Coding Standard

Coding Standard

Coding standards are a set of guidelines or best practices that developers follow to write clean, consistent, and maintainable code. These rules cover naming conventions, file structure, formatting, and more. The necessity of coding standards lies in improving readability, reducing errors, and enabling collaboration among developers. By following a consistent approach, teams can avoid confusion, ensure compatibility, and make it easier to maintain and scale projects over time. Coding standards also help in automating quality checks and adhering to industry norms.

1. Naming Conventions

Variables

Use snake_case for variable names.
Variable names should be meaningful and descriptive.
Example:

```
user_name = "John"
total_students = 25
```

Constants

Use ALL_CAPS with underscores separating words.
Define constants at the top of the module.

```
MAX_RETRIES = 3
DEFAULT_TIMEOUT = 60
```

Functions and Methods

Use snake_case for function and method names.
Function names should describe the action or behavior of the function.

```
def calculate_total_price(item_list):
return sum(item.price for item in item_list)
```

Classes

Use PascalCase for class names
Class names should be nouns and describe the object or entity they represent.

```
class UserProfile:
def __init__(self, name, age):
```

```
self.name = name
self.age = age
```

Modules and Packages

Use snake_case for module (file) names and package (directory) names.
Avoid using capital letters or special characters.

```
**Example:**

my_project/
models.py
views.py
forms.py
```

5. Exception Names

The class naming convention applies here. However, you should use the suffix “Error” on your exception names.

```
class CustomError(Exception):
    pass

class FileNotFoundError(Exception):
    pass
```

Layout Convention

Indentation

Use 4 spaces per indentation level. Never use tabs.

```
def example_function():
if True:
print("Indented with 4 spaces")
```

Tabs or Spaces

Tabs should be used solely to remain consistent with code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation.

Line Length

Limit all lines to length 120 characters

```
def example_function():
if True:
print("Indented with 4 spaces")
```

Blank Space

Use two blank lines before top-level function and class definitions.
Use one blank line to separate methods inside a class.

```
def first_function():
pass

def second_function():
pass

class MyClass:
def method_one(self):
pass


def method_two(self):
pass
```

Comments

Inline Comments

Use inline comments sparingly and make sure they explain why the code does something, not what it does.
Place inline comments on the same line, separated by two spaces.

```
x = x + 1 _# Increment x by 1_
```

Block Comments

Use block comments to explain a section of code.
Start each line with a # and a single space, aligning with the indentation level of the code.

```
_# This block calculates the average
# from a list of scores._
total = sum(scores)
average = total / len(scores)
```

Docstrings

Use triple quotes (""") for docstrings in functions, classes, and modules.
Docstrings should explain the purpose of the function, method, or class.

```
def greet(name):
"""
This function takes a name as input and returns a greeting string.
```

```
"""
return f"Hello, {name}!"
```

Imports

The import statement, just like any other statement or keyword in Python, should be used and added to the code properly following best practices. Here's how:

Multiple Imports

Multiple imports should usually be on separate lines.

```
import numpy
import pandas
import matplotlib
```

Always on the Top

Imports are always put at the top of the file i.e.

After any module comments and docstrings.
Before module globals and constants.

```
# import the numpy module
import numpy

### Import Modules in an Order
```

A good practice is toimport modules in the following order −

```
Standard library modules – e.g. sys, os, getopt, re.
Third-party library modules – e.g. ZODB, PIL.Image, etc.
Locally developed modules.
```