Le PEP8, en résumé

Internet, c'est la culture du TL;DR, donc plutôt je vais faire une petite synthèse des trucs les plus importants du PEP8, comme ça si vous avez la flemme de le lire, au moins vous aurez l'essentiel.

Ce texte liste les règles stylistiques recommandées, invitant toute la communauté Python à écrire un code de la même façon.

Je vais également y ajouter des éléments de style qui ne sont pas dedans, mais que j'ai pu constater comme étant les choix les plus courants dans les sources que j'ai pu lire.

Espaces

Les opérateurs doivent être entourés d'espaces.

Il faut faire ceci :

```
variable = 'valeur'

ceci == cela

1 + 2
```

Et non:

```
variable='valeur'

ceci==cela

1+2
```

Il y a deux exceptions notables.

La première étant qu'on groupe les opérateurs mathématiques ayant la priorité la plus haute pour distinguer les groupes :

```
a = x*2 - 1
b = x*x + y*y
c = (a+b) * (a-b)
```

La seconde est le signe = dans la déclaration d'arguments et le passage de paramètres :

```
def fonction(arg='valeur'): #  ça c'est ok
    pass

resultat = fonction(arg='valeur') #  ça aussi
```

On ne met pas d'espace à l'intérieur des parenthèses, crochets ou accolades.

Oui :

```
2 * (3 + 4)
```

```
def fonction(arg='valeur'):

{str(x): x for x in range(10)}

val = dico['key']
```

Non :

```
2 * ( 3 + 4 )

def fonction( arg='valeur' ):

{ str(x): x for x in range(10) }

val = dico[ 'key' ]
```

On ne met pas d'espace avant les deux points et les virgules, mais après oui.

Oui :

```
def fonction(arg1='valeur', arg2=None):

dico = {'a': 1}
```

Non :

```
def fonction(arg='valeur' , arg2=None) :

dico = {'a' : 1}
```

Lignes

Une ligne doit se limiter à 79 charactères. Cette limite, héritée des écrans touts petits, est toujours en vigeur car il est plus facile de scanner un code sur une courte colonne qu'en faisant des aller-retours constant.

Si une ligne est trop longue, il existe plusieurs manières de la racourcir :

```
foo = la_chose_au_nom_si_long_quelle_ne_tient_pas_sur(
          une,
          carte,
          de,
          munchkin)
```

Ici l'indentation entre le nom de la fonction et des paramètres est légèrement différente pour mettre en avant la distinction.

Une variante :

```
foo = la_chose_au_nom_si_long_quelle(ne, tient, pas, sur, carte
                                     une, de, munchkin)
```

Si c'est un appel chainé, on peut utiliser \ pour mettre à la ligne :

```
queryset = ModelDjangoALaNoix.objects\
                             .filter(banzai=True)\
```

```
                                .exclude(chawarma=False)
```

Si c'est une structure de données, on peut se la jouer langage fonctionel de la vibes du flex :

```
chiffres = [
    1, 2, 3,
    4, 5, 6,
]

contacts = {
    'Cleo': (),
    'Ramses': (
        ('maman', '0248163264'),
        ('papa', '01234567890'),
        ('mamie', '55555555'),
        ('momie', '066642424269')
    )
}
```

Séparer les fonctions et les classes à la racine d'un module par 2 lignes vides. Les méthodes par 1 ligne vide. Parfois je triche et je fais 3 et 2 au lieu de 2 et 1.

Les imports de plusieurs modules doivent être sur plusieurs lignes :

```
import sys
import os
```

Et non :

```
import sys, os
```

Bien sûr, ce n'est pas valable pour `from x import z`.

Souvenez-vous qu'on peut utiliser les parenthèses pour diviser de longues lignes. Par exemple :

```
from minibelt import (dmerge, get, iget, normalize,
                      chunks, window, skip_duplicates, flatten)
```

Idem pour les chaînes très longues :

```
s = ("Les chaînes Python sont automatiquement"
     "concaténées par la VM si elles sont "
     "uniquement séparées par des espaces "
     "ou sauts de lignes.")
```

Pour en revenir aux lignes d'import, on doit les ordonner ainsi :

- Import de module > import du contenu du module
- Import de la lib standard > import de libs tierces parties > import de votre projet

Exemple :

```
import os  # import module de la lib standard
import sys # on groupe car même type

from itertools import islice  # import du contenu du module
from collections import namedtuple # import groupe car même type
```

```python
import requests # import lib tierce partie
import arrow # on groupe car même type

from django.conf import settings # tierce partie, contenu du module
from django.shortcuts import redirect # on groupe car même type

from mon_projet.mon_module import ma_bite # mon projet
```

Format du fichier

Indentation : 4 espaces. Pas de tab. C'est tout. Ce n'est pas du PEP8, c'est juste que les codes qui utilisent les tabs sont en (très très très très très très très très) grande minorité dans la communauté Python. Donc faites pas chier. Sinon on vous attend à la sortie de l'école. Tous.

Encoding : UTF8 ou ASCII (ce qui est de l'UTF8 de toute façon).

Docstrings

(c.f. PEP257)

On utilise toujours des triples quotes :

```python
def fonction_avec_docstring_courte():
    """Résumé en une ligne."""
    pass
```

Si la docstring est longue (elle peut être très très très longue si vous le souhaitez) :

```python
def fonction():
    """Résumé en une ligne suivi d'une line vide.

    Description longue de la fonction qui
    se termine par une ligne vide puis une
    triple quotes sur sa propre ligne.

    """
```

Noms de variables

Lettres seules, en minuscule : pour les boucles et les indices.

Exemple :

```python
for x in range(10):
    print(x)

i = get_index() + 12
print(ma_liste[i])
```

Lettres minuscules + underscores : pour les modules, variables, fonctions et méthodes.

```python
une_variable = 10

def une_fonction():
    return locals() or {}
```

```
class UneClasse:
    def une_methode_comme_une_autre(self):
        return globals()
```

Lettres majuscules + underscores : pour les (pseudo) constantes.

```
MAX_SIZE = 100000  # à mettre après les imports
```

Camel case : nom de classe.

```
class CeciEstUneClasse:
    def methodiquement(self):
        pass
```

Si le nom contient un acronyme, on fait une entorse à la règle :

```
class HTMLParserCQFDDDTCCMB:
    def methodiquement(self):
        pass
```

On n’utilise PAS le mixedCase.