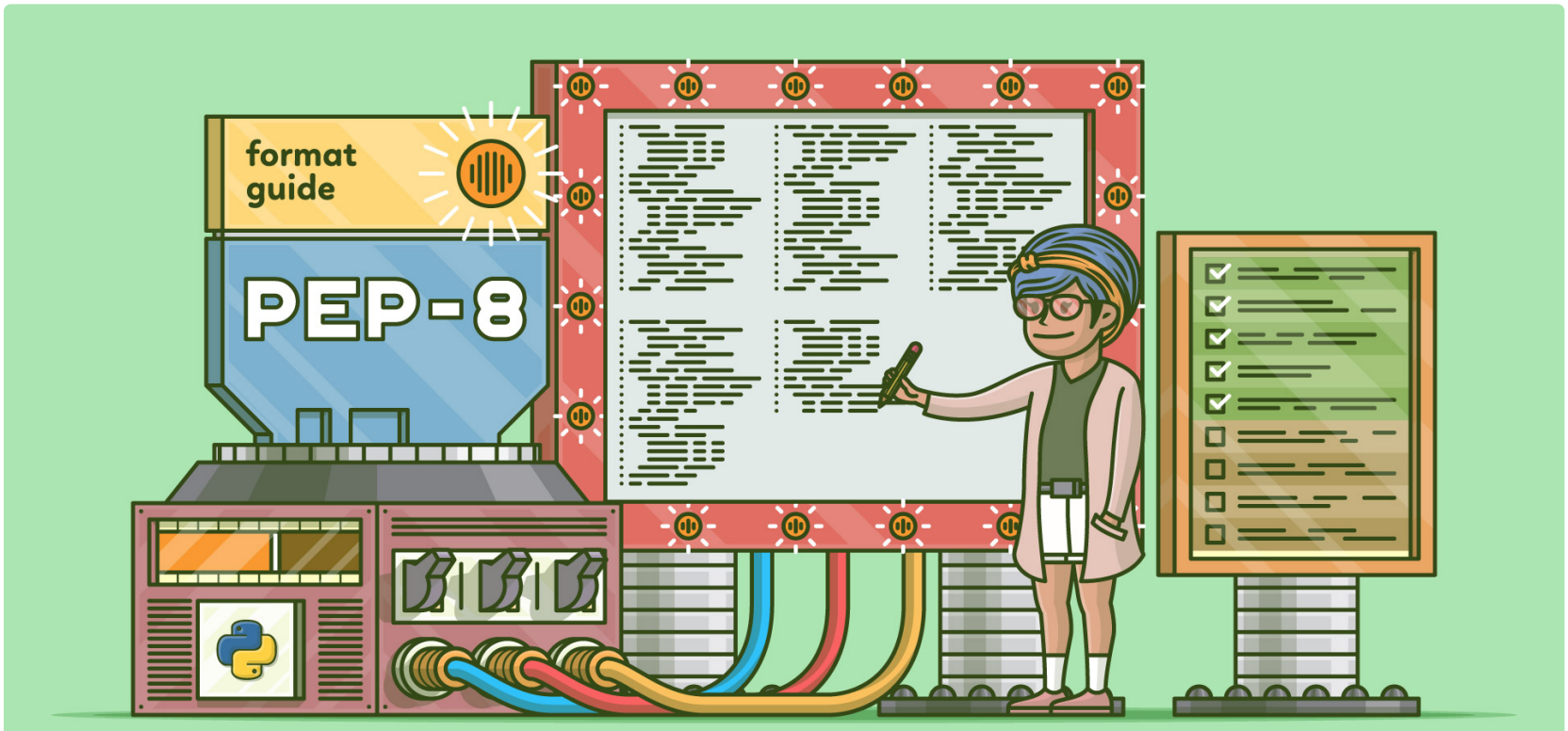



Real Python

# How to Write Beautiful Python Code With PEP 8

by Jasmine Finer 25 Comments best-practices intermediate python

Tweet Share Email

## Table of Contents



Improve Your Python

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Writing Beautiful Pythonic Code With PEP 8**

PEP 8, sometimes spelled PEP8 or PEP-8, is a document that provides guidelines and best practices on how to write Python code. It was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The primary focus of PEP 8 is to improve the readability and consistency of Python code.

PEP stands for Python Enhancement Proposal, and there are several of them. A PEP is a document that describes new features proposed for Python and docum

This tutorial outlines the key guidelines l such I have not covered some of the most documentation.

**By the end of this tutorial, you'll be abl**

- Write Python code that conforms to
- Understand the reasoning behind th
- Set up your development environm

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



**Free Bonus: 5 Thoughts On Python Mastery**, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.



## Why We Need PEP 8

> "Readability counts."
>
> — *The Zen of Python*

PEP 8 exists to improve the readability of Python code. But why is readability so important? Why is writing readable code one of the guiding principles of the Python language?

As Guido van Rossum said, "Code is read much more often than it is written." You may spend a few minutes, or a whole day, writing a piece of code to process user authentication. Once you've written it, you're never going to write it again. But you'll definitely have to read it again. That piece of code might remain part of a project you're working on. Every time you go back to that file, you'll have to remember what that code does and why you wrote it, so readability matters.

If you're new to Python, it can be difficult to remember what a piece of code does a few days, or weeks, after you wrote it. If you follow PEP 8, you can be sure that you've named your variables well. You'll know that you've added enough whitespace so it's easier to follow logical steps in your code. You'll also have commented your code well. All this will mean your code is more readable and easier to come back to. As a beginner, following the rules of PEP 8 can make learning Python a much more pleasant task.

Following PEP 8 is particularly important if you're looking for a development job. Writing clear, readable code shows professionalism. It'll tell an employer that you understand h

If you have more experience writing Python code, then you may need to collaborate with others. Writing readable code here is crucial. Other people, who may have never met you or seen your coding style before, will have to read and understand your code. Having guidelines that you follow and recognize will make it easier for others to read your code.

# Naming Conventions

> "Explicit is better than implicit."
>
> — *The Zen of Python*

When you write Python code, you have tc
Choosing sensible names will save you ti
certain variable, function, or class repres
that are difficult to debug.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



**Note**: Never use `l`, `O`, or `I` single letter r

```
Python

O = 2  # This may look like you're
```

## 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 with underscores. This style is called camel case. | `Model`, `MyClass` |
| 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`, `mypackage` |

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 Choose Names

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

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:

Python

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

This will work, but you'll have to keep tra
much clearer choice of names would be s

Python

```
>>> # Recommended
>>> name = 'John Smith'
>>> first_name, last_name = name.spli
>>> 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:

Python

```
# 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:

Python

```
# 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

> "Beautiful is better than ugly."
>
> — *The Zen of Python*

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

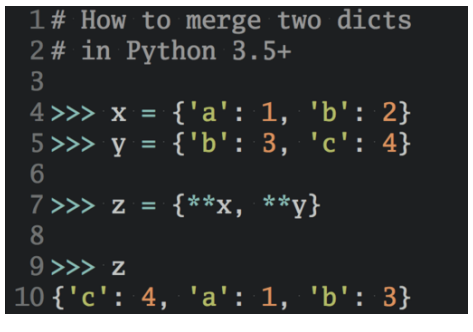
Improve Your Python

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:

Python

```
class MyFirstClass:
    pass


class MySecondClass:
    pass


def top_level_function():
    return None
```






**Surround method definitions inside cla[illegible]** another. It's good practice to leave only a single line between them:

Python

```
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:

Python

```
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 improv[illegible] visually understand how your code splits up into sections, a[illegible]

## Maximum Line Length and Line Breaking

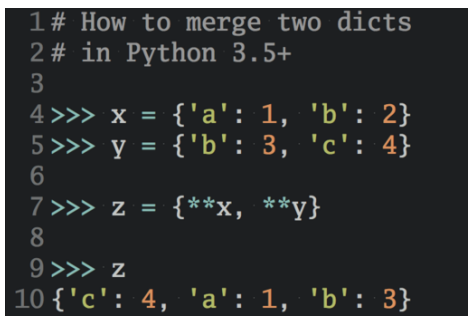
PEP 8 suggests lines should be limited to 79 characters. This is because it allows you to have multiple files open next to one another, while also avoiding line wrapping.

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:

Python

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






If it is impossible to use implied continua

Python

```
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:

Python

```
# 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:

Python

```
# 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

> "There should be one—and preferably only one—obvious way to do it."
>
> — *The Zen of Python*



Indentation, or leading whitespace, is extremely important i

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:

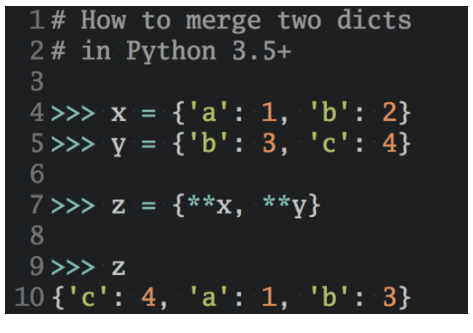
Python

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

The indented `print` statement lets Pytho
same indentation applies to tell Python w
given class.

The key indentation rules laid out by PEP

- Use 4 consecutive spaces to indicat
- Prefer spaces over tabs.






## Tabs vs. Spaces

As mentioned above, you should use spa
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:

Shell

```
$ 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:

Shell

```
$ 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:

Shell

```
$ 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 ope

Python

```
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:

Python

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

In this case, PEP 8 provides two alternati

- Add a comment after the final condi conditions from the nested code:

Python

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

- Add extra indentation on the line continuation:

Python

```
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:

Python

```
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:

Python

```
# 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:

Python

```
# 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:

Python

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

When you write PEP 8 compliant code, th
improve readability, you should indent a
doing this. The first is to align the indente
You are free to chose which method of in

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



### Where to Put the Closing Brace

Line continuations allow you to break lines inside parentheses, brackets, or braces. It's easy to forget about the closing brace, but it's important to put it somewhere sensible. Otherwise, it can confuse the reader. PEP 8 provides two options for the position of the closing brace in implied line continuations:

- Line up the closing brace with the first non-whitespace character of the previous line:

  Python

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

- Line up the closing brace with the first character of the line that starts the construct:

  Python

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

You are free to chose which option you use. But, as always, consistency is key, so try to stick to one of the above methods.

## Comments

> "If the implementation is hard to explain, it's a bad idea."
>
> — *The Zen of Python*

You should use comments to document code as it's written. It is important to document your code so that you, and any collaborators, can understand it. When you or someone else reads a comment, they should be able to easily understand the code the comment applies to and how it fits in with the rest of your code.

Here are some key points to remember when adding comments to your code:

- Limit the line length of comments and docstrings to 72 characters.
- Use complete sentences, starting with a capital letter.
- Make sure to update comments if you change your cod

## Block Comments

Use block comments to document a small section of code. They are useful when you have to write several lines of code to perform a single action, such as importing data from a file or updating a database entry. They are important as they help others understand the purpose and functionality of a given code block.

PEP 8 provides the following rules for writing block comments:

- Indent block comments to the same level as the code they describe.
- Start each line with a # followed by a
- Separate paragraphs by a line conta

Here is a block comment explaining the f
the 79 character line limit:

Python

```
for i in range(0, 10):
    # Loop over i ten times and print
    # new line character
    print(i, '\n')
```

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



Sometimes, if the code is very technical, then it is necessary to use more than one paragraph in a block comment:

Python

```
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
```

If you're ever in doubt as to what comment type is suitable, then block comments are often the way to go. Use them as much as possible throughout your code, but make sure to update them if you make changes to your code!

## Inline Comments

Inline comments explain a single statement in a piece of code. They are useful to remind you, or explain to others, why a certain line of code is necessary. Here's what PEP 8 has to say about them:

- Use inline comments sparingly.
- Write inline comments on the same line as the statement they refer to.
- Separate inline comments by two or more spaces from the statement.
- Start inline comments with a # and a single space, like block comments.
- Don't use them to explain the obvious.

Below is an example of an inline comment:

Python

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

Sometimes, inline comments can seem necessary, but you can use better naming conventions instead. Here's an example:

Python

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

Here, the inline comment does give extra information. However using `x` as a variable name for a person's name is bad practice. There's no need for the inline comment if you rename your variable:

Python

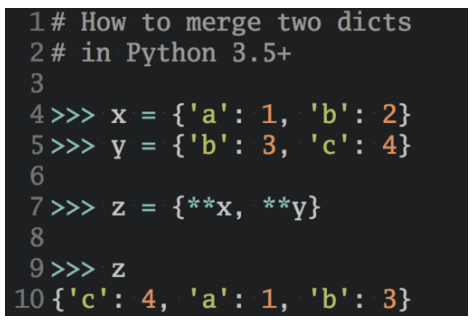
```
student_name = 'John Smith'
```

Finally, inline comments such as these ar

Python

```
empty_list = []  # Initialize empty l

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

Inline comments are more specific than b
which leads to clutter. You could get awa
comment, your code is more likely to be P






## Documentation Strings

Documentation strings, or docstrings, are strings enclosed in double (`"""`) or single (`'''`) quotation marks that appear on the first line of any function, class, method, or module. You can use them to explain and document a specific block of code. There is an entire PEP, PEP 257, that covers docstrings, but you'll get a summary in this section.

The most important rules applying to docstrings are the following:

- Surround docstrings with three double quotes on either side, as in `"""This is a docstring"""`.
- Write them for all public modules, functions, classes, and methods.
- Put the `"""` that ends a multiline docstring on a line by itself:

Python

```
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
```

- For one-line docstrings, keep the `"""` on the same line:

Python

```
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
```

For a more detailed article on documenting Python code, se
Mertz.

# Whitespace in Expressions and Statements

> "Sparse is better than dense."
>
> — *The Zen of Python*

Whitespace can be very helpful in expressions and statements when used properly. If there is not enough whitespace, then code can be difficult to read, as it's all bunched together. If there's too much whitespace, then it can be difficult to visually combine related terms in a sta

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```



## Whitespace Around Binary

Surround the following binary operators

- Assignment operators (=, +=, -=, and
- Comparisons (==, !=, >, <. >=, <=) and
- Booleans (and, not, or)

**Note**: When = is used to assign a default value to a function argument, do not surround it with spaces.

Python

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


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

When there's more than one operator in a statement, adding a single space before and after each operator can look confusing. Instead, it is better to only add whitespace around the operators with the lowest priority, especially when performing mathematical manipulation. Here are a couple examples:

Python

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

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

You can also apply this to `if` statements where there are multiple conditions:

Python

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

In the above example, the `and` operator has lowest priority. It may therefore be clearer to express the `if` statement as below:

Python

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

You are free to choose which is clearer, with the caveat that you must use the same amount of whitespace either side of the operator.

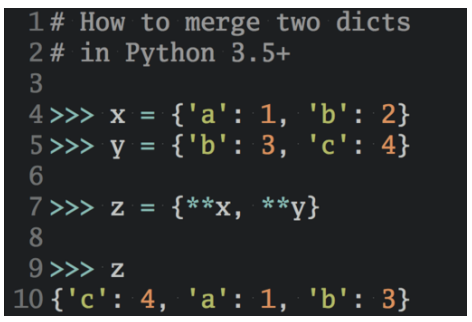
The following is not acceptable:

Python

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

In slices, colons act as a binary operators should be the same amount of whitespac

Python






```
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 :]
```

In summary, you should surround most operators with whitespace. However, there are some caveats to this rule, such as in function arguments or when you're combining multiple operators in one statement.

## When to Avoid Adding Whitespace

In some cases, adding whitespace can make code harder to read. Too much whitespace can make code overly sparse and difficult to follow. PEP 8 outlines very clear examples where whitespace is inappropriate.

The most important place to avoid adding whitespace is at the end of a line. This is known as **trailing whitespace**. It is invisible and can produce errors that are difficult to trace.

The following list outlines some cases where you should avoid adding whitespace:

- Immediately inside parentheses, brackets, or braces:

Python

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

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

- Before a comma, semicolon, or colon:

Python

```
x = 5
y = 6

# Recommended
print(x, y)

# Not recommended
```

```
print(x, y)
```

- Before the open parenthesis that starts the argument list of a function call:

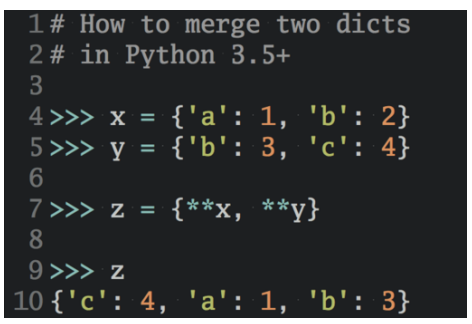
Python

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

# Recommended
double(3)

# Not recommended
double (3)
```

- Before the open bracket that starts

Python

```
# Recommended
list[3]

# Not recommended
list [3]
```






- Between a trailing comma and a closing parenthesis:

Python

```
# Recommended
tuple = (1,)

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

- To align assignment operators:

Python

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

# Not recommended
var1          = 5
var2          = 6
some_long_var = 7
```

Make sure that there is no trailing whitespace anywhere in your code. There are other cases where PEP 8 discourages adding extra whitespace, such as immediately inside brackets, as well as before commas and colons. You should also never add extra whitespace in order to align operators.

## Programming Recommendations

> "Simple is better than complex."
>
> — *The Zen of Python*

You will often find that there are several ways to perform a similar action in Python (and any other programming language for that matter). In this section, you'll see some of the suggestions PEP 8 provides to remove that ambiguity and preserve consistency.

**Don't compare boolean values to `True` or `False` using the equivalence operator.** You'll often need to check if a boolean value is True or False. When doing so, it is intuitive to do this with a statement like the one below:

Python

```python
# Not recommended
my_bool = 6 > 5
if my_bool == True:
    return '6 is bigger than 5'
```

The use of the equivalence operator, `==`, is unnecessary here. `bool` can only take values `True` or `False`. It is enough to write the following:

Python

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

This way of performing an `if` statement v

**Use the fact that empty sequences are** might be tempted to check the length of used in an `if` statement. Here's an examp

Python

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

However, in Python any empty list, string, or tuple is falsy. We can therefore come up with a simpler alternative to the above:

Python

```python
# Recommended
my_list = []
if not my_list:
    print('List is empty!')
```

While both examples will print out `List is empty!`, the second option is simpler, so PEP 8 encourages it.

**Use `is not` rather than `not ... is` in `if` statements.** If you are trying to check whether a variable has a defined value, there are two options. The first is to evaluate an `if` statement with `x is not None`, as in the example below:

Python

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

A second option would be to evaluate `x is None` and then have an `if` statement based on `not` the outcome:

Python

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

While both options will be evaluated correctly, the first is simpler, so PEP 8 encourages it.

**Don't use `if x:` when you mean `if x is not None:`.** Sometimes, you may have a function with arguments that are `None` by default. A common mistake when checking if such an argument, `arg`, has been given a different value is to use the following:

Python

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

This code checks that `arg` is truthy. Instead, you want to che

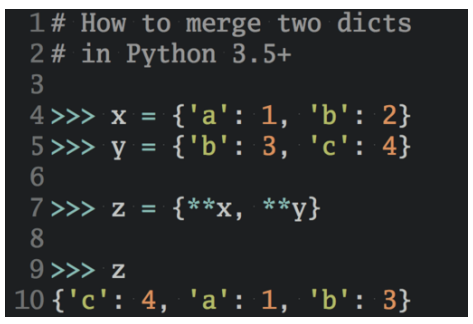
This code checks that arg is truthy. Instead, you want to check that arg is not None, so it would be better to use the following:

Python

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

The mistake being made here is assuming that `not None` and truthy are equivalent. You could have set `arg = []`. As we saw above, empty lists are evaluated as f
condition is not met, and so the code in t

**Use `.startswith()` and `.endswith()` inste**
suffixed, with the word `cat`, it might seem
have to hardcode the number of characte
Python list slicing what you are trying to






Python

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

However, this is not as readable as using `.startswith()`:

Python

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

Similarly, the same principle applies when you're checking for suffixes. The example below outlines how you might check whether a string ends in `jpg`:

Python

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

While the outcome is correct, the notation is a bit clunky and hard to read. Instead, you could use `.endswith()` as in the example below:

Python

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

As with most of these programming recommendations, the goal is readability and simplicity. In Python, there are many different ways to perform the same action, so guidelines on which methods to chose are helpful.

## When to Ignore PEP 8

The short answer to this question is never. If you follow PEP 8 to the letter, you can guarantee that you'll have clean, professional, and readable code. This will benefit you as well as collaborators and potential employers.

However, some guidelines in PEP 8 are inconvenient in the following instances:

- If complying with PEP 8 would break compatibility with existing software
- If code surrounding what you're working on is inconsistent with PEP 8
- If code needs to remain compatible with older versions of Python



## Tips and Tricks to Help Ensure Y

## Tips and Tricks to Help Ensure Your Code Follows PEP 8

There is a lot to remember to make sure your code is PEP 8 compliant. It can be a tall order to remember all these rules when you're developing code. It's particularly time consuming to update past projects to be PEP 8 compliant. Luckily, there are tools that can help speed up this process. There are two classes of tools that you can use to enforce PEP 8 compliance: linters and autoformatters.

### Linters

Linters are programs that analyze code and flag errors. They provide suggestions on how to fix the error. Linters are particularly useful when installed as extensions to your text editor, as they flag errors and stylistic problems while you write. In this section, you'll see an outline of how the linters work, with links to the text editor extensions at the end.

The best linters for Python code are the f

- `pycodestyle` is a tool to check your F

  Install `pycodestyle` using `pip`:

  Shell

  ```
  $ pip install pycodestyle
  ```

  You can run `pycodestyle` from the te

  Shell

  ```
  $ 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` is a tool that combines a debugger, `pyflakes`, with `pycodestyle`.

  Install `flake8` using `pip`:

  Shell

  ```
  $ pip install flake8
  ```

  Run `flake8` from the terminal using the following command:

  Shell

  ```
  $ 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:'
  ```

  An example of the output is also shown.

> **Note**: The extra line of output indicates a syntax error.

These are also available as extensions for Atom, Sublime Text, Visual Studio Code, and VIM. You can also find guides on setting up Sublime Text and VIM for Python development, as well as an overview of some popular text editors at *Real Python*.

### Autoformatters

Autoformatters are programs that refactor your code to conform with PEP 8 automatically. Once such program is `black`, which autoformats code following *most* of the rules in PEP 8. One big difference is that it limits line length to 88 characters, rather than 79. However, you can overwrite this by adding a command line flag, as you'll see in an example below.

Install `black` using `pip`. It requires Python 3.6+ to run:

Shell

```
$ pip install black
```

It can be run via the command line, as with the linters. Let's say you start with the following code that isn't PEP 8 compliant in a file called `code.py`:

Python

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

You can then run the following command

Shell

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

`code.py` will be automatically reformatted to look like this:

Python

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

If you want to alter the line length limit, then you can use the `--line-length` flag:

Shell

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

Two other autoformatters, `autopep8` and `yapf`, perform actions that are similar to what `black` does.

Another *Real Python* tutorial, Python Code Quality: Tools & Best Practices by Alexander van Tol, gives a thorough explanation of how to use these tools.

## Conclusion

You now know how to write high-quality, readable Python code by using the guidelines laid out in PEP 8. While the guidelines can seem pedantic, following them can really improve your code, especially when it comes to sharing your code with potential employers or collaborators.

In this tutorial, you learned:

- What PEP 8 is and why it exists
- Why you should aim to write PEP 8 compliant code
- How to write code that is PEP 8 compliant

On top of all this, you also saw how to use linters and autoformatters to check your code against PEP 8 guidelines.

If you want to learn more about PEP 8, then you can read the full documentation, or visit pep8.org, which contains the same information but has been nicely formatted. In these documents, you will find the rest of the PEP 8 guidelines not covered in this tutorial.

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Writing B**

## About **Jasmine Finer**

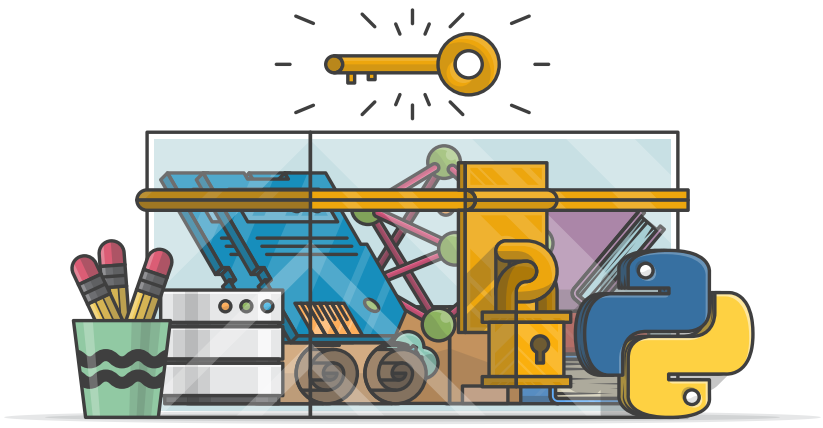
Jasmine is a Django developer, based in London.

» More about Jasmine

*Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:*

Aldren Brad Dan

Joanna





## What Do You Think?



**Real Python Comment Policy:** The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

## Keep Learning

Related Tutorial Categories: best-practices intermediate python

Recommended Video Course: Writing Beautiful Pythonic Code With PEP 8

## All Tutorial Topics

advanced | api | basics | best-practices | community | databases | data-science | devops | django | docker | flask | front-end | intermediate | machine-learning | python | testing | tools | web-dev | web-scraping



## Table of Contents