Cookiecutter Data Science

A logical, reasonably standardized, but flexible project structure for doing and sharing data science work.

Why use this project structure?

> We're not talking about bikeshedding the indentation aesthetics or pedantic formatting standards — ultimately, data science code quality is about correctness and reproducibility.

When we think about data analysis, we often think just about the resulting reports, insights, or visualizations. While these end products are generally the main event, it's easy to focus on making the products *look nice* and ignore the *quality of the code that generates them*. Because these end products are created programmatically, **code quality is still important**! And we're not talking about bikeshedding the indentation aesthetics or pedantic formatting standards — ultimately, data science code quality is about correctness and reproducibility.

It's no secret that good analyses are often the result of very scattershot and serendipitous explorations. Tentative experiments and rapidly testing approaches that might not work out are all part of the process for getting to the good stuff, and there is no magic bullet to turn data exploration into a simple, linear progression.

That being said, once started it is not a process that lends itself to thinking carefully about the structure of your code or project layout, so it's best to start with a clean, logical structure and stick to it throughout. We think it's a pretty big win all around to use a fairly standardized setup like this one. Here's why:

Other people will thank you

> Nobody sits around before creating a new Rails project to figure out where they want to put their views; they just run `rails new` to get a standard project skeleton like everybody else.

A well-defined, standard project structure means that a newcomer can begin to understand an analysis without digging in to extensive documentation. It also means that they don't necessarily have to read 100% of the code before knowing where to look for very specific things.

Well organized code tends to be self-documenting in that the organization itself provides context for your code without much overhead. People will thank you for this because they can:

- Collaborate more easily with you on this analysis
- Learn from your analysis about the process and the domain
- Feel confident in the conclusions at which the analysis arrives

A good example of this can be found in any of the major web development frameworks like Django or Ruby on Rails. Nobody sits around before creating a new Rails project to figure out where they want to put their views; they just run `rails new` to get a standard project skeleton like everybody else. Because that default project structure is *logical* and *reasonably standard across most projects*, it is much easier for somebody who has never seen a particular project to figure out where they would find the various moving parts.

Another great example is the Filesystem Hierarchy Standard (https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard) for Unix-like systems. The `/etc` directory has a very specific purpose, as does the `/tmp` folder, and everybody (more or less) agrees to honor that social contract. That means a Red Hat user and an Ubuntu user both know roughly where to look for certain types of files, even when using each other's system — or any other standards-compliant system for that matter!

Ideally, that's how it should be when a colleague opens up your data science project.

You will thank you

Ever tried to reproduce an analysis that you did a few months ago or even a few years ago? You may have written the code, but it's now impossible to decipher whether you should use `make_figures.py.old`, `make_figures_working.py` or `new_make_figures01.py` to get things done. Here are some questions we've learned to ask with a sense of existential dread:

- Are we supposed to go in and join the column X to the data before we get started or did that come from one of the notebooks?

- Come to think of it, which notebook do we have to run first before running the plotting code: was it "process data" or "clean data"?
- Where did the shapefiles get downloaded from for the geographic plots?
- *Et cetera, times infinity.*

These types of questions are painful and are symptoms of a disorganized project. A good project structure encourages practices that make it easier to come back to old work, for example separation of concerns, abstracting analysis as a DAG (https://en.wikipedia.org/wiki/Directed_acyclic_graph), and engineering best practices like version control.

Nothing here is binding

> "A foolish consistency is the hobgoblin of little minds" — Ralph Waldo Emerson (and PEP 8! (https://www.python.org/dev/peps/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds))

Disagree with a couple of the default folder names? Working on a project that's a little nonstandard and doesn't exactly fit with the current structure? Prefer to use a different package than one of the (few) defaults?

Go for it! This is a lightweight structure, and is intended to be a good *starting point* for many projects. Or, as PEP 8 put it:

> Consistency within a project is more important. Consistency within one module or function is the most important. ... However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Getting started

With this in mind, we've created a data science cookiecutter template for projects in Python. Your analysis doesn't have to be in Python, but the template does provide some Python boilerplate that you'd want to remove (in the `src` folder for example, and the Sphinx documentation skeleton in `docs`).

Requirements

- Python 2.7 or 3.5
- cookiecutter Python package (http://cookiecutter.readthedocs.org/en/latest/installation.html) >= 1.4.0: `pip install cookiecutter`

Starting a new project

Starting a new project is as easy as running this command at the command line. No need to create a directory first, the cookiecutter will do it for you.

```
cookiecutter https://github.com/drivendata/cookiecutter-data-science
```

Example

This recording has been archived

All unclaimed recordings (the ones not linked to any user account) are automatically archived 7 days after upload.

Directory structure

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summar
ies
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.
g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment,
 e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- Make this project pip installable with `pip install -e`
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.testrun.org
```

Opinions

There are some opinions implicit in the project structure that have grown out of our experience with what works and what doesn't when collaborating on data science projects. Some of the opinions are about workflows, and some of the opinions are about tools that make life easier. Here are some of the beliefs which this project is built on—if you've got thoughts, please contribute or share them.

Data is immutable

Don't ever edit your raw data, especially not manually, and especially not in Excel. Don't overwrite your raw data. Don't save multiple versions of the raw data. Treat the data (and its format) as immutable. The code you write should move the raw data through a pipeline to your final analysis. You shouldn't have to run all of the steps every time you want to make a new figure (see Analysis is a DAG), but anyone should be able to reproduce the final products with only the code in `src` and the data in `data/raw`.

Also, if data is immutable, it doesn't need source control in the same way that code does. Therefore, ***by default, the data folder is included in the `.gitignore` file.*** If you have a small amount of data that rarely changes, you may want to include the data in the repository. Github currently warns if files are over 50MB and rejects files over 100MB. Some other options for storing/syncing large data include AWS S3 (https://aws.amazon.com/s3/) with a syncing tool (e.g., `s3cmd` (http://s3tools.org/s3cmd)), Git Large File Storage (https://git-lfs.github.com/), Git Annex (https://git-annex.branchable.com/), and dat (http://dat-data.com/). Currently by default, we ask for an S3 bucket and use AWS CLI (http://docs.aws.amazon.com/cli/latest/reference/s3/index.html) to sync data in the `data` folder with the server.

Notebooks are for exploration and communication

Notebook packages like the Jupyter notebook (http://jupyter.org/), Beaker notebook (http://beakernotebook.com/), Zeppelin (http://zeppelin-project.org/), and other literate programming tools are very effective for exploratory data analysis. However, these tools can be less effective for reproducing an analysis. When we use notebooks in our work, we often subdivide the `notebooks` folder. For example, `notebooks/exploratory` contains initial explorations, whereas `notebooks/reports` is more polished work that can be exported as html to the `reports` directory.

Since notebooks are challenging objects for source control (e.g., diffs of the `json` are often not human-readable and merging is near impossible), we recommended not collaborating directly with others on Jupyter notebooks. There are two steps we recommend for using notebooks effectively:

1. Follow a naming convention that shows the owner and the order the analysis was done in. We use the format `<step>-<ghuser>-<description>.ipynb` (e.g., `0.3-bull-visualize-distributions.ipynb`).
2. Refactor the good parts. Don't write code to do the same task in multiple notebooks. If it's a data preprocessing task, put it in the pipeline at `src/data/make_dataset.py` and load data from `data/interim`. If it's useful utility code, refactor it to `src`.

Now by default we turn the project into a Python package (see the `setup.py` file). You can import your code and use it in notebooks with a cell like the following:

```
# OPTIONAL: Load the "autoreload" extension so that code can change
%load_ext autoreload

# OPTIONAL: always reload modules so that as you change code in src, it gets loaded
%autoreload 2

from src.data import make_dataset
```

Analysis is a DAG

Often in an analysis you have long-running steps that preprocess data or train models. If these steps have been run already (and you have stored the output somewhere like the `data/interim` directory), you don't want to wait to rerun them every time. We prefer `make` (https://www.gnu.org/software/make/) for managing steps that depend on each other, especially the long-running ones. Make is a common tool on Unix-based platforms (and is available for Windows ()). Following the `make` documentation (https://www.gnu.org/software/make/), Makefile conventions (https://www.gnu.org/prep/standards/html_node/Makefile-Conventions.html#Makefile-Conventions), and portability guide (http://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.69/html_node/Portable-Make.html#Portable-Make) will help ensure your Makefiles work effectively across systems. Here are some (http://zmjones.com/make/) examples (http://blog.kaggle.com/2012/10/15/make-for-datascientists/) to get started (https://web.archive.org/web/20150206054212/http://www.bioinformaticszen.com/post/decomplected-workflows-makefiles/). A number of data folks use `make` as their tool of choice, including Mike Bostock (https://bost.ocks.org/mike/make/).

There are other tools for managing DAGs that are written in Python instead of a DSL (e.g., Paver (http://paver.github.io/paver/#), Luigi (http://luigi.readthedocs.org/en/stable/index.html), Airflow (http://pythonhosted.org/airflow/cli.html), Snakemake (https://bitbucket.org/snakemake/snakemake/wiki/Home), Ruffus (http://www.ruffus.org.uk/), or Joblib (https://pythonhosted.org/joblib/memory.html)). Feel free to use these if they are more appropriate for your analysis.

Build from the environment up

The first step in reproducing an analysis is always reproducing the computational environment it was run in. You need the same tools, the same libraries, and the same versions to make everything play nicely together.

One effective approach to this is use virtualenv (https://virtualenv.pypa.io/en/latest/) (we recommend virtualenvwrapper (https://virtualenvwrapper.readthedocs.org/en/latest/) for managing virtualenvs). By listing all of your requirements in the repository (we include a `requirements.txt` file) you can easily track the packages needed to recreate the analysis. Here is a good workflow:

1. Run `mkvirtualenv` when creating a new project
2. `pip install` the packages that your analysis needs
3. Run `pip freeze > requirements.txt` to pin the exact package versions used to recreate the analysis
4. If you find you need to install another package, run `pip freeze > requirements.txt` again and commit the changes to version control.

If you have more complex requirements for recreating your environment, consider a virtual machine based approach such as Docker (https://www.docker.com/) or Vagrant (https://www.vagrantup.com/). Both of these tools use text-based formats (Dockerfile and Vagrantfile respectively) you can easily add to source control to describe how to create a virtual machine with the requirements you need.

Keep secrets and configuration out of version control

You *really* don't want to leak your AWS secret key or Postgres username and password on Github. Enough said — see the Twelve Factor App (http://12factor.net/config) principles on this point. Here's one way to do this:

Store your secrets and config variables in a special file

Create a `.env` file in the project root folder. Thanks to the `.gitignore`, this file should never get committed into the version control repository. Here's an example:

```
# example .env file
DATABASE_URL=postgres://username:password@localhost:5432/dbname
AWS_ACCESS_KEY=myaccesskey
AWS_SECRET_ACCESS_KEY=mysecretkey
OTHER_VARIABLE=something
```

Use a package to load these variables automatically.

If you look at the stub script in `src/data/make_dataset.py`, it uses a package called python-dotenv (https://github.com/theskumar/python-dotenv) to load up all the entries in this file as environment variables so they are accessible with `os.environ.get`. Here's an example snippet adapted from the `python-dotenv` documentation:

```python
# src/data/dotenv_example.py
import os
from dotenv import load_dotenv, find_dotenv

# find .env automagically by walking up directories until it's found
dotenv_path = find_dotenv()

# load up the entries as environment variables
load_dotenv(dotenv_path)

database_url = os.environ.get("DATABASE_URL")
other_variable = os.environ.get("OTHER_VARIABLE")
```

AWS CLI configuration

When using Amazon S3 to store data, a simple method of managing AWS access is to set your access keys to environment variables. However, managing mutiple sets of keys on a single machine (e.g. when working on multiple projects) it is best to use a credentials file (https://docs.aws.amazon.com/cli/latest/userguide/cli-config-files.html), typically located in `~/.aws/credentials`. A typical file might look like:

```
[default]
aws_access_key_id=myaccesskey
aws_secret_access_key=mysecretkey

[another_project]
aws_access_key_id=myprojectaccesskey
aws_secret_access_key=myprojectsecretkey
```

You can add the profile name when initialising a project; assuming no applicable environment variables are set, the profile credentials will be used be default.

Be conservative in changing the default folder structure

To keep this structure broadly applicable for many different kinds of projects, we think the best approach is to be liberal in changing the folders around for *your* project, but be conservative in changing the default structure for *all* projects.

We've created a `folder-layout` label specifically for issues proposing to add, subtract, rename, or move folders around. More generally, we've also created a `needs-discussion` label for issues that should have some careful discussion and broad support before being implemented.

Contributing

The Cookiecutter Data Science project is opinionated, but not afraid to be wrong. Best practices change, tools evolve, and lessons are learned. **The goal of this project is to make it easier to start, structure, and share an analysis.** Pull requests (https://github.com/drivendata/cookiecutter-data-science/pulls) and filing issues (https://github.com/drivendata/cookiecutter-data-science/issues) is encouraged. We'd love to hear what works for you, and what doesn't.

If you use the Cookiecutter Data Science project, link back to this page or give us a holler (https://twitter.com/drivendataorg) and let us know (mailto:info@drivendata.org)!

Links to related projects and references

Project structure and reproducibility is talked about more in the R research community. Here are some projects and blog posts if you're working in R that may help you out.

- Project Template (http://projecttemplate.net/index.html) - An R data analysis template
- "Designing projects (http://nicercode.github.io/blog/2013-04-05-projects/)" on Nice R Code
- "My research workflow (http://www.carlboettiger.info/2012/05/06/research-workflow.html)" on Carlboettifer.info
- "A Quick Guide to Organizing Computational Biology Projects (http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000424)" in PLOS Computational Biology

Finally, a huge thanks to the Cookiecutter (https://cookiecutter.readthedocs.org/en/latest/) project (github (https://github.com/audreyr/cookiecutter)), which is helping us all spend less time thinking about and writing boilerplate and more time getting things done.