# **Coding Standards and Best Practices Guidelines**

This template is a step by step guide to help you state coding standards and best practices guidelines for your new product or software:

## **1. General Principles**

### **1.1 Consistency**

* Maintain consistency in code style and structure throughout the project.
* Use a common coding standard (e.g., PEP 8 for Python, Google Java Style Guide for Java).

### **1.2 Readability**

* Write code that is easy to read and understand.
* Use meaningful names for variables, functions, and classes.
* Include comments and documentation to explain complex logic and decisions.

### **1.3 Maintainability**

* Write code that is easy to maintain and extend.
* Avoid hard-coding values; use constants or configuration files.
* Modularize code by breaking down functionality into smaller, reusable functions or classes.

## 

## 

## 

## 

## 

## **2. Naming Conventions**

### **2.1 Variables**

* Use descriptive names that convey the purpose of the variable.
* Use camelCase for variable names (e.g., totalAmount, userName).

### **2.2 Functions and Methods**

* Use verbs or verb phrases to name functions (e.g., calculateTotal, getUserInfo).
* Use camelCase for function and method names.

### **2.3 Classes**

* Use nouns or noun phrases to name classes (e.g., Customer, Order).
* Use PascalCase for class names.

### **2.4 Constants**

* Use all uppercase letters with underscores to separate words (e.g., MAX\_CONNECTIONS, DEFAULT\_TIMEOUT).

## **3. Code Structure and Formatting**

### **3.1 Indentation**

* Use consistent indentation (e.g., 4 spaces per level).
* Do not use tabs; use spaces for indentation.

### **3.2 Line Length**

* Limit line length to a maximum of 80-100 characters.
* Break long lines into multiple lines for better readability.

### 

### 

### **3.3 Blank Lines**

* Use blank lines to separate logical sections of code.
* Add blank lines around function and class definitions.

### **3.4 Braces**

* Use braces even for single-line conditional statements and loops.
* Place opening braces on the same line as the statement (e.g., if (condition) {).

## **4. Commenting and Documentation**

### **4.1 Inline Comments**

* Use inline comments sparingly and only for complex or non-obvious code.
* Place inline comments on the same line as the code they refer to, preceded by two spaces (e.g., int count = 0; // Initialize count).

### **4.2 Block Comments**

* Use block comments to explain code sections, algorithms, or important decisions.
* Place block comments above the code they refer to.

### **4.3 Documentation Comments**

* Use documentation comments to describe the purpose, parameters, and return values of functions and classes.
* Follow the documentation standard of your language (e.g., Javadoc for Java, docstrings for Python).

## 

## 

## 

## **5. Error Handling**

### **5.1 Exceptions**

* Use exceptions for handling errors and exceptional conditions.
* Avoid using exceptions for regular control flow.

### **5.2 Logging**

* Use logging to record significant events and errors.
* Include sufficient context in log messages to facilitate debugging.

## **6. Testing**

### **6.1 Unit Tests**

* Write unit tests for individual functions and methods.
* Ensure tests cover various edge cases and error conditions.

### **6.2 Integration Tests**

* Write integration tests to verify the interactions between different components.
* Use mock objects and stubs to isolate dependencies.

### **6.3 Test Coverage**

* Aim for high test coverage, but prioritise meaningful tests over achieving a specific percentage.
* Regularly run tests and include them in your continuous integration process.

## 

## 

## 

## **7. Code Reviews**

### **7.1 Peer Reviews**

* Conduct regular code reviews to ensure adherence to coding standards and best practices.
* Use code review tools to facilitate the review process (e.g., GitHub Pull Requests, Gerrit).

### **7.2 Review Guidelines**

* Focus on the code's functionality, readability, and maintainability.
* Provide constructive feedback and suggestions for improvement.

## **8. Security Best Practices**

### **8.1 Input Validation**

* Validate and sanitise all user inputs to prevent injection attacks.
* Use libraries and frameworks that provide built-in protection against common security vulnerabilities.

### **8.2 Authentication and Authorization**

* Use secure authentication and authorization mechanisms.
* Follow the principle of least privilege when granting access to resources.

### **8.3 Data Protection**

* Encrypt sensitive data at rest and in transit.
* Use secure storage for credentials and other sensitive information.

## 

## 

## **9. Performance Optimization**

### **9.1 Efficient Algorithms**

* Choose efficient algorithms and data structures to optimise performance.
* Avoid premature optimization; focus on clarity and correctness first.

### **9.2 Profiling**

* Use profiling tools to identify and address performance bottlenecks.
* Optimise code based on profiling results, not assumptions.