linters, intellisense, and more (we'll need this when we create environments and set up tests). Figure 3-9 shows some of the Python extensions available in Visual Studio Code.

² Black is a standard code formatter used across industries. The GitHub for black is <https://github.com/psf/black>

³ autopep8 automatically formats Python code conforming to PEP8 and can be found on GitHub at <https://github.com/hhatto/autopep8>

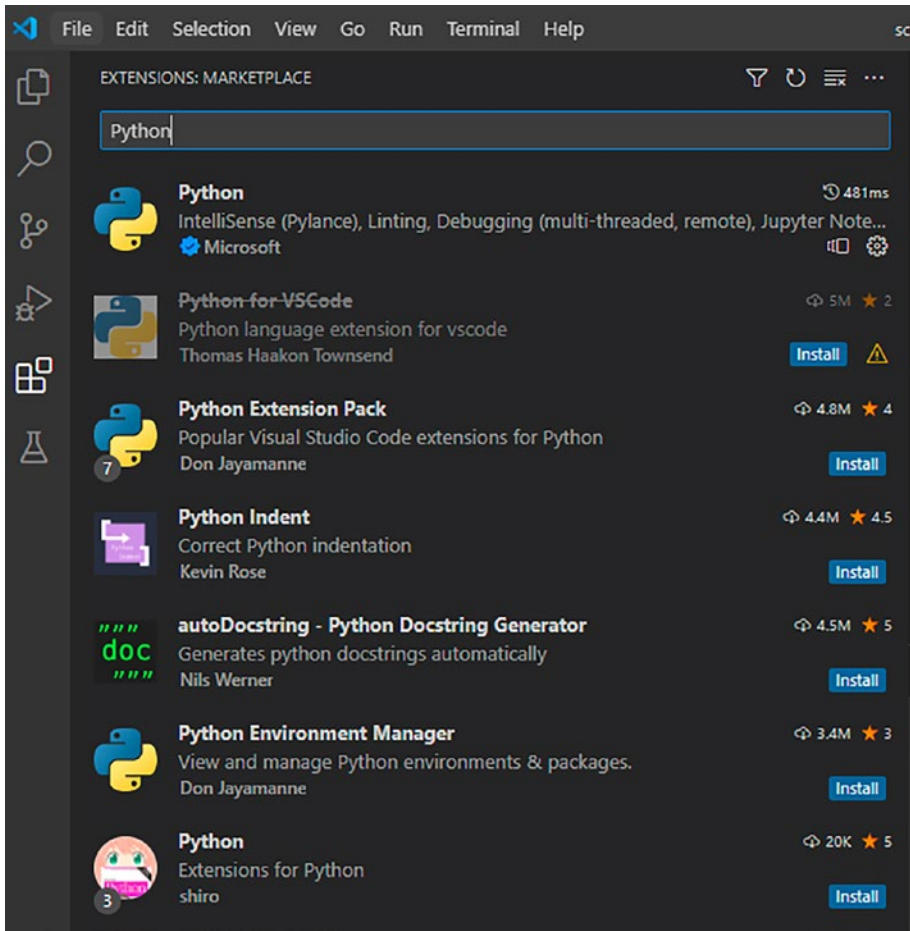


Figure 3-9. Python extensions available in Visual Studio Code editor

Cloud Notebooks and Google Colab

Cloud notebooks are a convenient way for data scientists to run code and use Python libraries in the cloud without having to install software locally. A cloud notebook such as Google Colab can be a good alternative to Visual Studio Code editor for running experiments or prototyping code. You type code into cells and can run cells in order. Figure 3-10 shows the MLOps lifecycle toolkit lab in Google Colab.

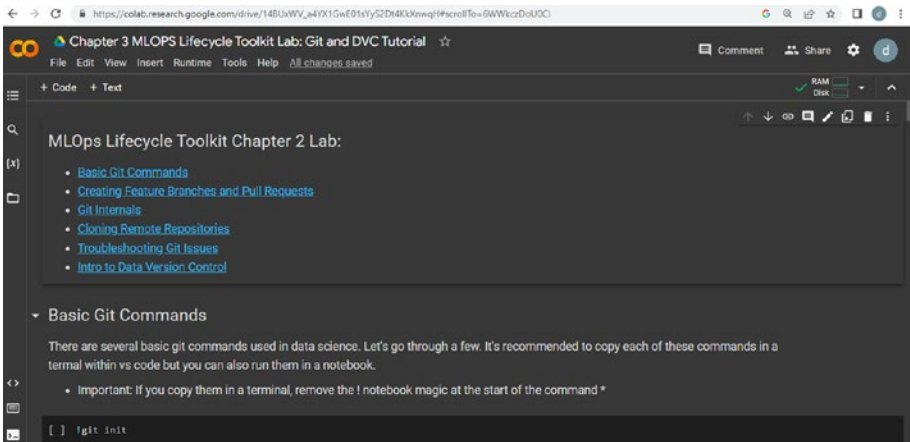


Figure 3-10. *The MLOps Lifecycle Toolkit Git and DVC Lab in Google Colab*

You can also change the theme of your notebook or connect to your GitHub through the tools ► settings menu. Figure 3-11 shows how to configure settings in Google Colab.

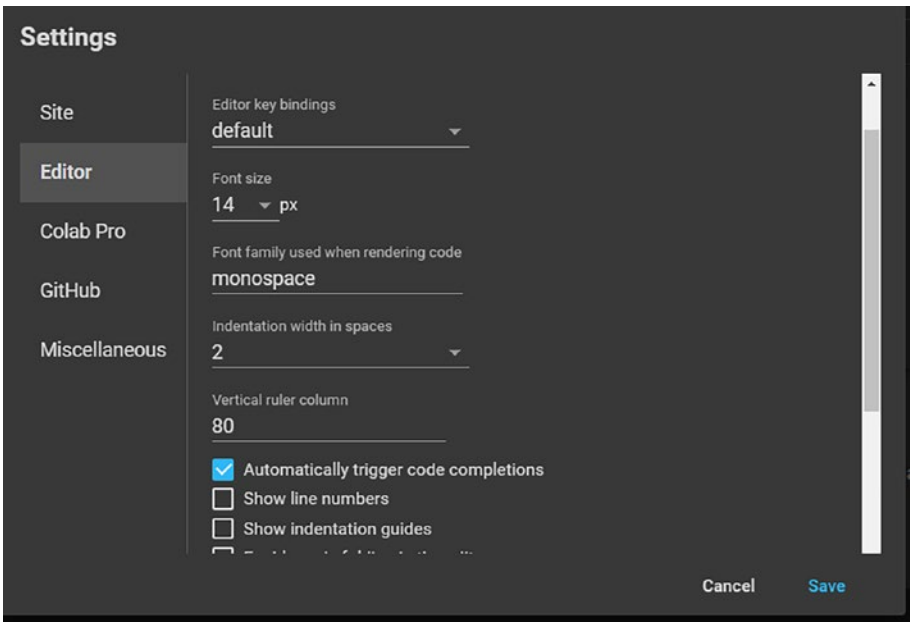


Figure 3-11. *Configuring notebook settings in Google Colab*

Programming Paradigms and Craftsmanship

What is craftsmanship in software? It refers to all of the high level skills you need for creating high quality data science code. Topics like naming conventions, documentation, writing tests, and avoiding common code smells all work toward writing higher quality code. Code that is high quality is often described as being “clean” which means it’s more readable and maintainable, and although it may still have a higher cognitive complexity overall than other software, technical debt can be reduced by taking these topics to heart. Let’s take a look at some of the elements of writing high quality data science code.

Naming Conventions and Standards in Data Science

If you don’t reduce tech debt in your project, you may find yourself working overtime when something happens in production. Part of minimizing tech debt and keeping the project readable is ensuring a consistent naming convention is used for variable names, functions, class names and files, modules, and packages.

Some guidelines for naming standards are as follows:

- Use descriptive names for variables and functions.
- Consider using verbs for function names describing what your function does.
- Refer to the style guide of the language PEP8 for Python (these include advice on indentation, white space, and coding conventions).
- Use smaller line sizes for more readable code.
- Avoid long function names and functions with too many parameters - break these out into smaller functions that do one thing.

Code Smells in Data Science Code

Code smells are anti-patterns that indicate brittle code or technical debt or places in the program that could be improved. An example in Python would be using too many nested loops or hardcoding data instead of using a variable.

You might hear the term “code smell” in programming especially if your organization requires regular code reviews. During this review process, you will look for code smells. It is good practice to remove code smells when you find them as they will incur technical debt if you leave them (they may also make it more painful for other people to maintain your code when you have to hand it off to someone else or fix it yourself in the future).

A good practice is to always assume you yourself will have to maintain the code in 6 months or even a year from now and to make sure your code can be clearly understood even after you’ve forgotten the details of how it works.

Documentation for Data Science Teams

Most data science projects, like other software projects, are lacking in documentation. Documentation for projects can come in a number of different formats and doesn’t necessarily have to mean a formal technical document; it depends on your team and the standards that have been established (if they exist). However, if coding standards don’t exist, here are some recommendations for creating awesome technical documentation:

- Use doc strings without hesitation.
- Create a central repository for documentation.
- Create an acceptance criterion in tickets if you’re using a board like JIRA or Azure DevOps.
- Socialize changes and ensure team members know how and where to add new documentation.

You've seen a few doc strings in the lab from the previous chapter already. We can use triple quotes underneath the function signature to describe briefly what the function does.

These doc strings are valuable because they can describe the following information:

- *What your function does*: If you find yourself trying hard to describe what your function does or realize it does more than one thing, you may want to break it up; therefore, going through this exercise of having doc strings for every function can improve quality of your code.
- *Description input, outputs, and data types*: Since languages like Python are dynamically typed, we can run into trouble by being too flexible with our data types. When you train a model on training data and forget it can take on a certain value in production, it could cause your program to crash. It's good practice to carefully consider the data types for any function arguments and, if not using type annotations, at least include the data type in the doc string.

Last but not least, make sure to update your documentation as requirements change. This leads us to the next tool in the toolkit for future proofing our code: TDD (test driven development).

Test Driven Development for Data Scientists

In data science projects especially, requirements can be fuzzy or ill-defined or missing all together in some cases. This can lead to surprises when there are huge gaps between what the user of the model or software expect and what you as the data scientist create and lead to longer release cycles or heavy refactoring down the line especially with feature engineering libraries.

One way to future-proof your code is to include tests with each function in your feature library. Of course this is time consuming, and on a real project, you may not have the time but it is strongly recommended. It only takes an hour or two to set up tests in Pytest or Hypothesis and create fixtures, and if you're using asserts already in your code, you can use these as the basis for your tests, and it will save you time if you need to debug your code in production. Figure 3-12 shows how to select a testing framework for TDD.

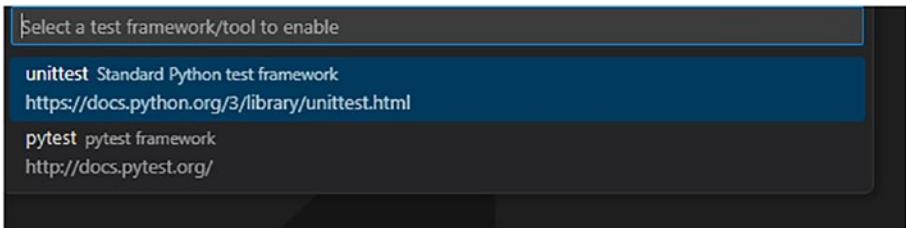


Figure 3-12. *Selecting a testing framework*

You may get import errors shown in Figure 3-13.

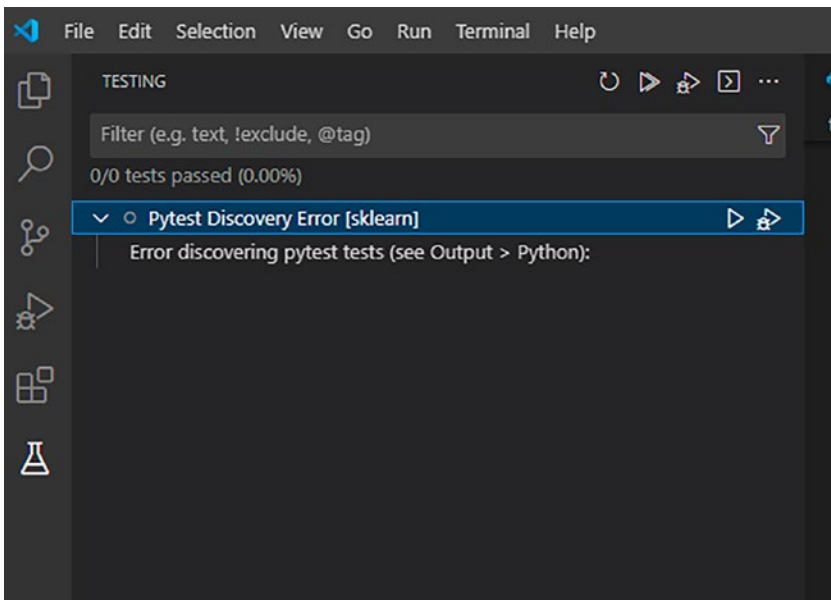


Figure 3-13. *Import errors are common when setting up Pytest in Visual Studio Code*

Once you fix the import errors, you can see tests by clicking the Testing icon in the Activity Bar and clicking run. A test that passes will have a green check mark to the left. You can run multiple tests at the same time. In the MLOps toolkit lab, you can create your own unit tests and fixtures (a way of passing data to tests) and play with this feature to incorporate testing into your own data science projects. Figure 3-14 shows how to run tests in Visual Studio Code.

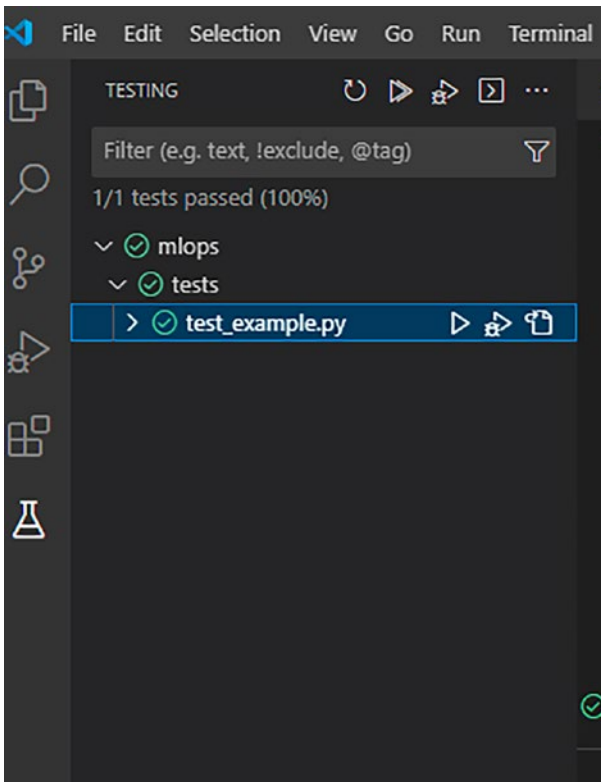


Figure 3-14. Running tests in Visual Studio Code

From Craftsmanship to Clean Code

There are many guidelines and principles for writing “clean code,” and as you become better developers, you will come to recognize code when it is clean. In data science, clean code is often an afterthought and often only comes after translating an ad hoc analysis into something worthy for production. However, here are several principles that a data scientist can use to reduce technical debt and write cleaner, more readable code:

- Be consistent! Consistency is key especially when it comes to naming variables.
- Use separate folders for feature engineering, data engineering, models, training, and other parts of the workflow.
- Use abstraction: Wrap low level code in a function.
- If your functions are too long, break them up; they probably do more than one thing violating the *SOLID principle* of single responsibility.
- Reduce the number of parameters you use in your functions if possible (unless maybe if you’re doing hyper-parameter tuning).
- Wrap lines and set a max line length in your editor.

Model Packages and Deployment

Data science software consists of a number of independent modules that work together to achieve a goal. For example, you have a training module, a feature engineering module, maybe several packages you use for missing values, or LightGBM for ranking and regression. All of these modules share something in common: You can install them, deploy them, and import them as individual deployable units called packages.

Choosing a Package Manager

Packages can consist of bundles of many modules, files, and functionality that are maintained together and are usually broader in scope than a single file, function, or module. In Python, you can use packages using a Conda or Pip or other package manager, but it's important to understand how to create your own python packages.

Setting up Packages in VS Code, use the command palette—CTRL + SHIFT + P keyboard shortcut (ensure to hold down CTRL, SHIFT, and P at the same time)—and select Python Create Environment. This is part of the Python extension package you installed earlier. Figures 3-15 through 3-18 show the detailed steps for configuring a Python environment in Visual Studio Code including selecting a package manager.

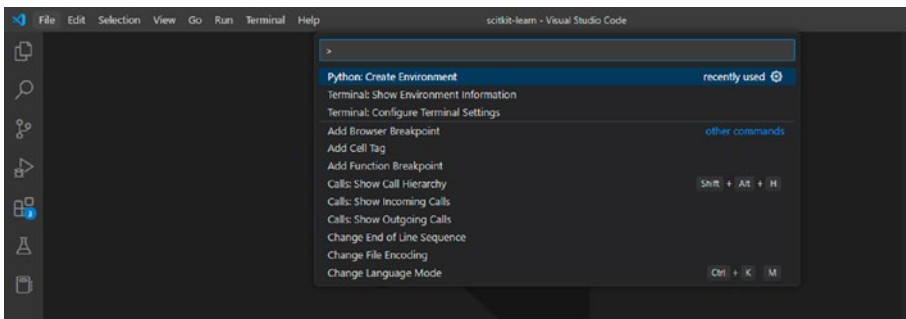


Figure 3-15. *Creating a Python environment*

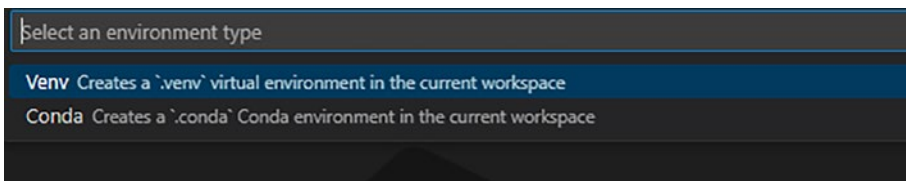


Figure 3-16. *Choosing between Conda and Virtual environment. Both are options in Visual Studio Code*

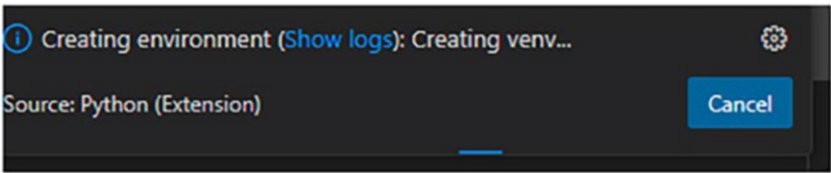


Figure 3-17. Visual Studio Code creating a new environment

```
(.venv) PS C:\Users\anon\Documents\scikit-learn\sklearn> pip install numpy
○ Collecting numpy
  Using cached numpy-1.24.2-cp311-cp311-win_amd64.whl (14.8 MB)
Installing collected packages: numpy
```

Figure 3-18. Once the environment is activated, you can install packages using your chosen package manager

Anaconda

What is Anaconda? Well, it's not a snake. Anaconda instead is bigger than any one tool and is an ecosystem unto itself. There's a virtual environment tool called Conda which is extremely popular on data science teams. It provides several commands for package management including the following:

- conda create
- conda install
- conda update
- conda remove
- conda info
- conda search
- conda config
- conda list

The command you'll use most often to create an environment with packages is given in Listing 3-2:

Listing 3-2. Conda create command for creating a new Conda environment

```
conda create --prefix ./envs matplotlib=3.5 numpy=1.2
```

For MLOPs, we want to go a step further and take a look at some more general package managers and their features.

Pipenv: Pipenv, which we'll use in our MLOps toolkit lab, tries to bring best in breed (bundler, composer, npm, yarn, and cargo) in package management to Python. Pipenv also treats Windows as a first class citizen which makes it ideal for some business environments. You don't have to worry about low level details of creating a virtualenv for your projects as pipenv handles this for you and even auto-generates the Pipfile describing package versions and Pipfile.lock which is used for deterministic builds. Since reproducibility of experiments is an important aspect of MLOps, deterministic builds are ideal especially for large projects where you have to juggle multiple versions of packages.

An example installing the Pandas package would be given in Listing 3-3.

Listing 3-3. pipenv command for creating a new Python environment

```
pipenv install pandas
```

You will then notice Pandas has been added to the Pipfile.

Installing Python Packages Securely

Have you ever been working on a model and realized you need to install xgboost or PyTorch or some other library? It worked before but this time the computer beeps and dumps a massive error log on your screen. You spend 3 hours debugging and searching on Stackoverflow for a solution only to realize the recipe only works for Windows, not Mac!

What should you do? Use Python environments. Python environments can save you a headache by providing isolation between software dependencies. We'll show you how to set this up in the next chapter. Once you set up a Python environment, you may notice you spend less time installing and managing Python package dependencies which frees up more time to work on data science tasks.

Navigating Open Source Packages for Data Scientists

Open source software packages are released under a license (typically permissive or copyleft like GPL) that allows its users to maintain control over using and accessing the software as well as distributing, studying, and changing. Many projects you use in data science are open source such as Scikit-Learn, PyTorch, and TensorFlow and can be found on GitHub.

Technical consideration when using open source software packages in data science are the following:

- PyPi and similar repositories can contain malware, and so packages should be trusted or scanned first (see Snyk⁴).
- Open source may be maintained by a community of dedicated volunteers so patches and updates may be at whim of the maintainer.
- Copyleft and other licensing may pose challenges for building enterprise software since you need to release the software under the same license (since software is often distributed as binaries).

⁴You can read more about the Snyk project at <https://docs.snyk.io/manage-issues/introduction-to-snyk-projects>

Common Packages for MLOps

Finally, we have enough knowledge to cover the central topic of this chapter which is packages specific to MLOps. Each of these packages provides pieces of the MLOps lifecycle such as experimentation, orchestration, training acceleration, feature engineering, or hyperparameter tuning. We can broadly separate these packages into two camps: ModelOps and DataOps.

DataOps Packages

DataOps is a collection of best practices, processes, and technologies borrowed from Agile software engineering that are designed to improve metrics like data quality, efficient data management, and continuous data delivery for data science and more broadly analytics. We need DataOps practices and experts when we're in the data engineering part of the MLOps lifecycle. Still, there are many concepts unique to MLOps such as feature groups and model registries that typical data engineering solutions do not have. In the following, we've compiled some of the tools you might encounter when working in the first stages of the MLOps lifecycle: data collection, data cleaning, feature engineering, and feature selection.

Jupyter Notebook

Jupyter notebooks as mentioned are a useful alternative to a local code editor like Visual Studio Code. You can use notebooks for prototyping code and running experiments. However, for MLOps, a Python script is preferable to a notebook for code for a number of reasons. For example, when you source control a Jupyter notebook, it is actually a JSON file that contains a combination of source code, text, and media output. This makes it more difficult to read the raw file compared to a Python script where you can read line by line.

Python scripts are also a standard way to represent code outside of data science, and you can use many different code editors from Visual Studio Code to text-based source code editors like Sublime Text, but beyond maintaining and readability, writing code as a script enables you to create larger software projects because your code can be organized into modules, packages. This structure is very important and enables you to understand the way the project is organized, reuse code, set up tests, and use automated tools like linters that make the software development process more efficient. Therefore, I hope you will consider using Python scripts with a code editor of your choice as opposed to Jupyter notebooks for production code.

JupyterLab Server

If you do insist on using Jupyter notebooks, there are a number of environments available. One environment we already mentioned was Google Colab, but if you want to run your notebook locally and have a customizable environment that could also be deployed as a service, you might consider JupyterLab.

JupyterLab server is a Python package that sits between JupyterLab and Jupyter Server and provides RESTful APIs and utilities that can be used with JupyterLab to automate a number of tasks for data science and so is useful for MLOps. This also leads us to another widely used platform for MLOps that also comes with a notebook-based environment.

Databricks

Databricks was created by the founders of Apache Spark, an open source software project for data engineering that allows training machine learning models at scale by providing abstractions like the PySpark dataframe for distributed data manipulation.

Databricks provides notebooks, personas, SQL endpoints, feature stores, and MLFlow within its PaaS offering which is also available in multiple cloud vendors including Azure and AWS with their own flavor of Databricks.

Besides MLFlow, a vital tool for an MLOps engineer to track model metrics and training parameters as well as register models and compare experiments, Databricks has a concept of a delta lakehouse where you can store data in parquet format with a delta log that supports features like time travel and partitioning.

We'll mention this briefly, but it could have its own chapter since this is a massive topic. Koalas is a drop-in solution although not 100% backward compatible with Pandas (of course, there's a lag between when a feature is supported in Pandas and when it becomes generally available in Pandas for Spark), but this is a great tool to add to your toolkit when you need to scale your workflow. While doing development in PySpark, you don't have to re-write all of your code; you use following import at the top of your file and use it like you would Pandas.

Dask: Dask is another drop-in solution for data wrangling similar to Pandas except with better support for multiprocessing and large data sets. The API is very similar to Pandas, but unlike Koalas or Pandas API for Spark, it is not really a drop-in solution

Modin: While Dask is a library that supports distributed computation, Modin supports scaling Pandas. It supports various backends including ray and Dask. Again, it's not 100% backward compatible and has a much smaller community than Pandas, so use with caution on a real project.

ModelOps Packages

ModelOps is defined by Gartner as “ focused primarily on the governance and lifecycle management of a wide range of operationalized artificial intelligence and decision models, including machine learning, knowledge graphs, rules, optimization, linguistic, and agent-based models.” Managing

models is difficult in part because there's code and data and many different types of models as we've seen from reinforcement learning to deep learning to shallow models in scikit-learn and bespoke statistical models.

We list some of the most popular tools for ModelOps in the following that you may encounter when you work in the later half of the MLOps lifecycle which includes model training, hyper-parameter tuning, model selection, model deployment, model management, and monitoring.

*Ray*⁵: Ray is a great tool for reinforcement learning; it is based on the actor model of distributed computation, in computer science,⁶ and allows you to use decorators to scale out functions which is convenient when you don't want to rewrite a lot of code.

*KubeFlow*⁷: KubeFlow is another open source machine learning tool for end to end workflows. It is built on top of Kubernetes and provides cloud-native interfaces for building pipelines and containerizing various steps of the machine learning lifecycle from training to deployment.

*Seldon*⁸: Have you ever been asked to deploy your machine learning models to production? First of all, what does that even mean? There are many ways to deploy a model. You could put it in a model registry, and you could containerize your model and deploy it to Docker Hub or another container registry, but for some use cases especially if an end user is going to be interacting with your model on demand, you'll be asked to expose the model as an API.

Building an API is not a trivial task. You need to understand gRPC or REST and at least be familiar with a framework like Flask if you're using Python. Fortunately, there are tools like Seldon that allow you to shortcut

⁵The Ray framework documentation can be found at <https://docs.ray.io/en/latest/>

⁶Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. <https://apps.dtic.mil/sti/pdfs/ADA157917.pdf>

⁷The KubeFlow project documentation can be found at www.kubeflow.org/docs/

⁸The Seldon project documentation can be found at <https://docs.seldon.io/projects/seldon-core/en/latest/index.html>

some of these steps and deploy models as gRPC or REST endpoints. Seldon in particular offers two models for servers: reusable and nonreusable. The definition of each is stated in the following.

- *Reusable model servers:* These are prepackaged model servers. You can deploy a family of models that are similar to each other, reusing the server. You can host models in an S3 bucket or blob storage account.
- *Nonreusable model servers:* This option doesn't require a central model repository, but you need to build a new image for each model as it's meant to serve a single model.

This leads us to the standard solution right now for registering your model, MLFlow. You had to create your own model storage and versioning system and way to log metrics and keep track of experiments. All of these important model management tasks (ModelOps) are made easier with MLFlow.

Model Tracking and Monitoring

MLFlow⁹ is the standard when it comes to creating your own experimentation framework. If you've ever developed loss plots and kept track of model metrics and parameters during hyper-parameter tuning, then you need to incorporate MLFlow into your project.

You can set up the MLFlow infrastructure as a stand-alone or part of Databricks (the original developers). We'll see this in action in later chapters.

⁹ MLFlow project documentation can be found at <https://mlflow.org/docs/latest/index.html>

*HyperOpt*¹⁰: Hyperopt is a framework for Bayesian hyper-parameter tuning, often done after the cross validation step but before training a model on the entire data set. There are also many algorithms available depending on the type of parameter search you need to do including the following:

- Random search
- Tree of Parzen Estimators
- Annealing
- Tree
- Gaussian Process Tree

*Horovod*¹¹: Horovod is a distributed deep learning framework for TensorFlow, Keras, PyTorch, and Apache's MXNet. When you need to accelerate the time it takes to train a model, you have the choice between GPU accelerated training and distributed training. Horovod is also available on Databricks and can be a valuable tool for machine learning at scale.

Packages for Data Visualization and Reporting

If you've ever had to do a rapid EDA or exploratory data analysis, you know how tedious it can be to have to write code for visualizations. Some people like writing algorithms and don't like visualization, whereas others who are good at libraries like Matplotlib or Seaborn become the de facto visualization experts on the team.

¹⁰The Hyperopt project can be found on GitHub at <https://github.com/hyperopt/hyperopt>

¹¹The Horovod project source code can be found at <https://github.com/horovod/horovod>

From an MLOps perspective, visualizations can be an “odd one out” in a code base and are difficult to deploy since creating interactive plots and dashboards requires special knowledge and tools. You should at least be familiar with a couple tools beyond Matplotlib for exploratory data analysis including the following:

- *Dash*¹²: Python library for creating interactive dashboards
- *PowerBI*: Visualization software from Microsoft. Useful for data science since you can embed Python and deploy to cloud

Lab: Developing an MLOps Toolkit Accelerator in CookieCutter

This lab is available on the Apress GitHub repository associated with this book. You will see in Chapter 3 the `mlops_toolkit` folder. We will use a package called `cookiecutter` to automate the process of setting up tests, train, data, models, and other folders needed in future chapters. Figure 3-19 shows the toolkit folders.

¹²The Dash project can be found on GitHub at <https://github.com/plotly/dash>

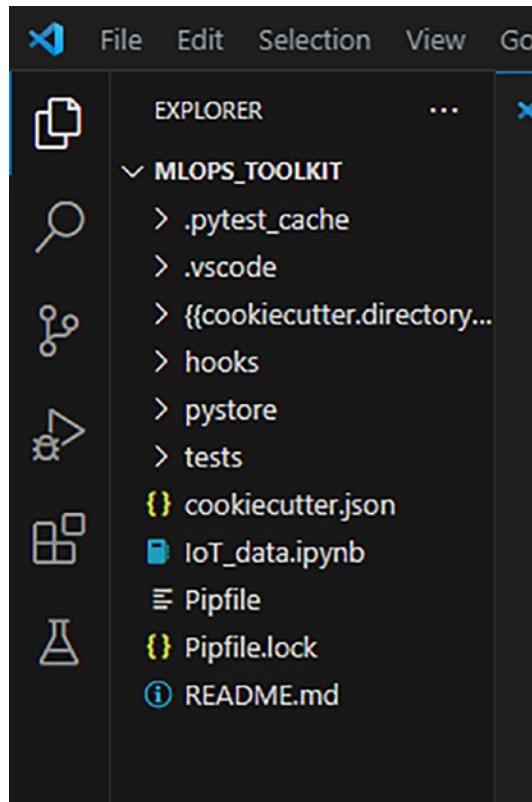


Figure 3-19. *MLOps toolkit folder structure*

You might be wondering what the point of having a template like this is. The primary reason is it goes toward establishing standards and code structure that borrows from experience across several industries. This pattern is tried and proven, and although it is slightly opinionated on use of testing framework and names of folders, you can easily customize it to your purposes.

We'll do exactly this by installing several packages that can support other stages of the MLOps lifecycle such as model training, validation, hyper-parameter tuning, and model deployment. The steps for setting up the lab are as follows:

Step 1. Clone the project locally and run the following command to open vs code:

Listing 3-4. Shortcut for opening Visual Studio Code¹³

code .

Step 2. Start a new vs. code terminal session (here we're using PowerShell but you can also use Bash) and cd into the `mlops_toolkit` directory. Figure 3-20 shows the root directory.

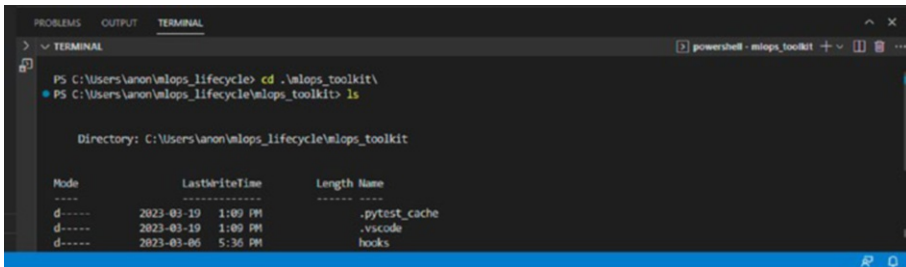


Figure 3-20. Root directory for MLOps toolkit supplementary material

Step 3. Clear the screen with the clear command and type as shown in Listing 3-5.

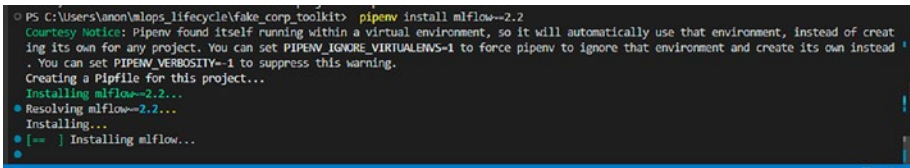
Listing 3-5. Installing Pandas package with a specific version number using Pipenv

```
pipenv install pandas~=1.3
```

Step 4. Check the Pipfile containing the following lines.

Step 5. Repeat steps 2–3 for the following packages: `numpy`, `pytest`, `hypothesis`, `sckit-learn`, `pyspark`, and `mlflow`. By default, the latest versions will be installed, but we recommend using the `~` operator with a major.minor version to allow security patches to come through. The output is shown in Figure 3-21.

¹³Tips and Tricks for Visual Studio Code <https://code.visualstudio.com/docs/getstarted/tips-and-tricks>



```

PS C:\Users\anon\mlops_lifecycle\fake_corp_toolkit> pipenv install mlflow==2.2
Courtesy Notice: Pipenv found itself running within a virtual environment, so it will automatically use that environment, instead of creating its own for any project. You can set PIPENV_IGNORE_VIRTUAL_ENVS=1 to force pipenv to ignore that environment and create its own instead. You can set PIPENV_VERBOSITY=-1 to suppress this warning.
Creating a Pipfile for this project...
Installing mlflow==2.2...
  Resolving mlflow==2.2...
  Installing...
  [== ] Installing mlflow...

```

Figure 3-21. The result of installing some Python packages with pipenv

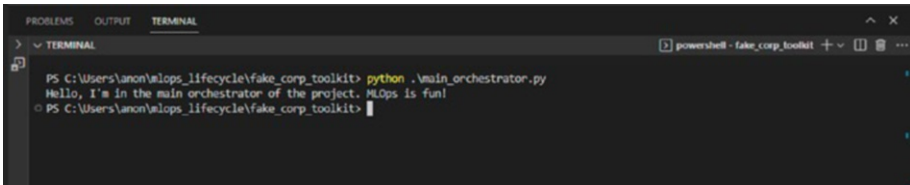
Step 6. **CTRL + SHIFT + P** to open the vs code command palette. Type python and choose pytest in the dropdown and select /tests folder.

Step 7. Click the tests icon in the Activity Bar and run all tests by clicking the “run” button.

Step 8. Run the following command with the custom name of your project.

Step 9. Cd into the folder you created and customize it to your own data science project. Here I used main_orchestrator.py for the file name.

Step 10. Python main_orchestrator.py should print a message to the screen as shown in Figure 3-22.



```

PROBLEMS OUTPUT TERMINAL
> TERMINAL
powershell - fake_corp_toolkit
PS C:\Users\anon\mlops_lifecycle\fake_corp_toolkit> python .\main_orchestrator.py
Hello, I'm in the main orchestrator of the project. MLops is fun!
PS C:\Users\anon\mlops_lifecycle\fake_corp_toolkit>

```

Figure 3-22. Running the main orchestrator should print a message to your screen

Step 11. Go through the Git fundamentals lab again if necessary, and add code and data version control by running two commands in a terminal (works both in PowerShell and Bash) as given in Listing 3-6:

Listing 3-6. Initializing source and data version control commands in a repo¹⁴

```
git init
dvc init
```

That's it! Not so bad and we've already set up tests, our very own custom monorepo, installed packages to support various stages of the lifecycle, and know how to set up code version control and data version control. In the next chapters, we'll go through the gritty details of MLOps infrastructure, model training, model inference, and model deployment, developing our toolkit further.

Summary

In this chapter, we gave an introduction to several tools for MLOps and data science including version control both for source code and data. We also talked about the differences between Jupyter notebooks and Python scripts and why Python scripts are the preferred format for MLOps. We looked at code editors like Visual Studio Code for working with Python scripts and talked about some of the tools, packages, and frameworks you may encounter in an MLOps workflow. Here is a summary of what we learned:

- Data and Code Version Control Systems
- Model Development and Training
- Model Packages and Deployment
- Model Tracking and Monitoring

In the next chapter, we will shift our attention to infrastructure and look at how we can begin to use some of the tools discussed in this chapter to build services to support the various stages of the MLOps lifecycle.

¹⁴DVC User Guide: <https://dvc.org/doc/user-guide>