Using the template

You've created your project. You've read the opinions section. You're ready to start doing some work.

Here's a quick guide of the kinds of things we do once our project is ready to go. We'll walk through this example using git and GitHub for version control and jupyter notebooks for exploration, but you can use whatever tools you like.

Set up version control

Often, we start by initializing a `git` repository to track the code we write in version control and collaborate with teammates. At the command line, you can do this with the following commands which do the following: turn the folder into a git repository, add all of the files and folders created by CCDS into source control (except for what is in the `.gitignore` file), and then make a commit to the repository.

```
# From inside your newly created project directory
git init
git add .
git commit -m "CCDS defaults"
```

We usually commit the entire default CCDS structure so it is easy to track the changes we make to the structure in version history.

Now that the default layout is committed, you should push it to a shared repository. You can do this through the interface of whatever source control platform you use. This may be GitHub, GitLab, Bitbucket, or something else.

If you use GitHub and have the gh cli tool you can easily create a new

repository for the project from the commandline.

You'll be asked a series of questions to set up the repository on GitHub. Once you're done you'll be able to push the changes in your local repository to GitHub.

Make as a task runner

GNU Make is a tool that is typically pre-installed on Linux and macOS systems, and we use it as a "task runner" for CCDS projects. This means that when we have a series of shell commands we might want to run, such setting up a virtual environment or syncing the data to cloud storage, we set them up as recipes in the `Makefile`. To use a recipe, simply call

where `RECIPE_NAME` is the same of a recipe like `requirements` or `sync_data_up`. Projects created by CCDS include a `Makefile` with several recipes we've predefined. You'll see them referenced in the sections below. To see a list of all available commands, just call

on its own.

Create a Python virtual environment

We often use Python for our data science projects. We use a virtual environment to manage the packages we use in our project. This is a way to keep the packages we use in our project separate from the packages we use in other projects. This is especially important when we are working on multiple projects at the same time.

Cookicutter Data Science supports a few options for Python virtual environment management, but no matter which you choose, you can create an environment with the following commands:

Once the environment is created, you'll want to make sure to activate it.

You'll have to do this following the instructions for your specific environment manager. We recommend using a shell prompt that shows you which environment you are in, so you can easily tell if you are in the right environment, for example starship. You can also use the command `which python` to make sure that your shell is pointing to the version of Python associated with your virtual environment.

Once you are sure that your environment is activated in your shell, you can install the packages you need for your project. You can do this with the following command:

Add your data

There's no universal advice for how to manage your data, but here are some recommendations for starting points depending on where the data comes from:

- **Flat files (e.g., CSVs or spreadsheets) that are static** - Put these files into your `data/raw` folder and then run `make sync_data_up` to push the raw data to your cloud provider.
- **Flat files that change and are extracted from somewhere** - Add a Python script to your source module in `data/make_dataset.py` that downloads the data and puts it in the `data/raw` folder. Then you can use this to get the latest and push it up to your cloud host as it changes (be careful not to override your raw data).
- **Databases you connect to with credentials** - Store your credentials in `.env`. We recommend adding a `db.py` file or similar to your `data` module that connects to the database and pulls data. If your queries generally fit into memory, you can just have functions in the `db.py` to load data that you use in analysis. If not, you'll want to add a script like above to download the data to the `data/raw` folder.

Check out a branch

We'll talk about code review later, but it's a good practice to use feature branches and pull requests to keep your development organized. Now that you have source control configured, you can check out a branch to work with:

```
git checkout -b initial-exploration
```

Open a notebook

Note

The following assumes you're using a Jupyter notebook, but while the specific commands for another notebook tool may look a little bit different, the process guidance still applies.

Now you're ready to do some analysis! Make sure that your project-specific environment is activated (you can check with `which jupyter`) and run `jupyter notebook notebooks` to open a Jupyter notebook in the `notebooks/` folder. You can start by creating a new notebook and doing some exploratory data analysis. We often name notebooks with a scheme that looks like this:

```
0.01-pjb-data-source-1.ipynb
```

- `0.01` - Helps keep work in chronological order. The structure is `PHASE.NOTEBOOK`. `NOTEBOOK` is just the Nth notebook in that phase to be created. For phases of the project, we generally use a scheme like the following, but you are welcome to design your own conventions:
    - `0` - Data exploration - often just for exploratory work
    - `1` - Data cleaning and feature creation - often writes data to `data/processed` or `data/interim`

    - `2` - Visualizations - often writes publication-ready viz to `reports`
    - `3` - Modeling - training machine learning models
    - `4` - Publication - Notebooks that get turned directly into reports
- `pjb` - Your initials; this is helpful for knowing who created the notebook and prevents collisions from people working in the same notebook.
- `data-source-1` - A description of what the notebook covers

Now that you have your notebook going, start your analysis!

Refactoring code into shared modules

As your project goes on, you'll want to refactor your code in a way that makes it easy to share between notebooks and scripts. We recommend creating a module in the `{{ cookiecutter.module_name }}` folder that contains the code you use in your project. This is a good way to make sure that you can use the same code in multiple places without having to copy and paste it.

Because the default structure is a Python package and is installed by default, you can do the following to make that code available to you within a Jupyter notebook.

First, we recommend turning on the `autoreload` extension. This will make Jupyter always go back to the source code for the module rather than caching it in memory. If your notebook isn't reflecting the latest changes from your changes to a `.py` file, try restarting the kernel and make sure `autoreload` is on. We add a cell at the top of the notebook with the following:

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

Now all your code should be importable. At the start of the CCDS project, you picked a module name. It's the same name as the folder that is in the root project directory. For example, if the module name were `my_project` you could use code by importing it like:

```
from my_project.data import make_dataset

data = make_dataset()
```

Now it should be easy to do any refactoring you need to do to make your code more modular and reusable.

Make your code reviewable

We try to review every line of code written at DrivenData. Data science code in particular has the risk of executing without erroring, but not being "correct" (for example, you use standard deviation in a calculation rather than variance). We've found the best way to catch these kinds of mistakes is a second set of eyes looking at the code.

Right now on GitHub, it is hard to observe and comment on changes that happen in Jupyter notebooks. We develop and maintain a tool called [nbautoexport] that automatically exports a `.py` version of your Jupyter noteobok every time you save it. This means that you can commit both the `.ipynb` and the `.py` to source control so that reviewers can leave line-by-line comments on your notebook code. To use it, you will need to add `nbautoexport` to your requirements file and then run `make requirements` to install it.

Once `nbautoexport` is installed, you can setup the nbautoexport tool for your project with the following commands at the commandline:

```
nbautoexport install
```

```
nbautoexport configure notebooks
```

Once you're done with your work, you'll want to add it to a commit and push it to GitHub so you can open a pull request. You can do that with the following commandline commands

```
git add .  # stage all changed files to include them in the commit
git commit -m "Initial exploration"  # commit the changes with a message
git push  # publish the changes
```

Now you'll be able to create a Pull Request in GitHub.

Changing the `Makefile`

There's no magic in the `Makefile`. We often add project-specific commands or update the existing ones over the course of a project. For example, we've added scripts to generate reports with pandoc, build and serve documentation, publish static sites from assets, package code for distribution, and more.

Installing Make on Windows

Unfortunately, GNU Make is not typically pre-installed on Windows. Here are a few different options for getting Make:

- **Use a package manager.** You will need to install the package manager first if you don't already have it.
  - chocolatey (entry for Make)
  - winget (entry for Make)

    ```
    winget install -e --id GnuWin32.Make
    ```

  - scoop (entry for Make)
- **Windows Subsystem for Linux**. WSL is a full, non-virtualized Linux

environment inside Windows. You can use it to run all of your data science workflows on Ubuntu, and it will have Make included. See instructions for installing WSL here.

- **Cygwin**. A Unix-like development environment that includes Make. See instructions about installing Cygwin here.
- **MinGW**. A GNU development environment that runs on Windows and includes Make. See information about installing MinGW here.