Code Quality, Code Formatting, and Linting

Outline

- Defining Clean Code
- Techniques to Maintain Clean Code
  - Code formatters and linting
- Utilizing pre-commit hooks to enforce coding standards and maintain code with Github.
- Exercise

What is Clean Code?

- Readable - Code is read more often than it is written. It should be easy to understand.
- Consistent formatting. Should follow PEP 8 standards for Python.
- Meaningful names - descriptive variable, function, and class names.
- Simple. Functions and classes should do one thing only, and do it well.
- Well-documented.
- Type hinted.
- Testable.

PEP 8 Naming Conventions

- Classes - CamelCase (MyClass)
- Variable - snake_case and lowercase (first_name)
- functions - snake_case and lowercase (quick_sort())
- Constants - snake_case and uppercase (PI = 3.14159)
- Modules should have short, snake_case names and all lowercase (numpy)
- Single quotes and double quotes are treated the same (just pick one and be consistent)
- Triple quotes should always be """Your text here""" not '''Your text here'''

Variable Naming

- - Use descriptive, lowercase names – Variables should be meaningful and easy to understand.
  - ✅ max_users = 100 ❌ mu = 100
- - Use underscores for multi-word names (snake case) – Improves readability.
  - ✅ user_count = 10 ❌ userCount = 10 (CamelCase is for classes, not variables)
- - Avoid single-letter names (except for short loops) – Be explicit.
  - ✅ temperature_celsius = 25.0 ❌ t = 25.0
- - Constants should be uppercase with underscores – Used for values that don't change.
  - ✅ MAX_RETRIES = 5 ❌ maxRetries = 5
- - Private variables start with an underscore – Signals internal use.
  - ✅ _cache = {} ❌ cache = {} (unless public)
- - Avoid reserved keywords – Prevent conflicts with Python's built-in functions.
  - ✅ class_name = "Intro to CI" ❌ class = "Intro to CI" (conflicts with the class keyword)

Functions Naming

- - Use lowercase with underscores – Improves readability and consistency.
  - ✅ def get_user_data(): ❌ def GetUserData(): (CamelCase is for classes)
- - Use descriptive names – Functions should clearly indicate their purpose.
  - ✅ def calculate_total_price(): ❌ def calc():
- - Use verbs for function names – Functions perform actions, so names should reflect that.
  - ✅ def fetch_records(): ❌ def records():
- - Use a leading underscore for internal or private functions – Signals intended internal use.
  - ✅ def _connect_to_db(): ❌ def connect_to_db(): (if meant to be private)
- - Avoid using built-in function names – Prevents accidental overrides.
  - ✅ def format_report(): ❌ def format(): (overrides Python's built-in format function)
- - Use double leading underscores only for name-mangling in classes – Rarely needed.
  - ✅ class Example: def __private_method(self): Double underscore automatically renamed within a class...
  - ❌ def __method(): (not necessary outside classes)

Class Naming

- Use CapWords (PascalCase) – Each word starts with a capital letter, with no underscores.
  - ✅ class DataProcessor: ❌ class data_processor:
- Class names should be nouns or noun phrases – Represents objects or entities.
  - ✅ class UserProfile: ❌ class processUser(): (Functions use verbs, not classes)
- Avoid abbreviations – Use clear and meaningful names.
  - ✅ class AuthenticationManager: ❌ class AuthMgr:
- Use leading underscores for internal classes – Signals that the class is for internal use only.
  - ✅ class _InternalHelper: ❌ class InternalHelper: (if not meant for external use)
- Use metaclass naming convention – Append "Meta" if defining a metaclass.
  - ✅ class CustomMeta(type): ❌ class CustomMetaclass:
- Exception classes should end with `Error` – Makes it clear they are exceptions.
  - ✅ class ValidationError(Exception): ❌ class ValidationIssue:

Line Formatting

Lines should not exceed 79 characters – Improves readability, especially in side-by-side comparisons.

```
def process_large_dataset(data):
    """Processes a dataset and returns useful statistics."""
```

```
def process_large_dataset_with_very_long_name(data, additional_parameters, more_para
    """This line is way too long and hard to read."""
```

Line Formatting

Avoid multiple statements or imports on the same line

Use separate lines for clarity.

```
import os
import sys

x = 5
y = 10
print(x + y)
```

```
import os, sys  # Harder to modify later

x = 5; y = 10; print(x + y)  # Harder to read
```

In-line Comments

- comments should not contradict the code
- comments should be complete sentences
- comments should have a space after the # sign with the first word capitalized
- don’t litter commented code throughout your software.

Documenting Code

```
def divide(a, b):
    """
    Divides two numbers.

    Parameters
    ----------
    a : float
        Numerator.
    b : float
        Denominator.

    Returns
    -------
    float
        Result of division.

    Raises
    ------
    ZeroDivisionError
        If b is zero.
    """
```

Coding Principles

- Don’t Repeat Yourself
- Keep it Simple
- Separation of Concerns
- Split classes into multiple subclasses, inheritances, abstractions, interfaces.
- SOLID Principles of Coding: (https://www.pentalog.com/blog/it-development-technology/solid-principles-object-oriented-programming/)

Methods to Improve Code Formatting

- Decorators
  - Define inner function inside function to call instead of defining inner function in each function call
  - Improves modularity

```
def ask_for_passcode(func):
    def inner():
        print('What is the passcode?')
        passcode = input()

        if passcode != '1234':
            print('Wrong passcode.')
        else:
            print('Access granted.')
            func()

    return inner


@ask_for_passcode
def start():
    print("Server has been started.")


@ask_for_passcode
def end():
    print("Server has been stopped.")


start()  # decorator will ask for password
end()  # decorator will ask for password
```

Methods to Improve Code Formatting

- Context Managers
  - Manage how to interact with external databases and files.
  - Automatically opens and closes files, avoiding complications when errors occur.

```
with open('wisdom.txt', 'w') as opened_file:
    opened_file.write('Python is cool.')

# opened_file has been closed.
```

```
file = open('wisdom.txt', 'w')
try:
    file.write('Python is cool.')
finally:
    file.close()
```

Methods to Improve Code Formatting

- Iterators
  - Use functions to iterate through variables

```
names = ["Mike", "John", "Steve"]
names_iterator = iter(names)

for i in range(len(names)):
    print(next(names_iterator))
```

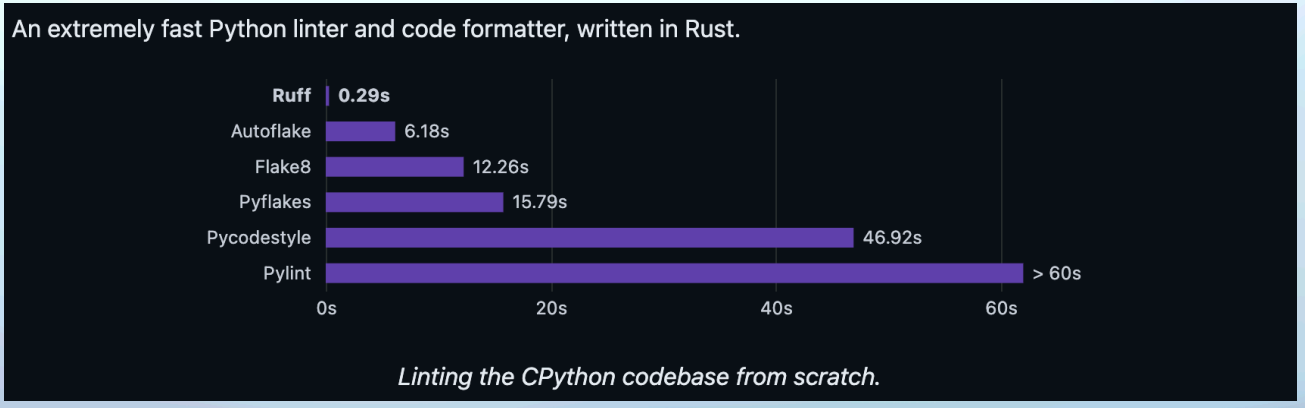
```
names = ["Mike", "John", "Steve"]

for name in names:
    print(name)
```

Linting and Code Formatting

- Linting identifies formatting errors that can alter functionality of code and can correct for formatting
  - Indentation errors, unused variables, etc. Enforces PEP 8 standards.
- Code formatting changes stylistic appearance of code
- Linting is distinct from formatting because linting analyzes how the code runs and detects errors whereas formatting only restructures how code appears.

Automated Linting and Code Formatting

- Pylint: Python Code Linter
- Flake8: Python Code Linter to identify style differences in code
- Black: code formatter
- Ruff: rust optimized code formatter and linter



Black: Automated Code formatting

- Black is an automated code formatter that is able to automatically format code to PEP8 standards

```python
import pytest
import os


# content of test_sample.py
def addition(x,        y):
    "addition function"
    return x + y


# @pytest.mark.parametrize("a, b", [(1,5), (2,6), (3,7), (4,8)])
def test_addition():
    "Test addition function"
    assert addition(5, 4) == (9)
```

```python
import pytest
import os


# content of test_sample.py
def addition(x, y):
    "addition function"
    return x + y


# @pytest.mark.parametrize("a, b", [(1,5), (2,6), (3,7), (4,8)])
def test_addition():
    "Test addition function"
    assert addition(5, 4) == (9)
```

```
(divisiontest) deanlab@SW575738BF divisiontest % black test_sample.py

reformatted test_sample.py

All done! ✨ 🍰 ✨
1 file reformatted.
(divisiontest) deanlab@SW575738BF divisiontest %
```

Ruff: Automated Code Linting

- Identify unused variables and imports for removal.
- Style guides for code and whitespace organization

```python
import pytest
import os


# content of test_sample.py
def addition(x, y):
    "addition function"
    return x + y


# @pytest.mark.parametrize("a, b", [(1,5), (2,6), (3,7), (4,8)])
def test_addition():
    "Test addition function"
    assert addition(5, 4) == (9)
```

```
(divisiontest) deanlab@SW575738BF divisiontest % ruff check test_sample.py
test_sample.py:1:8: F401 [*] `pytest` imported but unused
test_sample.py:2:8: F401 [*] `os` imported but unused
Found 2 errors.
[*] 2 fixable with the `--fix` option.
(divisiontest) deanlab@SW575738BF divisiontest %
```

Ruff: Automated Code Linting

- Removing unused variables and imports.

```
(divisiontest) deanlab@SW575738BF divisiontest % ruff check --fix .
Found 2 errors (2 fixed, 0 remaining).
(divisiontest) deanlab@SW575738BF divisiontest %
```

```python
# content of test_sample.py
def addition(x, y):
    "addition function"
    return x + y


# @pytest.mark.parametrize("a, b", [(1,5), (2,6), (3,7), (4,8)])
def test_addition():
    "Test addition function"
    assert addition(5, 4) == (9)
```

Configuring Ruff

- 700 different rules
  - Naming
  - Pydocstyles
  - Pyupgrade
  - Flake8 rules
- Rules can be configured to specific styles or ignored to match the needs of your project

https://docs.astral.sh/ruff/configuration/

```toml
# Exclude a variety of commonly ignored directories.
exclude = [
    ".bzr",
    ".direnv",
    ".eggs",
    ".git",
    ".git-rewrite",
    ".hg",
    ".ipynb_checkpoints",
    ".mypy_cache",
    ".nox",
    ".pants.d",
    ".pyenv",
    ".pytest_cache",
    ".pytype",
    ".ruff_cache",
    ".svn",
    ".tox",
    ".venv",
    ".vscode",
    "__pypackages__",
    "_build",
    "buck-out",
    "build",
    "dist",
    "node_modules",
    "site-packages",
    "venv",
]

# Same as Black.
line-length = 88
indent-width = 4
```

Configuring Ruff in IDE such as VSCODE

- Many IDEs such as vscode or pycharm have built in linters that identify smaller coding errors and improve code formatting
- Possible to install Ruff into vscode
- Linting is run when files are opened or saved

Integrate Ruff or Black into github using pre-commit hooks

- A good way to format code is when committing code into Github
- Linters and formatters such as Ruff and Black can be integrated into Github
- Install pre-commit in conda environment using pip install pre-commit or integrate pre-commit dependence in pyproject.toml
- Add a pre-commit config file called .pre-commit-config.yaml to project
- In yaml file: add ruff repo

```yaml
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v2.3.0
    hooks:
    -   id: check-yaml
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
-   repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
    -   id: black
- repo: https://github.com/charliermarsh/ruff-pre-commit
  # Ruff version.
  rev: 'v0.0.191'
  hooks:
    - id: ruff
      # Respect `exclude` and `extend-exclude` settings.
      args: ["--force-exclude"]
```

Conclusions

- Code formatting and organizing is an important part coding
- Code formatters and linters such as ruff can be used to automatically format and detects formatting errors in code
- Linting can be implemented as a precommit hook and can be part of IDEs such as vscode or pycharm
- Clean code will lead to more understandable, reliable, and reproducible code.

Exercise

- Set up Ruff locally in your environment.
- Set up a pre-commit hook to run Ruff and black and install it in pyproject.toml to format calculator codebase.

Further Reading

- Ruff documentation: https://docs.astral.sh/ruff/
- Black documentation: https://black.readthedocs.io/en/stable/
- Linting in vscode: https://code.visualstudio.com/docs/python/linting
- Pre-commit documentation: https://pre-commit.com

First Run Ruff Locally to identify errors

- Installing Ruff in your environment using
  - Pip install Ruff
- Once installed, go to folder where repo is located
- Go to src folder
- Type “ruff check .” In command line

```
(pycalcv3) deanlab@SW575738BF calculator-exercise % ruff check .
controller.py:5:1: F403 `from pycalc.constants import *` used; unable to detect undefined names
controller.py:73:40: F405 `ERROR_MSG` may be undefined, or defined from star imports
model.py:3:1: F403 `from pycalc.constants import *` used; unable to detect undefined names
model.py:23:5: E722 Do not use bare `except`
model.py:24:18: F405 `ERROR_MSG` may be undefined, or defined from star imports
pycalc.py:8:8: F401 [*] `os` imported but unused
pycalc.py:15:26: F401 [*] `PyQt5.QtCore.Qt` imported but unused
view.py:4:29: F401 [*] `PyQt5.QtWidgets.QApplication` imported but unused
Found 8 errors.
[*] 3 fixable with the `--fix` option.
(pycalcv3) deanlab@SW575738BF calculator-exercise %
```

Setting up a Pre-commit

- First Install Pre-commit
  - pip install pre-commit
  - Add dependency in pyproject.toml (it should already be added)
- create .pre-commit-config.yaml file and add to repo
- In .pre-commit-config.yaml file
  - Add the following pre-commit information
  -

Add this to your `.pre-commit-config.yaml`:

```yaml
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.1.2
  hooks:
    - id: ruff
```

Or, to enable autofix:

```yaml
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.1.2
  hooks:
    - id: ruff
      args: [--fix, --exit-non-zero-on-fix]
```

Editing .pre-commit-config.yaml file

- Configure .pre-commit-config.yaml file to use ruff
- Add ruff pre-commit to pre-commit.yaml file to include ruff
- Find information for other packages for pre-commits here:
  - https://pre-commit.com/
  - https://pre-commit.com/hooks.html
  - Push yaml file to repo on github

! .pre-commit-config.yaml

```yaml
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v2.3.0
    hooks:
    -   id: check-yaml
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
```

```yaml
- repo: https://github.com/charliermarsh/ruff-pre-commit
  # Ruff version.
  rev: 'v0.0.191'
  hooks:
    - id: ruff
      # Respect `exclude` and `extend-exclude` settings.
      args: ["--force-exclude"]
```

Using pre-commit

- To install pre-commit hooks from configuration yaml file
  - pre-commit install
  - This install pre-commit hooks for each upcoming commit
- To run pre-commit hooks on current files, go to specific directory and run
  - Pre-commit run --all-files
- This will identify all errors
  - We can autofix errors by specifying autofix.

Add this to your `.pre-commit-config.yaml`:

```yaml
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.1.2
  hooks:
    - id: ruff
```

Or, to enable autofix:

```yaml
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.1.2
  hooks:
    - id: ruff
      args: [--fix, --exit-non-zero-on-fix]
```