Software_Engineering_Practices_1

July 1, 2020

1 Clean and Modular Code

- **PRODUCTION CODE**: software running on production servers to handle live users and data of the intended audience. Note this is different from production quality code, which describes code that meets expectations in reliability, efficiency, etc., for production. Ideally, all code in production meets these expectations, but this is not always the case.
- **CLEAN**: readable, simple, and concise. A characteristic of production quality code that is crucial for collaboration and maintainability in software development.
- **MODULAR**: logically broken up into functions and modules. Also an important characteristic of production quality code that makes your code more organized, efficient, and reusable.
- **MODULE**: a file. Modules allow code to be reused by encapsulating them into files that can be imported into other files.

2 Refactoring Code

- **REFACTORING**: restructuring your code to improve its internal structure, without changing its external functionality. This gives you a chance to clean and modularize your program after you've got it working.
- Since it isn't easy to write your best code while you're still trying to just get it working, allocating time to do this is essential to producing high quality code. Despite the initial time and effort required, this really pays off by speeding up your development time in the long run.
- You become a much stronger programmer when you're constantly looking to improve your code. The more you refactor, the easier it will be to structure and write good code the first time.

3 Writing Clean Code: Meaningful Names

Tip: Use meaningful names

- **Be descriptive and imply type** - E.g. for booleans, you can prefix with `is_` or `has_` to make it clear it is a condition. You can also use part of speech to imply types, like verbs for functions and nouns for variables.

- **Be consistent but clearly differentiate** - E.g. `age_list` and `age` is easier to differentiate than `ages` and `age`.
- **Avoid abbreviations and especially single letters** - (Exception: counters and common math variables) Choosing when these exceptions can be made can be determined based on the audience for your code. If you work with other data scientists, certain variables may be common knowledge. While if you work with full stack engineers, it might be necessary to provide more descriptive names in these cases as well.
- **Long names != descriptive names** - You should be descriptive, but only with relevant information. E.g. good functions names describe what they do well without including details about implementation or highly specific uses.

Try testing how effective your names are by asking a fellow programmer to guess the purpose of a function or variable based on its name, without looking at your code. Coming up with meaningful names often requires effort to get right.

4 Writing Clean Code: Nice Whitespace

Tip: Use whitespace properly

- Organize your code with consistent indentation - the standard is to use 4 spaces for each indent. You can make this a default in your text editor.
- Separate sections with blank lines to keep your code well organized and readable.
- Try to limit your lines to around 79 characters, which is the guideline given in the PEP 8 style guide. In many good text editors, there is a setting to display a subtle line that indicates where the 79 character limit is.

For more guidelines, check out the PEP 8 guidelines for code layout.

5 Writing Modular Code

Tip: DRY (Don't Repeat Yourself)

Don't repeat yourself! Modularization allows you to reuse parts of your code. Generalize and consolidate repeated code in functions or loops.

Tip: Abstract out logic to improve readability

Abstracting out code into a function not only makes it less repetitive, but also improves readability with descriptive function names. Although your code can become more readable when you abstract out logic into functions, it is possible to over-engineer this and have way too many modules, so use your judgement.

Tip: Minimize the number of entities (functions, classes, modules, etc.)

There are tradeoffs to having function calls instead of inline logic. If you have broken up your code into an unnecessary amount of functions and modules, you'll have to jump around everywhere if

you want to view the implementation details for something that may be too small to be worth it. Creating more modules doesn't necessarily result in effective modularization.

Tip: Functions should do one thing

Each function you write should be focused on doing one thing. If a function is doing multiple things, it becomes more difficult to generalize and reuse. Generally, if there's an "and" in your function name, consider refactoring.

Tip: Arbitrary variable names can be more effective in certain functions

Arbitrary variable names in general functions can actually make the code more readable.

Tip: Try to use fewer than three arguments per function

Try to use no more than three arguments when possible. This is not a hard rule and there are times it is more appropriate to use many parameters. But in many cases, it's more effective to use fewer arguments. Remember we are modularizing to simplify our code and make it more efficient to work with. If your function has a lot of parameters, you may want to rethink how you are splitting this up.

6 Quiz: Refactoring - Wine Quality

In this exercise, we will write refactored code that analyzes a wine quality dataset taken from the UCI Machine Learning Repository here. Each row contains data on a wine sample, including several physicochemical properties gathered from tests, as well as a quality rating evaluated by wine experts.

[1]:
```
import pandas as pd
df = pd.read_csv('winequality-red.csv', sep=';')
df.head()
```

[1]:
```
   fixed acidity  volatile acidity  citric acid  …  sulphates  alcohol
quality
0            7.4              0.70         0.00  …       0.56      9.4
5
1            7.8              0.88         0.00  …       0.68      9.8
5
2            7.8              0.76         0.04  …       0.65      9.8
5
3           11.2              0.28         0.56  …       0.58      9.8
6
4            7.4              0.70         0.00  …       0.56      9.4
5

[5 rows x 12 columns]
```

6.1 Renaming columns

You want to replace the spaces in the column labels with underscores to be able to reference columns with dot notation. Here's one way you could've done it.

[2]:
```
################ One way to rename the column ###############
#############################################################

# new_df = df.rename(columns={'fixed acidity': 'fixed_acidity',
#                             'volatile acidity': 'volatile_acidity',
#                             'citric acid': 'citric_acid',
#                             'residual sugar': 'residual_sugar',
#                             'free sulfur dioxide': 'free_sulfur_dioxide',
#                             'total sulfur dioxide': 'total_sulfur_dioxide'
#                            })
# new_df.head()

################ Another way to rename the column ############
#############################################################

# labels = list(df.columns)
# labels[0] = labels[0].replace(' ', '_')
# labels[1] = labels[1].replace(' ', '_')
# labels[2] = labels[2].replace(' ', '_')
# labels[3] = labels[3].replace(' ', '_')
# labels[5] = labels[5].replace(' ', '_')
# labels[6] = labels[6].replace(' ', '_')
# df.columns = labels


df.columns = ['_'.join(column.split(' ')) for column in df.columns]
df.head()
```

[2]:
```
   fixed_acidity  volatile_acidity  citric_acid  …  sulphates  alcohol
quality
0            7.4              0.70         0.00  …       0.56      9.4
5
1            7.8              0.88         0.00  …       0.68      9.8
5
2            7.8              0.76         0.04  …       0.65      9.8
5
3           11.2              0.28         0.56  …       0.58      9.8
6
4            7.4              0.70         0.00  …       0.56      9.4
5

[5 rows x 12 columns]
```

6.2 Analyzing Features

Now that your columns are ready, you want to see how different features of this dataset relate to the quality rating of the wine. A very simple way you could do this is by observing the mean quality rating for the top and bottom half of each feature. The code below does this for four features. It looks pretty repetitive right now. Can you make this more concise?

You might challenge yourself to figure out how to make this code more efficient! But you don't need to worry too much about efficiency right now - we will cover that more in the next section.

[3]:
```
############### One way to analyse the feature ############
###########################################################

# median_alcohol = df.alcohol.median()
# for i, alcohol in enumerate(df.alcohol):
#     if alcohol >= median_alcohol:
#         df.loc[i, 'alcohol'] = 'high'
#     else:
#         df.loc[i, 'alcohol'] = 'low'
# df.groupby('alcohol').quality.mean()


def numeric_to_buckets(df, column_name):
    median = df[column_name].median()
    for i, val in enumerate(df[column_name]):
        if val >= median:
            df.loc[i, column_name] = 'high'
        else:
            df.loc[i, column_name] = 'low'
```

[4]:
```
for feature in df.columns[:-1]:
    numeric_to_buckets(df, feature)
    print(df.groupby(feature).quality.mean(), '\n')
```

```
fixed_acidity
high    5.726061
low     5.540052
Name: quality, dtype: float64

volatile_acidity
high    5.392157
low     5.890166
Name: quality, dtype: float64

citric_acid
high    5.822360
low     5.447103
Name: quality, dtype: float64
```

```
residual_sugar
high    5.665880
low     5.602394
Name: quality, dtype: float64

chlorides
high    5.507194
low     5.776471
Name: quality, dtype: float64

free_sulfur_dioxide
high    5.595268
low     5.677136
Name: quality, dtype: float64

total_sulfur_dioxide
high    5.522981
low     5.750630
Name: quality, dtype: float64

density
high    5.540574
low     5.731830
Name: quality, dtype: float64

pH
high    5.598039
low     5.675607
Name: quality, dtype: float64

sulphates
high    5.898917
low     5.351562
Name: quality, dtype: float64

alcohol
high    5.958904
low     5.310302
Name: quality, dtype: float64
```

7 Efficient Code

Knowing how to write code that runs efficiently is another essential skill in software development. Optimizing code to be more efficient can mean making it:

- Execute faster
- Take up less space in memory/storage

The project you're working on would determine which of these is more important to optimize for your company or product. When we are performing lots of different transformations on large amounts of data, this can make orders of magnitudes of difference in performance.

For more guidelines, check What makes sets faster than lists

8 Quiz: Optimizing - Common Books

Here's the code your coworker wrote to find the common book ids in `books_published_last_two_years.txt` and `all_coding_books.txt` to obtain a list of recent coding books.

[5]:
```
import time
import pandas as pd
import numpy as np
```

[6]:
```
with open('books_published_last_two_years.txt') as f:
    recent_books = f.read().split('\n')

with open('all_coding_books.txt') as f:
    coding_books = f.read().split('\n')
```

[7]:
```
start = time.time()
recent_coding_books = []

for book in recent_books:
    if book in coding_books:
        recent_coding_books.append(book)

print(len(recent_coding_books))
print('Duration: {} seconds'.format(time.time() - start))
```

```
96
Duration: 12.46284532546997 seconds
```

Tip #1: Use vector operations over loops when possible

Use numpy's `intersect1d` method to get the intersection of the `recent_books` and `coding_books` arrays.

[8]:
```
start = time.time()
recent_coding_books =  np.intersect1d(recent_books, coding_books)
print(len(recent_coding_books))
print('Duration: {} seconds'.format(time.time() - start))
```

```
96
Duration: 0.03615760803222656 seconds
```

Tip #2: Know your data structures and which methods are faster

Use the set's `intersection` method to get the common elements in `recent_books` and `coding_books`.

[9]:
```
start = time.time()
recent_coding_books =  set(recent_books).intersection(coding_books)
print(len(recent_coding_books))
print('Duration: {} seconds'.format(time.time() - start))
```

```
96
Duration: 0.008502721786499023 seconds
```

9 Quiz: Optimizing - Holiday Gifts

In the last example, you learned that using vectorized operations and more efficient data structures can optimize your code. Let's use these tips for one more example.

Say your online gift store has one million users that each listed a gift on a wish list. You have the prices for each of these gifts stored in `gift_costs.txt`. For the holidays, you're going to give each customer their wish list gift for free if it is under 25 dollars. Now, you want to calculate the total cost of all gifts under 25 dollars to see how much you'd spend on free gifts. Here's one way you could've done it.

[10]:
```
import time
import numpy as np
```

[11]:
```
with open('gift_costs.txt') as f:
    gift_costs = f.read().split('\n')

gift_costs = np.array(gift_costs).astype(int)  # convert string to int
```

[12]:
```
start = time.time()

total_price = 0
for cost in gift_costs:
    if cost < 25:
        total_price += cost * 1.08  # add cost after tax

print(total_price)
print('Duration: {} seconds'.format(time.time() - start))
```

```
32765421.23999867
Duration: 8.83368992805481 seconds
```

Here you iterate through each cost in the list, and check if it's less than 25. If so, you add the cost to the total price after tax. This works, but there is a much faster way to do this. Can you refactor this to run under half a second?

Refactor Code

Hint: Using numpy makes it very easy to select all the elements in an array that meet a certain condition, and then perform operations on them together all at once. You can them find the sum of what those values end up being.

[13]:
```
start = time.time()
total_price = sum(gift_costs[gift_costs < 25]*1.08)

print(total_price)
print('Duration: {} seconds'.format(time.time() - start))
```

```
32765421.23999867
Duration: 0.5967502593994141 seconds
```

10 Documentation

- **DOCUMENTATION**: additional text or illustrated information that comes with or is embedded in the code of software.
- Helpful for clarifying complex parts of code, making your code easier to navigate, and quickly conveying how and why different components of your program are used.
- Several types of documentation can be added at different levels of your program:
- In-line Comments - line level
- Docstrings - module and function level
- Project Documentation - project level like README file or a document

10.1 In-line comments

- In-line comments are text following hash symbols throughout your code. They are used to explain parts of your code, and really help future contributors understand your work.
- One way comments are used is to document the major steps of complex code to help readers follow. Then, you may not have to understand the code to follow what it does. However, others would argue that this is using comments to justify bad code, and that if code requires comments to follow, it is a sign refactoring is needed.
- Comments are valuable for explaining where code cannot. For example, the history behind why a certain method was implemented a specific way. Sometimes an unconventional or seemingly arbitrary approach may be applied because of some obscure external variable causing side effects. These things are difficult to explain with code.

10.2 Docstrings

Docstring, or documentation strings, are valuable pieces of documentation that explain the functionality of any function or module in your code. Ideally, each of your functions should always have a docstring.

Docstrings are surrounded by triple quotes. The first line of the docstring is a brief explanation of the function's purpose.

One line docstring

[14]:
```
def population_density(population, land_area):
    """Calculate the population density of an area."""
    return population / land_area
```

If you think that the function is complicated enough to warrant a longer description, you can add a more thorough paragraph after the one line summary.

Multi line docstring

[15]:
```
def population_density(population, land_area):
    """Calculate the population density of an area.

    Args:
    population: int. The population of the area
    land_area: int or float. This function is unit-agnostic, if you pass in␣
 ↪values in terms of square km or square miles the function will return a␣
 ↪density in those units.

    Returns:
    population_density: population/land_area. The population density of a
    particular area.
    """
    return population / land_area
```

The next element of a docstring is an explanation of the function's arguments. Here you list the arguments, state their purpose, and state what types the arguments should be. Finally it is common to provide some description of the output of the function. Every piece of the docstring is optional; however, doc strings are a part of good coding practice.

Resources:

- PEP 257 - Docstring Conventions
- NumPy Docstring Guide

10.3 Project Documentation

Project documentation is essential for getting others to understand why and how your code is relevant to them, whether they are potentials users of your project or developers who may contribute

to your code. A great first step in project documentation is your README file. It will often be the first interaction most users will have with your project.

Whether it's an application or a package, your project should absolutely come with a README file. At a minimum, this should explain what it does, list its dependencies, and provide sufficiently detailed instructions on how to use it. You want to make it as simple as possible for others to understand the purpose of your project, and quickly get something working.

Translating all your ideas and thoughts formally on paper can be a little difficult, but you'll get better over time and makes a significant difference in helping others realize the value of your project. Writing this documentation can also help you improve the design of your code, as you're forced to think through your design decisions more thoroughly. This also allows future contributors to know how to follow your original intentions.

Here are a few READMEs from some popular projects:

- Bootstrap
- Scikit-learn
- Stack Overflow Blog

11 Version Control in Data Science

11.1 Scenario #1

Scenario #1

STEP 1: You have a local version of this repository on your laptop, and to get the latest stable version, you pull from the develop branch.

- Switch to the develop branch

  ```
  git checkout develop
  ```

- Pull latest changes in the develop branch

  ```
  git pull
  ```

STEP 2: When you start working on this demographic feature, you create a new branch for this called demographic, and start working on your code in this branch.

- Create and switch to new branch called demographic from develop branch

  ```
  git checkout -b demographic
  ```

- Work on this new feature and commit as you go

  ```
  git commit -m 'added gender recommendations'
  ```

  ```
  git commit -m 'added location specific recommendations'
  ```

  ```
  ...
  ```

STEP 3: However, in the middle of your work, you need to work on another feature. So you commit your changes on this demographic branch, and switch back to the develop branch.

- Commit changes before switching

  ```
  git commit -m 'refactored demographic gender and location recommendations'
  ```

- Switch to the develop branch

  ```
  git checkout develop
  ```

STEP 4: From this stable develop branch, you create another branch for a new feature called friend_groups.

- Create and switch to new branch called friend_groups from develop branch

  ```
  git checkout -b friend_groups
  ```

STEP 5: After you finish your work on the friend_groups branch, you commit your changes, switch back to the development branch, merge it back to the develop branch, and push this to the remote repository's develop branch.

- Commit changes before switching

  ```
  git commit -m 'finalized friend_groups recommendations'
  ```

- Switch to the develop branch

  ```
  git checkout develop
  ```

- Merge friend_groups branch to develop

  ```
  git merge --no-ff friends_groups
  ```

- Push to remote repository

  ```
  git push origin develop
  ```

STEP 6: Now, you can switch back to the demographic branch to continue your progress on that feature.

- Switch to the demographic branch

  ```
  git checkout demographic
  ```

11.2 Scenario #2

Scenario #2

Step 1: You check your commit history, seeing messages of the changes you made and how well it performed.

- View log history

  ```
  git log
  ```

Step 2: The model at this commit seemed to score the highest, so you decide to take a look.

- Checkout a commit

  ```
  git checkout bc90f2cbc9dc4e802b46e7a153aa106dc9a88560
  ```

After inspecting your code, you realize what modifications made this perform well, and use those for your model.

Step 3: Now, you're pretty confident merging this back into the development branch, and pushing the updated recommendation engine.

- Switch to develop branch

  ```
  git checkout develop
  ```

- Merge friend_groups branch to develop

  ```
  git merge --no-ff friend_groups
  ```

- Push changes to remote repository

  ```
  git push origin develop
  ```

11.3 Resources

There's a great article on a successful git branching strategy that you should really read here.

Note on Merge Conflicts

For the most part, git makes merging changes between branches really simple. However, there are some cases where git will be confused on how to combine two changes, and asks you for help. This is called a merge conflict.

Mostly commonly, this happens when two branches modify the same file.

For example, in this situation, let's say I deleted a line that Andrew modified on his branch. Git wouldn't know whether to delete the line or modify it. Here, you need to tell git which change to take, and some tools even allow you to edit the change manually. If it isn't straightforward, you may have to consult with the developer of the other branch to handle a merge conflict.

You can learn more about merge conflicts and methods to handle them here.

Some other useful resource are:

- How to Version Control Your Production Machine Learning Models
- Versioning Data Science