Djangocong 2018 - Lille

Les outils pour améliorer votre code python

Par Stéphane "Twidi" Angel (twidi sur Github, Twitter...)

Sponsorisé par isshub.io

PEP

Python **E**xtension **P**roposal

- revue
- discutée
- approuvée
- intégrée
- ou rejetée

https://www.python.org/dev/peps/

PEP 20 -- The Zen of Python

```
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 that way may not be obvious at first unless you're Dutch.

Now is better than never.

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!
```

Coding party

Écrivons un peu de python dans un fichier `code.py`.

code.py

```
def MySum(x,x):
    return x+y
print(MySum(1,2))
```

shell

```
$ python code.py
  File "code.py", line 4
    def MySum(x,x):
    ^
SyntaxError: duplicate argument 'x' in function definition
```

flake8

```
Flake8: Your Tool For Style Guide Enforcement
```

shell

```
$ pip install flake8
```

pycodestyle

```
Simple Python style checker in one Python file
```

pyflakes

```
A simple program which checks Python source files for errors
```

mccabe

```
Python code complexity micro-tool
```

shell

```
$ flake8 code.py
code.py:1:1: F831 duplicate argument 'x' in function definition
code.py:2:14: F821 undefined name 'y'
```

code.py

```
def MySum(x,x):
    return x+y
print(MySum(1,2))
```

shell

```
$ flake8 code.py
code.py:1:1: F831 duplicate argument 'x' in function definition
code.py:1:13: E231 missing whitespace after ','
code.py:2:14: F821 undefined name 'y'
code.py:3:1: E305 expected 2 blank lines after class or function
definition, found 0
code.py:3:14: E231 missing whitespace after ','
```

PEP 8 -- Style Guide for Python Code

```
As PEP 20 says, “Readability counts”.

A style guide is about consistency.

Consistency with this style guide is important.

Consistency within a project is more important.

Consistency within one module or function is the most important.
```

```
However, know when to be inconsistent.

Sometimes style guide recommendations just aren't applicable.

When in doubt, use your best judgment.

Look at other examples and decide what looks best.

And don’t hesitate to ask!
```

---

http://pep8.org

- **Code layout**
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Should a line break before or after a binary operator?
  - Blank Lines
  - Source File Encoding
  - Imports
  - Module level dunder names

- **String Quotes**
- **Whitespace in Expressions and Statements**
- **When to use trailing commas**
- **Comments**
  - Block Comments
  - Inline Comments
  - Documentation Strings

- **Naming Conventions**
  - Overriding Principle
  - Descriptive: Naming Styles
  - Prescriptive: Naming Conventions
    - Names to Avoid
    - ASCII Compatibility
    - Package and Module Names
    - Class Names
    - Type variable names
    - Exception Names
    - Global Variable Names
    - Function Names
    - Function and method arguments

        - Method Names and Instance Variables
        - Constants
        - Designing for inheritance

- **Public and internal interfaces**
- **Programming Recommendations**
    - Function Annotations

Correction

shell

```
$ flake8 code.py
code.py:1:1: F831 duplicate argument 'x' in function definition
code.py:1:13: E231 missing whitespace after ','
code.py:2:14: F821 undefined name 'y'
code.py:3:1: E305 expected 2 blank lines after class
or function definition, found 0
code.py:3:14: E231 missing whitespace after ','
```

code.py

```
def MySum(x,x):
    return x+y
print(MySum(1,2))
```

code.py - corrigé

```
def MySum(x, y):
    return x+y


print(MySum(1, 2))
```

flake8 plugins

flake8-docstrings

```
Extension for flake8 which uses pydocstyle to check docstrings
```

pydocstyle

```
A static analysis tool for checking compliance with Python
docstring conventions.
Supports most of PEP 257 out of the box, but it should not be
considered a reference implementation.
```

PEP 257 -- Docstring Conventions

```
The aim of this PEP is to standardize the high-level structure of
docstrings:

what they should contain, and how to say it (without touching on
any markup syntax within docstrings).

The PEP contains conventions, not laws or syntax.
```

> A universal convention supplies all of maintainability, clarity,consistency, and a foundation for good programming habits too.What it doesn't do is insist that you follow it against your will.That's Python!"
>
> —Tim Peters on comp.lang.python, 2001-06-16

Specification

- What is a Docstring?
- One-line Docstrings
- Multi-line Docstrings
- Handling Docstring Indentation

flake8-docstrings

shell

```
$ pip install flake8-docstrings
```

```
$ flake8 code.py
code.py:1:1: D100 Missing docstring in public module
code.py:1:1: D103 Missing docstring in public function
```

code.py

```
'a simple script to add two numbers'
def MySum(x, y):
    'add x and y'
    return x+y


print(MySum(1, 2))
```

shell

```
$ flake8 code.py
code.py:1:1: D300 Use """triple double quotes"""
code.py:1:1: D400 First line should end with a period
code.py:2:1: E302 expected 2 blank lines, found 0
code.py:2:1: D300 Use """triple double quotes"""
code.py:2:1: D400 First line should end with a period
code.py:2:1: D403 First word of the first line should be properly
capitalized
```

code.py

```
"""A simple script to add two numbers."""


def MySum(x, y):
    """We add x and y."""
    return x+y


print(MySum(1, 2))
```

pylint

```
Pylint is a tool that checks for errors in Python code, tries to
enforce a coding standard and looks for code smells.

It can also look for certain type errors, it can recommend
suggestions about how particular blocks can be refactored and can
offer you details about the code's complexity.
```

shell

```
$ pip install pylint
```

shell

```
$ pylint code.py
No config file found, using default configuration
************* Module code
C:  4, 0: Function name "MySum" doesn't conform to snake_case
naming style (invalid-name)
C:  4, 0: Argument name "x" doesn't conform to snake_case naming
style (invalid-name)
C:  4, 0: Argument name "y" doesn't conform to snake_case naming
style (invalid-name)


--------------------------------------------------------------------
Your code has been rated at 0.00/10
```

code.py

```
"""a simple script to add two numbers."""


def my_sum(first, second):
    """We add first and second."""
    return first+second


print(my_sum(1, 2))
```

pylint.extensions.docparams

```
Parameter documentation checker
```

```
If you document the parameters of your functions, methods and
constructors and their types systematically in your code this
optional component might be useful for you.

Sphinx style, Google style, and Numpy style are supported.
```

Sphinx style

```
:param first: The first number to add
:type first: int
:param second: The second number to add
:type second: int
:returns: The result of the computation of first and second
:rtype: int
```

Google style

```
Args:
    first (int): The first number to add
    second (int): The second number to add

Returns:
    int: The result of the computation of first and second
```

Numpy style

```
Parameters
----------
first: int
    The first number to add
second: int
    The second number to add

Returns
-------
int
    The result of the computation of first and second
```

Installation

shell

```
$ pylint --generate-rcfile > .pylintrc
```

.pylintrc

```
[MASTER]
load-plugins=pylint.extensions.docparams
[DESIGN]
accept-no-param-doc=no
```

shell

```
$ pylint code.py
Using config file .pylintrc
************* Module code
W:  4, 0: "first, second" missing in parameter documentation
(missing-param-doc)
W:  4, 0: "first, second" missing in parameter type documentation
(missing-type-doc)


--------------------------------------------------------------------
Your code has been rated at 3.33/10
```

code.py

```
"""a simple script to add two numbers."""


def my_sum(first, second):
    """We add first and second.

    Parameters
    ----------
    first: int
        The first number to add
    second: int
        The second number to add
```

```
    -------
    int
        The result of the computation of first and second
    """
    return first+second


print(my_sum(1, 2))
```

flake8, suite

shell

```
$ flake8 code.py
code.py:4:1: D413 Missing blank line after last section
```

code.py

```
"""a simple script to add two numbers."""


def my_sum(first, second):
    """We add first and second.

    Parameters
    ----------
    first: int
        The first number to add
    second: int
        The second number to add
```

```
    -------
    int
        The result of the computation of first and second

    """
    return first+second


print(my_sum(1, 2))
```

MAIS...

Quand même, ça serait bien si...

Vraiment bien si...

On n'avait pas à...

Penser à tout ça...

Et qu'on avait...

Les auto-formateurs

Pour

- gagner du temps lors du code
- gagner du temps lors de la revue de code
- éviter les prises de bec lors de la revue de code
- éviter les débats interminables
- ne penser qu'au code
- gagner du temps
- gagner du temps
- gagner du temps

On recommence

code.py

```
def MySum(x,y):
    return x+y
print(MySum(1,2))
```

autopep8

```
A tool that automatically formats Python code to conform to the PEP
8 style guide.
```

```
It uses the pycodestyle utility to determine what parts of the code
needs to be formatted. autopep8 is capable of fixing most of the
formatting issues that can be reported by pycodestyle.
```

shell

```
$ pip install autopep8
```

shell

```
$ autopep8 code.py > code2.py
```

code2.py

```
def MySum(x, y):
    return x+y


print(MySum(1, 2))
```

yapf

Yet **a**nother **P**ython **f**ormatter

```
A formatter for Python files
```

```
The ultimate goal is that the code YAPF produces is as good as the
code that a programmer would write if they were following the style
guide.
```

shell

```
$ pip install yapf
```

shell

```
$ yapf code.py > code2.py
```

code2.py

```
def MySum(x, y):
    return x + y


print(MySum(1, 2))
```

black

```
The uncompromising Python code formatter
```

```
By using it, you agree to cede control over minutiae of hand-
formatting.

In return, Black gives you speed, determinism, and freedom from
pycodestyle nagging about formatting. You will save time and mental
energy for more important matters.
```

shell

```
$ pip install black
```

shell

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

code.py

```
def MySum(x, y):
    return x + y

print(MySum(1, 2))
```

Autres exemples (black)

avant

```
l = [1,
     2,
     3,
]
```

après

```
l = [1, 2, 3]
```

avant

```
TracebackException.from_exception(exc, limit, lookup_lines,
capture_locals)
```

après

```
TracebackException.from_exception(
    exc, limit, lookup_lines, capture_locals
)
```

avant

```
def very_important_function(
    template: str, *variables, file: os.PathLike, debug: bool =
False):
    """Applies `variables` to the `template` and writes to
`file`."""
    with open(file, 'w') as f:
        ...
```

après

```
def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to
`file`."""
    with open(file, "w") as f:
        ...
```

MAIS... les docstrings ???

- **Rien.**
- **Nada.**
- **Que tchi.**

Les éditeurs/IDE

- pycharm
- sublime
- vscode
- atom
- vim
- emacs
- & autres, surement

Example PyCharm

```
def MySum(x, y):
    """

    Parameters
    ----------
    x :
    y :

    Returns
    -------

    """
```

```
print(MySum(1, 2))
```

isort

```
Sort your python imports for you so you don't have to.
```

shell

```
$ pip install isort
```

avant

```
from my_lib import Object

print("Hey")

import os

from my_lib import Object3

from my_lib import Object2

import sys

from third_party import lib15, lib1, lib2, lib3, lib4, lib5, lib6,
```

```
import sys

from __future__ import absolute_import

from third_party import lib3

print("yo")
```

après

```
from __future__ import absolute_import

import os
import sys

from third_party import (lib1, lib2, lib3, lib4, lib5, lib6, lib7,
lib8,
                         lib9, lib10, lib11, lib12, lib13, lib14,
lib15)

from my_lib import Object, Object2, Object3

print("Hey")
```

Typing annotations

PEP 3107 -- Function Annotations

```
This PEP introduces a syntax for adding arbitrary metadata
annotations to Python functions.
```

```
def haul(item: Haulable, *vargs: PackAnimal) -> Distance:
    ...
```

```
def compile(source: "something compilable",
            filename: "where the compilable thing comes from",
            mode: "is this a single statement or a suite?"):
    ...
```

PEP 484 -- Type Hints

> [...]the community would benefit from a standard vocabulary and baseline tools within the standard library.
> This PEP introduces a provisional module to provide these standard definitions and tools, along with some conventions for situations where annotations are not available.

```python
def greeting(name: str) -> str:
    return 'Hello ' + name

msg = greeting('John')  # type: str
```

PEP 526 -- Syntax for Variable Annotations

```
This PEP aims at adding syntax to Python for annotating the types of
variables.
```

```
msg: str = greeting('John')
```

> **Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.**

> **Python restera un langage dynamiquement typé, et les auteurs n'ont aucun désir de rendre obligatoires les indications de type, jamais, même par convention.**

Annotations

code.py

```
def MySum(x: str, y: list) -> float:
    return x + y


print(MySum(1, 2))
```

shell

```
$ python code.py
3
```

mypy

```
Optional Static Typing for Python
```

```
Mypy is an experimental optional static type checker for Python
that aims to combine the benefits of dynamic (or "duck") typing and
static typing.

Mypy combines the expressive power and convenience of Python with a
powerful type system and compile-time type checking.
```

shell

```
$ pip install mypy
```

code.py

```python
def MySum(x: str, y: list) -> float:
    return x + y


print(MySum(1, 2))
```

shell

```
$ mypy code.py
code.py:2: error: Incompatible return value type (got "str",
expected "float")
code.py:2: error: Unsupported operand types for + ("str" and
"List[Any]")
```

```
"int"; expected "str"
code.py:5: error: Argument 2 to "MySum" has incompatible type
"int"; expected "List[Any]"
```

code.py

```
def MySum(x: int, y: int) -> int:
    return x + y

print(MySum(1, "foo8"))
```

shell

```
$ mypy code.py
code.py:5: error: Argument 2 to "MySum" has incompatible type
"str"; expected "int"
```

pyre

```
A performant type-checker for Python 3
```

```
We believe statically typing what are essentially fully dynamic
languages gives our code stability and improves developer
productivity.
```

shell

```
$ pip install pyre-check
```

code.py

```python
def MySum(x: str, y: list) -> float:
    return x + y


print(MySum(1, 2))
```

shell

```
$ pyre check
 ƛ Found 2 type errors!
code.py:2:4 Incompatible return type [7]: Expected `float` but got
`str`.
code.py:2:19 Incompatible parameter type [6]: Expected `str` but
```

Générateur d'annotations de type

- ancienne base de code
- fonctionne en analysant le code tourner (tests...)

pyannotate

```
Auto-generate PEP-484 annotations
```

MonkeyType

```
A system for Python that generates static type annotations by
collecting runtime types
```

Final code

avant

```
def MySum(x,x):
    return x+y
print(MySum(1,2))
```

après

```
"""a simple script to add two numbers."""


def my_sum(first, second):
    """We add first and second.

    Parameters
    ----------
    first: int
        The first number to add
    second: int
        The second number to add
```

```
    -------
    int
        The result of the computation of first and second

    """
    return first+second


print(my_sum(1, 2)
```

Déroulement classique

1. Configuration éditeur / IDE
    1. docstrings
    2. analyse de code
    3. ré-écriture automatique

2. Coder

3. Lancer les checks avant chaque commit (ou via commithook)
    1. lint
        1. black --check
        2. flake8
        3. pylint
        4. mypy
    2. tests

4. Si OK
    1. push
    2. pull/merge-request...

5. Revue de code
    1. **ENJOY**. Adieu:
        1. les pertes de temps,
        2. les amitiés rompues,
        3. Les jetages décran par la fenêtre,
        4. etc...

Conclusion

JUST DO IT

Vous et votre équipe me remercierez à la prochaine conf ;)

Example concret

mixt: https://github.com/twidi/mixt/

"Write html components directly in python and you have a beautiful but controversial MIXTure."

Merci

- au **public**, pour son soutien indéfectible. **You're awesome**.
- aux **organisateurs**. **You're awesome**.
- à **ma chérie** de m'avoir laisser partir ce week-end (je rigole chérie). **You're awesome**.
- à **mes parents**, sans qui... blablabla. **You're awesome**.
- **https://isshub.io**

```
Take back control of your Github issues, pull requests, and
notifications
```

Les slides seront publiés sur https://twidi.github.io/