Python Style Guide—2026 edition

Martin Henz

National University of Singapore
School of Computing

November 13, 2026

This is the style guide for the Python sublanguages (§1–4) used in the book *Structure and Interpretation of Computer Programs, Python Edition* (SICPy). It follows PEP 8, the style that most Python programmers use, so what you practice here carries over to your future courses and to shared codebases, and it adds what SICPy needs. In the beginning of a first-year course, instructors may just *encourage* adherence to the style, and then gradually enforce adherence in order to improve communication between learners and staff.

The point of a style is that **a reader sees the structure of your program at a glance**. In Python, the structure is also what the interpreter reads: indentation and line breaks are part of the syntax. Many of the errors that beginners take for logic errors in their programs are formatting errors. Follow this guide, and you avoid them.

Indentation

The body of a function, and of an **`if`**, **`elif`**, **`else`**, **`while`** or **`for`** statement, is indented by **four spaces** more than the line that ends with the colon. Use spaces, never tabs, and never a mixture of the two. (In the Source Academy editor, the Tab key inserts spaces.)

```
# good style
def make_adder(x):
    def the_adder(y):
        if x >= 0:
            return x + y
        else:
            return -x + y
    return the_adder

# bad style: indentation of two spaces (legal, but hard to read)
def make_adder(x):
  def the_adder(y):
    return x + y
  return the_adder

# illegal Python: inconsistent indentation
def make_adder(x):
    def the_adder(y):
        if x >= 0:
            return x + y
          else:
            return -x + y
    return the_adder
```

Python reports an `IndentationError` when the indentation of a line does not match any line above it. The cure is to indent every block by the same amount, as in the good example.

Line length

A line should have **at most 79 characters**. In the playground, the vertical line in the editor marks the limit: no line should cross it. Long lines are hard to read, and hard to compare when you work side by side with others.

How to break a line

Python ends a statement at the end of the line. A line break is allowed in two places: **inside parentheses `( )`, brackets `[ ]` or braces `{ }`**, and after a **backslash** at the end of the line. Anywhere else, the line break ends the statement, and Python reports an error such as "Unexpected token" at the end of the line.

```
# illegal Python: the statement ends after the +
total = first_amount +
        second_amount

# good style: the break is inside parentheses
total = (first_amount +
         second_amount)

# legal, but avoid: a backslash at the end of the line
total = first_amount + \
        second_amount
```

Prefer parentheses to the backslash. The backslash looks out of place, and it is fragile: nothing may follow it, not even a space or a comment, or the line is an error that is hard to see. So when a long expression must be broken, put it in parentheses. This is why you will see **`return`** `(` followed by an expression on several lines: the parentheses are what allows the line breaks (see *Conditional expressions* below). Do not put parentheses around short expressions: write **`return`** `x + 1`, not **`return`** `(x + 1)`.

Break lines after commas in arguments and parameters, and before **`if`** and **`else`** in conditional expressions. When you break a long expression before or after a binary operator, do it the same way throughout your program.

Spaces

Put a single space on both sides of a binary operator, and after each comma.

```
# good style
x = 1 + 1
y = (x + 2) * 3
message = "zero" if x == 0 else "not zero"
my_function(1, 2, 3)

# bad style
x=1+1
y=(x+2)*3
message = "zero" if x==0 else "not zero"
my_function(1,2,3)
```

Do not put a space between a unary minus and its operand: write `-x`, not `- x`. Do not put spaces *inside* parentheses, brackets and braces, or *before* the parentheses of a function application or a function definition.

```
# good style
def my_function(arg1, arg2, arg3):
    return [arg1, arg2, arg3]
```

```
result = my_function(1, 2, 3)
negative_x = -x

# bad style
def my_function (arg1, arg2, arg3):
    return [ arg1, arg2, arg3 ]

result = my_function ( 1, 2, 3 )
negative_x = - x
```

Trailing whitespace. There should be no whitespace (spaces or tabs) before a newline: you cannot see it, and it causes problems in tools that handle program text, such as version control systems. Clean up all such *trailing whitespace* before you submit your program.

Blank lines

Separate the top-level function definitions of your program by two blank lines. Inside a function, use a single blank line, sparingly, to separate groups of statements.

```
# good style
def square(x):
    return x * x


def cube(x):
    return x * square(x)


result = cube(3)
```

Function definitions

In **def** `name(parameters):`, there is no space before the opening parenthesis and none before the colon. The body starts on the next line, indented by four spaces.

Conditional statements

Write **if**, **elif** and **else** each at the start of a line, followed by an indented block on the next lines. Do not put the body after the colon, even if it consists of one statement.

```
# good style
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0

# bad style: body on the same line as the condition
def sign(x):
    if x > 0: return 1
    elif x < 0: return -1
    else: return 0
```

Do not put parentheses around the condition: write `if x > 0:`, not `if (x > 0):`.

Conditional expressions

A conditional expression is meant for a short choice between two values. If it fits in the line, keep it there.

```
# good style
def absolute(x):
    return x if x >= 0 else -x

aspect_ratio = 4 / 3 if landscape else 3 / 4
```

Long conditional expressions are not natural in Python. If the parts are lengthy, or you need more than one `else`, use a conditional statement instead, with `return` in each branch.

```
# good style
def ackermann(x, y):
    if y == 0:
        return 0
    elif x == 0:
        return 2 * y
    elif y == 1:
        return 2
    else:
        return ackermann(x - 1, ackermann(x, y - 1))
```

Use a conditional expression that does not fit in the line only if you need an expression, for example in the body of a lambda expression, or as an argument. Then put it in parentheses, and put the parts that start with `if` or `else` on lines of their own, aligned with the first part. Do not use more lines than needed.

```
# acceptable: an expression is needed here
count = lambda jacket: (
    count_buttons(jacket) if is_checkered(jacket)
    else 1
)

# debatable: a statement may be more natural
def ackermann(x, y):
    return (
        0 if y == 0
        else 2 * y if x == 0
        else 2 if y == 1
        else ackermann(x - 1, ackermann(x, y - 1))
    )

# bad style: line too long
def ackermann(x, y):
    return 0 if y == 0 else 2 * y if x == 0 else 2 if y == 1 else ackermann(x - 1, acker

# bad style: too much indentation
count = lambda jacket: (
    count_buttons(jacket) if is_checkered(jacket)
        else 1
)

# illegal Python: the line break is not inside parentheses
count = lambda jacket: count_buttons(jacket) if is_checkered(jacket)
    else 1
```

Arguments and parameters

The arguments of function applications, and the parameters of function definitions and lambda expressions, are written in one line, if they fit. If they do not fit, they are aligned vertically: either under the first argument (after the opening parenthesis), or indented by four spaces on the lines after the opening parenthesis, which is then the last character of its line.

```
# good style: aligned under the first argument
def execute_application(fun, args, succeed, fail):
    return succeed(apply_primitive_function(fun, args),
                   fail)

# good style: indented by four spaces
def make_execution_function(
    inst, labels, machine, pc, flag, stack, ops
):
    return assemble_instruction(
        instruction_source(inst),
        labels,
        machine,
        pc,
    )

# bad style: the arguments are neither aligned nor indented
result = reduce(add, 0, map(square,
    enumerate_interval(1,
        10)))
```

When the arguments are on lines of their own, the closing parenthesis goes on a line of its own, aligned with the start of the line that has the opening parenthesis (as above), or at the end of the last argument.

Lambda expressions

Keep the parameters and the body expression of a lambda expression in one line, if they are short enough. If they are lengthy, put the lambda expression in parentheses, break the line after the colon, and indent the body by four spaces.

```
# good style
def count_buttons(garment):
    return reduce(
        lambda sleaves, total: sleaves + total,
        0,
        map(
            lambda jacket: (
                count_buttons(jacket) if is_checkered(jacket)
                else 1
            ),
            garment
        )
    )
```

Import directives

If the imported names do not fit in one line, put them in parentheses, indent them by four spaces, and put the closing parenthesis on a line of its own.

```
# good style
from rune import (
    beside, heart,
    show
)
show(beside(heart, heart))
```

Names

When naming constants or variables, choose names that are meaningful to your reader and that make sense in the application domain. Names contain only lower-case letters of the roman alphabet, digits, and underscores. Use underscores to separate words (*snake case*). Avoid the single letters `l` (lower-case L), `O` and `I`, which look like digits.

```
# good names
my_variable
x_1
draw_connected_proportional

# bad names
myVariable
Draw_Connected_Proportional
l
```

Nested scope

Do not use the same name in a nested scope. Python does not warn you, and you can easily change the meaning of a name by accident.

```
# bad program
x = 1


def f(x):
    # here, the name x of the global frame is
    # "shadowed" by the parameter x
    return x + 1
```

Comments

A comment starts with `#` and a space. Write a comment on a line of its own, above the code that it explains and at the same indentation, and say *why*, not *what*.

```
# good style
# The accumulator is the sum of the elements seen so far.
def sum_iter(xs, acc):
    return acc if is_null(xs) else sum_iter(tail(xs), acc + head(xs))
```

If your program does not run

Many error messages of the Python interpreter come from formatting. Check, in this order:

Message or symptom	**Likely cause**
"Unexpected token" at the end of a line	A line break outside parentheses: put the expression in parentheses, or keep it on one line. (A space after a backslash at the end of a line is an error, too, with the message "invalid syntax".)
`IndentationError`	Inconsistent indentation: indent every block by four spaces, and do not mix tabs and spaces.
"Unexpected token: indent"	A line is indented that should not be: line it up with the previous statement.
A missing colon	Every `def`, `if`, `elif`, `else`, `while` and `for` line ends with a colon.
A block that seems to end too early or too late	The indentation of its lines differs: align them under each other.

A program that follows this guide can still be wrong, of course. But if it is, you can be sure that the reason is the logic and not the layout.