Cookiecutter Data Science — Organize your Projects — Atom and Jupyter

Robert R.F. DeFilippi Follow
Sep 22, 2018 · 11 min read

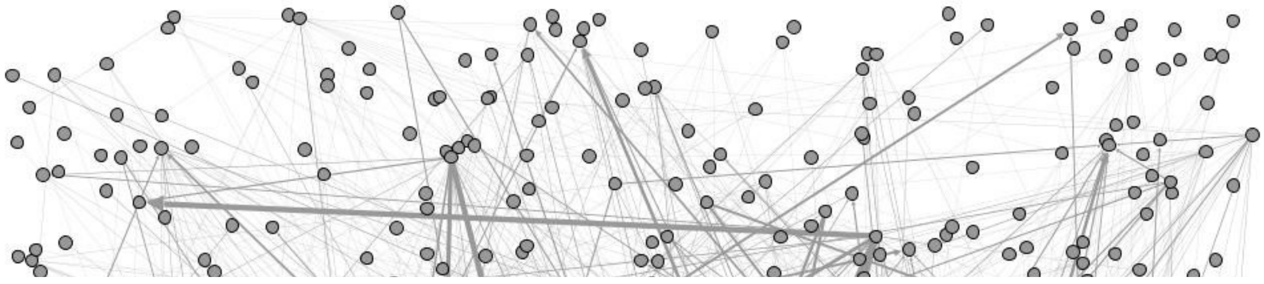
One thing prevalent in most data science departments is messy notebooks and messy code. There *are* examples of beautiful notebooks out there, but for the most part notebook code is rough … really rough. Not to mention all the files, functions, visitations, reporting metrics, etc. scattered through files and folders with no real structure to them.

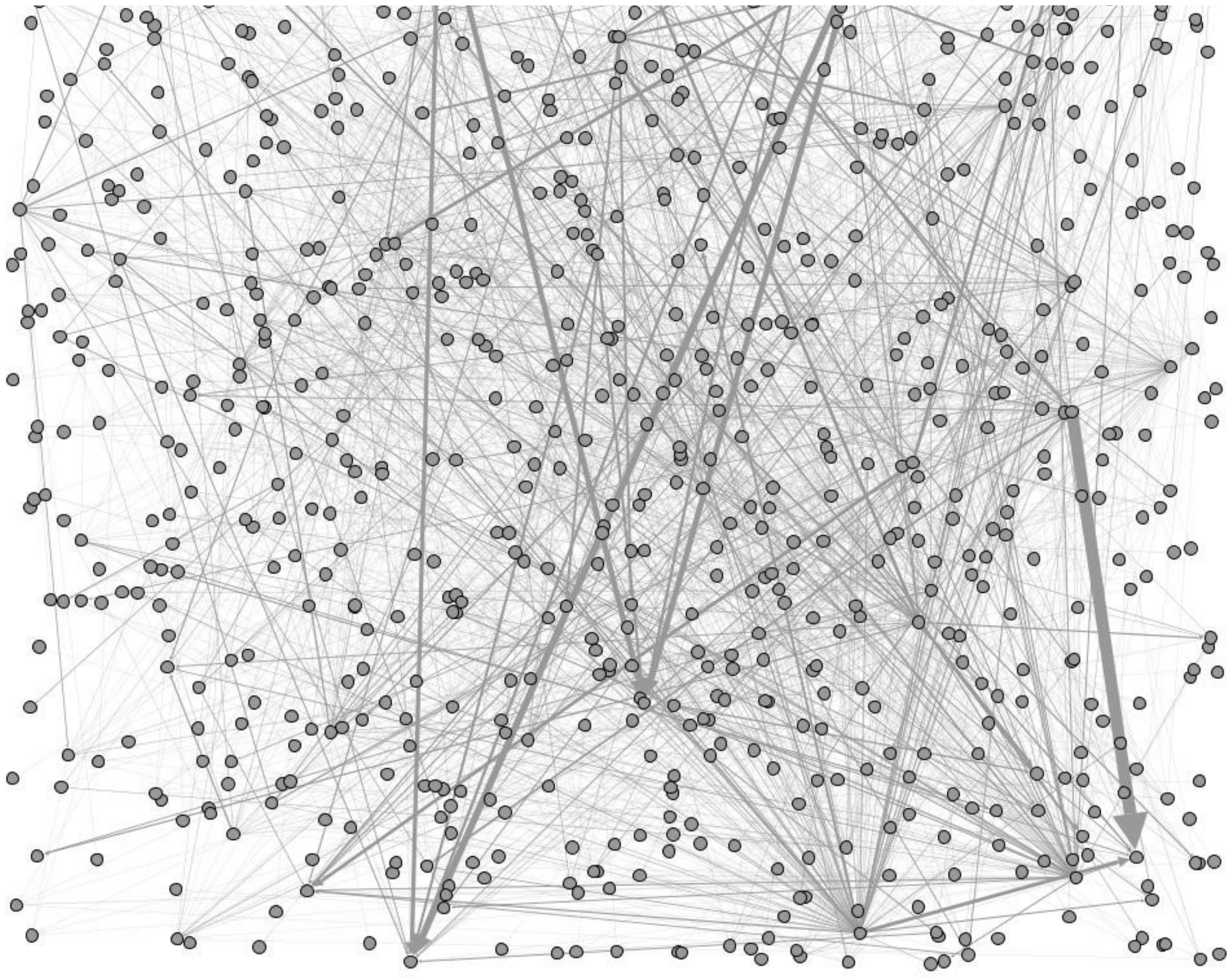
I have bee guilty of going back to a previous projects, searching for the `matplotlib` function I've written 50 times before that creates a certain graph with a certain metric. And, I found it in cell 200 after searching through a few different notebooks. Not efficient at all.

Now consider a project you're currently doing, and think if are you making this harder or easier for yourself in a few weeks time when you need to come back and review the code, or find something specific like a figure or the original data set?

One of the creeds some software development is not to repeat yourself. So, let's find out how we can do that.

Also, I give a lot of credit to Go Data Driven's blog, which was a huge source of inspiration for this post.





This … is a mess

How I Think About Projects

Here are some of the main principles of projects I use to keep myself organized (sane) :

1. Implementing version control;
2. Setting up a virtual environment for reproducible results;
3. Separating raw data, intermediate data, and final data;
4. Documenting work, and;
5. Separate modules for custom functions.

Now let’s get into this.

Git Version Control

COMMENT
DATE
CREATED MAIN LOOP & TIMING CONTROL
14 HOURS AGO
ENABLED CONFIG FILE PARSING
9 HOURS AGO
MISC BUGFIXES
5 HOURS AGO
CODE ADDITIONS/EDITS
4 HOURS AGO
MORE CODE
4 HOURS AGO
HERE HAVE CODE
4 HOURS AGO
AAAAAAAA
3 HOURS AGO
ADKFJSLKDFJSDKLFJ
3 HOURS AGO
MY HANDS ARE TYPING WORDS
2 HOURS AGO
HAAAAAAAAANDS
2 HOURS AGO
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

This is normal

Git should be standard for most data scientists, but it's not at the moment. I've seen Git used as a place just to store final work but it should be used throughout the project and a way for the entire team to collaborate with each other.

I use it because I forget why I've made certain changes to a file, and want a way to review my work with annotations and versions.

If you don't have an account, get one here and start keeping track of your code. I won't go over the entire process, but this article will show you how to get your first `$ git commit -m "My first commit message"` and `$ git push` for your project.

Now you can have version control over each project because every commit is simply a new version saved which can be referenced.

Once you start pushing code, the thing that will slow productivity to a halt after reviewing a 6-month project with 100s of commits, will be reading your old commit messages and asking "WTF does this mean?" To solve this problem, make sure all your messages are in the imperative mood.

Taking this from Chris Beams,

> *A properly formed Git commit subject line should always be able to complete the following sentence:*
>
> *"If applied, this commit will* your subject line here"

This will help greatly when you're making different models. In the past, I would have `modelv1.h5` , `modelv2.h5` , etc. and I could not remember why each of them was different or why I changed the weighting on something.

With Git, I can now just use one name `model.h5` and put my update comments in the commit message. E.g. I know I've added a new layer in my Keras model so my commit message would be `$ git commit -m "Add additional layer with 50 neurons"` .

If I ever need to go back and see what was involved in a certain model I'll be able to view or rollback to particular commit and see the work I've done.

This saves so much time. Trust me.

$ virtualenv

Now that we have version control down, let's look at making sure someone else can reproduce our work with a virtual environment.

I was against using virtual environments when I first started learning data science because they seemed like a pain to set up and maintain. However, they will save so much headache as your project grows and you need to start sharing information.

More on them here and here.

The main reason for using virtual environments is package version control for packages and allowing you to easily import custom functions from different parts of your project, keeping you organized. Someone might be using `pandas 0.23` but you're using `pandas 0.19` , and you don't want to upgrade because it's going to break something in another project.

By having a virtual environment you'll be able to start with a fresh new version of Python — it's like getting a new car — and install packages with specific versions directly into each project without any conflict from anything you might have previously installed. So, you'll be able to have project `Foo` using `pandas 0.23` and then project `Bar` using `pandas 0.19`, without any chance of conflict with each other.

If you don't have the required packages to set up your virtual environment, you can get them by following the instructions below.

```
# Get a virtual environment

# Python2
$ pip install virtualenv

# Python3.6+
# Already installed with venv
```

Now let's set up the environment.

```
# Python 2:
$ virtualenv env

# Python 3.6+
$ python3 -m venv <<name_of_directory>>
```

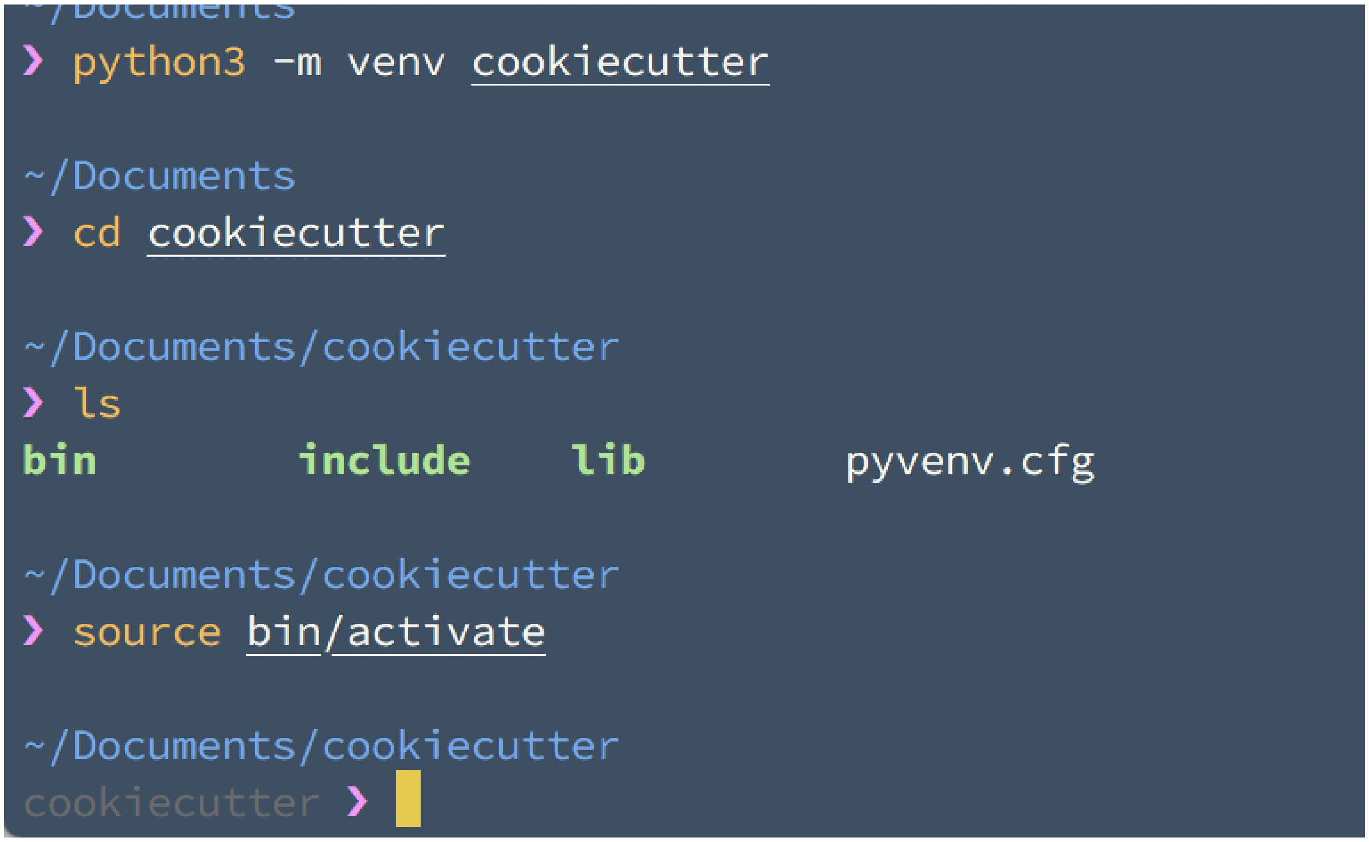
If your directory does not exist, specify what you want as `name_of_directory` and it will be created. Else, the `venv` will just be installed into the chosen directory.

Once that is done, activate your environment.

```
$ source name_of_directory/bin/activate
```

If all goes well, you should see a change in your terminal's prompt showing the name of the environment on the command line. And, to deactivate the environment, just run `$ deactivate` in the root of your project.

```
~/Documents
```



```
~/Documents
❯ python3 -m venv cookiecutter

~/Documents
❯ cd cookiecutter

~/Documents/cookiecutter
❯ ls
bin          include    lib          pyvenv.cfg

~/Documents/cookiecutter
❯ source bin/activate

~/Documents/cookiecutter
cookiecutter ❯
```

The name of our venv is 'cookiecutter'

Easy.

Next, install a fresh version of Jupyter into your environment.

```
(env_name) $ pip3 install jupyter
```

We have a fresh version of Python and Jupyter to work with for our project.

Try some cookiecutter with that data

Enter cookiecutter which creates your project structure. There is *a lot* with this package that is useful, but I've found when I'm just doing projects on my own I don't need all of the features and packages provided.

And, that's ok. Because every project is different and you'll have different requirements depending on what you work on.The package is structured so that you should only keep the features you need, and if you find a structure that works you're able to import it easily on the next project.

Installation and using it is easy. Just two lines of code to run.

```
# Install
$ pip3 install cookiecutter

# cd to the directory you want to start a new project in
cookiecutter https://github.com/drivendata/cookiecutter-data-science
```

That is not the only structure available, as you can look through some additional options here.

Here is what mine looks like:

```
├── README.md          <- Front page of the project. Let everyone
│                         know the major points.
│
├── models             <- Trained and serialized models, model
│                         predictions, or model summaries.
│
├── notebooks          <- Jupyter notebooks. Use set naming
│                         E.g. `1.2-rd-data-exploration`.
│
├── reports            <- HTML, PDF, and LaTeX.
│   └── figures        <- Generated figures.
│
├── requirements.txt   <- File for reproducing the environment
│                         `$ pip freeze > requirements.txt`
├── data
│   ├── external       <- Third party sources.
│   ├── interim        <- In-progress intermediate data.
│   ├── processed      <- The final data sets for modelling.
│   └── raw            <- The original, immutable data.
│
└── src                <- Source code for use in this project.
    ├── __init__.py    <- Makes src a Python module.
    │
    ├── custom_func.py <- Various custom functions to import.
    │
    ├── data           <- Scripts to download or generate data.
    │   └── make_dataset.py
    │
    ├── features       <- Scripts 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
    │
    └── viz            <- Scripts to create visualizations.
        └── viz.py
```

If someone wants to recreate your project with the same package versions, all you need to do is export the list with `freeze` so they can install the same versions as the project.

```
# In project root

$ pip freeze > requirements.txt

# And to install the packages

$ pip install -r requirements.txt
```

Data Folders

I think this is one of the most import sections of cookiecutter. The `data/` folders that allow you to organize yourself and control your data sources. It is so simple that I should have been organizing my projects like this from the start.

There are four standard folders:

- External;
- Interim;
- Processed, and;
- Raw.

By segmenting your data this way you'll be able to save your in-progress work, and ensure that your raw data has an immutable folder to live in. You'll also be able to quickly and logically access your data rather than just importing a single lump `.csv` which may or may not be immutable.

No more accidentally saving over the raw data because everything was stored in the same folder, or dealing with file names like `dataV1.csv` or `finalV1.csv`, because everything will be (should be) put in its own place.

Documenting The Project

Not a huge concern, a lot of data science projects are done in Jupyter which allows the reader to (hopefully) follow the project logically. Allowing others to reproduce the work, should they have access to the correct packages and the same data the author had.

And now that cookiecutter has segmented the project more, the notebooks should be easier to read rather than a mess of code, functions, and results as they have been moved to other parts of the project.

E.g. Your `matplotlib` functions can now live in `./src/viz` and all the results from these functions can be saved to `./reports/figures`.

Get Your Functions In

I used to have cells scattered all over my notebooks with custom functions, that I would later use in the project. And, the more I would write the harder it became to keep track of them all for the project. Let alone try to debug them or test them.

Even finding them for a project in the future to reuse became a major pain.

Later I would write all my function in `.src/funct/Foo.py` and stash them in different folders for some semblance of organization much like an actual software project. This would allow me to partition my project, so my notebook had the structure of the what was being analyzed while the functions lived in `Foo.py`, and were imported into my notebook.

```
.src/                   # Main Folder
  └──funct/             # Sub Folder
       ├── Foo.py       # Class with custom methods
       └── __init__.py
```

However, I would need to import the files into my notebook like this.

```
# Meanwhile in my old notebooks ...

import os

currDir = os.path.dirname(os.path.realpath("__file__"))
rootDir = os.path.abspath(os.path.join(currDir, '..'))
sys.path.insert(1, rootDir + '/src/funct') # import func package

# Now I can finally import Foo from the funct package

from func import Foo

print(Foo.foo()) # 'foo'
```

Not very elegant at all, and prone to checking Stack Overflow on how to use `os.path.dirname` and `os.path.realpath` correctly.

Now, with cookiecutter, you're able to easily link everything in your `./src/` with your notebook so you can `import` the `Foo.py` directly.

```
# Link ./src/ with your notebook

# Run this in the project root

$ pip3 install --editable .
>>Obtaining file:///Users/foo/cookiecutter/cookiecutter-template
>>Installing collected packages: src
>>Running setup.py develop for src
>>Successfully installed src

$ pip3 list
Package    Version Location
---------- ------- --------------------------------------------------
-----
pip        18.0
setuptools 39.0.1
src        0.1.0   /Users/foo/cookiecutter/cookiecutter-template
```

So this allows us to use the `./src` folders provided with cookiecutter to easily import custom functions into your notebook. And, allow us to have a common location to put custom functions for easy retrieval for later projects.

```
# Back in my notebook
from src.funct import Foo

print(Foo.foo()) # foo
```

Note, if you do have issues with `pip3 install – editable .` and you cannot import your `/src` package, try `$ python setup.py develop – no-deps` in the project root.

If that still does not work just copy and paste this code at the top of your notebook. It will allow you to import `src` as a package. However, this is not suggested.

```
if __name__ == '__main__' and __package__ is None:
    from os import sys, path

sys.path.append(path.dirname(path.dirname(path.abspath("__file__"))))
```

Using Jupter and Atom Together

It's inefficient to write all the `*.py` files in Jupyter, so instead use an IDE which has a linter, interactive debugger, and other features that make it more preferable to writing *some* code your project. Jupyter has its strengths, but there are some things that only an IDE can do.

Usually, I use IntelliJ or PyCharm. However, because not everyone has access to that editor I'll show the examples in Atom. It also has an IDE feature now, so it will be easier to run debugging within this editor.

```
getGrammarScopes () { return atom.config.get('ide-type
getLanguageName () { return 'TypeScript' }
getServerName () { return 'SourceGraph' }

startServerProcess () {
  const args = [ 'node_modules/javascript-typescript-l
```

TypeScriptLanguageClient
getSuggestions
requestAndCleanSuggestion

```
    return super.spawnChildNode(args, { cwd: path.join(_
  }

  preInitialization (connection) {
    connection.onCustom('$/partialResult', () => {}) //
  }

  async getSuggestions (request) {
    const prefix = request.prefix.trim()
    const server = await this._serverManager.getServer(r
    if (server == null) {
      Promise.resolve([])
    }
TypeScriptLanguageClient.getSuggestions(request: any): Promise<any>
                                                              ) {
      return Promise.resolve([])
    }

    if (prefix.length > 0 && prefix != '.'  && server.cu
      // fuzzy filter on this.currentSuggestions
```

Atom IDE in Action

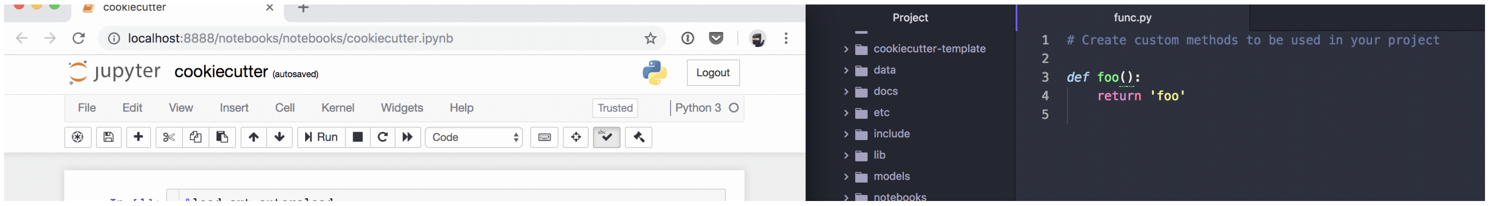
So let’s run this editor next to our Juptyer notebook. Just split (as shown below) your screen, with one side with Jupyter and one with the IDE.

As a note, you need to make sure you have two magic functions imported at the beginning of your notebook.

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

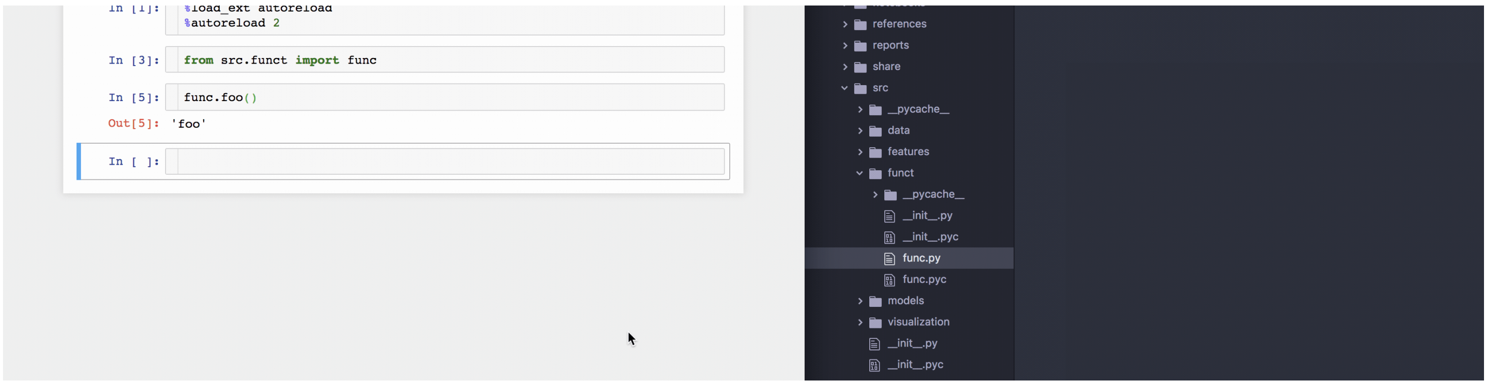
The suffix `2` is used after `autoreload`, to reload all modules every time *before* executing the Python code typed. This will enable us to edit functions in Atom, debug them, and quickly import the code into our notebook, rather than having to write out a debug the function directly in Jupyter.

I’ve made a gif to illustrate how this looks in practice.



```
# Create custom methods to be used in your project

def foo():
    return 'foo'
```



```
In [1]: %load_ext autoreload
        %autoreload 2

In [3]: from src.funct import func

In [5]: func.foo()
Out[5]: 'foo'

In [ ]:
```

This also save a lot of time

With the IDE up and running next to Jupyter, not only are you able to easily edit your files but you can also view your entire project easily rather than having to flip back and forth between windows in your browser.

Final Thoughts

At the beginning of the article, I outlined the five things I think about when doing projects, and I hope each point has been explained with the "how" I approach each of them.

1. Implementing version control (Git)
2. Setting up a virtual environment for reproducible results (`venv`);
3. Separating raw data, intermediate data, and final data (`./data`);
4. Documenting work (Jupyter notebooks), and;
5. Separate modules for custom functions(`./src`).

Of course, all of this is all made possible with cookiecutter. Without this package, I would still be writing mediocre code in a disorganized mess. However, I'm now able to more easily organize myself, and most importantly, find old code to reuse in future projects.

I hope these methods outlined help you on your next project, and you've learned something new.

Cheers.

Additional Reading

Home - Cookiecutter Data Science

A project template and directory structure for Python data science projects.

drivendata.github.io

A Quick Guide to Organizing [Data Science] Projects (updated for 2018)

Noble's advice transcends computational biology and is broadly applicable to the field of data science.

medium.com

GoDataDrivenBlog

GoDataDrivenBlog

GoDataDrivenBlogblog.godatadriven.com

Python Data Science Cookiecutter Software Development Jupyter