

# Qualité de code et convention de codage

## Règles de codage

Les règles de codage sont un ensemble de règles à suivre pour uniformiser les pratiques de développement logiciel, diffuser les bonnes pratiques de développement et éviter les erreurs de développement "classiques" au sein d'un groupe de développeurs.

Les règles de codage s'articulent autour de plusieurs thèmes, les plus courants étant :

- Le nommage et l'organisation des fichiers du code source
- le style d'indentation
- Les conventions de nommage, ou règles de nommage
- Les commentaires et documentation du code source
- Recommandations sur la déclaration des variables
- Recommandations sur l'écriture des instructions, des structures de contrôle et l'usage des parenthèses dans les expressions.

Afin d'associer règles et qualité de code, nous pouvons ajouter :

- Utiliser un gestionnaire de version comme GitHub ou GitLab
  - o <https://github.com/>
  - o <https://gitlab.com/gitlab-org>
- Intégrer les tests tout au long du cycle de développement et utiliser un API de test.
  - o <https://www.jetbrains.com/pycharm>

## Nommage

- les identificateurs des constantes doivent être composés de majuscules ou de tirets bas.

les identificateurs des classes doivent être écrits en camel case (sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot) en commençant par une lettre majuscule.

- les identificateurs des méthodes doivent être écrits en camel case en commençant par une lettre minuscule.

La notation hongroise est, en programmation informatique, une convention de nommage des variables et des fonctions qui met en avant soit leur usage, soit leur type. Cette notation utilisée couramment consiste à faire précéder le nom de la variable d'un préfixe reflétant son type. Ce préfixe est toujours écrit en minuscule puis la première lettre suivante en majuscule. Par exemple la variable booléenne danger est préfixée par un b pour indiquer un type booléen : bDanger.

Il y a des outils de vérification comme PyChecker <http://pychecker.sourceforge.net/>

## En python

Ce site liste les règles stylistiques recommandées, invitant toute la communauté Python à écrire un code de la même façon (Le PEP8, en résumé) <http://sametmax.com/le-pep8-en-resume/>

## 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 caractères. Cette limite, héritée des écrans tous petits, est toujours en vigueur 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 raccourcir :

```
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 chaîné, 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 fonctionnel 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

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 :

```

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) :

```

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

```

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.

```
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.

## Sources

1. [https://fr.wikipedia.org/wiki/R%C3%A8gles\\_de\\_codage](https://fr.wikipedia.org/wiki/R%C3%A8gles_de_codage)
2. [https://fr.wikipedia.org/wiki/Notation\\_hongroise](https://fr.wikipedia.org/wiki/Notation_hongroise)
3. <http://pychecker.sourceforge.net/>
4. <https://www.python.org/dev/peps/pep-0020/>
5. <https://www.pylint.org/>
6. <http://sametmax.com/le-pep8-en-resume/>

## Base Types

integer, float, boolean, string, bytes

```
int 783 0 -192 0b010 0o642 0xF3
      zero binary octal hexa
float 9.23 0.0 -1.7e-6
bool True False
str "One\nTwo"
  escaped new line
  'I\'m'
  escaped '
bytes b"toto\xfe\775"
      hexadecimal octal
```

Multiline string:  
"""X\tY\tZ  
1\t2\t3"""  
escaped tab

⚡ immutables

## Container Types

- ordered sequences, fast index access, repeatable values
  - list** [1, 5, 9] ["x", 11, 8.9] ["mot"]
  - tuple** (1, 5, 9) 11, "y", 7.4 ("mot", )

Non modifiable values (immutables) ⚡ expression with only commas → tuple
- key containers, no a priori order, fast key access, each key is unique
  - dict** {"key": "value"} dict(a=3, b=4, k="v")
  - (key/value associations) {1: "one", 3: "three", 2: "two", 3.14: "pi"}
  - set** {"key1", "key2"} {1, 9, 3, 0} **set** {}
  - ⚡ keys=hashable values (base types, immutables...) **frozenset** immutable set empty

## Identifiers

for variables, functions, modules, classes... names

a...zA...Z followed by a...zA...Z\_0...9

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

Ⓢ a toto x7 y\_max BigOne  
Ⓢ 8y and for

## Variables assignment

⚡ assignment ⇔ **binding** of a name with a value

- evaluation of right side expression value
- assignment in order with left side names

```
x=1.2+8+sin(y)
a=b=c=0 assignment to same value
y, z, r=9.2, -7.6, 0 multiple assignments
a, b=b, a values swap
a, *b=seq unpacking of sequence in
*a, b=seq item and list
x+=3 increment ⇔ x=x+3
x-=2 decrement ⇔ x=x-2
x=None « undefined » constant value
del x remove name x
```

## Conversions

**type** (expression)

can specify integer number base in 2<sup>nd</sup> parameter

```
int("15") → 15
int("3f", 16) → 63
int(15.56) → 15 truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56, 1) → 15.6 rounding to 1 decimal (0 decimal → integer number)
bool(x) False for null x, empty container x, None or False x; True for other x
str(x) → "..." representation string of x for display (cf. formatting on the back)
chr(64) → '@' ord('@') → 64 code → char
repr(x) → "..." literal representation string of x
bytes([72, 9, 64]) → b'H\t@'
list("abc") → ['a', 'b', 'c']
dict([(3, "three"), (1, "one")]) → {1: 'one', 3: 'three'}
set(["one", "two"]) → {'one', 'two'}
```

separator **str** and sequence of **str** → assembled **str**

```
':'.join(['toto', '12', 'pswd']) → 'toto:12:pswd'
```

**str** splitted on whitespaces → **list** of **str**

```
"words with spaces".split() → ['words', 'with', 'spaces']
```

**str** splitted on separator **str** → **list** of **str**

```
"1,4,8,2".split(",") → ['1', '4', '8', '2']
```

sequence of one type → **list** of another type (via list comprehension)

```
[int(x) for x in ('1', '29', '-3')] → [1, 29, -3]
```

## Sequence Containers Indexing

for lists, tuples, strings, bytes...

negative index	-5	-4	-3	-2	-1
positive index	0	1	2	3	4

```
lst=[10, 20, 30, 40, 50]
```

positive slice	0	1	2	3	4	5
negative slice	-5	-4	-3	-2	-1	

**Items count**  
**len(lst) → 5**  
⚡ index from 0 (here from 0 to 4)

Individual access to **items** via **lst[index]**

```
lst[0] → 10 ⇒ first one
lst[1] → 20
lst[-1] → 50 ⇒ last one
lst[-2] → 40
```

On mutable sequences (**list**), remove with **del lst[3]** and modify with assignment **lst[4]=25**

Access to **sub-sequences** via **lst[start slice: end slice: step]**

```
lst[:-1] → [10, 20, 30, 40]
lst[1:-1] → [20, 30, 40]
lst[:2] → [10, 30, 50]
lst[1:3] → [20, 30]
lst[-3:-1] → [30, 40]
lst[3:] → [40, 50]
```

Missing slice indication → from start / up to end.

On mutable sequences (**list**), remove with **del lst[3:5]** and modify with assignment **lst[1:4]=[15, 25]**

## Boolean Logic

Comparisons : < > <= >= == != (boolean results)

**a and b** logical and both simultaneously

**a or b** logical or one or other or both

⚡ pitfall : **and** and **or** return **value** of **a** or of **b** (under shortcut evaluation).  
⇒ ensure that **a** and **b** are booleans.

**not a** logical not

**True**  
**False** } True and False constants

## Statements Blocks

```
parent statement:
├── statement block 1...
├── ...
├── parent statement:
│   ├── statement block 2...
│   ├── ...
└── next statement after block 1
```

⚡ configure editor to insert 4 spaces in place of an indentation tab.

## Modules/NAMES Imports

module **truc** ⇔ file **truc.py**

```
from monmod import nom1, nom2 as fct
  → direct access to names, renaming with as
import monmod
  → access via monmod.nom1 ...
```

⚡ modules and packages searched in python path (cf **sys.path**)

## Conditional Statement

statement block executed only if a condition is true

**if logical condition:**  
→ statements block

Can go with several **elif**, **elif...** and only one final **else**. Only the block of first true condition is executed.

```
if age <= 18:
    state = "Kid"
elif age > 65:
    state = "Retired"
else:
    state = "Active"
```

⚡ with a var **x**:  
if bool(x) == True: ⇔ if x:  
if bool(x) == False: ⇔ if not x:

## Maths

floating numbers... approximated values

Operators: + - \* / // % \*\*

Priority (...)

integer ÷ ÷ remainder

@ → matrix × python3.5+numpy

```
(1+5.3)*2 → 12.6
abs(-3.2) → 3.2
round(3.57, 1) → 3.6
pow(4, 3) → 64.0
```

⚡ usual order of operations

angles in radians

```
from math import sin, pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
sqrt(81) → 9.0
log(e**2) → 2.0
ceil(12.5) → 13
floor(12.5) → 12
```

modules **math**, **statistics**, **random**, **decimal**, **fractions**, **numpy**, etc. (cf. doc)

## Exceptions on Errors

Signaling an error:  
**raise ExcClass(...)**

Errors processing:  
**try:**  
→ normal processing block  
**except Exception as e:**  
→ error processing block

⚡ **finally** block for final processing in all cases.

```
normal
raise X()
error processing
error
raise
```



