Python coding standard \_\_\_

Introduction\_\_\_

**Coding standards**

**Coding standards** are collections of rules and **guidelines** that determine the programming style, procedures, and methods defined for a specific language to be followed for each piece of code in a programming language.  
 These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc.

**purpose of using coding standard\_\_\_\_**

1. A coding standard gives a uniform appearance to the codes written by different engineers.
2. It improves readability, and maintainability of the code and it reduces complexity also.
3. It helps in code reuse and helps to detect error easily.
4. It promotes sound programming practices and increases efficiency of the programmers.
5. 40%–80% of the lifetime cost of a piece of software goes to maintenance.[[3]](https://en.wikipedia.org/wiki/Coding_conventions#cite_note-3)
6. Hardly any software is maintained for its whole life by the original author.
7. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
8. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
9. Coding guidelines increase the efficiency of the software and reduces the development time.
10. Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
11. It reduces the hidden cost for developing the software.

Coding standard of python we will be following\_\_\_

**1.Indentation\_\_**

**Use 4-space indentation and no tabs.**

Examples:

# Aligned with opening delimiter.

grow = function\_name(variable\_one, variable\_two,

variable\_three, variable\_four)

# First line contains no argument. Second line onwards

# more indentation included to distinguish this from

# the rest.

def function\_name(

variable\_one, variable\_two, variable\_three,

variable\_four):

print(variable\_one)

The 4 space rule is not always mandatory and can be overruled for continuation line.

**Indentation Following Line Breaks**

When you’re using line continuations to keep lines to under 79 characters, it is useful to use indentation to improve readability. It allows the reader to distinguish between two lines of code and a single line of code that spans two lines. There are two styles of indentation you can use.

The first of these is to align the indented block with the opening delimiter: example\_\_

def function(arg\_one, arg\_two,

arg\_three, arg\_four):

return arg\_one

Sometimes you can find that only 4 spaces are needed to align with the opening delimiter. This will often occur in if statements that span multiple lines as the if, space, and opening bracket make up 4 characters. In this case, it can be difficult to determine where the nested code block inside the if statement begins:

x = 5

if (x > 3 and

x < 10):

print(x)

**2.use of Comments**

use of comments to document code as it’s written. It is important to document our code so that we, and any collaborators, can understand it. When we or someone else reads a comment, they should be able to easily understand the code the comment applies to and how it fits in with the rest of your code.

Here are some when adding comments to code:

the line length of comments and docstrings to 72 characters.

complete sentences, starting with a capital letter.

update comments if code is chnaged

Block Comments

Use of block comments to document a small section of code. They are useful when we have to write several lines of code to perform a single action, such as importing data from a file or updating a database entry. They are important as they help others understand the purpose and functionality of a given code block.

PEP 8 provides the following rules for writing block comments:

Indent block comments to the same level as the code they describe.

Start each line with a # followed by a single space.

Separate paragraphs by a line containing a single #.

Here is a block comment explaining the function of a for loop. Note that the sentence wraps to a new line to preserve the 79 character line limit:

for i in range(0, 10):

# Loop over i ten times and print out the value of i, followed by a

# new line character

print(i, '\n')

Sometimes, if the code is very technical, then it is necessary to use more than one paragraph in a block comment:

def quadratic(a, b, c, x):

# Calculate the solution to a quadratic equation using the quadratic

# formula.

#

# There are always two solutions to a quadratic equation, x\_1 and x\_2.

x\_1 = (- b+(b\*\*2-4\*a\*c)\*\*(1/2)) / (2\*a)

x\_2 = (- b-(b\*\*2-4\*a\*c)\*\*(1/2)) / (2\*a)

return x\_1, x\_2

Inline Comments

Inline comments explain a single statement in a piece of code. They are useful to remind us, or explain to others, why a certain line of code is necessary. Here’s what PEP 8 has to say about them:

inline comments to be used sparingly.

Write inline comments on the same line as the statement they refer to.

Separate inline comments by two or more spaces from the statement.

Start inline comments with a # and a single space, like block comments.

Below is an example of an inline comment:

x = 5 # This is an inline comment

Sometimes, inline comments can seem necessary, but we can use better naming conventions instead. Here’s an example:

x = 'John Smith' # Student Name

Here, the inline comment does give extra information. variable:

student\_name = 'John Smith'

**3.Documentation Strings use**

Documentation strings, or docstrings, are strings enclosed in double (""") or single (''') quotation marks that appear on the first line of any function, class, method, or module. we can use them to explain and document a specific block of code. There is an entire PEP, PEP 257, that covers docstrings, but you’ll get a summary in this section.

The most important rules applying to docstrings are the following:

Surround docstrings with three double quotes on either side, as in """This is a docstring""".

Write them for all public modules, functions, classes, and methods.

Put the """ that ends a multiline docstring on a line by itself:

def quadratic(a, b, c, x):

"""Solve quadratic equation via the quadratic formula.

A quadratic equation has the following form:

ax\*\*2 + bx + c = 0

There always two solutions to a quadratic equation: x\_1 & x\_2.

"""

x\_1 = (- b+(b\*\*2-4\*a\*c)\*\*(1/2)) / (2\*a)

x\_2 = (- b-(b\*\*2-4\*a\*c)\*\*(1/2)) / (2\*a)

return x\_1, x\_2

**4.Use of spaces for comma’s\_\_\_**

For comma’s example\_\_a = f(1, 2) + g(3, 4)

For operator\_\_

**5.Wrap of lines so that they don’t exceed 79 characters :**The Python standard library is conservative and requires limiting lines to 79 characters. The lines can be wrapped using parenthesis, brackets, and braces. They should be used in preference to backslashes.  
Example:

with open('/path/from/where/you/want/to/read/file') as file\_one, \

open('/path/where/you/want/the/file/to/be/written', 'w') as file\_two:

file\_two.write(file\_one.read())

**6.Use of Python’s default *UTF-8* or *ASCII* encodings and not any fancy encodings**, if it is meant for international environment.

**7.imports\_\_**

Imports should usually be on separate lines. Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

For example\_\_

# Correct:

from subprocess import Popen, PIPE

**8. Naming Conventions :**

**Choosing name \_\_\_**

Choosing names for our variables, functions, classes, and so forth can be challenging. we should put a fair amount of thought into your naming choices when writing code as it will make our code more readable. The best way to name our objects in Python is to use descriptive names to make it clear what the object represents.

When naming variables, we may be tempted to choose simple, single-letter lowercase names, like x. But, unless you’re using x as the argument of a mathematical function, it’s not clear what x represents. Imagine you are storing a person’s name as a string, and we want to use string slicing to format their name differently. we could end up with something like this:

name = 'John Smith'

first\_name, last\_name = name.split()

print(last\_name, first\_name, sep=', ')

'Smith, Joh

**Class Names­­\_\_\_**

Class names should normally use the CapWords convention and camelcase

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

class MyClass:

x = 5

**Type Variable Names**

Names of type variables introduced in PEP 484 should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes \_co or \_contra to the variables used to declare covariant or contravariant behavior correspondingly

from typing import TypeVar

VT\_co = TypeVar('VT\_co', covariant=True)

KT\_contra = TypeVar('KT\_contra', contravariant=True)

**Exception Names**

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

**Function Names**

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

For example

def my\_function():

print("Hello from a function")

**Variable Names**

Use a lowercase single letter, word, or words. Separate words with underscores to improve readability..

For example\_\_

x, var, my\_variable

**method name\_\_\_\_**

Use a lowercase word or words. Separate words with underscores to improve readability.

For example\_\_\_

class class\_name

def method\_name () :

......

# method body

......

**Constant name\_\_\_**

Use an uppercase single letter, word, or words. Separate words with underscores to improve readability

For example\_\_ CONSTANT, MY\_CONSTANT, MY\_LONG\_CONSTANT.

**Module name\_\_\_**

Use a short, lowercase word or words. Separate words with underscores to improve readability

For example\_\_ module.py, my\_module.py

**9.Control structures in python\_\_\_**

**For making descions ,brancing** if /if-else /if-elif-el/while/for are used with colon and proper indentation and brancing and break should be used .bracets should not be use at al any cost..

For example

**Selections\_\_\_**

a = 33  
b = 33  
if b > a:  
  print("b is greater than a")  
elif a == b:  
  print("a and b are equal")

loops\_\_\_

i = 1  
while i < 6:  
  print(i)  
  i += 1

**10. Characters that should not be used for identifiers :**‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names as these are similar to the numerals one and zero.

**11. non-ASCII characters in identifiers**if there is only the slightest chance people speaking a different language will read or maintain the code.

**12. While naming of function of methods always use *self* for the first argument** to instance methods and *cls* for the first argument to class methods.If a functions argument name matches with reserved words then it can be written with a trailing comma. For e.g., class\_

we can refer to this simple program to know how to write an understandable code:

|  |
| --- |
| # Python program to find the  # factorial of a number provided by the user.    # change the value for a different result  num = 7    # uncomment to take input from the user  #num = int(input("Enter a number: "))    factorial = 1    # check if the number is negative, positive or zero  if num < 0:      print("Sorry, factorial does not exist for negative numbers")  elif num == 0:      print("The factorial of 0 is 1")  else:      for i in range(1,num + 1):          factorial = factorial\*i    print("The factorial of",num,"is",factorial)  **13.file naming convention in python\_\_\_**  modules (filenames) should have *short, all-lowercase names*, and they can contain underscores; |