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