PEP 8 -- Style Guide for Python Code

https://www.python.org/dev/peps/pep-0008/
<https://www.python.org/dev/peps/pep-0008/>

Chapter 2: Syntax and Style Guidlines

> **code is read much more often than it is written** -- Guido's key insights

One of the main feature of Python is that it forces the developers to write the code in proper formatting and failing to do so will always result in invalid code. This has been achived by removing the need for markers for block, such as "{ }" in C/C++/Java and using spaces/tab instead.

In this section we will be discussing major points of language syntax such as comments, blocks multiline code and introduction to PEP 8

The command `print` inserts spaces between expressions that are received as a parameter, and a newline character at the end, unless it receives a comma at the end of the parameter list.

Blocks

In Python, code blocks are defined by the use of indentation, which should be constant in the code block, but it is considered good practice to maintain consistency throughout the project and avoid mixing tabs and spaces.

The line before the block always ends with a colon (:) and is a control structure of the language or a statement of a new structure (function, class for example).

Code layout

Lines and Indentation

Python don't braces to mark blocks of code for class, function definitions or flow controls. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is not , but all statements within the block must be indented the same amount.

Example:

```
if True:
    print("True")
else:
    print("False")
```

```
True
```

```
if True:
            print("True")
else:
            print("False")
```

```
True
```

```
if (True):  print("True") else:  print("False")
```

```
  File "<ipython-input-1-15fb41c8267c>", line 1
    if (True):  print("True") else:  print("False")
                                  ^
SyntaxError: invalid syntax
```

Execution of below block of code will result in `error` as the indentation level is not uniform in python IDE.

```
if True:
    print("Answer")
    print("True")
else:
   print("Answer")
```

```
    print("False")


Answer
True
```

The closing brace/bracket/parenthesis on multi-line 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 multi-line construct, as in:

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

Comments

The character `#` marks the beginning of a comment. Any text after the `#` will be ignored until the end of the line, with the exception of functional comments.

Functional comments are used to:

- change the encoding of the source file of the program by adding a comment with the text `# - * - coding: <encoding> - # -` at the beginning of the file, in which `<encoding>` is the file encoding (usually latin1 or utf-8). Changing encoding is required to support characters that are not part of the English language, in the source code of the program.
- define the interpreter that will be used to run the program on UNIX systems, through a comment starting with `#!` at the beginning of the file, which indicates the path to the interpreter (usually the comment line will be something like `#! / usr / bin / env python`).

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single # .

```
# Date:
# Time:
# Author:
# Method Name:
# Description:
```

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Tips for 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!
- Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.
- You should use two spaces after a sentence-ending period.
- When writing English, follow Strunk and White.
- Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

```
#!/usr/bin/env python

# A code line that shows the result of 7 times 3
print (7 * 3) # This is also a comment
"""
This is a multi
line comment / String
"""
st = """
This is a multi line

String
"With me"
'
'
"""
print(st)
```

```
21

This is a multi line

String
"With me"
'
'
```

Indentation

- Use 4 spaces per indentation level
- If your code has continuation lines, then they should align wrapped elements either vertically using Python''s implicit line joining inside parentheses, brackets and braces, or using a **hanging indent**.
- When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

```
# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Aligned with opening delimiter.
foo = long_function_name("var_one", "var_two",
                         "var_three", "var_four")


# Hanging indents should add a level.
foo = long_function_name(
    "var_one", "var_two",
    "var_three", "var_four")
```

Tabs or Spaces?

> **""" Spaces are the preferred indentation method """**

Tabs should be used solely to remain consistent with code that is already indented with tabs.

> Python 3 disallows mixing the use of tabs and spaces for indentation.

Maximum Line Length

- Limit all lines to a maximum of 79 characters.
- For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
- Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.
- The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.
- Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the nominal line length from 80 to 100 characters (effectively increasing the maximum length to 99 characters), provided that comments and docstrings are still wrapped at 72 characters.
- The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72).
- The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.
- Backslashes may still be appropriate at times. For example, long, multiple with -statements cannot use implicit continuation, so backslashes are acceptable:

```
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())
```

Multiline code

In Python, multi-line code is supported by using the backslash character (`\`) at the end of the line or parentheses, brackets or braces in expressions that use such characters.

Examples of broken lines:

```
# A line broken by backslash
a = 7 * 3 + \
5 / 2


# A list (broken by comma)
b = ['a', 'b', 'c',
'd', 'e']

# A function call (broken by comma)
c = range(1,
        11)

# Prints everything
print(a, b, c)


# For i on the list 234, 654, 378, 798:
for i in [234, 654, 378, 798]:
    # If the remainder dividing by 3 is equal to zero:
    if i % 3 == 0: #{
       # Prints...
        print (i, '/ 3 =', i / 3)
    #}
```

The operator `%` computes the modulus (remainder of division).

line break before or after a binary operator

Donald Knuth explains the traditional rule in his Computers and Typesetting series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations".

Following the tradition from mathematics usually results in more readable code:

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally.

> *For new code Knuth's style is suggested*.

Blank Lines

- Surround top-level function and class definitions with **two blank lines**.
- Method definitions inside a class are surrounded by a **single blank line**.
- Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).
- Use blank lines in functions, sparingly, to indicate logical sections.
- Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

Imports

Imports should usually be on separate lines, e.g.:

```
# Yes:
import os
import sys
```

```
# No:
import sys, os

# It's okay to say this though:
from subprocess import Popen, PIPE
```

Imports should be grouped in the following order:

- standard library imports
- related third party imports
- local application/library specific imports

blank line between each group of imports is adviced.