NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT.

PEP 8 in Python

What is PEP

The PEP is an abbreviation form of Python Enterprise Proposal. Writing code with proper logic is a key factor of programming, but many other important factors can be affect the code's quality. The developer's coding style makes the code much reliable, and every developer should keep in mind that Python strictly follows the way of order and format of the string.

Adaptive a nice coding style makes the code more readable. The code becomes easy for end-user.

PEP 8 is a document that provide various 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.

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

Why PEP 8 is Important

PEP 8 enhances the readability of the Python code. Creator of Python, Guido van Russum said: "Code is much more often than it is written." The code can be written in a few minutes, a few hours, or a whole day but once we have written the code, we will never rewrite it again But sometimes, we need to read the code again and again.

At this point, we must have an idea of why we wrote the particular line in the code. The code should reflect the meaning of each line. That why readability is so much important.

Below are few important guidelines for writing effective code that can be read by other as well.

NAMING CONVENTION.

When we write the code, we need to assign name to many things such as variable, functions, classes, packages, and a lot more things. Selecting a proper name will save time and energy. When we look back to the file after sometime, we can easily recall what a certain variable, function, or class

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

represents. Developers should avoid closing choosing inappropriate names. The naming convention in Python is slighty messy, but there are certain conventions that we can follow easily.

For Example,

Single lowercase letter

a = 10

Single uppercase letter

A = 10

Lowercase

var = 10

Uppercase

VAR = 10

Lower case with underscores

number_of_apple = 10

Upper case with underscores

NUMBER_OF_APPLE = 10

Capitalized Each Words

NumberOfBook = 10

NAME STYLE

Below is the table that specifies some of the common naming styles in Python. Consider the following table.

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

TYPE	NAMING CONVENTION	EXAMPLE
Function	We should use the lowercase words or separates words by the underscore.	myfunction, my_function
Variable	We should use a lowercase letter, words or separate words to enhance the readability.	a, var, variable_name
Class	The first letter of class name should be capitalized; use camel case. Do not separate words with underscore.	MyClass, Form, Model.
Method	We should use a lowercase letter, words, or separate words to enhance readability.	class_method, method.
Constant	We should use a ~~lowercase~~ shout uppercase letter, words, or separate words to enhance the readability.	MYCONSTANT, CONSTANT, MY_CONSTANT
Module	We should use a lowercase letter, words, or separate words to enhance the readability.	Module_name .py, module.py
Package	We should use a lowercase letter, words, or separate words to enhance the readability. Do not separate word with the underscore.	package, mypackage

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FY/IT

Above are some common naming conventions that are useful to beautify the Python code. For additional improvement, we should choose the name carefully.

CODE LAYOUT.

The Code Layout defines how much the code is readable. In this section, we will learn how to use whitespace to improve code readability.

INDENTATION :- Unlike other programming languages, the indentation is used to define the code block in Python. The indentitions are the important part of the Python programming language and it determines the level of lines of code. Generally, we use the 4 space for indentation. Let understand the following example.

For Example;

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

In the above example, the indented print statement will get executed if the condition of if statement is true.

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

This indentation defines the code block and tells us what statements execute when a function is called or condition trigger.
We can also use the TABS to provide the consecutive spaces to indicate, but whitespaces are the most preferable.
Python 2 allows the mixing of tab and spaces but we will get an error in Python 3.
INDENTATION FOLLOWING LINE BREAK:- It is essential to use indentation when using line continuations to keep the line to fewer than 79 characters. It provides the flexibility to determining between two lines of code and a single line of code that extends two lines.
For example,

CODE:
```
# Aligned with opening definition:
obj = func_name(argument_one, argument_two,
                argument_three, argument_four
```

We can use the following structure.

CODE:
```
def function_name (
    argument_one, argument_two, argument_three,
    argument_four ):
    print (argument_two)
for = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

USE DOCSTRINGS :- Python provides the two types of document strings or docstring - single line and multiple lines. We use the triple quotes to define a single line or multiline quotes. Basically, these are used to describe the function or particular program. Let's understand the following example.

For example,

CODE:
```
def add (a, b):
    """ This is simple add method """
    """ This is
    a
    simple add program to add
    the two numbers. """
```

SHOULD A LINE BREAK BEFORE OR AFTER A BINARY OPERATOR :- This lines break before or after a binary operation is a traditional approach. But it 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 previous line.

For Example,

CODE:
```
# Wrong:
# operators sit far away from their operands.
```

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

```
marks = (english_marks +
         math_marks +
         (science_marks + biology_marks)+.
         Physics_marks.
```

As we can see in the above example, it seems quite messy to read. We can solve such types of problem by using the following structure.

For Example,

CODE:
```
# Correct
# easy to match operators with operands.
Total_mark = (English_marks
        + math_marks
        + (science_marks + biology marks)
        + physics_marks.
```

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

IMPORTING MODULE:- We should import the modules in separate line as follows.

CODE:
```
import pygame
import os
import sys
```

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

We should not import the modules like this.

CODE: import os, sys

We can also use the following approach.

CODE: from subprocess import Popen, PIPE

The import statement should be return 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.

CODE: import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example

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

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

BLANK LINES :- Blank lines can be improved the readability of Python code. If many lines of code bunched together the code will become harder to read. We can remove this by using the many blank vertical lines, and the reader might need to scroll more than necessary. Follow the below instructions to add vertical whitespace.

- Top-level function and classes with two lines - Put the extra vertical space around them so that it can be understandable.

CODE:
```
class FirstClass:
    pass

class SecondClass
    pass

def main_function():

    return None
```

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

- Single blank line inside classes - The functions that we define in the class is related to one another. Let's see the following example -

CODE:
```
class FirstClass:
    def method_one(self):
        return None

    def second_two(self):
        return None
```

- Use blank lines inside the function - Sometimes, we need to write a complicated function has consists of several steps before the return statement. So we can add the blank line between each step. Let's understand the following example.

CODE:
```
def cal_variance(n_list):
    list_sum = 0
    for n in n_list:
        list_sum = list_sum + n
    mean = list_sum / len(n_list)
```

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

```
    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 whitespace to improve the readability of code.

PUT THE CLOSING BRACES.

We can break lines inside parentheses, brackets using the line continuations. PEP 8 allows us to use closing braces in implies line continuations. Let's understand the following example.

- LINE UP THE CLOSING BRACE WITH THE FIRST NON-WHITESPACE.

CODE:

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

NAME: KHOT YASH KISHOR, ROLL NO: 39, CLASS: FYIT

- LINE UP THE CLOSING BRACES WITH THE FIRST CHARACTER OF LINE.

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

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

COMMENTS.

Comments 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 comments anyone can able to understand the code. But we should remember the following points.

- Start with the capital letter, and write complete sentence.
- Update the comment in case of a change in code.
- Limit the line length of comments and docstrings to 72 characters.

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

BLOCK COMMENTS:- Block comments are the good choice for the small section of code. Such comments are useful when we write several line codes to peerform a single action such as iterating a loop. They help us to understand the purpose 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 the # followed by a single space.
- Separate line using the single #.

Let's see the following code.
For example,

CODE:
```
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.

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

INLINE COMMENTS :- Inline 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 inline comments.

- Start comments with # and single space.
- Use inline comments carefully.
- We should separate the inline comments on the same line as the statement they refer.

Following is the example of inline comments.

CODE: a = 10 # The a is variable that holds integer value.

Sometimes, we can use the naming convention to replace the inline comments.

CODE: x = "Peter Decosta" # This is a student name.

We can use the following name convention.

CODE: Student_name = 'Peter Decosta'

NAME: KHUT YASH KISHOR, ROLL NO: 39, CLASS: FYIT

Inline comments are essential but block comments make the code more readable.

AVOID UNNECESSARY ADDING WHITESPACES.

In some cases, use of whitespaces can make the code much harder to read. Too much whitespace can make code overly sparse and difficult to understand. We should avoid adding whitespaces at the end of a line. This is known as ~~nothing~~ trailing whitespaces
Let's see the following examples.

~~Ex~~ Example - 1

CODE:
```
# Recommended
list1 = [1,2,3]

# Not Recommended
list1 = [1, 2, 3]
```

Example - 3

CODE:
```
x = 5
y = 6

# Recommended
print(x,y)
```

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

```
#Not Recommended
print (x , y)
```

PROGRAMMING RECOMMENDATION.

As we know that, there are several methods to perform similar tasks in Python. In this section, we will see some of the suggestions of PEP 8 to improve the consistency.

AVOID COMPARING BOOLEAN VALUES USING THE EQUIVALENCE OPERATOR :-

CODE:

```
#Not Recommended
bool_value = 10 > 5
if bool_value == True
    return '10 is bigger than 5'
```

We shouldn't use the equivalence operator == to compare the boolean values. It can only take the True or False. Let's see the following example.

For Example,

CODE:

```
#Recommended
if my_bool:
    return '10 is bigger than 5'
```

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

The approach is simple that's why PEP 8 encourages it.

EMPTY SEQUENCES ARE FALSE IN IF STATEMENTS:
If we want to check whether a given list is empty, we might need to check the length of list, so we need to avoid the following approach.

CODE:
```
# Not Recommended
list1 = []
if not len(list1):
    print('List is empty!')
```

However, if there is an empty list, set or tuple. We can use the following way to check.

CODE:
```
# Recommended
list1 = []
if not list1:
    print ('List is Empty!')
```

The Second method is more appropriate; that why PEP 8 encourages it.

NAME: KHOT YASH KISHOR, ROLL NO: 39, CLASS: FYIT

DON'T USE NOT IS IN IF STATEMENT:-

There are two options to check whether a variable has a defined value. The first option is with x is not NONE, as in the following example.

CODE:
```
# Recommended
if x is not None:
    return 'x exists!'
```

A second option is to evaluate x is None and if statement is based on not the ~~same~~ outcome.

CODE:
```
# Not Recommended.
if not x is None:
    return 'x exists!'
```

Both options are correct but the first one is simple. so PEP8 ~~err~~ encourages it.

USE STRING METHODS INSTEAD OF STRING MODULE:-

String Methods are always much faster and share the same API with unicode strings. Overside this rule if backward compatibility with Python

NAME: KHOT YASH KISHOR, ROLL NO: 39, CLASS: FYIT

older than 2.0 is required.
Use ".startswith() and ".endswith() instead of string slicing to check for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone.
For example,

CODE:
```
#Not Recommended
if foo[:3] == 'bar':
```

A second Method is use startwith() for less error.

CODE:
```
# Recommended
if foo.startwith('bar'):
```

OBJECT TYPE COMPARISONS :- Object Type Comparisons should always use instance() instead of comparing types directly.
For Example,

CODE:
```
# Not Recommended
if type(obj) is type(1):
```

NAME : KHUT. YASH. KISHOR, ROLL NO : 39, CLASS : FYIT

A second method is to use isinstance() instead of comparing types directly:

CODE:
```
# Recommended
if isinstance(obj, int):
```

When checking if an object is a string, keep in mind that it might be a unicode string too!
In python 2, str and unicode have a common base class, basestring, so you can do

CODE:
```
if isinstance(obj, basestring).
```

Note that in Python 3, unicode and basestring no longer exist (there is only str) and a bytes object is no longer a kind of a string (it is a sequence of integers instead)

For sequences, (strings, list, tuples), use the fact that empty sequences are false:

CODE:
```
# RECommended
if not seq:
if seq:
```

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: T.Y.I.T

CODE:
```
# Not Recommended.
if len(seq):
if not len(seq):
```

FUNCTION ANNOTATIONS

With the acceptance of PEP 484, the style rules for function annotations are changing.

In order to be forward compatible, function annotations in Python 3 code should preferably use PEP 484 syntax. (These are some formatting recommendations for annotations.)

The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.

However, outside the stdlib, experiments within the rules of PEP 484 are now encouraged. For example, making up a large third party library of application with PEP 484 style type annotations, reviewing how easy it was to add those annotations, and observing whether their presence increases code understandability.

NAME: KHOT. YASH. KHOT, ROLL NO: 39, CLASS: FYIT.

The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.

For code that wants to make a different use of function annotations it is recommended to put a comment of the form:

CODE: # type : ignore

near the top of the file : this tells type checker to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in PEP484)

Like linters, type checkers are optional, separate tools. Python interpreters by default should not issue any messages due to type checking and should not alter their behaviour based on annotations.

Users who don't want to use type checkers are free to ignore them. However it is expected that users of third

NAME: KHOT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

party library packages may want to run type checkers over those packages. For this purpose PEP 484 recommends the use of stub files: .pyi files that are read by the type checker in preference of the corresponding .py files. Stub files can be distributed with a library, or separately (with the library author's permission) through the typeshed repo.

For code that needs to be backwards compatible, type annotations can be added in the form of comments.

AUTOFORMATTING YOUR PYTHON CODE

Formatting won't be a problem when you are working with small problems. But just imagine having to ~~focus~~ follow the correct formatting rules for a complex program running into thousands of lines! This will definitely a task to achieve.

Autoformatter is a program that identifies formatting errors and fixes them in place. Black is one such autoformatter that takes the burden off your shoulder

NAME: KHUT. YASH. KISHOR, ROLL NO: 39, CLASS: FYIT

by automatically formatting your Python code to one that conforms to the PEP 8 style of coding.

CONCLUSION.

We have discussed the PEP 8 guidelines to make the code remove ambiguity and enhance readability. These guidelines improve the code especially when sharing the code with potential employees or collaborators.

We have discussed PEP and ~~today~~ what it is and why it was, how to write code that is PEP 8 compliant.