Style Guide Pep-008

- Purpose
    - good code is read more than it is written, ensure **well commented and clearly readable**
    - ensure arguments are easily distinguishable and **visually grouped**
- General Formatting
    - use **4 spaces** (not ***\t***) to indent
    - on multiline lists/dicts/etc. add **ending bracket/paren on new line** that matches the indent of the first line for clarity
    - limit all lines to **max 79 characters**, or 72 char. for doctrings/comments
    - binary operator (+, -, /, etc.) in a sequence of operations should be **AFTER line break**, not before
    - **Surround with blank lines**:
        1. Top level function/class: 2
        2. Methods inside class: 1
    - Trailing comma ONLY when required to define a tuple with no second element
    - **Docstrings** (using """Docstring content here""" after def line): ALWAYS included for all modules, functions, classes, and methods. Even non-public objects should have documentation of their purpose.
- Whitespaces To Avoid:
    - Immediately **inside parentheses**, brackets or braces | # **Wrong**: ***spam(ham[1], { eggs: 2 })***
    - Between a **trailing comma** and a following close parenthesis | # **Wrong**: ***bar = (0,)***
    - Immediately **before a comma**, semicolon, or colon |
        1. # **Wrong**: ***if x == 4 : print x , y ; x , y = y , x***
    - Extra spaces **before/after operator** (**=**, **>**, **<**, **+**, **-**, etc.)
    - **Trailing** whitespace
    - around = sign when defining a **keyword argument** for function
    - **DO INCLUDE** at least 2 spaces **between inline comments and statement**:
        1. ***x = x + 1   # Increment x***
- Characters
    - encoding: **UTF-8**
    - **ascii characters only** for identifiers (variables, function/class names, etc.) and comments
        1. unless specifically testing for non-ascii, or author's name requires non-ascii
- Import Section
    - import different **modules on different lines**, and
        1. # Wrong: ***import sys, os***
    - always **import at top** of file
    - **Imports should be grouped** in the following order (blank line between each group):
        1. Standard library imports.
        2. Related third party imports.
        3. Local application/library specific imports.
    - **Avoid wildcard imports** (from <module> import *), make clear which packages are required
    - 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.

- Naming Conventions
  - Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.
  - Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
  - **Package**: short, all-lowercase names. Underscores are not ok.
  - **Module**: short, all-lowercase names. Underscores ok for readability.
  - **Class**: CapWords 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***. (Perhaps better is to avoid such clashes by using a synonym.)
  - Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.
    1. public no underscores
    2. '***cls***' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.
    3. If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores.
  - To better support introspection, modules should explicitly declare the names in their public API using the ***__all__*** attribute. Setting ***__all__*** to an empty list indicates that the module has no public API.
- Code writing
  - Do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form ***a += b*** or ***a = a + b***. This optimization is fragile even in CPython (it only works for some types) and isn't present at all in implementations that don't use refcounting.
  - 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***
  - 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.
  - Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.
    1. # Wrong: ***f = lambda x: 2*x***
    2. # Correct: ***def f(x): return 2*x***
  - When catching exceptions, mention specific exceptions whenever possible instead of using a bare ***except:*** clause. Unless:
    1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
    2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with raise. try...finally can be a better way to handle this case.
  - For all try/except clauses, **limit the try clause to the absolute minimum** amount of code necessary. Again, this avoids masking bugs.
  - Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources:
    1. # Correct:

       with conn.begin_transaction():

        do_stuff_in_transaction(conn)

    ii. # Wrong:

        with conn:
        do_stuff_in_transaction(conn)

    iii. The latter example doesn't provide any information to indicate that the ***__enter__*** and ***__exit__*** methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

- **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 ***''.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
- 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):***
- Compare True/False using the variable itself, not **==** or ***is***. Just ***if greeting:***
- For sequences, (strings, lists, tuples), use the fact that empty sequences are false:
    i. # Correct: ***if not seq***: / ***if seq:***
- Within ***try***/***except***/***finally***, DO NOT use ***return***/***break***/***continue*** statement inside ***finally***:
    i. will implicitly cancel any active exception that is propagating through the ***try*** statement

- Types
    - Use type annotations wherever possible

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

    - When used in a type hint, the expression None is considered equivalent to type(None).
    - PEP 526 introduced variable annotations. The style recommendations for them are similar to those on function annotations described above

```
code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
```