Python Coding Standards

IGNITE BRILLIANCE

ARRAY

Contents

Python Coding Standards

v 0.1

Purpose

This document outlines the standards and practices to be followed when writing Python code in our organization. Adherence ensures readability, maintainability, and consistency across projects.

General Principles

1. **Readability First**: Code should be easy to read and understand for anyone familiar with Python.
2. **Consistency**: Follow the same patterns and conventions across all projects.
3. **Simplicity**: Avoid unnecessary complexity. Prefer clear and concise solutions.
4. **Testable**: Code should be structured to enable comprehensive testing.
5. **Performance**: Write efficient and optimized code but not at the expense of clarity.

---

1. Code Style and Formatting

Adhere to the PEP 8 style guide with additional rules. Formatting tools should be run as part of the code commit process (set up hooks to automatically run linters) to enforce formatting standards.

Indentation

- Use **4 spaces** per indentation level. Do not use tabs

Line Length

- Limit all lines to a maximum of **88 characters** (default for black formatter)

Naming Conventions

- **Variables/Functions**: Use `snake_case` (e.g., process_data)
- **Classes**: Use `PascalCase` (e.g., DataProcessor)
- **Constants**: Use `UPPERCASE_SNAKE_CASE` (e.g., MAX_RETRIES).

Imports

- Organize imports into three sections:
    1. Standard library imports
    2. Third-party imports
    3. Local application imports
- Sort imports alphabetically for better readability

Docstrings

- Use PEP 257 for docstrings
- Use triple double quotes (""") for docstrings
- Include module-level, class-level, and function-level docstrings

Commenting

- Use inline comments sparingly. Over commenting the code base is undesirable
- Comments should be used to explain complex logic or not straightforward decisions and not to state the obvious. For complex logic, add a block comment above the relevant code
- Do not leave commented out snippets in production code

2. Logging

- Use Python's built-in logging module instead of print statements
- Define a consistent logging configuration
- Agreed logging format is the following:

  `%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n`

  Explanation:

  `%d{HH:mm:ss.SSS}` – the timestamp in HH:mm:ss.SSS format (hours:minutes.milliseconds): **14:23:45.678**

  `[%t]` – the name of the thread (e.g., main thread): **[main]**

  `%-5level` – the log level, left-padded to ensure alignment: **INFO**

  `%logger{36}` – The logger name truncated to a max 36 characters: **com.example.MyClass**

  `%msg%` – the actual log message: **Processing completed successfully.**

  `%n` – the log message ends with a newline

  Example log message:

  14:23:45.678 [main] INFO com.example.MyClass - Processing completed successfully.

- Use log levels according to their purpose:
  - DEBUG: Detailed diagnostic output
  - INFO: General runtime events
  - WARNING: Indications of potential issues
  - ERROR: Errors that do not halt the program
  - CRITICAL: Severe errors causing application failure

- Best practice: use lazy string eval for logging to avoid unnecessary executions:

```
logger.debug("debug output: %s", my_var)
```

---

3. Project Structure

Use the following standard structure for Python projects, subject to conform to the specific project if needed:

project_name/

├── src/

    ├── __init__.py

    ├── main.py

    ├── module_name/

        ├── __init__.py

        ├── handlers.py

        └── utils.py

├── tests/

    ├── __init__.py

    ├── test_main.py

    └── test_module_name.py

├── configs/

    └── config.yaml or similar

├── packages.txt

├── README.md

├── requirements.txt

├── DockerFile

├── dockerignore

├── .gitignore

├── setup.py

Key Points:

- Place application code under the `src/` directory
- Organize by feature/module
- Place all tests in the `tests/` directory, mirroring the structure of `src/`
- Configuration files and .env files should go into the configs/ folder.

---

4. Version Control

- Follow Git best practices:
    - Use feature branches (feature/feature-name) for new features.
    - Use main or master as the default stable branch.
    - Write meaningful commit messages that describe the change in a concise way, including JIRA id

---

5. Testing

- Use **pytes**t for testing
- Achieve at least **90% test coverage** for critical components and **80%** for the entire project
- Organize tests into unit, integration, and functional categories
- Run tests automatically in the CI pipeline

Tools:

- Code coverage: pytest --cov=src/
- Linting: flake8, pylint
- Formatting: black and isort

---

6. Dependency Management

- Use requirements.txt or poetry for dependency management
- Use pip-tools to manage and pin dependencies

---

7. Security

- Avoid hardcoding secrets (e.g., API keys). Use environment variables with python-decouple or dotenv
- Regularly update dependencies to patch known vulnerabilities
- Use static code analysis tools like **bandit** for identifying security issues

---

8. Code Reviews

- Every merge request must be reviewed by at least one peer and linked to a specific assigned JIRA task item
- Checklist for reviewers:
    - Is the code well-structured and readable?
    - Are there sufficient tests?
    - Does the code adhere to the coding standards?

9. Continuous Integration/Deployment

- Use CI/CD tools to automate:
    - Running tests.
    - Code quality checks.
    - Deployment pipelines.
- Dockerize the project by providing docker file. It helps with portability, encapsulation and keeping dependencies in one place for easy installation.

---

10. Documentation

- Maintain a README.md file with:
    - Clear project description
    - Setting pre-requisites (e.g. setting environment variables)
    - Step by step setup instructions and also how to run the application
- Use tools for auto-generating documentation (e.g. Sphinx)

Tool Summary

- **Linting**: flake8, pylint
- **Formatting**: black, isort
- **Testing**: pytest
- **Code Quality**: mypy, bandit
- **CI/CD**: GitHub Actions, Jenkins
- **Documentation**: Sphinx, Markdown

11. Best practices

Adhere to software development best practices by following the guidelines when it makes sense.

Guidelines

DRY – don't repeat yourself. Avoid using duplicate code to make maintenance easier

KISS – keep it simply simple. Writing code that is simple to understand and simple enough to test

Separation of concerns – modules should have clearly defined scope with minimal coupling/overlapping with others. Business logic should be separated from API interfaces and data access layers.

Component based approach – producing coherent but extendable components help with reusability of the code.

SOLID – follow these principles in Object Oriented design:

Principle	**Concept**	**Benefit**
Single Responsibility	One responsibility per class	Easier maintainability
Open-Close	Open to extension, closed to modification	Protects existing functionality
Liskov Substitution	Derived types must be substitutable for their base types	Enforces proper use of inheritance
Interface Segregation	Smaller, specific interfaces	Reduce unnecessary dependencies
Dependency Inversion	Depending on abstractions, not concrete implementations	Decouples concrete implementations by abstraction