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**.

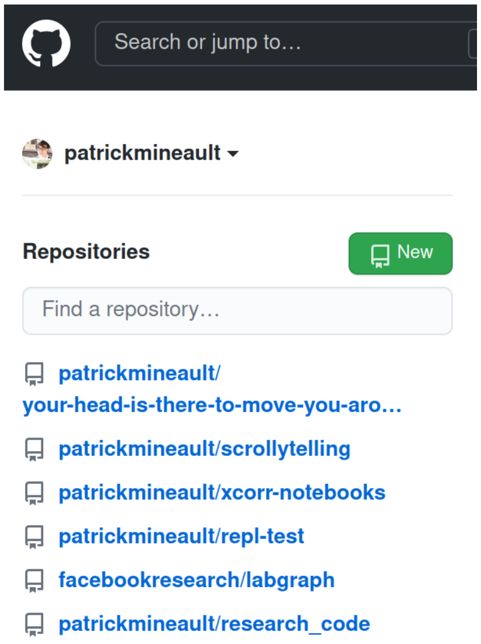
Search or jump to…
patrickmineault
Repositories
New
Find a repository…
patrickmineault/your-head-is-there-to-move-you-aro…
patrickmineault/scrollytelling
patrickmineault/xcorr-notebooks
patrickmineault/repl-test
facebookresearch/labgraph
patrickmineault/research_code

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.

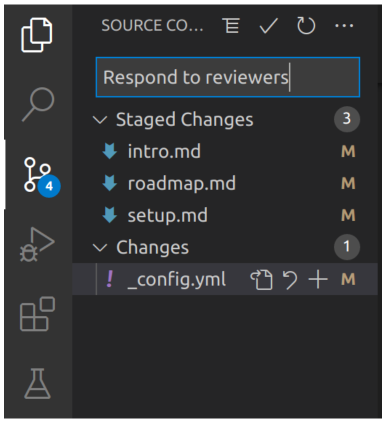

Figure 1.2: The git panel in VSCode.

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 `docker` - among others - to manage dependencies. Which one you prefer is a matter of personal

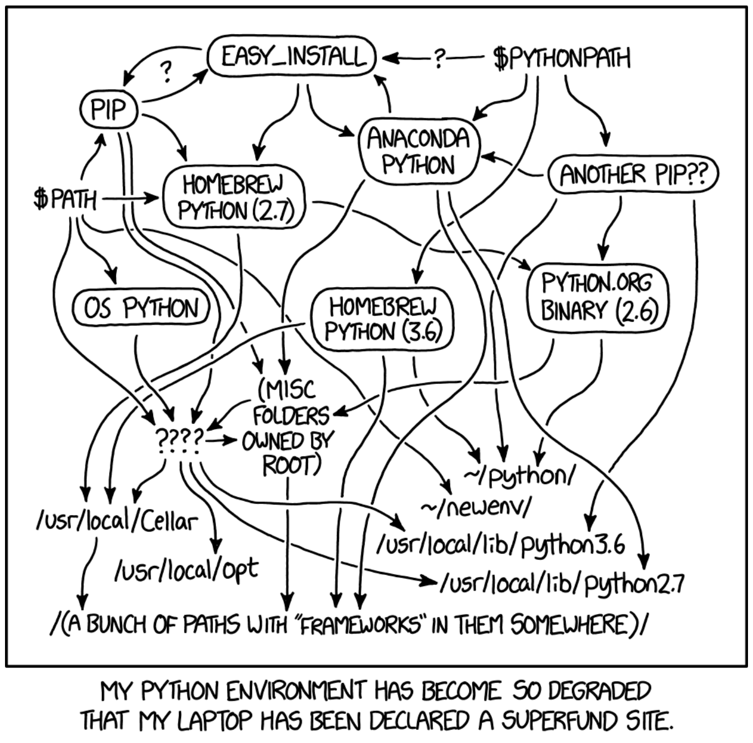

Figure 1.3: Python environments can be a real pain. From xkcd.com by Randall Munroe.

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 additional 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 this Carpentries tutorial. 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.

In many different programming frameworks - Ruby on Rails, React, etc. - people 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.