12/09/2018

Python pour l'ingénieur

Règles de codage

Module de formation

Introduction

- Les règles de codage qui sont présentées ici sont issues du document PEP 8 « *Style Guide for Python Code* ».
  https://www.python.org/dev/peps/pep-0008

- Le document PEP 257 est également utilisé pour ce qui concerne les *docstrings*.
  https://www.python.org/dev/peps/pep-0257/

- Les documents PEP (« *Python Enhancement Proposal* ») sont des propositions d'amélioration du langage Python, qui précèdent nécessairement toute évolution ultérieure du langage.

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
```

Espaces

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

Espaces

- 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' ]
```

Espaces

- 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 caractères. Cette limite, héritée des écrans touts petits, est toujours en vigueur car il est plus facile de scanner un code sur une courte colonne qu'en faisant des allers-retours constants.
- 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,
          seule,
          ligne)
```

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

Lignes

- Une variante :

```
foo = la_chose_au_nom_si_long_quelle(ne, tient, pas, sur, une,
                                     seule, ligne)
```

- Pour un appel chainé, on peut utiliser \ pour mettre à la ligne :

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

Lignes

- Pour les structures de données :

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


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

Lignes

- Séparer les fonctions et les classes à la racine d'un module par 2 lignes vides.
- Séparer les méthodes par 1 ligne vide.

- Les imports de plusieurs modules doivent être sur plusieurs lignes :

```
import sys
import os
```

- Et non :

```
import sys, os
```

- On peut utiliser les parenthèses pour diviser de longues lignes :

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

Lignes

- Les lignes d'import doivent être ordonnées comme suit :
  - (Import de module) AVANT (import du contenu du module)
  - (Import de la lib standard) AVANT (import de libs tierces parties) AVANT (import de votre projet)

```
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 groupé car même type

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_classe # mon projet
```

Format du fichier

- Indentation : 4 espaces. Pas de tabulation (paramétrer votre éditeur pour que la tabulation soit convertie en 4 espaces).

- Encoding : UTF8
  - Il n’est pas nécessaire de déclarer l’encoding en entête du code source si vous utilisez UTF8.

Docstrings

- On utilise toujours des triples quotes :

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

- Si la docstring est longue :

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

    Description longue de la fonction qui
    se termine par une ligne vide.
    Décrire les arguments de la fonction, sa valeur
    de retour, ce qu'elle fait, etc.

    """
```

⇒ Voir https://www.python.org/dev/peps/pep-0257/#multi-line-docstrings

Noms de variables

- Pour les boucles et les indices : lettres seules, en minuscule.

```
for x in range(10):
    print(x)
i = get_index() + 12
print(ma_liste[i])
```

- Pour les modules, variables, fonctions et méthodes : lettres minuscules + underscores.

```
une_variable = 10
def une_fonction():
    return locals() or {}

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

Noms de variables

- Pour les constantes : Lettres majuscules + underscores.

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

- Nom de classe : camel case.

```
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.