

INTRODUCTION

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

Why PEP 8 is Important?

PEP 8 enhances the readability of the Python code, but why is readability so important? Let's understand this concept.

Creator of Python, Guido van Rossum 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's why readability is so much important.

Syntax and semantics

Python is meant to be an easily readable language. Its formatting is visually uncluttered, and often uses English keywords where other languages use punctuation. Unlike many other languages, it does not use curly brackets to delimit blocks, and semicolons after statements are allowed but rarely used. It has fewer syntactic exceptions and special cases than C or Pascal.[7]

Naming Conventions

When you write Python code, you have to name a lot of things: variables, functions, classes, packages, and so on. Choosing sensible names will save you time and energy later. You'll be able to figure out, from the name, what a certain variable, function, or class represents. You'll also avoid using inappropriate names that might result in errors that are difficult to debug.

Naming Styles

The table below outlines some of the common naming styles in Python code and when you should use them:

Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	function, my_function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words	Model, MyClass

	with underscores. This style is called camel case.	
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	class_method, method
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	module.py, my_module.py
Package	Use a short, lowercase word or words. Do not separate words with underscores.	package, mypackag “Beautiful is better than ugly.” – The Zen of Python e

These are some of the common naming conventions and examples of how to use them. But in order to write readable code, you still have to be careful with your choice of letters and words. In addition to choosing the correct naming styles in your code, you also have to choose the names carefully. Below are a few pointers on how to do this as effectively as possible.

How to write the code name

Choosing names for your variables, functions, classes, and so forth can be challenging. You should put a fair amount of thought into your naming choices when writing code as it will make your code more readable. The best way to name your objects in Python is to use descriptive names to make it clear what the object represents.

When naming variables, you may be tempted to choose simple, single-letter lowercase names, like `x`. But, unless you’re using `x` as the argument of a mathematical function, it’s not clear what `x` represents. Imagine you are storing a person’s name as a string, and you want to use string slicing to format their name differently. You could end up with something like this:

```
>>>>> # Not recommended
>>> x = 'John Smith'
>>> y, z = x.split()
>>> print(z, y, sep=', ')
'Smith, John'
```

This will work, but you’ll have to keep track of what `x`, `y`, and `z` represent. It may also be confusing for collaborators. A much clearer choice of names would be something like this:

```
>>>>> # Recommended
>>> name = 'John Smith'
>>> first_name, last_name = name.split()
>>> print(last_name, first_name, sep=', ')
'Smith, John'
```

Similarly, to reduce the amount of typing you do, it can be tempting to use abbreviations when choosing names. In the example below, I have defined a function `db()` that takes a single argument `x` and doubles it:

```
# Not recommended
def db(x):
    return x * 2
```

At first glance, this could seem like a sensible choice. `db()` could easily be an abbreviation for `double`. But imagine coming back to this code in a few days. You may have forgotten what you were trying to achieve with this function, and that would make guessing how you abbreviated it difficult.

The following example is much clearer. If you come back to this code a couple of days after writing it, you'll still be able to read and understand the purpose of this function:

```
# Recommended
def multiply_by_two(x):
    return x * 2
```

The same philosophy applies to all other data types and objects in Python. Always try to use the most concise but descriptive names possible.

Code Layout

How you lay out your code has a huge role in how readable it is. In this section, you'll learn how to add vertical whitespace to improve the readability of your code. You'll also learn how to handle the 79 character line limit recommended in PEP 8.

Blank Lines

Vertical whitespace, or blank lines, can greatly improve the readability of your code. Code that's bunched up together can be overwhelming and hard to read. Similarly, too many blank lines in your code makes it look very sparse, and the reader might need to scroll more than necessary. Below are three key guidelines on how to use vertical whitespace.

Surround top-level functions and classes with two blank lines. Top-level functions and classes should be fairly self-contained and handle separate functionality. It makes sense to put extra vertical space around them, so that it's clear they are separate:

```
class MyFirstClass:  
    pass
```

```
class MySecondClass:  
    pass
```

```
def top_level_function():  
    return None
```

Surround method definitions inside classes with a single blank line. Inside a class, functions are all related to one another. It's good practice to leave only a single line between them:

```
class MyClass:  
    def first_method(self):  
        return None  
  
    def second_method(self):  
        return None
```

Use blank lines sparingly inside functions to show clear steps. Sometimes, a complicated function has to complete several steps before the return statement. To help the reader understand the logic inside the function, it can be helpful to leave a blank line between each step.

In the example below, there is a function to calculate the variance of a list. This is two-step problem, so I have indicated each step by leaving a blank line between them. There is also a blank line before the return statement. This helps the reader clearly see what's returned:

```
def calculate_variance(number_list):  
    sum_list = 0  
    for number in number_list:  
        sum_list = sum_list + number  
    mean = sum_list / len(number_list)  
  
    sum_squares = 0  
    for number in number_list:  
        sum_squares = sum_squares + number**2  
    mean_squares = sum_squares / len(number_list)  
  
    return mean_squares - mean**2
```

If you use vertical whitespace carefully, it can greatly improved the readability of your code. It helps the reader visually understand how your code splits up into sections, and how those sections relate to one another.

Maximum Line Length and Line Breaking

PEP 8 suggests lines should be limited to 79 characters. This is because it allows you to have multiple files open next to one another, while also avoiding line wrapping.

Of course, keeping statements to 79 characters or less is not always possible. PEP 8 outlines ways to allow statements to run over several lines.

Python will assume line continuation if code is contained within parentheses, brackets, or braces:

```
def function(arg_one, arg_two,
            arg_three, arg_four):
    return arg_one
```

If it is impossible to use implied continuation, then you can use backslashes to break lines instead:

```
from mypkg import example1, \
    example2, example3
```

However, if you can use implied continuation, then you should do so.

If line breaking needs to occur around binary operators, like + and *, it should occur before the operator. This rule stems from mathematics. Mathematicians agree that breaking before binary operators improves readability. Compare the following two examples.

Below is an example of breaking before a binary operator:

```
# Recommended
total = (first_variable
        + second_variable
        - third_variable)
```

You can immediately see which variable is being added or subtracted, as the operator is right next to the variable being operated on.

Now, let's see an example of breaking after a binary operator:

```
# Not Recommended
total = (first_variable +
        second_variable -
```

```
third_variable)
```

Here, it's harder to see which variable is being added and which is subtracted.

Breaking before binary operators produces more readable code, so PEP 8 encourages it. Code that consistently breaks after a binary operator is still PEP 8 compliant. However, you're encouraged to break before a binary operator.

Indentation

Indentation, or leading whitespace, is extremely important in Python. The indentation level of lines of code in Python determines how statements are grouped together.

Consider the following example:

```
x = 3
if x > 5:
    print('x is larger than 5')
```

The indented print statement lets Python know that it should only be executed if the if statement returns True. The same indentation applies to tell Python what code to execute when a function is called or what code belongs to a given class.

The key indentation rules laid out by PEP 8 are the following:

Use 4 consecutive spaces to indicate indentation.

Prefer spaces over tabs.

Tabs vs. Spaces

As mentioned above, you should use spaces instead of tabs when indenting code. You can adjust the settings in your text editor to output 4 spaces instead of a tab character, when you press the Tab key.

If you're using Python 2 and have used a mixture of tabs and spaces to indent your code, you won't see errors when trying to run it. To help you to check consistency, you can add a -t flag when running Python 2 code from the command line. The interpreter will issue warnings when you are inconsistent with your use of tabs and spaces:

```
$ python2 -t code.py
code.py: inconsistent use of tabs and spaces in indentation
```

If, instead, you use the -tt flag, the interpreter will issue errors instead of warnings, and your code will not run. The benefit of using this method is that the interpreter tells you where the inconsistencies are:

```
$ python2 -tt code.py
File "code.py", line 3
    print(i, j)
        ^
```

TabError: inconsistent use of tabs and spaces in indentation

Python 3 does not allow mixing of tabs and spaces. Therefore, if you are using Python 3, then these errors are issued automatically:

```
$ python3 code.py
File "code.py", line 3
    print(i, j)
        ^
```

TabError: inconsistent use of tabs and spaces in indentation

You can write Python code with either tabs or spaces indicating indentation. But, if you're using Python 3, you must be consistent with your choice. Otherwise, your code will not run. PEP 8 recommends that you always use 4 consecutive spaces to indicate indentation.

Indentation Following Line Breaks

When you're using line continuations to keep lines to under 79 characters, it is useful to use indentation to improve readability. It allows the reader to distinguish between two lines of code and a single line of code that spans two lines. There are two styles of indentation you can use.

The first of these is to align the indented block with the opening delimiter:

```
def function(arg_one, arg_two,
            arg_three, arg_four):
    return arg_one
```

Sometimes you can find that only 4 spaces are needed to align with the opening delimiter. This will often occur in if statements that span multiple lines as the if, space, and opening bracket make up 4 characters. In this case, it can be difficult to determine where the nested code block inside the if statement begins:

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

In this case, PEP 8 provides two alternatives to help improve readability:

Add a comment after the final condition. Due to syntax highlighting in most editors, this will separate the conditions from the nested code:

```
x = 5
if (x > 3 and
    x < 10):
    # Both conditions satisfied
    print(x)
```

Add extra indentation on the line continuation:

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

An alternative style of indentation following a line break is a hanging indent. This is a typographical term meaning that every line but the first in a paragraph or statement is indented. You can use a hanging indent to visually represent a continuation of a line of code. Here's an example:

```
var = function(
    arg_one, arg_two,
    arg_three, arg_four)
```

Note: When you're using a hanging indent, there must not be any arguments on the first line. The following example is not PEP 8 compliant:

```
# Not Recommended
var = function(arg_one, arg_two,
    arg_three, arg_four)
```

When using a hanging indent, add extra indentation to distinguish the continued line from code contained inside the function. The following example is difficult to read because the code inside the function is at the same indentation level as the continued lines:

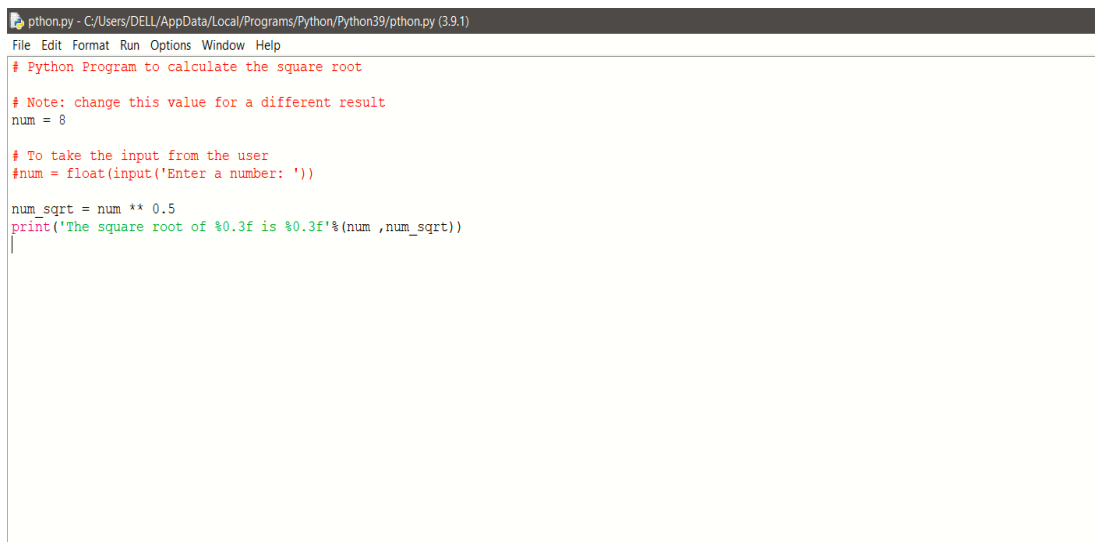
```
# Not Recommended
def function(
    arg_one, arg_two,
    arg_three, arg_four):
    return arg_one
```


Instead, it's better to use a double indent on the line continuation. This helps you to distinguish between function arguments and the function body, improving readability:

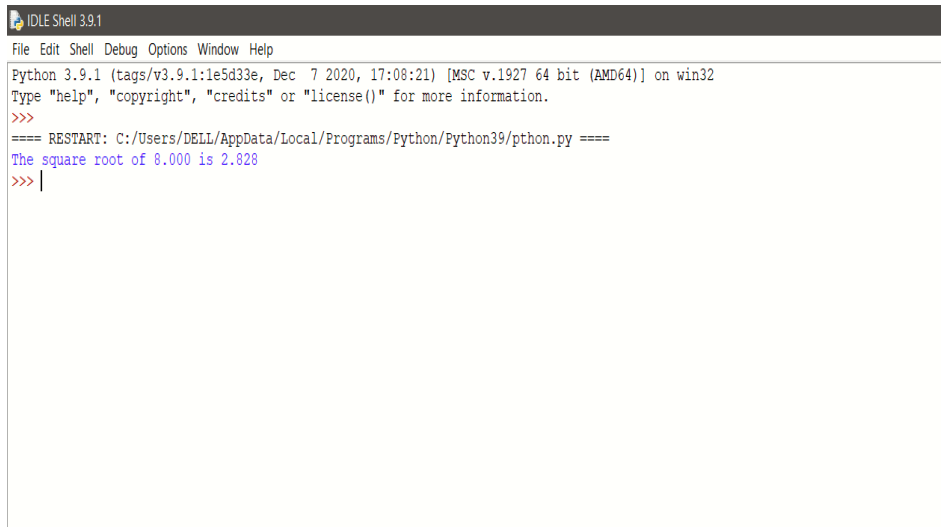
```
def function(  
    arg_one, arg_two,  
    arg_three, arg_four):  
    return arg_one
```

When you write PEP 8 compliant code, the 79 character line limit forces you to add line breaks in your code. To improve readability, you should indent a continued line to show that it is a continued line. There are two ways of doing this. The first is to align the indented block with the opening delimiter. The second is to use a hanging indent. You are free to choose which method of indentation you use following a line break.

Code



```
pthon.py - C:/Users/DELL/AppData/Local/Programs/Python/Python39/pthon.py (3.9.1)  
File Edit Format Run Options Window Help  
# Python Program to calculate the square root  
  
# Note: change this value for a different result  
num = 8  
  
# To take the input from the user  
#num = float(input('Enter a number: '))  
  
num_sqrt = num ** 0.5  
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))  
|
```



```
IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python39/pthon.py ====
The square root of 8.000 is 2.828
>>> |
```

Thank you