**Python Coding Standards and Guidelines**

**Table of Contents**

1. [General Principles](#1-general-principles)
2. [Coding Practices](#2-coding-practices)
3. [Project Structure](#3-project-structure)
4. [Class and Method Standards](#4-class-and-method-standards)
5. [Variables and Imports](#5-variables-and-imports)
6. [Logging and Exception Handling](#6-logging-and-exception-handling)
7. [Constants and Enums](#7-constants-and-enums)
8. [Documentation](#8-documentation)
9. [Future Aspirations](#9-future-aspirations)

**1. General Principles**

**Readability, Maintainability, and Consistency**

* **Readability**: Code should be easy to read and understand by others.
* **Maintainability**: Write code that can be easily maintained and updated.
* **Consistency**: Consistent coding styles improve team collaboration.

**PEP 8 and PEP 20 Compliance**

* **PEP 8**: Follow Python's official style guide for code layout and formatting.
* **PEP 20 (The Zen of Python)**: Embrace simplicity and clarity in your code.

**2. Coding Practices**

**Use of Tools**

* **Flake8**: For style guide enforcement and code linting.
* **Black**: For automatic code formatting.
* **Pylint**: For code analysis and enforcing coding standards.

**Naming Conventions**

* **Variables and Functions**: Use lowercase with underscores (my\_variable).
* **Classes**: Use CapWords convention (MyClass).

**Introducing New Libraries**

* **Avoid Unnecessary Dependencies**
  + Only use libraries that are essential.
  + Consult the team before adding new or experimental libraries.

**3. Project Structure**

**Directories and Modules**

* Organize code into logical directories and packages.
* **Example Structure**:

arduino

Copy code

project\_name/

├── src/

│ ├── module1/

│ ├── module2/

├── tests/

├── config/

├── scripts/

├── data/

└── README.md

**File Naming**

* Use descriptive, lowercase names with underscores for files and directories.

**4. Class and Method Standards**

**Avoid Excessive Class Variables**

* **Minimize Class Variables**
  + Avoid storing large or mutable objects like dataframes as class variables.
  + Pass data between methods explicitly.

**Class Structure**

1. **Imports**
   * Start with standard library imports (sys, os).
   * Then third-party libraries (pandas, numpy).
   * Followed by local module imports.
2. **Constructor (\_\_init\_\_)**
   * Initialize essential variables and set up the logger.
   * Do not perform heavy computations.

python

Copy code

class MyClass:

def \_\_init\_\_(self):

self.logger = logging.getLogger(\_\_name\_\_)

1. **Main Method (run)**
   * Implement a run() method as the entry point.
   * Use this method to orchestrate the workflow.

python

Copy code

def run(self):

self.logger.info("Starting process")

data = self.load\_data()

result = self.process\_data(data)

self.save\_results(result)

**5. Variables and Imports**

**Variable Naming and Passing**

* **Consistency Across Methods**
  + Keep variable names consistent when passing between methods.
  + Enhances readability and traceability.
* **Explicit Returns**
  + Return modified objects rather than modifying in place.

python

Copy code

dataset\_df = self.aggregate\_data(dataset\_df)

**Import Rules**

* **Import Order**
  1. Standard library imports.
  2. Third-party library imports.
  3. Local application imports.
* **Avoid Wildcard Imports**
  1. Do not use from module import \*.
  2. Explicitly import required functions or classes.

**6. Logging and Exception Handling**

**Logging**

* **Mandatory Logging**
  + Every method should contain at least one log statement.
  + Use appropriate logging levels:
    - DEBUG: Detailed information for diagnosing problems.
    - INFO: Confirmation that things are working as expected.
    - WARNING: An indication that something unexpected happened.
    - ERROR: Due to a more serious problem, the software has not been able to perform some function.
    - CRITICAL: A serious error, indicating the program itself may be unable to continue running.
* **Example**

python

Copy code

def process\_data(self, data):

self.logger.info("Processing data")

# Processing logic

**Exception Handling**

* **Try-Except Blocks**
  + Use try-except blocks to handle exceptions gracefully.
  + Log exceptions with error messages and stack traces.
* **Module-Level Exception Handling**
  + Each module should handle exceptions at the entry point to prevent crashes.

python

Copy code

def run(self):

try:

self.logger.info("Starting main process")

# Main process logic

except Exception as e:

self.logger.error("An error occurred", exc\_info=True)

**7. Constants and Enums**

**Use of Enums**

* **Defining Constants**
  + Use Enum classes to define constants like column names.
* **Example**

python

Copy code

from enum import Enum

class ColumnNames(Enum):

ID = "ID"

NAME = "Name"

AGE = "Age"

# Usage in code

data[ColumnNames.ID.value]

* **Benefits**
  + Improves code readability.
  + Reduces errors from typos.
  + Centralizes the management of constants.

**8. Documentation**

**Docstrings**

* **Purpose**
  + Provide clear explanations of what the code does.
  + Assist others in understanding and using your code.
* **Guidelines**
  + Use triple quotes for docstrings.
  + Include a brief description, arguments, and return values.
* **Example**

python

Copy code

def calculate\_sum(a, b):

"""Calculate the sum of two numbers.

Args:

a (int): The first number.

b (int): The second number.

Returns:

int: The sum of a and b.

"""

return a + b

**Comments**

* **When to Comment**
  + Use comments to explain complex or non-obvious parts of the code.
  + Keep comments up-to-date with code changes.
* **Style**
  + Use # for inline comments.
  + Place comments above the code they refer to.

**9. Future Aspirations**

**Adoption of Advanced Tools**

* **YAPF**
  + Consider using Google's YAPF for automatic code formatting.
  + Aligns code style with Google's Python style guide.
* **Reference**
  + Google's Python Style Guide: <https://google.github.io/styleguide/pyguide.html>

**Additional Best Practices**

**Configuration Management**

* **Centralized Configurations**
  + Use configuration files (e.g., YAML, JSON) to manage settings.
  + Avoid hardcoding values in the code.

**Directory Management**

* **Standard Directories**
  + Separate code, tests, data, and configuration files into distinct directories.
  + Enhances organization and clarity.

**Testing**

* **Unit Tests**
  + Write unit tests for individual functions and methods.
  + Use testing frameworks like unittest or pytest.
* **Integration Tests**
  + Ensure different modules work together correctly.