Coding Best Practices

Moinak Pyne

October 2021

Contents

Clean and Modular code

Improve Code Efficiency

Add Effective Documentation

Version Control

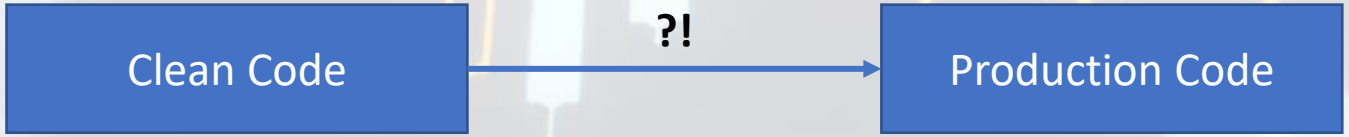
Testing

Logging

Code Reviews

Modular and Meaningful Code

- **Clean code**: code that is readable, simple and concise.
- **Modular code**: Code that is logically broken into functions, classes and modules. Modularizing makes the code organized, efficient and reusable.
- **Production-ready code**: Code that meets expectations for production in reliability, efficiency, and other aspects (white spaces & PEP-8 layout).
- **Production code**: Software running on prod. servers to handle live users and data of the intended audience.



https://www.python.org/dev/peps/pep-0008/?#code-lay-out

Modular and Meaningful Code

- **Refactoring**: Restructuring code to improve the internal structure without changing the external functionality.
- Its never easy to write the best code when you are trying to get it working, allocating time to refactoring is essential for producing high quality code
- Despite the time and effort, it pays off in the long run with development time in the long run
- As a programmer, it develops the skills to make it easier to structure and write good code at the first go.

Efficient Code

Optimizing code to be more efficient can mean making it:

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

Approaches:

- Use vector operations over loops when possible (list comprehensions, lambda functions)
- Know the data structures in use and which methods work faster (Googling really helps!)

Documentation

Why document?

- Clarify complex parts of code
- Navigate code easily
- Describe use and purpose of components

- Major types:
  - Inline comments: line level
  - Docstrings: module and function level
    - https://www.python.org/dev/peps/pep-0257/
    - https://numpydoc.readthedocs.io/en/latest/format.html
  - Project Documentation: project level
    - https://github.com/twbs/bootstrap
    - https://github.com/scikit-learn/scikit-learn
    - https://github.com/jjrunner/stackoverflow

```
# Compute the silhouette scores for each sample
sample_silhouette_values = silhouette_samples(X_subset, labels_subset)
y_lower = 10
for i in range(n_clusters):
    ith_cluster_silhouette_values = sample_silhouette_values[labels_subset == i]
    ith_cluster_silhouette_values.sort()
    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i
    c = cluster_labels == i
    cluster_color = np.median(img[i], 0).reshape((1,1,3))[0,0]
    cluster_color = cm.spectral(float(i) / n_clusters)
    ax.fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_silhouette_values,
        facecolor=cluster_color, edgecolor=cluster_color, alpha=0.7)
```

```
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
```

Version Control

Managing different versions of source code, is useful for:

- Maintain detailed history of project
- If needed, jump back to any point to recover
- Revision history is not enough:
  - Label changes
  - Provide detailed explanations of why a change was made
  - Ability to move between versions
  - Make a change A, make an edit B, then get back change A without affecting B

- In practice:
  - Development in branch
  - Combined code from all developers in main
  - Production in sperate branch

Testing and Data Science

- Problems that could occur in data science aren't always easily detectable
  - incorrect encoding
  - features used inappropriately
  - unexpected data breakages

- To catch these errors, proper testing is required:
  - **Unit test:** A type of test that covers a "unit" of code—usually a single function—independently from the rest of the program.
  - **Test-driven development (TDD):** write tests for tasks before writing the code to implement those tasks.

- https://www.predictiveanalyticsworld.com/machinelearningtimes/four-ways-data-science-goes-wrong-and-how-test-driven-data-analysis-can-help/6947/
- https://speakerdeck.com/pycon2014/getting-started-testing-by-ned-batchelder

Testing and Data Science

Unit Tests

- Test a unit of code independently
- Procedure:
  - Write a function with test cases
  - Use a tool like *pytest*
- Benefits:
  - Can be used repeatedly
  - Execution doesn't stop at failure

```
def test_nearest_square_5():
    assert(nearest_square(5) == 4)

def test_nearest_square_n12():
    assert(nearest_square(-12) == 0)

def test_nearest_square_9():
    assert(nearest_square(9) == 9)

def test_nearest_square_23():
    assert(nearest_square(23) == 16)
```

```
[jlee:.../soft_eng_practices/testing]$ pytest
============================= test session starts ==========
platform darwin -- Python 3.6.3, pytest-3.6.2, py-1.5.4, pluggy-0.6.0
rootdir: /Users/jlee/Udacity/dsnd/soft_eng_practices/testing, inifile:
collected 4 items

test_nearest.py ..F.
```

Test Driven Development

- *Test-driven development*: Writing tests before you write the code that's being tested. Implementation is complete when the test is passed
- Tests can check for different scenarios and edge cases. Hence provide immediate feedback
- When refactoring or adding to the code, TDD helps make sure rest of the code did not break (repeatability)

Logging

- Logging is the process of recording messages to describe events that have occurred while running the code.
  - Debug: use this level for anything that happens in the program
  - Error: for recording failures
  - Info: for recording all actions for regularly scheduled tasks

Python

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

Shell

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

Code Reviews

Based on the concepts discussed so far:

Is the code clean and modular?

- Can it be understood clearly; does it use meaningful names and white spaces; is there duplicated code; can a layer of abstraction be added; are all functions and modules necessary

Is the code efficient?

- Are there loops and can they be vectorized; can better data structures be used; can the number of calculations be shortened; can generators or multi-processing help optimize

Is the documentation effective?

- Are the inline comments concise; is there complex code that's missing documentation; are the docstrings effective; are project documentation provided

Is the code well tested?

- Does the code have test coverage; are edge/interesting cases tested; are tests readable; can tests be more efficient

Is the logging effective?

- Are log messages clear, concise, and professional; are all relevant info included; are appropriate logging levels used

Code Reviews

Suggestions:

- *Pylint* for PEP-8
- Explain issues and make suggestions
- Provide code examples

- https://www.kevinlondon.com/2015/05/05/code-review-best-practices.html
- https://github.com/lyst/MakingLyst/tree/master/code-reviews


Thank
You