Code Quality, Code Formatting, and Linting

Dushyant Mehra

Outline

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

What is Code Quality?

What is Code Quality?

- Essentially, code that is considered good:
- Does what it should.
- Follows a consistent style.
- It is easy to understand.
- Has been well-documented.
- It can be tested.

What is Code Quality?

- Essentially, code that is considered good:
- Does what it should.
- **Follows a consistent style.**
- **It is easy to understand.**
- Has been well-documented.
- It can be tested.

Principles of Code Formatting: Code Standards

- PEP8 Standards
  - file organization
  - programming-practices and principles
  - code formatting (indentation, declarations, statements)
  - naming conventions
  - comments

Naming Conventions

- class names should be CamelCase (MyClass)
- variable names should be snake_case and all lowercase (first_name)
- function names should be snake_case and all lowercase (quick_sort())
- constants should be snake_case and all 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)

Variable Naming

- Use descriptive and revealing names
- Avoid ambiguous abbreviations
- Use similar vocabulary
- Don’t use values that aren’t defined
- Use solution domain names
- Don't add redundant context
- Remove unused variable

Function Naming

- Use verbs for function names
- Do not use different words for the same concept
- Write short and simple functions
- Functions should only perform a single task
- Keep your arguments at a minimum
- Avoid side effects
  - A function produces a side effect if it does anything other than take a value in and return another value or values. For example, a side effect could be writing to a file or modifying a global variable.
- Remove unused functions
- Don't use flags in functions
  - Break function into smaller components

Line Formatting

- indent using 4 spaces (spaces are preferred over tabs)
- lines should not be longer than 79 characters
- avoid multiple statements on the same line
- top-level function and class definitions are surrounded with two blank lines
- method definitions inside a class are surrounded by a single blank line
- imports should be on separate lines

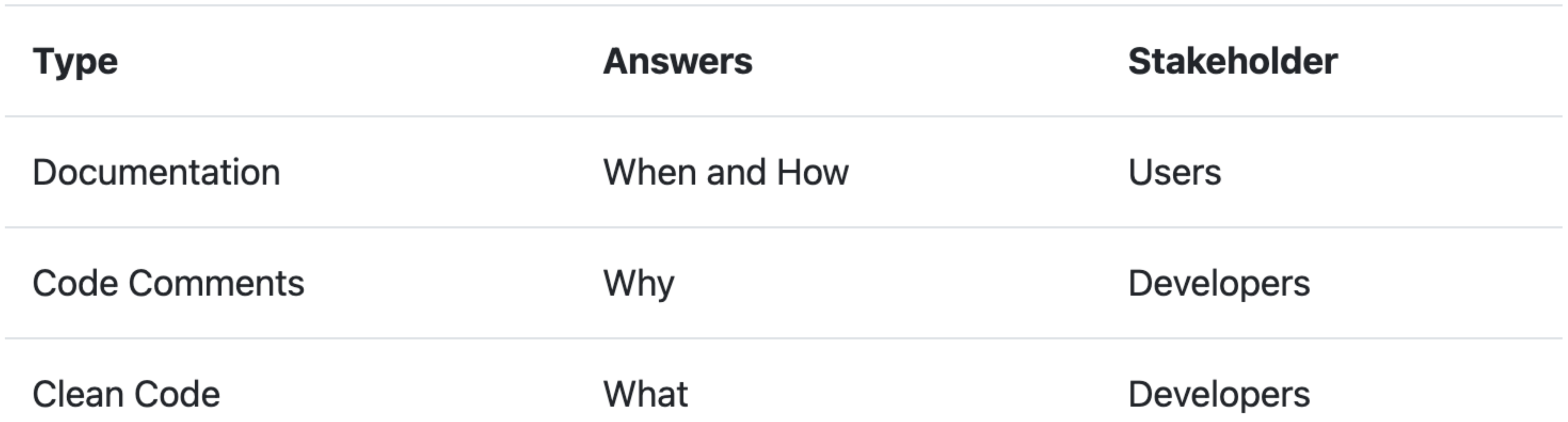
Whitespace

- avoid extra spaces within brackets or braces
- avoid trailing whitespace anywhere
- always surround binary operators with a single space on either side
- if operators with different priorities are used, consider adding whitespace around the operators with the lowest priority
- don't use spaces around the = sign when used to indicate a keyword argument

Commenting Code

- comments should not contradict the code
- comments should be complete sentences
- comments should have a space after the # sign with the first word capitalized
- multi-line comments used in functions (docstrings) should have a short single-line description followed by more text
- Don’t leave in commented code

Commenting vs Documentation vs Clean Code



Type	Answers	Stakeholder
Documentation	When and How	Users
Code Comments	Why	Developers
Clean Code	What	Developers

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/)

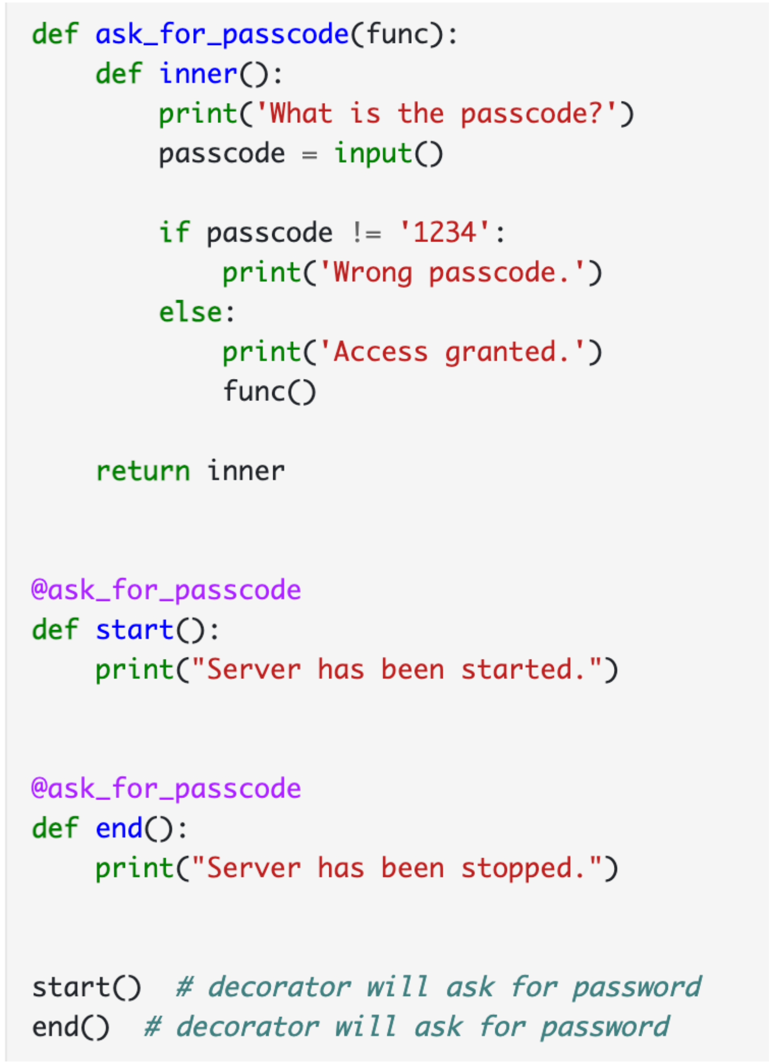
Why is Code Quality and Formatting Important?

Why is Code Quality and Formatting Important?

- Readability: formatting code improves organization and makes it easier to read your code
- Team Support: formatting code in the same way allows other team members to read code to understand functions. Often multiple people work on a single code base and utilize similar functions
- Easier to Find Bugs: Formatting code will make it easier to find errors in code

Methods to Improve Code Formatting

- Decorators
  - Define inner function inside function to call instead of defining inner function in each function call
  - Improves modularity
- Context Managers
- Iterators
- Generators



```python
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

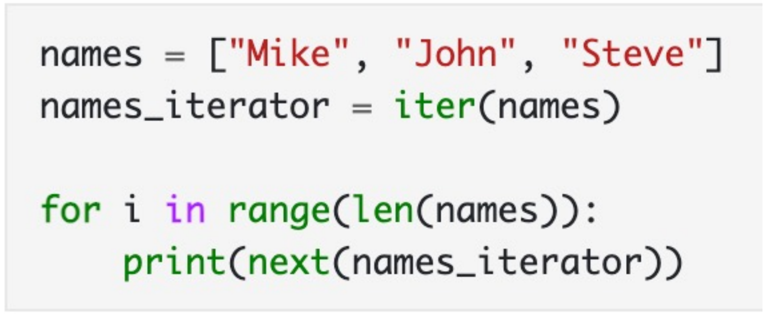
- Decorators
  - Define inner function inside function to call instead of defining inner function in each function call
  - Improves modularity
- Context Managers
  - Manage how to interact with external databases and files
- Iterators

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

# opened_file has been closed.
```

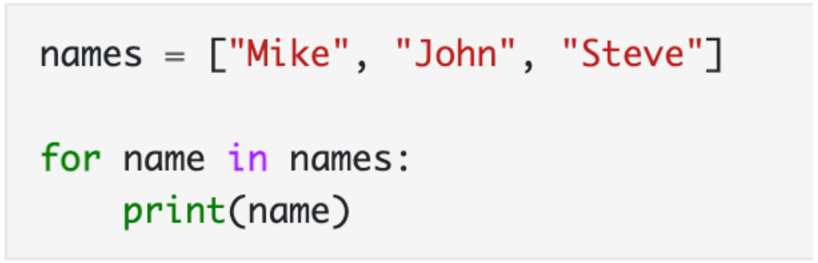
```
file = open('wisdom.txt', 'w')
try:
    file.write('Python is cool.')
finally:
    file.close()
```

Methods to Improve Code Formatting

- Decorators
  - Define inner function inside function to call instead of defining inner function in each function call
  - Improves modularity
- Context Managers
  - Manage how to interact with external databases and files
- 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))
```



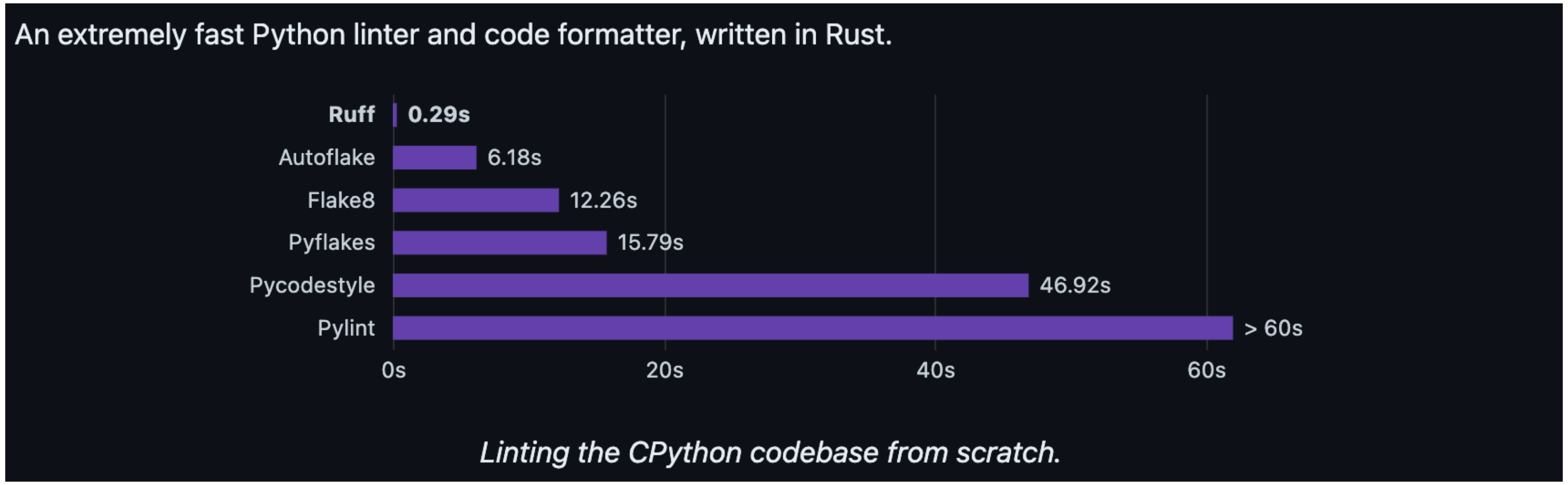
```
names = ["Mike", "John", "Steve"]

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
    - Check for code complexity
    - Enforce PEP-8 code 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. Note: Stylistic and syntactical code detection is enabled by the Language Server.

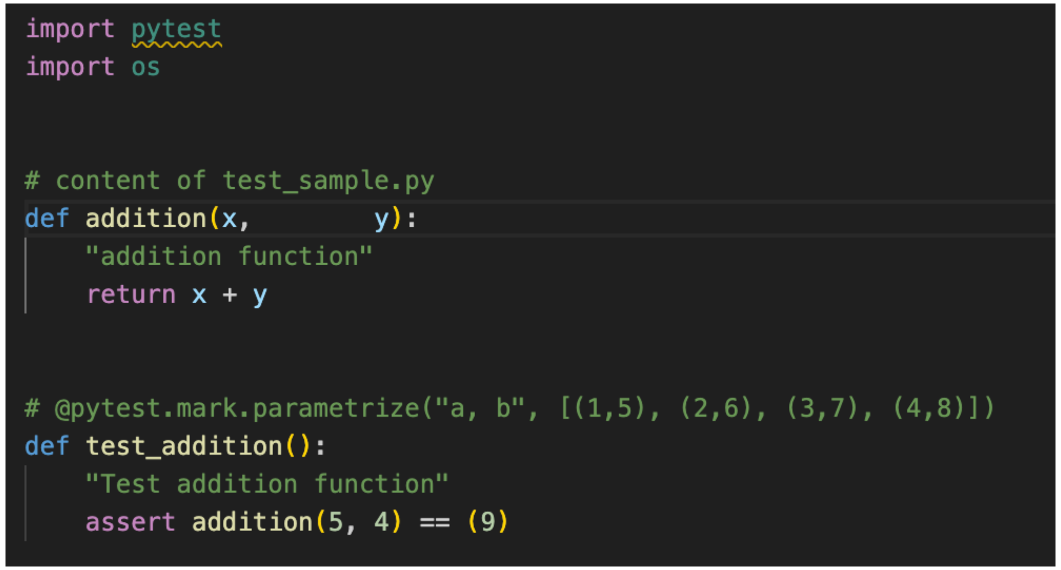
Automated Linting and Code Formatting

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

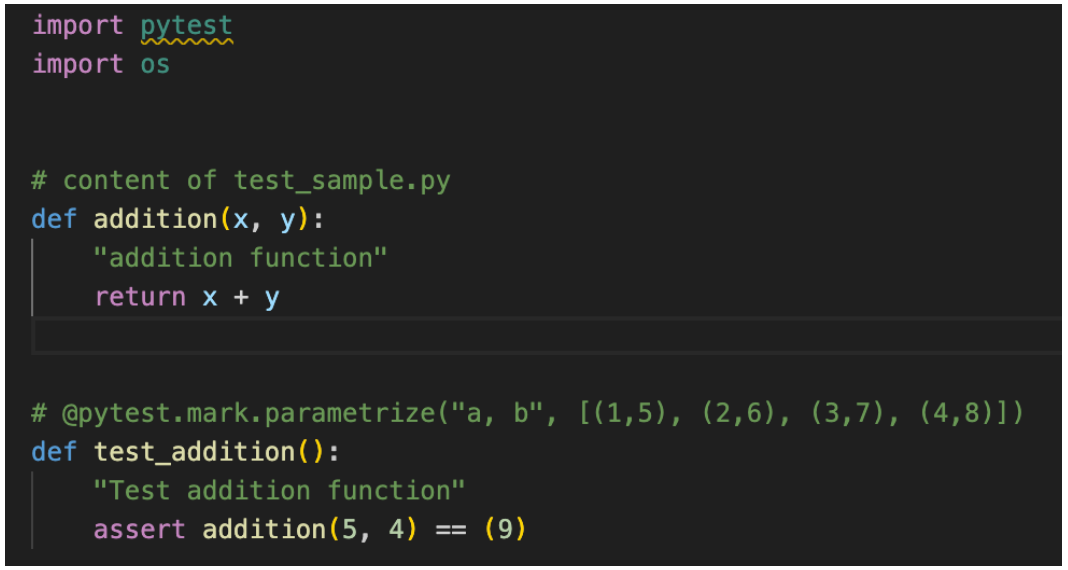


Black: Automated Code formatting

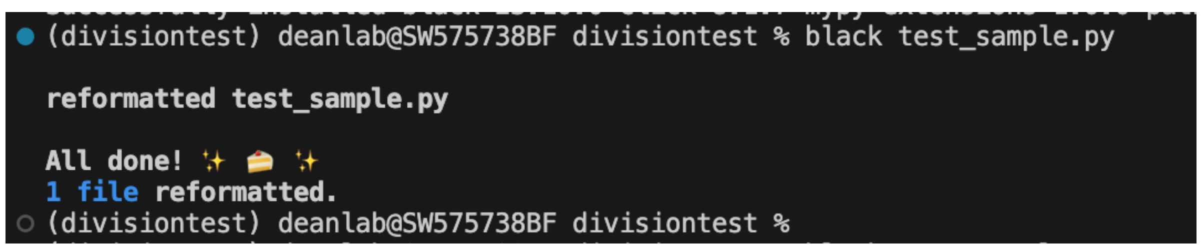
- Black is an automated code formatter that is able to automatically format code to PEP8 standards



```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)
```



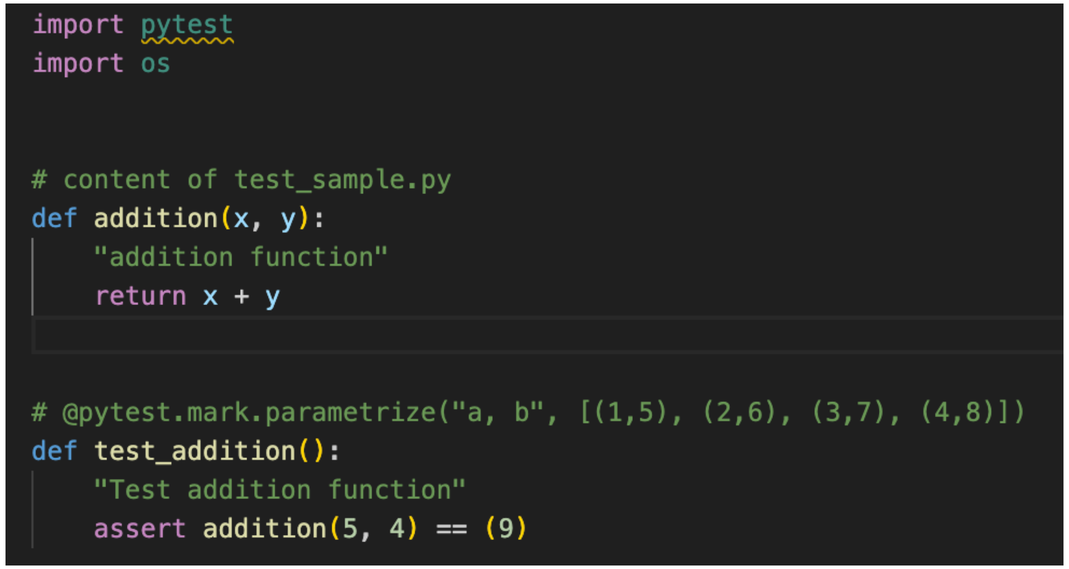
```python
import pytest
import os


# 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)
```



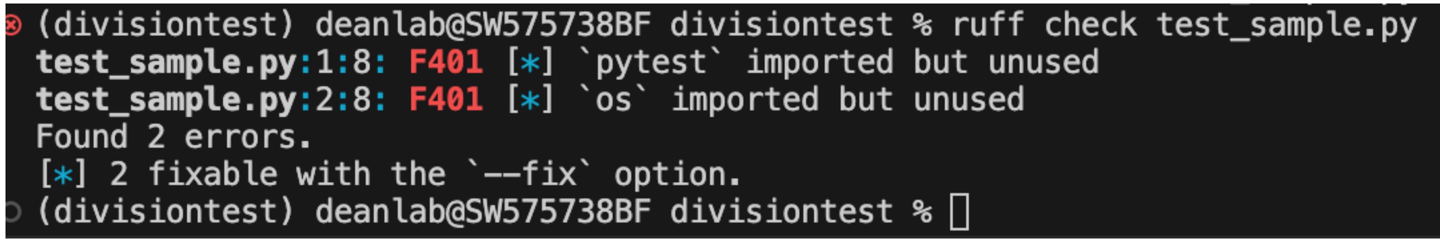
```
(divisiontest) deanlab@SW575738BF divisiontest % black test_sample.py

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
- 700 different rules



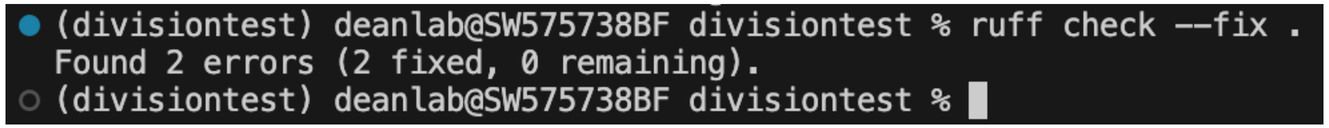
```python
import pytest
import os


# 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)
```



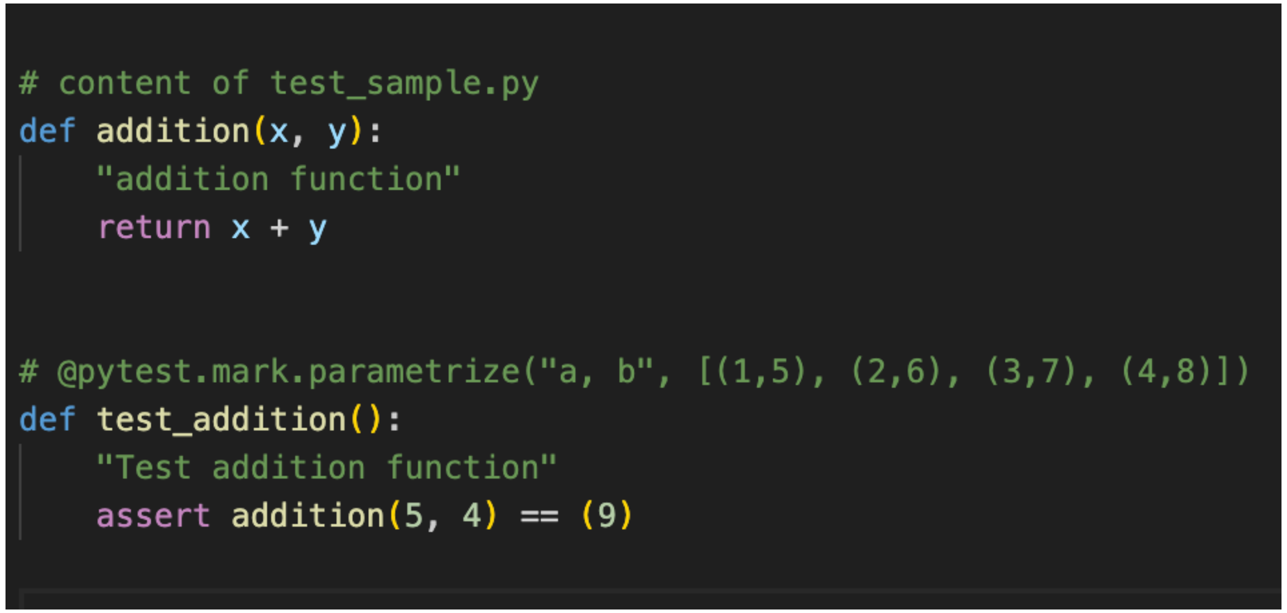
```
(divisiontest) deanlab@SW575738BF divisiontest % ruff check test_sample.py
test_sample.py:1:8: F401 [*] `pytest` imported but unused
test_sample.py:2:8: F401 [*] `os` imported but unused
Found 2 errors.
[*] 2 fixable with the `--fix` option.
(divisiontest) deanlab@SW575738BF divisiontest % 
```

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

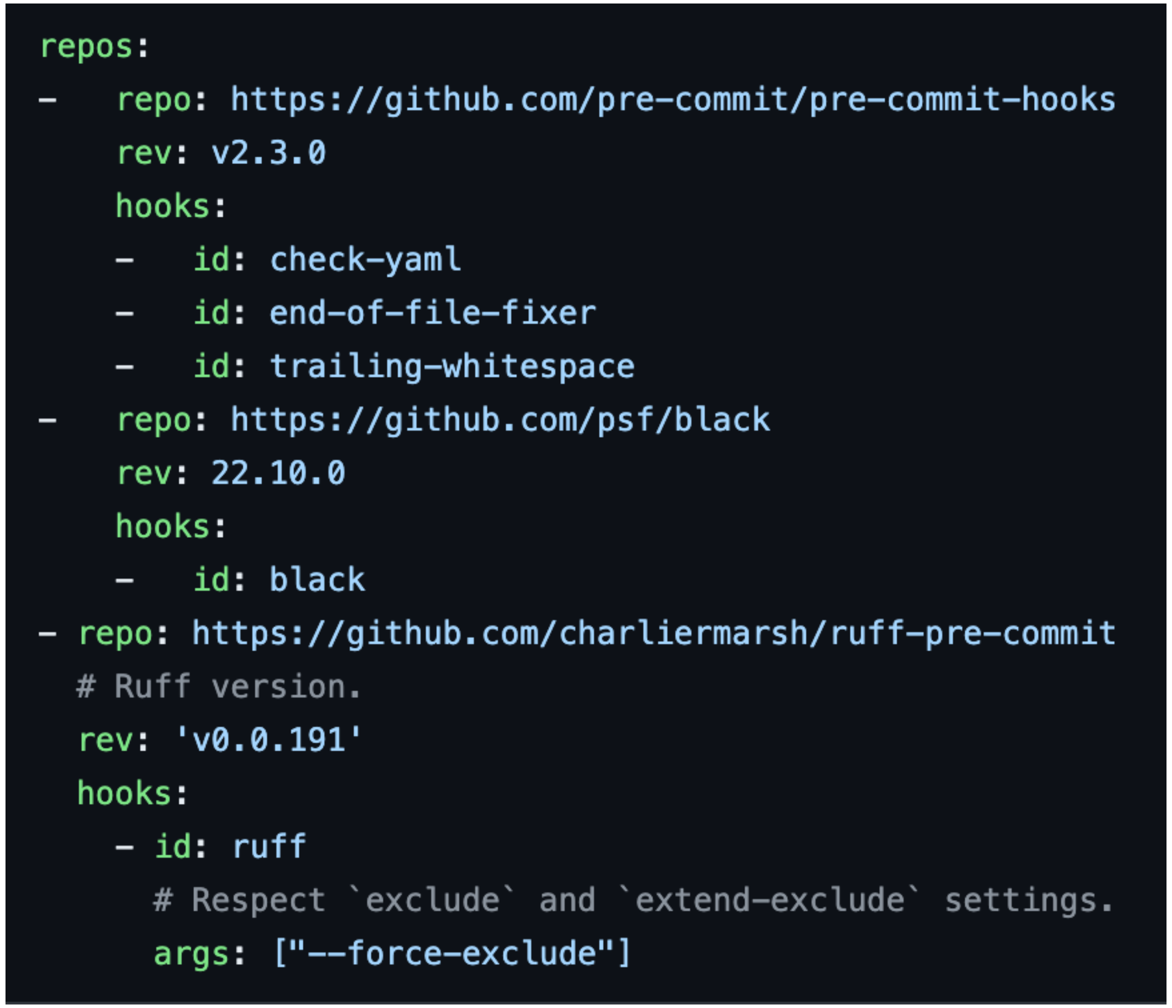
- 700 different rules
  - Naming
  - Pydocstyles
  - Pyupgrade
  - Flake8 rules
- Rules can be configured to specific styles or ignored to match the needs of your project

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#:~:text=Linting%20highlights%20syntactical%20and%20stylistic,that%20can%20lead%20to%20errors.
- Pre-commit documentation: https://pre-commit.com