Red Hat's Coding Standards

Red Hat has told us to use general standards such as PEP8; so PEP8 will be used for this project. Please read below for detail.

General Python Coding Standards

Most developers generally use **PEP8** to format their Python code - **PEP8** is Python's style guide as well as a program that you can run against your code to check whether its formatted to Python's coding standards. To use **PEP8** run: **`pep8 file_name.py`**, it will output nothing if your code has no issues and is correctly formatted to PEP's conventions. And otherwise it will print a list of issues in layout, naming conventions, comments and so on. Install this using **`sudo yum install pep8`**. See below for some rules:

- Indentation:
  - 4 spaces per indentation level
  - Use spaces not tabs -- do not use both
- Line Length
  - Limit all lines to 79 characters
    - For docstrings or comments limit lines to 72 characters
  - For wrapping lines use Python's line continuation via parentheses, brackets and braces -- this is preferred to the backslash method
    - ie. put brackets around an expression and it continue over multiple lines
- Wrapping lines:
  - Break *after* binary operator eg.
    - ```
      if (bla or
          bla2)
      ```
      as opposed to
      ```
      if (bla
          or bla2)
      ```
  - The continued line should line up after the open bracket on previous line (as above) OR there should be no arguments on the first line and the other lines should be indented to distinguish from the other lines (as below)
    - ```
      # More indentation included to distinguish this from
      # the rest.
      def long_function_name(
              var_one, var_two, var_three,
              var_four):
          print(var_one)
      ```
  - The closing parenthesis on multi-line constructs can either line up with the first non-whitespace character of the last line (as eg1) OR lined up with the first character of the first line (as eg2)
    - eg1:
      ```
      my_list = [
      ```

```
    1, 2, 3,
    4, 5, 6,
    ]
```

- eg2:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
```

- Blank Lines
  - Class and function definitions should be separated by two blank lines
  - Method definitions in a class should be separated by one blank line
- Imports
  - Imports go at the top of the file
  - The order of imports is:
    - standard library
    - related third party
    - local application/library specific
  - Wildcard imports should be avoided (`import *`)
  - Imports should be on separate lines, with the exception of using the *`from`* keyword where either way is acceptable
  - Put a blank line between groups of imports
  - Put any `__all__` specification after imports
  - Absolute imports are recommended, however explicit relative imports are acceptable
    - absolute import:
      - `import mypkg.sibling`
    - explicit relative import:
      - `from . import sibling`
    - implicit relative import (don't do this):
      - `import bla`
      - `from bla import *`
- Whitespace
  - binary operators should have whitespace on either side
  - for operators that have different priority add whitespace around the lowest priority. But never have more than one space, and always have equal whitespace on both sides of the operator eg.
    - `hypot2 = x*x + y*y`
  - No whitespace:
    - inside parentheses/brackets/braces
    - before comma, semicolon, colon
    - before the open bracket that starts the argument list of a function call
    - before the open brace that starts an indexing or slicing
    - for aligning assignment ie. **don't** do this:

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

    - Don't use spaces around the = sign used to indicate a keyword argument or default parameter value eg.
        - `def complex(real, imag=0.0)`
- Multiple statements on the same line are generally discouraged
    - With some exception to loops or conditionals
        - Don't do this for multi-clause statements or long lines
- Comments
    - Complete sentences - first word capitalised unless its a variables/function
        - Usually use full stops, unless its a short comment
    - Two spaces after each sentence-ending full stop
    - Inline comments
        - use sparingly
        - should be separated by two spaces from the code
        - start with a # and a single space
    - Block comments go before their corresponding code
        - each line starts with a # followed by a single space, unless there is indented text inside the comment
        - Paragraphs are separated by a single #
- Documentation Strings
    - Write these for all public methods, functions, classes and modules
    - Not necessary for private methods but should have comments describing what it it does
    - These should appear after the `def` line
    - the `"""` that ends the docstring should be on a line by itself
        - unless its a one-liner, in which case it should be on the same line
    - There's no blank line either before or after the docstring
    - The docstring is a phrase ending in a period.
    - Prescribes the function's effect as a command ("Do this", "Return that"), not as a description
    - PEP257 explains docstrings in more detail
- Version bookkeeping
    - If you have subversion, cvs or rcs crud in your source file then do this:
        - ```
          __version__ = "$Revision: 380301e300a6 $"
          # $Source$
          ```
    - This should be included after module's docstring and before any code, separated by blank lines above and below.
- Naming conventions
    - Public variables/functions that are visible to the user should follow conventions that reflect usage rather than implementation
    - Constants should be all capitals with underscore
          ```

- Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
- Package and Module Names
  - Short, all-lowercase names, underscores can be used if needed for readability - but underscores are usually discouraged
  - When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. _socket).
- Class Names
  - Use CapsWords
  - Functions naming convention can be used in cases where the interface is documented and used mostly as a callable.
  - There's a separate convention for builtin names: most are single words (or two words run together), with CapsWords used only for exception names and constants.
- Exception Names
  - Same as classes, with the suffix "Error"
- Global Variable Names
  - Same as functions
  - Modules used via `from M import *` should use the `__all__` mechanism to prevent exporting globals, or prefix globals with an underscore (to indicate these globals are "module non-public").
- Function Names
  - lowercase with underscores separating words
- 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, append a trailing underscore rather than use an abbreviation or spelling corruption.
- Method Names and Instance Variables
  - use function names rules
  - 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
    - 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.)
    - Double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed
- Designing for inheritance
  - a class's methods and instance variables should be public or non-public
    - if in doubt, choose non-public

- Public attributes - expect unrelated clients of your class to use
- Non-public attributes - not intended to be used by third parties
- Public attributes should have no leading underscores.
- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods.
    - keep the functional behavior side-effect free, but side-effects such as caching are fine.
    - Avoid using properties for computationally expensive operations

- Public and internal interfaces
    - Backwards compatibility guarantees apply only to public interfaces
    - Documented Interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces
    - All undocumented interfaces should be assumed to be internal.
    - To support introspection, modules should 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.
    - Imported names should be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module's API

- Programming Recommendations
    - use `.join()` as opposed to concatenation using +=
    - Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators
    - When implementing ordering operations with rich comparisons, 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 a name.
        - eg. `def f(x): return 2*x`
        - as opposed to:
        - `f = lambda x: 2*x`
    - 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 what went wrong, not the locations where exceptions are raised
    - When deliberately replacing an inner exception with another (ie. "raise X"), make sure relevant information is exchanged
    - use `raise ValueError('message')` instead of the older form `raise ValueError, 'message'`

- When catching exceptions, mention specific exceptions instead of using a bare `except:` clause
- limit the `try` clause to the absolute minimum amount of code necessary
- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes
- Use `isinstance()` to compare types
- empty string, lists and tuples return false - so don't use `len()` to check if empty
- Don't compare boolean values to True or False using ==

For a more indepth list read the PEP8 doc page.