**PEP8 Python coding standards**

<https://peps.python.org/pep-0008/#introduction>

Linting tools like pylint and flake8 can be used in VS code which are based on PEP8.

**Imports**

* Imports should usually be on separate lines
* Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
* 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)
* Imports should be grouped in the following order:
  + Standard library imports.
  + Related third party imports.
  + Local application/library specific imports.
* Add a space between different types of imports.
* Keep two blank lines surrounding classes and top-level functions. The methods inside of the class should be surrounded by a single blank line only.
* The preferred method of indentation is spaces, the 4 spaces indentation is accepted and accurate, but still, most people prefer tab indentation. Do not to mix both spaces and tabs for indentation.
* Avoid wildcard imports (from <module> import \*)

**Arguments Indentation:**

* Continuation lines should align wrapped elements either vertically using Python’s implicit line joining inside parentheses, brackets and braces, or using a hanging indent [1]. 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:

|  |
| --- |
| ***# Correct:***  *# 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**)** |

|  |
| --- |
| ***# Wrong:***  *# 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**)** |

* When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it’s worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces. This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement. Acceptable options in this situation include, but are not limited to:

|  |
| --- |
| *# No extra indentation.*  **if** **(**this\_is\_one\_thing **and**  that\_is\_another\_thing**):**  do\_something**()**  *# Add a comment, which will provide some distinction in editors*  *# supporting syntax highlighting.*  **if** **(**this\_is\_one\_thing **and**  that\_is\_another\_thing**):**  *# Since both conditions are true, we can frobnicate.*  do\_something**()**  *# Add some extra indentation on the conditional continuation line.*  **if** **(**this\_is\_one\_thing  **and** that\_is\_another\_thing**):**  do\_something**()** |

**Length of code line:**

* The length of the line should not be greater than 79 characters. In the case of docstrings and comments where a block of text is large, it is limited to 72 characters. For long multiple case statements, the backslashes are permissible.

|  |
| --- |
| def sample\_function(arg1, arg2):  '''  The document string length for a single line should be less than  72 characters. So that long texts should be adjusted in a single  window  '''    # code has maximum lengths of 79 characters, can use backslash  # to break the line  list\_of\_subjects = [  'Physics', 'Chemistry', 'Mathematics', 'Biology', ‘Bio’, \  ] |

* The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list. or it may be lined up under the first character of the line that starts the multiline construct, as in:

|  |
| --- |
| my\_list **=** **[**  **1,** **2,** **3,**  **4,** **5,** **6,**  **]**  my\_list **=** **[**  **1,** **2,** **3,**  **4,** **5,** **6,**  **]**  result **=** some\_function\_that\_takes\_arguments**(**  'a'**,** 'b'**,** 'c'**,**  'd'**,** 'e'**,** 'f'**,**  **)**  result **=** some\_function\_that\_takes\_arguments**(**  'a'**,** 'b'**,** 'c'**,**  'd'**,** 'e'**,** 'f'**,**  **)** |

**Whitespaces and blanklines:**

* Avoid extra white spaces
  + there must be a single white space around both sides of an operator,
  + one after the comma and
  + none inside opening or closing of parenthesis.
* 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.
* Both single quotes and double quotes are acceptable in python web development, you should use both if you need quotes inside quotes to avoid syntax error and extra backslash.

|  |
| --- |
| # Examples of commas and whitespaces    x, y = 30, "text inside quotes"  z = 'text inside quotes'  if x == 30: print(x, y, z)  # how to use quotes inside quotes  text = "This text is using 'the single quote' inside double quote"  print(text)  *# Correct:*  spam**(**ham**[1],** **{**eggs**:** **2})**  *# Wrong:*  spam**(** ham**[** **1** **],** **{** eggs**:** **2** **}** **)**  *# Correct:*  foo **=** **(0,)**  *# Wrong:*  bar **=** **(0,** **)**  Immediately before a comma, semicolon, or colon:  *# Correct:*  **if** x **==** **4:** print**(**x**,** y**);** x**,** y **=** y**,** x  *# Wrong:*  **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.

|  |
| --- |
| *# Correct:*  *#in slice the colon acts as a binary operator*  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**]**  *# Wrong:*  ham**[**lower **+** offset**:**upper **+** offset**]**  ham**[1:** **9],** ham**[1** **:9],** ham**[1:9** **:3]**  ham**[**lower **:** **:** upper**]**  ham**[** **:** upper**]** |
| *# Correct:*  spam**(1)**  *# Wrong:*  spam **(1)**  *# Correct:*  dct**[**'key'**]** **=** lst**[**index**]**  *# Wrong:*  dct **[**'key'**]** **=** lst **[**index**]**  *# Correct:*  x **=** **1**  y **=** **2**  long\_variable **=** **3**  *# Wrong:*  x **=** **1**  y **=** **2**  long\_variable **=** **3** |

* 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:

|  |
| --- |
| *# Correct:*  i **=** i **+** **1**  submitted **+=** **1**  x **=** x**\*2** **–** **1 #Different priorities of the operators**  hypot2 **=** x**\***x **+** y**\***y  c **=** **(**a**+**b**)** **\*** **(**a**-**b**) #Use your own judgement**  *# Wrong:*  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.

|  |
| --- |
| *# Correct:*  *#normal rules for colon in function annotations*  **def** munge**(**input**:** AnyStr**):** **...**  **def** munge**()** **->** PosInt**:** **...**  *# Wrong:*  **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:

|  |
| --- |
| *# Correct:*  **def** complex**(**real**,** imag**=0.0):**  **return** magic**(**r**=**real**,** i**=**imag**)**  *# Wrong:*  **def** complex**(**real**,** imag **=** **0.0):**  **return** magic**(**r **=** real**,** i **=** imag**)** |

**Using 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:

|  |
| --- |
| *# Correct:*  FILES **=** **(**'setup.cfg'**,)**  *# Wrong:*  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):

|  |
| --- |
| *# Correct:*  FILES **=** **[**  'setup.cfg'**,**  'tox.ini'**,**  **]**  initialize**(**FILES**,**  error**=True,**  **)**  *# Wrong:*  FILES **=** **[**'setup.cfg'**,** 'tox.ini'**,]**  initialize**(**FILES**,** error**=True,)** |

**Naming Conventions:**

* Use grammatically correct variable names, the class name should start with an uppercase and must follow camelCase convention If more than two words are to be used.
* In the same way, a function name should be joined with an underscore, and it must be lowercase.
* In method arguments, always use self as the first argument to declare an instance variable. In the same way, use ‘cls’ for the first argument for the class method.
* If the function name clashes with a reserved argument, use an underscore instead of a wrong spelling.
* Constants are declared in all capital letters.

|  |
| --- |
| # class name follows camelcase convention  class StudentDetails:    def \_\_init\_\_(self, first\_name, last\_name):  self.first\_name = first\_name  self.last\_name = last\_name    # Method name, variable names in lowercase joined with an underscore  def grade\_score(self, marks\_obtained):  # constants in capital  GRACE = 2  marks\_obtained = GRACE + marks\_obtained  if marks\_obtained > 90:  self.student\_grade = 'A'  elif marks\_obtained > 70:  student\_grade = 'B'  else:  student\_grade = 'C' |

**Use Exception Handling for every critical situation**

|  |
| --- |
| try:  file = open('filename.txt')  file.write('Hello World')    except Exception as e:  print('Cannot open the file :', e)    finally:  # Make sure to close the file afterwards  file.close() |

* 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.
* Design exception hierarchies based on the distinctions that code catching the exceptions is likely to need, rather than the locations where the exceptions are raised.
* Aim to answer the question “What went wrong?” programmatically, rather than only stating that “A problem occurred”
* When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause.

|  |
| --- |
| **try:**  **import** platform\_specific\_module  **except** **ImportError:**  platform\_specific\_module **=** **None** |

* Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs:

|  |
| --- |
| *# Correct:*  **try:**  value **=** collection**[**key**]**  **except** **KeyError:**  **return** key\_not\_found**(**key**)**  **else:**  **return** handle\_value**(**value**)**  *# Wrong:*  **try:**  *# Too broad!*  **return** handle\_value**(**collection**[**key**])**  **except** **KeyError:**  *# Will also catch KeyError raised by handle\_value()*  **return** key\_not\_found**(**key**)** |

**Writing functions:**

* Documenting every method with proper specification of parameters, return type, and data types. Try to avoid multiple returns from a function, a single generic return is preferred

|  |
| --- |
| # documenting a function  def get\_grades(marks):  """  Summary: getting grades from marks  Description: This function takes marks as an argument and returns grades  params:  marks(int) : marks obtained  grade(string) : grade achieved  """  if marks > 90:  grade = 'A'  elif marks > 70:  grade = 'B'  else:  grade = 'C'    return grade |

* Always use the DRY(Don’t repeate yourself) principle to reuse the code. The best way to do it is to use functions and classes. The common functions can be put into a separate utils.py file and can be used several times instead of creating similar functions again and again.

|  |
| --- |
| # function to read the file read  def file\_read(filename):  with open(filename, 'r') as f:  return f.read()  qualities = file\_read('quality.txt')  description = file\_read('description.txt')  summary = file\_read('summary.txt') |

**Should a Line Break Before or After a Binary Operator?**

|  |
| --- |
| *# Wrong:*  *# operators sit far away from their operands*  income **=** **(**gross\_wages **+**  taxable\_interest **+**  **(**dividends **-** qualified\_dividends**)** **-**  ira\_deduction **-**  student\_loan\_interest**)** |
| *# Correct:*  *# easy to match operators with operands*  income **=** **(**gross\_wages  **+** taxable\_interest  **+** **(**dividends **-** qualified\_dividends**)**  **-** ira\_deduction  **-** student\_loan\_interest**)** |

**Source File Encoding**

* Code in the core Python distribution should always use UTF-8.
* In the standard library, non-UTF-8 encodings should be used only for test purposes. Use non-ASCII characters sparingly, preferably only to denote places and human names. If using non-ASCII characters as data, avoid noisy Unicode characters like z̯̯͡a̧͎̺l̡͓̫g̹̲o̡̼̘ and byte order marks.
* All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren’t English).

**Comments:**

* Keep 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
* 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.
* Block comments : starts with a # and a single space (unless it is indented text inside the comment).
* Inline Comments: Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement.

|  |
| --- |
| x **=** x **+** **1** *# Compensate for border* |

* Documentation Strings: Conventions for writing good documentation strings (a.k.a. “docstrings”) are immortalized in PEP 257. 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.
  + 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.

|  |
| --- |
| *"""Return a foobang*  *Optional plotz says to frobnicate the bizbaz first.*  *"""*  *"""Return an ex-parrot."""* |

**Programming Recommendations**

* Use is not operator rather than not ... is.
* Comparisons to singletons like None should always be done with is or is not, never the equality operators.
* Also, beware of writing if x when you really mean if x is not None – e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context.

|  |
| --- |
| *# Correct:*  **if** foo **is** **not** **None:**  *# Wrong:*  **if** **not** foo **is** **None:** |

* Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier. The first form means that the name of the resulting function object is specifically ‘f’ instead of the generic ‘<lambda>’. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

|  |
| --- |
| *# Correct:*  **def** f**(**x**):** **return** **2\***x  *# Wrong:*  f **=** **lambda** x**:** **2\***x |

* 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:

|  |
| --- |
| *# Correct:*  **with** conn**.**begin\_transaction**():**  do\_stuff\_in\_transaction**(**conn**)**  *# Wrong:*  **with** conn**:**  do\_stuff\_in\_transaction**(**conn**)** |

* 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):

|  |
| --- |
| *# Correct:*  **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**)**  *# Wrong:*  **def** foo**(**x**):**  **if** x **>=** **0:**  **return** math**.**sqrt**(**x**)**  **def** bar**(**x**):**  **if** x **<** **0:**  **return**  **return** math**.**sqrt**(**x**)** |

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

|  |
| --- |
| *# Correct:*  **if** foo**.**startswith**(**'bar'**):**  *# Wrong:*  **if** foo**[:3]** **==** 'bar'**:** |

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

|  |
| --- |
| *# Correct:*  **if** isinstance**(**obj**,** int**):**  *# Wrong:*  **if** type**(**obj**)** **is** type**(1):** |

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

|  |
| --- |
| *# Correct:*  *# Empty sequences are false*  **if** **not** seq**:**  **if** seq**:**  *# Wrong:*  **if** len**(**seq**):**  **if** **not** len**(**seq**):** |

Don’t compare boolean values to True or False using ==:

|  |
| --- |
| *# Correct:*  **if** greeting**:**  *# Wrong:*  **if** greeting **==** **True:**  *# Worse Wrong:*  **if** greeting **is** **True:** |

* 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:

|  |
| --- |
| *# Wrong:*  **def** foo**():**  **try:**  **1** **/** **0**  **finally:**  **return** **42** |

**VARIABLE ANNOTATIONS**

* 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:

|  |
| --- |
| *# Correct:*  code**:** int  **class** Point**:**  coords**:** Tuple**[**int**,** int**]**  label**:** str **=** '<unknown>'  *# Wrong:*  code**:**int *# No space after colon*  code **:** int *# Space before colon*  **class** Test**:**  result**:** int**=0** *# No spaces around equality sign* |

**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.

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 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).
* \_\_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

* Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names.
* Identifiers used in the standard library must be ASCII compatible
* 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 should normally use the CapWords convention (Camel case).
* Function names should be lowercase, with words separated by underscores as necessary to improve readability. Variable names follow the same convention as function names.
* 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.
* 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.
* Python mangles these names with the class name: if class Foo has an attribute named \_\_a, it cannot be accessed by Foo.\_\_a. (An insistent user could still gain access by calling Foo.\_Foo\_\_a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.
* Constants are usually defined on a module level and written in all capital letters with underscores separating words.
* Use tuples when data is non-changeable, dictionaries when you need to map things, and lists if your data can change later on.
* Use the ‘with’ statement while opening a file, the ‘with’ statement closes the file even if there is an exception raised