NEBULA FRAMEWORK
Style guideline V.1

This document gives coding conventions for the **Nebula Framework** comprising the standard library in the main Python distribution within the development "Best of the Best Practices" (BOBP).

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

Index

Background

Python is the main dynamic language and QML is the user interface markup language used at Nebula Framework. This style guide is a list of *dos and don'ts* for Python - Nebula modules.

To help you format code correctly, exist command-line platforms, pycodestyle (previously known as pep8) who helps you to check your code performance in base to PEP8 facto code style for Python. Install it by running the following command in your terminal:

```
pip install pycodestyle
```

Then run it on a file or series of files to get a report of any violations.

```
pycodestyle optparse.py
```

```
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

The program autopep8 can be used to automatically reformat code in the PEP 8 style. Install the program with:

```
pip install autopep8
```

Use it to format a file in-place with:

```
autopep8 --in-place optparse.py
```

Excluding the --in-place flag will cause the program to output the modified code directly to the console for review. The --aggressive flag will perform more substantial changes and can be applied multiple times for greater effect.

Style

Naming

Function names, variable names, and file names should be descriptive; eschew abbreviation. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your module, and do not abbreviate by deleting letters within a word. And of course, in English.

- Variables, functions, methods, packages, modules

```
lower_case_with_underscores
```

- Classes and Exceptions

```
CapWords
```

- Protected methods and internal functions

```
_single_leading_underscore(self, ...)
```

- Private methods

```
__double_leading_underscore(self, ...)
```

- Constants

```
ALL_CAPS_WITH_UNDERSCORES
```

General:

`module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name.`

Always use a .py filename extension. Never use dashes (-).

Names to avoid

- Single character names except for counters or iterators. You may use "e" as an exception identifier in try/except statements.

```
for e in elements:
        e.mutate()
```

- Avoid redundant labeling, and avoid one-letter variables (esp. l, O, I)..
- Dashes (-) in any package/module name
- __double_leading_and_trailing_underscore__ names (reserved by Python)

Naming convention

- "Internal" means internal to a module, or protected or private within a class.
- Prepending a **single underscore (_) has some support for protecting module variables and functions** (not included with from module import *). While prepending a **double underscore (__ aka "dunder") to an instance variable or method effectively makes the variable or method private** to its class (using name mangling) we discourage its use as it impacts readability and testability and isn't *really* private.
- Place related classes and top-level functions together in a module. Unlike Java, there is no need to limit yourself to one class per module.
- **Use CapWords for class names, but lower_with_under.py for module names**. Although there are some old modules named CapWords.py, this is now discouraged

because it's confusing when the module happens to be named after a class. ("wait – did I write import StringIO or from StringIO import StringIO?")

- Underscores may appear in *unittest* method names starting with test to separate logical components of the name, even if those components use CapWords. One possible pattern is test<MethodUnderTest>_<state>; for example testPop_EmptyStack is okay. There is no One Correct Way to name test methods.

File naming

Python filenames must have a .py extension and must not contain dashes (-). This allows them to be imported and unittested. If you want an executable to be accessible without the extension, use a symbolic link or a simple bash wrapper containing exec "$0.py" "$@".

General guideline

While Python supports making things private by using a leading double underscore __ (aka. "dunder") prefix on a name, this is discouraged. Prefer the use of a single underscore. They are easier to type, read, and to access from small unittests. Lint warnings take care of invalid access to protected members.

Table 1. General guidelines

Type	Public	Internal
Packages	`lower_with_under`	
Modules	`lower_with_under`	`lower_with_under`
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`lower_with_under()`	`_lower_with_under()`

Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected)
Method Names	lower_with_under()	_lower_with_under() (protected)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	
Long Variable Name	long_long_long_variable_name	long_long_long_variable_name

Main

Even a file meant to be used as an executable should be importable and a mere import should not have the side effect of executing the program's main functionality. The main functionality should be in a main() function.

In Python, pydoc as well as unit tests require modules to be importable. Your code should always check if __name__ == '__main__' before executing your main program so that the main program is not executed when the module is imported.

```
def main():
    ...

if __name__ == '__main__':
    main()
```

All code at the top level will be executed when the module is imported. Be careful not to call functions, create objects, or perform other operations that should not be executed when the file is being pydoced.

Function length

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on function length. If a function exceeds about ~ 60 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. **Keeping your functions short and simple makes it easier for other people to read and modify your code**.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Indentation

Use 4 spaces--never tabs. Enough said.

Imports

Reusability mechanism for sharing code from one module to another.

Use import statements for packages and modules only, not for individual classes or functions. Note that there is an explicit exemption for imports from the typing module.

Try to keep the namespace management convention simple. The source of each identifier is indicated in a consistent way; x.Obj says that object Obj is defined in module x.

- Use `import x` for importing packages and modules.
- Use `from x import y` where x is the package prefix and y is the module name with no prefix.
- Use `from x import y as z` if two modules named y are to be imported or if y is an inconveniently long name.
- Use `import y as z` only when z is a standard abbreviation (e.g., np for numpy).

For example the module sound.effects.echo may be imported as follows:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

Imports should be on separate lines.

Yes:

```
import os
import sys
```

No:

```
import os, sys
```

Do not use relative names in imports. Even if the module is in the same package, use the full package name. This helps prevent unintentionally importing a package twice.

Package

Import each module using the full pathname location of the module. All new code should import each module by its full package name.

Imports should be as follows:

Yes:

```
# Reference absl.flags in code with the complete name (verbose).
import absl.flags
from doctor.who import jodie

FLAGS = absl.flags.FLAGS
```

No: *(assume this file lives in doctor/who/ where jodie.py also exists)*

```
# Unclear what module the author wanted and what will be imported.  #The actual
import behavior depends on external factors controlling sys.path.
# Which possible jodie module did the author intend to import?
import jodie
```

The directory the main binary is located in should not be assumed to be in sys.path despite that happening in some environments. This being the case, code should assume that import jodie refers to a third party or top level package named jodie, not a local jodie.py.

Default Argument Values

You can specify values for variables at the end of a function's parameter list, e.g., def foo(a, b=0):. If foo is called with only one argument, b is set to 0. If it is called with two arguments, b has the value of the second argument.

Okay to use with the following caveat:

Do not use mutable objects as default values in the function or method definition.

Yes:

```
def foo(a, b=None):
         if b is None:
             b = []
def foo(a, b: Optional[Sequence] = None):
         if b is None:
             b = []
def foo(a, b: Sequence = ()):  # Empty tuple OK since tuples are immutable
         ...
```

No:

```
def foo(a, b=[]):
         ...
def foo(a, b=time.time()):  # The time the module was loaded???
         ...
def foo(a, b=FLAGS.my_thing):  # sys.argv has not yet been parsed...
         ...
```

Properties

A way to wrap method calls for getting and setting an attribute as a standard attribute access when the computation is lightweight.

Use properties for accessing or setting data where you would normally have used simple, lightweight accessor or setter methods.

Use properties in new code to access or set data where you would normally have used simple, lightweight accessor or setter methods. Properties should be created with the `@property` decorator.

Inheritance with properties can be non-obvious if the property itself is not overridden. Thus one must make sure that accessor methods are called indirectly to ensure methods overridden in subclasses are called by the property (using the Template Method DP).

Yes:

```
import math

     class Square(object):
         """A square with two properties: a writable area and a read-only
perimeter.

         To use:
         >>> sq = Square(3)
         >>> sq.area
         9
         >>> sq.perimeter
         12
         >>> sq.area = 16
         >>> sq.side
         4
         >>> sq.perimeter
         16
         """

         def __init__(self, side):
             self.side = side

         @property
         def area(self):
             """Gets or sets the area of the square."""
             return self._get_area()

         @area.setter
         def area(self, area):
             return self._set_area(area)

         def _get_area(self):
```

```
        """Indirect accessor to calculate the 'area' property."""
        return self.side ** 2

    def _set_area(self, area):
        """Indirect setter to set the 'area' property."""
        self.side = math.sqrt(area)

    @property
    def perimeter(self):
        return self.side * 4
```

True/False evaluations

Use the "implicit" false if at all possible. Python evaluates certain values as False when in a boolean context. A quick "rule of thumb" is that all "empty" values are considered false, so 0, None, [], {}, '' all evaluate as false in a boolean context.

Example: `if foo:` rather than `if foo != []:`. There are a few caveats that you should keep in mind though:

- Never use `==` or `!=` to compare singletons like None. Use `is` or `is not`.
- 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 be a value that's false in a boolean context!
- Never compare a boolean variable to False using ==. Use `if not x:` instead. If you need to distinguish `False` from `None` then chain the expressions, such as `if not x` and x `is not None:`.
- For sequences (strings, lists, tuples), use the fact that empty sequences are false, so `if seq:` and `if not seq:` are preferable to `if len(seq):` and `if not len(seq):` respectively.
- When handling integers, implicit false may involve more risk than benefit (i.e., accidentally handling `None` as `0`). You may compare a value which is known to be an integer (and is not the result of `len()`) against the integer 0.

- Note that '0' (i.e., 0 as string) evaluates to true.

Yes:

```
if not users:
         print('no users')

     if foo == 0:
         self.handle_zero()

     if i % 10 == 0:
         self.handle_multiple_of_ten()

     def f(x=None):
         if x is None:
             x = []
```

No:

```
if len(users) == 0:
         print('no users')

     if foo is not None and not foo:
         self.handle_zero()

     if not i % 10:
         self.handle_multiple_of_ten()

     def f(x=None):
         x = x or []
```

Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block to handle errors or other exceptional conditions.

Exceptions are allowed but must be used carefully.

Exceptions must follow certain conditions:

- Raise exceptions like this: `raise MyError('Error message')` or `raise MyError()`. Do not use the two-argument form (raise MyError, 'Error message').
- Make use of built-in exception classes when it makes sense. For example, `raise a ValueError` if you were passed a negative number but were expecting a positive one. Do not use assert statements for validating argument values of a public API. assert is used to ensure internal correctness, not to enforce correct usage nor to indicate that some unexpected event occurred. If an exception is desired in the latter cases, use a raise statement. For example:

  Yes:

```
def connect_to_next_port(self, minimum):
    """Connects to the next available port.

    Args:
      minimum: A port value greater or equal to 1024.
    Raises:
      ValueError: If the minimum port specified is less than 1024.
      ConnectionError: If no available port is found.
    Returns:
      The new minimum port.
    """
    if minimum < 1024:
      raise ValueError('Minimum port must be at least 1024, not %d.' % (minimum,))
    port = self._find_next_open_port(minimum)
    if not port:
      raise ConnectionError('Could not connect to service on %d or higher.' %
(minimum,))
    assert port >= minimum, 'Unexpected port %d when minimum was %d.' % (port,
minimum)
    return port
```

No:

```
def connect_to_next_port(self, minimum):
    """Connects to the next available port.

    Args:
      minimum: A port value greater or equal to 1024.
    Returns:
      The new minimum port.
    """
    assert minimum >= 1024, 'Minimum port must be at least 1024.'
    port = self._find_next_open_port(minimum)
    assert port is not None
    return port
```

- Libraries or packages may define their own exceptions. When doing so they must inherit from an existing exception class. Exception names should end in `Error` and should not introduce stutter (`foo.FooError`).

- Never use catch-all `except:` `statements`, or `catch Exception` or `StandardError`, unless you are re-raising the exception or in the outermost block in your thread (and printing an error message). Python is very tolerant in this regard and `except:` will really catch everything including misspelled names, `sys.exit()` calls, Ctrl+C interrupts, unittest failures and all kinds of other exceptions that you simply don't want to catch.

- Minimize the amount of code in a try/except block. The larger the body of the try, the more likely that an exception will be raised by a line of code that you didn't expect to raise an exception. In those cases, the try/except block hides a real error.

- When capturing an exception, use `as` rather than a comma.

- Use the `finally` clause to execute code whether or not an exception is raised in the `try` block. This is often useful for cleanup, i.e., closing a file.

Rules

Semicolon

Do not terminate your lines with semicolons, and do not use semicolons to put two statements on the same line.

Line length

Maximum line length is *80 characters*.

Exceptions:

- Long import statements.
- URLs, pathnames, or long flags in comments.
- Long string module level constants not containing whitespace that would be inconvenient to split across lines such as URLs or pathnames.
- Pylint disable comments. (e.g.: # pylint: disable=invalid-name)

Do not use backslash line continuation except for with statements requiring three or more context managers.

Parentheses

Use parentheses sparingly.

It is fine, though not required, to use parentheses around tuples. Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.

Yes:

```
if foo:
        bar()
     while x:
         x = bar()
     if x and y:
         bar()
     if not x:
         bar()
     # For a 1 item tuple the ()s are more visually obvious than the comma.
     onesie = (foo,)
     return foo
     return spam, beans
     return (spam, beans)
     for (x, y) in dict.items(): ...
```

No:

```
if (x):
        bar()
     if not(x):
         bar()
     return (foo)
```

Whitespace

Follow standard typographic rules for the use of spaces around punctuation.

No whitespace inside parentheses, brackets or braces.

Yes:

```
spam(ham[1], {eggs: 2}, [])
```

No:

```
spam( ham[ 1 ], { eggs: 2 }, [ ] )
```

No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon, except at the end of the line.

Yes:

```
if x == 4:
         print(x, y)
     x, y = y, x
```

No:

```
if x == 4 :
         print(x , y)
     x , y = y , x
```

No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.

Surround binary operators with a single space on either side for assignment (=), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), and Booleans (and, or, not). Use your better judgment for the insertion of spaces around arithmetic operators (+, -, *, /, //, %, **, @).

Yes:

```
spam(1)
dict['key'] = list[index]
x == 1
```

No:

```
spam (1)
dict ['key'] = list [index]
x==1
```

Never use spaces around = when passing keyword arguments or defining a default parameter value, with one exception: when a type annotation is present, *do* use spaces around the = for the default parameter value.

Yes:

```
def complex(real, imag=0.0): return Magic(r=real, i=imag)
def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0): return Magic(r = real, i = imag)
def complex(real, imag: float=0.0): return Magic(r = real, i = imag)
```

Functions and methods

In this section, “function” means a method, function, or generator.

A function must have a docstring, unless it meets all of the following criteria:

- not externally visible
- very short
- obvious

A docstring should give enough information to write a call to the function without reading the function's code. The docstring should be descriptive ("""Fetches rows from a Bigtable.""") rather than imperative ("""Fetch rows from a Bigtable."""). A docstring should describe the function's calling syntax and its semantics, not its implementation. For tricky code, comments alongside the code are more appropriate than using docstrings.

A method that overrides a method from a base class may have a simple docstring sending the reader to its overridden method's docstring, such as """See base class.""". The rationale is that there is no need to repeat in many places documentation that is already present in the base method's docstring. However, if the overriding method's behavior is substantially different from the overridden method, or details need to be provided (e.g., documenting additional side effects), a docstring with at least those differences is required on the overriding method.

Certain aspects of a function should be documented in special sections, listed below. Each section begins with a heading line, which ends with a colon. Sections should be indented two spaces, except for the heading.

Args:

List each parameter by name. A description should follow the name, and be separated by a colon and a space. If the description is too long to fit on a single 72-character line, use a hanging indent of 2 or 4 spaces (be consistent with the rest of the file).

The description should include required type(s) if the code does not contain a corresponding type annotation.

If a function accepts *foo (variable length argument lists) and/or **bar (arbitrary keyword arguments), they should be listed as *foo and **bar.

Returns: (or Yields: for generators)

Describe the type and semantics of the return value. If the function only returns None, this section is not required. It may also be omitted if the docstring starts with Returns or Yields (e.g.

"""Returns row from Bigtable as a tuple of strings.""") and the opening sentence is sufficient to describe return value.

Raises:

List all exceptions that are relevant to the interface.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """
    Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.


   :param     big_table: An open Bigtable Table instance.
   :param     keys: A sequence of strings representing the key of each table row
            to fetch.
   :param     other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    :return:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    :raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

Classes

Classes should have a docstring below the class definition describing the class. If your class has public attributes, they should be documented here in an Attributes section and follow the same formatting as a function's Args section.

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

If a class inherits from no other base classes, explicitly inherit from object. This also applies to nested classes.

Yes:

```
class SampleClass(object):
         pass
```

```
class OuterClass(object):

    class InnerClass(object):
        pass


class ChildClass(ParentClass):
    """Explicitly inherits from another class already."""
```

No:

```
class SampleClass:
        pass


    class OuterClass:

        class InnerClass:
            pass
```

Conventions: Ways to do things

There are ways to write code, which are practical for reducing memory usage, increasing speed and make your code easier to read.

Access a Dictionary Element

Don’t use the dict.has_key() method. Instead, use x in d syntax, or pass a default argument to dict.get().

Yes:

```
d = {'hello': 'world'}
```

```
print d.get('hello', 'default_value') # prints 'world'
print d.get('thingy', 'default_value') # prints 'default_value'

# Or:
if 'hello' in d:
    print d['hello']
```

No:

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']    # prints 'world'
else:
    print 'default_value'
```

Short Ways to Manipulate Lists

List comprehensions provide a powerful, concise way to work with lists.

Generator expressions follow almost the same syntax as list comprehensions but return a generator instead of a list.

Creating a new list requires more work and uses more memory. If you are just going to loop through the new list, prefer using an iterator instead.

Yes:

```
valedictorian = max((student.gpa, student.name) for student in graduates)
```

No:

```
# needlessly allocates a list of all (gpa, name) entires in memory
valedictorian = max([(student.gpa, student.name) for student in graduates])
```

Use list comprehensions when you really need to create a second list, for example if you need to use the result multiple times.

If your logic is too complicated for a short list comprehension or generator expression, consider using a generator function instead of returning a list.

Yes:

```
def make_batches(items, batch_size):
    """
    >>> list(make_batches([1, 2, 3, 4, 5], batch_size=3))
    [[1, 2, 3], [4, 5]]
    """
    current_batch = []
    for item in items:
        current_batch.append(item)
        if len(current_batch) == batch_size:
            yield current_batch
            current_batch = []
    yield current_batch
```

Never use a list comprehension just for its side effects.

Yes:

```
for x in sequence:
    print(x)
```

No:

```
[print(x) for x in sequence]
```

For filtering use a list comprehension or generator expression.

Yes:

```
# comprehensions create a new list object
```

```
filtered_values = [value for value in sequence if value != x]

# generators don't create another list
filtered_values = (value for value in sequence if value != x)
```

No:

```
# Filter elements greater than 4
a = [3, 4, 5]
for i in a:
    if i > 4:
        a.remove(i)
#Don't make multiple passes through the list.
while i in a:
    a.remove(i)
```

Read from a file

Use the with open syntax to read from files. This will automatically close files for you.

Yes:

```
with open('file.txt') as f:
    for line in f:
        print line
```

No:

```
f = open('file.txt')
a = f.read()
print a
f.close()
```

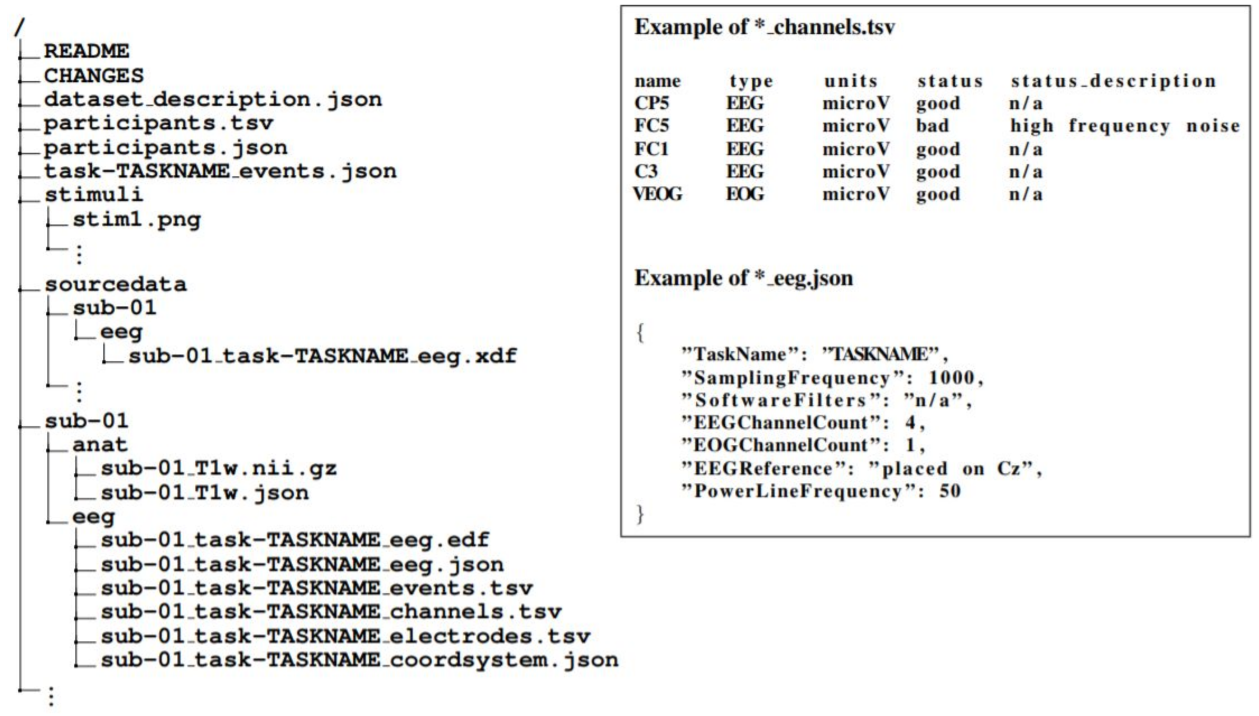
The with statement is better because it will ensure you always close the file, even if an exception is raised inside the with block.

Obviously, we do not cover all the conventions here, for this you can consult PEP8.

Signal format

The Brain Imaging Data Structure (BIDS) project is a quickly evolving effort among the human brain imaging research community to create standards allowing researchers to readily organize and share study data within and between laboratories.

The specification follows the general BIDS: Each subject has a directory of raw data containing subdirectories for each session and modality. This is accompanied by a dataset_description.json file and a metadata file with the suffix _eeg.json, that specifies the task, the EEG system used (amplifier, hardware filter, cap, placement scheme, etc.)



```
/
├─README
├─CHANGES
├─dataset_description.json
├─participants.tsv
├─participants.json
├─task-TASKNAME_events.json
├─stimuli
│ ├─stim1.png
│ └─:
├─sourcedata
│ ├─sub-01
│ │ └─eeg
│ │   └─sub-01_task-TASKNAME_eeg.xdf
│ └─:
├─sub-01
│ ├─anat
│ │ ├─sub-01_T1w.nii.gz
│ │ └─sub-01_T1w.json
│ └─eeg
│   ├─sub-01_task-TASKNAME_eeg.edf
│   ├─sub-01_task-TASKNAME_eeg.json
│   ├─sub-01_task-TASKNAME_events.tsv
│   ├─sub-01_task-TASKNAME_channels.tsv
│   ├─sub-01_task-TASKNAME_electrodes.tsv
│   └─sub-01_task-TASKNAME_coordsystem.json
└─:
```

Example of *_channels.tsv

```
name   type   units    status   status_description
CP5    EEG    microV   good     n/a
FC5    EEG    microV   bad      high frequency noise
FC1    EEG    microV   good     n/a
C3     EEG    microV   good     n/a
VEOG   EOG    microV   good     n/a
```

Example of *_eeg.json

```
{
    ”TaskName”: ”TASKNAME”,
    ”SamplingFrequency”: 1000,
    ”SoftwareFilters”: ”n/a”,
    ”EEGChannelCount”: 4,
    ”EOGChannelCount”: 1,
    ”EEGReference”: ”placed on Cz”,
    ”PowerLineFrequency”: 50
}
```

Figure 1. A prototypical directory tree of a BIDS dataset containing EEG data. At the root level of the directory, the README, CHANGES, and dataset_description.json files provide basic information about the dataset. A participants.tsv data file is accompanied by a participants.json file, which contains the description of the columns in its associated .tsv file. The panel in the upper right of the figure provides examples on the typical format within a .tsv and .json file. Usually, each .tsv file is accompanied by a .json file that provides metadata. The EEG data and anatomical MRI scans are saved per subject within the eeg and anat subdirectories respectively. If the original data is not supported by BIDS, it can be included in an additional source data directory. Finally, a stimuli directory contains the stimuli that were presented to the participants in the experiment.

Code documentation

If you got here you realized how important it is to document everything you do. But if not, then let me quote something Guido mentioned to us at a recent PyCon:

```
"Code is more often read than written."
-- Guido Van Rossum
```

When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important. If you're like me, you've probably opened up old codebases and wondered to yourself, "What in the world was I thinking?" If you're having a problem reading your own code, imagine what your users or other developers are experiencing when they're trying to use or contribute to your code.

Conversely, I'm sure you've run into a situation where you wanted to do something in Python and found what looks like a great library that can get the job done. However, when you start using the library, you look for examples, write-ups, or even official documentation on how to do something specific and can't immediately find the solution.

After searching, you come to realize that the documentation is lacking or even worse, missing entirely. This is a frustrating feeling that deters you from using the library, no matter how great or efficient the code is.

```
"It doesn't matter how good your software is, because if the
documentation is not good enough, people will not use it."
-- Daniele Procida
```

In this chapter, you'll learn from the ground up how to properly document your Python code from the smallest of scripts to the largest of Python/Nebula projects to help prevent your users from ever feeling too frustrated to use or contribute to your project.

Commenting and Documenting Code

First we will distinguish some aspects of what is to comment and what is to document.

Commenting is describing your code to/for developers. The intended main audience is the maintainers and developers of the Nebula code. In conjunction with well-written code, comments help to guide the reader to better understand your code and its purpose and design. Code tells you how; Comments tell you why.

Documenting code is describing its use and functionality to your users. While it may be helpful in the development process, the main intended audience is the users.

The following section describes how and when to comment your code.

Commenting code

Comments are created in Python using the pound sign (#) and should be brief statements no longer than a few sentences.

For Nebula the comments should have a maximum length of 72 characters. If a comment is going to be greater than the comment char limit, using multiple lines for the comment is appropriate.

To start a comment, if is inline then 2 spaces are added before # and 1 space after #. If is after a function or class the 2 spaces before the # are not added.

```
def foo(x):
    # A very long statement that just goes on and on and on and on and
    # never ends until after it's reached the 80 char limit
    print(f'This is a loooooooooooooooooooooooooooooooooooooooooooooooong
statement by {x}')
```

Comment has several purposes, including:

- Planning and Reviewing: When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. Remember to remove these comments once the actual coding has been implemented and reviewed/tested.

```
# First step
# Second step
# Third step
```

- Code Description: Comments can be used to explain the intent of specific sections of code.

```
# Attempt a connection based on previous settings. If unsuccessful,
# prompt user for new settings.
```

- Algorithmic Description: When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

```
# Using quick sort for performance gains and how increase the signal
```

- Tagging: The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: BUG, FIXME, and TODO.

```
# TODO: Add condition for when val is None
```

Comments to your code should be kept brief and focused. Avoid using long comments when possible. Additionally, you should use the following four essential rules as suggested by PEP8:

- Keep comments as close to the code being described as possible. Comments that aren't near their describing code are frustrating to the reader and easily missed when updates are made.
- Don't use complex formatting (such as tables or ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.
- Don't include redundant information. Assume the reader of the code has a basic understanding of programming principles and language syntax.
- Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.

Remember that comments are designed for the reader, including yourself, to help guide them in understanding the purpose and design of the software.

Commenting Code via Type Hinting

Type Hinting is an additional form to help the readers of your code. It allows the developer to design and explain portions of their code without commenting.

```
# This is how you annotate a function definition
def stringify(num: int) -> str:
    return str(num)

def hello_name(name: str) -> str:
    return (f"Hello {name}")

# And here's how you specify multiple arguments
def plus(num1: int, num2: int) -> int:
```

```python
    return num1 + num2

# Add default value for an argument after the type annotation
def f(num1: int, my_float: float = 3.5) -> float:
    return num1 + my_float


# You can of course split a function annotation over multiple lines
def send_email(address: Union[str, List[str]],
               sender: str,
               cc: Optional[List[str]],
               bcc: Optional[List[str]],
               subject='',
               body: Optional[List[str]] = None
               ) -> bool:
```

From examining the type hinting, in the second example function you can immediately tell that the function expects the input variable to be of a type str, or string. You can also tell that the expected output of the function will be of a type str, or string, as well. While type hinting helps reduce comments, take into consideration that doing so may also make extra work when you are creating or updating your project documentation.

You can read more about Type Hinting in this link: PEP484

Docstrings

Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your Nebula project's documentation.

Along with docstrings, Python also has the built-in function help() that prints out the objects docstring to the console. Here's a quick example:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors are specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 # Truncated for readability
```

Python has one more feature that simplifies docstring creation. Instead of directly manipulating the __doc__ property, the strategic placement of the string literal directly below the object will automatically set the __doc__ value. Here's what happens with the same example as above:

```
def say_hello(name):
    """
    A simple function that says hello... Richie style
    """
    print(f"Hello {name}, is it me you're looking for?")


>>> help(say_hello)
Help on function say_hello in module __main__:
```

```
say_hello(name)
    A simple function that says hello... Richie style
```

Knowing what is docstring, it will be understood then that the behavior of __doc__ is known, because this is modifiable in case you want to perform. If you want to read more about, follow this link: PEP257. For a better review, let's observe how the documentation of a function or a class is done in its absence.

The docstring format is followed by PEP257 and formatting type reStructured Text. This is the official Python documentation standard; Not beginner friendly but feature rich.

Follow this structure:

```
'''
:param self:
:param myParam1:
:param myParam2:
:return:
'''
```

An a preview, a sample of how it would apply:

```
"""
Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
```

```
"""
```

Example:

```
class Animal:
    """
    A class used to represent an Animal

    ...

   :attribute    says_str : str
        a formatted string to print out what the animal says
   :attribute     name : str
        the name of the animal
    :attribute   sound : str
        the sound that the animal makes
    :attribute   num_legs : int
        the number of legs the animal has (default 4)

    :method      says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"


    def __init__(self, name, sound, num_legs=4):
        """
        :param   name : str
            The name of the animal
        :param   sound : str
            The sound the animal makes
        :param   num_legs : int, optional
            The number of legs the animal (default is 4)
        """

        self.name = name
        self.sound = sound
        self.num_legs = num_legs

    def says(self, sound=None):
        """
        Prints what the animals name is and what sound it makes.
        If the argument `sound` isn't passed in, the default Animal
        sound is used.

        :param   sound : str, optional
            The sound the animal makes (default is None)

        :raises  NotImplementedError:
            If no sound is set for the animal or passed in as a
            parameter.
        """

        if self.sound is None and sound is None:
            raise NotImplementedError("Silent Animals are not
supported!")

        out_sound = self.sound if sound is None else sound
        print(self.says_str.format(name=self.name, sound=out_sound))
```

Package docstrings should be placed at the top of the package's __init__.py file. This docstring should list the modules and sub-packages that are exported by the package.

Module docstrings are similar to class docstrings. Instead of classes and class methods being documented, it's now the module and any functions found within. Module docstrings are placed at the top of the file even before any imports. Module docstrings should include the following:

- A brief description of the module and its purpose
- A list of any classes, exception, functions, and any other objects exported by the module

The docstring for a module function should include the same items as a class method:

- A brief description of what the function is and what it's used for
- Any arguments (both required and optional) that are passed including keyword arguments
- Label any arguments that are considered optional
- Any side effects that occur when executing the function
- Any exceptions that are raised
- Any restrictions on when the function can be called

Nebula Project Documentation

Nebula's projects come in all sorts of shapes, sizes, and purposes, but all of them generally need a input like signals or images, and produces a series of outputs or a specific output. The way you document your project should suit your specific situation. Keep in mind who the users of your project are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

```
Project_root/
```

```
|
|-- project/  # Project source code
|-- docs/
|-- README
|-- HOW_TO_CONTRIBUTE
|-- CODE_OF_CONDUCT
|-- examples.py
```

Main Sections of the docs Folder

1. Tutorials: Lessons that take the reader by the hand through a series of steps to complete a projects (or meaningful exercise). Geared towards the users learning.
2. How-To Guides: Guides that take the reader through the steps required to solve a common problem (problem-oriented recipes).
3. References: Explanations that clarify and illuminate a particular topic. Geared towards understanding.
4. Explanations: Technical descriptions of the machinery and how to operate it (key classes, functions, APIs, and so forth). Think Encyclopedia article.

In the end, you want to make sure that your users have access to the answers to any questions they may have. By organizing the Nebula project in this manner, you'll be able to answer those questions easily and in a format they'll be able to navigate quickly.

Nebula's projects can be generally subdivided into two major types: Private, and Free/Open Source.

Private Project

Private projects are projects intended for laboratory use only and generally aren't shared with other users or laboratories. Documentation can be pretty light on these types of projects. There are some recommended parts to add as needed:

- Readme: A brief summary of the project and its purpose. Include any special requirements for installation or operating the project.
- examples.py: A Nebula script (can be Python, QML or another) file that gives simple examples of how to use the project.

Remember, even though private projects are intended for you personally, you are also considered a user. Think about anything that may be confusing to you down the road and make sure to capture those in either comments, docstrings, or the readme.

Free/Open Source Project

Free/Open Source projects are projects that are intended to be shared with a large group of users and can involve large development teams, in case. These projects should place as high of a priority on project documentation as the actual development of the project itself. Some of the recommended parts to add to the project are the following:

- Readme: A brief summary of the project and its purpose. Include any special requirements for installing or operating the project. Additionally, add any major changes since the previous version. Finally, add links to further documentation, bug reporting, and any other important information for the project.
- How to Contribute: This should include how new contributors to the project can help. This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.
- Code of Conduct: Defines how other contributors should treat each other when developing or using your software. This also states what will happen if this code is

broken. If you're using Github or GitLab, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.

- License: A plaintext file that describes the license your project is using. For Free/Open Source projects especially, consider adding this.
- docs: A folder that contains further documentation. The next section describes more fully what should be included and how to organize the contents of this folder.

Reference

You can check for more in the following documents, and of course the content here is inspired by:

Guido van Rossum, Nick Coghlan. (2001). PEP 8 -- Style Guide for Python Code. 07/06/2019, de Python Software Fundation Sitio web: https://www.python.org/dev/peps/pep-0008/

Guido van Rossum, Łukasz Langa. (2014). PEP 484 -- Type Hints. 07/06/2019, de Python Software Fundation Sitio web: https://www.python.org/dev/peps/pep-0484/

Guido van Rossum. (2001). PEP 257 -- Docstring Conventions. 07/06/2019, de Python Software Fundation Sitio web: https://www.python.org/dev/peps/pep-0257/

Google. (2018). Google Python Style Guide. 07/06/2019, de Alphabet Sitio web: https://google.github.io/styleguide/pyguide.html

Pernet, C. R., Appelhoff, S., Flandin, G., Phillips, C., Delorme, A., & Oostenveld, R. (2018, December 6). BIDS-EEG: an extension to the Brain Imaging Data Structure (BIDS) Specification for electroencephalography. https://doi.org/10.31234/osf.io/63a4y