Best Practices when Writing Code

"Writing Code in Econ" Series

- Overview (tutorial_workflow_0overview.html)
- Previous: Project Organization (tutorial_workflow_2project_org.html)

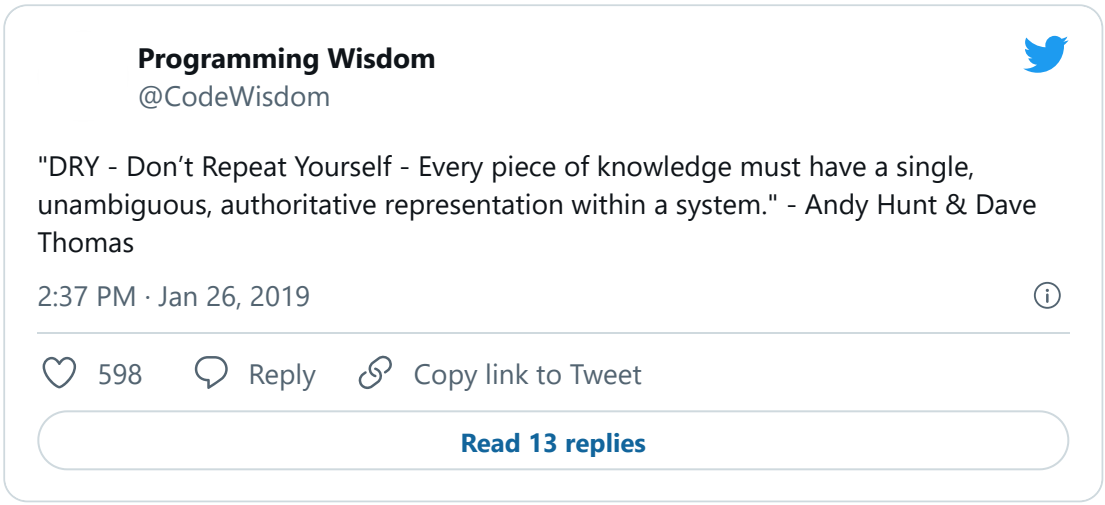
Page Contents

> Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.
>
> ~ Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

Writing easy-to-read code is important, especialy when you're working with a team. Code that's easy to read improves everyone's productivity, even when the "other" person reading the code is Future You, because Future You may not rememeber what Present You is doing today and why.

Note: This guide is written with Python in mind, but very little of it is specific to Python. The principles are just as helpful in Stata, R, and other languages.

1 "Don't Repeat Yourself"



1.1 No Copy-and-pasting

Variables and algorithms should be defined in exactly one place. This is a cardinal rule. If you copy and paste code to use it in multiple places, it is difficult to make sure any changes you make later also get copied and pasted.

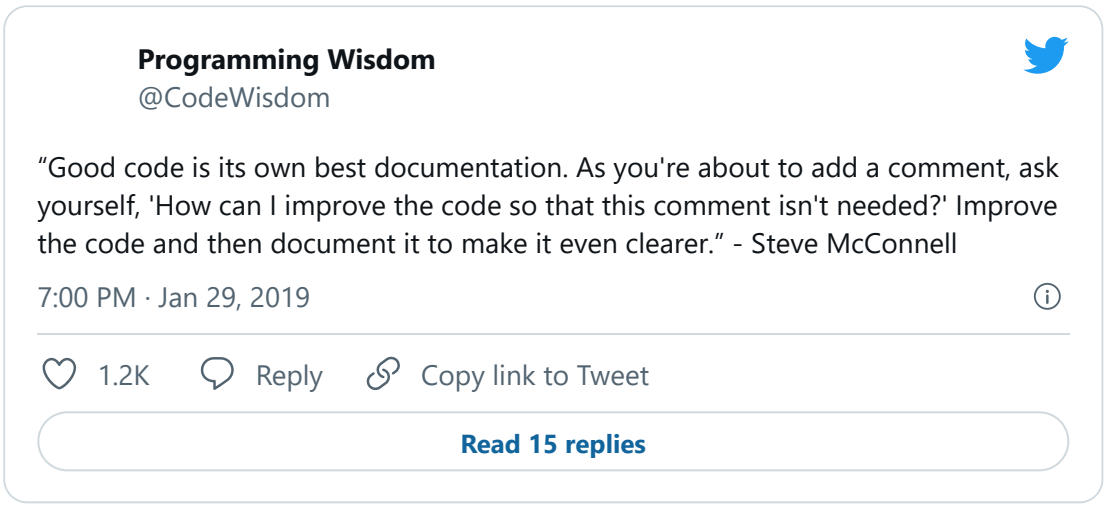
If a file name (e.g., `c:\data\subsample2.csv`) is defined in only one place and imported elsewhere, you can change the file name in that single place and know for certain the change will propagate through the rest of your code.

If you run lots of robustness checks on the same sample of data, do that data prep in a single function and import that function in every robustness check script. That way you know for a certainty that every regression is using exactly the same sample, even if you come in and change something later.

1.2 Automate empirical results completely

Figures and tables created by your code should be inserted directly into presentations or papers without manual editing. You don't want to have to tweak tables every time you re-create them. You're bound to insert a typo, forget one of the steps in tweaking, etc. Don't repeat yourself!

2 Write self-documenting code



Give descriptive names to variables, functions, and files. Err on the side of names being too long and descriptive. It takes longer to type these names but the added information will be worth it in the long run. (Part of the reason you should be using tools like a good text editor and a good console like CMDer is that they have predictive text completion and other tools to help with long variable and file names.) Use comments to clarify why a particular coding choice was made. A few examples:

- The filename `reg_sale_price.py` is beter than `reg.py`. A name like `table1.py` is completely forbidden.
- It is helpful to name functions or variables that only take on True/False values with `is_` or `has_`. See `is_triangle` in the example code below.

A few other naming conventions we tend to follow:

- If a function only deals with a single Pandas DataFrame, call it `df`.
- Indexes in loops should still be descriptive. However, the index inside a list comprehension can be something short like `i` or `x`, provided there is only one variable being looped over.
- Degrees longitude and latitude will tend to be called `x` and `y` respectively. This is not a requirement but we have found that it's easy to mix up "latitude" and "longitude".

2.1 No "magic" numbers

A "magic number" (https://stackoverflow.com/questions/47882/what-is-a-magic-number-and-why-is-it-bad) is a number in code that is directly used, usually with no information about what the number represents or why that specific number is used.

Consider the following code fragment which has been adapted from a script behind several well-known published papers:

```
# Bad:
vector_A = np.zeros(3109, 1)
vector_B = np.zeros(565, 1)
vector_C = np.zeros(3109, 1)
```

What do the numbers `3109` and `565` represent? Nobody knows. You could add a comment

```
# Less bad:
# Here 3109 is number of counties in the sample, 565 is time periods
vector_A = np.zeros(3109, 1)
vector_B = np.zeros(565, 1)
vector_C = np.zeros(3109, 1)
```

But the best way is Don't Repeat Yourself: Define these numbers in exactly one place--especially if they're used in multiple places like here--and let the code be self explanatory.

```
# Good:
NUM_COUNTIES = 3109
NUM_TIME_PERIODS = 565
vector_A = np.zeros(NUM_COUNTIES, 1)
vector_B = np.zeros(NUM_TIME_PERIODS, 1)
vector_C = np.zeros(NUM_COUNTIES, 1)
```

2.2 Use functions like paragraphs

The main unit of code should be functions. Even if you're only going to use a function once, breaking your script into functions makes it easier to read.

Suppose it takes 30 lines of code to create a balanced panel within your data. Even if you only do it once, a line like `df = create_balanced_panel(df)` is much easier to digest than the 30 lines themselves.

2.3 Use docstrings

Python has a special convention called docstrings. At the beginning of a script or the beginning of a function, you can add a multiline string with triple quotes `"""` that will serve as the documentation for that script or function. The special `help` function will display the docstring of any object passed to it, e.g., `help(my_function)` will print the docstring of `my_function` if it exists.

```
"""
This is the script docstring. It describes the main purpose of a script. It
Can be many lines long.

It will also preserve formatting.
"""


def my_function():
    """
    This is the docstring for `my_function`. It will be printed on the
    console whenever I call `help(my_function)`.
    """
    pass
```

Most other languages do not use docstrings, but you can you block comments, like `/* */` in Stata, to achieve the same goal.

2.4 Scripts should only contain functions, constants, and an "if-main" block

IMPORTANT: Most of the guidance on this page is stylistic, aimed at making you more productive as a programmer while having litte impact on what your code actually does. This section is different. It does have a stylistic element, but not using an "if-main" block can significantly impact how your script runs.

When you import something from another Python file, the entire file is executed. Suppose you want to import a function called `computation_that_takes_forever` which is used in script `cleandata.py` that looks like this

```
1   import pandas as pd
2
3   MAX_ITERATIONS = 1000
4
5
6   def computation_that_takes_forever(df):
7       """ This takes a long time """
8       df = df ** df ** df
9       # Other stuff
10      return df
11
12  df = pd.read_csv('giant_dataset.csv')
13  df = computation_that_takes_forever(df)
```

When you run `from cleandata import computation_that_takes_forever` , the entire `cleandata.py` script gets run, **including the data loading and processing on lines 9 and 10!!** We don't want to run this whole data cleaning process every time we import the function.

The solution is to put that execution in a different function or in an "if-main" block like so:

```
1  import pandas as pd
2
3  MAX_ITERATIONS = 1000
4
5
6  def computation_that_takes_forever(df):
7      """ This takes a long time """
8      df = df ** df ** df
9      # Other stuff
10     return df
11
12 if __name__ == '__main__':
13     df = pd.read_csv('giant_dataset.csv')
14     df = computation_that_takes_forever(df)
```

Any code inside the `if __name__ == '__main__'` block will only be executed if the script is called directly from the command line or via `%run` in IPython. It is *not* run if the script is imported by another script. So in our new version of `cleandata.py`, lines 10 and 11 only get executed if we run `python cleandata.py` from the command line or `%run cleandata.py` inside IPython.

3 Use a consistent style

Just like for writing prose, there are style guides for writing code which help make your writing easier to read. Python has an official style guide called PEP8 that contains more rules than I'll go over here. However, there is a great Python tool called Flake8 that will automatically check your code for PEP8 and syntax errors. It can be integrated into the Atom editor using the `linter-flake8` plugin.

A few examples of important PEP8 rules that we'll follow:

3.1 Lines should be less than 80 characters wide

Fortunately line wrapping in Python is very easy. Anything within parentheses can be broken across lines, including function calls:

```
std_devs = find_std_dev(variable1,
                        variable2,
                        variable3)
```

Even if a line of code isn't 80 characters long, it's often better to break it into several lines for clarity.

```
bad_seconds_per_year = 60 * 60 * 24 * 365

good_seconds_per_year = (
    60 *     # seconds per minute
    60 *     # minutes per hour
    24 *     # hours per day
    365      # days per year
)

bad_dataframe_chain = df.rename(columns={'Yearly Avg': 'mean'}).drop('dumb_var', axis=1)
bad_dataframe_chain = bad_dataframe_chain.set_index('state_id')

good_dataframe_chain = (df
                        .rename(columns={'Yearly Avg': 'mean'})
                        .drop('dumb_var', axis=1)
                        .set_index('state_id'))
```

Long strings can be wrapped in parentheses as well and will automatically be concatenated. Just don't forget to add spaces where necessary.

```
one_long_string = (
    "When in the course of human events "
    "it becomes necessary for one "
    "people to dissolve the political "
    "bands which have connected them "
    "with another and to assume among "
    "the powers of the earth, the "
    "separate and equal station to which "
    "the Laws of Nature and of Nature's "
    "God entitle them, a decent respect "
    "to the opinions of mankind requires "
    "that they should declare the causes "
    "which impel them to the separation."
)
```

This also holds for imports, which can also be broken across lines using parens

```
from datasource import (load_data_1, load_data_2, load_data_3, load_data_4,
                        load_data_5)
```

Most languages have special syntax that allows you to break long commands across several lines. In Stata, `///` says "the following line is really part of this one. You can also change the delimiter to `;` using `#delimiter ;`.

3.2 Indentation

Whitespace is important in Python and messing up indentation can cause your code to crash.

- Do not use tabs to indent. Use 4 spaces. Your editor should have a setting for this, so that when you hit the tab key the editor inserts 4 spaces instead of a tab code (`\t`). (We do this because each editor displays `\t` tabs in different ways, but all editors display spaces the same.)
- When you break a line using parentheses, the next line should line up with the open parenthesis on the line above. If the open parenthesis is alone on that line, indent once.

```
# This is good
from datasource import (load_data_1, load_data_2, load_data_3, load_data_4,
                        load_data_5)
# This is bad
from datasource import (load_data_1, load_data_2, load_data_3, load_data_4,
    load_data_5)

# This is good
good_seconds_per_year = (
    60 *     # seconds per minute
    60 *     # minutes per hour
    24 *     # hours per day
    365      # days per year
)
# This is bad
bad_seconds_per_year = (
            60 *     # seconds per minute
            60 *     # minutes per hour
            24 *     # hours per day
            365      # days per year
)
```

3.3 Imports

- Imports go at the top of the file.
- *NEVER* import an entire package like this: `from numpy import *`.
- Separate and order imports like so

```
import os                                # Standard library (come with Python)
import re

import numpy as np                       # Third-party packages
import pandas as pd

from drillinginfo import clean_wells     # Packages developed by our team

from util.env import data_path           # Imports from *this* project
```

3.4 Don't use the same variable name for different things

Python (and Stata and R) are dynamically-typed languages, which means that they just figure out what kind of data a variable is, like a string, an integer or a decimal float. As a result, it's common to see people re-use variable names for different things. For example:

```
# Bad
to_clean = 'c:/data/raw_data.csv'
to_clean = pd.read_csv(to_clean)
```

This short example isn't too bad, but on larger scales it can be very confusing when a variable you thought was a DataFrame ends up being a string or vice versa. This type changing is also computationally slower. (It also messes up the `mypy` linter/type-checker.) A better solution is to be more explicit in naming your variables:

```
# Good
raw_data_path = 'c:/data/raw_data.csv'
raw_data = pd.read_csv(raw_data_path)
```

3.5 Make lowercase/uppercase meaningful

Variable and function names are all lowercase with underscores:

```
# Good
my_descriptive_name = 'abc'

def make_results_significant():
    return '***'

# Bad
NoCamelCase = True
```

Names of constants, especially project-wide parameters, are written in all capitals and almost always placed at the top of the script right after the imports:

```
MAX_TIME_PERIODS = 10
ROBUST_ERRORS = True
```

If you create any custom Python objects (classes), their name should begin with a capital and follow "Camel case" or "CapWords" convention:

```
class MyDataStorage(object):

    def __init__(self):
        pass
```

3.6 Other stuff

- Spaces around assignments: `x = 7` not `x=7` .

- No spaces around keyword variables: `function(arg1=y, arg2=3)`.
- Spaces after commas (just like in prose).
- Functions meant to be local (subroutines not meant to be imported by other scripts) should start with an underscore, e.g., `_drop_missings()`.
- Two lines between unrelated functions. One line between auxiliary functions:

```
def primary_func1():
    # Stuff

def _aux_to_1():
    # Stuff

def _another_aux_to_1():
    # Stuff


def primary_func2():
    # Stuff

def _aux_to_2():
    # Stuff
```

4 Example Script

```
"""
Task: read the words from the file `tmp.txt` and calculate each word's score
based on the "value" of its letters, where A=1, B=2, etc. Then calculate how
many words in the file are "triangle" numbers. A number T is a triangle number
if there is an integer n such that T = n * (n + 1) / 2.

Solution to Project Euler Problem 42. https://projecteuler.net/problem=42
"""
from string import ascii_uppercase

import numpy as np
import pandas as pd


LETTER_SCORE = {ascii_uppercase[x - 1]: x for x in range(1, 27)}


def word_score(word):
    """ Calculate total letter score for `word` """
    score = 0
    for letter in word:
        score += LETTER_SCORE[letter]
    return score


def is_triangle(x):
    """
    Use definition of triangle number and the quadratic formula to see if
    `x` is a triangle number.
    """
    positive_root = _positive_quadratic_root(x)
    return positive_root == int(positive_root)

def _positive_quadratic_root(x):
    a = 1
    b = 1
    c = -2 * x

    positive_root = (-1 * b + np.sqrt(b ** 2 - 4 * a * c)) / (2 * a)

    return positive_root


def read_and_prep_data():
    df = pd.read_csv('tmp.txt', header=None)
    df.columns = ['word']
    return df
```

```
if __name__ == '__main__':
    df = read_and_prep_data()
    df['word_score'] = df['word'].apply(word_score)
    df['is_triangle'] = df['word_score'].apply(is_triangle)

    print(df['is_triangle'].sum())
```

4.1 Example Bad Script

Each of the functions in the above script is referenced only once, so why write re-usable functions at all? Below is the same script re-written without functions or an if-main block. Is it easier or harder to follow what's going on in the code?

```
from string import ascii_uppercase
import numpy as np
import pandas as pd

df = pd.read_csv('tmp.txt', header=None)
df.columns = ['word']
df['word_score'] = 0
LETTER_SCORE = {ascii_uppercase[x - 1]: x for x in range(1, 27)}
for idx, word in df['word'].iteritems():
    for letter in word:
        df.loc[idx, 'word_score'] += LETTER_SCORE[letter]
df['positive_root'] = (-1 * 1 + np.sqrt(1 ** 2 - 4 * 1 * -2 * df['word_score'])) / (2 * 1)
df['is_triangle'] = (df['positive_root'] == df['positive_root'].astype(int))
print(df['is_triangle'].sum())
```

5 Follow Robust Development Practices

5.1 Build your code incrementally in a script

It's pretty common for data work in social science to look like this: You open Stata, R, Python, etc., and start poking around on the command line, interacting with the data until you get where you want. Then you use the command history (or your memory) to reconstruct what you did and put it in a script.

This is a bad way to work. Reconstructing exactly what you did is often difficult. At best, you're doing everything twice. Jupyter notebooks were designed in part to address this problem. However, as mentioned above, our work doesn't always play nice with Jupyter notebooks.

You can avoid these problems by writing your script incrementally. Start with a script that's empty except for the if-main block. Write the beginnings of your first function in the if-main block:

```
import pandas as pd

if __name__ == '__main__':
    # Prep data for regression
    df = pd.read_csv('data.csv')
    df = df[df['state'] == 'TX']    # Restrict to Texas
    # Do other cleaning
```

Now run your script using `%run` in IPython. Use some basic interactivity to figure out any bugs (e.g., maybe the state variable isn't called "state"). After you fix a problem in your script, `%run` it again. Keep doing this until you're done with the given task (e.g., prepping the data for a regression), then move that code into a function.

```
import pandas as pd

def prep_data_for_reg():
    """ Prep data for regression """
    df = pd.read_csv('data.csv')
    df = df[df['state'] == 'TX']    # Restrict to Texas
    # Do other cleaning...
    return df

if __name__ == '__main__':
    df = prep_data_for_reg()
```

Now you can start work on your next task in the if-main block in the same way. This is also a good time to commit your changes in Git if you haven't already done so.

When you're done, there should be a very simple (maybe empty) if-main block.

5.2 Don't run snippets of code in isolation

Spyder and Stata's do-file editor both have buttons that let you highlight a few lines of code and run only those. *Don't use them.* Run scripts from the beginning. It is too easy to spend ages "fixing" a piece of code by running it as a snippet only to find that when you run the script as a whole, you never actually fixed the problem (or introduced a new one).

5.3 Use the IPython terminal and a text editor

Jupyter and Spyder are great tools for analyzing data in Python. However, it is hard to keep track of your computing environemnt (which versions of packages are loaded, etc.) while using these tools. Our preferred solution is to use a robust text editor (Atom, Vim, Emacs) alongside a CMDer window running the IPython terminal. You can then use the `%run` command inside IPython to run your code.

And while Stata has a built-in text editor, it lacks most features you should look for in a good editor.

5.4 Manage your installed libraries using a virtual environment

Let's say you work on Project A for a while using Pandas 0.18 and Numpy 1.12 and some other Python packages. You submit Project A and start on Project B. Project B requires some features in Pandas 0.22, so you upgrade. While you're working on Project B, you get a revise and resubmit on Project A. (Contrats!) But when you try to run your code from Project A, it doesn't work with Pandas 0.22.

Virtual Environments were created to solve this problem. A virtual environment is a quarantined installation of Python that is independent from any other installations, including your default system Python. When you want to run code within a given virtual environment, you just activate it. The details of how you do that depend on the virtual environment software you use. We will use the Anaconda tool `conda`. There are lots of tutorials for `conda` online, e.g., here (https://towardsdatascience.com/getting-started-with-python-environments-using-conda-32e9f2779307).

You can avoid conflicting versions across projects by having a separate virtual environment for each one.

5.5 Use universal, immutable identifiers whenever possible

For example, use FIPS, not your own made up county ID's. Any time you introduce your own arbitrary identifiers, you run the risk of introducing inconsistencies between datasets.

For example, Stata's `egen group()` command is not guaranteed to be consistent across different scripts. You can use `group()` in two different scripts to create IDs, and the IDs will merge with each other such that `assert _merge == 3`, but there's no guarantee that the merge is correct.

Contact

Github (http://github.com/dmsul)

LinkedIn (https://www.linkedin.com/in/daniel-sullivan-42134687)

Twitter (https://twitter.com/Dan_M_Sullivan)

Google Scholar (https://scholar.google.com/citations?user=naLX3CIAAAAJ)

Email (mailto:sullivan.dm3@gmail.com)

Work Email (mailto:daniel.m.sullivan@jpmchase.com)

Back to top