

Coding habits for data scientists

Data engineering

Blog

By David Tan

Published: October 21, 2019

If you've tried your hand at machine learning or data science, you know that code can get **messy** [↗](#), quickly.

Typically, code to train ML models is written in Jupyter notebooks and it's full of (i) side effects (e.g. print statements, pretty-printed dataframes, data visualisations) and (ii) glue code without any abstraction, modularisation and automated tests. While this may be fine for notebooks targeted at teaching people about the machine learning process, in real projects it's a recipe for unmaintainable mess. The lack of good coding habits makes code hard to understand and consequently, modifying code becomes painful and error-prone. This makes it increasingly difficult for data scientists and developers to evolve their ML solutions.

In this article, we'll share techniques for identifying bad habits that add to complexity in code as well as habits that can help us partition complexity.

What contributes to complexity?

"One of the most important techniques for managing software complexity is to design systems so that

developers only need to face a small fraction of the overall complexity at any given time. "

- John Ousterhout [↗](#)

To tackle complexity, we must first know what it looks like. Something is complex when it's **composed of interconnected parts** [↗](#). Every time we write code in a way that adds another moving part, we increase complexity and add one more thing to hold in our head.

While we cannot — and should not try to — escape from the essential complexity of a problem, we often add unnecessary accidental complexity and unnecessary cognitive load through bad practices such as:

- **Not having abstractions.** When we write all code in a single Python notebook or script without abstracting it into functions or classes, we force the reader to read many lines of code and figure out the “how”, to find out what the code is doing.
- **Long functions that do multiple things.** This forces us to have to hold all the intermediate data transformations in our head, while working on one part of the function.
- **Not having unit tests.** When we refactor, the only way to ensure that we haven't broken anything is to restart the kernel and run the entire notebook(s). We're forced to take on the complexity of the whole codebase even though we just want to work on one small part of it.

Complexity is unavoidable, but it can be compartmentalized. In our homes, when we don't actively organise and rationalise where, why and how we place things, mess accumulates and what should have been a simple task (e.g. finding a key) becomes unnecessarily time-consuming and frustrating. The same applies to our codebase. New code is constantly being added for data cleaning, feature engineering, bug fixes, handling new data, and so on. Unless we vigilantly maintain our codebase and continuously refactor (and we can't refactor without unit tests), mess and complexity are guaranteed.

In the remainder of this article, we'll share some common bad habits that increase complexity and better habits that help to manage complexity:

- Keep code clean
- Use functions to abstract away complexity
- Smuggle code out of Jupyter notebooks as soon as possible
- Apply test driven development
- Make small and frequent commits

Habits for reducing complexity

Keep code clean

Unclean code adds to complexity by making code difficult to understand and modify. As a consequence, changing code to respond to business needs becomes increasingly difficult, and sometimes even impossible.

One such bad coding habit (or “code smell”) is dead code. Dead code is code which is executed but whose result is never used in any other computation. Dead code is yet another unrelated thing that developers have to hold in our head when coding. For example, compare these two code samples:

```
# bad example

df = get_data()

print(df)

# do_other_stuff()

# do_some_more_stuff()

df.head()

print(df.columns)

# do_so_much_stuff()

model = train_model(df)


# good example

df = get_data()

model = train_model(df)
```

Clean code practices have been written about extensively in [several languages](#), including [Python](#). We've adapted these "clean code" principles, and you can find them in this [clean-code-ml repo](#):

- Design ([code samples](#))
 - Don't expose your internals (Keep implementation details hidden)
- Disposables ([code samples](#))
 - Remove dead code
 - Avoid print statements (even glorified print statements such as **df.head()**, **df.describe()**, **df.plot()**)
- Variables ([code samples](#))
 - Variable names should reveal intent
- Functions ([code samples](#))
 - Use functions to keep code "DRY" (Don't Repeat Yourself)
 - Functions should do one thing

Use functions to abstract away complexity

Functions simplify our code by abstracting away complicated implementation details and replacing them with a simpler representation — its name.

Imagine you're in a restaurant. You're given a menu. Instead of telling you the name of the dishes, this menu spells out the recipe for each dish. For example, one such dish is:

Step 1. In a large pot, heat up the oil. Add carrots, onions and celery; stir until onion is soft. Add herbs and garlic and cook for a few more minutes.

Step 2. Add in lentils, add tomatoes and water. Bring soup to a boil and then reduce heat to let it simmer for 30 minutes. Add spinach and cook until spinach is soft. Finally, season with vinegar, salt and pepper.

It would have been easier for us if the menu hid all the steps in the recipe (i.e. the implementation details) and instead gave us the name of the dish (an interface, an abstraction of the dish). (Answer: that was lentil soup).

To illustrate this point, here's a code sample from a notebook in Kaggle's Titanic competition before and after refactoring.

```
# bad example
pd.qcut(df['Fare'], q=4, retbins=True)[1] # returns array([0.,
7.8958, 14.4542, 31.275, 512.3292])

df.loc[ df['Fare'] <= 7.90, 'Fare'] = 0
df.loc[(df['Fare'] > 7.90) & (df['Fare'] <= 14.454), 'Fare'] = 1
df.loc[(df['Fare'] > 14.454) & (df['Fare'] <= 31), 'Fare'] = 2
df.loc[ df['Fare'] > 31, 'Fare'] = 3
df['Fare'] = df['Fare'].astype(int)
df['FareBand'] = df['Fare']

# good example (after refactoring into functions)
df['FareBand'] = categorize_column(df['Fare'], num_bins=4)
```

What did we gain by abstracting away the complexity into functions?

- **Readability.** We just have to read the interface (i.e. `categorize_column()`) to know what it's doing. We didn't have to read each line or search the internet for things that we don't understand (e.g. `pd.qcut`). If I still didn't understand what the function did based on its name and usage, I can read its unit tests or its definition.
- **Testability.** Because it's now a function, we can easily write a unit test for it. If we accidentally change its behaviour, the unit tests will fail and give us feedback within milliseconds.
- **Reusability.** To repeat the same transformation on any column (e.g. 'Age', or 'Income'), we just need one line (not seven lines) of code.

When we refactor to functions, our entire notebook can be simplified and made more elegant:

```
# bad example

# good example
df = impute_nans(df, categorical_columns=['Embarked'],
                  Continuous_columns=['Fare', 'Age'])
df = add_derived_title(df)
df = encode_title(df)
```


```
df = add_is_alone_column(df)
df = add_categorical_columns(df)
X, y = split_features_and_labels(df)

# an even better example. Notice how this reads like a story
prepare_data = compose(impute_nans,
                        add_derived_title,
                        encode_title,
                        add_is_alone_column,
                        add_categorical_columns,
                        split_features_and_labels)

X, y = prepare_data(df)
```

Our mental overhead is now drastically reduced. We're no longer forced to process many many lines of implementation details to understand the entire flow. Instead, the abstractions (i.e. functions) abstract away the complexity and tell us what they do, and save us from having to spend mental effort figuring out how they do it.

Smuggle code out of Jupyter notebooks as soon as possible

In interior design, there is a concept (the "[Law of Flat Surfaces](#) ) which states that "any flat surface within a home or office tends to accumulate clutter." Jupyter notebooks are the flat surface of the ML world.



Sure, Jupyter notebooks are great for quick prototyping. But it's where we tend to put many things — glue code, print statements, glorified print statements (**df.describe()** or **df.plot()**), unused import statements and even stack traces. Despite our best intentions, so long as the notebooks are there, mess tends to accumulate.

Notebooks are useful because they give us fast feedback, and that's often what we want when we're given a new dataset and a new problem. However, the longer the notebooks become, the harder it is to get feedback on whether our changes are working.

In contrast, if we had extracted our code into functions and Python modules and if we have unit tests, the test runner will give us feedback on our changes in a matter of seconds, even when there are hundreds of functions.

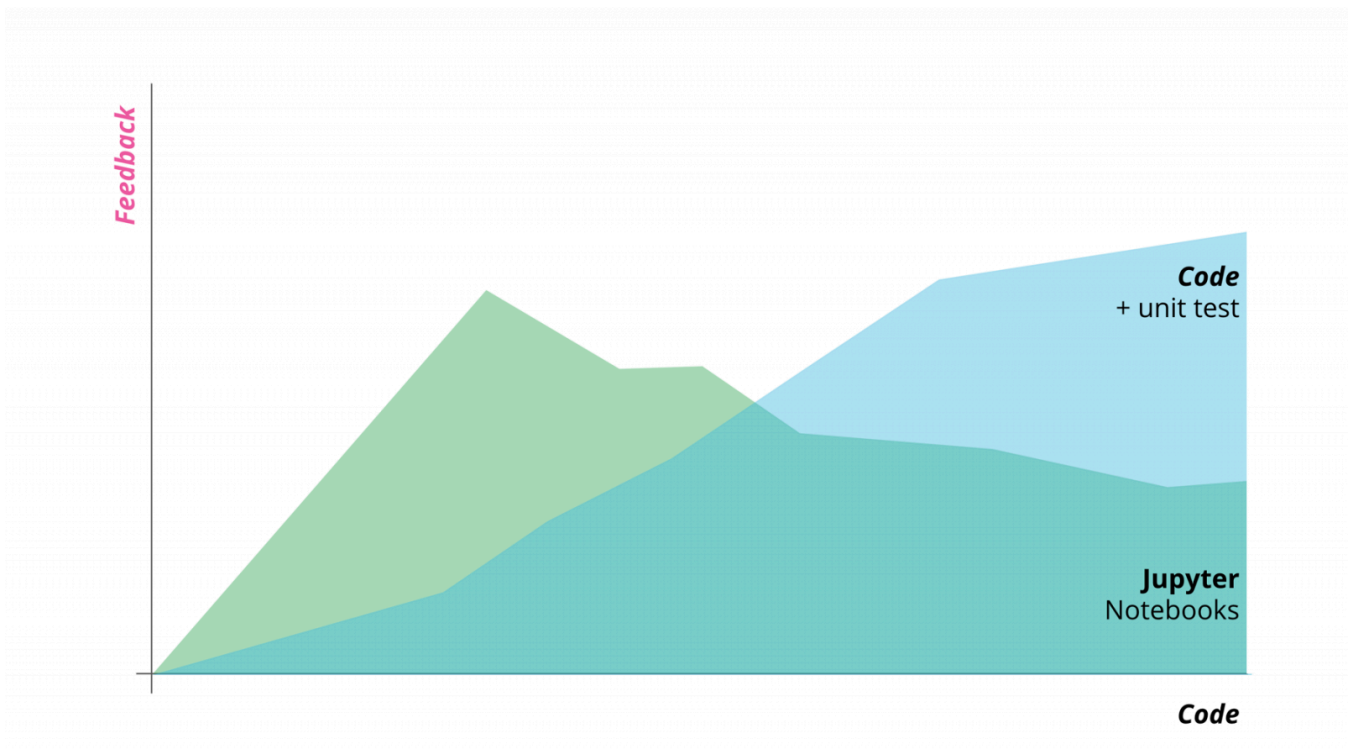


Figure 1: The more code we have, the harder it is for notebooks to give us fast feedback on whether everything is working as expected

Hence, our goal is to move code out of notebooks into Python modules and packages as early as possible. That way they can rest within the safe confines of unit tests and domain boundaries. This will help to manage complexity by providing a structure for organizing code and tests logically and make it easier for us to evolve our ML solution.

So, how do we move code out of Jupyter notebooks?

Assuming you already have your code in a Jupyter notebook, you can follow this process:

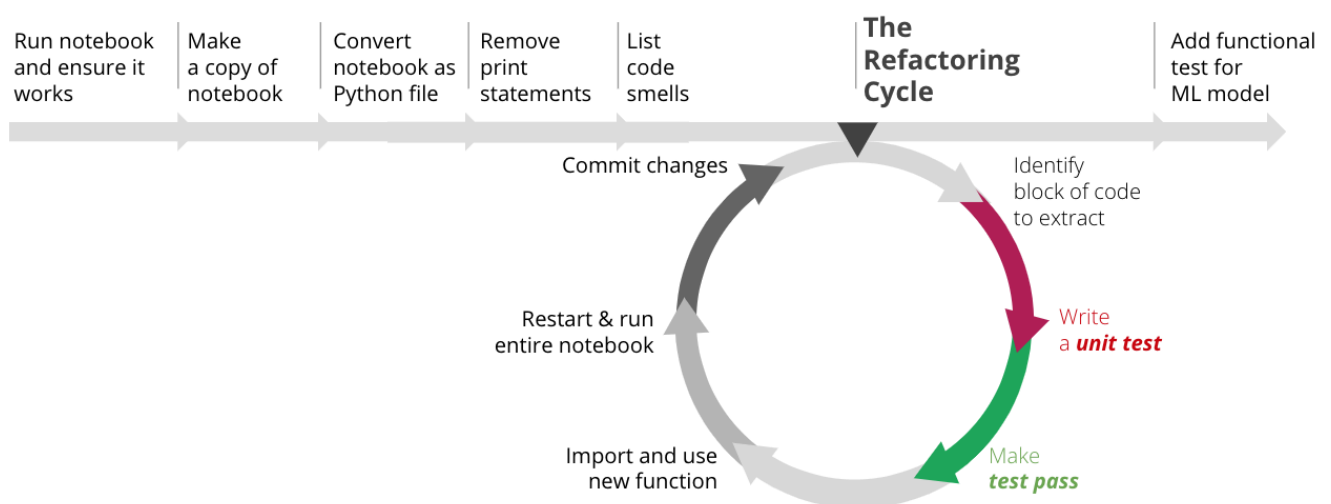


Figure 2: How to refactor a Jupyter notebook [↗](#)

The details of each step in this process (e.g. how to run tests in watch mode) can be

found in the [clean-code-ml repo](#).

Apply test-driven development

So far, we've talked about writing tests after the code is already written in the notebook. This recommendation isn't ideal, but it's still far better than not having unit tests.

There is a myth that we cannot apply test-driven development (**TDD**) to machine learning projects. To us, this is simply untrue. In any machine learning project, most of the code is concerned with data transformations (e.g. data cleaning, feature engineering) and a small part of the codebase is actual machine learning. Such data transformations can be written as pure functions that return the same output for the same input, and as such, we can apply TDD and reap its benefits. For example, TDD can help us break down big and complex data transformations into smaller bite-size problems that we can fit in our head, one at a time.

As for testing that the actual machine learning part of the code works as we expect it to, we can write functional tests to assert that the metrics of the model (e.g. accuracy, precision, etc) are above our expected threshold. In other words, these tests assert that the model functions according to our expectations (hence the name, functional test). Here's an [example](#) of such a test:

```
import unittest
from sklearn.metrics import precision_score, recall_score

from src.train import prepare_data_and_train_model

class TestModelMetrics(unittest.TestCase):
    def test_model_precision_score_should_be_above_threshold(self):
        model, X_test, Y_test = prepare_data_and_train_model()
        Y_pred = model.predict(X_test)

        precision = precision_score(Y_test, Y_pred)

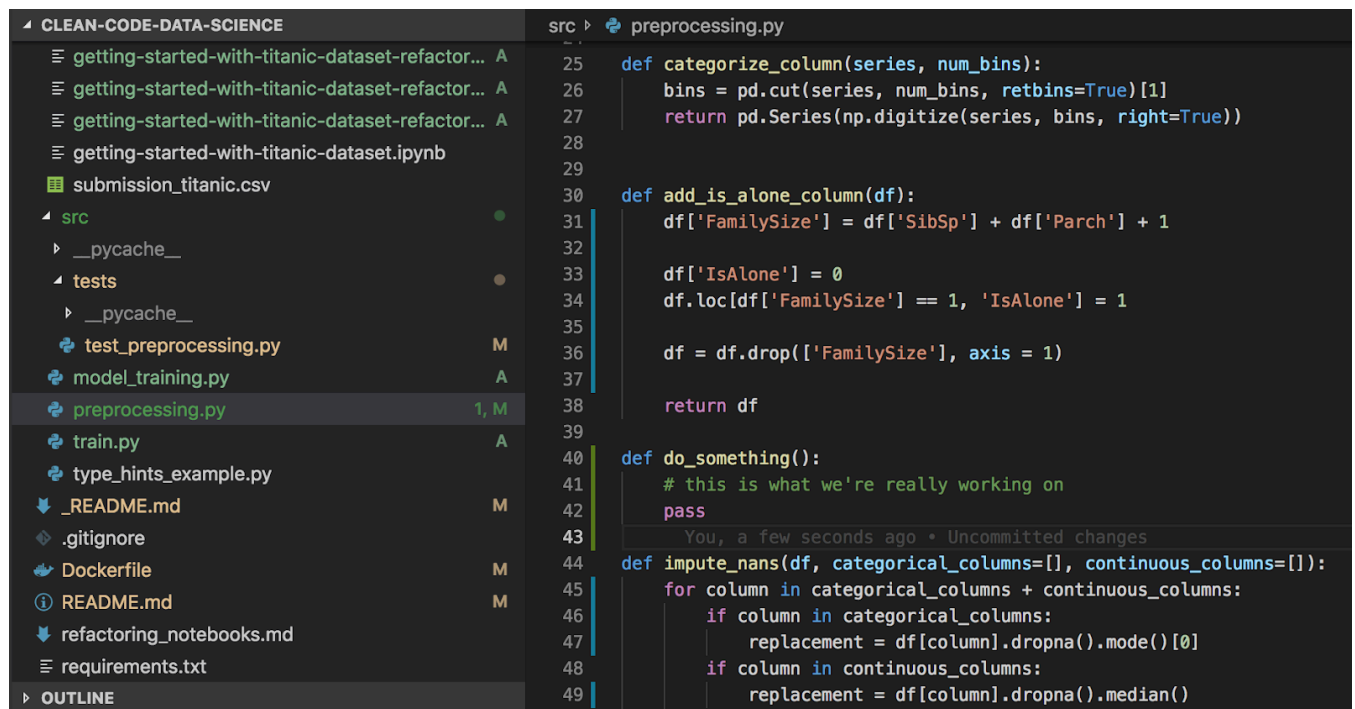
        self.assertGreaterEqual(precision, 0.7)
```

Make small and frequent commits

When we don't make small and frequent commits, we increase mental overhead. While we're working on this problem, the changes for earlier ones are still shown as

uncommitted. This distracts us visually and subconsciously; it makes it harder for us to focus on the current problem.

For example, look at the first and second images below. Can you find out which function we're working on? Which image gave you an easier time?



```
def categorize_column(series, num_bins):
    bins = pd.cut(series, num_bins, retbins=True)[1]
    return pd.Series(np.digitize(series, bins, right=True))

def add_is_alone_column(df):
    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

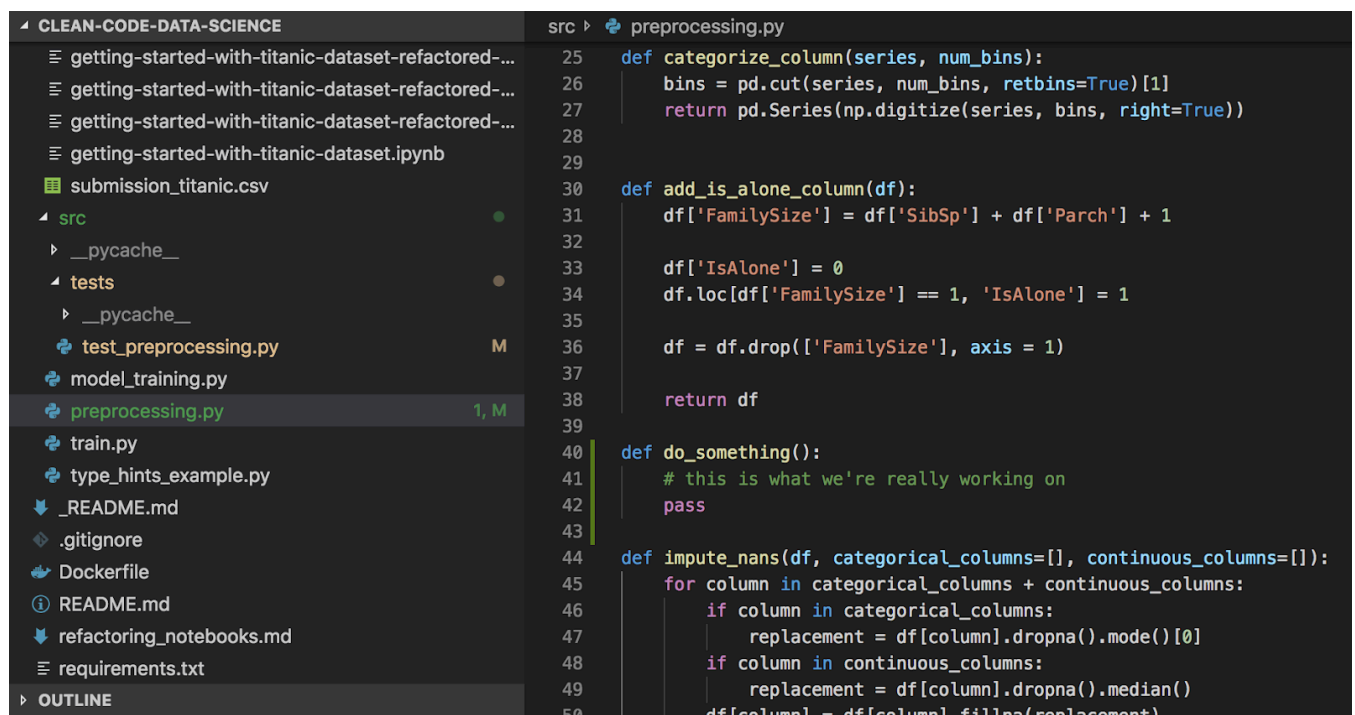
    df['IsAlone'] = 0
    df.loc[df['FamilySize'] == 1, 'IsAlone'] = 1

    df = df.drop(['FamilySize'], axis = 1)

    return df

def do_something():
    # this is what we're really working on
    pass

def impute_nans(df, categorical_columns=[], continuous_columns=[]):
    for column in categorical_columns + continuous_columns:
        if column in categorical_columns:
            replacement = df[column].dropna().mode()[0]
        if column in continuous_columns:
            replacement = df[column].dropna().median()
```



```
def categorize_column(series, num_bins):
    bins = pd.cut(series, num_bins, retbins=True)[1]
    return pd.Series(np.digitize(series, bins, right=True))

def add_is_alone_column(df):
    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

    df['IsAlone'] = 0
    df.loc[df['FamilySize'] == 1, 'IsAlone'] = 1

    df = df.drop(['FamilySize'], axis = 1)

    return df

def do_something():
    # this is what we're really working on
    pass

def impute_nans(df, categorical_columns=[], continuous_columns=[]):
    for column in categorical_columns + continuous_columns:
        if column in categorical_columns:
            replacement = df[column].dropna().mode()[0]
        if column in continuous_columns:
            replacement = df[column].dropna().median()
        df[column] = df[column].fillna(replacement)
```

When we make small and frequent commits, we get the following benefits:

- Reduced visual distractions and cognitive load.
- We needn't worry about accidentally breaking working code if it's already been committed.

- In addition to [red-green-refactor](#) ↗, we can also [red-red-red-revert](#) ↗. If we were to inadvertently break something, we can easily fall back checkout to the latest commit, and try again. This saves us from wasting time undoing problems that we accidentally created when we were trying to solve the essential problem.

So, how small of a commit is small enough? Try to commit when there is a single group of logically related changes and passing tests. One technique is to look out for the word “and” in our commit message, e.g. “Add exploratory data analysis and split sentences into tokens and refactor model training code”. Each of these three changes could be split up into three logical commits. In this situation, you can use [git add --patch](#) ↗ to stage code in smaller batches to be committed.

Conclusion

“I'm not a great programmer; I'm just a good programmer with great habits. ”

Kent Beck

Pioneer of Extreme Programming and xUnit testing frameworks

These are habits that have helped us manage complexity in machine learning and data science projects. We hope it helps you become more agile and productive in your data projects as well.

If you're interested in examples of how organisations can adopt continuous delivery practices to help machine learning practitioners and projects be agile, see [here](#) and [here](#).

Disclaimer: The statements and opinions expressed in this article are those of the author(s) and do not necessarily reflect the positions of Thoughtworks.

Keep up to date with our latest insights

[Explore our content](#)

Connect with us



© 2024 Thoughtworks, Inc.