How to Write Beautiful Python Code With PEP 8

Jasmine Finer ⋮⋮ 7.2.2024

by Jasmine Finer Feb 07, 2024 intermediate best-practices

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 Alyssa Coghlan. The primary focus of PEP 8 is to improve the **readability** and **consistency** of Python code.

By the end of this tutorial, you'll be able to:

- Write Python code that **conforms to PEP 8**
- **Understand the reasoning** behind the guidelines laid out in PEP 8
- Set up your development environment so that you can **start writing PEP 8 compliant Python code**

PEP stands for **Python Enhancement Proposal**, and there are many PEPs. These documents primarily describe new features proposed for the Python language, but some PEPs also focus on design and style and aim to serve as a resource for the community. PEP 8 is one of these style-focused PEPs.

In this tutorial, you'll cover the key guidelines laid out in PEP 8. You'll explore beginner to intermediate programming topics. You can learn about more advanced topics by reading the full PEP 8 documentation.

Get Your Code Click here to download the free sample code that shows you how to write PEP 8 compliant code.

Take the Quiz: Test your knowledge with our interactive “How to Write Beautiful Python Code With PEP 8” quiz. You’ll receive a score upon completion to help you track your learning progress:

Interactive Quiz

How to Write Beautiful Python Code With PEP 8

In this quiz, you'll test your understanding of PEP 8, the Python Enhancement Proposal that provides guidelines and best practices on how to write Python code. By working through this quiz, you'll revisit the key guidelines laid out in PEP 8 and how to set up your development environment to write PEP 8 compliant Python code.

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, according to the Zen of Python?

Note: You may encounter the term *Pythonic* when the Python community refers to code that follows the idiomatic writing style specific to Python. Pythonic code adheres to Python’s design principles and philosophy and emphasizes readability, simplicity, and clarity.

As Guido van Rossum said, “Code is read much more often than it’s 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.

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 of this will mean your code is more readable and easier to come back to. If you’re a beginner, following the rules of PEP 8 can make learning Python a much more pleasant task.

Note: 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 how to structure your code well.

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 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 potentially confusing names that might result in errors that are difficult to debug.

One suggestion is to never use `l`, `O`, or `I` single letter names as these can be mistaken for `1` and `0`, depending on what typeface a programmer uses.

For example, consider the code below, where you assign the value `2` to the single letter `O`:

Python

Doing this may look like you're trying to reassign `2` to zero. While making such a reassignment isn't possible in Python and will cause a syntax error, using an ambigious variable name such as `O` can make your code more confusing and harder to read and reason about.

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`, `python_function`
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	`x`, `var`, `python_variable`
Class	Start each word with a capital letter. Don't separate words with underscores. This style is called camel case or Pascal case.	`Model`, `PythonClass`

Type	Naming Convention	Examples
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`, `PYTHON_CONSTANT`, `PYTHON_LONG_CONSTANT`
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	`module.py`, `python_module.py`
Package	Use a short, lowercase word or words. Don't separate words with underscores.	`package`, `pythonpackage`

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 this 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're 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`

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:

Python ✅ `Recommended`

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

Python ❌ `Not recommended`

At first glance, this could seem like a sensible choice. The name `db()` could 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

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

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

Therefore, PEP 8 suggests surrounding top-level functions and class definitions with two blank lines.

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

Python

In the code example, you can see a class definition with two instance methods that are separated from one another with a single blank line.

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, you can leave a blank line between each logical step.

In the example below, there's a function to calculate the variance of a list. This is two-step problem, so you can indicate the two separate steps by leaving a blank line between them:

Python

```
def calculate_variance(numbers):
    sum_numbers = 0
    for number in numbers:
        sum_numbers = sum_numbers + number
    mean = sum_numbers / len(numbers)

    sum_squares = 0
    for number in numbers:
        sum_squares = sum_squares + number**2
    mean_squares = sum_squares / len(numbers)

    return mean_squares - mean**2
```

In this code example, you separated the logical steps with a blank line in between them to improve readability. There is also a blank line before the `return` statement. This helps the reader clearly see what the function returns.

If you use vertical whitespace carefully, it can greatly improve 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 allows you to have multiple files open next to one another, while also avoiding line wrapping.

Of course, keeping statements to 79 characters or fewer isn't always possible. Therefore, PEP 8 also 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

In this example, you moved `arg_three` and `arg_four` onto a new line, indented at the same level as the first argument. You can split your code like that because of Python's implicit line joining inside of parentheses.

If it's impossible to use implied continuation, then you can use backslashes (`\`) to break lines instead:

Python

However, any time that you *can* use implied continuation, then you should prefer that over using a backslash.

If you need to break a line around binary operators, like + and *, then you should do so *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`

You can immediately see which variable Python will add or subtract, as the operator is right next to the variable it operates on.

Now, here's an example of breaking *after* a binary operator:

Python ❌ `Not recommended`

Here, it's harder to see which variable Python is adding and which one it's subtracting.

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 in Python. The indentation level of lines of code in Python determines how Python groups statements together.

Consider the following example:

Python

The indented call to `print()` lets Python know that it should only execute this line if the `if` statement returns `True`. The same indentation applies to tell Python what code to execute when you're calling a function or what code belongs to a given class.

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

- Use four consecutive spaces to indicate indentation.
- Prefer spaces over tabs.

While Python code will work with any amount of *consistent* indentation, four spaces is a widespread convention in the Python community, and you should stick to it as well.

Tabs vs Spaces

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

Python 3 doesn't allow mixing of tabs and spaces. Write the following code and make sure to use spaces where indicated with a dot (`·`) and a tab character where you can see the (`⇥`) symbol:

Python `mixed_indentation.py`

```
1def mixed_indentation(greet=True):
2····if greet:
3········print("Hello")
4⇥   print("World")  # Indented with a tab.
5
6mixed_indentation()
```

The difference is invisible when you're just looking at the code, but make sure to use a tab character for indentation in line 4, while using four space characters for the other indentations.

You can also download the file `mixed_indentation.py`:

Get Your Code Click here to download the free sample code that shows you how to write PEP 8 compliant code.

If you're using Python 3 and run code that mixes tabs and spaces, then you'll get an error:

Shell

```
$ python mixed_indentation.py
  File "./mixed_indentation.py", line 4
    print("World")  # Indented with a tab.
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, then you must be consistent with your choice. Otherwise, your code won't run.

Note: Legacy Python 2 won't raise an error if you mix tabs and spaces. This can lead to ambiguous situations depending on how a text editor interprets a tab character.

If you're interested in trying this out, then you can install a version of Python 2 using pyenv. If you run `mixed_indentation.py` with Python 2, then you won't get to see an error message unless you use the `-t` or `-tt` option.

PEP 8 recommends that you always use four consecutive spaces to indicate indentation.

Indentation Following Line Breaks

When you're using line continuations to keep lines under 79 characters, it's 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:

1. Alignment with the opening delimiter
2. Hanging indent

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

Python

Sometimes you can find that four spaces exactly align with the opening delimiter. If you're curious about how PEP 8 suggests resolving these edge cases, then you can expand the collapsible section below:

Line continuations that produce visual conflicts often occur in `if` statements that span multiple lines because the `if`, space, and opening bracket make up four characters. In this case, it can be difficult to determine where the nested code block inside the `if` statement begins:

Python

PEP 8 doesn't have a strict stance on how to address this situation, but it provides two alternatives to help improve readability.

One possibility is to *add a comment* after the final condition. Due to syntax highlighting in most editors, this will separate the conditions from the nested code:

Python

The comment visually separates the multiline `if` statement from the indented code block.

Another option that PEP 8 mentions is to *add extra indentation* on the line continuation:

Python

By adding extra indentation on the line continuation, you get a quick visual idea of which indented code is still part of the `if` statement and which code is part of the indented block.

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:

Python

You indented the first argument, `arg_one`, using a hanging indent. Further line continuations should follow the same level of indentation as the first indented line.

Note that if 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`

This example doesn't comply with PEP 8 because you put `arg_one` and `arg_two` on the first line next to the opening parentheses.

When you're using a hanging indent, then you should also 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`

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 ✅ `Recommended`

When you write code following the style guide for Python 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's a continued line.

As shown above, there are two ways of doing this:

1. Align the indented block with the opening delimiter.
2. Use a hanging indent.

You're free to choose either of these two approaches for indenting your code following a line break.

Where to Put the Closing Bracket

Line continuations allow you to break lines inside parentheses, brackets, or braces. The closing bracket may not be your primary focus when programing, but it's still important to put it somewhere sensible. Otherwise, it can confuse the reader.

PEP 8 provides two options for the position of the closing bracket in implied line continuations:

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

Python

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

  Python

You're free to choose either option. 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's 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 that the comment applies to and know 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 code.

With these key points in mind, it's time to look into the nitty-gritty of PEP 8's recommendations regarding comments.

Block Comments

Use block comments to document a small section of code. These 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're important in helping 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 that they describe.
- Start each line with a # followed by a single space.
- Separate paragraphs by a line containing a single #.

Here's a block comment explaining the function of a `for` loop. Note that the sentence wraps to a new line to preserve the 79-character line limit:

Python

```
for number in range(0, 10):
    # Loop over `number` ten times and print out the value of `number`
```

```
    # followed by a newline character.
    print(number, "\n")
```

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

Python

```
# Calculate the solution to a quadratic equation using the quadratic
# formula.
#
# A quadratic equation has the following form:
# ax**2 + bx + c = 0
#
# 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)
```

If you're ever in doubt as to which 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're 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 from the statement by two or more spaces.
- 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

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

Python ❌ Not recommended

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 ✅ `Recommended`

Finally, inline comments such as the ones shown below are bad practice because they state the obvious and clutter the code:

Python ❌ `Not recommended`

Neither of the two comments adds information that the code doesn't already clearly show. It's better to avoid writing such comments.

Note: You should write your code so that it's self-explanatory whenever possible, and reserve comments for situations where additional explanation is necessary. For example, you might need comments to explain *why* you wrote your code in a certain way.

Inline comments are more specific than block comments, and it's easy to add them when they're not necessary, which leads to clutter. You can get away with only using block comments. Unless you're sure you need an inline comment, your code is more likely to follow the style guide for Python code if you stick to using only block comments.

Documentation Strings

Documentation strings, or docstrings, are strings enclosed in triple double quotation marks (`"""`) or triple single quotation marks (`'''`) that appear on the first line of any function, class, method, or module:

Python `documented_module.py`

You use docstrings to explain and document a specific block of code. They're an important part of Python, and you can access the docstring of an object using its `.__doc__` attribute or the `help()` function:

Python

```
>>> import documented_module

>>> documented_module.__doc__
'This is a docstring.'

>>> help(documented_module)
Help on module documented_module:

NAME
    documented_module - This is a docstring.

FILE
    ./documented_module.py
```

While PEP 8 mentions docstrings, they represent a large enough topic that there's a separate document, PEP 257 that's entirely dedicated to docstrings.

The main takeaway is that docstrings are a structured approach to documenting your Python code. You should write them for all public modules, functions, classes, and methods.

If the implementation is straightforward, then you can use a one-line docstring, where you can keep the whole docstring on the same line:

Python

If the implementation is more complex, then you'll need more lines to create a useful docstring. In that case, you should start with an overview description in the first line and end that line with a period.

Then you can use more text to document the code object. In this part of the docstring, you can also include a description of the arguments and return value.

Finally, you should put the three quotation marks that end a multiline docstring on a line by themselves:

Python

```
def quadratic(a, b, c):
    """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 a more detailed tutorial on documenting Python code that also covers the different docstring styles, see Documenting Python Code: A Complete Guide.

Whitespace in Expressions and Statements

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

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

Whitespace Around Binary Operators

For best readability according to PEP 8, surround the following binary operators with a single space on either side:

- Assignment operators: `=`, `+=`, `-=`, and so forth
- **Comparisons**: `==`, `!=`, `>`, `<`. `>=`, `<=`, `is`, `is not`, `in`, and `not in`
- **Booleans**: `and`, `not`, and `or`

When you use the equal sign (=) to assign a default value to an argument, don't surround it with spaces:

Python

Avoiding whitespace for indicating default values for arguments keeps function and method definitions more concise.

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

Python

If you use whitespace to group multiple operators according to their operator precedence, then it'll make your code better readable.

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

Python ❌ `Not recommended`

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

Python ✅ `Recommended`

You're free to choose the one that's clearer, with the caveat that you must use the same amount of whitespace on either side of the operator.

The following isn't acceptable:

Python ❌ `Not recommended`

In this example, you're using inconsistent amounts of whitespace on either side of the operators. PEP 8 discourages such inconsistency in your code because it makes it harder to correctly interpret the code when you read it.

In slices, colons act as binary operators. Therefore, the rules outlined in the previous section apply, and there should be the same amount of whitespace on either side. The following examples of list slices are valid:

Python ✅ Recommended

```
a_list[3:4]

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

# In an extended slice, you must surround both colons
# with the same amount of whitespace.
a_list[3:4:5]
a_list[x+1 : x+2 : x+3]

# You omit the space if you omit a slice parameter.
a_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 your 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's invisible and can produce noisy diffs when working with version control and may even produce errors in some situations:

Python `trailing_whitespace.py`

In the example file above, you attempted to continue the assignment expression over two lines using the line continuation marker. However, you left a trailing whitespace after the backslash (`\`) and before the newline character.

This trailing whitespace prevents Python from understanding it as a line-continuation marker and will cause a syntax error:

Shell

While Python will notice the problem and inform you about it, it's best practice to just avoid any trailing whitespace in your Python code.

PEP 8 also outlines some other cases where you should *avoid* whitespace:

- Immediately inside parentheses, brackets, or braces:

  Python

- Before a comma, semicolon, or colon:

  Python

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

  Python

- Before the open bracket that starts an index or slice:

  Python

- Between a trailing comma and a closing parenthesis:

  Python

- To align assignment operators:

  Python

The most important takeaway is to make sure that there's no trailing whitespace anywhere in your code. Then, there are other cases where PEP 8 discourages adding extra whitespace, such as immediately inside brackets, as well as before commas and colons. You also shouldn't add extra whitespace in order to align operators.

Programming Recommendations

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

You'll often find that there are several ways to perform a similar action in Python, as in any other programming language. In this section, you'll see some of the suggestions that 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. You may want to do this with a statement like the one below:

Python ❌ `Not recommended`

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

Python ✅ `Recommended`

This way of performing an `if` statement with a Boolean requires less code and is simpler, so PEP 8 encourages it.

Use the fact that empty sequences are falsy in `if` statements. If you want to check whether a list is empty, you might be tempted to check the length of the list. If the list is empty, then its length is `0` which is equivalent to `False` when you use it in an `if` statement. Here's an example:

Python ❌ `Not recommended`

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

Python ✅ `Recommended`

While both examples will print out `List is empty!`, the second option is more straightforward to read and understand, so PEP 8 encourages it.

Use `is not` rather than `not ... is` in `if` statements. If you're 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 ✅ `Recommended`

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

Python ❌ `Not recommended`

While Python will evaluate both options correctly, the first is more straightforward, 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 a different value is to use the following:

Python ❌ `Not recommended`

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

Python ✅ `Recommended`

The mistake here is assuming that `not None` and truthy are equivalent, but they aren't. You could have set `arg` to an empty list (`[]`). As you saw above, empty lists are also evaluated as falsy in Python. So,

even though you assigned a value to `arg`, the condition isn't met, and Python won't execute the code in the body of the `if` statement:

Python

You can see that the two approaches produce different results when you're working with values that are falsy in Python.

Use `.startswith()` and `.endswith()` instead of slicing. If you were trying to check if the string `word` was prefixed or suffixed with the word *cat*, then it might seem sensible to use list slicing. However, list slicing is prone to error, and you have to hard-code the number of characters in the prefix or suffix. It's also not clear to someone less familiar with Python list slicing what you're trying to achieve:

Python ❌ `Not recommended`

However, this isn't as readable as using `.startswith()`:

Python ✅ `Recommended`

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`

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`

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 can be 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

PEP 8 dedicates a short section to note that you shouldn't apply PEP 8 if there are good reasons not to. It also mentions that it's most important to keep code consistent within the context where you're writing it.

Tips and Tricks to Help Ensure Your Code Follows PEP 8

There's a lot to remember to make sure your code follows the style guide for Python code. 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 can help you enforce these style rules: **linters** and **autoformatters**.

Linters

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

Some good linters for Python code are **`pycodestyle`**, **`flake8`**, and **`ruff`**. You can try them out with the following code snippet that contains formatting that isn't compliant with PEP 8:

Python `unfashionable.py`

```
import math

numbers = [1,2,\
3,4]

def add_all_numbers_from_collection(
    number_one, number_two, number_three,
    number_four):
    return number_one+number_two + number_three +number_four

print (add_all_numbers_from_collection( *numbers ))
```

`pycodestyle` is a tool to check your Python code against some of the style conventions in PEP 8.

You can install `pycodestyle` using `pip`:

Shell

Then, you can run `pycodestyle` from your terminal, passing it the Python file that you want to check as an argument:

Shell

```
$ pycodestyle unfashionable.py
unfashionable.py:3:13: E231 missing whitespace after ','
unfashionable.py:3:15: E231 missing whitespace after ','
unfashionable.py:3:16: E502 the backslash is redundant between brackets
unfashionable.py:4:1: E128 continuation line under-indented for visual indent
unfashionable.py:4:2: E231 missing whitespace after ','
unfashionable.py:6:1: E302 expected 2 blank lines, found 1
unfashionable.py:8:5: E125 continuation line with same indent as next logical line
unfashionable.py:9:50: E225 missing whitespace around operator
unfashionable.py:11:1: E305 expected 2 blank lines after class or function
definition, found 1
unfashionable.py:11:6: E211 whitespace before '('
unfashionable.py:11:40: E201 whitespace after '('
unfashionable.py:11:49: E202 whitespace before ')'
unfashionable.py:11:52: W292 no newline at end of file
```

The linter outputs the line and column numbers where it encountered a style violation. It also gives you the error code for the specific style violation, together with a short description of the issue.

Another popular option for linting your Python code is `flake8`. It's a tool that combines other projects, such as error detection using `pyflakes`, with `pycodestyle`.

Again, you can install `flake8` using `pip`:

Shell

Then, you can run `flake8` from the terminal and pass it the file that you want to check as an argument:

Shell

```
$ flake8 unfashionable.py
unfashionable.py:1:18: E999 SyntaxError: unexpected character after line
continuation character
```

Oh! It looks like your code also contained a syntax error in addition to all the formatting mess! While `pycodestyle` didn't flag it, `flake8` identifies the error and shows you how to fix it.

After you've fixed the syntax error by removing the trailing whitespace in line 3, running `flake8` again will show nearly the same style violations as you've seen before:

Shell

```
$ flake8 unfashionable.py
unfashionable.py:1:1: F401 'math' imported but unused
unfashionable.py:3:13: E231 missing whitespace after ','
unfashionable.py:3:15: E231 missing whitespace after ','
unfashionable.py:3:16: E502 the backslash is redundant between brackets
```

```
unfashionable.py:4:1: E128 continuation line under-indented for visual indent
unfashionable.py:4:2: E231 missing whitespace after ','
unfashionable.py:6:1: E302 expected 2 blank lines, found 1
unfashionable.py:8:5: E125 continuation line with same indent as next logical line
unfashionable.py:9:50: E225 missing whitespace around operator
unfashionable.py:11:1: E305 expected 2 blank lines after class or function
definition, found 1
unfashionable.py:11:6: E211 whitespace before '('
unfashionable.py:11:40: E201 whitespace after '('
unfashionable.py:11:49: E202 whitespace before ')'
unfashionable.py:11:52: W292 no newline at end of file
```

In addition to the PEP 8 style violations marked as `E` errors, you also got an `F` error type. That error tells you that you have an unused import of the `math` module in your script.

While `pycodestyle` purely lints PEP 8 style violations, `flake8` combines multiple tools and can therefore also help you to identify syntax errors, logical errors such as unused imports, and even complexity issues. You'll explore `ruff` soon.

Note: Another popular linter for your Python code is pylint.

Many linters are also available as extensions for Sublime Text, Visual Studio Code, and even VIM. If you haven't found your favorite IDE or text editor yet, then you can start by learning about VS Code or Sublime Text.

Alright, so the linter gave you a lot of good tips on what to fix in `unfashionable.py`. But it seems like it'll be a lot of work to apply all these suggestions. It'd be nice if your computer could do that work for you. That's exactly what you can use autoformatters for.

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.

Note: Two other autoformatters, `autopep8` and `yapf`, perform actions that are similar to what `black` does. Ruff also supports autoformatting.

You can install `black` using `pip`:

Shell

You can run `black` through the command line, same as you did with the linters before. Take another look at the code in `unfashionable.py` that you want to fix:

Python `unfashionable.py`

```
import math

numbers = [1,2,\
3,4]

def add_all_numbers_from_collection(
    number_one, number_two, number_three,
    number_four):
    return number_one+number_two + number_three +number_four

print (add_all_numbers_from_collection( *numbers ))
```

Note that this version of the code doesn't include the trailing whitespace that you fixed earlier on. If your code would produce a syntax error, then `black` will tell you about it and won't be able to format your code until you've fixed the error.

You can then run the following command through the command line:

Shell

After `black` has automatically reformatted `unfashionable.py`, it'll look like this:

Python `unfashionable.py`

```
import math

numbers = [1, 2, 3, 4]


def add_all_numbers_from_collection(number_one, number_two, number_three,
number_four):
    return number_one + number_two + number_three + number_four


print(add_all_numbers_from_collection(*numbers))
```

That looks significantly better than before, but all the arguments to `add_all_numbers_from_collection()` make the function definition zoom right past the PEP 8 suggested 79-character limit. As you learned before, `black` uses 88 characters as the limit instead.

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

Shell

After you limited the line length in the second run, your autoformatted code now looks great and nicely follows the style guide for Python code:

Python `unfashionable.py`

```
import math

numbers = [1, 2, 3, 4]


def add_all_numbers_from_collection(
    number_one, number_two, number_three, number_four
):
    return number_one + number_two + number_three + number_four


print(add_all_numbers_from_collection(*numbers))
```

However, you may have noticed that there's still the unused import of `math` at the top of your file. You can still identify the issue when you check your file using `flake8`:

Shell

Wouldn't it be nice if there was a tool that combined linting and formatting under one roof?

Combined Tooling

Ruff is a popular tool in the Python community that can act as both a linter and an autoformatter. It's a command-line tool that's written in Rust and therefore manages to execute *very* fast.

You can install Ruff using `pip`:

Shell

You can now use Ruff both for linting and for formatting your code:

Shell

```
$ ruff check unfashionable.py
unfashionable.py:1:8: F401 [*] `math` imported but unused
Found 1 error.
[*] 1 fixable with the `--fix` option.
```

Ruff immediately finds the unused import and even gives you an option to fix it. Follow the suggestion from the output and fix that unused `math` import automatically:

Shell

Your unused import statement is history. If you run the check on the initial unformatted `unfashionable.py` file, then you'll notice that Ruff doesn't flag the style violations that `pycodestyle` or `flake8` flagged for you. In the default setting, Ruff omits flagging most `E` type errors—but you can change that in the settings file.

And because Ruff is both a linter and an autoformatter, you don't even need to worry about it. If you run the tool's `format` command, then it'll fix all the PEP 8 style violations without you even needing to know that they were there in the first place:

Shell

With the default settings, the Ruff autoformatter will produce following formatted code:

Python `unfashionable.py`

Note that the line numbers again break later than PEP 8 recommends, but you can change this setting—and many more—in Ruff's settings.

If you want to learn more about working with linters and autoformatters, then you can take a look at Python Code Quality: Tools & Best Practices, which gives a thorough explanation of how to use some of 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 follows the style guide for Python code
- How to **use linters and autoformatters** to check your code against PEP 8 guidelines and apply some of them automatically

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 formatted for better readability. In these documents, you'll find the rest of the PEP 8 guidelines that you didn't encounter in this tutorial.

Do you have any pet peeves that are different from what PEP 8 suggests, or do you fully subscribe to the style guide for Python code? Share your opinion in the comments below.

Get Your Code Click here to download the free sample code that shows you how to write PEP 8 compliant code.

Take the Quiz: Test your knowledge with our interactive “How to Write Beautiful Python Code With PEP 8” quiz. You’ll receive a score upon completion to help you track your learning progress:

Interactive Quiz

How to Write Beautiful Python Code With PEP 8

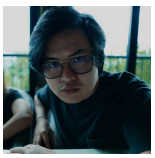
In this quiz, you'll test your understanding of PEP 8, the Python Enhancement Proposal that provides guidelines and best practices on how to write Python code. By working through this quiz, you'll revisit the key guidelines laid out in PEP 8 and how to set up your development environment to write PEP 8 compliant Python code.

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**

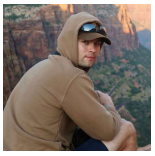

```
# How to merge two dicts
# in Python 3.5+

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

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

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

About **Jasmine Finer**

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

Geir Arne

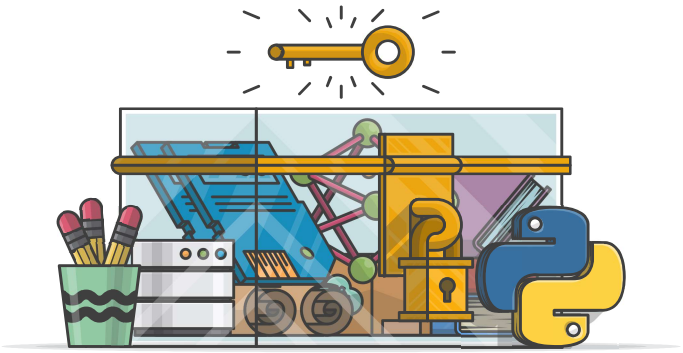
Joanna

Kate

Martin

Level Up Your Python Skills »

Master Real-World Python Skills
With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!