NOTES 4

Welcome to software engineering practices, part 2

In part 2 of software engineering practices, you'll learn about the following practices of software engineering and how they apply in data science.

- Testing
- Logging
- Code reviews

Testing

Testing your code is essential before deployment. It helps you catch errors and faulty conclusions before they make any major impact. Today, employers are looking for data scientists with the skills to properly prepare their code for an industry setting, which includes testing their code.

Testing and data science

- Problems that could occur in data science aren’t always easily detectable; you might have values being encoded incorrectly, features being used inappropriately, or unexpected data breaking assumptions.
- To catch these errors, you have to check for the quality and accuracy of your *analysis* in addition to the quality of your *code*. Proper testing is necessary to avoid unexpected surprises and have confidence in your results.

- Test-driven development (TDD): A development process in which you write tests for tasks before you even write the code to implement those tasks.
- Unit test: A type of test that covers a “unit” of code—usually a single function—independently from the rest of the program.

Resources

- Four Ways Data Science Goes Wrong and How Test-Driven Data Analysis Can Help: Blog Post
- Ned Batchelder: Getting Started Testing: Slide Deck and Presentation Video

Unit tests

We want to test our functions in a way that is repeatable and automated. Ideally, we'd run a test program that runs all our unit tests and cleanly lets us know which ones failed and which ones succeeded. Fortunately, there are great tools available in Python that we can use to create effective unit tests!

Unit test advantages and disadvantages

The advantage of unit tests is that they are isolated from the rest of your program, and thus, no dependencies are involved. They don't require access to databases, APIs, or other external sources of information. However, passing unit tests isn’t always enough to prove that our program is working successfully. To show that all the parts of our program work with each other properly, communicating and transferring data between them correctly, we use integration tests. In this lesson, we'll focus on unit tests; however, when you start building larger programs, you will want to use integration tests as well.

To learn more about integration testing and how integration tests relate to unit tests, see Integration Testing. That article contains other very useful links as well.

Unit testing tools

To install `pytest`, run `pip install -U pytest` in your terminal. You can see more information on getting started here.

- Create a test file starting with `test_`.
- Define unit test functions that start with `test_` inside the test file.
- Enter `pytest` into your terminal in the directory of your test file and it detects these tests for you.

`test_` is the default; if you wish to change this, you can learn how in this `pytest` configuration.

In the test output, periods represent successful unit tests and Fs represent failed unit tests. Since all you see is which test functions failed, it's wise to have only one `assert` statement per test. Otherwise, you won't know exactly how many tests failed or which tests failed.

Your test won't be stopped by failed `assert` statements, but it will stop if you have syntax errors.

Test-driven development and data science

- *Test-driven development*: Writing tests before you write the code that's being tested. Your test fails at first, and you know you've finished implementing a task when the test passes.
- Tests can check for different scenarios and edge cases before you even start to write your function. When start implementing your function, you can run the test to get immediate feedback on whether it works or not as you tweak your function.
- When refactoring or adding to your code, tests help you rest assured that the rest of your code didn't break while you were making those changes. Tests also helps ensure that your function behavior is repeatable, regardless of external parameters such as hardware and time.

Test-driven
development for data science is relatively new and is experiencing a lot of experimentation and breakthroughs. You can learn more about it by exploring the following resources.

- Data Science TDD
- TDD for Data Science
- TDD is Essential for Good Data Science Here's Why
- Testing Your Code (general python TDD)

Logging

Logging is valuable for understanding the events that occur while running your program. For example, if you run your model overnight and the results the following morning are not what you expect, log messages can help you understand more about the context in those results occurred. Let's learn about the qualities that make a log message effective.

Log messages

Logging is the process of recording messages to describe events that have occurred while running your software. Let's take a look at a few examples, and learn tips for writing good log messages.

Tip: Be professional and clear

```
Bad: Hmmm... this isn't working???
Bad: idk.... :(
Good: Couldn't parse file.
```

Tip: Be concise and use normal capitalization

```
Bad: Start Product Recommendation Process
Bad: We have completed the steps necessary and will now proceed with the recommendation pr
ocess for the records in our product database.
Good: Generating product recommendations.
```

Tip: Choose the appropriate level for logging

Debug: Use this level for anything that happens in the program.
Error: Use this level to record any error that occurs.
Info: Use this level to record all actions that are user driven or system specific, such as regularly scheduled operations.

Tip: Provide any useful information

```
Bad: Failed to read location data
Good: Failed to read location data: store_id 8324971
```

Code reviews

Code reviews benefit everyone in a team to promote best programming practices and prepare code for production. Let's go over what to look for in a code review and some tips on how to conduct one.

- Code reviews
- Code review best practices

Questions to ask yourself when conducting a code review

First, let's look over some of the questions we might ask ourselves while reviewing code. These are drawn from the concepts we've covered in these last two lessons.

Is the code clean and modular?

- Can I understand the code easily?
- Does it use meaningful names and whitespace?
- Is there duplicated code?
- Can I provide another layer of abstraction?
- Is each function and module necessary?
- Is each function or module too long?

Is the code efficient?

- Are there loops or other steps I can vectorize?
- Can I use better data structures to optimize any steps?
- Can I shorten the number of calculations needed for any steps?
- Can I use generators or multiprocessing to optimize any steps?

Is the documentation effective?

- Are inline comments concise and meaningful?
- Is there complex code that's missing documentation?
- Do functions use effective docstrings?
- Is the necessary project documentation provided?

Is the code well tested?

- Does the code high test coverage?
- Do tests check for interesting cases?
- Are the tests readable?
- Can the tests be made more efficient?

Is the logging effective?

- Are log messages clear, concise, and professional?

- Do they include all relevant and useful information?
- Do they use the appropriate logging level?

Tips for conducting a code review

Now that we know what we're looking for, let's go over some tips on how to actually write your code review. When your coworker finishes up some code that they want to merge to the team's code base, they might send it to you for review. You provide feedback and suggestions, and then they may make changes and send it back to you. When you are happy with the code, you approve it and it gets merged to the team's code base.

As you may have noticed, with code reviews you are now dealing with people, not just computers. So it's important to be thoughtful of their ideas and efforts. You are in a team and there will be differences in preferences. The goal of code review isn't to make all code follow your personal preferences, but to ensure it meets a standard of quality for the whole team.

Tip: Use a code linter

This isn't really a tip for code review, but it can save you lots of time in a code review. Using a Python code linter like pylint can automatically check for coding standards and PEP 8 guidelines for you. It's also a good idea to agree on a style guide as a team to handle disagreements on code style, whether that's an existing style guide or one you create together incrementally as a team.

Tip: Explain issues and make suggestions

Rather than commanding people to change their code a specific way because it's better, it will go a long way to explain to them the consequences of the current code and *suggest* changes to improve it. They will be much more receptive to your feedback if they understand your thought process and are accepting recommendations,

rather than following commands. They also may have done it a certain way intentionally, and framing it as a suggestion promotes a constructive discussion, rather than opposition.

```
BAD: Make model evaluation code its own module - too repetitive.

BETTER: Make the model evaluation code its own module. This will simplify models.py to be less repetitive and focus primarily on building models.

GOOD: How about we consider making the model evaluation code its own module? This would simplify models.py to only include code for building models. Organizing these evaluations methods into separate functions would also allow us to reuse them with different models without repeating code.
```

Tip: Keep your comments objective

Try to avoid using the words "I" and "you" in your comments. You want to avoid comments that sound personal to bring the attention of the review to the code and not to themselves.

```
BAD: I wouldn't groupby genre twice like you did here... Just compute it once and use that for your aggregations.

BAD: You create this groupby dataframe twice here. Just compute it once, save it as groupby_genre and then use that to get your average prices and views.

GOOD: Can we group by genre at the beginning of the function and then save that as a groupby object? We could then reference that object to get the average prices and views without computing groupby twice.
```

Tip: Provide code examples

When providing a code review, you can save the author time and make it easy for them to act on your feedback by writing out your code suggestions. This shows you are willing to spend some extra time to review their code and help them out. It can also just be much quicker for you to demonstrate concepts through code rather than explanations.

Let's say you were reviewing code that included the following lines:

```
first_names = []
last_names = []
```

```
for name in enumerate(df.name):
    first, last = name.split(' ')
    first_names.append(first)
    last_names.append(last)

df['first_name'] = first_names
df['last_names'] = last_names
```

```
BAD: You can do this all in one step by using the pandas str.split method.
GOOD: We can actually simplify this step to the line below using the pandas str.split meth
od. Found this on this stack overflow post: https://stackoverflow.com/questions/14745022/h
ow-to-split-a-column-into-two-columns
```

```
df['first_name'], df['last_name'] = df['name'].str.split(' ', 1).str
```