PEP 8: The Style Guide for Python Code A Quick Reference Cheatsheet

Guiding Principles

The core philosophy of PEP 8 is that code is read much more often than it is written. The guidelines provided are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.
Core Tenet: *"Readability counts."* (from PEP 20, The Zen of Python)
Hierarchy of Consistency: A style guide is about consistency. The hierarchy of importance is:

1. Consistency within a single function or method.
2. Consistency within a single module or file.
3. Consistency within a project.
4. Consistency with the PEP 8 guide.

When to Be Inconsistent: Know when to ignore a guideline. Good reasons include :

- Applying the rule would make the code **less readable**.
- To be consistent with surrounding code that already breaks the rule (a chance to clean up, but not required).
- The code predates the guideline and there is no other reason to modify it.
- To maintain compatibility with older Python versions.
- **Never** break backwards compatibility just to comply with PEP 8.

Code Lay-out

Indentation
Use 4 spaces per indentation level. Do not mix tabs and spaces.

```
def my_function(x):
    if x > 0:
        print("Positive")
```

For continuation lines, align wrapped elements either vertically or using a hanging indent. A hanging indent should be further indented.

```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Hanging indent (more indentation)
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

```
# Arguments on first line and hanging indent
# are not allowed
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

Maximum Line Length
Limit all lines to a maximum of **79 characters**. For docstrings and comments, limit lines to **72 characters**. Use implicit line continuation inside parentheses, brackets, and braces. Use explicit line continuation (\) if not applicable.

```
with open('/path/to/file/read') as f_1, \
     open('/path/to/file/write', 'w') as f_2:
    f_2.write(f_1.read())
```

Binary Operators
Break *before* binary operators for better readability.

```
# Recommended style
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Blank Lines

- **Two blank lines** around top-level function and class definitions.
- **One blank line** around method definitions inside a class.
- Use blank lines sparingly inside functions to indicate logical sections.

Source File Encoding
Always use UTF-8 for Python code.
Imports
Imports should be on separate lines and grouped in the following order, separated by a blank line :

1. Standard library imports.
2. Related third-party imports.
3. Local application/library specific imports.

Absolute imports are recommended.

```
import os
import sys

from third_party import lib1
from third_party import lib2

from my_prjct import my_module
from my_prjct.sub_pckge import another_module
```

```
import sys, os # Don't put on one line
```

Whitespace in Expressions and Statements

Avoid extraneous whitespace. The goal is to improve readability by creating a consistent visual grammar for parsing code.

Avoid in these situations:

Immediately inside parentheses, brackets or braces.

```
spam(ham, {eggs: 2})
```

```
spam( ham[ 1 ], { eggs: 2 } )
```

Immediately before a comma, semicolon, or colon.

```
if x == 4: print(x, y); x, y = y, x
```

```
if x == 4 : print(x , y) ; x , y = y , x
```

Immediately before the open parenthesis of a function call or indexing.

```
spam(1)
my_dict['key'] = my_list[index]
```

```
spam (1)
my_dict ['key'] = my_list [index]
```

More than one space around an assignment operator to align it with another.

```
x = 1
y = 2
long_variable = 3
```

```
x             = 1
y             = 2
long_variable = 3
```

Always use around these operators:

Use a single space around these binary operators:

- Assignment: `=`, `+=`, `-=`, etc.
- Comparisons: `==`, `<`, `>`, `!=`, `<=`, `>=`, `in`, `not in`, `is`, `is not`.
- Booleans: `and`, `or`, `not`.

```
i = i + 1
submitted += 1
x = x * 2 - 1
c = (a + b) * (a - b)
```

Keyword Arguments & Function Annotations

Do NOT use spaces around `=` for keyword arguments or default parameter values.

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

Exception: When combining a type annotation with a default value, DO use spaces around the `=`.

```
def munge(sep: str = None):...
def munge() -> int:...
```

```
def munge(input: str=None):...
def munge()->int:...
```

Trailing Commas

Trailing commas are often useful. When a list, tuple, or dict definition is expected to be extended over time, place each element on its own line and add a trailing comma. This makes version control diffs cleaner as adding a new item doesn't modify the previous line.

```
files = [
    'setup.cfg',
    'tox.ini',
]
```

They are mandatory for single-element tuples.

```
my_tuple = ('hello',)
```

```
my_tuple = ('hello') # This is a string
```

Programming Recommendations

Comparisons to Singletons

Comparisons to singletons like `None` should always be done with `is` or `is not`, never with equality operators.

```
if my_var is not None:
    #...
```

```
if my_var!= None: # Prone to error with
↪  objects
    #...
```

Boolean Value Checks

Don't compare boolean values to `True` or `False` using `==`.

```
if greeting:
    #...
```

```
if greeting == True:
    #...
```

Naming Conventions

Naming conventions provide implicit documentation, allowing developers to infer the type and purpose of an identifier from its name alone.

Special Underscore Forms:

- `_single_leading_underscore`: Weak "internal use" indicator. E.g., `from M import *` does not import objects whose names start with an underscore.
- `single_trailing_underscore_`: Used to avoid conflicts with keywords, e.g., `class_='primary'`.
- `__double_leading_underscore`: Invokes name mangling when used in a class context (e.g., in class `Foo`, `__bar` becomes `_Foo__bar`).
- `__double_leading_and_trailing__`: "Magic" objects or attributes that live in user-controlled namespaces. Never invent such names, only use them as documented (e.g., `__init__`, `__str__`).

Names to Avoid:

Never use the characters 'l' (lowercase L), 'O' (uppercase o), or 'I' (uppercase i) as single-character variable names, as they are indistinguishable from '1' and '0' in some fonts.

Type	Convention	Example
Module	`lcwu`	`my_module.py`
Package	`short_lowercase`	`mypackage`
Class	`CapWords` (CamelCase)	`class MyClass:`
Exception	`CapWords`	`class MyError(Exception):`
Type Variable	`CapWords`	`AnyStr = TypeVar('AnyStr', str, bytes)`
Function	`lcwu`	`def my_function():`
Variable	`lcwu`	`my_variable = 5`
Method	`lcwu`	`def calcul_total(self):`
Instance Var	`lcwu`	`self.my_inst_var`
Method Arg	`self` for instance, `cls` for class	`def method(self, arg1):`
Constant	`UCWU`	`MAX_OVERFLOW = 100`

lcwu = lower_case_with_underscores
UCWU = UPPER_CASE_WITH_UNDERSCORES

Comments and Docstrings

Block Comments

Indented to the same level as the code they describe. Each line starts with a `##` followed by a single space.

```
# This block comment applies to the
# following loop, explaining its purpose.
for i in range(10):
  ...
```

Inline Comments

Use sparingly. Separate from the statement by at least two spaces.

```
x = x + 1  # Compensate for border
```

```
x = x + 1 # This is an obvious comment
\end{tcblower}
```

Documentation Strings (Docstrings)

Write docstrings for all public modules, functions, classes, and methods (see PEP 257). The closing """ should be on a line by itself for multi-line docstrings.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
  ...
```

Sources

Official PEP8 Documentation:
https://peps.python.org/pep-0008/

Date: 25 October 2025