Python syntax and best practices

Introduction

Python has a list of best practices that should be used to create professional and easy-to-read program code. This document describes the main practices. The Python community have listed all these practices in the PEP8 style guide for Python code. This document describes the main best practices you should adhere to.

Code layout

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. And the indentation is required and they have an impact on how the Python code runs.

Use 4 spaces per indentation level. Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent, the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Here's an example of how your code should look:

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

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Do not use the following syntax with your code

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Naming rules

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand.

Keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable message_1 but not 1_message. You should name functions and variables in lowercase with the words separated by underscores, as this will improve readability.
- Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, greeting_message works, but greeting message will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print.
- Variable names should be short but descriptive. For example, name is better than n, student_name is better than s_n, and name_length is better than length_of_persons_name.
- Be careful when using the lowercase letter l and the uppercase letter O because they could be confused with the numbers 1 and 0.

Here's some examples using these variable naming conventions

```
names = "Python"                    # variable name
job_title = "Software Engineer"     # variable name with underscore
populated_countries_list = []       # variable name with underscore
```

You should use one underscore (_) as a prefix for the internal variable of a class, where you don't want an outside class to access the variable. This is just a convention; Python doesn't make a variable with a single underscore prefix private.

It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated. As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

Here is a list summarising these conventions

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return

del	import	try
elif	in	while
else	is	with
except	lambda	yield

Spaces are the preferred indentation method.

Tabs should be used solely to remain consistent with code that is already indented with tabs.

Python 3 disallows mixing the use of tabs and spaces for indentation.

Whitespace in Expressions and Statements

Avoid extraneous whitespace in the following situations:

Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

Between a trailing comma and a following close parenthesis.

```
Yes: foo = (0,)
No:  bar = (0, )
```

Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted.

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```

Do not do this:

```
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dct['key'] = lst[index]
No:  dct ['key'] = lst [index]
```

More than one space around an assignment (or other) operator to align it with another.

Yes

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

No

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

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. The first word should be capitalised, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.

When writing English, follow Strunk and White.

Python coders from non-English speaking countries: please write your comments in English, unless you are sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

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

Other Recommendations

Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.

Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present. (See Function Annotations below for more about function annotations.)

Yes:

```
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...
```

No:

```
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter.

Yes:

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

No:

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

When combining an argument annotation with a default value, however, use spaces around the = sign:

Yes:

```
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

No:

```
def munge(input: AnyStr=None): ...
def munge(input: AnyStr, limit = 1000): ...
```

Compound statements (multiple statements on the same line) are generally discouraged.

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

When to Use Trailing Commas

Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses.

Yes:

```
FILES = ('setup.cfg',)
```

OK, but confusing:

```
FILES = 'setup.cfg',
```

When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples).

Yes:

```
FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
```

No:

```
FILES = ['setup.cfg', 'tox.ini',]
initialize(FILES, error=True,)
```