Code Guidelines

School Simplified IT Dept. - Bot Development

0. Table of Contents

1. Introduction

To understand code better and faster and to have consistency through the code, it's important to follow and adhere to code guidelines. Code guidelines is about consistency. Consistency in code is important. Consistency within a project is more important. Consistency within one module or function is the most

important. This document provides the guidelines for Python Code at School Simplified IT Dept. – Bot Development and is intended to be a reference of compliance with those guidelines. The guidelines are based on the PEP8 – Style Guide for Python Code and the PEP257 – Docstring Conventions and also provides internal guidelines to work with the library discord.py.

2. Code Layout

2.1. Indentation

Use 4 spaces for one indentation level. **In PyCharm, 1 tab is 4 spaces.**

2.1.1. Function definition and call

```
# 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)
```

```
# 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)
```

2.1.2. If-Statements

For `if`-statements there are no specific rules on how to do indent. However, acceptable options include, but are not limited to:

```
# No extra indentation.
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# No extra indentation with an empty line after if-statement.
# PREFERRED
if (this_is_one_thing and
    that_is_another_thing):

    do_something()
```

2.1.3. Closing brace/bracket/parenthesis

The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

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,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

2.2. Line Length

We don't apply this rule, even when PEP8 requires to limit all lines to a maximum length of 79 characters. However, limit it to something reasonable. In PyCharm the limit is 120 characters by default which is marked with a thin, grey line.

IMAGE HERE

The preffered way to wrapping long lines is by using the Python's implied line continuation inside brackets. Long lines can also be broken over multiple lines by using a **backslash** for line continuation if it's needed.

```
# Python's implied line continuation inside brackets
if (True == True and False == False
    or False == False and True == True)

# Backslash line continuation
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

2.3. Line Break at Operators

For readability it's recommended to line break **before** the operator (see below). However, it's not mandatory.

```
# Recommended:

income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

2.4. Blank Lines

Top-level classes and functions: Surrounded by 2 blank lines

```
import sys


class Foo:
    # ...


def func():
    # ...
```

Methods inside a class: Surrounded by 1 blank line

```
class Foo:

    def foo():
        # ...

    def boo():
        # ...
```

Extra blank lines may be used (sparingly) to separate groups of code. However it should be reasonable, means no extra blank lines between a bunch of one-liners.

```
# Correct:

from datetime import datetime, timedelta

# Block 1: Calculation
factor_1 = 1 + 1
factor_2 = 2 + 2
result = factor_1 * factor_2

# Block 2: Datetime creation
now = datetime.now()
time_add = timedelta(hours=result)
dt = now + time_add
```

```
# Wrong:

from datetime import datetime, timedelta

result = (1+1) * (2+2)
# this blank line is unnecessary
now = datetime.now() + timedelta(hours=result)
```

2.5. Imports

- Imports should be on separate lines.

```
# Correct:
import sys
import os
```

```
# Wrong:
import sys, os
```

Imports like `from ... import` on one line is **allowed**.

```
from discord.ext import commands, menus
```

- Imports are always at the top of the file, just after any module comments and docstrings, and before module variables and constants

  Also, imports should be **grouped** in the following order:

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

  Between each group put a **blank line** and sort it **alphabetically**.

```
"""
This module puts a random number into a google spreadsheet cell.
"""

# Standard library imports
import random

# Related third party imports
import gspread

# Local application/library specific imports
from core.common import GOOGLE_CREDS


# Module variables
gspread_user = GOOGLE_CREDS["username"]
gspread_pw = GOOGLE_CREDS["password"]
```

  To sort the imports, you can use isort which does that for you.

- 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. It's also prone for bugs.

2.6. 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:

```
"""
This module handles stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'School Simplified IT Dept. - Bot Development'

import os
import sys
```

3. String Quotes

In Python, single-quoted strings and double-quoted strings are the same. You can either use double-quotes (`"Hello World!"`) or single quotes (`'Hello World!'`).

For triple-quoted strings, always use double quotes to be consistent with the docstring convention.

4. Whitespace in Expressions and Statements

4.1 Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

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

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

- Between a trailing comma and a following close parenthesis:

```
# 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. Exception: when a slice parameter is omitted, the space is omitted:

```
# Correct:
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]
```

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

```
# Correct:
spam(1)
```

```

```

```
# Wrong:
spam (1)
```

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

```
# Correct:
dct['key'] = lst[index]
```

```
# Wrong:
dct ['key'] = lst [index]
```

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

```
# Correct:
x = 1
y = 2
long_variable = 3
```

```
# Wrong:
x             = 1
y             = 2
long_variable = 3
```

4.2 Other important things

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker.
- 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 one whitespace around the operators with the lowest priority(ies).

```
# Correct:
i = i + 1
submitted += 1
x = x*2 - 1
```

```
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

```
# 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. (See **Function Annotations** below for more about function annotations.):

```
# Correct:
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)
```

- 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)
```

When combining an argument annotation with a default value, however, do use spaces around the `=` sign:

```
# Correct:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

```
# Wrong:
def munge(input: AnyStr=None): ...
def munge(input: AnyStr, limit = 1000): ...
```

- Don't use multiple statements on the same line

```
# Wrong
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

Rather do:

```
# Correct:
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Definitely not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)
```

```
if foo == 'blah': one(); two(); three()
```

5. 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! Ensure that your comments are clear and easily understandable to others.

5.1 Documentation Strings

- Write docstrings for **all public modules, functions, classes and methods.** For non-public (`def __foo(): ...`) methods you don't have to use docstrings. A description with a comment is useful though. This comment should appear after the `def` line.

Multiline docstrings:

```
"""
Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

One liner docstrings:

```
"""Return an ex-parrot."""
```

6. Naming Conventions

6.1 Descriptive: Naming Styles

The following naming styles are commonly used:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`

- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase)
  Note: When using acronyms in CapWords, capitalize all the letters of the acronym. HTTPServerError is better than HttpServerError.
- `mixedCase`
- `Capitalized_Words_With_Underscores` (**Dont use this**)

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 <module> import *` does not import objects whose names start with one 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 in class `FooBar`, `__boo` becomes `_FooBar__boo`. Means invoking`FooBar.__boo` won't work.
- `__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.**

6.7 Prescriptive: Naming Conventions

Names to avoid as single character variable names:

- `l` (loweercase letter "el")
- `O` (uppercase letter "oh")
- `I` (uppercase letter "eye") In some fonts these characters are indistinguishable.

6.7.1 Package and Module Names

Short, all-lowercase names, underscores only if it improves readability (e.g. `var = "string"`)

6.7.2 Class Names

CamelCase convention (e.g. `class FooBar`)

6.7.3 Function and Variable Names

Lowercase, with words separated by underscores for readability

```
# Correct:
def create_foo():
  my_var = "string"
  num = 1
```

6.7.4 Method Names and Instance Variables

Same rule as for functions and variable names.

- Use one leading underscore **only for non-public methods and instance variables** (not two) -> `_my_var`, `_create_foo`.
- Use two leading underscores **only to avoid name clashes with subclasses**. Although if class `FooBar` has an attribute named `__bar`, it cannot be accessed by `Foo.__bar`.

6.7.5 Constants

Defined on a module level and written in all capital letters with underscores for separating words. (e.g. `MAX_TIME`, `TOTAL`)

6.7.6 Designing for Inheritance