PEP-8

PEP-8 là một chuẩn code dùng trong ngôn ngữ lập trình python. Chúng được viết vào năm 2001 bởi Guido van Rossum, Barry Warsaw và Nick Coghlan. PEP 8 là tài liệu dành cho lập trình viên mới bắt đầu hoặc trình độ trung cấp mà đây không phải topic cho lập trình viên cao cấp. Bạn có thể đọc cụ thể ở Đây, bài viết chỉ tóm tắt các hướng dẫn trong PEP 8.

1. Naming Conventions (Quy tắc đặt tên)

Khi lập trình, chúng ta sẽ có rất nhiều thứ phải đặt tên như: tên biến, tên hàm, tên lớp, tên phương pháp, … Đặt tên tốt sẽ tạo cho người nhìn một cảm giác thoải mái, lập trình tốt hơn và dễ bảo trì, debug.\

Lưu ý: Không bao giờ sử dụng tên đơn, l, O hoặc I vì chúng có thể bị nhầm thành 1 và 0, tùy thuộc vào kiểu chữ:

O = 2 # This may look like you're trying to reassign 2 to zero

Phong cách đặt tên

Dưới đây là một số kiểu đặt tên phổ biến trong python:

Loại	Quy tắc đặt tên	Ví dụ
Hàm số	Sử dụng chữ cái thường. Phân tách các từ bởi dấu gạch dưới.	function, my_function
Biến	Sử dụng chữ cái thường. Có thể sử dụng một kí tự, một từ hoặc nhiều từ. Phân tách các từ bởi dấu gạch dưới.	x, var, my_variable
Lớp	Chữ cái đầu của mỗi từ là in hoa. Không phân tách các từ với nhau bằng dấu gạch dưới. Cách viết này được gọi là camel case.	Model, MyClass
Phương pháp	Sử dụng chữ cái thường. Phân tách các từ bởi dấu gạch dưới.	class_method, method
Hằng số	Sử dụng chữ in hoa. Có thể sử dụng một chữ cái, một từ hoặc nhiều từ. Phân tách các từ bởi dấu gạch dưới.	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT
Mô-đun	Viết ngắn gọn, sử dụng chữ thường. Phân tách các từ bởi dấu gạch dưới.	module.py, my_module.py
Gói	Viết ngắn gọn sử dụng chữ thường. Không phân tách các từ bởi dấu gạch dưới.	package, mypackage

Chọn tên thế nào cho phù hợp

Đặt tên khi lập trình thực tế lại là vấn đề rất thách thức. Thường chúng ta mất rất nhiều thời gian để đặt tên sao cho chuẩn và khoa học. Cách tốt nhất để đặt tên biến cho đối tượng đó là mô tả tên mà đối tượng đó thể hiện.

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



'Smith, John'
```

Thay x, y và z bằng tên biến rõ ràng hơn. Ví dụ như:

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



'Smith, John'
```

Tương tự như sau:

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

Khi lập trình ta cố gắng tối giản tên nhất có thể nhưng trong trường hợp sau ta không hiểu hàm số nhằm mục đích gì khi chỉ nhìn vào tên hàm.

Tại thời điểm viết ta có thể hiểu nó là double nhưng sau một thời gian đọc lại thì rất là khó đoán đó là hàm gì. Ta có thể sử dụng tên hàm sau dễ hiểu hơn.

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

Các đối tượng khác như lớp, hằng số, gói, mô-đun,… cũng được ứng dụng tương tự. Cố gắng sử dụng tên ngắn gọn nhưng vẫn mô tả đối tượng tốt nhất có thể.

2. Code Layout (Giao diện code)

“Đẹp thì tốt hơn là xấu.”

The Zen of Python.

Một giao diện code đẹp có vai trò rất quan trọng đối với khả năng đọc code của bạn. Chúng ta thêm khoảng trắng để cải thiện khả năng đọc code của bạn. Trong PEP 8, một dòng được đề xuất giới hạn bởi 79 kí tự.

Blank lines (Dòng Trống)

Khoảng trắng hoặc dòng trống có thể cải thiện đáng kể khả năng đọc code của bạn. Code mà chụm lại với nhau có thể rất khó đọc. Tương tự, quá nhiều dòng trống trong code của bạn làm cho nó trông rất thưa thớt và người đọc có thể cần phải cuộn màn hình nhiều hơn cần thiết. Dưới đây là ba hướng dẫn chính về cách sử dụng khoảng trắng.

Bao quanh các hàm và lớp cấp cao nhất với hai dòng trống

Các lớp và hàm cấp cao nhất phải khá khép kín và xử lý các chức năng riêng biệt. Cần đặt thêm không gian theo chiều dọc xung quanh chúng để nó rõ ràng chúng tách biệt nhau:

```
class MyFirstClass:
    pass


class MySecondClass:
    pass


def top_level_function():
    return None
```

Phương pháp bao quanh yêu cầu bên trong các lớp cần một dòng trống duy nhất

Trong một lớp, các hàm đều liên quan đến nhau. Nó được thực hiện tốt khi chỉ để lại một dòng duy nhất giữa chúng:

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

    def second_method(self):
        return None
```

Sử dụng các dòng trống một cách tiết kiệm bên trong các hàm để hiển thị các bước rõ ràng

Đôi khi, một hàm phức tạp phải hoàn thành một số bước trước câu lệnh return. Để giúp người đọc hiểu logic bên trong hàm, nên để một dòng trống giữa mỗi bước.

```
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
```

Nếu bạn sử dụng khoảng trắng một cách cẩn thận, nó có thể cải thiện đáng kể khả năng đọc code của bạn. Nó giúp người đọc hiểu một cách trực quan cách code của bạn chia thành các phần và cách các phần đó liên quan đến nhau.

Tối đa hóa độ dài dòng và ngắt dòng

PEP 8 gợi ý các dòng nên được giới hạn ở 79 ký tự. Điều này là do nó cho phép bạn có nhiều tệp được mở cạnh nhau để quan sát, đồng thời tránh việc ngắt dòng.

Tất nhiên, việc giữ các câu lệnh có 79 ký tự trở xuống không phải lúc nào cũng có thể. PEP 8 phác thảo các cách để cho phép các câu lệnh chạy trên một số dòng.

Python sẽ tiếp tục dòng nếu code được chứa trong dấu ngoặc đơn, ngoặc hoặc dấu ngoặc:

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

Bạn có thể sử dụng dấu gạch chéo để ngắt dòng:

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

Nếu cần ngắt dòng xung quanh các toán tử nhị phân, như + và *, thì nó sẽ xảy ra trước toán tử. Quy tắc này bắt nguồn từ toán học. Các nhà toán học đồng ý rằng việc phá vỡ trước các toán tử nhị phân để cải thiện khả năng đọc. So sánh hai ví dụ sau.

Dưới đây là một ví dụ về việc ngắt dòng trước một toán tử nhị phân:

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

Bạn có thể thấy ngay biến nào đang được thêm hoặc bớt, vì toán tử nằm ngay bên cạnh biến đang được sử dụng.

Bây giờ, hãy để xem một ví dụ về việc ngắt dòng sau một toán tử nhị phân:

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

Ở đây, muốn xem biến nào đang được thêm và biến nào được bớt đi sẽ khó hơn. Phá vỡ trước khi các toán tử nhị phân tạo ra code dễ đọc hơn, vì vậy PEP 8 khuyến khích nó.

3. Identation (Thụt lề)

"Nên có một cách khác mà tốt nhất là một cách duy nhất rõ ràng để thực hiện"

The Zen of Python

Sự thụt lề, hoặc khoảng trắng hàng đầu, cực kỳ quan trọng trong Python. Mức độ thụt dòng của code trong Python xác định cách các câu lệnh được nhóm lại với nhau.

Xem xét ví dụ dưới đây:

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

Câu lệnh in được thụt lề cho Python biết rằng nó chỉ nên được thực thi nếu câu lệnh *If* trả về True. Việc thụt lề tương tự áp dụng để cho Python biết code nào sẽ được thực hiện khi hàm được gọi hoặc code nào thuộc về một lớp nhất định.

Các quy tắc lùi lề chính được đặt ra bởi PEP 8 như sau:

- Sử dụng 4 khoảng trắng liên tiếp để chỉ ra vị trí thụt lề.
- Ưu tiên khoảng trống hơn các tab.

Tab & Khoảng trống

Như đã đề cập ở trên, bạn nên sử dụng khoảng trống thay vì các tab khi thụt lề. Bạn có thể điều chỉnh các cài đặt trong trình soạn thảo văn bản của mình để xuất 4 khoảng trống thay vì ký tự tab, khi bạn nhấn phím Tab.

Nếu bạn sử dụng Python 2 và đã sử dụng hỗn hợp các tab và khoảng trống để thụt lề code của mình, bạn sẽ không gặp lỗi khi chạy nó. Để giúp bạn kiểm tra tính nhất quán, bạn có thể thêm flag -t khi chạy mã Python 2 từ dòng lệnh. Trình biên dịch sẽ đưa ra cảnh báo khi bạn không nhất quán với việc sử dụng các tab và dấu cách:

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

Thay vào đó, nếu bạn sử dụng flag -tt, trình biên dịch sẽ đưa ra lỗi thay vì cảnh báo và code của bạn sẽ không chạy. Lợi ích của việc sử dụng phương pháp này là trình biên dịch cho bạn biết sự không nhất quán ở đâu:

```
$ python2 -tt code.py
  File "code.py", line 3
    print(i, j)
              ^
TabError: inconsistent use of tabs and spaces in indentation
```

Python 3 không cho phép trộn các tab và khoảng trống. Do đó, nếu bạn đang sử dụng Python 3, thì các lỗi này sẽ tự động báo:

```
$ python3 code.py
  File "code.py", line 3
    print(i, j)
               ^
TabError: inconsistent use of tabs and spaces in indentation
```

Bạn có thể viết mã Python bằng các tab hoặc khoảng trống biểu thị thụt lề. Nhưng, nếu bạn sử dụng Python 3, bạn phải nhất quán với lựa chọn của mình. Nếu không, code của bạn sẽ không chạy. PEP 8 khuyến nghị bạn luôn sử dụng 4 khoảng trống liên tiếp để biểu thị thụt lề.

Thụt lề theo sự ngắt dòng

Khi bạn sử dụng các phần tiếp theo dòng để giữ các dòng dưới 79 ký tự, sẽ rất hữu ích khi sử dụng thụt lề để cải thiện khả năng đọc. Nó cho phép người đọc phân biệt giữa hai dòng code và một dòng code kéo dài hai dòng.

Có hai kiểu thụt đầu dòng bạn có thể sử dụng. Đầu tiên là căn chỉnh khối thụt lề với dấu phân cách mở:

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

Đôi khi bạn có thể thấy rằng chỉ cần 4 khoảng trống để căn chỉnh với dấu phân cách mở. Điều này thường xảy ra trong các câu lệnh *If* trải dài trên nhiều dòng như if, dấu cách và dấu ngoặc mở tạo thành 4 ký tự. Trong trường hợp này, có thể khó xác định nơi khối code lồng nhau bên trong câu lệnh *If* bắt đầu:

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

Trong trường hợp này, PEP 8 cung cấp hai lựa chọn thay thế để giúp cải thiện khả năng đọc:

- Thêm một bình luận sau điều kiện cuối cùng. Việc tô sáng cú pháp trong hầu hết các trình soạn thảo sẽ tách các điều kiện khỏi code lồng nhau:

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

- Thêm thao tác thụt lề trên dòng lệnh tiếp theo.

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

Một kiểu thay thế thụt lề đầu dòng sau ngắt dòng là **hanging indent** (thụt lề treo). Đây là một thuật ngữ đánh máy có nghĩa là trong mọi dòng nhưng đầu đoạn hoặc câu được thụt lề. Bạn có thể sử dụng thụt lề treo để thể hiện trực quan sự tiếp nối của một dòng code. Dưới đây là một ví dụ:

```
var = function(
    arg_one, arg_two,
```

```
    arg_three, arg_four)
```

Lưu ý: Khi bạn sử dụng thụt lề treo, không được có bất kỳ đối số nào trên dòng đầu tiên. Ví dụ sau không tuân thủ PEP 8:

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

Khi sử dụng thụt lề treo, thêm thụt đầu dòng để phân biệt dòng tiếp tục với code có trong hàm. Ví dụ sau rất khó đọc vì code bên trong hàm ở cùng mức thụt lề như dòng tiếp theo:

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

Thay vào đó, tốt hơn là sử dụng một thụt lề kép trên dòng tiếp theo. Điều này giúp bạn phân biệt giữa các đối số hàm và thành phần hàm, cải thiện khả năng đọc:

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

Khi bạn viết mã tuân thủ PEP 8, giới hạn dòng 79 ký tự buộc bạn phải thêm ngắt dòng trong code của mình. Để cải thiện khả năng đọc, bạn nên thụt dòng tiếp tục để cho thấy rằng đó là dòng tiếp tục. Có hai cách để làm điều này. Cách đầu tiên là căn chỉnh khối thụt lề với dấu phân cách mở. Thứ hai là sử dụng một thụt lề treo. Bạn có thể tự do lựa chọn phương pháp thụt đầu dòng nào bạn sử dụng sau khi ngắt dòng.

Nên đặt dấu đóng ngoặc ở đâu ?

Dòng tiếp tục cho phép bạn ngắt các dòng bên trong dấu ngoặc đơn, ngoặc hoặc, dấu ngoặc. Rất dễ quên dấu đóng ngoặc nhưng điều quan trọng là phải đặt nó ở đâu đó hợp lý. Nếu không, nó có thể gây nhầm lẫn cho người đọc. PEP 8 cung cấp hai tùy chọn cho vị trí của dấu đóng trong các dòng tiếp theo:

- Sắp xếp dấu đóng ngoặc với *ký tự không phải khoảng trắng đầu tiên* của dòng trước:

```
list_of_numbers = [
    1, 2, 3,
```

```
    4, 5, 6,
    7, 8, 9
    ]
```

- Sắp xếp dấu đóng ngoặc với kí tự đầu tiên của dòng bắt đầu bằng cấu trúc:

```
list_of_numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]
```

Bạn có thể tự do chọn cách đặt dấu đóng ngoặc. Nhưng, như mọi khi, tính nhất quán là chìa khóa, vì vậy hãy cố gắng tuân theo một trong những phương pháp trên.

4. Comments (Bình luận)

"Sẽ rất tệ nếu bạn không thể giải thích được code của bạn"

The Zen of Python

Bạn nên viết bình luận cho các tài liệu code của bạn khi viết chúng. Việc đưa dẫn chứng cho bài code của bạn rất quan trọng, điều này giúp không chỉ bạn mà bất kỳ cộng sự nào của bạn có thể hiểu nó. Khi bạn hay ai đó đọc một lời bình, họ sẽ dễ dàng hiểu được đoạn code mà bạn bổ sung lời bình luận và cách mà lời bình đó hỗ trợ phần còn lại của đoạn code.

Dưới đây là một vài điểm lưu ý chính cần nhớ khi thêm bình luận vào đoạn code của bạn:

- Giới hạn độ dài của dòng bình luận và chuỗi văn bản tối đa 72 kí tự.
- Sử dụng một câu hoàn chỉnh, bắt đầu với chữ viết hoa.
- Hãy chắc chắn là bạn sẽ cập nhật lời bình khi bạn thay đổi đoạn code của mình.

Bình luận khối

Sử dụng bình luận khối cho một phần nhỏ của đoạn code. Chúng có hữu ích khi bạn phải viết nhiều dòng code để diễn tả một công việc đơn lẻ như nhập dữ liệu từ một tập tin hay cập nhật một mục cơ sở dữ liệu. Chúng rất quan trọng vì chúng giúp người khác hiểu mục đích và chức năng của một khối code nhất định.

PEP8 cung cấp một số quy tắc của việc viết khối bình luận dưới đây:

- Thụt lề của khối bình luận cùng mức với đoạn code mà nó miêu tả.
- Bắt đầu mỗi dòng với kí tự #, theo sau nó là một dấu cách đơn.
- Tách các đoạn trong khối bình luận bằng một dòng chỉ chứa kí tự #.

Dưới đây là một đoạn khối giải thích chức năng của vòng lặp for. Chú ý rằng kết thúc câu bắt đầu một dòng mới để duy trì giới hạn 79 kí tự của một dòng:

```
for i in range(0, 10):
    # Loop over i ten times and print out the value of i, followed by a
    # new line character
    print(i, '\n')
```

Đôi khi, nếu đoạn code rất khó hiểu thì sẽ cần nhiều hơn một đoạn trong một khối bình luận:

```
def quadratic(a, b, c, x):
    # Calculate the solution to a quadratic equation using the quadratic
    # formula.
    #
    # There are always two solutions to a quadratic equation, x_1 and
x_2.
    x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
    x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)
    return x_1, x_2
```

Nếu bạn phân vân loại bình luận nào phù hợp thì khối bình luận sẽ hữu ích với bạn. Sử dụng nhiều nhất có thể trong đoạn code của bạn, nhưng hãy chắc chắn cập nhật chúng khi bạn sửa đổi đoạn code của mình

Inline comments (Bình luận trên cùng một dòng)

Inline comment (Bình luận trên cùng một dòng) giải thích một câu lệnh trong một đoạn code. Chúng rất có hữu ích trong việc gợi nhắc cho bạn hay giải thích cho người khác, tại sao một dòng code nhất định là cần thiết. Đây là những gì PEP 8 phải nói về chúng:

- Sử dụng *inline comment* một cách hạn chế.
- Viết *inline comment* trên cùng dòng với câu lệnh mà chúng nhắm tới.
- Tách *inline comment* với câu lệnh bằng hai hay nhiều dấu cách.
- Bắt đầu *inline comment* với dấu # và một dấu cách, như *khối bình luận.*
- Đừng sử dụng chúng để giải thích những điều hiển nhiên.

Dưới đây là một ví dụ về *inline comment:*

```
x = 5  # This is an inline comment
```

Đôi lúc, *inline comment* có thể xem là cần thiết, nhưng bạn nên sử dụng quy ước đặt tên để thay thế. Đây là một ví dụ:

```
x = 'John Smith'  # Student Name
```

Ở đây, *inline comment* đã cung cấp thêm thông tin chi tiết. Tuy nhiên sử dụng *x* như một biến tên cho tên một người là việc làm thiếu không ngoan. Không cần *inline comment* nếu bạn có thể đổi tên biến:

```
student_name = 'John Smith'
```

Cuối cùng, *inline comment* như dưới đây thật là ngớ ngẩn vì chúng giải thích một sự thật hiển nhiên và khiến cho đoạn code trở nên lộn xộn:

```
empty_list = []  # Initialize empty list


x = 5
x = x * 5  # Multiply x by 5
```

Inline comment chuyên biệt hơn *khối bình luận,* rất dễ dàng để thêm *inline comment* nhưng khi không cần thiết thì chúng dẫn đến sự lộn xộn cho đoạn code. Bạn có thể thoát khỏi việc chỉ sử dụng *khối bình luận* trừ khi bạn chắc chắn là bạn cần một *inline comment* nếu không bạn cứ dính với *khối bình luận* và tuân thủ các lưu ý của PEP 8.

Chuỗi tài liệu

Chuỗi tài liệu hay chuỗi văn bản là chuỗi nằm trong nháy đúp " " hay nháy đơn ' ' xuất hiện trên dòng đầu tiên của bất kỳ hàm, lớp, phương thức, mô-đun. Bạn có thể sử dụng chúng để giải thích và đưa dẫn chứng một khối code đặc biệt. Có một PEP trọn vẹn, PEP 257, bao gồm các chuỗi văn bản, nhưng bạn sẽ hiểu được một cách tóm tắt trong phần này.

Nguyên tắc quan trọng nhất ứng dụng vào chuỗi văn bản được chỉ ra dưới đây:

- Bao quanh chuỗi văn bản là ba dấu nháy đôi ở hai bên, ví dụ " " " This is a docstring " " "
- Viết chúng cho tất cả loại mô-đun, hàm, lớp và phương thức.
- Đặt dấu " " " khi kết thúc một chuỗi văn bản đa dòng trên một dòng riêng sau đó:

```
def quadratic(a, b, c, x):
    """Solve quadratic equation via the quadratic formula.

    A quadratic equation has the following form:
    ax**2 + bx + c = 0

    There always two solutions to a quadratic equation: x_1 & x_2.
    """
    x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
    x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)
```

```
    return x_1, x_2
```

- Đối với loại chuỗi văn bản một dòng, giữ dấu “ “ “ trên cùng dòng đó:

```
def quadratic(a, b, c, x):
    """Use the quadratic formula"""
    x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
    x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)

    return x_1, x_2
```

5. Khoảng trắng trong biểu thức và câu lệnh

“Thưa thì tốt hơn dày”

The Zen of Python.

Khoảng trắng có thể rất hữu ích trong biểu thức và câu lệnh khi sử dụng đúng cách. Nếu không có đủ khoảng trắng thì đoạn code có thể khó đọc. Nếu có quá nhiều khoảng trống thì khó có thể kết hợp các mục liên quan trong câu lệnh một cách trực quan.

Khoảng trắng xung quanh toán tử nhị phân

Bao quanh các toán tử nhị phân với một khoảng trắng ở hai bên:

- Phép gán (=, +==, -=, v.v.)
- Phép so sánh (==, !=, >, <, <=, >=) và (is, is not, not in)
- Booleans (and, not, or)

Chú ý: Khi dấu = được sử dụng để gán cho một giá trị mặc định cho một đối số hàm thì không nên bao quanh nó với khoảng trắng.

```
# Recommended
def function(default_parameter=5):
    # ...


# Not recommended
def function(default_parameter = 5):
    # ...
```

Khi có nhiều hơn một toán tử trong một câu lệnh, việc thêm mọt khoảng trống trước và sau mỗi toán tử có thể gây nhầm lẫn. Thay vào đó, tốt hơn là chỉ thêm khoảng trắng xung quanh các toán tử với mức ưu tiên thấp nhất, đặc biệt khi biểu diễn các phép toán. Dưới đây là một vài ví dụ:

```
# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not Recommended
y = x ** 2 + 5
z = (x + y) * (x - y)
```

Bạn cũng có thể ứng dụng khoảng trắng vào câu lệnh *If* khi có nhiều điều kiện:

```
# Not recommended
if x > 5 and x % 2 == 0:
    print('x is larger than 5 and divisible by 2!')
```

Trong ví dụ trên, toán tử "*and*" có mức ưu tiên thấp nhất. Do đó nó có thể sẽ rõ ràng hơn để biểu diễn câu lệnh *If* dưới đây:

```
# Recommended
if x>5 and x%2==0:
    print('x is larger than 5 and divisible by 2!')
```

Bạn nên thoải mái khi chọn cái này rõ ràng hơn, với cảnh báo rằng bạn phải sử dụng khoảng trắng ở hai bên toán tử.

Ví dụ dưới đây là không được phép :

```
# Definitely do not do this!
if x >5 and x% 2== 0:
    print('x is larger than 5 and divisible by 2!')
```

Trong ví dụ trên, dấu hai chấm hoạt động như một toán tử nhị phân. Do đó, các quy tắc được chỉ ra ở phần trước đó áp dụng, phải có cùng một khoảng trắng ở hai bên. Các ví dụ sau đây liệt kê các khoảng trắng hợp lệ:

```
list[3:4]

# Treat the colon as the operator with lowest priority
list[x+1 : x+2]

# In an extended slice, both colons must be
# surrounded by the same amount of whitespace
```

```
list[3:4:5]
list[x+1 : x+2 : x+3]

# The space is omitted if a slice parameter is omitted
list[x+1 : x+2 :]
```

Tóm lại, bạn nên bao quanh phần lớn toán tử với khoảng trắng. Tuy nhiên, có một số cảnh báo đến nguyên tắc này, như trong câu lệnh có chứa hàm hay khi bạn kết hợp nhiều toán tửt trong một câu lệnh.

Lúc nào nên tránh thêm khoảng trắng ?

Trong một số trường hợp, việc thêm khoảng trắng có thể làm cho đoạn code khó đọc hơn. Quá nhiều khoảng trắng có thể làm cho việc code quá thưa thớt và khó theo dõi. PEP 8 phác thảo các ví dụ rất rõ ràng trong đó khoảng trắng không phù hợp.

Vị trí quan trọng nhất để tránh thêm khoảng trắng là ở cuối dòng. Điều này được gọi là dấu khoảng trắng. Nó là vô hình và có thể tạo ra các lỗi khó theo dõi.

Danh sách sau đây chỉ ra một số trường hợp bạn nên tránh thêm khoảng trắng:

- Ngay lập tức trong dấu ngoặc đơn, dấu ngoặc vuông hay ngoặc nhọn:

```
# Recommended
my_list = [1, 2, 3]

# Not recommended
my_list = [ 1, 2, 3, ]
```

- Trước dấu phẩy, chấm phẩy hoặc hai chấm:

```
x = 5
y = 6

# Recommended
print(x, y)

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

- Trước dấu mở ngoặc bắt đầu một danh sách các lệnh gọi hàm:

```
def double(x):
    return x * 2

# Recommended
double(3)

# Not recommended
double (3)
```

- Trước dấu mở ngoặc bắt đầu một mục lục:

```
# Recommended
list[3]

# Not recommended
list [3]
```

- Giữa dấu phẩy và dấu ngoặc đóng:

```
# Recommended
tuple = (1,)

# Not recommended
tuple = (1, )
```

- Để căn chỉnh các toán tử:

```
# Recommended
var1 = 5
var2 = 6
some_long_var = 7

# Not recommended
var1          = 5
var2          = 6
```

```
some_long_var = 7
```

Hãy chắc chắn rằng không có khoảng trắng ở bất kỳ đâu trong đoạn code của bạn. Có những trường hợp khác mà PEP 8 không khuyến khích thêm khoảng trắng, chẳng hạn như ngay bên trong dấu ngoặc hay trước dấu phẩy và dấu hai chấm. Bạn cũng nên không bao giờ thêm khoảng trống để căn chỉnh các toán tử.

6. Khuyến nghị lập trình

"Đơn giản tốt hơn là phức tạp."

The Zen of Python.

Bạn thường sẽ thấy rằng có một số cách để thực hiện một hành động tương tự như trong Python (và bất kỳ ngôn ngữ lập trình nào khác cho vấn đề đó). Trong phần này bạn sẽ thấy một số gợi ý mà PEP 8 cung cấp giúp loại bỏ sự mơ hồ đó và duy trì tính nhất quán.

Đừng so sánh các giá trị booleans với True hoặc False sử dụng toán tử tương đương

Bạn thường sẽ cần kiểm tra xem giá trị boolean là Đúng hay Sai. Khi làm như vậy, nó sẽ trở nên trực quan hơn khi thực hiện những câu lệnh dưới đây:

```
my_bool = 6 > 5
if my_bool == True:
    return '6 is bigger than 5'
```

Việc sử dụng toán tử tương đương, ==, là không cần thiết ở đây. bool chỉ có thể lấy giá trị Đúng hoặc Sai. Điều đó là đủ để viết như sau:

```
# Recommended
if my_bool:
    return '6 is bigger than 5'
```

Cách thực hiện câu lệnh If với boolean này yêu cầu ít code hơn và đơn giản hơn, vì vậy PEP 8 khuyến khích nó.

Sử dụng định nghĩa khi kiểm tra các chuỗi trống trong câu lệnh if

Nếu bạn muốn kiểm tra xem một danh sách có trống không, bạn có thể muốn kiểm tra độ dài của danh sách. Nếu danh sách trống, độ dài chuỗi bằng 0 tương đương với Sai khi được sử dụng trong câu lệnh *If*. Dưới đây là một ví dụ:

```
# Not recommended
my_list = []
if not len(my_list):
    print('List is empty!')
```

Mặc dù cả hai ví dụ sẽ in ra *List is empty!*, Tùy chọn thứ hai đơn giản hơn, vì vậy PEP 8 khuyến khích nó.

Sử dụng *is not* là chứ không phải là *not…is* trong câu lệnh *If*

Nếu bạn đang cố kiểm tra xem một biến có giá trị xác định hay không, có hai tùy chọn. Đầu tiên là đánh giá một câu lệnh If với *x is not None*, như trong ví dụ dưới đây:

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

Tùy chọn thứ hai sẽ là đánh giá *x is not None* và sau đó có câu lệnh if dựa trên kết quả *not* là kết quả:

```
# Not recommended
if not x is None:
    return 'x exists!'
```

Mặc dù cả hai tùy chọn sẽ được đánh giá chính xác, nhưng tùy chọn đầu tiên đơn giản hơn, vì vậy PEP 8 khuyến khích nó.

Đừng sử dụng *if x:* khi bạn có ý là *if x not None:*

Đôi khi, bạn có thể có một hàm với các đối số là None mặc định. Một lỗi phổ biến khi kiểm tra đối số arg, đã được đưa ra một giá trị khác, sử dụng như sau:

```
# Not Recommended
if arg:
    # Do something with arg...
```

Code này kiểm tra rằng arg có tồn tại. Thay vào đó, bạn muốn kiểm tra xem arg không phải là None, tốt hơn nếu sử dụng như sau:

```
# Recommended
if arg is not None:
    # Do something with arg...
```

Sai lầm đang được thực hiện ở đây là giả định rằng không phải None và sự thật là tương đương. Bạn có thể đã đặt arg = [ ]. Như chúng ta đã thấy ở trên, các danh sách trống được đánh giá là sai lệch trong Python. Vì vậy, mặc dù đối số arg đã được chỉ định điều kiện không được đáp ứng và do đó mã trong phần thân của câu lệnh If sẽ không được thực thi.

Sử dụng *.startswith ()* và *.endswith () thay* vì cắt lát

Nếu bạn đang cố kiểm tra xem một từ chuỗi có tiền tố hay hậu tố hay không, với từ *cat*, có vẻ hợp lý khi sử dụng danh sách cắt. Tuy nhiên, việc cắt danh sách dễ bị lỗi và bạn phải

mã hóa số lượng ký tự trong tiền tố hoặc hậu tố. Nó cũng không rõ ràng với một người không quen với danh sách cắt của Python điều mà bạn đang cố gắng đạt được:

```
# Not recommended
if word[:3] == 'cat':
    print('The word starts with "cat"')
```

Tuy nhiên, cách này không dễ đọc như sử dụng *.startswith():*

```
# Recommended
if word.startswith('cat'):
    print('The word starts with "cat"')
```

Tương tự, cùng một nguyên tắc được áp dụng khi bạn kiểm tra các hậu tố. Ví dụ dưới đây phác thảo cách bạn có thể kiểm tra xem một chuỗi kết thúc bằng jpg:

```
# Not recommended
if file_name[-3:] == 'jpg':
    print('The file is a JPEG')
```

Mặc dù kết quả là chính xác, nhưng ký hiệu này hơi khó sử dụng và khó đọc. Thay vào đó, bạn có thể sử dụng *.endswith ()* như trong ví dụ bên dưới:

```
# Recommended
if file_name.endswith('jpg'):
    print('The file is a JPEG')
```

Như với hầu hết các khuyến nghị lập trình này, mục tiêu là dễ đọc và đơn giản. Trong Python, có nhiều cách khác nhau để thực hiện cùng một hành động, vì vậy các hướng dẫn về lựa chọn phương pháp là hữu ích.

7. Khi nào thì có thể lờ đi PEP 8

Câu trả lời ngắn cho câu hỏi này là **không bao giờ**. Nếu bạn tuân thủ theo mọi lời khuyên của PEP 8, bạn có thể đảm bảo rằng bạn sẽ có code sạch, chuyên nghiệp và dễ đọc. Điều này sẽ có lợi cho bạn cũng như các cộng tác viên và nhà tuyển dụng tiềm năng.

Tuy nhiên, một vài chỉ dẫn trong PEP 8 là không thuận tiện trong một số trường hợp dưới đây:

- Nếu tuân thủ PEP 8 sẽ phá vỡ tính tương thích với phần mềm hiện có.
- Nếu phần code xung quanh những gì bạn đang làm là không phù hợp với PEP 8.
- Nếu code cần duy trì tương thích với các phiên bản cũ hơn của Python.

Mẹo và thủ thuật giúp bạn đảm bảo code tuân thủ theo PEP 8

Có rất nhiều điều cần nhớ để đảm bảo code của bạn tuân thủ PEP 8. Nó có thể được ưu tiên để ghi nhớ tất cả các quy tắc này khi bạn phát triển code. Nó đặc biệt tốn thời gian để cập nhật các dự án trong quá khứ để tuân thủ PEP 8. May mắn thay, có những công cụ có thể giúp tăng tốc quá trình này. Có hai loại công cụ mà bạn có thể sử dụng để thực thi tuân thủ PEP 8: *linters* và *autoformatters.*

Linters

Linters là các chương trình phân tích code và lỗi flag. Họ cung cấp các đề xuất về cách sửa lỗi. Linters đặc biệt hữu ích khi được cài đặt làm tiện ích mở rộng cho trình soạn thảo văn bản của bạn, vì chúng đánh dấu các lỗi flag và các vấn đề về kiểu dáng trong khi bạn viết. Trong phần này, bạn sẽ thấy một phác thảo về cách thức hoạt động của các linters, với các liên kết đến phần mở rộng trình soạn thảo văn bản ở cuối.

Các linters tốt nhất cho code Python là như sau:

- *pycodestyle* là một công cụ để kiểm tra code Python của bạn so với một số kiểu quy ước trong PEP 8. Cài đặt *pycodestyle* sử dụng *pip:*

```
$ pip install pycodestyle
```

Hoặc sử dụng câu lệnh:

```
$ pycodestyle code.py
code.py:1:17: E231 missing whitespace after ','
code.py:2:21: E231 missing whitespace after ','
code.py:6:19: E711 comparison to None should be 'if cond is None:'
```

- *flake8* là một công cụ kết hợp một trình gỡ lỗi, *pyflakes*, với Cài đặt *pycodestyle* sử dụng *pip*

```
$ pip install flake8
```

Hoặc sử dụng câu lệnh:

```
$ flake8 code.py
code.py:1:17: E231 missing whitespace after ','
code.py:2:21: E231 missing whitespace after ','
code.py:3:17: E999 SyntaxError: invalid syntax
code.py:6:19: E711 comparison to None should be 'if cond is None:'
```

Lưu ý: Dòng *output* thêm vào giúp chỉ ra lỗi cú pháp.

Chúng cũng có sẵn dưới dạng các phần mở rộng cho Atom, Sublime Text, Visual Studio Code và VIM. Bạn cũng có thể tìm thấy các hướng dẫn về cách thiết lập Sublime Text và

VIM để phát triển Python, cũng như tổng quan củamột số trình soạn thảo văn bản phổ biến tại Real Python.

Autoformatters (Tự động định dạng)

Autoformatters là các chương trình tự động cấu trúc lại code của bạn để tuân thủ PEP 8. Khi chương trình như vậy là *black*, code tự động định dạng tuân theo hầu hết các quy tắc trong PEP 8. Một điểm khác biệt lớn là nó giới hạn độ dài dòng 88 ký tự, thay vì 79. Tuy nhiên, bạn có thể ghi đè lên điều này bằng cách thêm dòng lệnh, như bạn sẽ thấy trong một ví dụ bên dưới.

Cài đặt *black* bằng pip. Nó yêu cầu Python 3.6+ để chạy:

```
$ pip install black
```

Nó có thể được chạy thông qua dòng lệnh, như với các linters. Hãy thử bắt đầu với đoạn code sau đây không tuân thủ PEP 8 trong một tệp có tên *code.py*

```
for i in range(0,3):
    for j in range(0,3):
        if (i==2):
            print(i,j)
```

Sau đó, bạn có thể chạy lệnh sau thông qua yêu cầu:

```
$ black code.py
reformatted code.py
All done! ✨ 🍰 ✨
```

code.py sẽ được tự động định dạng lại để trông như thế này:

```
for i in range(0, 3):
    for j in range(0, 3):
        if i == 2:
            print(i, j)
```

Nếu bạn muốn thay đổi giới hạn độ dài dòng, thì bạn có thể sử dụng flag --line-length:

```
$ black --line-length=79 code.py
reformatted code.py
All done! ✨ 🍰 ✨
```

Hai autoformatters khác, autopep8 và yapf, thực hiện các hành động tương tự như những gì black làm. Một hướng dẫn Real Python, Python Code Quality: Tool & Best Practices của Alexander van Tol, đưa ra lời giải thích kỹ lưỡng về cách sử dụng các công cụ này.