* Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

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

You should put a blank line between each group of imports.

* Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):
* import mypkg.sibling
* from mypkg import sibling

from mypkg.sibling import example

[Whitespace in Expressions and Statements](https://www.python.org/dev/peps/pep-0008/#id26)

[Pet Peeves](https://www.python.org/dev/peps/pep-0008/#id27)

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.

Yes:

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]

No:

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

[Other Recommendations](https://www.python.org/dev/peps/pep-0008/#id28)

* 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)

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

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)

* Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present. (See[Function Annotations](https://www.python.org/dev/peps/pep-0008/#function-annotations) below for more about function annotations.)

Yes:

def munge(input: AnyStr): ...

def munge() -> AnyStr: ...

No:

def munge(input:AnyStr): ...

def munge()->PosInt: ...

* When combining an argument annotation with a default value, use spaces around the = sign (but only for those arguments that have both an annotation and a default).

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](https://www.python.org/dev/peps/pep-0008/#id29)

Trailing commas are usually optional, except they are mandatory when making a tuple of one element (and in Python 2 they have semantics for the print statement). 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,)

# [Comments](https://www.python.org/dev/peps/pep-0008/#id30)

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 capitalized, 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 120% sure that the code will never be read by people who don't speak your language.

## [Block Comments](https://www.python.org/dev/peps/pep-0008/#id31)

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](https://www.python.org/dev/peps/pep-0008/#id32)

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

## [Documentation Strings](https://www.python.org/dev/peps/pep-0008/#id33)

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in [PEP 257](https://www.python.org/dev/peps/pep-0257).

* Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
* [PEP 257](https://www.python.org/dev/peps/pep-0257) describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, e.g.:
* """Return a foobang
* Optional plotz says to frobnicate the bizbaz first.
* """
* For one liner docstrings, please keep the closing """ on the same line.

# [Naming Conventions](https://www.python.org/dev/peps/pep-0008/#id34)

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

## [Overriding Principle](https://www.python.org/dev/peps/pep-0008/#id35)

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

## [Descriptive: Naming Styles](https://www.python.org/dev/peps/pep-0008/#id36)

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

* b (single lowercase letter)
* B (single uppercase letter)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (or CapWords, or CamelCase -- so named because of the bumpy look of its letters [[4]](https://www.python.org/dev/peps/pep-0008/#id11)). This is also sometimes known as StudlyCaps.

Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.

* mixedCase (differs from CapitalizedWords by initial lowercase character!)
* Capitalized\_Words\_With\_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st\_mode, st\_size, st\_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

* \_single\_leading\_underscore: weak "internal use" indicator. E.g. from M import \* does not import objects whose name starts with an underscore.
* single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g.
* Tkinter.Toplevel(master, class\_='ClassName')
* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes\_FooBar\_\_boo; see below).
* \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

## [Prescriptive: Naming Conventions](https://www.python.org/dev/peps/pep-0008/#id37)

### [Names to Avoid](https://www.python.org/dev/peps/pep-0008/#id38)

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

* Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable).

Yes:

def foo(x):

if x >= 0:

return math.sqrt(x)

else:

return None

def bar(x):

if x < 0:

return None

return math.sqrt(x)

No:

def foo(x):

if x >= 0:

return math.sqrt(x)

def bar(x):

if x < 0:

return

return math.sqrt(x)

* Use string methods instead of the string module.

String methods are always much faster and share the same API with unicode strings. Override this rule if backwards compatibility with Pythons older than 2.0 is required.

* Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone. For example:

Yes: if foo.startswith('bar'):

No: if foo[:3] == 'bar':

* Object type comparisons should always use isinstance() instead of comparing types directly.
* Yes: if isinstance(obj, int):
* No: if type(obj) is type(1):

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2, str and unicode have a common base class, basestring, so you can do:

if isinstance(obj, basestring):

Note that in Python 3, unicode and basestring no longer exist (there is only str) and a bytes object is no longer a kind of string (it is a sequence of integers instead)

* For sequences, (strings, lists, tuples), use the fact that empty sequences are false.
* Yes: if not seq:
* if seq:
* No: if len(seq):
* if not len(seq):
* Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.
* Don't compare boolean values to True or False using ==.
* Yes: if greeting:
* No: if greeting == True:
* Worse: if greeting is True:

## [Function Annotations](https://www.python.org/dev/peps/pep-0008/#id52)

With the acceptance of [PEP 484](https://www.python.org/dev/peps/pep-0484), the style rules for function annotations are changing.

* In order to be forward compatible, function annotations in Python 3 code should preferably use [PEP 484](https://www.python.org/dev/peps/pep-0484) syntax. (There are some formatting recommendations for annotations in the previous section.)
* The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.
* However, outside the stdlib, experiments within the rules of [PEP 484](https://www.python.org/dev/peps/pep-0484) are now encouraged. For example, marking up a large third party library or application with [PEP 484](https://www.python.org/dev/peps/pep-0484) style type annotations, reviewing how easy it was to add those annotations, and observing whether their presence increases code understandability.
* The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.
* For code that wants to make a different use of function annotations it is recommended to put a comment of the form:
* # type: ignore

near the top of the file; this tells type checker to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in [PEP 484](https://www.python.org/dev/peps/pep-0484).)

* Like linters, type checkers are optional, separate tools. Python interpreters by default should not issue any messages due to type checking and should not alter their behavior based on annotations.
* Users who don't want to use type checkers are free to ignore them. However, it is expected that users of third party library packages may want to run type checkers over those packages. For this purpose [PEP 484](https://www.python.org/dev/peps/pep-0484) recommends the use of stub files: .pyi files that are read by the type checker in preference of the corresponding .py files. Stub files can be distributed with a library, or separately (with the library author's permission) through the typeshed repo [[5]](https://www.python.org/dev/peps/pep-0008/#id12).
* For code that needs to be backwards compatible, type annotations can be added in the form of comments. See the relevant section of [PEP 484](https://www.python.org/dev/peps/pep-0484) [[6]](https://www.python.org/dev/peps/pep-0008/#id13).

## [Variable annotations](https://www.python.org/dev/peps/pep-0008/#id53)

[PEP 526](https://www.python.org/dev/peps/pep-0526) introduced variable annotations. The style recommendations for them are similar to those on function annotations described above:

* Annotations for module level variables, class and instance variables, and local variables should have a single space after the colon.
* There should be no space before the colon.
* If an assignment has a right hand side, then the equality sign should have exactly one space on both sides.
* Yes:
* code: int
* class Point:
* coords: Tuple[int, int]
* label: str = '<unknown>'
* No:
* code:int # No space after colon
* code : int # Space before colon
* class Test:
* result: int=0 # No spaces around equality sign
* Although the [PEP 526](https://www.python.org/dev/peps/pep-0526) is accepted for Python 3.6, the variable annotation syntax is the preferred syntax for stub files on all versions of Python (see [PEP 484](https://www.python.org/dev/peps/pep-0484) for details).