*Python Coding Standards*

**Imports Blank Lines and the Indentations**

The import should be in a particular sequence. First, the standard libraries, then the third party, and last, the local libraries should be imported. If you only need a single function/class from the import, do an absolute import. It makes your code much cleaner, more accurate, and easier to identify. Don’t forget to add a space between different types of imports.

There should be two blank lines surrounding classes and top-level functions. The methods inside of the class should be surrounded by a single blank line only. The preferred method of indentation is spaces, the 4 spaces indentation is accepted and accurate, but still, most people prefer tab indentation. Please keep in mind not to mix both spaces and tabs for indentation.

**Import only the packages (especially standard one) that are required for the code**.

***For example:-***

# Don't forget to add a space between different group of imports

# first of all, the standard library imports

import standard\_library\_import\_a

import standard\_library\_import\_b

# then, the third party imports

# at the last, local library import

# two blank lines for top level functions

def top\_level\_function(argument):

# A standard four space indent

print(argument)

**The length of the line and the line breaks**

The length of the line should not be greater than 79 characters. In the case of docstrings and comments where a block of text is large, it is limited to 72 characters. For long multiple case statements, the backslashes are permissible. For using log statements with binary operators, python suggests breaking the formula line before the binary operator for better readability.

**For Example:**

def sample\_function(arg1, arg2):

**'''**  
The document string length for a single line should be less than  
72 characters. So that long texts should be adjusted in a single  
window  
'''    
#  code has maximum lengths of 79 characters, can use backslash  
# to break the line  
list\_of\_subjects = [  
'Physics', 'Chemistry', 'Mathematics', 'Biology',  ‘Bio’, \  
]

**Whitespaces, Trailing commas and String quotes**

One should avoid extra white spaces, there must be a single white space around both sides of an operator, one after the comma and none inside opening or closing of parenthesis. Both single quotes and double quotes are acceptable in [python web development](https://www.zenesys.com/python-development-company), you should use both if you need quotes inside quotes to avoid syntax error and extra backslash.

**For example:-**

# Examples of commas and whitespaces

x, y = 30 , "Sample text"

# how to use quotes inside quotes

text = "This text is using 'the single quote' inside double quote”

print(text)

**Naming conventions**

Use grammatically correct variable names, the class name should start with an uppercase and must follow PascalCase convention If more than two words are to be used. In the same way, a function name should be joined with an underscore, and it must be lowercase. In method arguments, always use self as the first argument to declare an instance variable. In the same way, use ‘cls’ for the first argument for the class method. If the function name clashes with a reserved argument, use an underscore instead of a wrong spelling. Constants are declared in all capital letters.

**For example:-**

# class name follows PascalCase convention

class StudentDetails:

def \_\_init\_\_(self, first\_name, last\_name):

self.first\_name = first\_name

self.last\_name = last\_name

# Method name, variable names in lowercase joined with an underscore

def grade(self, marks\_obtained):

# constants in capital

GRACE = 2

marks\_obtained = GRACE + marks\_obtained

if marks\_obtained > 90:

self.student\_grade = 'A'

elseif marks\_obtained > 70:

student\_grade = 'B'

else:

student\_grade = 'C'

**User defined functions (UDF)**

Function Names: Use lowercase words separated by underscores (snake\_case).

Variable Names: Also use snake\_case for variable names

Function Docstrings: Include a docstring that explains what the function does. Use triple quotes for multi-line docstrings.

Add Type Hints: Use type hints for parameters and return types to improve code clarity.

def calculate\_area(radius: float) -> float:

pass

Keep Functions Short: A function should ideally do one thing. If a function is too long or complicated, consider breaking it down into smaller helper functions.

Use Default Arguments Wisely: Default values should be immutable (e.g., None, strings, numbers, tuples).

def greet(name: str, greeting: str = "Hello") -> str:

return f"{greeting}, {name}!"

Explicit Returns: Always use explicit return statements. Avoid functions that perform operations but don't return a value unless it’s clear that the return value is unnecessary.

**Some Other Coding Recommendations:**

There are certain coding recommendations that should be kept in mind to be consistent with an errorless and quality of code. I will list down the points and then explain them later in the code.

**Exception handling**

**For example:-**

try:

file = open('filename.txt')

file.write('Hello World')

except Exception as e:

print('Cannot open the file :', e)

finally:

# Make sure to close the file after

file.close()

**Documentation**

Documenting every method with proper specification of parameters, return type, and data types. Try to avoid multiple returns from a function, a single generic return is preferred.

**For example:-**

def get\_grades(marks):

"""

Summary: getting grades from marks

Description: This function takes marks as an argument and returns grades

params:

marks(int) : marks obtained

grade(string) : grade achieved

"""

if marks > 90:

grade = 'A'

elseif marks > 70:

grade = 'B'

else:

grade = 'C'

return grade

**Use dry (DON’T REPEAT YOURSELF):**

Always use the DRY principle to reuse the code. The best way to do it is to use functions and classes. The common functions can be put into a separate utils.py file and can be used several times instead of creating similar functions again and again.

Suppose if you need to read three files, instead of writing code for file read thrice, you can read it as a function and save your time.

**For example:-**

# function to read the file read

def file\_read(filename):

with open(filename, 'r') as f:

return f.read()

qualities = file\_read('quality.txt')

description = file\_read('description.txt')

summary = file\_read('summary.txt')

**Use the ‘With’ statement while opening a file, The ‘With’ statement closes the file even If there is an exception raised.**

**For Example:-**

import csv

# opening a file, with statement is used

with open('filename.csv', 'r') as file:

csv\_reader = csv.reader(file)

for line in csv\_reader:

print(line)

**Avoid Magic Numbers and Strings:**

Use constants for better readability and maintainability.

# Bad - Using magic number

if age >= 18 and age <= 65:

pass

# Good - Using named constants

MIN\_AGE = 18

MAX\_AGE = 65

if MIN\_AGE <= age <= MAX\_AGE:

pass

**Use Enumerations for Constants:**

Improve readability by using enumerations for related constants.

# Bad - Using plain constants

MONDAY = 0

TUESDAY = 1

# Good - Using enums

from enum import Enum

class Weekday(Enum):

MONDAY = 0

TUESDAY = 1

**Avoid Nested Loops and Deep Nesting:**

Limit nesting for improved readability.

# Bad - Deep nesting

for i in range(10):

for j in range(10):

if condition(i, j):

do\_something(i, j)

# Good - Reduced nesting

for i in range(10):

if condition(i):

for j in range(10):

if condition(j):

do\_something(i, j)

**String Formatting and F-Strings:**

Use f-strings for concise string formatting.

# Bad - String concatenation

full\_name = first\_name + ' ' + last\_name

# Good - F-strings

full\_name = f'{first\_name} {last\_name}'

**Code Organization Best Practices**

As your projects get larger and more complex, organizing your code becomes even more important. This involves thoughtfully structuring your files and directories, using modules and packages effectively, and following established design patterns. Following the Python best practices for organization will help ensure your code maintains readability and is easier to debug and modify.

* **Group Related Files.** Use modules and packages to organize your code into different groups. A module is a file that will house Python code while a package groups related modules together.
* **Mark Directories as Python Packages.** Use init.py files to indicate your directory is a Python package and use it to execute package initiation code or to specify API.
* **Use Import Statements.** Import statements allow you to use code from one module to another. This is useful for splitting your codebase into logical groups.
* **D.R.Y. Code (Don’t Repeat Yourself).** If you find yourself writing the same code in multiple places, you should organize it into a function or class that can be reused more efficiently while also reducing the probability for errors.
* **Use a Defined Structure.** Structuring larger projects also helps keep them organized.
* **Relative Imports.** Relative imports let you reference code from one module to another.

**Performance Optimization Best Practices**

While Python may not be the fastest programming language, there are several ways you can optimize the code for better performance. This includes writing efficient code by avoiding unnecessary calculations, making good use of data structures and algorithms, and using tools and libraries designed for performance. Make Python better by using the performance best practices below:

* **Use Built-In Functions and Libraries.** Built-in features are already optimized for performance, so use them whenever possible.
* **Use Local Variables.** Local variables are faster than global variables.
* **Use List Comprehensions and Generators.** These are both faster and use less memory than equivalent code written in loops.
* **Use “Slots” in Classes.** Use this function to use less memory when creating multiple instances of a class.
* **Avoid Excess/Unnecessary Data Structures.** Use the code that, as simply as possible, will perform what you need it to. Avoid using options that also perform other functions.

**Security Best Practices**

Security should be a priority in any development project. This means writing secure code, handling sensitive data carefully, and using libraries and tools that help protect against common vulnerabilities. Libraries such as PyCrypto and requests for secure HTTP communication can help you secure your Python applications. Python coding standards and best practices for security include:

* **Input Validation.** Validate and sanitize all user input to help protect against SQL injection and cross-site scripting.
* **Secure Library Use.** Only use secure and well-maintained Python libraries when it comes to tasks related to security.
* **HTTPS for Web Apps.** Always use HTTPS instead of HTTP when building a web application to ensure data is encrypted.
* **Limit Exec and Eval Use.** These functions can execute arbitrary code, which can be a security risk.
* **Don’t Hard Code Sensitive Information.** Never include private information in the hard code; use variables or secure configuration files instead.

Security when creating code of any kind, including Python, is paramount. For one, it ensures data protection for sensitive information, such as personal details and financial information. It also helps secure system integrity and instill trust in users. Adhering to code security standards also ensures legal compliance with regulations like GDPR and HIPAA.

Popular Secure Python Libraries

Using secure libraries will help protect your code from malicious attacks. Some of the most popular and best secure Python libraries include [pyOpenSSL](https://www.pyopenssl.org/en/stable/) for SSL/TLS protocol, [bcrypt](https://pypi.org/project/bcrypt/) for password hashing, [Paramiko](https://www.paramiko.org/) for SSH2 protocol, and [cryptography](https://cryptography.io/en/latest/) for primitives.

**Scalability Best Practices**

Finally, as your application grows, you should also be thinking about scalability. This means designing your systems to handle increased loads effectively, organizing your code for large code bases, and leveraging Python's features and libraries that help with scalability. Here are some Python best practices for scalability:

* **Use Efficient Data Structures and Algorithms.** Choosing the right [data structures and algorithms](https://www.appacademy.io/blog/top-algorithms-and-data-structures-you-really-need-to-know) can greatly affect the scalability of your program.
* **Concurrent and Parallel Execution.** Python has several libraries, such as [concurrent futures](https://docs.python.org/3/library/concurrent.futures.html) and [multiprocessing](https://docs.python.org/3/library/multiprocessing.html?highlight=multiprocessing#module-multiprocessing), that allow for concurrent or parallel execution of tasks to utilize multiple CPU cores and threads.
* **Asynchronous Processing.** For I/O-bound tasks, using asynchronous processing can make your program more scalable. [Python's asyncio library](https://docs.python.org/3/library/asyncio.html) provides tools for writing single-threaded concurrent code.
* **Cache Data.** Caching can improve your application's performance and scalability by storing the result of expensive operations and reusing them.
* **Use a Load Balancer.** For web applications, use a load balancer to distribute traffic among multiple instances of your application.