Hue Learn

Wise Work

DNAi World

Python Coding Guidelines

Best Practices to follow

by Hue Learn, Wise Work, DNAi World

Contents

1. Introduction

Python programming is incredibly versatile. It's coding best practices ensures that your code is consistently clean and readable. It encourages code reusability, reduces the likelihood of bugs, and makes it easier to maintain and restructure. By adhering to these standards and practices developers can harness the full potential of Python's elegant syntax.

Here are some widely accepted coding guidelines for Python:

2. Naming Conventions

- Formatting & Syntax Best Practices

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, python_function

Type	Naming Convention	Examples
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, python_variable
Class	Start each word with a capital letter. Don't separate words with underscores. Use Pascal case style.	Model, PythonClass
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,PYTHON_CONSTANT,PYTHON_LONG_CONSTANT
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	module.py, python_module.py
Package	Use a short, lowercase word or words. Don't separate words with underscores.	package, pythonpackage

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.

1. **Block Comments**

Use block comments to document a small section of code. Rules for writing block comments:

- Indent block comments to the same level as the code that they describe.
- Start each line with a # followed by a single space.
- Separate paragraphs by a line containing a single #.

Right Coding Style.

```
# Calculate the solution to a quadratic equation using the quadratic

# formula.

# A quadratic equation has the following form:

#ax**2 + bx + c = 0
```

Right Coding Style.

```
# 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)
```

2. **Inline Comments**

- Use inline comments sparingly.
- Write inline comments on the same line as the statement they refer to.
- Separate inline comments from the statement by two or more spaces.
- Start inline comments with a # and a single space, like block comments.

Right Coding Style.

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

- Don't use them to explain the obvious.

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

Wrong Coding Style.

```
empty_list = [] # Initialize empty list x = 5

x = x * 5 # Multiply x by 5

X = "John Smith" # Student Name
```

Right Coding Style.

```
student name = "John Smith"
```

Neither of the two comments adds information that the code doesn't already clearly show. It's better to avoid writing such comments.

3. Documentation Strings

- Documentation strings, or docstrings, are strings enclosed in triple double quotation marks (""") or triple single quotation marks (''') that appear on the first line of any function, class, method, or module.
- Write docstrings for all public modules, functions, classes, and methods.
- A docstring should be a short description of the function or class followed by an explanation of the parameters and return values.
- Can access the docstring of an object using its "*__doc__*" attribute or the help() function.

- The main takeaway is that docstrings are a structured approach to documenting your Python code.

You use docstrings to explain and document a specific block of code. They're an important part of Python,: You should write them for all public modules, functions, classes, and methods.

If the implementation is straightforward, then you can use a one-line docstring, where you can keep the whole docstring on the same line.

Right Coding Style.

```
def adder(a,b):
  """Add a to b"""
  return a+b
```

you should put the three quotation marks that end a multiline docstring on a line by themselves:

Right Coding Style.

```
def quadratic(a, b, c):
    """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
```

3. Whitespace in Expressions and Statements

1. Whitespace can be very helpful in expressions and statements

When you use the equal sign (=) to assign a default value to an argument, don't surround it with spaces.

Right Coding Style.

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

Wrong Coding Style.

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

Avoiding whitespace for indicating default values for arguments keeps function and method definitions more concise. When there's more than one operator in a statement, then it can look confusing if you add a single space before and after each operator. Instead, it's better to only add whitespace around the operators with the lowest priority, especially when performing mathematical manipulation. Here are a couple of examples.:

Right Coding Style.

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

Wrong Coding Style.

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

If you use whitespace to group multiple operators according to their operator precedence, then it'll make your code better readable. You can also apply this to if statements where there are multiple conditions.

Wrong Coding Style.

```
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:

Right Coding Style.

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

You're free to choose the one that's clearer, with the caveat that you must use the same.

The following isn't acceptable.

Wrong Coding Style.

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

In this example, you're using inconsistent amounts of whitespace on either side of the operators n slices, colons act as binary operators. Therefore, the rules outlined in the previous section apply, and there should be the same amount of whitespace on either side. The following examples of list slices are valid.

Right Coding Style.

```
a_list[3:4]

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

# In an extended slice, you must surround both colons
# with the same amount of whitespace.
a_list[3:4:5]
a_list[x+1 : x+2 : x+3]

# You omit the space if you omit a slice parameter.
a_list[x+1 : x+2 :]
```

In summary, you should surround most operators with whitespace.

2. When to Avoid Adding Whitespace

The most important place to avoid adding whitespace is at the end of a line. This is known as trailing whitespace.

Right Coding Style.

```
x = 1 + 2 + \
    3 + 4
```

In the example file above, you attempted to continue the assignment expression over two lines using the line continuation marker. However, you left a trailing whitespace after the backslash () and before the newline character.

This trailing whitespace prevents Python from understanding it as a line-continuation marker and will cause a syntax error.

Right Coding Style.

```
$ python trailing_whitespace.py
  File "trailing_whitespace.py", line 1
    x = 1 + 2 + \
                ^
SyntaxError: unexpected character after line continuation character
```

While Python will notice the problem and inform you about it, it's best practice to just avoid any trailing whitespace in your Python code.

- Immediately inside parentheses, brackets, or braces.

Right Coding Style.

```
numbers = [1, 2, 3]
```

Wrong Coding Style.

```
numbers = [ 1, 2, 3, ]
```

- Before a comma, semicolon, or colon:

Right Coding Style.

```
x=5
y=6

print(x, y)
```

Wrong Coding Style.

```
print(x , y)
```

Right Coding Style.

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

- Before the opening parenthesis that starts the argument list of a function call.

Right Coding Style.

```
double(3)
```

Wrong Coding Style.

```
double (3)
```

- Before the open bracket that starts an index or slice

Right Coding Style.

```
a_list[3]
```

Wrong Coding Style.

```
a_list [3]
```

- Between a trailing comma and a closing parenthesis.

Right Coding Style.

```
a_list [3]
```

Wrong Coding Style.

```
a_tuple = (1, )
```

- To align assignment operators:

Right Coding Style.

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

Wrong Coding Style.

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

The most important takeaway is to make sure that there's no trailing whitespace anywhere in your code. You also shouldn't add extra whitespace in order to align operators.

4. Programming Recommendations

1. Don't compare Boolean values to True or False using the equivalence operator. You'll often need to check if a Boolean value is true or false. You may want to do this with a statement like the one below:

Wrong Coding Style.

```
is_bigger = 6 > 5
if is_bigger == True:
    return "6 is bigger than 5"
```

The use of the equivalence operator (==) is unnecessary here. bool can only take values True or False. It's enough to write the following:

Right Coding Style.

```
is_bigger = 6 > 5
if is_bigger:
    return "6 is bigger than 5"
```

This way of performing an if statement with a Boolean requires less code and is simpler.

2. Use the fact that empty sequences are false in if statements.

If you want to check whether a list is empty, you might be tempted to check the length of the list. If the list is empty, then its length is 0 which is equivalent to False when you use it in an if statement. Here's an example:

Wrong Coding Style.

```
a_list = []
if not len(a_list):
    print("List is empty
```

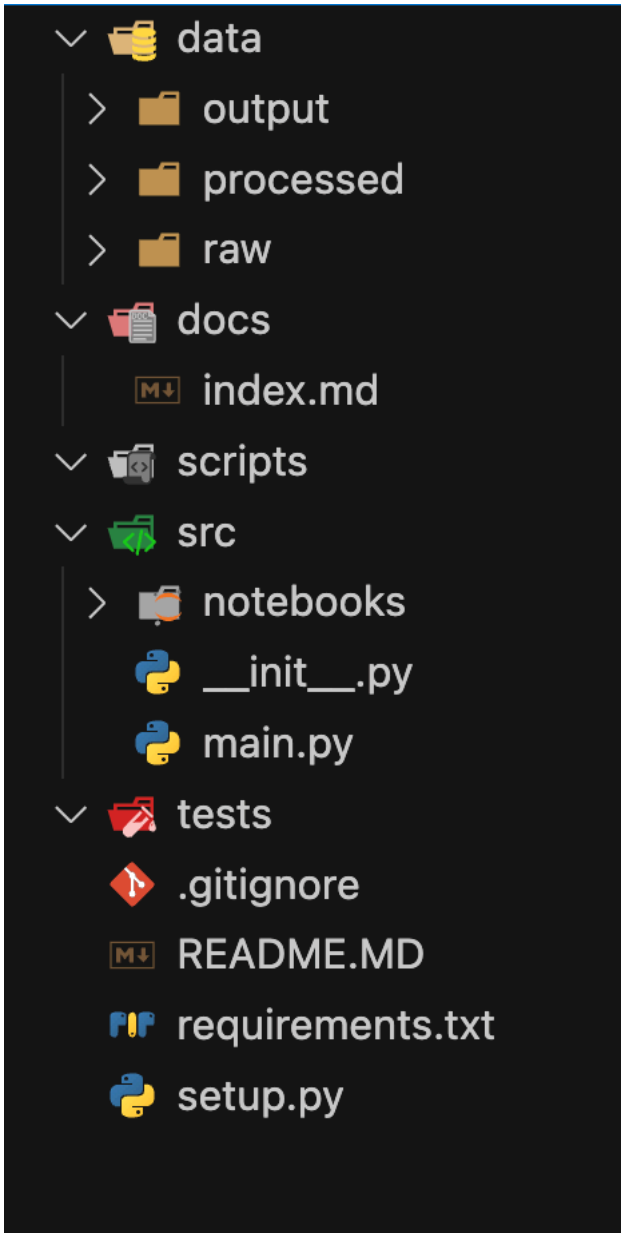
However, in Python any empty list, string, or tuple is false. You can therefore come up with a simpler alternative to the above.

Right Coding Style.

```
a_list = []
if not a_list:
    print("List is empty!")
```

While both examples will print out List is empty!, the second option is more straightforward to read and understand, so PEP 8 encourages it.

5. Folder Structure for Python Project



6. Formatter

A tool that automatically adjusts the structure and layout of your code to conform to a specific coding style or set of guidelines. It standardizes aspects such as indentation, line spacing, and the placement of brackets, making the code easier to read and maintain across teams. Formatters are especially useful in enforcing consistency, which reduces cognitive load when reading code and minimizes conflicts in code reviews.

6.1. Black

The most popular Python code formatter that ensures consistent styling by adhering to PEP 8 standards. It's widely accepted by the Python community for its strict formatting.

6.2. autopep8

A formatter that automatically formats Python code to follow PEP 8 guidelines. It's more customizable than Black but still widely used.

In Python, a formatter like Black will convert:

Wrong Coding Style.

```
def  add( x, y ):return  x+y
```

Right Coding Style.

```
def add(x, y):
    return x + y
```

7. Linter

A linter is a static code analysis tool that checks your code for programming errors, bugs, stylistic inconsistencies, and potentially problematic patterns. Linters don't modify your code, instead, they provide warnings or errors that guide you to fix problems.

7.1. Flake8

A highly extensible linter that checks Python code for PEP 8 compliance, errors, and complexity issues. It can be combined with plugins for additional rules.

Flake8 might flag a warning for missing a new line at the end of the file or not following PEP 8 spacing standards.

Wrong Coding Style.

```
def foo():
    print("Hello")
foo()
```

7.2. Pylint

A comprehensive linter that analyzes Python code for errors, enforces coding standards, and suggests code improvements. It's highly configurable and can integrate with IDEs.

8. Type Checkers

8.1. mypy

is a type checker for Python that helps enforce type annotations and catches potential bugs in statically typed Python code.

9. Code Quality Tools

SonarQube is a tool that analyzes code quality and security vulnerabilities across multiple languages. It combines the functionality of both linters and static code analysis to offer in-depth reports.

10. Conclusion

Incorporating these guidelines and tools into your workflow will lead to more maintainable, scalable, and bug-free Python code, fostering better collaboration and smoother project development.