Python Coding Standards

A guide for Tutor-Hub Developers Team

--Prepared By: Sayed Abu Noman Siddik

Table of Contents

Naming Conventions

Naming Styles

The table below outlines some of the common naming styles in Python code and when you should use them:

Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	function, my_function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case.	Model, MyClass
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	class_method, method
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	module.py, my_module.py
Package	Use a short, lowercase word or words. Do not separate words with underscores.	package, mypackage

How to Choose Names ?

General Tips to Begin With

These tips can be applied to name any entity and should be followed **religiously**.

- Try to **follow the same pattern** – consistency is the key!

```
thisVariable, ThatVariable, some_other_variable, BIG_NO
```

- **Avoid using long names** while not being too frugal with the name either

```
this_could_be_a_bad_name = "Avoid this!"
t = "This isn\'t good either"
```

- Use sensible and descriptive names. This will help later on when you try to remember the purpose of the code

```
X = "My Name"  # Avoid this
full_name = "My Name"  # This is much better
```

- Avoid using names that begin with numbers

```
1_name = "This is bad!"
```

- Avoid using special characters like @, !, #, $, etc. in names

```
phone_ # Bad name
```

Naming Variables

- Variable names should always be in **lowercase**

```
blog = "Analytics Vidhya"
```

- For longer variable names, include **underscores** to separate_words. This improves readability

```
blog = "Analytics Vidhya"
```

- Try not to use single-character variable names like **'I'** (uppercase i letter), **'O'** (uppercase o letter), **'l'** (lowercase L letter). They can be indistinguishable from numerical 1 and 0. Have a look:

```
O = 0 + l + I + 1
```

- Follow the same naming convention for Global variables

Naming Functions

- Follow the lowercase with underscores naming convention
- Use expressive names

```
# Avoid
def con():
    ...
# This is better.
def connect():
    ...
```

- If a function argument name clashes with a keyword, use a trailing underscore instead of using an abbreviation. For example, turning *break* into *break_* instead of *brk*

```
# Avoiding name clashes.
def break_time(break_):
    print("Your break time is", break_,"long")
```

Class names

- Follow CapWord (or camelCase or StudlyCaps) naming convention. Just start each word with a capital letter and do not include underscores between words

```
# Follow CapWord convention
class MySampleClass:
    pass
```

- If a class contains a subclass with the same attribute name, consider adding double underscores to the class attribute

This will make sure the attribute *__age* in class *Person* is accessed as ***_Person__age***. This is Python's name mangling and it makes sure there is no name collision

```
class Person:
    def __init__(self):
        self.__age = 18

obj = Person()
obj.__age  # Error
obj._Person__age  # Correct
```

- Use the suffix “Error” for exception classes

```python
class CustomError(Exception):
    """Custom exception class"""
```

Naming Class Methods

- The first argument of an instance method (the basic class method with no strings attached) should always be *self*. This points to the calling object
- The first argument of a class method should always be *cls*. This points to the class, not the object instance

```python
class SampleClass:
    def instance_method(self, del_):
        print("Instance method")

    @classmethod
    def class_method(cls):
        print("Class method")
```

Package and Module names

- Try to keep the name short and simple
- The lowercase naming convention should be followed
- Prefer underscores for long module names
- Avoid underscores for package names

```
testpackage # package name
sample_module.py # module name
```

Constant names

- Constants are usually declared and assigned values within a module
- The naming convention for constants is an aberration. Constant names should be all CAPITAL letters
- Use underscores for longer names

```python
# Following constant variables in global.py module
PI = 3.14
GRAVITY = 9.8
SPEED_OF_Light = 3*10**8
```

Code Layout

Blank Lines

Vertical whitespace, or blank lines, can greatly improve the readability of your code. Code that's bunched up together can be overwhelming and hard to read. Similarly, too many blank lines in your code makes it look very sparse, and the reader might need to scroll more than necessary. Below are three key guidelines on how to use vertical whitespace.

Surround top-level functions and classes with two blank lines:

- Top-level functions and classes should be separated by two blank lines

```python
# Separating classes and top level functions.
class SampleClass():
    pass


def sample_function():
    print("Top level function")
```

Surround method definitions inside classes with a single blank line:

- Methods inside a class should be separated by a single blank line

```python
# Separating methods within class.
class MyClass():
    def method_one(self):
        print("First method")

    def method_two(self):
        print("Second method")
```

Use blank lines sparingly inside functions to show clear steps:

- Blank lines can be used sparingly within functions to separate logical sections. This makes it easier to comprehend the code

```
def remove_stopwords(text):
    stop_words = stopwords.words("english")
    tokens = word_tokenize(text)
    clean_text = [word for word in tokens if word not in
stop_words]

    clean_text = ' '.join(clean_text)
    clean_text = clean_text.lower()

    return clean_text
```

- Try not to include blank lines between pieces of code that have related logic and function

```
def remove_stopwords(text):
    stop_words = stopwords.words("english")
    tokens = word_tokenize(text)
    clean_text = [word for word in tokens if word not in
stop_words]

    return clean_text
```

Maximum Line Length and Line Breaking

- No more than 79 characters in a line

When you are writing code in Python, you cannot squeeze more than 79 characters into a single line. That's the limit and should be the guiding rule to keep the statement short.

- You can break the statement into multiple lines and turn them into shorter lines of code

```
# Breaking into multiple lines.
num_list = [y for y in range(100)
            if y % 2 == 0
            if y % 5 == 0]
print(num_list)
```

If it is impossible to use implied continuation, then you can use backslashes to break lines instead:

```
from mypkg import example1, \
    example2, example3
```

However, if you can use implied continuation, then you should do so.

If line breaking needs to occur around binary operators, like + and *, it should occur before the operator. This rule stems from mathematics. Mathematicians agree that breaking before binary operators improves readability. Compare the following two examples.

Below is an example of breaking before a binary operator:

```
# Recommended
total = (first_variable
         + second_variable
         - third_variable)
```

You can immediately see which variable is being added or subtracted, as the operator is right next to the variable being operated on.

Now, let's see an example of breaking after a binary operator:

```
# Not Recommended
total = (first_variable +
         second_variable -
         third_variable)
```

Here, it's harder to see which variable is being added and which is subtracted.

Breaking before binary operators produces more readable code, so PEP 8 encourages it. Code that *consistently* breaks after a binary operator is still PEP 8 compliant. However, you're encouraged to break before a binary operator.

Indentation

Tabs vs. Spaces

It is the single most important aspect of code layout and plays a vital role in Python. Indentation tells which lines of code are to be included in the block for execution. Missing an indentation could turn out to be a critical mistake.

Indentations determine which code block a code statement belongs to. Imagine trying to write up a nested for-loop code. Writing a single line of code outside its respective loop may not give you a syntax error, but you will definitely end up with a logical error that can be potentially time-consuming in terms of debugging.

Follow the below mentioned key points on indentation for a consistent structure for your Python scripts.

- Always follow the **4-space** indentation rule

```python
# Example
if value<0:
    print("negative value")

# Another example
for i in range(5):
    print("Follow this rule religiously!")
```

- Prefer to use spaces over tabs

It is recommended to use Spaces over Tabs. But Tabs can be used when the code is already indented with tabs.

```python
if True:
    print('4 spaces of indentation used!')
```

- Break large expressions into several lines

There are several ways of handling such a situation. One way is to align the succeeding statements with the opening delimiter.

```
# Aligning with opening delimiter.
def name_split(first_name,
               middle_name,
               last_name)

# Another example.
ans = solution(value_one, value_two,
               value_three, value_four)
```

A second way is to make use of the 4-space indentation rule. This will require an extra level of indentation to distinguish the arguments from the rest of the code inside the block.

```
# Making use of extra indentation.
def name_split(
        first_name,
        middle_name,
        last_name):
    print(first_name, middle_name, last_name)
```

Finally, you can even make use of **"hanging indents"**. Hanging indentation, in the context of Python, refers to the text style where the line containing a parenthesis ends with an opening parenthesis. The subsequent lines are indented until the closing parenthesis.

```
# Hanging indentation.
ans = solution(
    value_one, value_two,
    value_three, value_four)
```

- Indenting **if-statements** can be an issue

if-statements with multiple conditions naturally contain 4 spaces – if, space, and the opening parenthesis. As you can see, this can be an issue. Subsequent lines will also be indented and there is no way of differentiating the if-statement from the block of code it executes. Now, what do we do?

Well, we have a couple of ways to get our way around it:

```
# This is a problem.
if (condition_one and
    condition_two):
    print("Implement this")
```

One way is to use an extra level of indentation of course!

```
# Use extra indentation.
if (condition_one and
        condition_two):
    print("Implement this")
```

Another way is to add a comment between the if-statement conditions and the code block to distinguish between the two:

```
# Add a comment.
if (condition_one and
    condition_two):
    # this condition is valid
    print("Implement this")
```

Indentation Following Line Breaks

- Break line before binary operators

If you are trying to fit too many operators and operands into a single line, it is bound to get cumbersome. Instead, break it into several lines for better readability.

Now the obvious question – break before or after operators? The convention is to break before operators. This helps to easily make out the operator and the operand it is acting upon.

```
# Break lines before the operator.
gdp = (consumption
      + government_spending
      + investment
      + net_exports
      )
```

Where to Put the Closing Brace

- Brackets need to be closed

Let's say you have a long dictionary of values. You put all the key-value pairs in separate lines but where do you put the closing bracket? Does it come in the last line? The line following it? And if so, do you just put it at the beginning or after indentation?

There are a couple of ways around this problem as well.

One way is to align the closing bracket with the first non-whitespace character of the previous line.

```
#
learning_path = {
    'Step 1' : 'Learn programming',
    'Step 2' : 'Learn machine learning',
    'Step 3' : 'Crack on the hackathons'
    }
```

The second way is to just put it as the first character of the new line.

```
learning_path = {
    'Step 1' : 'Learn programming',
    'Step 2' : 'Learn machine learning',
    'Step 3' : 'Crack on the hackathons'
}
```

You are free to choose which option you use. But, as always, consistency is key, so try to stick to one of the above methods.

Comments

Getting Familiar with Proper Python Comments

Understanding an uncommented piece of code can be a strenuous activity. Even for the original writer of the code, it can be difficult to remember what exactly is happening in a code line after a period of time.

Therefore, it is best to comment on your code then and there so that the reader can have a proper understanding of what you tried to achieve with that particular piece of code.

Here are some key points to remember when adding comments to your code:

- Limit the line length of comments and docstrings to 72 characters.
- Use complete sentences, starting with a capital letter.
- Make sure to update comments if you change your code.

Block Comments

Use block comments to document a small section of code. They are useful when you 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
```

If you're ever in doubt as to what comment type is suitable, then block comments are often the way to go. Use them as much as possible throughout your code, but make sure to update them if you make changes to your code!

Inline Comments

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

- Use inline comments 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.
- Don't use them to explain the obvious.

Below is an example of an inline comment:

```python
x = 5  # This is an inline comment
```

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

```python
x = 'John Smith'  # Student Name
```

Here, the inline comment does give extra information. However using x as a variable name for a person's name is bad practice. There's no need for the inline comment if you rename your variable:

```python
student_name = 'John Smith'
```

Finally, inline comments such as these are bad practice as they state the obvious and clutter code:

```python
empty_list = []  # Initialize empty list

x = 5
x = x * 5  # Multiply x by 5
```

Inline comments are more specific than block comments, and it's easy to add them when they're not necessary, which leads to clutter. You could get away with only using block comments so, unless you are sure you need an inline comment, your code is more likely to be PEP 8 compliant if you stick to block comments.

Documentation Strings

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. You 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 multi line docstring on a line by itself:

```
def quadratic(a, b, c, x):
    """Solve quadratic equations via the quadratic formula.

    A quadratic equation has the following form:
    ax**2 + bx + c = 0

    There are 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
```

- For one-line docstrings, keep the `"""` on the same line:

```
def quadratic(a, b, c, x):
    """Use the quadratic formula"""
    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
```

Imports

Part of the reason why a lot of data scientists love to work with Python is because of the plethora of libraries that make working with data a lot easier. Therefore, it is given that you will end up importing a bunch of libraries and modules to accomplish any task in data science.

- Should always come at the top of the Python script
- Separate libraries should be imported on separate lines

```python
import numpy as np
import pandas as pd

df = pd.read_csv(r'/sample.csv')
```

- Imports should be grouped in the following order:
    - Standard library imports
    - Related third party imports
    - Local application/library specific imports
- Include a blank line after each group of imports

```python
import numpy as np
import pandas as pd
import matplotlib
from glob import glob
import spaCy
import mypackage
```

- Can import multiple classes from the same module in a single line

```python
from math import ceil, floor
```

Whitespace in Expressions and Statements

Whitespace can be very helpful in expressions and statements when used properly. If there is not enough whitespace, then code can be difficult to read, as it's all bunched together. If there's too much whitespace, then it can be difficult to visually combine related terms in a statement

Whitespace Around Binary Operators

Surround the following binary operators with a single space on either side:

- Assignment operators (=, +=, -=, and so forth)
- Comparisons (==, !=, >, <. >=, <=) and (is, is not, in, not in)
- Booleans (and, not, or)

Note: When = is used to assign a default value to a function argument, do not surround it with spaces.

```python
# Recommended
def function(default_parameter=5):
    # ...


# Not recommended
def function(default_parameter = 5):
    # ...
```

When there's more than one operator in a statement, adding a single space before and after each operator can look confusing. Instead, it is better to only add whitespace around the operators with the lowest priority, especially when performing mathematical manipulation. Here are a couple examples:

```
# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not Recommended
y = x ** 2 + 5
z = (x + y) * (x - y)
```

You can also apply this to if statements where there are multiple conditions:

```
# Not recommended
if x > 5 and x % 2 == 0:
    print('x is larger than 5 and divisible by 2!')
```

In the above example, the and operator has lowest priority. It may therefore be clearer to express the if statement as below:

```
# Recommended
if x>5 and x%2==0:
    print('x is larger than 5 and divisible by 2!')
```

You are free to choose which is clearer, with the caveat that you must use the same amount of whitespace either side of the operator.

The following is not acceptable:

```
# Definitely do not do this!
if x >5 and x% 2== 0:
    print('x is larger than 5 and divisible by 2!')
```

In slices, colons act as binary operators. Therefore, the rules outlined in the previous section apply, and there should be the same amount of whitespace either side. The following examples of list slices are valid:

```
list[3:4]

# Treat the colon as the operator with lowest priority
list[x+1 : x+2]

# In an extended slice, both colons must be
# surrounded by the same amount of whitespace
list[3:4:5]
list[x+1 : x+2 : x+3]

# The space is omitted if a slice parameter is omitted
list[x+1 : x+2 :]
```

In summary, you should surround most operators with whitespace. However, there are some caveats to this rule, such as in function arguments or when you're combining multiple operators in one statement.

When to Avoid Adding Whitespace

In some cases, adding whitespace can make code harder to read. Too much whitespace can make code overly sparse and difficult to follow. PEP 8 outlines very clear examples where whitespace is inappropriate.

The most important place to avoid adding whitespace is at the end of a line. This is known as trailing whitespace. It is invisible and can produce errors that are difficult to trace.

The following list outlines some cases where you should avoid adding whitespace:

Immediately inside parentheses, brackets, or braces:

```
# Recommended
my_list = [1, 2, 3]

# Not recommended
my_list = [ 1, 2, 3, ]
```

- Before a comma, semicolon, or colon:

```python
x = 5
y = 6

# Recommended
print(x, y)

# Not recommended
print(x , y)
```

- Before the open parenthesis that starts the argument list of a function call:

```python
def double(x):
    return x * 2

# Recommended
double(3)

# Not recommended
double (3)
```

- Before the open bracket that starts an index or slice:

```python
# Recommended
list[3]

# Not recommended
list [3]
```

- Between a trailing comma and a closing parenthesis:

```
# Recommended
tuple = (1,)

# Not recommended
tuple = (1, )
```

- To align assignment operators:

```
# Recommended
var1 = 5
var2 = 6
some_long_var = 7

# Not recommended
var1          = 5
var2          = 6
some_long_var = 7
```

Make sure that there is no trailing whitespace anywhere in your code. There are other cases where PEP 8 discourages adding extra whitespace, such as immediately inside brackets, as well as before commas and colons. You should also never add extra whitespace in order to align operators.

References:

https://wikileaks.org/ciav7p1/cms/page_26607631.html
https://realpython.com/python-pep8/#when-to-avoid-adding-whitespace
https://www.analyticsvidhya.com/blog/2020/07/python-style-guide/
https://realpython.com/python-pep8/
https://www.datacamp.com/community/tutorials/pep8-tutorial-python-code