Page No.
Date

SHAIKH SULTAN,
FYIT, 80

IT TOOLS PEP 8 Assignment.

INTRODUCTION:

- What is PEP?

→ - The PEP is an abbreviation form of Python Enterprise Proposal.
- Writing code with a proper logic is a key factor of Programming.
- PEP 8 is a document that provide various ~~guild~~ guidelines to write the readable in Python
- PEP 8 describes how the developer can write beautiful code.
- It was officially written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan.
- The main aim of PEP is to enhance the readability and consistency of code

- CODE - LAYOUT:

- INDENTATION:

- Use 4 spaces per indentation level.

- Unlike other programming languages, the indention is used to define the code block in python
- The indentation part of the python programming language and it determines the level of lines of code
- EG: x = 5

```
if x == 5:
    print ('x is larger than 5')
```

- In the above example, the indented print statement will get executed if the condition of ~~its~~ if statement is true
- This indentation defines the code block and tells us what statement execute when a function is called or contion trigger.

- 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 disallows : mixing 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 restriction (docstring or comments), the line length should be limited to 72 character
  - Some teams strongly ~~pe~~ prefer a longer line ~~break~~ length
  - For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the line length limit upto 99 characters, provided that comment and docstring are still wrapped at 72 character
  - The Python standard library is conservative and require limiting lines to 79 characters

- SHOULD A LINE BREAK BEFORE OR AFTER A BINARY OPERATOR
  - The lines before or after a binary operator is a tradition approach
  - But its affects the readability extensively because the operators are scattered across the different screens, and each operator is kept away from its operand and onto the ~~pe~~ previous line

Page No.
Date

• EG: Total_marks = (English_marks
+ maths_mark
+ (science_marks - biology marks)
+ physics-marks

Python allows us to break line before or after a binary operator, as long as the convention is consistent locally.

- IMPORTING MODULE
- We should import the module in the separates line as follows

Eg: From subprocess import Popen, PIPE
The import statement should be written at the top of the file or just after any module comment. Absolute imports are the recommended because they are more readable and tend to be better behaved

import mypkg.sibling
from mypkg import sibling
from mypkg import example

However, we can use the explicit relative imports instead of absolutes, imports especially dealing with complex packages

- BLANK LINES

- Blank lines can be improved the readability of Python code.
- If many lines of code bunched together the code will becomes harder to read
- We can remove this by using the many blank vertical line, and the reader might need to scroll more than necessary
- Top-level function and classes with two lines : Put the extra vertical space around them so, that it can be understandable.

```
EG: class Firstclass:
        pass

    class Secondclass:
        pass

    def main_function():
        return None
```

- Single blank line inside classes: This functions that we define in the class is related to one another.

```
EG: class Firstclass:
        def method_one(self):
            return None

        def second_two(self):
            return None
```

Page No.
Date

- Use blank lines inside the function:
sometimes, we need to write a complicated function has consists of several step before the return statement
- We can add the blank line between each step
- EG: def cal_variance(n_list):
    list_sum = 0
    For n in n_list
        list_sum = list_sum + n
    mean = list_sum / ten(n_list)

    square_sum = 0
    for n in n_list:
        square_sum = square_sum + n**2
    mean_squares = square_sum / len(n_list)
    return mean_squares - mean**2

The above way can remove the white spaces to improve the readability of code

- PUT THE CLOSING BRACES -

- We can break lines ~~like~~ inside parentheses, brackets using the line continuations
- PEP 8 allows us to use closing braces in implies line continuations.
- EG: - Line up the closing brace with the first non-whitespace

```
list_numbers = [
        5, 4, 1
        4, 6, 3
        7, 8, 9
        ]
```

- - Line up the closing braces with the first character of line

```
list_numbers = [
        5, 4, 1
        4, 6, 3
        7, 8, 9
]
```

Both method are suitable to use, but consistency is key, so choose any one and continue with it

Page No.
Date

- COMMENTS:
- comment are the integral part of the any programming language
- These are the best way to explain the code
- When we documented our code with the proper comment anyone can able to the understand the code
- But we should remember the following points
  - start with the capital latter, and write complete sentence
  - update the comment in case of a change in code
  - limit the line length of comment and docstring to 72 character

BLOCK COMMENT

- Block comment are the good choice for the small section of code.
- such comment are useful when we write several line codes to perform a single action such as iterating a loop
- They help us to understand the process of the code.

PEP 8 provides the following rules to write comment block :

- Indent block comment should be at the same level
- start each line with # followed by a single space
- separate line using the single #
- EG: for i in range (0, 5):
  # loop will iterate over i five times and print out the value of i
  # new line character
  print (i, '\n')

We can use more than paragraph for the technical code.

-INLINE COMMENTS

- linline comments are used to explain the single statement in a piece of code
- We can quickly get the idea of why we wrote that particular line of code
- PEP 8 specifies the following rules for the inline comments
  - start comment with the # and single space
  - Use inline comment carefully
  - We should separate the inline comment on the same line as the statement they refer

- EG: a=10 # The a is variable that holds integer value

Sometimes, we can use the naming convention to replace the inline comment

- x = 'Peter Decosta' # This is a student name

We can use the following naming convention

- student_name = 'Peter Decosta'

Inline comment are essential but block comments make the coder more readable

- Avoid Unnecessary Adding Whitespaces
- In some cases, use of whitespaces can make the code much harder to read
- Too much whitespaces can make code overly spares and difficuilt to understand
- We should avoid adding whitespace at the end of a line
- This known as trailing whitespaces

EG: # Recommended
list 1 = [1, 2, 3]

Not Recommended
List 1 = [1, 2, 3]

EG: x = 5
y = 6
# Recommended
print (x, y)
Not recommended
print (x, y)

- STRING QUOTES:
- In python, single-quoted strings and doubled-quoted strings are the same
- This PEP does not make a recommendation for this.
- Pick a rule and stick to it
- When a string contain single or double quotes character, however, use the other one to avoid back slashes in the string, it improve readability
- For triple-quoted strings, always use double quote character with the docstring convention in PEP 257

- WHEN TO USE TRAILING COMMAS

- Trailing commas are usally optional except they are mandatory when making a tuple of one element
- For clarity, it is recommended to surround the latter in (technically redundant) parentheses :
- EG: # Correct

```
FILES = ('setup.cfg')
```

Wrong

```
FILES = 'setup.cfg'
```

- When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items are expected to be extended over time
- EG: # correct

```
FILES = [
    'setup.cfg',
    'tox.ini',
]
initialize(FILES,
           error = True,
           )
```

Wrong :

```
FILES = ['setup.cfg', 'tox.ini', ]
initialize (FILES, error = True)
```

- NAMING CONVENTIONS

- The naming convention of Python's library are a bit of mess, so we'll never get this completely consistent -nevertheless, here are the currenty recommended naming standards,

- Overiding Principles :

- Name that are visible to the user as public part of the API should follow conventions that reflect usage rather than implementation

- Descriptive : Naming styles

- There are a lot of different naming styles.

- It helps to be able to recognize what naming style is being used, independently from what they are used for :

- The following naming styles are commonly distinguished :

- b (single lowercase letter)
- B (single upper case letter)
- lower case
- lowercase_with_underscore

- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- capitalized words (or capwords, or camelc-
  ase --- so named because of the bum
  py look of its letter)
- mixed case (differs from capitalized wo-
  rd by initial lowercase character!)
- Capitalized_Word_With_Underscore (ugly!)
- In addition the following special
  form using leading or trailing unde-
  rscores are recognized
- _single_leading_underscore : weak "inter-
  nal use" indicator
- EG: from M import *
  does not import object whose
  names start with an underscore
- single_trailing_underscore_
  used by convention to avoid conflicts
  with Python keyword
- EG: tkinter.Toplevel(master, class_=
  'className')
- __double_leading_underscore:
  when naming a class attribute, invokes
  name mangling (inside class FooBar,
  __boo becomes _FooBar__boo;)
- __double_leading_and_trailing_underscore__

"magic" object or attribute that live in user-controlled namespaces
- EG: __init__, __import__ or __file__
  Never invent such names; only use them as documented.

- Prescriptive: Naming convention
- Names to Avoid:
- Never use the character 'l' (lowercase letter el), 'O' (uppercase letter oh) or 'I' (uppercase letter eye) as single character variable names
- In some fonts, these characters are indistinguishable from the numerals one and zero
- When tempted to use 'l', use 'L' instead
- ASCII COMPATIBILITY
- Identifier used in the standard library must be ASCII compatible as describe in the policy section of PEP 3131
- Packages and Module Names
- Modules should have short, all-lower case names,
- Underscores can be used in the module name if it is improves readability
- Python packages should also have short, all lowercase names, although the use of underscores is discouraged

Page No.
Date

- When an extension module written in C or C++ has an accompanying Python module that provides a higher level

- CLASS NAME
  - class names should normally use the capwords convention
  - The naming convention for function may be used instead in cases where the interface is documented and used primarily as a callable
- FUNCTIONS AND VARIABLE NAME
  - function names should be lowercase, with word separated by underscore as necessary to improve readability
  - variable names follow the same convention as function name.
- 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 reversed keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption

Page No.
Date

Thus _ class _ is better than clss.

- CONSTANTS
- constants are usally defined on a module level and written in all capital letters with underscores separating words.
- EG: MAX_OVERFIOW and TOTAL.