

Good Research Code handbook

Patrick Mineault

Wednesday 2nd November, 2022

Contents

I. Introduction	iii
Preface to the print edition	v
How to read this book	ix
The neuroscience of coding: a primer	xiii
II. Lessons	1
1. Setting up your project	3
2. Keep things tidy	15
3. Write decoupled code	21
4. Testing your code	33
5. Write good documentation	43
6. Document your project	49
7. Make coding social	57
8. A sample project: Zipf's law	61
III. Extras	69
How to test numerical code: CKA	71
Use good tools	79
Acknowledgements	83

Part I.

Introduction

Preface to the print edition

Introduction

Is my code fast? No. But is it well documented? No. But does it work? Also no.

—@KyleMorgenstein

This handbook is for grad students, postdocs and PIs who do a lot of programming as part of their research. It will teach you, in a practical manner, how to organize your code so that it is easy to understand and works reliably.

Most people who write research code are not trained in computer science or software engineering. It can feel like an uphill battle when you have to write code without the right training. Do you ever:

- Feel like you don't know what you're doing
- Feel like an impostor
- Write code with lots of bugs
- Hate your code and don't want to work on it
- Have trouble finishing projects
- Contemplate buying a small organic farm in upstate Vermont and read far too much about goat husbandry

Did I hit a nerve? Yes?! Then you're in for a treat! This book will help you get from 0 to 1 on good software engineering practices for research projects.

Prerequisites

I've tried to write this book in a progressive manner: some content is targeted at complete novices, other to programmers who are farther along on their journey. However, I generally assume that you have some familiarity with the following:

- **Python:** this intro is Python-centric. You can write good code for Matlab, R, or Julia, but we won't cover that here. You don't need to be a Python expert, but you'll get the most out of this if you've been using Python on a regular basis for at least a month, and if you have some passing familiarity with the python data science ecosystem (numpy, matplotlib, pandas, etc.).

- **Git & Github:** a lot of the practices introduced here will require you to change your code, which could cause existing functionality to break. You might even accidentally delete something important! Mastering git and github means you will have a time machine for your code, so you can revert to an earlier state. [There's a great intro to git for beginners from software carpentries](#).
- **The command line:** You will need to run some commands on the command line to implement some of the advice in this book. I'm going to assume that you have some familiarity with running commands from a Unix-style shell (e.g. bash). [There's a great intro to the unix shell from software carpentries](#). If you're using Windows, you will still be able to run many tools from the Windows command prompt. Long term, your life will be easier if you [install the Windows Subsystem for Linux \(WSL\)](#) which will give you access to a Unix-style shell.

Some of the examples I use are neuroscience-inspired—but neuroscience background is absolutely not a requirement. Appreciation for unicorns is a big plus.

Why did I make this?

I'm [Patrick Mineault](#). I did my PhD in computational neuroscience at McGill University, in Montreal. I wrote a lot of not very good code, mostly in Matlab. One time, my code was in a non-working state for an entire month—I would furiously type on the keyboard all day in the hopes it would eventually work, and it didn't. It made me sad. But I managed to graduate. Then I did a postdoc. More of the same.

Eventually, I decided to properly study CS. I studied data structures, algorithms and software engineering practices, and I got a big-boy job as a software engineer at Google in California. It was then that I learned the error of my ways. I had lost time during my research days because I didn't know how to organize and write code that didn't self-destruct out of spite. But this is fixable! With knowledge!

At the invitation of Ella Batty, I gave a workshop for the students in neuroscience at Harvard in January of 2021 on writing good research code. The feedback was overwhelmingly positive, so I decided to expand it into this handbook. I hope you enjoy it!

About the print edition

If you're reading this, congratulations! You're reading the bespoke, artisanal, small-batch print edition. A few folks have asked me for a print edition, and I was happy to oblige. It's an experiment that I hope to expand in the coming years. It was made possible by last-minute feature additions in CurveNote, the wonderful new project that aims to make MyST Markdown the standard for publishing scientific articles and books.



Figure 1.: It me!

How to read this book

Roadmap

This handbook covers practices and tools from big to small. We'll discuss how to write and organize modular code, how to write docs, how to make your code robust. We'll finally reveal the correct number of spaces to use to indent code . At the end of the main lessons, we'll go through an example project step-by-step.

This roadmap shows some of the concepts we'll cover in the book. Take stock of where you are now. Come back to this figure after you've read parts of this book: you'll be surprised how much you can learn in a few hours!

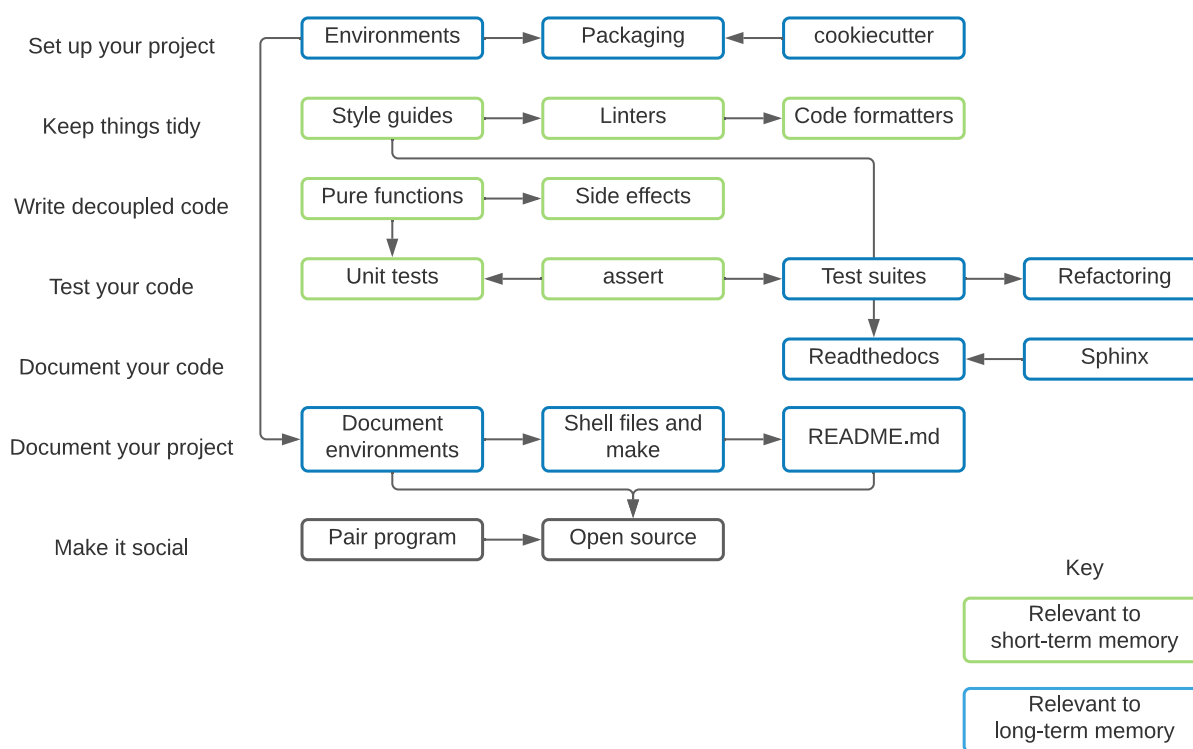


Figure 1.: Some concepts we'll cover in this book. I've highlighted in green and blue different concepts which are relevant to short-term and to long-term memory, respectively: we'll discuss what that means in the next section. How many concepts are you familiar with now? How many do you know well?

Conventions

This book uses a number of conventions throughout. The command line is indicated by a \$ prompt:

```
$ ls -al
```

The Python command prompt is indicated by >>>:

```
$ python
>>> import antigravity
```

Sometimes I will ask you a question, and will hide the answer behind a spoilers panel. For example, what is the answer to life, the universe, and everything?

Spoilers

42

At the end of each main lesson, I ask you to put one of the lessons into practice through a 5-minute exercise. It looks like this:

5-minute exercise

Brush your teeth.

Code

This handbook refers to code in several repositories:

- [True neutral cookie cutter](https://github.com/patrickmineault/true-neutral-cookiecutter): github.com/patrickmineault/true-neutral-cookiecutter
- [CKA example](https://github.com/patrickmineault/codebook-examples): github.com/patrickmineault/codebook-examples
- [Zipf's law example project](https://github.com/patrickmineault/zipf): github.com/patrickmineault/zipf
- [Source for the text of the book](https://github.com/patrickmineault/codebook): github.com/patrickmineault/codebook
- [Tweaked sphinx book theme](https://github.com/patrickmineault/sphinx-book-theme): [patrickmineault/sphinx-book-theme](https://github.com/patrickmineault/sphinx-book-theme)

You can use these as references when you're working on your own projects.

Breaking the cycle of frustration

Learning to code is a lifelong journey. The upper ceiling on programming skills is very high. Much like cooking, coding can be done on a purely utilitarian basis, or it can be elevated to high art. [Professional programmers with decades of experience go on months-long retreats to acquire new skills.](#) A good frame for getting better at coding is to think of it as a *craft*. Reading this book is a great way to refine your craft through focused practice .

When I talk to students about writing code, a lot of them describe feelings like:

- guilt
- shame
- cringe
- frustration

Note

Julia Evans [has a lovely zine](#) on dealing with frustrating bugs.

Much of that frustration comes from a mismatch between what you want to accomplish (a lot) and what you have the ability to accomplish (not as much as you'd like). Programming instruction often emphasizes *exploration*, embracing errors as learning opportunities, and encouraging you to let your imagination run free. This theory of change implies that you will learn a lot by simply programming a lot every day. You might then feel guilty and inadequate when you're not as proficient as you want after several years of daily programming.

In fact, unstructured exploration is a very inefficient way to learn a complex skill like programming. It's like expecting a student to rediscover calculus by themselves after teaching them the rudiments of algebra! Research shows that *structured instruction*—like the one in this book—is far more effective at teaching programming skills. You're taking the right step by reading this book!

Our social contract

You might have experience talking about computer stuff with somebody more experienced, and left feeling discouraged. Perhaps they were dismissive, or snooty, or just kind of a jerk. It's an unfortunate tendency in our profession, and I want to break that cycle, because I think that this lack of *psychological safety* can make it hard to become proficient at coding.

Important

This is a safe space. You're in a learning environment. There will be no tests. The advice I give here is non-binding and non-sanctimonious.

It's ok to write bad code when you're learning or you're in a hurry. You have deadlines! Remember to come back to the bad code and tidy it up after. Don't let the perfect be the enemy of the good. And remember, once you are empowered with this new knowledge, to be nice to beginners who are going through the same journey that you have.

The neuroscience of coding: a primer

Brains & coding

Optional

This section is optional—it's about the neuroscience of writing code. Feel free to skip to the next section if it doesn't speak to you.

What makes coding uniquely difficult? Coding is hard on your memory. This book's theme is that **writing good research code is about freeing your memory**. Neuroscientists distinguish different subtypes of memory, and coding strongly depends on two subtypes:

- Working memory
- Long-term memory

Let's look at how these two types of memory are involved during programming.

Working memory

Note

Felienne Hermans has a fantastic [talk](#) and [book](#) on programming in the brain.

What do you think coding looks like in the brain? The term *programming language* makes it seem that reading code is like reading natural language. Neuroscientists have looked at what circuits are engaged when people read code. [Ivanova and colleagues at MIT](#) measured brain activity while people read Python in the scanner. They found that activations didn't overlap much with conventional language and speech areas.

Instead, they found high overlap with the **multiple-demand system**, a network of areas typically recruited during math, logic and problem solving. The multiple-demand system is also involved in **working memory**, a type of memory that can hold information for a short amount of time. You may have heard that people can only hold approximately 7 items in working memory—for instance, the digits of a phone number. This idea was popularized by the Harvard psychologist George Miller in the 1950's. While there's debate about the exact number of items we can remember in the short term, neuroscientists generally agree that our working memory capacity is very limited.

Programming involves juggling in your mind many different pieces of information: the definition of the function you want to write, the name of variables you defined, the boundary conditions of the `for` loop you're writing, etc. If you have too many details to juggle, eventually you will forget about one detail, and you will have to slowly reconstruct that detail from context. Do that enough times, and you will completely cease to make progress. Your working memory is very precious indeed.

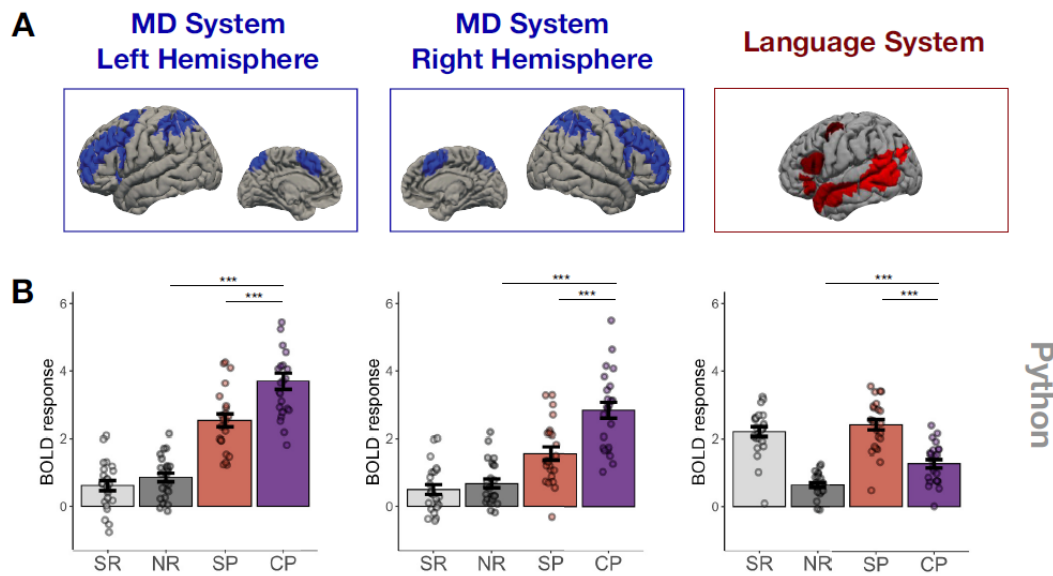


Figure 1.: Code problems (CP; purple bars) created activations with higher overlap with the multiple demand system (left and center) than with the language system (right) compared with other tasks like sentence problems (SP), sentence reading (SR) or non-word reading (NR). From Ivanova et al. (2020), used under a CC-BY 4.0 license.

Saving our working memory

One of our strongest weapons against overloading our working memory is *convention*. Conventions mean that you don't have to remember details in your working memory: you can rely on your long-term memory instead. For example, let's say you want to call a helper function that splits a url into its constituent parts. You might instinctively know that the function to call is `split_url` and not `splitUrl` or `splitURL` or `URLSplitterFactory().do`. That's because Python has a convention that says that you separate words in a variable name with underscores (snake case). If you abide by the convention, you've just saved yourself a working memory slot. To be clear, it's a completely arbitrary convention: JavaScript uses a different convention (camel case) and it works fine.

We'll see many more examples of organizing code such that it saves our working memory:

- Writing small functions
- Writing functions with low number of side effects
- Writing decoupled functions
- Following the Python style guide to the letter
- Using an auto-formatter

Your working memory is precious, save it!

Long-term memory

What is this?

—You, squinting at code you wrote a year ago

Research code in particular is challenging on **long-term memory**. This is because:

- The project's endpoint might be unclear
- Correct can be hard to define
- There can be lots of exploration and dead ends before you produce a unit of research
- Sometimes, manual steps are involved requiring human judgement
- You have to remember all the dead ends for the code to even make sense

Has this ever happened to you?

You work on a project for many months, and you submit a paper. You receive reviews months later requiring revisions. You sit down to code and it takes you several days to figure out how to run a supplementary analysis that ended up taking only a few lines of code.

While the scientific method, as traditionally taught, is fairly linear, real lab work often involves a high nonlinear process of discovery.

Future you will have forgotten 90% of that process. Code that sticks to convention, is tidy and well-documented will be far easier to use in the future than clever, obtuse code. Boring code is good code!

Discussion

Simple is better than complicated. Complicated is better than complex.

—The Zen of Python

We've seen that coding, especially research code, is difficult along two axes:

- working memory
- long-term memory

We've discussed some of our overall strategies to deal with these limitations, namely:

- using convention
- keeping code tidy
- documenting our code

Let's see how you can implement these strategies in practice. Let's jump in!

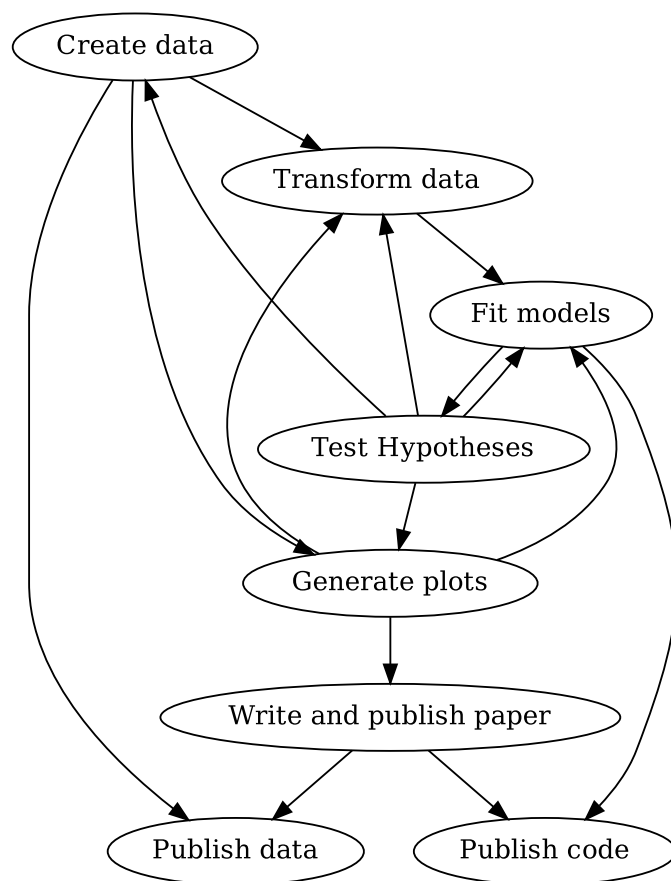


Figure 2.: Generating a research paper can be a messy process.

Part II.

Lessons

1. Setting up your project

Set up your project

Setting up a organized project will help you remain productive as your project grows. The broad steps involved are:

1. Pick a name and create a folder for your project
2. Initialize a git repository and sync to Github
3. Set up a virtual environment
4. Create a project skeleton
5. Install a project package

The end result will be a logically organized project skeleton that's synced to version control.

Warning

I will present most of the project setup in the terminal, but you can do many of these steps inside of an IDE or file explorer.

Pick a name and create a folder for your project

When you start a project, you will need to decide how to structure it. As an academic, a project will tend to naturally map to a paper. Therefore, **one project = one paper = one folder = one git repository** is a generally a good default structure.

Note

You might want to create extra standalone projects for tools you re-use across different papers.

Pick a short and descriptive name for your project and create a folder in your Documents folder. For instance, when I created the project for this book, the first step was to create the codebook folder:

```
~/Documents$ mkdir codebook
```

Initialize a git repository and sync to Github

Since git is such a core tool to manage code-heavy projects, I recommend that you set it up immediately. The way I prefer to do this is by going to [Github](#) and clicking the big green **New** button to create a new repository. I name the remote the same as my local folder and hit **Create Repository**.

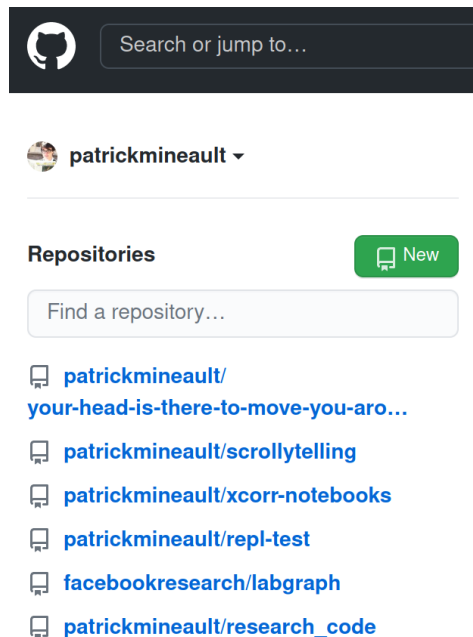


Figure 1.1.: The big green New button.

I then follow Github's instructions to initialize the repo. In `~/Documents/codebook`, I run:

Note

I've never attempted to remember these commands. I always copy and paste.

```
echo "# codebook" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/patrickmineault/codebook.git
git push -u origin main
```

How often do you think you should commit to git?

Spoilers

Depending on your pace, you should aim to commit your code from *a few times a day* to *a few times per week*. Don't wait until the project is almost finished before you start to commit.

The general rule of thumb is that one commit should represent a unit of related work. For example, if you made changes in 3 files to add a new functionality, that should be *one* commit. Splitting the commit into 3 would lose the relationship between the changes; combining these changes with 100 other changed files would make it very hard to track down what changed. Try to make your git commit messages meaningful, as it will help you keep track down bugs several months down the line.

If you don't use git very often, you might not like the idea of committing to git daily or multiple times per day. The git command line can feel like a formidable adversary; GUIs can ease you into it. I used to use the git command line exclusively. These days, I tend to prefer [the git panel in VSCode](#).

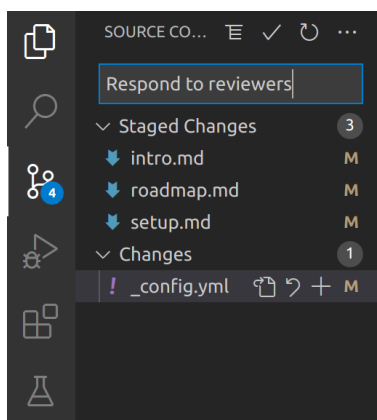


Figure 1.2.: The git panel in VSCode.

Set up a virtual environment

Why do I use virtual Python environments? So I don't fuck up all my local shit.

—[Nick Wan](#)

Many novices starting out in Python use one big monolithic Python environment. Every package is installed in that one environment. The problem is that this environment is not documented anywhere. Hence, if they need to move to another computer, or they need to recreate the environment from scratch several months later, they're in for several hours or days of frustration.

The solution is to use a *virtual environment* to manage dependencies. Each virtual environment specifies which versions of software and packages a project uses. The specs can be different for different projects, and each virtual environment can be easily swapped, created, duplicated or destroyed. You can use software like conda, pipenv, poetry, venv, virtualenv, asdf or

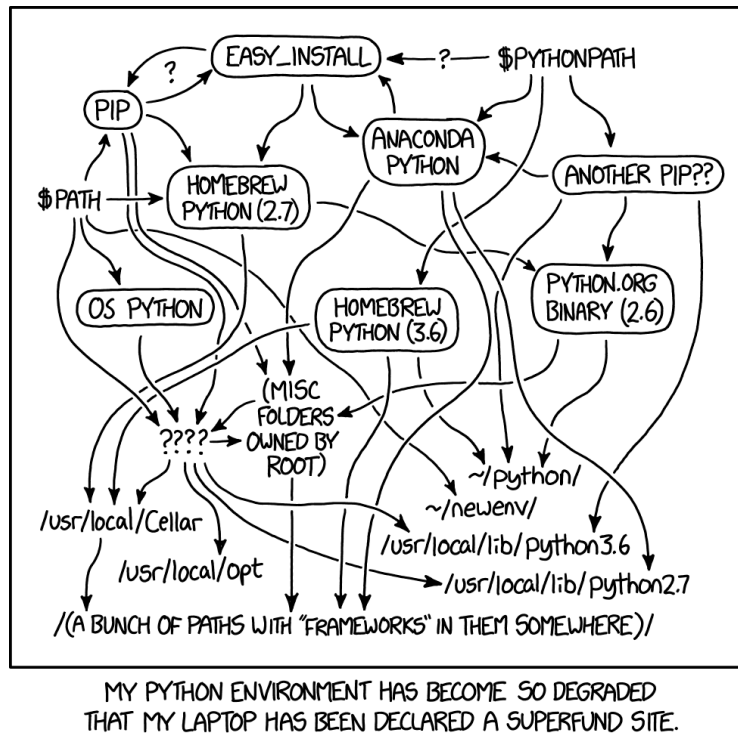


Figure 1.3.: Python environments can be a real pain. From xkcd.com by Randall Munroe.

`docker`—among others—to manage dependencies. Which one you prefer is a matter of personal taste and [countless internet feuds](#). Here I present the `conda` workflow, which is particularly popular among data scientists and researchers.

Conda

`Conda` is the *de facto* standard package manager for data science-centric Python. `conda` is both a package manager (something that installs package on your system) and a virtual environment manager (something that can swap out different combinations of packages and binaries—virtual environments—easily).

Once `conda` is installed—for instance, through `miniconda`—you can create a new environment and activate it like so:

```
~/Documents/codebook$ conda create --name codebook python=3.8
~/Documents/codebook$ conda activate codebook
```

From this point on, you can install packages through the `conda` installer like so:

```
(codebook) ~/Documents/codebook$ conda install pandas numpy scipy matplotlib seaborn
```

Now, you might ask yourself, can I use both `pip` and `conda` together?

Spoilers

You can use pip inside of a conda environment. A big point of confusion is how conda relates to pip. For conda:

- Conda is both a package manager and a virtual environment manager
- Conda can install big, complicated-to-install, non-Python software, like `gcc`
- Not all Python packages can be installed through conda

For pip:

- pip is just a package manager
- pip only installs Python packages
- pip can install every package on PyPI in addition to local packages

conda tracks which packages are pip installed and will include a special section in `environment.yml` for pip packages. [However, installing pip packages may negatively affect conda's ability to install conda packages correctly after the first pip install.](#) Therefore, people generally recommend installing **big conda packages first**, then installing **small pip packages second**.

Export your environment

To export a list of dependencies so you can easily recreate your environment, use the `export env` command:

```
(codebook) ~/Documents/codebook$ conda env export > environment.yml
```

You can then commit `environment.yml` to document this environment. You can recreate this environment—when you move to a different computer, for example—using:

```
$ conda env create --name recoveredenv --file environment.yml
```

This export method will create a well-documented, perfectly *reproducible* conda environment on your OS. However, it will document low-level, OS-specific packages, which means it won't be *portable* to a different OS. If you need portability, you can instead write an `environment.yml` file manually. Here's an example file:

```
name: cb
channels:
  -conda-forge
  -defaults
dependencies:
  -python=3.8
```

```
-numpy=1.21.2
-pip
-pip:
  -tqdm==4.62.3
```

pip and conda packages are documented separately. Note that pip package versions use == to identify the package number, while conda packages use =. If you need to add dependencies to your project, change the `environment.yml` file, then run this command to update your conda environment:

```
(cb) $ conda env update --prefix ./env --file environment.yml --prune
```

You can [read more about creating reproducible environments in the Carpentries tutorial on conda](#). You can also [use the `environment.yml` file for this book's repo](#) as an inspiration.

Create a project skeleton

Note

This project skeleton combines ideas from [shablona](#) and [good enough practices in scientific computing](#).

Many different programming frameworks—Ruby on Rails, React, etc.—use a highly consistent directory structure from project to project, which makes it seamless to jump back into an old project. In Python, things are much less standardized. I went into a deep rabbit hole looking at different directory structures suggested by different projects. Here's a consensus structure you can use as inspiration:

```
|-- data
|-- docs
|-- results
|-- scripts
|-- src
|-- tests
-- .gitignore
-- environment.yml
-- README.md
```

Let's look at each of these components in turn.

Folders

- **data:** Where you put raw data for your project. You usually won't sync this to source control, unless you use very small, text-based datasets (<10 MBs).
- **docs:** Where you put documentation, including Markdown and reStructuredText (reST). Calling it `docs` makes it easy to publish documentation online through Github pages.

- `results`: Where you put results, including checkpoints, hdf5 files, pickle files, as well as figures and tables. If these files are heavy, you won't put these under source control.
- `scripts`: Where you put scripts—Python and bash alike—as well as .ipynb notebooks.
- `src`: Where you put reusable Python modules for your project. This is the kind of python code that you import.
- `tests`: Where you put tests for your code. We'll cover testing in a later lesson.

You can create this project structure manually using `mkdir` on the command line:

```
$ mkdir {data,docs,results,scripts,src,tests}
```

Files

- `.gitignore` contains a list of files that git should ignore.
- `README.md` contains a description of your project, including installation instructions. This file is what people see by default when they navigate to your project on GitHub.
- `environment.yml` contains the description of your conda environment.

`.gitignore` can be initialized to the following:

```
*.egg-info
data
```

A `README.md` should have already been created during the initial sync to Github. You can either create an `environment.yml` file manually or export an exhaustive list of the packages you are currently using:

```
$ conda env export > environment.yml
```

Install a project package

Warning

Creating a project package is slightly annoying, but the payoff is quite substantial: your project structure will be clean, you won't need to change Python's path, and your project will be pip installable.

You might notice a flaw in the preceding project structure. Let's say you create a reusable `lib.py` under the `src` folder, with a function `my_very_good_function`. How would you reference that function in `scripts/use_lib.py`? This doesn't work:

```
>>> from ..src.lib import my_very_good_function
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: attempted relative import with no known parent package
```

You need to tell Python where to look for your library code. You have two options, change your Python path, or create an installable package. I recommend the installable package route, but cover the Python path route first because you're likely to encounter it in other projects.

Change your Python path (not recommended)

You can put the `src` folder on your Python path. To do so, you can [append the `src` folder to the system variable `PYTHONPATH` when `bash` starts up \(in `~/.bashrc`\)](#). You might alternatively dynamically append to the system path from Python, via:

```
import sys
sys.path.append('/home/me/Documents/codebook/src')

from src.lib import my_very_good_function
```

This pattern is also frequently used in jupyter notebooks—I often see it in code cells at the top of notebooks.

The disadvantage of changing the path is that it tends to be pretty brittle. You have to hard-code the name of folders in multiple places. If they move, you will break your package. It won't work on another computer with different paths, so it will make it hard to share your project with colleagues. Furthermore, dynamic paths don't play well with IDEs like [vscode](#) that can only look in the static environment, so you won't get automatic code completion.

Create a pip-installable package (recommended)

This is a more scalable solution. [The packaging ecosystem in Python can feel frankly daunting](#), but a lot of it we don't need for our purposes. Creating a locally pip installable package actually only involves a few steps.

1. Create a `setup.py` file Create a `setup.py` file in the root of your project. Here's a minimal setup file:

```
from setuptools import find_packages, setup

setup(
    name='src',
    packages=find_packages(),
)
```

2. Create a `__init__.py` file Create an empty `__init__.py` file under the `src` directory. This will allow the `find_packages` function to find the package.

```
(codebook) ~/Documents/codebook $ touch src/__init__.py
```

Your files should now look like:

```
|-- data
|-- doc
|-- results
|-- scripts
|-- src
|   -- __init__.py
|-- tests
-- .gitignore
-- environment.yml
-- README.md
-- setup.py
```

3. pip install your package Now comes the fun part, installing the package. You can do so using:

```
(codebook) ~/Documents/codebook $ pip install -e .
```

`.` indicates that we're installing the package in the current directory. `-e` indicates that the package should be editable. That means that if you change the files inside the `src` folder, you don't need to re-install the package for your changes to be picked up by Python.

4. Use the package Once the package is locally installed, it can be easily used *regardless of which directory you're in*. For instance:

```
(codebook) ~/Documents/codebook $ echo "print('hello world')" > src/helloworld.py
(codebook) ~/Documents/codebook $ cd scripts
(codebook) ~/Documents/codebook/scripts $ python
>>> import src.helloworld
hello world
>>> exit()
(codebook) ~/Documents/codebook/scripts $ cd ~
(codebook) ~ $ python
>>> import src.helloworld
hello world
```

How does this work? When you install a package in editable mode, Python essentially adds your code to its path. That makes it available from anywhere. The path is changed in such a way that `conda`, `vscode` and other tools are aware that your package is installed, so all these tools will know where to find your code.

Note

To find out where the code for an installed package is located, print the module info in the Python console:

```
>>> import src
>>> src
<module 'src' from '/home/pmin/Documents/codebook/src/__init__.py'>
>>> import numpy as np
>>> np
<module 'numpy' from '/envs/cb/lib/python3.8/site-packages/numpy/__init__.py'>
```

5. (optional) Change the name of the package Note that the name of the folder which contains the code, `src`, becomes the name of the package. If you'd like to rename the package, for example to `cb`, change the name of the folder:

```
(codebook) ~/Documents/codebook $ mv src cb
```

If Python doesn't pick up your changes for whatever reason, re-install your package like so:

```
(codebook) ~/Documents/codebook $ pip install -e .
```

Note

setuptools knows which folder contains your package by looking for a `__init__.py` at the root of that folder.

Use the true-neutral cookiecutter

If doing all this for every new project sounds like a lot of work, you can save yourself some time using the *true neutral* cookiecutter, which creates the project skeleton outlined above automatically. `cookiecutter` is a Python tool which generates project folders from templates. You can install it in the base conda environment with:

```
(base) ~/Documents $ pip install cookiecutter
```

To create the `codebook` folder with all its subfolders and [setup.py](#), use the following:

```
(base) ~/Documents $ cookiecutter gh:patrickmineault/true-neutral-cookiecutter
```

Note

There are many other interesting cookiecutters. Check out the [data science cookiecutter](#) for a more elaborate data science project template.

This will create an instance of the `true-neutral-cookiecutter` project skeleton, which is hosted on my personal github. Follow the prompts and it will create the folder structure above, including the setup file. Next, pip install the package you've created for yourself, and sync to your own remote repository, following the github instructions.

Discussion**Note**

You can reorganize an existing project to align better with the guidelines here. **Make sure to back up everything!**

Using structured projects linked to git will help your long-term memory. You will be able to instantly understand how files are laid out months after you've last worked on that project. Using a virtual environment will allow you to recreate that environment in the far future. And git will give you a time machine to work with.

Writing for your future self has an added bonus: it can make it easier for *other people* to use your project. Consider this: everything at Google is in one giant repository with [billions of lines of code](#). As a new software engineer, you're invited to commit to that repository during your first week. Because everything is organized according to [strict conventions](#), so it's not as *terrifying* as it sounds to jump in. Structure is what enables sustainable growth.

5-minute exercise

Create an empty project with the true-neutral cookiecutter.

2. Keep things tidy

Does it spark joy?
—Marie Kondo

Keeping things consistent and tidy will free your working memory from having to remember extraneous information like variable names and paths.

Use the style guide

Generally, Python uses:

- Snake case for variables and module: `variable_name`, `my_module.py`
- Camel case for class name: `MyClass`
- Camel case with spaces for jupyter notebook: `Analyze Brain Data.ipynb`

You might know that Python generally uses 4 spaces for indentation, and that files are expected to be at most 80 columns long. These and many other elements of style are written in [PEP 8](#). Code that adheres to style tends to be easier to read.

Note

Big software companies like Google [have their own coding style guide](#). Even Guido von Rossum, inventor of Python, had to follow Google's style guide when he was working at Google.

Reading style guides is nobody's idea of a good time, but thankfully tools exist to help you maintain good coding style. If you prefer to eventually learn the rules, you can install `flake8` or `pylint`. Both tools are *linters*—detectors of bad style—which allow you to find and correct common deviations from the style guide. The ideal place to use the linter is inside of an IDE, [for example VSCode](#). It's also possible to use linters from the command line.

A more radical way to impose style is to use a *code formatter*. A linter suggests fixes which you implement yourself; a formatter fixes issues automatically whenever you save a file. `black` [imposes consistent Python style](#), and has plugins for all the popular IDEs. `black` is particularly useful to run on old code with haphazard style; run it once to upgrade code to a standard format.

Delete dead code

Code that gets developed over time can accumulate lots of dead ends and odds and ends. After a while, the majority of the code in your project might be dead: code that never gets called. You know who hates dead code? You, in three months. Navigating a project that contains stale or

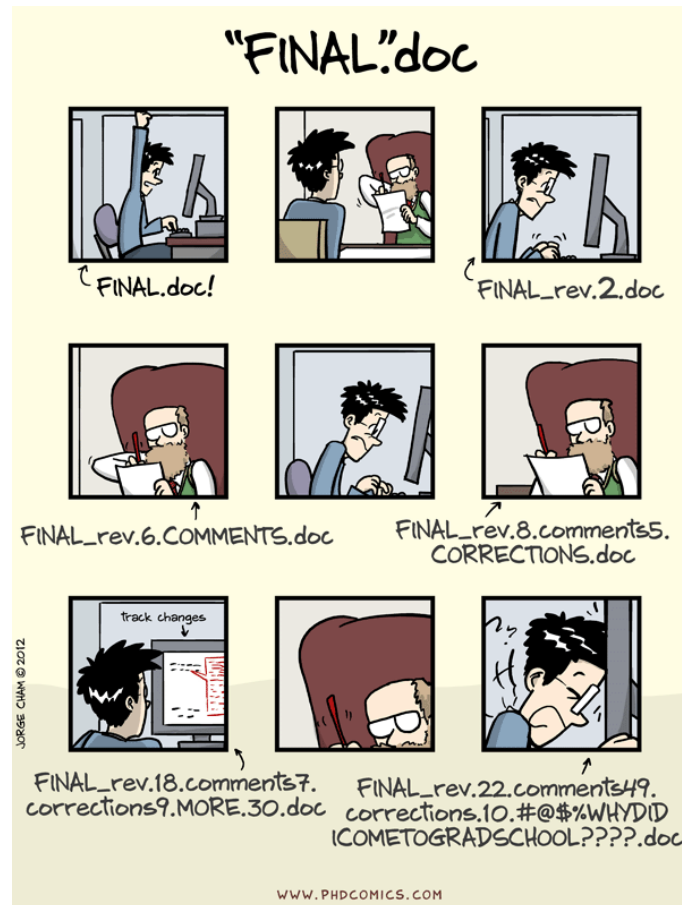


Figure 2.1.: Dead code is a liability. From "Piled Higher and Deeper" by Jorge Cham.
www.phdcomics.com

incorrect code can waste a huge amount of time. Whenever you're about to put aside a project for a long time—for instance, after submitting a manuscript—*clean up dead code*. Delete dead code from the main branch. With git and github, you have access to a time machine, so you can always revert if you mess up.

If you're not used to this workflow, you might be scared of messing something up. You can download an archive of the repo before the cleanup to reassure yourself. If you've been diligent about committing and pushing code to Github, however, deleting dead code is a safe process. [vulture can help you find dead code in your codebase](#). Unit tests can help you verify that your codebase will still run after you eliminate dead code—we will cover this in a later lesson.

Keep jupyter notebooks tidy

If you use notebooks to develop software, you are probably using the wrong tool.
—Yihui Xie

Jupyter notebooks present a special challenge to keep tidy because of their inherently nonlinear nature. It's commonplace for long-running notebooks to accumulate lots of cruft: dead cells, out-of-date analyses, inaccurate commentary. Moreover, it takes a lot of discipline to put imports and functions at the start of notebooks. They don't play well with source control, so it can be hard to track down what has changed from one version of a notebook to another.

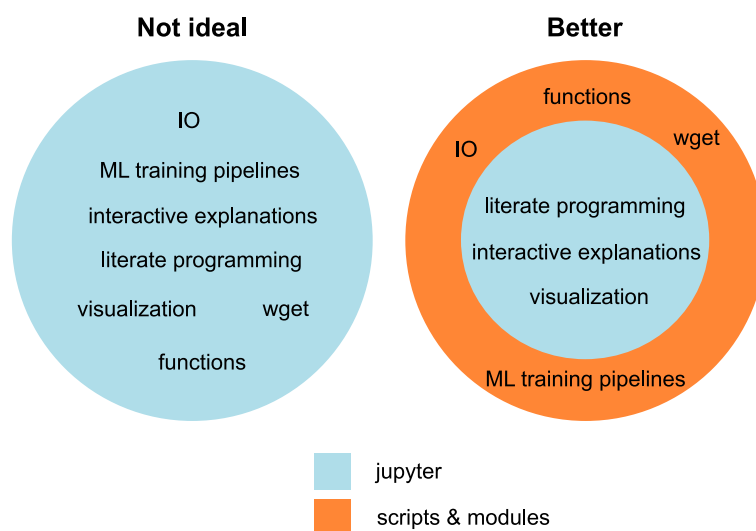


Figure 2.2.: Jupyter notebooks are very good at literate programming – play to their strengths by focusing your jupyter notebooks on mixing explanations, text and graphics.

My [somewhat controversial advice](#) is to keep IO, long-running pipelines, and functions and classes out of jupyter notebooks. Jupyter notebooks excel at literate programming—mixing code, textual explanations, and graphics. If you focus the scope of your jupyter notebooks to literate programming, you'll reduce the amount of cruft that you will need to clean up. As a side benefit, you'll be able to develop more software inside of an IDE—like VSCode or PyCharm—which has a deeper understanding of your code and supports powerful multi-file search. As such, you'll be well on your way to develop testable code.

Tip

When you start a jupyter notebook, write 3-5 bullet points on what you want the analysis to accomplish. You'd be surprised how much this prevents notebooks from getting out of hand.

Make sure your notebooks run top to bottom

Pimentel et al. (2021) found that only about 25% of jupyter notebooks scraped from GitHub ran from top to bottom without error. When you have context, it might only take a couple of minutes to re-order the cells and fix the errors; in several months, you could waste days on this.

Therefore, before you commit a notebook to git, get into the habit of *restarting the kernel and running all*. Often, you will find that the notebook will not run top to bottom; fix the underlying error, and *restart and run all* until your notebook runs again. If it's impractical to restart a notebook because you have a long-running pipeline in a cell, and executing the whole notebook takes a long time, move the relevant code outside the notebook and into a separate script. As a rule of thumb, a jupyter notebook *should run from top to bottom in a minute or less*.

Be productive mixing modules and notebooks

It can be difficult to co-develop a notebook and a module side-by-side, because whenever you change the module you will need to reload the library, often by restarting the kernel. Running these two magics—special commands recognized by jupyter—at the top of your notebook will ensure that the module is automatically reloaded whenever you change it.

```
%load_ext autoreload
%autoreload 2
```

Now, it will feel uncomfortable to move away from jupyter notebooks for some workflows. You might be used to writing small snippets of code and then interact with it immediately to see whether it works: moving the code to a module means you can't use it in this very immediate fashion. We'll fix this discomfort later as we learn about [testing](#).

Refactor comfortably

Refactoring and cleaning up a notebook can be a pain in the jupyter environment: moving a cell across several screens is a pain. [jupyter text can seamlessly translate between a regular jupyter notebook and a markdown-based textual representation](#). In my opinion, refactoring and moving cells around is far easier in the text representation. Checking in the jupyter text representation of a notebook to source control also makes it easy to compare different versions of the same analysis.

Move imports and function definitions to the top of your notebook. Look at Markdown headers and verify that they meaningfully summarize the analysis presented in that section. I find that it's better to write good headings and little long-form text at the start of an analysis, when the analysis still has a lot of room to shift, and to fill in the text later. Read the descriptions and check that they're up-to-date. Delete cells with obsolete analyses from the bottom of notebook: you can always recover them with source control if you've checked in the jupyter text representation.

Discussion

There should be one—and preferably only one—obvious way to do it.
—[The Zen of Python](#)

It's easy to mock style guides as pedantic nitpicking. After all, style, like spelling, is ultimately arbitrary. However, adhering to a standard style can help you preserve your working memory. Don't spend precious mental energy making lots of micro-decisions about variable names and how many spaces to put after a parenthesis: use the style guide. If there's an obvious way of doing things, do it that way.

The short term advantage of using consistent structure compounds over time. Once you've put aside a project for long enough, you will need to reacquaint yourself with it anew, and cruft and dead ends will no longer make sense. Maintaining good code hygiene will make your future self happy.

5-minute exercise

Install `pylint` and run on a script you're currently working on. What did you learn?

3. Write decoupled code

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
—Kernighan’s law

Code which does a little bit of everything at once is hard to work with. Reasoning about a function which reads inputs, does math, calls a remote server and writes outputs in a tightly coupled bundle can easily become overwhelming: there’s too many moving pieces to keep in your head. Here, I show how you can spot tightly coupled code and iteratively improve it.

Code smells and spaghetti code

A code smell is an issue with the source code of a program that indicates there might be some larger underlying issue. For example :

- *Mysterious names*: variables have names which don’t indicate their function
- *Magic numbers*: unique values with unexplained meaning
- *Duplicated code*: large portions of duplicated code with small tweaks
- *Uncontrolled side effects and variable mutations*: code is written so that it’s unclear where and when variables are changed (more on this later)
- *Large functions*: big, unwieldy functions that do a little bit of everything
- *High cyclomatic complexity*: lots of nested ifs and for loops
- *Globals*: Using globals for things that don’t strictly need to be global
- *Embedded configuration*: paths and filenames are hardcoded in ways that the code is not portable to another computer

Note

Cyclomatic complexity is the number of linearly independent paths (e.g. branches of an if statement) in a function

When code has a lot of code smells, it can become brittle to the point of becoming hard or impossible to change. At that point, your productivity goes to zero. Such code is often deemed *spaghetti code*, code so tightly wound that when you pull on one strand, the entire thing unravels.

Note

It is a sad fact that delicious, delicious carbs have such a negative connotation in programming circles.

Development AntiPattern:
Spaghetti Code

spa-ghet-ti code [Slang] an undocumented piece of software source code that cannot be extended or modified without extreme difficulty due to its convoluted structure.

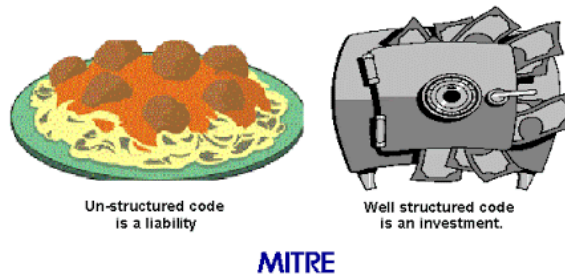


Figure 3.1.: Spaghetti code. From Brown et al. (1998), AntiPatterns. Notice the late 90's clipart. Spaghetti code has been around for a long time

What does spaghetti code look like?

I wanted to find an example of real-life spaghetti code, without being mean-spirited. `wave_clus` is a piece of software that runs in Matlab which is very useful to neuroscientists, and which I've personally used and appreciated. It's used to tease out signals from different neurons. Here's [an excerpt from real code from a function](#):

```
function isi_reject_button_Callback(hObject, eventdata, handles,developer_mode)
set (hObject,'value',1);
b_name = get(gcbo,'Tag');
cn = str2double(regexpi(b_name, '\d+', 'match'));

eval(['set(handles.isi' int2str(cn) '_accept_button','value',0);'])
main_fig = findobj( 0, 'type', 'figure', 'tag', 'wave_clus_figure');
USER_DATA = get(main_fig,'userdata');
classes = USER_DATA{6};
if cn == 3
    if nnz(classes==3)==0
        nlab = imread('filelist_wc.xlj','jpg');
        figure('color','k'); image(nlab); axis off; set(gcf,'NumberTitle','off');
    end
end
end
```

The fact that it's Matlab code can help us to evaluate this code with a little bit of distance: a beginner's mindset, if you will. We see many different code smells:

- Parsing a magic integer (cn) from the name of a component
- Using magic numbers (USER_DATA{6}, classes==3)
- Using globals (USER_DATA)
- Mixing input and output (imread and figure)
- A nested if statement which could be flattened out
- Using eval

The problem with spaghetti code is not that it doesn't work, is that it's *inscrutable* and *brittle*.

Note

Sometimes code is so brittle that it's impossible to modify to run on a modern system. This is why every lab has a machine running ancient software in a backroom somewhere with a sticky note written DO NOT UPGRADE.

Making code better

Now that we've seen real world bad code, let's consider what makes good code.

Separate concerns

Great code exhibits *separation of concerns*:

- a function does one thing
- a module assembles functions which all work towards the same goal
- a class mostly modifies its own members rather than other objects

For instance, your data loading function should just load data, and your computation function should just compute. Each function should be small and should stand on its own. If a function is longer than a screen's worth of code (80 columns, 40 lines), it's usually a good sign it's time to split it off into two. **Make small functions.**

Learn to identify and use pure functions

When novices are introduced to Python functions, they usually start with pure functions. Pure functions follow the *canonical data flow*:

- the inputs come from the arguments
- the outputs are returned with the `return` statement

For instance, this function which adds two numbers together follows the canonical data flow:

```
def add_two_nums(num0, num1):
    return num0 + num1
```

Note

A stateless function doesn't have persistent state that carries over from call to call. A function which uses a global has state.

This function is also *deterministic*, and *stateless*. This is the computational analogue of a mathematical function. You can think of them as unchanging black boxes: you put stuff in, computation happens, you get a result out, and neither the input or the black box gets changed. Because of this, pure functions are easy to reason about: they are the best kind of function. *If something that you write makes sense as a pure function, write it that way.*

Avoid side effects

Some languages enforce using pure functions: they are functional programming languages. Python, however, is a multi-paradigm programming language: you can write functional code, object-oriented code, imperative code, and mix and match as desired. While this flexibility allows us to use the right tool for each job, it offers numerous possibilities to shoot yourself in the foot.

Much Python code uses non-pure functions with *side effects*, which can be hard to reason about. A *side effect* is anything that happens outside the canonical data flow from arguments to return, including:

- modifying a global
- modifying a static local variable
- modifying an argument
- doing IO, including printing to the console, drawing on the screen or calling a remote server

For instance, consider this function which reverses a list:

```
def reversi(arr):  
    """Reverses a list."""  
    for i in range(len(arr) // 2):  
        arr[-i-1], arr[i] = arr[i], arr[-i-1]  
    return arr
```

Note

You can reverse a list directly with `arr[::-1]`, but we don't use this primitive here for the sake of illustration.

This function has a side effect: it modifies its argument `arr`. In Python and many other languages, arguments of complex types like lists, dictionaries and objects are passed by reference, which means they can be modified by the function. That breaks the normal data flow: the arguments are also returns!

In the base Python library, a function which modifies its argument returns `None`. For instance, the function `sort` sorts its input argument, modifies it in place, and returns `None`. This function has a side effect, but it's obvious: if a function returns `None`, yet does useful work, it must have a side effect. When we both modify an argument and return it, we break this convention and confuse the Python reader. We can fix this by returning `None` in our function (good), or returning an entirely new list (better):

Good code

```
def reversi(arr):
    """Reverses a list."""
    for i in range(len(arr) // 2):
        arr[-i-1], arr[i] = arr[i], arr[-i-1]
    return None
```

Better code

```
def reversi(arr):
    """Reverses a list."""
    reved = []
    for i in range(len(arr)):
        reved.append(arr[len(arr)-1-i])
    return reved
```

Functions with side effects can be hard to reason about: you often need to understand their internals and state in order to use them properly. They're also harder to test. **Not every function with side effects is problematic, however.** My pragmatic advice is to first learn to spot and understand pure functions. Then organize your code so that many functions are pure, and those that are not are well-behaved.

For instance:

- Write functions which modify their arguments, or return values, but not both
- Concentrate your IO in their own functions rather than sprinkling them throughout the code
- Use classes to encapsulate state rather than using stateful functions and globals. Python classes use the convention that private variables, which shouldn't be modified from outside, start with an `_`. For example, `self._x` denotes a class member `_x` which should be managed by the class itself.

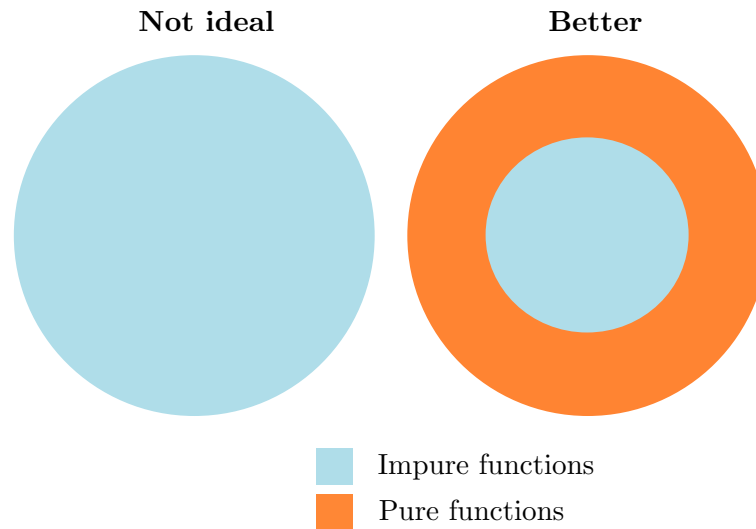


Figure 3.2.: Shrinking the amount of impure functions in your code will make it easier to reason about. Graphic [inspired by CodeRefinery](#).

Make your code more Pythonic

If it ain't broke, fix it till it is.
—[Steve Porter](#)

Sometimes, code smells come from a lack of knowledge about the language. *Reading other people's code*, *pair programming*, and *reading programming books* can help fine tune your knowledge of a programming language and get out of this local minimum.

Move away from Matlab idioms Some common issues are caused by using an idiom from another programming language in Python, where it doesn't translate. Many people using the Python data science ecosystem either come from a Matlab background, or were trained by someone who came from a Matlab background. As a consequence, they'll tend to write Matlab-like code, for example:

Using magic columns numbers to index into a numpy array. Matlab has a matrix type that it uses for many things. You might use the tenth column of a matrix to store a timestamp, which creates hard-to-read code. Dataframes are more appropriate to store parallel data. In 3 months from now, `A.timestamp` will be far clearer than `A[:, 10]`. [You can learn the basics of pandas in a weekend](#), and it's a great time investment.

Using unnamed dimensions in numpy. Similarly, tensors with multiple dimensions can pose problems. If you have a mini-batch of images you're preparing for a deep learning pipeline, did the dimensions go `batch_size x channels x height x width`, or `batch_size x width x height x channels`? [xarray](#) and [named tensors in pytorch](#) give you named dimensions, which will reduce your confusion down the line.

Using bespoke casting for string formatting. Python has a great string formatting method since version 3.6: [the f-string](#). Generating a file name for a checkpoint with `f"{model_name}-{iteration}.pkl"` is more intuitive and readable than `model_name + '_' + str(iteration) + '.pkl'`.

Avoiding for loops. You may have been taught to avoid for loops as much as possible in

Matlab by vectorizing everything. However, this often sacrifices readability. Tricky indexing may look clever, but can be next to impossible to debug. Because Python has a different performance profile than Matlab, some Matlab-specific optimizations won't make your code faster. For instance, unlike Matlab vectors, Python lists are very cheap to grow. Appending to a list in a for loop and then turning that list into a numpy array in a single call has little performance overhead compared to preallocating.

To be clear, modern Matlab doesn't have to be written this way: for instance, Matlab has a capable dataframe class. But a lot of people that come from Matlab still use old Matlab idioms. I've written three more tutorials to ease your transition away from Matlab, available on xcorr.net [1] [2] [3].

Put it all together

Let's introduce a small-scale example that we will make progressively better. We want to write a function which does three things:

- Loads a file
- Counts the words in the files
- Writes the counted words to an output file.

We might code that as:

```
def count_words_in_file(in_file, out_file):
    counts = {}
    with open(in_file, 'r') as f:
        for l in f:
            # Split words on spaces.
            W = l.lower().split(' ')
            for w in W:
                if w != '':
                    if w in counts:
                        counts[w] += 1
                    else:
                        counts[w] = 0

    with open(out_file, 'w') as f:
        for k in counts.keys():
            f.write( k + ", " + str(counts[k]) + "\n")
```

Can you spot the code smells in the previous code?

Spoilers

Here are some code smells in this example:

- Using one-character variable names.
- Using lots of nested for and if statements (6 levels of indent)

- Using bespoke string formatting
- Mixing IO and computation

Split IO and computation

Let's start by splitting IO and computation.

```
def count_words(text):
    # Split words on spaces.
    counts = {}
    W = text.lower().split(' ')
    for w in W:
        if w != '':
            if w in counts:
                counts[w] += 1
            else:
                counts[w] = 0

    return counts

def count_words_in_file(in_file, out_file):
    with open(in_file, 'r') as f:
        counts = count_words(f.read())

    with open(out_file, 'w') as f:
        for k in counts.keys():
            f.write(k + ", " + str(counts[k]) + "\n")
```

`count_words` is now a pure function: it takes in a string and returns a dict, and it has no side effects. This isolation can make it a little easier to notice bugs in the function. Indeed, if we run the code on a few test strings, we notice that this function does not work as expected.

```
>>> count_words("hello world")
{'hello': 0, 'world': 0}
>>> count_words("hello world\n\nhello")
{'hello': 0, 'world\n\nhello': 0}
```

We have an off-by-one error in counts, and we are also not correctly dealing with newline characters. Let's fix this.

```
def count_words(text):
    # Split words on spaces.
    counts = {}
    W = text.lower().replace('\n', ' ').split(' ')
    for w in W:
        if w != '':
            if w in counts:
                counts[w] += 1
```

```

        else:
            counts[w] = 1

    return counts

```

Now:

```

>>> count_words("hello world")
{'hello': 1, 'world': 1}
>>> count_words("hello world\n\nhello")
{'hello': 2, 'world': 1}

```

Not only have we improved the *legibility* of the code, we have improved its *correctness*.

Decrease the amount of nesting

Our code has too many levels of nesting. One underlying cause is that we have two branches for when a key of dictionary exists and when it doesn't. Python has a collection in its standard library specifically built to deal with this: `collections.defaultdict`.

```

import collections

def count_words(text):
    # Split words on spaces.
    counts = collections.defaultdict(int)
    W = text.lower().replace('\n', ' ').split(' ')
    for w in W:
        if w != '':
            counts[w] += 1

    return counts

```

Note

`collections.defaultdict(int)` creates a dict with a default integer value of 0.

The other level of nesting is due to detecting empty words, `w != ''`. We can decrease the nesting of the counting code by skipping to the next iteration upon encountering an empty word:

```

import collections

def count_words(text):
    # Split words on spaces.
    counts = collections.defaultdict(int)
    W = text.lower().replace('\n', ' ').split(' ')
    for w in W:
        if w == '':
            continue
        counts[w] += 1

```

```
return counts
```

However, the root of the problem is that there can be multiple space characters next to each other: this creates empty words. One solution is to replace multiple spaces with one space. With regular expressions, this can be done in one line:

```
import collections
import re

def count_words(text):
    # Split words on spaces.
    counts = collections.defaultdict(int)
    W = re.sub(r"\s+", " ", text.lower()).split(' ')
    for w in W:
        counts[w] += 1

    return counts
```

`re.sub(r"\s+", " ", ...)` finds one or more instances of consecutive whitespace characters (including spaces, newlines and tabs), and replaces them with exactly one space. You could argue that this is less readable than the original version. Regular expressions can certainly be cryptic. However, from a feature perspective, this is an improvement because it deals with tabs and newlines properly.

```
>>> count_words("hello  world\n\n\tworld")
defaultdict(int, {'hello': 1, 'world': 2})
```

Improving code will require us to make many judgement calls like this. Should we prefer features over cleanliness of code? There's no one true solution!

Improving legibility

We can improve legibility further by using descriptive names and using f-strings. The finalized code can be compared with the old code side-by-side.

Improved

```
import collections
import re

def count_words(text):
    """Split words on spaces."""
    counts = collections.defaultdict(int)
    words = re.sub(r"\s+", " ", text.lower()).split(' ')
    for word in words:
        counts[word] += 1
    return counts
```



```
def count_words_in_file(in_file, out_file):
    with open(in_file, 'r') as f:
        counts = count_words(f.read())

    with open(out_file, 'w') as f:
        for word, count in counts.items():
            f.write(f"{word},{count}\n")
```

Original

```
def count_words_in_file(in_file, out_file):
    counts = {}
    with open(in_file, 'r') as f:
        for l in f:
            # Split words on spaces.
            W = l.lower().split(' ')
            for w in W:
                if w != '':
                    if w in counts:
                        counts[w] += 1
                    else:
                        counts[w] = 0

    with open(out_file, 'w') as f:
        for k in counts.keys():
            f.write(k + "," + str(counts[k]) + "\n")
```

It's not perfect, but it's an improvement over the original in terms of legibility, correctness and feature set.

Discussion

Decoupled code is something we all strive towards. Yet, code often has a natural tendency to become more and more tightly wound over time until it becomes an unmanageable mess. The strategies in this chapter will help you to identify code smells and correct them.

Pure functions are easier to reason about, because they're stateless. This preserves your working memory when you're reading code. Code that follows Python's idioms is also easier on your working memory: when you see a line of code that is a familiar pattern, you can see the entire line as one chunk rather than multiple disparate bits. That takes one working memory slot rather than several.

Changing existing code is not without its dangers, however. Perhaps we will make a mistake and introduce a bug in our code! How can we check that our improved code still does the correct thing? We'll cover this in the next chapter on testing.

5-minute exercise

Take a long function in a script you currently are working on, and split it in two. What challenges did you face?

4. Testing your code

Test your code

Most scientists who write software constantly test their code. That is, if you are a scientist writing software, I am sure that you have tried to see how well your code works by running every new function you write, examining the inputs and the outputs of the function, to see if the code runs properly (without error), and to see whether the results make sense. Automated code testing takes this informal practice, makes it formal, and automates it, so that you can make sure that your code does what it is supposed to do, even as you go about making changes around it.

—[Ariel Rokem](#)

Automated testing is one of the most powerful techniques that professional programmers use to make code robust. Having never used testing until I went to industry, it changed the way I write code for the better.

Testing to maintain your sanity

When you run an experiment and the results of the analysis don't make sense, you will go through a process of eliminating one potential cause after the other. You will investigate several hypotheses, including:

- the data is bad
- you're loading the data incorrectly
- your model is incorrectly implemented
- your model is inappropriate for the data
- the statistical test you used is inappropriate for the data distribution

Testing can help you maintain your sanity by decreasing the surface of things that might be wrong with your experiment. Good code yells loudly when something goes wrong. Imagine that you had an experimental setup that alerted you when you had a ground loop, or that would sound off when you use the wrong reagent, or that would text you when it's about to overheat: how many hours or days would you save?

Unit testing by example

Unit testing is the practice of testing a *unit* of code, typically a single function. The easiest way to understand what that means is to illustrate it with a specific example. The Fibonacci sequence is defined as:

$$F(x) \equiv F(x-1) + F(x-2) \quad (4.1)$$

$$F(0) \equiv 0 \quad (4.2)$$

$$F(1) \equiv 1 \quad (4.3)$$

The first few items in the Fibonacci sequence are:

$$F = 0, 1, 1, 2, 3, 5, 8, 13, 21, \dots \quad (4.4)$$

Let's write up a naive implementation of this.

```
def fib(x):
    if x <= 2:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

Let's say that a colleague brings you this code and asks you to check that the code they've written up works. How would check whether this code works?

Spoilers

You could run this code on the command line with different inputs and check that the code works as expected. For instance, you expect that:

```
>>> fib(0)
0
>>> fib(1)
1
>>> fib(2)
1
>>> fib(6)
8
>>> fib(40)
102334155
```

You could also run the code with bad inputs, to check whether the code returns meaningful errors. For example, the sequence is undefined for negative numbers or non-integers.

Informal testing can be done in an interactive computing environment, like the `ipython` REPL or a jupyter notebook. Run the code, check the output, repeat until the code works right—it's a workflow you've probably used as well.

Lightweight formal tests with `assert`

One issue with informal tests is that they often have a short shelf life. Once the code is written and informal testing is over, you don't have a record of that testing. You might even discard the tests you wrote in jupyter! We can make our tests stick with `assert`.

`assert` is a special statement in Python that throws an error whenever the statement is false. For instance,

```
>>> assert 1 == 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Notice that there are no parentheses between `assert` and the statement. `assert` is great for inline tests, for example checking whether the shape of a matrix is as expected after permuting its indices.

We can also assemble multiple `assert` operations to create a lightweight test suite. You can hide your asserts behind an `__name__ == '__main__'` statement, so that they will only run when you directly run a file. Let's write some tests in `fib.py`:

```
def fib(x):
    if x <= 2:
        return 1
    else:
        return fib(x-1) + fib(x-2)

if __name__ == '__main__':
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(6) == 8
    assert fib(40) == 102334155
    print("Tests passed")
```

Now we can run the tests from the command line:

```
$ python fib.py
Traceback (most recent call last):
  File "fib.py", line 8, in <module>
    assert fib(0) == 0
AssertionError
```

We see our test suite fail immediately for `fib(0)`. We can fix up the boundary conditions of the code, and run the code again. We repeat this process until all our tests pass. Let's look at the fixed up code:

```
def fib(x):
    if x == 0:
        return 0
    if x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

if __name__ == '__main__':
```

```

assert fib(0) == 0
assert fib(1) == 1
assert fib(2) == 1
assert fib(6) == 8
assert fib(40) == 102334155
print("Tests passed")

```

While the first few tests pass, the last one hangs for a long time. What's going on here?

Refactoring with confidence with tests

Our `fib(N)` function hangs for a large value of `N` because it spawns a lot of repeated computation. `fib(N)` calls both `fib(N-1)` and `fib(N-2)`. In turn, `fib(N-1)` calls `fib` twice, and so on and so forth. Therefore, the time complexity of this function scales exponentially as $O(2^N)$: it's very slow.

We can re-implement this function so that it keeps a record of previously computed values. One straightforward way of doing this is with a global cache. **We keep our previously implemented tests**, and rewrite the function:

```

cache = {}
def fib(x):
    global cache
    if x in cache:
        return cache[x]
    if x == 0:
        return 0
    elif x == 1:
        return 1
    else:
        val = fib(x-1) + fib(x-2)
        cache[x] = val
        return val

if __name__ == '__main__':
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(6) == 8
    assert fib(40) == 102334155
    print("Tests passed")

```

Running this new and improved script, we see:

```

$ python fib.py
Tests passed

```

Hurray! We can be confident that our code works as expected. What if we want to refactor our code so that it doesn't use globals? Not a problem, we keep the tests around, and we rewrite the code to use an inner function:

```
def fib(x):
    cache = {}
    def fib_inner(x):
        nonlocal cache
        if x in cache:
            return cache[x]
        if x == 0:
            return 0
        elif x == 1:
            return 1
        else:
            val = fib_inner(x-1) + fib_inner(x-2)
            cache[x] = val
            return val
    return fib_inner(x)

if __name__ == '__main__':
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(6) == 8
    assert fib(40) == 102334155
    print("Tests passed")
```

Running the module again, our tests still pass! Testing helps us refactor with confidence because we can immediately tell whether we've introduced new bugs in our code.

Testing pure functions

With pure functions, such as `fib`, we can readily come up with ways to test whether the code works or not. We can check:

- *Correctness for typical inputs*, e.g. $F(5) = 5$
- *Edge cases*, e.g. $F(0) = 0$
- *Errors* with bad input, e.g. $F(-1) \rightarrow \text{error}$
- *Functional goals are achieved*, e.g. that the function works for large numbers

Pure functions don't require elaborate setups to test properly, and indeed they have some of the highest *bang for your buck* when it comes to testing. If in your current workflow, you would have manually checked whether a procedure yielded reasonable results, write a test for it.

Tip

If something caused a bug, write a test for it. 70% of bugs are old bugs that keep reappearing.

Testing with a test suite

Testing with `assert` hidden behind `__name__ == '__main__'` works great for small-scale testing. However, once you have a lot of tests, it starts to make sense to group them into a *test suite*

and run them with a *test runner*. There are two main frameworks to run unit tests in Python, `pytest` and `unittest`. `pytest` is the more popular of the two, so I'll cover that here.

To install `pytest` on your system, first run:

```
pip install -U pytest
```

Writing a test suite for `pytest` is a matter of taking our previous unit tests and putting them in a separate file, wrapping them in functions which start with `test_`. In `tests/test_fib.py`, we write:

```
from src.fib import fib
import pytest

def test_typical():
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(6) == 8
    assert fib(40) == 102334155

def test_edge_case():
    assert fib(0) == 0

def test_raises():
    with pytest.raises(NotImplementedError):
        fib(-1)

    with pytest.raises(NotImplementedError):
        fib(1.5)
```

Notice that `pytest` primarily relies on the `assert` statement to do the heavy lifting. `pytest` also offers extra functionality to deal with special test cases. `pytest.raises` creates a context manager to verify that a function raises an expected exception.

Running the `pytest` utility from the command line, we find:

```
$ pytest test_fib.py
...
    def fib_inner(x):
        nonlocal cache
        if x in cache:
            return cache[x]
>     if x == 0:
E         RecursionError: maximum recursion depth exceeded in comparison

../src/fib.py:7: RecursionError
===== short test summary info =====
FAILED test_fib.py::test_raises-RecursionError: maximum recursion depth exceed
===== 1 failed, 2 passed in 1.18s =====
```

Notice how informative the output of `pytest` is compared to our homegrown test suite. `pytest` informs us that two of our tests passed—`test_typical` and `test_edge_case`—while the last

one failed. Calling our `fib` function with a negative argument or a non-integer argument will make the function call itself recursively with negative numbers - it never stops! Hence, Python eventually will generate a `RecursionError`. However, our tests are expecting a `NotImplementedError` instead! Our test correctly detected that the code has this odd behavior. We can fix it up like so:

```
def fib(x):
    if x % 1 != 0 or x < 0:
        raise NotImplementedError('fib only defined on non -negative integers.')
    cache = {}
    def fib_inner(x):
        nonlocal cache
        if x in cache:
            return cache[x]
        if x == 0:
            return 0
        elif x == 1:
            return 1
        else:
            val = fib_inner(x-1) + fib_inner(x-2)
            cache[x] = val
            return val
    return fib_inner(x)
```

Now we can run tests again.

```
$ pytest test_fib.py
===== test session starts =====
platform linux -- Python 3.8.8, pytest -6.2.4, py -1.10.0, pluggy -0.13.1
rootdir: /home/pmin/Documents/codebook
plugins: anyio -3.1.0
collected 3 items

test_fib.py ... [100%]

===== 3 passed in 0.02s =====
```

They pass!

Testing non-pure functions and classes

I claimed earlier that *pure functions* are the easiest to test. Let's see what we need to do to test non-pure functions. For a *nondeterministic* function, you can usually give the random seed or random variables needed by the function as arguments, turning the nondeterministic function into a deterministic one. For a *stateful* function, we need to additionally test that:

- *Postconditions are met*, that is, the internal state of the function or object is changed in the expected way by the code

Classes are stateful, so we'll need to inspect their state after calling methods on them to make sure they work as expected. For example, consider this `Chronometer` class:

```
import time

class Chronometer:
    def start(self):
        self.t0 = time.time()

    def stop(self):
        return time.time()-self.t0
```

We might want to check that the `t0` variable is indeed set by the `start` method.

For a function with *I/O side effects*, we'll need to do a little extra work to verify that it works. We might need to create mock files to check whether inputs are read properly and outputs are as expected. `io.StringIO` and the `tempfile` module can help you create these mock objects. For instance, suppose we have a function `file_to_upper` that takes in an input and an output filename, and turns every letter into an uppercase:

```
def file_to_upper(in_file, out_file):
    fout = open(out_file, 'w')
    with open(in_file, 'r') as f:
        for line in f:
            fout.write(line.upper())
    fout.close()
```

Writing a test for this is a little tortured:

```
import tempfile
import os

def test_upper():
    in_file = tempfile.NamedTemporaryFile(delete=False, mode='w')
    out_file = tempfile.NamedTemporaryFile(delete=False)
    out_file.close()
    in_file.write("test123\nthetest")
    in_file.close()
    file_to_upper(in_file.name, out_file.name)
    with open(out_file.name, 'r') as f:
        data = f.read()
        assert data == "TEST123\nTHETEST"
    os.unlink(in_file.name)
    os.unlink(out_file.name)
```

With remote calls and persistent storage, testing can rapidly become quite complex.

A hierarchy of tests

We've been focused so far on *unit tests*. However, there are many different kinds of tests that people use.

- *Static tests*: your editor parses and runs your code as you write it to figure out if it will crash

- *Inline asserts*: test whether intermediate computations are as expected
- *Unit tests*: test whether one function or unit of code works as expected
- *Docstring tests*: unit tests embedded in docstrings
- *Integration tests*: test whether multiple functions work correctly together
- *Smoke tests*: test whether a large piece of code crashes at an intermediate stage
- *Regression tests*: tests whether your code is producing the same outputs that it used to in previous versions
- *End-to-end tests*: literally a robot clicking buttons to figure out if your application works as expected

The point is not to overwhelm you with the possibilities, but to give you a glossary of testing so you know what to look for when you're ready to dig deeper.

Write lots of tiny unit tests

My proposal to you is modest:

1. Isolate numeric code.
2. Make numeric functions pure if practical.
3. Write tests for the numeric code
4. Write tests for the critical IO code

You're going to get a lot of bang for your buck by writing unit tests - inline asserts and regression tests are also high payoff-to-effort. Aim for each unit test to run in 1 ms. The faster each test runs, the better for your working memory. More than 5 seconds and you'll be tempted to check your phone.

What do you think is the ideal ratio of test code to real code?

Spoilers

There's no ideal number per say, but 1:1 to 3:1 is a commonly quoted range for library code. For one-off code, you can usually get away with less test coverage. For more down-to-earth applications, 80% test coverage is a common target. [You can use the Coverage.py package to figure out your test coverage.](#)

Now you're playing with power

Testing is the key to refactor with confidence. Let's say that your code looks ugly, and you feel like it's time to refactor.

1. Lock in the current behavior of your code with regression tests

2. Check that the tests pass
3. Rewrite the code to be tidy
4. Correct the code
5. Iterate until tests pass again

You can call `pytest` with a specific filename to run one test suite. For a larger refactor, you can run all the tests in the current directory with:

```
$ pytest .
```

If you want, you can even integrate this workflow into github by running tests every time you push a commit! This is what's called *continuous integration*. It's probably overkill for a small-scale project, but know that it exists.

Discussion

Writing tests is not part of common scientific practice yet, but I think it deserves a higher place in scientific programming education.

Testing allows you to decrease the uncertainty surface of your code. With the right tests, you can convince yourself that parts of your code are *correct*, and that allows you to concentrate your debugging efforts. Keeping that uncertainty out of your head saves your working memory, and debugging will be faster and more efficient. At the same time, code with tests is less stressful to refactor, so you will be able to continuously improve your code so that it doesn't slide towards an unmanageable mess of spaghetti.

Testing is not an all-or-none proposition: you can start writing lightweight inline tests in your code today. Find a commented out `print` statement in your code. Can you figure out how to replace it with an `assert`?

5-minute exercise

Find a commented out `print` statement in your code and transform it into an `assert`.

5. Write good documentation

Document your code

Documentation is a love letter that you write to your future self.
—Damian Conway

When I say documentation, what comes to mind? For many people, *documenting code* is synonymous with *commenting code*. That's a narrow view of documentation. I take a larger view here—documentation is any meta-information that you write *about* the code. In that larger view, all of these are documentation:

- Single-line comments
- Multi-line comments

Note

A unit test is meta-code which tells you the normative behavior of other code. It's a kind of documentation. Woah.

- Unit tests
- Docstrings at the top of functions
- Docstrings at the top of modules
- Makefiles
- Bash files
- [README.md](#)
- Usage documentation
- Tutorial jupyter notebooks on using the code
- Auto-generated documentation hosted on readthedocs
- Websites with tutorials

In this chapter, I'll talk about documenting small units of code—functions and modules. I will cover things that are not conventionally considered documentation, but that nevertheless clarify how to use code. In the next section, I'll discuss documenting larger units of code: programs, projects and pipelines.

Raise errors

Errors should never pass silently.
—The Zen of Python

People don't read manuals. That includes *you*. What people do read are *error messages*. Consider the following function stub, which is meant to convolve two vectors together:

```
def conv(A, B, padding='valid'):
    """
    Convolves the 1d signals A and B.

    Args:
        A (np.array): a 1d numpy array
        B (np.array): a 1d numpy array
        padding (str): padding type (valid, mirror)

    Returns:
        (np.array) The convolution of two vectors.
    """
    pass
```

This is a fine docstring; it tells you how to use the code. Now consider the alternative function:

```
def conv(A, B, padding='valid'):
    assert A.ndim == 1, "A must be one dimensional"
    assert B.ndim == 1, "B must be one dimensional"
    if padding not in ('valid', 'mirror'):
        raise NotImplementedError(
            f"{padding} not implemented.")
    pass
```

This code does not tell you how to use it: it *yells* at you if you use it wrong. The first way relies on your good nature to read the docs; the second way *forces you* to use the code as it was intended. I would argue that the second is better. There are several ways to generate user errors:

- `assert`: When an assert doesn't pass, it raises an `AssertionError`. You can optionally add an error message at the end.
- `NotImplementedError`, `ValueError`, `NameError`: [Commonly used, generic errors](#) you can raise. I probably overuse `NotImplementedError` compared to other types.
- Type hints: Python 3 has type hints, and you can optionally enforce type checking using decorators with [enforce](#) or [pytypes](#). Type checking is a bit controversial because it goes against Python's dynamic nature. It depends on your use case: if you like them, use them.

The unit tests that we discussed last chapter are another mechanism through which you can raise errors: not in the main code path of your code, but in a secondary path that you run through the `pytest` command line.

Write in-line comments

Don't comment bad code—rewrite it.

—Kernighan & Plaugher, cited by Robert Martin in [Clean Code](#)

In-line comments are often used to explain away bad code; you'd be better off rewriting the code rather than to explain the mess. Instead, aim to write code so that it needs few in-line comments. For instance, this code snippet uses meaningless variables names, so we have to explain its function in a comment:

```
# Iterate over lines
for l in L:
    pass
```

Instead, we can clarify the code by using more meaningful variable names:

```
for line in lines:
    pass
```

Before commenting in-line on a piece of code, ask yourself: could I write this in a way that the code is self-explanatory? Then change your code to achieve that. You can then reserve in-line comments to give context that is not readily available in the code itself. There are some essential things you should comment in-line:

- *References to papers* with page numbers and equation numbers (e.g. see Mineault et al. (2011), Appendix equation 2 for definition)
- *Explanations of tricky code*, and why you wrote it the way you did, and why it does the thing that it does
- *TODOs*. Your Python editor recognizes special TODO comments and will highlight them.

```
# TODO(pmin): Implement generalization to non-integer numbers
```

You can similarly highlight code that needs to be improved with `FIXME`.

Write docstrings

Python uses multi-line strings—docstrings—to document individual functions. Docstrings can be read by humans directly. They can be also be read by machines to create HTML documentation, so they're particularly useful if your code is part of a publicly available package. There are three prevalent styles of docstrings:

Note

[Sphinx](#) is the standard way to generate HTML documentation in Python. Sphinx is very powerful: this book is generated by jupyterbook, which uses sphinx to do its job!

- [reST \(reStructuredText\)](#)
- [Google style](#)
- [Numpy style](#)

reST is more prevalent because it's the default in Sphinx, but I think the Google style is easier to read for humans and I prefer it. Here's how you would document a function which counts the number of occurrences of a line:

Google-style

```
def count_line(f, line):
    """
    Counts the number of times a line occurs. Case -sensitive.

    Arguments:
        f (file): the file to scan
        line (str): the line to count

    Returns:
        int: the number of times the line occurs.
    """
    num_instances = 0
    for l in f:
        if l.strip() == line:
            num_instances += 1

    return num_instances
```

Numpy-style

```
def count_line(f, line):
    """
    Counts the number of times a line occurs. Case -sensitive.

    Parameters
    ---
    f: file
        the file to scan
    line: str
        the line to count

    Returns
    ---
    int
        the number of times the line occurs.
```



```

"""
num_instances = 0
for l in f:
    if l.strip() == word:
        num_instances += 1

return num_instances

```

reST

```

def count_word(f, line):
    """
    Counts the number of times a line occurs. Case -sensitive.

    :param f: the file to scan
    :type f: file
    :param line: the line to count
    :type line: str
    :returns: the number of times the line occurs.
    :rtype: int
    """
    num_instances = 0
    for l in f:
        if l.strip() == line:
            num_instances += 1

    return num_instances

```

Docstrings for this function will appear in the REPL and in jupyter notebook when you type `help(count_word)`. They will also be parsed and displayed in IDEs like vscode and PyCharm.

See which style of docstring you prefer and stick to it. Autodocstring, an extension in vscode, can you help you [automatically generate a docstring stub](#). It uses the Google style by default.

Warning

Docstrings can age poorly. When your arguments change, it's easy to forget to change the docstring accordingly. I prefer to wait until later in the development process when function interfaces are stable to start writing docstrings.

Publish docs on Readthedocs

Note

Sphinx can auto-generate docs from Google and Numpy-style docstrings [with a plugin](#).

You know those sweet static docs that you see on readthedocs? You can generate this kind of documentation from docstrings using Sphinx. [There's a great tutorial to get you started](#), but in essence getting basic generated docs is a matter of typing 4 commands:

```
pip install sphinx
cd docs
sphinx-quickstart
make html
```

[Uploading the docs to readthedocs](#) (or Github pages or netlify) is a one-command affair. Docs which focus exclusively on usage (what arguments to use, their types, the returns) are of pretty limited use by themselves. They're powerful when combined with high-level instructions, tutorials and walkthroughs. We'll cover how to write these in the next chapter.

Discussion

Good documentation helps maintain the long-term memory of a project. Very tricky code must be documented with care so that the memory of its intent and implementation is preserved. However, if you have a choice between documenting tricky code and refactoring the code so that it's less tricky, you'll often find that refactoring code pays off over the long term. Similarly, it's often more productive to write unit tests that lock in how the code works than to explain how the code *should* work in words. Document code that needs to be documented, improve the code that can be improved, and develop the wisdom to tell them apart.

5-minute exercise

Write a docstring for a function you've worked on.

6. Document your project

Research code is written in spurts and fits. You will often put code aside for several months and focus your energy on experiments, passing qualifying exams or working on another project. When you come back to your original project, you will be lost. A little prep work by writing docs will help you preserve your knowledge over the long run. In the previous chapter, I showed how to document small units of your code. In this section, I talk about how to document entire projects.

Document pipelines

It's a common practice to use graphical tools (GUIs) to perform analyses. It also happens more often than most people are willing to admit that different variants of a pipeline are run by commenting and un-commenting code. Both of these practices make it hard to reproduce a result 6 months down the line. What was run, and when?

One approach is to textually document in detail what piece of code was run to obtain results. This method can be tedious and error-prone. It's usually worth it to push as much computation as possible into reproducible pipelines which are self-documenting. That way, there's no ambiguity about how results were produced.

Manual steps involving GUI tools should produce results which can be ingested by text-based pipelines. For instance, a interactive GUI to define regions-of-interest (ROI) should export the ROI coordinates in a way that the pipeline can ingest it.

Write console programs

Instead of commenting and un-commenting code, we can have different code paths execute depending on flags passed as command line arguments. Console programs can be written through the `argparse` library, which is part of the Python standard library, or through external libraries like `click`. As a side benefit, these libraries document the intent of the flags and generate help for them.

Let's say you create a command line program `train_net.py` that trains a neural net. This training script has four parameters you could change: model type, number of iterations, input directory, output directory. Rather than changing these variables in the source code, you can pass them as command line arguments. Here's how you can write that:

Note

Real training scripts for neural nets can take dozens of parameters.

```
import argparse
```

```
def main(args):
```

```

# TODO(pmin): Implement a neural net here.
print(args.model) # Prints the model type.

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Train a neural net")

    parser.add_argument("--model", required=True,
                        help="Model type (resnet or alexnet)")
    parser.add_argument("--niter", type=int, default=1000,
                        help="Number of iterations")
    parser.add_argument("--in_dir", required=True,
                        help="Input directory with images")
    parser.add_argument("--out_dir", required=True,
                        help="Output directory with trained model")

    args = parser.parse_args()
    main(args)

```

A nice side benefit of using `argparse` is that it automatically generates help at the command line.

```

$ python train_net.py -h
usage: train_net.py [-h] --model MODEL [--niter NITER] --in_dir IN_DIR --out_dir
OUT_DIR

```

Train a neural net

optional arguments:

```

-h, --help            show this help message and exit
--model MODEL          Model type (resnet or alexnet)
--niter NITER          Number of iterations
--in_dir IN_DIR        Input directory with images
--out_dir OUT_DIR      Output directory with trained model

```

External flags thus allow you to run different versions of the same script in a standardized way.

Commit shell files

Once you've refactored your code to take configuration as command line flags, you should record the flags that you used when you invoke your code. You can do this using a *shell file*. A shell file contains multiple shell commands that are run one after the other.

Consider a long-running pipeline involving `train_net.py`. This pipeline starts with downloading images from the internet stored in an AWS S3 bucket; trains a neural net; then generates plots. We can document this pipeline with a shell file. In `pipeline.sh`, we have:

```

#!/bin/bash

# This will cause bash to stop executing the script if there's an error
set -e

# Download files

```

```
aws s3 cp s3://codebook-testbucket/images/ data/images --recursive

# Train network
python scripts/train_net.py --model resnet --niter 100000 --in_dir data/images \
  --out_dir results/trained_model

# Create output directory
mkdir results/figures/

# Generate plots
python scripts/generate_plots.py --in_dir data/images \
  --out_dir results/figures/ --ckpt results/trained_model/model.ckpt
```

This shell file serves both as runnable code and as documentation for the pipeline. Now we know how our figure was generated! Don't forget to check in this shell file to git to have a record of this file.

Danger

Bash is quirky: the syntax is awkward and it's pretty easy to shoot yourself in the foot. If you're going to write elaborate shell scripts, use [shellcheck](#), which will point out common mistakes in your code. Your favorite editor probably has a plugin for `shellcheck`.

Also, check out [Julia Evans' zine](#) on bash. It's a life saver.

Document pipelines with make

Scientific pipelines often take the shape of DAGs—directed acyclic graphs. This means, essentially, that programs flow from inputs to output in a directed fashion. The `train_net` pipeline above is a DAG with 4 steps. Other DAGs can be more elaborate, for example the 12-step DAG which generates figures shown in Van Vliet (2020) :

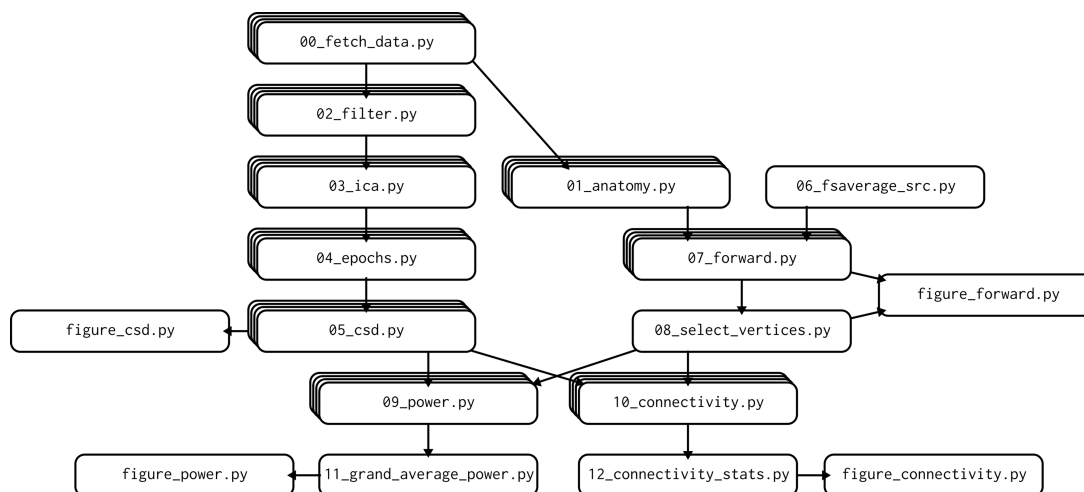


Figure 6.1.: DAG from Van Vliet (2020). CC-BY 4.0

You can document how all the steps of the DAG fit together with a bash file. As your pipelines

grow, it can be painful to restart a pipeline that fails in the middle. This could lead to a proliferation of bash files corresponding to different subparts of your pipeline, and pretty soon you'll have a mess of shell scripts on your hands: basically recreating the commenting/uncommenting workflow we had in Python, this time in bash!

You can use a more specialized build tool to build and document a pipeline. GNU make has been a standard tool for compiling code for several decades, and is becoming more adopted by the data science and research communities. A Makefile specifies both the inputs to each pipeline step and its outputs. To give you a flavor of what a Makefile looks like, this file implements the DAG to train a neural net and plot diagnostic plots:

```
.PHONY: plot
plot: results/trained_model/model.ckpt results/figures
    python scripts/generate_plots.py --in_dir data/images --out_dir \
        results/figures/ --ckpt results/trained_model/model.ckpt

results/trained_model/model.ckpt: data/images
    python scripts/train_net.py --model resnet --niter 100000 \
        --in_dir data/images --out_dir results/trained_model

data/images:
    aws s3 cp s3://codebook-testbucket/images/ data/images --recursive

results/figures:
    mkdir results/figures/
```

Note

make can be pretty finicky. For instance, you must use *tabs*, not spaces, to indent. Use an editor like *vscode* to spot issues early.

The plot can be created with `make plot`. The Makefile contains a complete description of the inputs and outputs to different scripts, and thus serves as a self-documenting artifact. [Software carpentry](#) has an excellent tutorial on make. What's more, make only rebuilds what needs to be rebuilt. In particular, if the network is already trained, make will detect it and won't retrain the network again, skipping ahead to the plotting task.

make uses a domain-specific language (DSL) to define a DAG. It might feel daunting to learn yet another language to document a pipeline. There are numerous alternatives to make that define DAGs in pure Python, including [doit](#). There are also more complex tools that can implement Python DAGs and run them in the cloud, including [luigi](#), [airflow](#), and [dask](#).

Record the provenance of each figure and table

Before you submit a manuscript, you should create a canonical version of the pipeline used to generate the figures and tables in the paper, and re-run the pipeline from scratch. That way, there will be no ambiguity as to the provenance of a figure in the final paper: it was generated by the canonical version of the pipeline.

However, before you get to this final state, it's all too easy to lose track of which result was generated by which version of the pipeline. It is extremely frustrating when your results change

and you can't figure out why. If you check in figures and results to source control as you generate them, in theory you have access to a time machine. However, there's information embedded in figures and tables about the state of the code *when the figures were generated*, only about the state of the code *when the figures were committed*, and there can be a significant lag between the two.

A lightweight workaround is to record the git hash with each result. The git hash is a long string of random digits corresponding to a git commit. You can see these hashes using `git log`:

```
$ git log
commit 3b0c0665465a8ea4cd862058e107b76041acae0f (HEAD -> main, origin/main)
Author: Patrick Mineault <patrick.mineault@gmail.com>
Date:   Wed Sep 8 13:34:31 2021 -0400

    Clean up setup instructions

commit 70deb0e7c9bffe4cdc73d813df8115d99606601c
Author: Patrick Mineault <patrick.mineault@gmail.com>
Date:   Wed Sep 8 00:41:10 2021 -0400
```

Here, `3b0c06...` is the git hash of my latest commit. You can read the current git hash using the `gitpython` library and append it to the name of file. For example, instead of recording `figure.png`, I can record `figure.3b0c066546.png` using:

```
import git
import matplotlib.pyplot as plt

repo = git.Repo(search_parent_directories=True)
short_hash = repo.head.object.hexsha[:10]

# Plotting code goes here...
plt.savefig(f'figure.{short_hash}.png')
```

Now there's no ambiguity about how that figure was generated. You can repeat the same process with csv files and other results.

Note

Most services in this space are closed-source, cloud-based commercial services available for free to researchers. [Wandb](#) and [Neptune](#) record machine learning results. [Gigantum](#) keeps a rich log of jupyter notebook executions. [datalad](#) is libre software that can [record and document dataset manipulations](#).

A more full-featured way of doing this is to use a specialized tool to post results to a centralized database. Most of the offerings in this space are cloud-based commercial services. You post your results—whether scalars, whole tables, figures, etc.—to a centralized server through a bit of python code, and it takes care of versioning. In the screenshot below, I used [wandb.ai](#) to record the outcome of a machine learning pipeline. The record tells me what command that was run, the git hash, meta-information about which computer was used to run the pipeline, as

well as the outcome. There is no ambiguity about provenance.

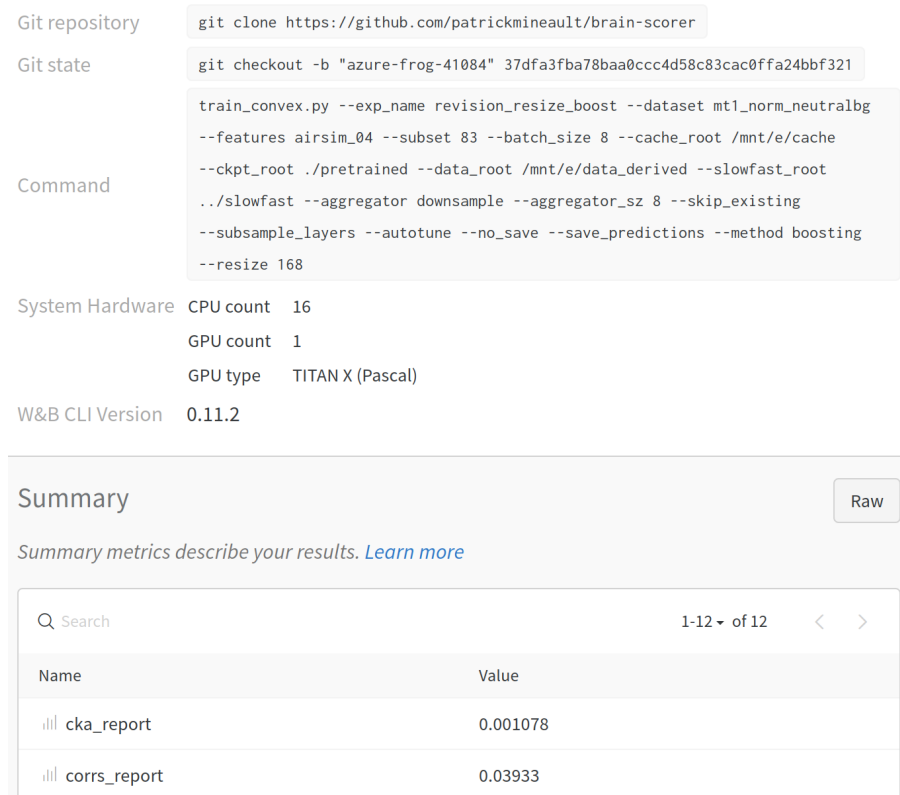


Figure 6.2.: Record of one deep-learning run in wandb.ai

Document projects

In addition to documenting pipelines, it's important to write proper textual documentation for your project. The secret is that once you've written good unit tests and have documented your pipelines, you won't have a lot of text docs to write.

Write a `README.md` file

`README.md` is the often first entry point to your code that you and others will see. This is the file that's rendered when you navigate to your github repository. What are some of the elements in a good `README`?

- A one-sentence description of your project
- A longer description of your project
- Installation instructions
- General orientation to the codebase and usage instructions
- Links to papers

- Links to extended docs
- License

Importantly, keep your `README.md` up-to-date. A good `README.md` file helps strangers understand the value of your code: it's as important as a paper's abstract.

Write Markdown docs

Note

Different environments support slightly different variants of Markdown: Remarkable, CommonMark, pandoc and MyST. This book is written in MyST Markdown.

Markdown has taken over the world of technical writing. Using the same format everywhere creates tremendous opportunities, so I highly recommend that you write your documentation in Markdown. With the same text, you can generate:

- *Digital notes.* [Notion](#), [notable](#), [Obsidian](#), [HackMD](#), GitHub.
- *Blogs.* [Wordpress](#), [Jekyll](#)
- *Wikis.* GitHub.
- *Static sites.* [Jekyll](#), [eleventy](#), GitHub Pages
- *Executable books.* [jupyterbook](#) generates this book.
- *Papers.* [CurveNote](#) uses MyST Markdown under the hood.
- *Slide decks.* [Pandoc](#) via conversion to LaTeX/Beamer.
- *readthedocs-style documentation.* [Sphinx](#) using MyST.

The same Markdown can be deployed in different environments depending on what exactly you want to accomplish. For some projects, the `README.md` file will be all that is needed. Others will want a static site that shows highlights of the paper. Yet other projects will be well-served by blog posts which discuss in longer form the tradeoffs involved in the design decisions.

Creating your documentation in Markdown you make it really easy for you to eventually migrate to another format. I tend to use a combination of all these tools. For instance, I write notes on papers in Notion; I then export those notes to markdown as stubs for my Wordpress blog, for pandoc slides or for jupyterbook.

Discussion

Some things are in our control and others not. [...] If [a thing] concerns anything not in your control, be prepared to say that it is nothing to you.
—Epictetus

Documentation contains the long-term memory of a project. When you're writing documentation, you're ensuring that your code will remain useful for a long time. Code should be self-documenting to a large degree, and you should put effort into automating most of the steps involved in generating figures and tables in your paper. However, you will still need to write some textual documentation.

You can take that occasion to reflect on your project. Sometimes, you'll find that it's more productive to rewrite bad code than to write complex explanations for it. At other times, especially at the very end of a project, refactoring will not be worth the effort, and you will have to let things go. As part of the documentation, you can write about how you could improve on your project. You can use a framing device like *three lessons I learned in this project*. Learn from your mistakes and do better next time.

5-minute exercise

Add a [README.md](#) file to a project you're working on right now

7. Make coding social

Make it social

Maybe the real *good research code* is the friends we made along the way.
—Patrick Mineault

People think that programming is a solitary activity. Engineers, and software engineers in particular, are caricatured as socially inept, basement-dwelling dweebs in popular culture. The reality is that programming is a highly social activity. At a place like Google, for instance, programmers are in constant contact with other programmers, through:

- readability reviews
- code reviews
- design reviews
- pair programming
- reading groups
- retreats
- performance reviews

This is how you get better at programming: by programming with people who are better at programming than you.

Pair program

Pair programming is a very effective method of sharing knowledge through active practice. It's commonly used in industry. Two programmers collaborate actively on a programming task. In the traditional driver-and-navigator style, two programmers are physically co-located. One of them—the *driver* or pilot – physically types the code into the terminal or code editor. They think about micro-issues in the code (tactics): what goes into the body of a `for` loop, the syntax of the code, etc.

The *navigator* or co-pilot assist the driver in telling them what to write. They typically focus on macro issues: what a function should accomplish, how the code is structured, etc. They can also perform other tasks on a second laptop, for example looking up documentation.

Note

I wrote this book in part to become better at the skills I'm now teaching you. Teaching is a great way to learn.

Pair programming forces you to hone your communication skills. Nothing quite reveals your gaps in knowledge than trying to explain to someone what is going on with a piece of code. Sometimes, the people you're pairing with can immediately fill your gap in knowledge; in other cases, you can both learn about a new subject at the same time. Pair programming is especially effective at transmitting knowledge about under-documented systems. You may have explained to a new student in your lab how to perform mysterious experimental procedure **X**. The best way to transmit that knowledge is to have the student attempt to perform the procedure in front of you: active practice enhances learning.

Note

I learned about `Ctrl+Shift+R` (reverse search in bash) through pair programming.

Finally, pair programming can help you learn someone's productivity shortcuts. Seeing somebody work comfortably in an unfamiliar environment is enlightening. I have seen programmers be productive in vim, and it is a sight to behold.

To many, pair programming sounds like a nightmare. It's certainly *uncomfortable*. Things might go too slow or too fast for you, and it can be mentally draining. It's best to do it in short increments (e.g. one hour). In all cases, remember to turn your empathy up to 11, and be excellent to each other.

Although pair programming was traditionally done by physically co-located programmers, many find remote pair programming more comfortable. Screen sharing in Zoom works but can feel intrusive: imagine accidentally showing your inbox or IMs. Instead, you can use an IDE where you can see the other person's cursor. Some environments do this:

- [VSCode Liveshare for in-IDE editing](#)
- [DeepNote](#) and [cocalc](#) for jupyter notebooks
- [Replit](#) for pure Python in the browser

Set up code review in your lab

Reading and critiquing other people's code is a great way to learn. *Code review* is the practice of peer reviewing other people's code. You can use [Github pull requests to give and receive line-by-line feedback on code](#) asynchronously. Alternatively, you can set up a code review circle in your lab. It works like a lab meeting, but instead of doing presentations, you all read code and comment on it at once. Again, it's uncomfortable and awkward, but you can learn a lot this way.

Participate in open source

Maybe your local environment isn't ideal for you to become better at programming. Perhaps you're the only person in your lab that programs. Becoming part of an open source project is a great way to find like-minded people you can learn from.

Joining an open source project doesn't have to be an all or none affair. You can dip your toe in by opening an issue on a software project that you use. Are people responsive? Are they nice? If yes, then you can increase your involvement by starting a pull request to add a feature

to a project. Generally, it's better to open an issue first to tell people about your plans; many larger projects are careful not to introduce new features, because it increases the amount of long-term maintenance people need to do.

Sometimes, you can pick a moribund open source project and maintain it. And if something doesn't exist yet, you can start your own open source project. You will do everything wrong, but you will learn a ton.

Find your community

In addition to projects, there are communities of programmers that support each other. [Hacker spaces](#) promote self-reliance through learning technical skills, which include fabrication, wood-working, sewing and programming. Many meetups exist, focused on Python, data science, deep learning, or more. Some groups are designed as safe spaces for underrepresented people, for example [PyLadies](#).

Finally, you can go on a programmer's retreat to deep dive into a tech, for example through the [Recurse Center](#). This will put you in contact with many other like-minded people who are in different places in their learning journey. Creating that deep web of connections will be invaluable to making you feel connected to the community at large.

Discussion

As we've seen throughout this book, there are many ways you can improve how you write code. However, one of the highest leverage actions you can take to improve your craft is to immerse yourself in a community of practice. Be excellent to each other, learn from each other and give back to the community .

5-minute exercise

Schedule a pair programming session with a lab mate.

8. A sample project: Zipf's law

Let's look at how we can use the suggested organization in a real project. We use the example of calculating Zipf's law for a series of English texts, which was suggested in the book [Research Software Engineering in Python](#), and was released under a CC-BY license. [You can see the completed project on Github](#) at `github.com/patrickmineault/zipf`.

What is Zipf's law, anyway?

Zipf's law comes from quantitative linguistics. It states the most used word in a language is used twice as much as the second most used word, three times as much of the third, etc. It's an empirical law that holds in different languages and texts. There's a lot of interesting theories about why it should hold: [have a look at the Wikipedia article if you're curious](#). A generalized form of Zipf's law states that:

$$p(r) = \frac{1}{Cr^\alpha} \quad (8.1)$$

That is, the frequency of r^{th} most frequent word in a language is a power law with exponent α ; the canonical Zipf's law is obtained when $\alpha = 1$. C is a constant that makes the distribution add up to 1.

Our project

The goal of the project is to measure that Zipf's law holds in three freely available texts. We'll proceed as follows:

1. download some texts from project Gutenberg
2. compute the distribution of words
3. fit Zipf's law to each of them
4. output metrics
5. plot some diagnostics

Let's consider different ways to organize the different subcomponents and define how they should interact with each other. It's clear that the processing has the nice structure of a (linear) directed acyclic graph (DAG).

Approach 1. We could make each box a separate script with command line arguments. Each script would be backed with module code that would help in a common library. Then, we would glue these scripts together with a bash file, or perhaps with make. This approach has a lot of merit, and in fact was the one that was originally suggested by the book: it's clean and it's standardized. One inconvenience is that it involves writing a lot of repetitive code in order to



Figure 8.1.: The Zipf project can be described as a linear directed acyclic graph (DAG)

define command line interfaces. There's also a bit of overhead in writing both a script and module code for each subcomponent.

Note

The amount of cruft for command line interfaces could be reduced significantly using the [click](#) library.

Approach 2. We could make each box a separate function, held in a module. Then, we would glue these functions together with a Python script. This Python script would have its own command line arguments, which we could set in a bash script. One merit of the approach is that it has less overhead—fewer files and cruft—than the first approach.

The tradeoff between these two approaches lies in the balance between generality and complexity. The first is a bit more flexible than the second. We can't run a single component of our analysis separately from the command line: we'll need to implement tests instead, which feel a little less intuitive. If we wanted to run an analysis of tens of thousands of books, parallelizing would also be easier with the first approach (e.g. [with make](#)).

Note

Different people—and sometimes the same person at different points in time—can disagree on the very best approach for a particular problem. In the end, what matters more is that the process through which the code was deliberate. If you put some thought into the organization: you're 90% of the way there.

For this example, however, I have a slight preference for the second approach, so that's the one we will implement. Keeping the number of command line tools to create to one means we'll worry less about cruft and more about the computations. Let's take a look at the DAG again to see how we'll split the job:

We'll create one module to compute the distribution of words, and another to fit Zipf's law. We will create tests for each of those two modules. We'll wrap the two modules as well as glue code inside a command line tool. Plotting will take place in jupyter notebook, which makes it easy to change plots interactively.

I encourage you to also have a look at the Python RSE github repo for an alternative implementation to examine the tradeoffs in different implementations.

Note

Most of the code is taken verbatim from the [Py-RSE repo](#). Our emphasis here is on organizing project files.

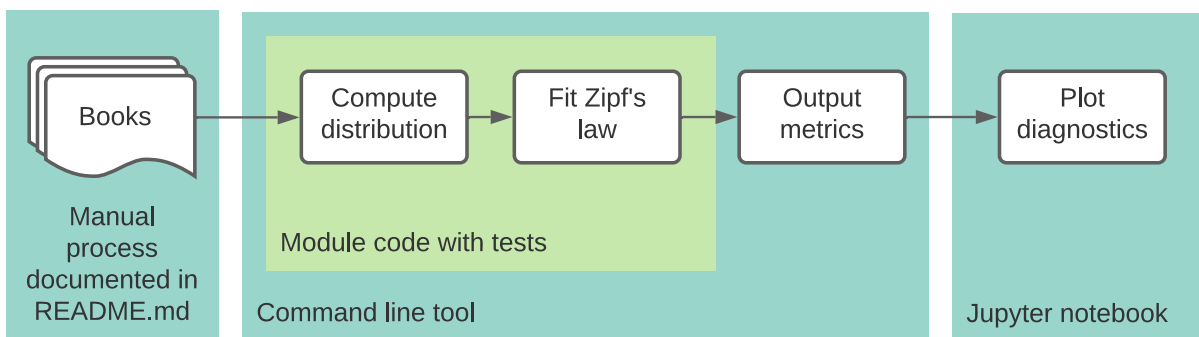


Figure 8.2.: The Zipf project split into different sub-components.

Setup

Let's proceed with setting up the project. `zipf` is a reasonable project name, so let's go ahead and create a new project with that name:

```
(base) ~/Documents $ cookiecutter gh:patrickmineault/true-neutral-cookiecutter
```

Use `zipf`, `zipf`, and `zipf` as the answers to the first 3 questions, and "Zipf's law project" as the description. Then proceed to create an environment and save it:

```
cd zipf
conda create --name zipf python=3.8
conda activate zipf
conda env export > environment.yml
```

Then we can sync to a Github remote. My favorite way of doing this is to use the GUI in `vscode`, which saves me from going to Github to create the remote *and* then type on the command line to sync locally. We can fire up `vscode` using:

```
code .
```

Then, in the git panel, we hit "Publish to Github" to locally set up git and create the Github remote in one shot.

Now we have a good looking project skeleton! We can set up `black` in `vscode` so that whenever a file is saved, it is formatted in a standard way. Time to add some code to it.

Note

If you prefer, you can instead go through github.com to create a new repo and follow the command line instructions there.

Download the texts

We want to download three texts and put them in the data folder. Ideally, we'd do this automatically, for example with a bash file with calls to `wget`. However, the terms and conditions from Project Gutenberg state:

The website is intended for human users only. Any perceived use of automated tools to access the Project Gutenberg website will result in a temporary or permanent block of your IP address.

—[Project Gutenberg](#)

Caution

Document manual steps in [README.md](#)!

Not a big deal! We can download the files manually and document the source in the `README.md` file. Let's download the following files manually and put them in the data folder:

- [Dracula](#) → `data/dracula.txt`
- [Frankenstein](#) → `data/frankenstein.txt`
- [Jane Eyre](#) → `data/jane_eyre.txt`

Count the words

The next step is to calculate word counts for each text. We will create a module called `parse_text` in the `zipf` folder. It will contain a function `count_words` that takes in a text string and outputs a set of counts. This function will itself call a helper function `_clean_gutenberg_text`. Note that the function starts with an underscore to indicate that it is meant to be a private function used internally by the module.

`_clean_gutenberg_text` will explicitly filter out boilerplate from project Gutenberg texts. There's a significant amount of boilerplate at the start of e-books and license information at the end, which might skew the word count distribution. Thankfully, there are phrases in the e-books which delimit the main text, which we can detect:

- START OF THE PROJECT GUTENBERG EBOOK
- END OF THE PROJECT GUTENBERG EBOOK

Note

[Ensuring data quality is huge chunk of the workload of data scientists.](#)

Because we only have three books in our collection, we can manually test that each book is processed correctly. However, if we ever process a fourth, unusually formatted text, and our detection didn't work, we want our pipeline to fail in a graceful way. Thus, we add in-line `assert` statements to make sure our filter finds the two delimiters in reasonable locations in the text.

```
def _clean_gutenberg_text(text):
    """
    Find fences in a Gutenberg text and select the text between them.
    """
    start_fence = "start of the project gutenberg ebook"
    end_fence = "end of the project gutenberg ebook"
    text = text.lower()
    start_pos = text.find(start_fence) + len(start_fence) + 1
    end_pos = text.find(end_fence)

    # Check that the fences are at reasonable positions within the text.
    assert 0.000001 < start_pos / len(text) <= 0.1
    assert 0.9 < end_pos / len(text) <= 1.0

    return text[start_pos:end_pos]
```

Now we can use call this function in another wrapper function that counts words like so:

```
import string
import collections

def count_words(f, clean_text=False):
    """
    Count words in a file.

    Arguments:
        f: an open file handle
        clean_text (optional): a Boolean, if true, filters out boilerplate
            typical of a Gutenberg book.

    Returns:
        A dict keyed by word, with word counts
    """
    text = f.read()
    if clean_text:
        text = _clean_gutenberg_text(text)

    chunks = text.split()
    npunc = [word.strip(string.punctuation) for word in chunks]
    word_list = [word.lower() for word in npunc if word]
    word_counts = collections.Counter(word_list)
    return dict(word_counts)
```

To test that the `count_words` function works as intended, we make a sample txt file in the `tests` folder which contains dummy data:

```
The

*** START OF THE PROJECT GUTENBERG EBOOK ***
OF
TO to
I i i
```

```
and and and and
THE The      THE the thE
[...]
*** END OF THE PROJECT GUTENBERG EBOOK ***
```

OF

If our function works correctly, it should ignore all the text outside of the start and end fences. It should return:

```
{'the': 5, 'and': 4, 'i': 3, 'to': 2, 'of': 1}
```

We set up a test function in `test_parse_text.py` that loads this test text and verifies that the word counts are correctly measured. The test can be run with:

```
$ pytest tests/test_parse_text.py
```

The resulting module and `test` can be viewed on Github.

Calculate Zipf's law

Once we have word counts, we can calculate Zipf's law based on word counts. We use the method proposed in [Moreno-Sanchez et al. \(2016\)](#), which is implemented in the book *Research Software Engineering with Python*. The method calculates the maximum likelihood α parameter for the empirical distribution of ranks. It starts with $\alpha = 1$ and refines the estimate through gradient descent. We implement this method in `fit_distribution.py`.

It's quite easy to make a mistake in calculating the exponent by incorrectly implementing the error function. To protect ourselves against this, in our test suite, we generate data from a known distribution with $\alpha = 1$ and verify that the correct exponent is estimated. We put this test, `test_fit_distribution.py` under the tests folder.

Write the command line interface

We are now ready to write glue code to run our pipeline from the command line. We create a script, `run_analysis.py`, under the scripts folder. Our script takes in two different command line arguments, or flags:

- `in_folder`: the input folder containing txt files.
- `out_folder`: where we will output the results

Note

Keep your command line programs to less than 20 flags. Beyond that, they become a pain to maintain.

We create this command line interface using `argparse`. This looks like this:

```

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Compute zipf distribution")
    parser.add_argument("--in_folder", help="the input folder")
    parser.add_argument("--out_folder", help="the output folder")
    args = parser.parse_args()

    main(args)

```

The main function first verifies that there are files to be processed and creates the output folder if necessary. To manipulate paths, it uses the `pathlib` module. The function then runs each of the stages of the pipeline in turn until completion. We don't write tests this part of the code, as we've lifted the error-prone aspects of the code elsewhere. Again, the goal is rarely to get 100% coverage, but rather to have good coverage where it counts.

The code uses a for loop to construct a table of results, which is then loaded into pandas, and then dumped to disk as csv. Pandas is not strictly necessary to write a csv file, but it's the default choice to work with tabular data in Python, so it is used here. [Have a look at the script here.](#)

Make summary figures

Finally, to generate figures, we prefer the jupyter notebook environment. Polishing figures takes some trial and error, and the jupyter notebook environment makes it easy to iterate on figures. The notebook, in `scripts/visualize_results.ipynb`, takes in results files as csvs, and generates figures and summary tables from them.

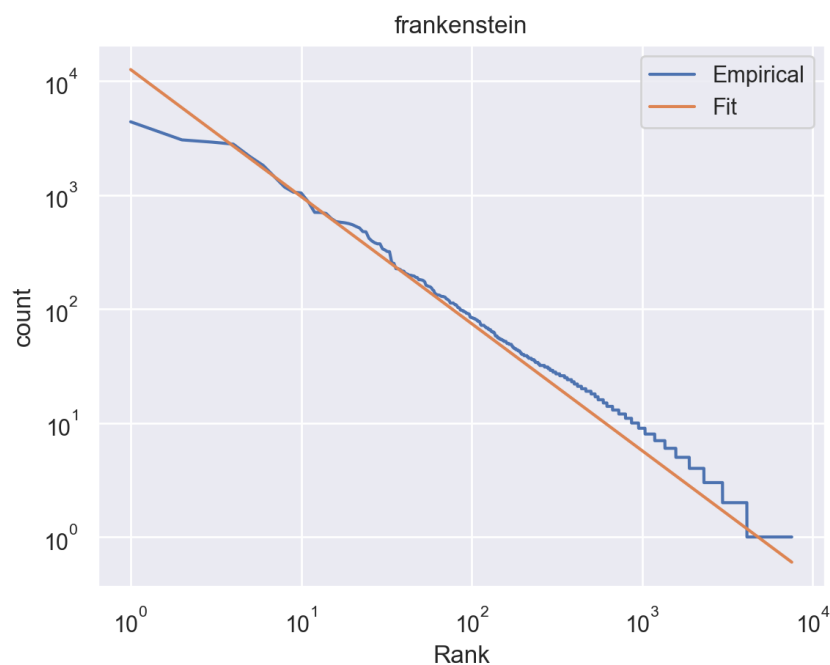


Figure 8.3.: A plot generated by our pipeline. Zipf distribution estimated on the novel Frankenstein by Mary Shelley.

We used the standard choice of matplotlib for the figures, although with the stylesheet from seaborn, which is more aesthetically pleasing. Note, however, that there are a number of excellent libraries for interactive graphics, so if this was critical, we could certainly apply this here. There is no computation in the notebook, only figure and summary table generation, which keeps the code in the notebook to a minimum.

What did we learn?

We saw how to organize an analysis according to the organization proposed in this handbook. We created a new project folder and module with the true neutral cookiecutter. If this analysis were part of a larger project—a paper or book chapter—we would re-use the same project folder, and add new pipelines and analyses to the project.

When I started this project, I was surprised by how many macro- and micro-decisions were needed to be made about how a pipeline works and how it's organized. A little bit of thinking ahead can avoid days of headaches later on. In this case, we chose a lightweight structure with one entry script calling module functions. These module functions are short and most of them are pure functions. Pure functions are easier to test. Indeed, we wrote unit tests to check that these functions worked as intended. That way, once the code was written and tests passed, it was crystal clear that the code worked as intended. We separated module code from glue code and plotting code. The resulting code is decoupled and easily maintainable.

Writing code in this organized way is effortful. Over the long term, this methodical approach is more efficient, and more importantly, it's less stressful.

5-minute exercise

How would you change this pipeline so that it computes error bars for the parameters of the Zipf distribution through bootstrapping? Would you need to change existing functions? What function would you need to add? *If you're feeling adventurous, go ahead and implement it! It's definitely more than a 5-minute project!*

Part III.

Extras

How to test numerical code: CKA

Test numerical code: CKA

Let's look at an extended example of testing numerical code. This example implements a computational method called CKA which was introduced [in this paper](#). Importantly, CKA is not already implemented in scipy or sci-kit learn or in any other pip installable package: we're flying solo .

Does this make you nervous? It should! It's easy to make a mistake in a computational pipeline and get the wrong results. With the structured approach that I've introduced in this handbook, you can work with far more confidence.

Background

We're going to implement centered kernel alignment (CKA) and apply it on test data. Because I wanted the method not to be implemented already in a major Python package, it had to be pretty obscure. I don't expect anyone reading this to have already heard of CKA, so a quick intro is in order. In brief, *CKA is a way to compare two matrices, in the same way that Pearson's correlation can compare two vectors*. It has applications to studying the brain and neural nets.

Comparing deep neural nets and brains

Deep artificial neural nets perform fabulous feats, whether it's detecting objects in images or translating speech to text. Neuroscientists often wonder whether or not these neural nets do their work in ways similar to the brain. To answer this question, they rely on methods that follow the same core recipe:

1. pick a battery of stimuli
2. measure the response of a brain to the battery of stimuli (in a MRI scanner, let's say)
3. measure the response of a deep neural net to the battery of stimuli.
4. Compare the two sets of responses

If the two sets of responses are similar, that means the brain and the deep neural net are aligned in some sense. We collect the responses into two matrices, X and Y which are of size $N \times K$ and $N \times L$, respectively; N is the number of stimuli. Then the game, in the final step, is to compare the two matrices X and Y . CKA is a metric introduced in Kornblith et al. (2019) that can be used to compare two matrices which has some nice properties.

Definition

CKA is defined as:

$$CKA(\mathbf{X}, \mathbf{Y}) = \frac{\|\mathbf{X}^T \mathbf{Y}\|_2^2}{\|\mathbf{X}^T \mathbf{X}\|_2 \|\mathbf{Y}^T \mathbf{Y}\|_2} \quad (1)$$

The columns X and Y are centered. $\|\mathbf{Z}\|_2 \equiv \sqrt{\sum_{ij} Z_{ij}^2}$ is the Frobenius norm, the root of the sum of squares of the entries.

When the CKA is 0, the two representations are maximally different; when it is 1, the two representations are maximally similar. You might notice that the formula resembles Pearson's correlation:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (2)$$

In fact, CKA is the square of the Pearson's correlation when X and Y are vectors. You can thus think of the CKA as a generalization of Pearson's correlation for matrices.

Initial implementation

Based off of the definition above, we can implement a working version of CKA with the following code. In `cka_step1.py`:

```
import numpy as np

def cka(X, Y):
    # Implements linear CKA as in Kornblith et al. (2019)

    # Center X and Y
    X -= X.mean(axis=0)
    Y -= Y.mean(axis=0)

    # Calculate CKA
    XTX = X.T.dot(X)
    YTY = Y.T.dot(Y)
    YTX = Y.T.dot(X)

    return (YTX ** 2).sum() / np.sqrt((XTX ** 2).sum() * (YTY ** 2).sum())
```

Now, is this function *correct*? In these ten lines of code, there's a lot of trickiness going on:

- This function centers the columns of X ... or does it? Should it remove `X.mean(axis=1)` instead?
- Is this function pure? Does it change its arguments X and Y ?
- In the last line: is this the correct definition of the Frobenius norm? Or are we off by a squaring factor?

Indeed, a lot can go wrong in implementing this short function. Let's write down some tests to reassure ourselves that this function does what it needs to do.

Writing our first test

The first test is the identity test: the CKA of a matrix with itself is 1, just like with Pearson's correlation. Let's write the identity test as part of a test suite.

Let's code CKA tests. We will turn properties of CKA listed in the paper into tests. In `cka_step1.py`, we write:

```
from cka import cka
import numpy as np

def test_identity():
    # Create a random matrix, check it is perfectly correlated with itself.
    X = np.random.randn(100, 2)
    assert cka(X, X) == 1.0
```

Great! Now we can run our test suite with pytest:

```
(cb) ~/Documents/codebook_examples/cka$ pytest .
===== test session starts =====
platform linux -- Python 3.8.11, pytest -6.2.5, py -1.10.0, pluggy -1.0.0
rootdir: /home/pmin/Documents/codebook_examples/cka
plugins: anyio -3.3.0
collected 1 item

test_cka.py F [100%]

===== FAILURES =====
_____ test_identity _____

    def test_identity():
        # Create a random matrix, check it is perfectly correlated with itself.
        X = np.random.randn(100, 2)
>       assert cka(X, X) == 1.0
E       assert 0.9999999999999994 == 1.0
```

Here we've run into one of the tricky bits about writing numerical code: numerical instability. 1.0 is very close to 0.9999999999999994, but it's not exactly equal. We can replace our test with a more lenient one. Numpy's `np.testing.assert_allclose` can test that two arrays are close enough to each other entry-wise:

```
def test_identity_lenient():
    # Create a random matrix, check it is perfectly correlated with itself.
    X = np.random.randn(100, 2)
    np.testing.assert_allclose(cka_start(X, X), 1.0)
```

And now we find the tests pass. Let's add one more test to the mix: a matrix and itself are perfectly correlated, *regardless of the order of their columns*. We can make a new test for that.

```
def test_column_swaps():
    # A matrix is perfectly correlated with itself even with column swaps.
```

```
X = np.random.randn(100, 2)
c = cka_start(X[:, [0, 1]], X[:, [1, 0]])
np.testing.assert_allclose(c, 1.0)
```

And now the tests pass:

```
(cb) ~/Documents/codebook_examples/cka$ pytest .
===== test session starts =====
platform linux -- Python 3.8.11, pytest -6.2.5, py -1.10.0, pluggy -1.0.0
rootdir: /home/pmin/Documents/codebook_examples/cka
plugins: anyio -3.3.0
collected 2 items

test_cka_step1.py ..                                     [100%]
```

Checking centering

Now let's add another test, to verify that our CKA implementation if centering correctly. It shouldn't matter how columns are centered, so we can add an offset and verify that we obtain the same result:

```
def test_centering():
    # Check that a matrix is perfectly correlated with itself even with adding
    # column offsets
    X = np.random.randn(100, 2)
    Xp = X.copy()
    Xp[:, 1] += 1.0

    c = cka(X, Xp)
    np.testing.assert_allclose(c, 1.0)
```

Run it in pytest—it works! That means we did the centering correctly. Indeed, we correctly removed `X.mean(axis=0)` from `X` and `Y.mean(axis=0)` from `Y`. But wait a minute—when we center in our function, do we change the original matrix? We can add a test to check that:

```
(cb) ~/Documents/codebook_examples/cka$ pytest .
===== test session starts =====
platform linux -- Python 3.8.11, pytest -6.2.5, py -1.10.0, pluggy -1.0.0
rootdir: /home/pmin/Documents/codebook_examples/cka
plugins: anyio -3.3.0
collected 4 items

test_cka_step1.py ...F                                     [100%]

===== FAILURES =====
_____ test_pure _____
```

```
def test_pure():
    # Check that a function doesn't change the original matrices
    X = np.random.randn(100, 2)
```

```

Xp = X.copy()
Xp[:, 1] += 1.0

Xp_original = Xp.copy()
c = cka(X, Xp)
> np.testing.assert_allclose(Xp_original[:, 1], Xp[:, 1])
E      AssertionError:
E      Not equal to tolerance rtol=1e-07, atol=0

```

We see that this function modifies its argument. If you scroll back up to the `cka` definition, you can see that we used the `--` in-place assignment operator. This caused the original matrix to change. If this tripped you up: don't worry! I was very confused by this as well. This line changes the original array:

```
X -= X.mean(axis=0)
```

But this line returns a copy of the matrix:

```
X = X-X.mean(axis=0)
```

Who knew! This kind of subtle semantic difference can really trip you up. We can clarify the intent of the code using `copy()` to indicate that we don't want to change the original array: this way, the function's intent is very clear. In `cka_step2.py`, we write the function a different way:

```

import numpy as np

def cka(X, Y):
    # Implements linear CKA as in Kornblith et al. (2019)
    X = X.copy()
    Y = Y.copy()

    # Center X and Y
    X -= X.mean(axis=0)
    Y -= Y.mean(axis=0)

    # Calculate CKA
    XTX = X.T.dot(X)
    YTY = Y.T.dot(Y)
    YTX = Y.T.dot(X)

    return (YTX ** 2).sum() / np.sqrt((XTX ** 2).sum() * (YTY ** 2).sum())

```

Now we copy our old tests into `test_cka_step2.py`, and the issue is fixed:

```

(cb) ~/Documents/codebook_examples/cka$ pytest test_cka_step2.py
===== test session starts =====
platform linux -- Python 3.8.11, pytest -6.2.5, py -1.10.0, pluggy -1.0.0
rootdir: /home/pmin/Documents/codebook_examples/cka
plugins: anyio -3.3.0

```

collected 4 items

test_cka_step2.py

[100%]

More properties

CKA has several more properties which we can test. Many of these are listed in the CKA paper:

- the CKA is the square of the Pearson's correlation when X and Y are vectors
- the CKA is insensitive to rotations
- the CKA is insensitive to scaling the entire matrix
- the CKA is sensitive to scaling different columns differently

Here, it becomes useful to create a few helper functions to generate sample signals: matrices made of sinusoids of different frequencies in each column. We'll remove our reliance on random data: it's generally good practice to have *deterministic* tests. Non-deterministic tests that sometimes work and sometimes don't are *flaky*, and they can be a pain. Let's put it all together:

```
from cka_step2 import cka
import numpy as np
import pytest

def _get_one():
    X = np.cos(.1 * np.pi * np.arange(10)).reshape((-1, 1))
    Y = np.cos(2 + .07 * np.pi * np.arange(10)).reshape((-1, 1))
    return X, Y

def _get_multi():
    X = (np.cos(.1 * np.pi * np.arange(10).reshape((-1, 1)) *
          np.linspace(.5, 1.5, num=3).reshape((1, -1))))
    Y = (np.cos(.5 + .07 * np.pi * np.arange(10).reshape((-1, 1)) *
          np.linspace(.7, 1.3, num=4).reshape((1, -1))))
    return X, Y

def test_identity_lenient():
    """Create a random matrix, check it is perfectly correlated with itself."""
    X, _ = _get_multi()
    np.testing.assert_allclose(cka(X, X), 1.0)

def test_column_swaps():
    """A matrix is perfectly correlated with itself even with column swaps."""
    X, _ = _get_multi()
    c = cka(X[:, [0, 1]], X[:, [1, 0]])
    np.testing.assert_allclose(c, 1.0)

def test_centering():
    """A matrix is perfectly correlated with itself with column offsets."""
    X, _ = _get_multi()
    Xp = X.copy()
    Xp[:, 1] += 1.0
```

```

c = cka(X, Xp)
np.testing.assert_allclose(c, 1.0)

def test_pure():
    """Check that a function doesn't change the original matrices."""
    X, _ = _get_multi()
    Xp = X.copy()
    Xp[:, 1] += 1.0

    Xp_original = Xp.copy()
    c = cka(X, Xp)
    np.testing.assert_allclose(Xp_original[:, 1], Xp[:, 1])

def test_corr():
    """The CKA of two vectors is the square of the correlation coefficient"""
    X, Y = _get_multi()
    c1 = cka(X, Y)
    c2 = np.corrcoef(X.squeeze(), Y.squeeze())[0, 1] ** 2
    np.testing.assert_allclose(c1, c2)

def test_isoscaling():
    """CKA is insensitive to scaling by a scalar"""
    X, Y = _get_multi()
    c1 = cka(X, Y)
    c2 = cka(2.0 * X, -1 * Y)
    np.testing.assert_allclose(c1, c2)

def test_rotation():
    """CKA is insensitive to rotations"""
    X, Y = _get_multi()
    X0 = X[:, :2]
    X0p = X0 @ np.array([[1, -1], [1, 1]]) / np.sqrt(2)
    c1 = cka(X0, Y)
    c2 = cka(X0p, Y)
    np.testing.assert_allclose(c1, c2)

def test_no_iso():
    """CKA is sensitive to column scaling"""
    X, Y = _get_multi()
    X0 = X[:, :2]
    X0p = X0 @ np.array([[1, 1], [10, 1]])
    c1 = cka(X0, Y)
    c2 = cka(X0p, Y)
    assert abs(c1-c2) > .001

```

It's starting to get quite long! For numeric code, it's not unusual that the test code should be several times longer than the code it is testing. Indeed, when you introduce a new numerical method, you might spend days testing it on different inputs to check that it gives reasonable outputs. Testing takes this common practice and formalizes it. Now, you can rest assured that the code works as intended.

Dealing with wide matrices

Before we add more features to the code, it's important to make sure that what is already there is correct. It's all too easy to build in a vacuum, and we are left debugging a giant chunk of code.

In our case, a nice feature we might want is the ability to deal with wide matrices. The implementation we have works well for tall, skinny matrices. However, neural nets are generally over-parametrized and frequently have big intermediate representations. The paper introduces another method to compute the CKA with these wide matrices that is far more memory-efficient. We can change our implementation to deal with these larger matrices efficiently—and of course, add more tests to make sure we didn't mess up anything! Tests are what allow us to move with confidence. [Take a look at the final version of the code](#) to see how we can test that the code works as expected.

Final thoughts

We've gone through a complex example of testing numerical code. We built infrastructure to test the code, found a gnarly bug, corrected it, and continued to build a large test suite. We were then able to expand our code to deal with new conditions. In the end, we could be confident that our code is correct.

Use good tools

Use the tools introduced in each section

For your convenience, here is a list of tools and packages discussed in each section of the book.

Set up your project

git a command line tool for code versioning

github a website where you can share code

conda a command line package manager and virtual environment manager

setuptools a Python library to define pip installable packages

cookiecutter a command line tool to create projects from templates

Keep things tidy

flake8 and **pylint** command line linters for Python

black a command line auto-formatter for Python code, with plugins for popular IDEs

vulture a Python package to identify dead code in your codebase

jupyter a command line tool to seamlessly translate between regular jupyter notebooks and a markdown-based representation

Write decoupled code

pandas a Python library to represent columnar data

xarray a Python library to represent multidimensional tensors with named dimensions

collections a Python standard library of containers, including defaultdict, counter, etc.

Test your code

pytest a Python library for unit testing, along with a command line utilities

Write good docs

argparse a Python library to parse command-line arguments, part of the Python standard library

shellcheck a command-line tool that checks for common errors in bash scripts, with plugins for popular IDEs

make a command-line tool to define and run directed acyclic graphs of computation

sphinx a command-line tool to generate documentation from Python code and text files

readthedocs a website to host documentation

Make it social

vscode live share IDE extension for code sharing

deeptime and **cocalc** collaborative jupyter notebooks in the browser

replit collaborative IDE in the browser

Choose an IDE

Integrated development environments (IDE) can help you develop faster and make it easy to implement some of the productivity tips I've discussed previously. Preferred IDEs change from year to year, as new editors become favored while others are shunned. Don't be surprised if in three years you'll be using a different IDE.

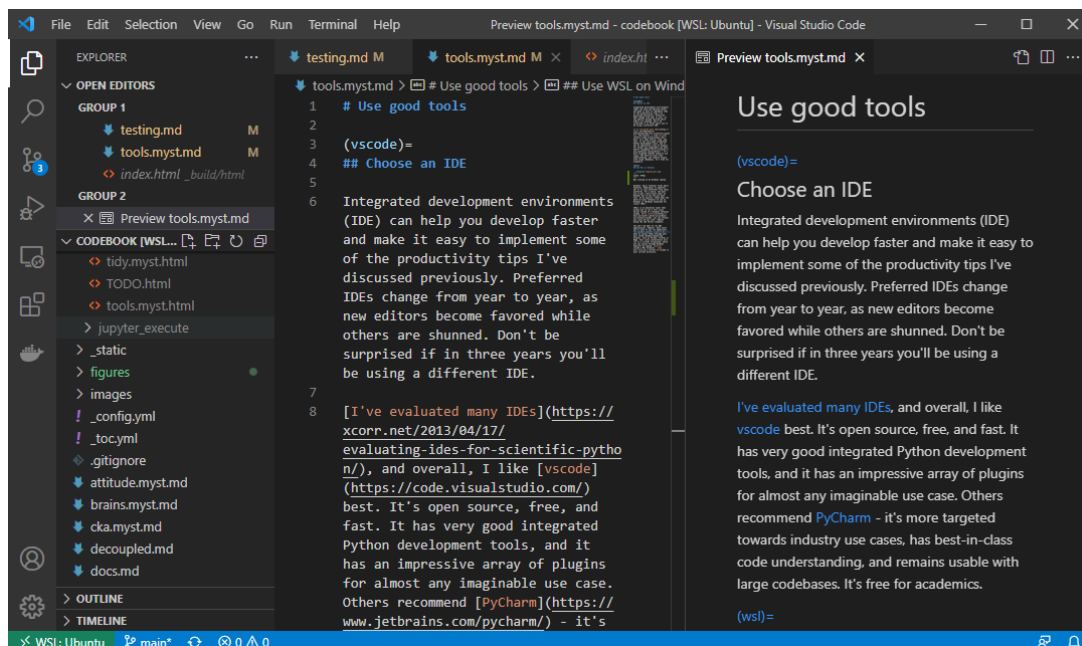
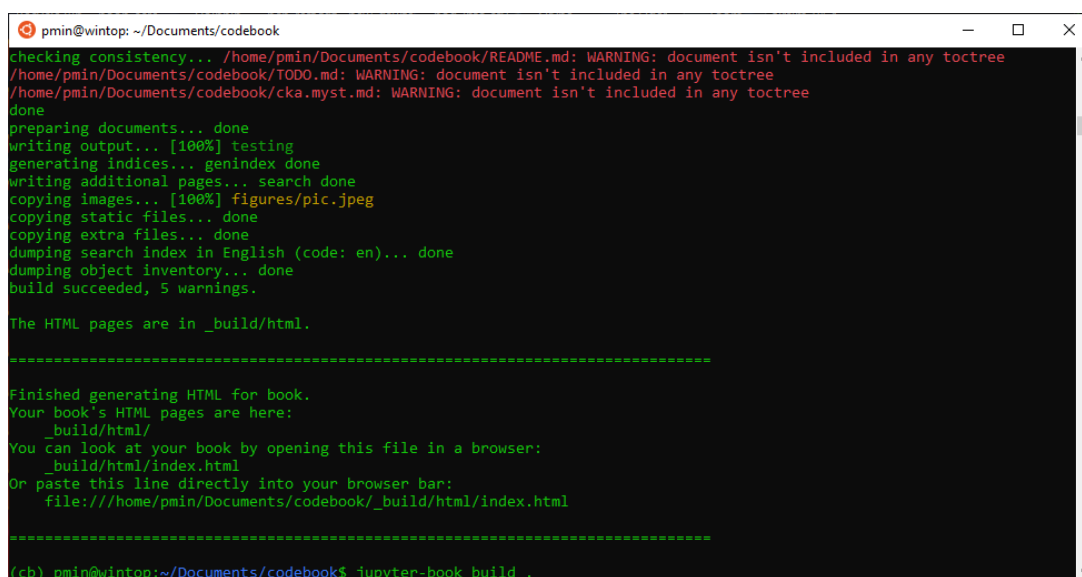


Figure 1.: Editing the Markdown source for this page in vscode.

I've evaluated many IDEs, and overall, I like [vscode](#) best. It's open source, free, and fast. It has very good integrated Python development tools, and it has an impressive array of plugins for almost any imaginable use case. The git and github tools are particularly well integrated, which makes it easy to do source control outside of the command line. There is an integrated debugger, as well as a terminal, so it's one-stop shop for development.

Others recommend [PyCharm](#): it has best-in-class code understanding, and scales well to large codebases. It's free for academics.

Use WSL on Windows



```
pmin@wintop: ~/Documents/codebook
checking consistency... /home/pmin/Documents/codebook/README.md: WARNING: document isn't included in any toctree
/home/pmin/Documents/codebook/TOD0.md: WARNING: document isn't included in any toctree
/home/pmin/Documents/codebook/cka.myst.md: WARNING: document isn't included in any toctree
done
preparing documents... done
writing output... [100%] testing
generating indices... genindex done
writing additional pages... search done
copying images... [100%] figures/pic.jpeg
copying static files... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded, 5 warnings.

The HTML pages are in _build/html.

=====

Finished generating HTML for book.
Your book's HTML pages are here:
_build/html/
You can look at your book by opening this file in a browser:
_build/html/index.html
Or paste this line directly into your browser bar:
file:///home/pmin/Documents/codebook/_build/html/index.html
=====

(cb) pmin@wintop:~/Documents/codebook$ jupyter-book build .
```

Figure 2.: WSL running on my Windows laptop

Windows' basic terminal lacks basic features. Powershell is powerful but it is very different from other platforms. For a while, the best way to get a Unix-style shell on Windows was to use the git bash tool. In my opinion, these days the best-in-class terminal to use on Windows is *Windows subsystem for Linux* (WSL).

WSL is an emulation layer that allows you to run a full Linux kernel inside of a Windows terminal window. [Once installed](#), you can install any Linux OS you like, Ubuntu being the *de facto* standard.

You won't be able to run GUI applications. However, many tools you'll want to use run webservers, for example jupyter. You'll be able to access these through your your normal Windows-based web browser, such as Chrome, Firefox or Edge. Your Linux installation will run in a virtual filesystem, which you can access through Windows explorer by typing `explorer` inside a WSL terminal. `code .` will fire up a version of `vscode` in your current directory.

Acknowledgements

I wouldn't have been able to write this book without the help of many people. They have made me a better programmer. Thanks first and foremost to Ella Batty for inviting me to give the workshop that inspired this book. Thanks to the reviewers, Tyler Sloan, Elizabeth DuPre and Martin Heroux who made the talk much better. Thanks to Ivan Savov for inspiring me to write in long form and for reviewing early versions. Thanks to everyone who gave me feedback on the book, including Kaytee Flick, Tyler Manning, Felix Taschbach, Konrad Kording and Russ Poldrack.

References

This book was inspired by many other long-form books, papers and tutorials. Check them out:

- Felienne Hermans. [The Programmer's Brain](#).
- Irving et al. [Research Software Engineering with Python](#).
- The Turing Way Community. [The Turing Way: A Handbook for Reproducible Data Science](#).
- [Software Carpentry lessons](#).
- Greg Wilson et al. [Good enough practices in scientific computing](#).
- The CodeRefinery Community. [Materials from CodeRefinery](#).

This book was generated by [jupyterbook](#), which builds on [Sphinx](#). The stylesheet is an adaptation of [tufte.css](#). The unicorn logo is from [twemoji](#), released under a CC-BY 4.0 license. The PDF version of this book was generated via [CurveNote](#).