Ankit Aggarwal

ankit1601@gmail.com

Summary

This document is focused on coding styles and recommendation described in pep8 for better scalability, modularity and coding styles. Mainly focusses to eliminated linting errors

PytHOn Coding styles and Recommendations based on PEP8

Code styles

Table of Contents

[1 Code Lay-out 3](#_Toc34070057)

[1.1 Use 4 spaces per indentation level 3](#_Toc34070058)

[1.2 Tabs or Spaces? 3](#_Toc34070059)

[1.3 Maximum Line Length 4](#_Toc34070060)

[1.4 Should a Line break Before or After a Binary Operator? 4](#_Toc34070061)

[1.5 Blank Lines 5](#_Toc34070062)

[1.6 Source File Encoding 5](#_Toc34070063)

[1.7 Imports 5](#_Toc34070064)

[1.8 Module Level Dunder Names 7](#_Toc34070065)

[2 String Quotes 7](#_Toc34070066)

[3 Whitespace in Expressions and Statements 8](#_Toc34070067)

[4 When to Use Trailing Commas 10](#_Toc34070068)

[5 Comments 10](#_Toc34070069)

[5.1 Block Comments 10](#_Toc34070070)

[5.2 Inline Comments 10](#_Toc34070071)

[5.3 Documentation Strings 10](#_Toc34070072)

[6 Naming Conventions 11](#_Toc34070073)

[6.1 Overriding Principle 11](#_Toc34070074)

[6.2 Descriptive: Naming Styles 11](#_Toc34070075)

[6.3 Prescriptive: Naming Conventions 12](#_Toc34070076)

[6.3.1 Names to Avoid 12](#_Toc34070077)

[6.3.2 ASCII Compatibility 12](#_Toc34070078)

[6.3.3 Package and Module Names 12](#_Toc34070079)

[6.3.4 Class Names 12](#_Toc34070080)

[6.3.5 Type Variable Names 12](#_Toc34070081)

[6.3.6 Exception Names 12](#_Toc34070082)

[6.3.7 Global Variable Names 12](#_Toc34070083)

[6.3.8 Function and Variable Names 12](#_Toc34070084)

[6.3.9 Function and Method Arguments 13](#_Toc34070085)

[6.3.10 Method Names and Instance Variables 13](#_Toc34070086)

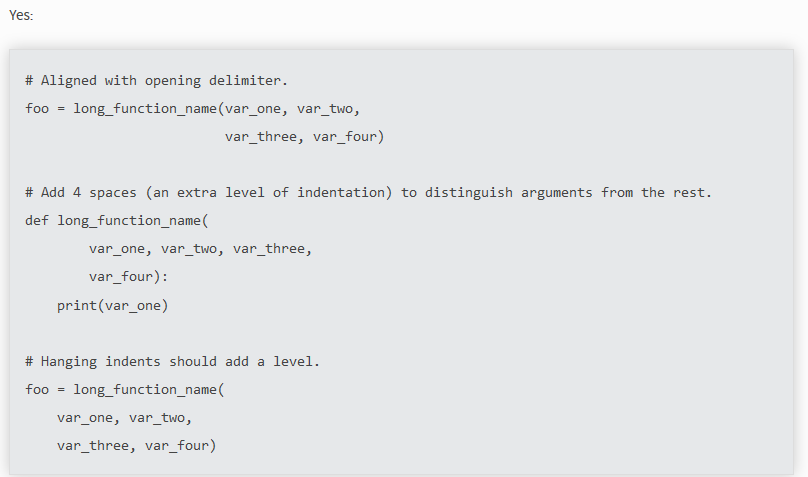
[6.3.11 Constants 13](#_Toc34070087)

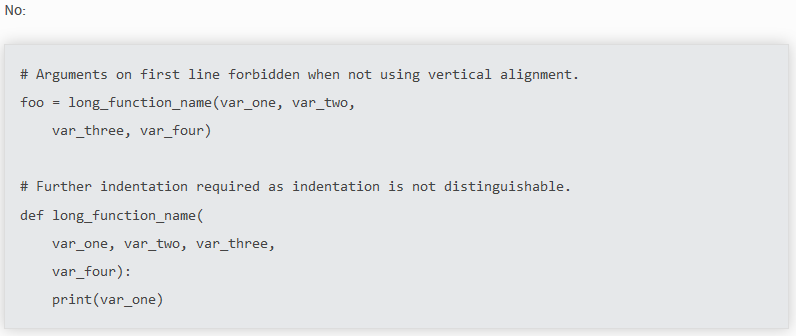
[7 Programming Recommendations 13](#_Toc34070088)

**Python PEP8 Recommendations:**

# Code Lay-out

## Use 4 spaces per indentation level



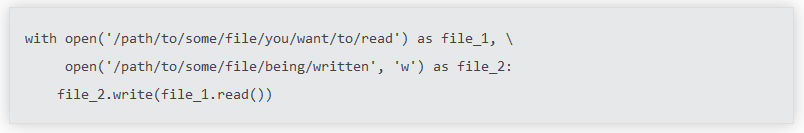


## Tabs or Spaces?

* 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.**
* Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.

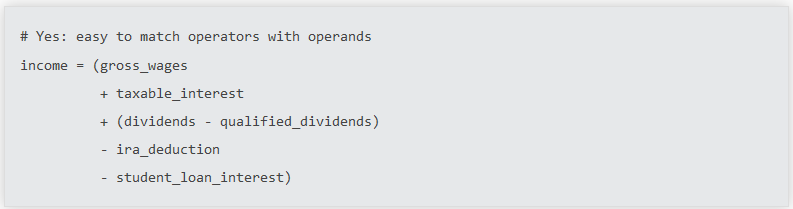
## Maximum Line Length

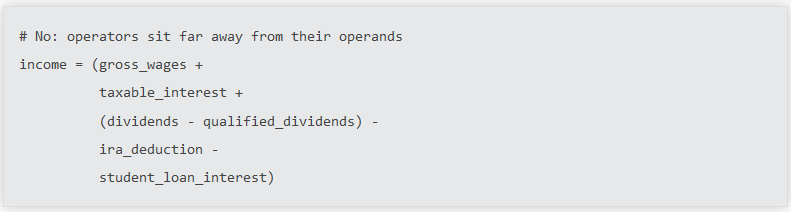
* Limit all lines to a maximum of 79 characters.
* For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
* Limiting the required editor window width makes it possible to have several files open side-by-side and works well when using code review tools that present the two versions in adjacent columns.



## Should a Line break Before or After a Binary Operator?

* For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye must do extra work to tell which items are added and which are subtracted:





## Blank Lines

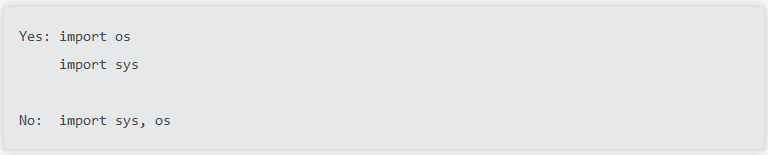
* **Surround top-level function and class definitions with two blank lines**
* Method definitions inside a class are surrounded by a single blank line.
* Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).
* Use blank lines in functions, sparingly, to indicate logical sections (IF ELSE etc.)

## Source File Encoding

* Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).
* Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration.
* In the standard library, non-default encodings should be used only for test purposes or when a comment or docstring needs to mention an author name that contains non-ASCII characters; otherwise, using \x, \u, \U, or \N escapes is the preferred way to include non-ASCII data in string literals.
* All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible
* string literals and comments must also be in ASCII.
* The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors

## Imports

* Imports should usually be on separate lines:



from subprocess import Popen, PIPE (this is acceptable)

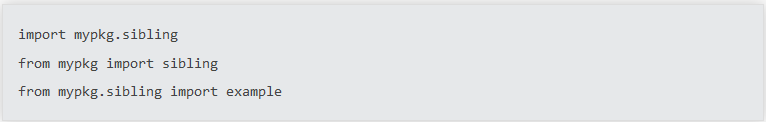
* Imports are always put at the top of the file, just after any module comments and docstrings, and before module global and constants

Imports should be grouped in the following order:

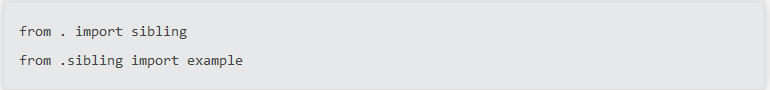
1. Standard library imports.
2. Related third party imports.
3. Local application/library specific imports.

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

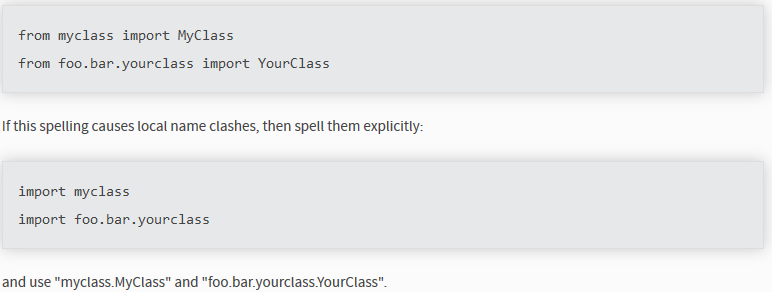
* Absolute imports are recommended, as they are usually more readable and tend to be better behaved. It helps us better identify code.

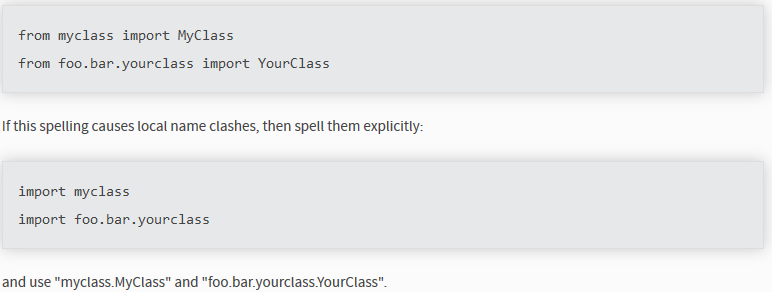


* However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:



* When importing a class from a class-containing module, it's usually okay to spell this





* Wildcard imports (from <module> import \*) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools

## Module Level Dunder Names

* Module level "dunders" (i.e. names with two leading and two trailing underscores) such as \_\_all\_\_, \_\_author\_\_, \_\_version\_\_, etc. should be placed after the module docstring but before any import statements except from \_\_future\_\_ imports. Python mandates that future-imports must appear in the module before any other code except docstrings:

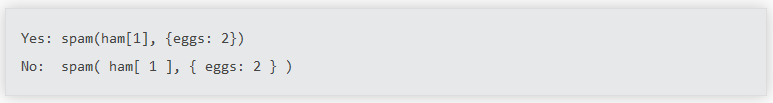


# String Quotes

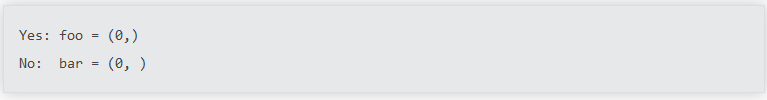
* In Python, single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.
* **For triple-quoted strings, always use double quote characters to be consistent with the docstring convention in PEP 257**.

# Whitespace in Expressions and Statements

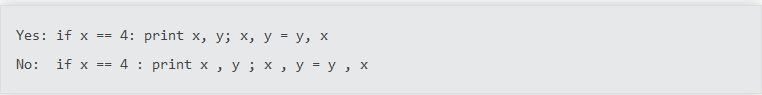
* Avoid extraneous whitespace in the following situations:



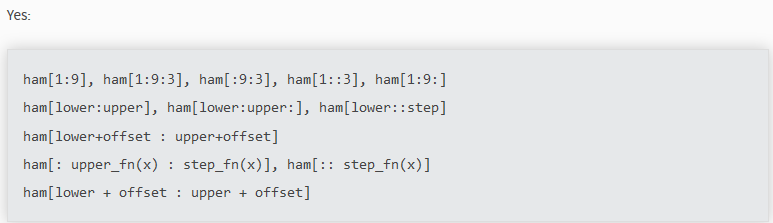
* Between a trailing comma and a following close parenthesis

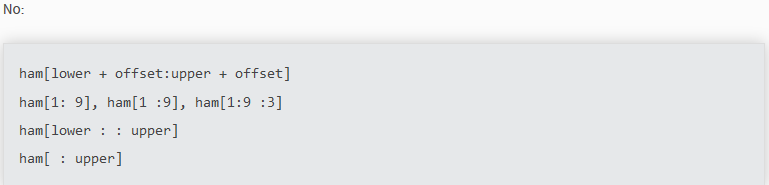


* Immediately before a comma, semicolon, or colon:

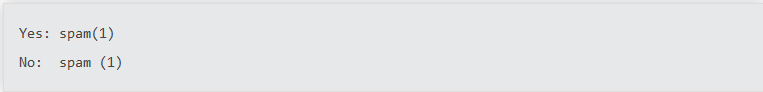


* However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is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

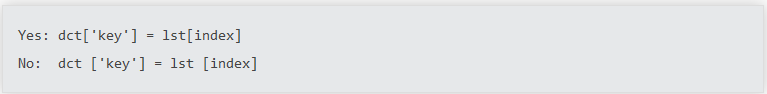




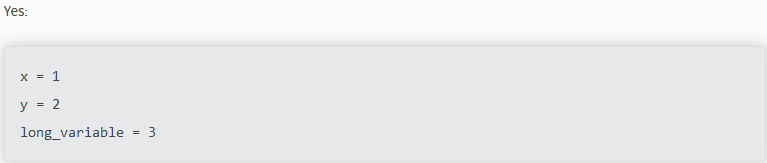
* Immediately before the open parenthesis that starts the argument list of a function call:



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

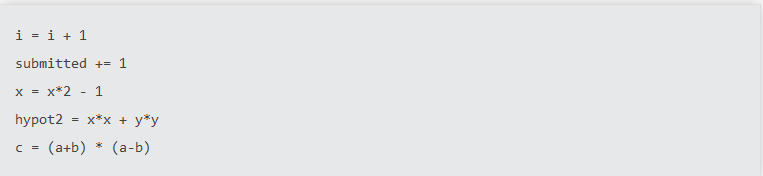


* More than one space around an assignment (or other) operator to align it with another.

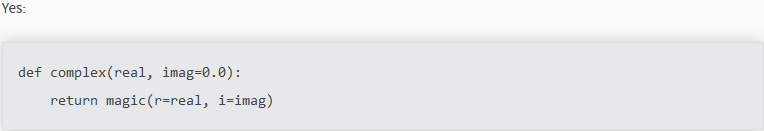




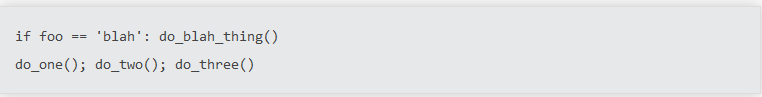
* **Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).**
* If operators with different priorities are used, consider adding whitespace around 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



* **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\**

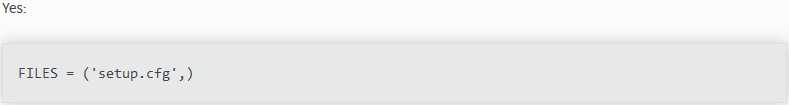


* Compound statements (multiple statements on the same line) are generally discouraged.



# When to Use Trailing Commas

* 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.



# 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 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

* 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

## Documentation Strings

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
* For one liner docstrings, please keep the closing """ on the same line.

# Naming Conventions

## Overriding Principle

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

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]). 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!)
* \_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 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

### Names to Avoid

* 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.

### ASCII Compatibility

* Identifiers used in the standard library must be ASCII compatible as described in the [policy section](https://www.python.org/dev/peps/pep-3131/#policy-specification) of [PEP 3131](https://www.python.org/dev/peps/pep-3131)

### Package and Module Names

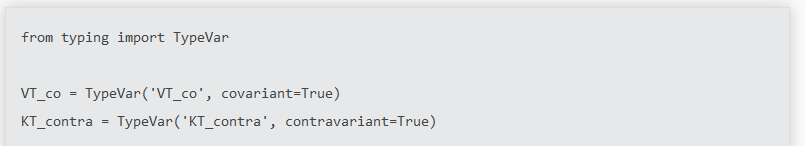
* Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
* Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

### Class Names

* Class names should normally use the **CapWords** convention

### Type Variable Names

* Names of type variables introduced in [PEP 484](https://www.python.org/dev/peps/pep-0484) should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes \_co or \_contra to the variables used to declare covariant or contravariant behavior correspondingly



### Exception Names

* Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error)

### Global Variable Names

* Let's hope that these variables are meant for use inside one module only
* The conventions are about the same as those for functions.
* Modules that are designed for use via from M import \* should use the \_\_all\_\_ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

### Function and Variable Names

* Function names should be lowercase, with words separated by underscores as necessary to improve readability.
* Variable names follow the same convention as function names.
* mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

### Function and Method Arguments

* Always use self for the first argument to instance methods.
* Always use cls for the first argument to class methods.
* If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss.

### Method Names and Instance Variables

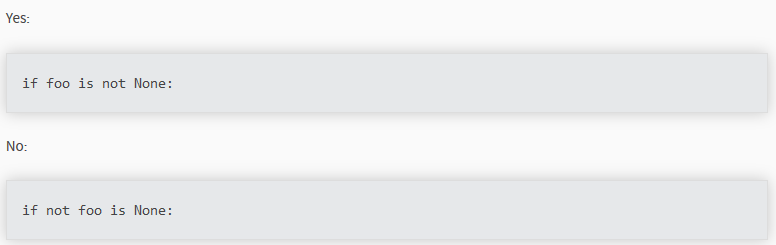
* Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.
* Use one leading underscore only for non-public methods and instance variables.
* To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

### Constants

* Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX\_OVERFLOW and TOTAL.

# Programming Recommendations

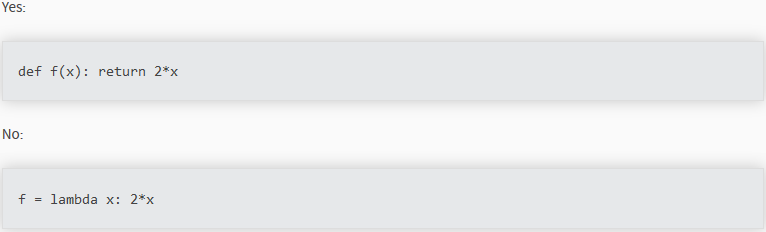
* Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such)
* Comparisons to singletons like None should always be done with is or is not, never the equality operators.
* Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable and preferred.



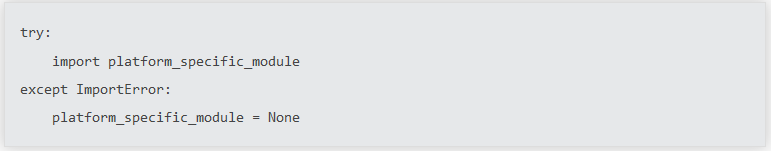
* When implementing ordering operations with rich comparisons, it is best to implement all six operations (\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_) rather than relying on other code to only exercise a particular comparison.
* To minimize the effort involved, the **functools.total\_ordering**() decorator provides a tool to generate missing comparison methods.

PEP 207 indicates that reflexivity rules are assumed by Python. Thus, the interpreter may swap y > x with x < y, y >= x with x <= y, and may swap the arguments of x == y and x != y. The sort() and min() operations are guaranteed to use the < operator and the max() function uses the > operator. However, it is best to implement all six operations so that confusion doesn't arise in other contexts

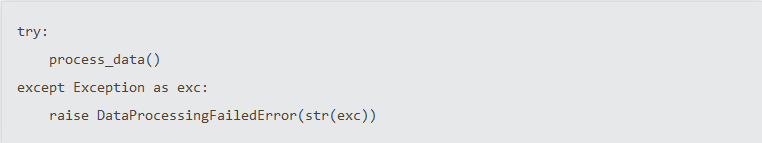
* Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier. This is more useful for tracebacks and string representations in general.



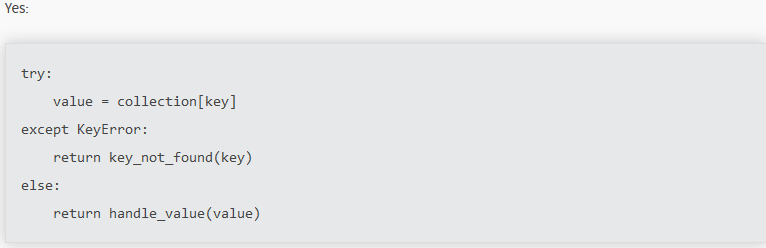
* Derive exceptions from **Exception** rather than **BaseException**. Direct inheritance from **BaseException** is reserved for exceptions where catching them is almost always the wrong thing to do. Aim to answer the question "What went wrong?" programmatically, rather than only stating that "A problem occurred". Class naming conventions apply here, although you should add the suffix "Error" to your exception classes
* Use exception chaining appropriately. In Python 3, "raise X from Y" should be used to indicate explicit replacement without losing the original traceback.
* When catching exceptions, mention specific exceptions whenever possible instead of using a bare except clause:

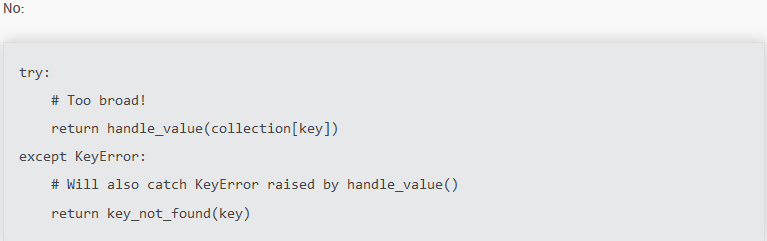


* When binding caught exceptions to a name, prefer the explicit name binding syntax



* When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of errno values.
* Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs.



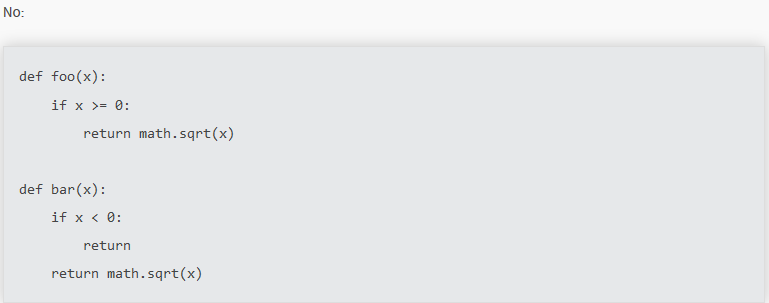


* When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.
* Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources.

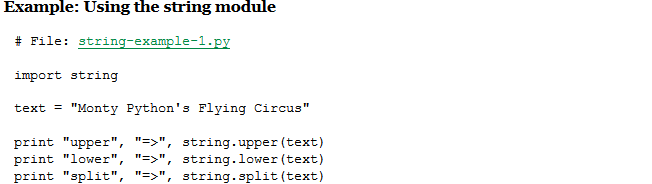


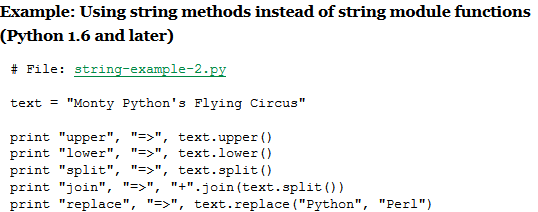
* Be consistent in **return statements**. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return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



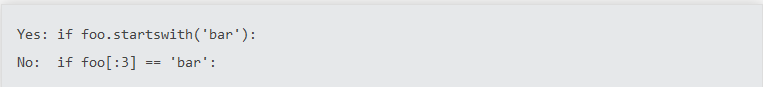


* Use string methods instead of the string module.

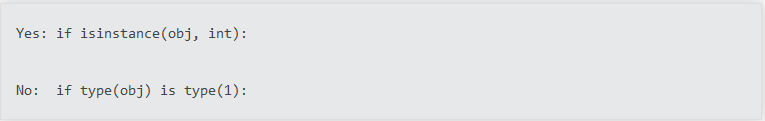




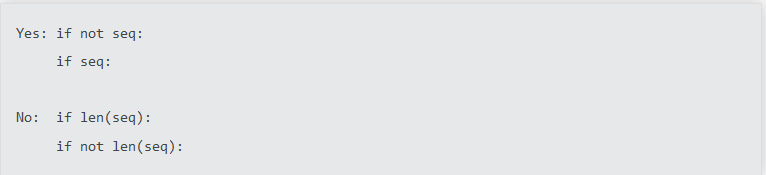
* Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone:



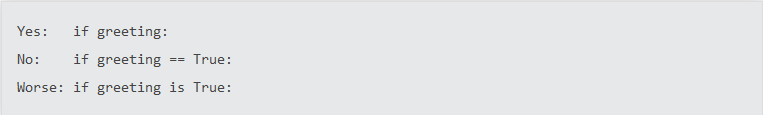
* Object type comparisons should always use isinstance() instead of comparing types directly.



* For sequences, (strings, lists, tuples), use the fact that empty sequences are false.



* 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 ==.



* Use of the flow control statements return/break/continue within the finally suite of a try...finally, where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite.

