Chitrank Dixit
January 25th, 2018

PYTHON +1

# PEP-8 Tutorial: Code Standards in Python

With this beginner tutorial, you'll start to explore PEP-8, Python's style guide, so that you can start formatting your code correctly to maximize its readability!

PEP-8 or the Python Enhancement Proposal presents some of the key points that you can use to make your code more organized and readable. As Python creator Guido Van Rossum says:

> *The code is read much more often than it is written.*

In this post, you'll start to explore PEP-8 with some code examples! You'll cover the following topics:

- You'll first get introduced to PEP-8, what it is and why you need it;

programmers. Should you use tabs or spaces? You'll discover the anser in this section;
- You maybe wouldn't expect it, but there are guidelines for maximum line length;
- Also, there is a proposed way of handling blank lines;
- Next, whitespaces in expressions and statements is something that you can also easily tackle as a beginner;
- What is encoding and why would you need it when working with Python? What is the default encoding for Python3? The source line encoding section will tackle all of this.
- You probably do imports frequently when you're coding. This section will tackle topics such as the order of your imports, absolute and relative imports, and wildcard imports, etc.;
- Documentation is essential for keeping track of all aspects of an application and improves the overall quality of the end product. Comments are essential here!
- Do you know model level dunder names? These are particularly helpful in docstrings!
- Lastly, you'll also learn more about naming conventions: you'll discover how you can come up with function names, what type of naming style would you typically use, and much more;
- Is Your Code PEP-8 Compliant? Is definitely a question you should ask yourself. That's why the last section covers some tools to help you to check your code to see whether or not it adheres to the guidelines presented in this post and the many more that you didn't cover here!

The Python programming language has evolved over the past year as one of the most favourite programming languages. This language is relatively easy to learn than most of the programming languages. It is a multi-paradigm, it has lots of open source modules that add up the utility of the language and it is gaining popularity in data science and web development community.

However, you can use the benefits of Python only when you know how to express better with your code. Python was made with some goals in mind, these goals can be seen when you type `import this`.

```
import this


The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
```

```
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The above are the 20 principles that Python programming uses. You also see "Readability Counts" in the output above, which should be your main concern while writing code: other programmers or data scientists should understand and should be able to contribute to your code so that it can solve the task at hand.

The following sections will give you some more insights into how you can accomplish the above!

### Indentation

When programming in Python, indentation is something that you will definitely use. However, you should be careful with it, as it can lead to syntax errors. The recommendation is therefore to use 4 spaces for indentation. For example, this statement uses 4 spaces of indentation:

```
if True:
    print("If works")
```

And also this `for` loop with `print` statement is indented with 4 spaces:

```
print(element)
```

When you write a big expression, it is best to keep the expression vertically aligned. When you do this, you'll create a "hanging indent".

Here are some examples of the hanging indent in big expressions, which show some variations of how you can use it:

1.

```
value = square_of_numbers(num1, num2,
                          num3, num4)
```

2.

```
def square_of_number(
     num1, num2, num3,
     num4):
 return num1**2, num2**2, num3**2, num4**2
```

3.

```
value = square_of_numbers(
              num1, num2,
              num3, num4)
```

4.

```
    "Rama",
    "John",
    "Shiva"
]
```

5.

```
dict_of_people_ages = {
    "ram": 25,
    "john": 29,
    "shiva": 26
}
```

Every developer, working with Python or another programming language, asks him or herself the question at some point whether to use tabs or spaces for indentation. The difference between tabs and spaces is an ongoing discussion in the community. Check out, for example, this article.

Generally, spaces are the preferred indentation means but if you find some Python scripts already using the tabs, you should go on doing indentation with tabs. Otherwise, you should change the indentation of all the expressions in your script with spaces.

**Note** that Python 3 doesn't allow mixing tabs and spaces for indentation. That's why you should choose one of the two and stick with it!

#### Maximum Line Length

your Python code.

Following this target number has many advantages. A couple of them are the following:

- It is possible to open files side by side to compare;
- You can view the whole expression without scrolling horizontally which adds to better readability and understanding of the code.

Comments should have 72 characters of line length. You'll learn more about the most common conventions for comments later on in this tutorial!

In the end, it is up to you what coding conventions and style you like to follow if you are working in a small group and it is acceptable for most of the developers to divert from the maximum line length guideline. However, if you are making or contributing to an open source project, you'll probably want and/or need to comply with the maximum line length rule that is set out by PEP-8.

While using the `+` operator, you can better use a proper line break, which makes your code easier to understand:

| You should use... | You should avoid... |
| --- | --- |

```
            B +                                 +   B
            C)                                  +   C)
```

### Blank Lines

In Python scripts, top-level function and classes are separated by two blank lines. Method definitions inside classes should be separated by one blank line. You can see this clearly in the following example:

```
class SwapTestSuite(unittest.TestCase):
    """
        Swap Operation Test Case
    """
    def setUp(self):
        self.a = 1
        self.b = 2

    def test_swap_operations(self):
        instance = Swap(self.a,self.b)
        value1, value2 =instance.get_swap_values()
        self.assertEqual(self.a, value2)
        self.assertEqual(self.b, value1)


class OddOrEvenTestSuite(unittest.TestCase):
    """
        This is the Odd or Even Test case Suite
    """
    def setUp(self):
        self.value1 = 1
        self.value2 = 2
```

```
        instance1 = OddOrEven(self.value1)
        instance2 = OddOrEven(self.value2)
        message1 = instance1.get_odd_or_even()
        message2 = instance2.get_odd_or_even()
        self.assertEqual(message1, 'Odd')
        self.assertEqual(message2, 'Even')
```

The classes `SwapTestSuite` and `OddOrEvenTestSuite` are separated by two blank lines, whereas the method definitions, such as `.setUp()` and `.test_swap_operations()` only have one blank line to separate them.

### Whitespaces in Expressions and Statements

You should try to avoid whitespaces when you see your code is written just like in the following examples:

**You should use...**

```
func(data, {pivot: 4})
```

```
indexes = (0,)
```

```
if x == 4: print x, y; x, y = y, x
```

```
dct['key'] = lst[index]
```

```
x = 1
y = 2
long_variable = 3
```

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```

```
i = i + 1
submitted += 1
x = x2 - 1
hypot2 = xx + yy
c = (a+b)  (a-b)
```

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...
```

```
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

```
FILES = ('setup.cfg',)


FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
```

These examples were taken from PEP-8.

### Source File Encoding

A computer cannot store "letters", "numbers", "pictures" or anything else; It can only store and work with bits, which can only have binary values: `yes` or `no`, `true` or `false`, `1` or `0`, etc. As you already know, a computer works with electricity; This means that an "actual" bit is the presence or absence of a blip of

with `1` and `0`.

To use bits to represent anything at all besides bits, you need a set of rules. You need to convert a sequence of bits into something like letters, numbers and pictures using an encoding scheme or encoding. Examples of encoding schemes are ASCII, UTF-8, etc:

- The American Standard Code for Information Interchange (ASCII) is the most common format for text files in computers and on the Internet. In this type of files, each alphabetic, numeric, or special character is represented with a 7-bit binary number (a string of seven 0s or 1s).
- Unicode Worldwide Character Standard, or Unicode in short, is a system for "the interchange, processing, and display of the written texts of the diverse languages of the modern world". In short, Unicode is designed to accommodate all of the world's known writing systems. Unicode currently employs three different encodings to represent Unicode character sets: UTF-8, UTF-16 and UTF-32.
  - UTF-16 is a Unicode encoding that is variable-length: code points are encoded with one or two 16-bit code units.
  - UTF-8 is another type of Unicode variable-length encoding, using one to four 8-bit bytes.
  - UTF-32 is a fixed-length encoding that uses exactly 32 bits per Unicode code point.

post.

Now, why is this important?

You will have found out that strings are among the most commonly used data types in Python. As you can expect, there will be a time when you want to work with strings that either contain or entirely made up of characters that are not part of the standard ASCII set. After all, it could be that you have to work with texts that contain accented characters, such as `á`, `ž`, `ç` etc.

Now, in Python 3, UTF-8 is the default source encoding. But, for those of you who use Python 2, you probably will already know that the default there is ASCII.

But what then if you have a string that contains a non-ASCII character, like `"Flügel"`?

When you reference the string in Python 2, you'll get the following:

```
>>> s
'Fl\xfcgel'
```

This doesn't look like your string! What happens when you print it?

```
Flügel
```

Printing gave you the value that you assigned to the variable. The non-ASCII character `ÃŒ` was encoded. That's why you got back `\xfc` when you referenced the string. To handle this, you can make use of the `.encode()` and `.decode()` string methods. The former returns an 8-bit string version of the Unicode string, encoded in the requested encoding, while the latter interprets the string using the given encoding.

### Imports

Importing libraries and/or modules is something that you'll often do when you're working with Python for data science. As you might already know, you should always import libraries at the start of your script.

**Note** that if you do many imports, you should make sure to state each import on a single line.

Take a look at the following table to understand this a bit better:

| You should use... | You should avoid... |
| --- | --- |
| `from config import settings` | `import os, sys` |

```
import os
import sys
```

Additionally, you should take into account that there is an order that you need to respect when you're importing libraries. In general, you can follow this order:

1. Standard library imports.
2. Related third-party imports.
3. Local application/library specific imports.

#### Absolute and Relative Imports

Next, it's good to know the difference between absolute and relative imports. In general, absolute imports are preferred in Python, as it adds up more readability. However, as your application becomes more complex, you can go on using the relative imports also. Implicit relative imports should never be used and have been removed in Python 3.

But what are these absolute and relative imports?

- An absolute import is an import that uses the absolute path of the function or class, separated by `.`. For example,

```
import sklearn.linear_model.LogisticRegression
```

position where your Python file exists. You could use this type of import if your project structure is growing, as it will make your project more readable. That means that, if you have a Python project structure like the following:

```
.
├── __init__.py
├── __init__.pyc
├── __pycache__
│   ├── __init__.cpython-35.pyc
│   ├── bubble_sort.cpython-35.pyc
│   ├── selection_sort.cpython-35.pyc
├── bubble_sort.py
├── heap_sort.py
├── insertion_sort.py
├── insertion_sort.pyc
├── merge_sort.py
├── merge_sort.pyc
├── quick_sort.py
├── radix_sort.py
├── selection_sort.py
├── selection_sort.pyc
├── shell_sort.py
├── tests
│   ├── test1.py
```

You could use a relative import to import the bubble sort algorithm `BubbleSort`, stored in `bubble_sort.py` in `test1`. That would look like this:

```
from ..bubble_sort import BubbleSort
```

you can check PEP 328.

#### Wildcard Imports

Lastly, you should try to avoid wildcard imports, because they do not add to the readability; You have no view on which classes, methods or variables you are using from your module, for example:

```
from scikit import *
```

### Comments

Comments are used for in-code documentation in Python. They add to the understanding of the code. There are lots of tools that you can use to generate documentation, such as comments and docstrings, for your own module. Comments should be more verbose so that when someone reads the code, the person would get the proper understanding of the code and how it is being used with other pieces of the code.

Comments start with the `#` symbol. Anything written after the hashtag does not get executed by the interpreter. For example, the following code chunk will only give back `"This is a Python comment"`.

```
# This is a Python single line comment
print("This is a Python comment")
```

**Remember**: In the previous section, you read that comments should have 72 characters of line length!

This being said, there are three types of comments:

- You use block comments to explain code that is more complex or unfamiliar to others. These are typically longer-form comments and they apply to some or all of the code that follows. Block comments are indented at the same level as the code. Each line of a block comment begins with the hashtag `#` and a single space. If you need to use more than one paragraph, they should be separated by a line that contains a single `#`.

Take a look at the following excerpt, taken from the `scikit-learn` library, to understand what these comments look like:

```
if Gram is None or Gram is False:
     Gram = None
     if copy_X:
         # force copy. setting the array to be fortran-ordered
         # speeds up the calculation of the (partial) Gram matrix
         # and allows to easily swap columns
         X = X.copy('F')
```

- You should use inline comments sparingly, even though they can be effective when you need to explain some parts of your code. They also might help you to remember what a specific line of code means or can come in handy when you're collaborating with someone who is unfamiliar with all aspects

statement, following the code itself. These comments also start with `#` and a single space.

For example:

```
counter = 0  # initialize the counter
```

- You write documentation strings or docstrings at the start of public modules, files, classes and methods. These type of comments start with `"""` and end with `"""`:

```
"""
    This module is intended to provide functions for scientific computing
"""
```

### Module Level dunder Names

Now that you have read what docstrings are, you should also know about Module Level dunders, or names with two leading and two trailing underscores, are very effective in Python. These are the special names that Python defines so that it won't get conflicted to the user-defined functions or names. For more information, you can check out this article.

A module level dunder like (`__all__`, `__author__`, `__version__`) should be placed at the module main docstring and should be before all

imports before any other code, except the docstrings:

```
"""
    Algos module consists of all the basic algorithms and their implementation
"""

from __future__ import print

__all__ = ['searching', 'sorting']
__version__ = '0.0.1'
__author__ = 'Chitrank Dixit'

import os
import sys
```

**Tip**: check out the following conventions for writing docstrings.

### Naming Conventions

When you program in Python, you'll most certainly make use of a naming convention, a set of rules for choosing the character sequence that should be used for identifiers which denote variables, types, functions, and other entities in source code and documentation.

If you're not sure what naming styles are out there, consider the following ones:

- `b` or a single lowercase letter;
- `B` or a single uppercase letter;

- `UPPERCASE`
- `lower_case_with_underscores`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords`, which is also known as `CapWords`, `CamelCase` or `StudlyCaps`.
- `mixedCase`
- `Capitalized_Words_With_Underscores`
- `_single_leading_underscore`: weak "internal use" indicator. for example, `from M import *` does not import objects whose name starts with an underscore.
- `single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, for example, `Tkinter.Toplevel(master, class_='ClassName')`
- `__double_leading_underscore`: when naming a class attribute, invokes name mangling (inside class `FooBar`, `__boo` becomes `_FooBar__boo`).
- `__double_leading_and_trailing_underscore__`: "magic" objects or attributes that live in user-controlled namespaces. For example, `__init__`, `__import__` or `__file__`. You should never invent such names, but only use them as documented.

#### General Naming Conventions

The following table shows you some general guidelines on how to name your identifiers:

| | |
|---|---|
| Module | lowercase |
| Class | CapWords |
| Functions | lowercase |
| Methods | lowercase |
| Type variables | CapWords |
| Constants | UPPERCASE |
| Package | lowercase |

- Do not use 'l', 'O' or 'I' as a single variable name: these characters look similar to zero (`0`) and (`1`) in some fonts.
- Generally, it's good to use short names if possible. In some cases, you can use underscores to improve readability.

If you want to know more about the exceptions to the general naming conventions, check out this article.

#### Is Your Code PEP-8 Compliant?

After learning more about PEP-8, you're probably wondering how you can check whether or not your code actually complies with these guidelines (and more that haven't been covered in this tutorial!).

you should definitely consider looking into the handy `pep8` module, the `coala` package and some of the other alternatives that are described in the next sections!

#### Python `pep8` Package

The `pep8` package intends to search for PEP-8 incompatibility in your code and suggests changes that you can make to make your code follow the PEP-8.

you can install pep8 module using `pip`.

```
$ pip install pep8
```

For example,

```
$ pep8 --first optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

You can also view the source code where the incompatibility is found with the `--show-source` argument:

```
testsuite/E40.py:2:10: E401 multiple imports on one line
import os, sys
         ^
    Imports should usually be on separate lines.

    Okay: import os\nimport sys
    E401: import sys, os
```

Or you can display how often each error was found by adding `--statistics`:

```
$ pep8 --statistics -qq Python-2.5/Lib
232     E201 whitespace after '['
599     E202 whitespace before ')'
631     E203 whitespace before ','
842     E211 whitespace before '('
2531    E221 multiple spaces before operator
4473    E301 expected 1 blank line, found 0
4006    E302 expected 2 blank lines, found 1
165     E303 too many blank lines (4)
325     E401 multiple imports on one line
3615    E501 line too long (82 characters)
612     W601 .has_key() is deprecated, use 'in'
1188    W602 deprecated form of raising exception
```

**Tip**: also make sure to check out other modules, such as `flake8`, `autopep8` or `pylint`!

#### Analyzing Your Code with `coala`

`coala` provides linting and fixing for all the languages but you are more concerned about Python programming here, you can

```
$ pip3 install coala-bears
```

In the code chunk above, you see that you actually install `coala-bears`: `bears` are plugins or simple modules that extend the capability of your `coala` and vary from language to language. In this case, you want to use `pep8bear`, which finds the PEP-8 incompatible code and fixes that in place. You should definitely consider using this to check your Python code.

```
$ coala -S python.bears=PEP8Bear python.files=\*\*/\*.py \
python.default_actions=PEP8Bear:ApplyPatchAction --save
# other output ...
Executing section python...
[INFO][11:03:37] Applied 'ApplyPatchAction' for 'PEP8Bear'.
[INFO][11:03:37] Applied 'ApplyPatchAction' for 'PEP8Bear'.
```

#### pep8online: Check Your Python Code Online

On top of the handy `pep8` module and the `coala` package, you can also check whether your Python code is PEP-8 compliant by going to pep8online. This site has an online editor which allows you to just paste in your code, press the "Check code" button! As a result, you'll get some feedback on what you need to improve. Nice and handy!

### Conclusion

quality due to anxiety of releasing features faster. However, the practices that were described in this tutorial -and the many more that weren't covered here- should be part of your develop-staging-test-deploy cycle. This benefits everyone working on the project to understand and most of the times making changes in the code can be done without digging deep and understanding the code by starting debuggers. If you are working on an open-source project, your contributors would find PEP-8 a bliss and would understand your code better as this is the universal standard, each Python developer follows this.

Now that you have gone through this tutorial, it's a very good idea to check out the PEP-8 for yourself! There is much more to discover.

If you have any more tips on how to comply to PEP-8 or if you think we left something important out of this article, don't hesitate to let us know @DataCamp.