

Bonnes pratiques de programmation en Python - INSA Lyon FIMI

v1.3 – 10/09/2026

Ce document recense les bonnes pratiques et interdits de programmation adoptés en informatique au FIMI.

Table des matières

1	Guide de style	1
1.1	Préambule	1
1.2	Indentations	2
1.3	Import	2
1.4	Noms de variables et fonctions	3
1.5	Espaces autour des opérateurs	3
1.6	Espace autour d'une virgule	4
1.7	Espace autour d'une parenthèse	4
1.8	Chaîne de caractères	4
1.9	Commentaire	4
1.10	Docstring	5
2	Organisation d'un fichier Python	5
3	Liste des interdits en Python en fonction des semestres	6
3.1	For/break	6
3.2	For/continue	6
3.3	Changement du type d'une variable	6
3.4	Déclaration de plusieurs variables sur la même ligne	7
3.5	Encadrement $x > y > z$	7
3.6	Listes de types mixtes (<i>c.-à-d.</i> contenant des objets de types différents)	7
3.7	Définition d'une fonction dans une fonction	8
3.8	Utilisation des <code>*args</code> et <code>**kwargs</code>	8
3.9	Listes en compréhension	8
3.10	Utilisation de fonctions built-in : <code>sum()</code> , <code>min()</code> , <code>max()</code> , <code>index()</code> , <code>pop()</code> , <code>insert()</code>	8
3.11	Utilisation du <code>in</code> ailleurs que dans un <code>for</code>	9
3.12	Slicing	9
3.13	<code>return</code> dans une boucle	9
3.14	Plusieurs <code>return</code> dans une fonction	9

1 Guide de style

1.1 Préambule

Qu'est-ce qu'un guide de style ? Un guide de style (ou *style guide* en anglais) est un document rassemblant des conventions de code spécifiques d'un langage de programmation.

À quoi sert un guide de style ? Un guide de style permet de produire des codes homogènes, propres et formatés qui sont ainsi facilement lisibles par tout le monde. En s'habituant à coder de cette manière, il est plus facile pour une tierce personne (ou pour vous-même en revenant sur votre code quelques jours/semaines plus tard) de comprendre très rapidement la structuration du code et donc ce qu'il fait.

Existe-t-il d'autres guides de style ? Il existe de très nombreux guides de style. Certains développeurs de langage de programmation fournissent un guide de style qui sert de base à la communauté de programmeurs. C'est le cas en Python avec le **PEP 8** (<https://www.python.org/dev/peps/pep-0008/>), et aussi en Java (<https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>).

Chaque communauté de programmeurs, entreprise ou projet peut avoir son propre guide de style.

Exemples :

- Guide de style de Google en Python : <https://google.github.io/styleguide/pyguide.html>
- Guide de style du kernel Linux en C : <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>
- Guide style de Epic Games en C++ : <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/>

Vous trouverez ci-dessous les conventions de code adoptées au FIMI. Ces règles sont en grande partie basées sur le PEP 8 défini par les développeurs de Python (<https://www.python.org/dev/peps/pep-0008/>).

1.2 Indentations

En programmation, on appelle indentation l'ajout d'espaces au début des lignes d'un fichier source afin de faire apparaître la structure logique du code.

L'indentation doit être définie par **4 espaces**.

La majorité des environnements de développement (IDE) permet de paramétrer cette propriété pour qu'en appuyant sur la touche tabulation (Tab) de votre clavier, cela ajoute automatiquement 4 espaces. Pour annuler un niveau d'indentation, il suffit généralement d'appuyer sur Shift + Tab.

1.3 Import

En Python, l'*import* permet de faire appel à du code écrit dans un autre fichier.

Les imports de modules doivent être faits :

- Toujours au début du fichier. Ne jamais faire d'imports au milieu ou en fin d'un fichier Python
- Chaque module sur sa propre ligne.
- En important le module par son nom ou alias ou en important seulement les parties ou fonctions d'intérêt.
- Évitez d'importer tout le contenu d'un module avec `*` : `from ... import *`

Exemples :

```
# Bonne pratique
import matplotlib # Chaque module sur sa ligne
import numpy
import numpy as np # import par alias
from numpy.random import randint # import d'une fonction ou sous partie
```

```
# Ne pas faire
import matplotlib, numpy
from numpy import *
```

1.4 Noms de variables et fonctions

Les **noms de variables** et les **noms de fonctions** doivent tous être écrits **en minuscules** avec des séparations entre mots par des **tirets bas** "`_`".

Exemples :

```
# Bonne pratique
ma_variable = 1
tableau = 2
def ma_fonction(x):

# Ne pas faire
maVariable = 1
Tableau = 2
def mafonction(x):
```

1.5 Espaces autour des opérateurs

Toujours mettre un espace avant et après les opérateurs d'affectation (`=`, `+=`, `-=`, `*=...`), de comparaison (`==`, `<`, `!=`, `>=`, `in`, `is not...`) et booléens (`and`, `or`, `not`).

Exemples :

```
# Bonne pratique
une_liste = [1, 2, 3]
compteur += 1
une_liste == une_autre_liste
not gagne && compteur < 10

# Ne pas faire
une_liste=[1, 2, 3]
compteur +=1
une_liste ==une_autre_liste
not gagne&& compteur<10
```

Pour les opérateurs arithmétiques, mettre des espaces avant et après l'opérateur dans les expressions simples. Pour les expressions plus complexes, mettre des espaces autour de l'opérateur de plus basse priorité.

Exemples :

```
# Bonne pratique
i = i + 1
x = x*2 - 1
calcul = x*x + y*y
c = (a+b) * (a-b)

# A ne pas faire
i=i+1
x = x * 2 - 1
calcul = x * x +y*y
c = (a + b) *(a - b)
```

1.6 Espace autour d'une virgule

Dans la majorité des cas, il faut coller la virgule au caractère précédent et mettre une espace après la virgule.

Exemples :

```
# Bonne pratique
ma_liste = [1, 2, 3]
ma_fonction(var1, var2, var3)
mon_tuple = (5, 3)

# A ne pas faire
ma_liste = [1,2 , 3]
ma_fonction(var1,var2 , var3)
mon_tuple = (5 , 3)
```

1.7 Espace autour d'une parenthèse

Ne jamais mettre d'espace juste après une parenthèse ouvrante et juste avant une parenthèse fermante.

Exemples :

```
# Bonne pratique
expression = (a+1) - b
mon_tuple = (5, 3)
ma_fonction(5, b)

# A ne pas faire
expression = ( a+1 ) - b
mon_tuple = ( 5, 3)
ma_fonction (5, b )
```

1.8 Chaîne de caractères

En Python une chaîne de caractères peut être déclarée avec des doubles ou simples guillemets. Néanmoins, nous conseillons de privilégier les doubles guillemets pour éviter les problèmes lors de l'utilisation du caractère apostrophe.

Exemples :

```
# Bonne pratique
var_1 = "Bonjour !"
var_2 = 'Salut !'

# Chaîne de caracteres incluant une apostrophe :
chaîne_1 = "j'ai"
chaîne_2 = 'j\'ai'
```

1.9 Commentaire

Deux types de commentaires peuvent être faits dans un code Python :

— Commentaires bloc

Un commentaire bloc consiste en une ou plusieurs lignes de texte placées juste au-dessus des lignes de code commentées. Chaque ligne doit commencer par un `#` et une espace et doit être indentée au même niveau que le bloc commenté.

— Commentaires ligne

Un commentaire ligne consiste en une ligne de texte placée sur la même ligne qu'une ligne de code. Le commentaire doit commencer par un `#` et une espace et doit être placé juste après le code qu'il commente.

Exemples :

```
# Bonne pratique
for i in range(5):
    # Ceci est un
    # commentaire bloc
    print("Exemple") # Ceci est un commentaire ligne

# A ne pas faire
for i in range(5):
# Ceci n'est pas un
# commentaire bloc bien formate
    print("Exemple")
```

Il est conseillé de ne pas mettre d'accent (e.g., à, é, è, ê, î, ï, ô, ù) ni dans le code ni dans les commentaires, pouvant parfois générer des erreurs d'interprétation du script Python.

1.10 Docstring

Une docstring (documentation string) est un texte permettant de décrire une fonction, une classe ou un module. Une docstring doit être entourée de trois guillemets et être placée juste avant le code. Par exemple la docstring d'une fonction doit être placée juste après la signature de la fonction. Le PEP 257 regroupe l'ensemble des bonnes pratiques pour écrire une docstring <https://www.python.org/dev/peps/pep-0257/>.

Exemples :

```
def ma_fonction():
    """Ceci est la docstring de ma_fonction.
       Décrire ce que fait ma_fonction
    """

    # Code de la fonction
```

2 Organisation d'un fichier Python

Un fichier Python doit être organisé de la manière suivante :

- Commencer par faire tous les imports de modules
- Puis écrire toutes les fonctions
- Puis écrire le script

Exemple :

```
import numpy as np # Imports

def genere_random_liste(nb): # Toutes les fonctions utilisées dans le script
    liste = []
    for i in range(nb):
        liste += [np.random.randint()]
    return liste

# Debut du script apres toutes les fonctions
nb = 10
genere_random_liste(nb)
print(liste)
```

3 Liste des interdits en Python en fonction des semestres

Plusieurs pratiques Python sont interdites au FIMI dans un but pédagogique. Le tableau ci-dessous récapitule ces pratiques et la durée de leur interdiction, et les sections suivantes les détaillent. **L'utilisation d'une pratique interdite dans une question d'IE/IEFS, même si le programme est correct, sera comptée comme fausse.**

	Semestre 1	Semestre 2	Semestre 3	Semestre 4
For/break	Interdit			
For/continue	Interdit			
Changement du type d'une variable	Interdit			
Déclaration de plusieurs variables sur la même ligne	Selon les cas			
Encadrement $x > y > z$	Interdit			
Liste de types mixtes	Interdit		Autorisé	
Définition d'une fonction dans une fonction	Interdit			
*args et **kwargs	Interdit			
Listes en compréhension	Interdit		Autorisé	
Fonctions built-in : <code>sum()</code> , <code>min()</code> , <code>max()</code> , <code>index()</code> , <code>pop()</code> , <code>insert()</code>	Interdit		Autorisé (sauf IE1)	
Utilisation du <code>in</code> ailleurs que dans un <code>for</code>	Interdit		Autorisé	
Slicing	Interdit		Autorisé	
Return dans une boucle	Interdit			
Plusieurs return dans une fonction	Interdit	Autorisé		
Tuples	Autorisé			

3.1 For/break

Il est interdit d'utiliser un `for/break` dans vos codes au FIMI. Nous avons choisi cette règle car dans l'immense majorité des cas, l'utilisation d'un `for/break` peut être remplacée par la structure de contrôle `while`. Cette règle permet de faire en sorte que la structure `while` soit maîtrisée à la sortie du FIMI.

En règle générale, l'utilisation du mot clef `break` n'est pas autorisée au FIMI car il est souvent utilisé par simplicité pour éviter de réfléchir à l'algorithmique.

3.2 For/continue

Il est interdit d'utiliser un `for/continue` dans vos codes au FIMI. Nous avons choisi cette règle car dans l'immense majorité des cas, l'utilisation d'un `for/continue` peut être remplacée par une condition bien construite dans la boucle, en inversant le test logique ou en structurant proprement le bloc d'instructions. Cette règle permet de faire en sorte que la logique des conditions et le flux d'exécution soient parfaitement maîtrisés à la sortie du FIMI.

3.3 Changement du type d'une variable

Le langage Python n'est pas un langage statiquement typé contrairement à d'autres langages comme le Java ou le C. C'est-à-dire, qu'en Python, il n'est pas nécessaire de préciser le type d'une variable lorsqu'elle est déclarée. L'une des conséquences est qu'une fois une variable déclarée, il est possible de lui affecter un nouvel objet d'un type différent de celui initialement affecté.

Exemple :

```
var = 5 # Declaration d'une variable "var" et affectation de l'objet entier de valeur 5

var = [1, 2, 3] # Affectation d'un objet de type liste à la variable "var".
```

Nous interdisons cette pratique en FIMI, car lorsque des variables changent de type au cours d'un programme, il est beaucoup plus difficile de comprendre ce que fait ce programme. Cette règle est d'autant plus vraie que le programme est long. Nous vous conseillons de continuer à utiliser cette règle même en dehors du FIMI.

3.4 Déclaration de plusieurs variables sur la même ligne

La déclaration de plusieurs variables sur une même ligne est autorisée seulement si :

- les variables sont de même type
- il est logique d'associer les variables car elles définissent des propriétés/objets/structures similaires.

Exemple :

```
# Bonne pratique
x, y = 0, -10 # Deux variables entieres definissant les coordonnees d'un point
longueur, largeur = 10, 7 # Deux variables entieres definissant la taille d'un rectangle
mini, maxi = 0, 10 # Deux variables entieres definissant le min et max d'une liste

# A ne pas faire
compteur, liste_initiale = 0, [0, 0, 0] # Deux variables de types differents
mini, compteur, maxi, nombre_tour = -5, 0, 100, 25 # Des variables de meme type, mais sans
                                                    # rapports les unes avec les autres
```

3.5 Encadrement $x > y > z$

Il est interdit de combiner des opérateurs de comparaisons sans utiliser d'opérateurs booléens (**and**, **or**, **not**) Nous interdisons cette pratique en FIMI car nous voulons nous assurer que vous sachiez exprimer, combiner et inverser des expressions booléennes en utilisant des opérateurs booléens et pas seulement sous forme mathématique.

Exemple :

```
# Bonne pratique
valeur_a_encadrer = 10
if (valeur_a_encadrer < 5) and (valeur_a_encadrer > 10):
    ...

# A ne pas faire
if (5 < valeur_a_encadrer < 10):
    ...
```

3.6 Listes de types mixtes (*c.-à-d.* contenant des objets de types différents)

Il est interdit de créer des listes contenant des objets de types différents en première année (1A). Dans la majorité des cas, si vous avez envie de créer une liste contenant des objets de types différents, c'est que vous avez mal pensé votre algorithme et qu'il faut créer plusieurs objets séparés.

Exemple :

```
# Bonne pratique
liste_ville = ["Paris", "Marseille", "Lyon"] # Remarque : Un dictionnaire (hors programme)
liste_nb_habitants = [2.1, 0.8, 0.5]        # aurait ete encore mieux adapte
```

```
# A ne pas faire
liste_ville_nb_habitants = ["Paris", 2.1, "Marseille", 0.8, "Lyon", 0.5]
```

3.7 Définition d'une fonction dans une fonction

Il est interdit de déclarer une fonction à l'intérieur d'une autre fonction. Toutes vos fonctions doivent être écrites au même niveau dans le fichier. Cette organisation permet d'identifier immédiatement l'ensemble des fonctions disponibles (voir la section Organisation d'un fichier Python).

Exemple :

```
# Bonne pratique
def pow2(x):
    return x*x
def f(x,y):
    return x + pow2(y)

# A ne pas faire
def f(x,y):
    def pow2(x):
        return x*x
    return x + pow2(y)
```

3.8 Utilisation des *args et **kwargs

Il est interdit d'utiliser les arguments `*args` et `**kwargs` en paramètre de fonctions. Chaque fonction doit avoir des paramètres clairement définis et nommés.

Nous avons choisi cette règle car l'utilisation de paramètres arbitraires masque la signature réelle de la fonction et nuit à la lisibilité du code pour les autres développeurs.

3.9 Listes en compréhension

Il est interdit d'utiliser les listes en compréhension en première année (1A) au FIMI. Nous avons choisi cette règle car elles cachent le déroulement d'une boucle, ce qui empêche de bien comprendre comment est construite une liste pas à pas.

En deuxième année (2A), nous supposons que les boucles sont acquises et cette pratique devient alors autorisée, mais non encouragée.

Exemple :

```
# Bonne pratique
liste = []
for i in range(n):
    liste.append(0)

# A ne pas faire
liste = [0 for i in range(n)]
```

3.10 Utilisation de fonctions built-in : `sum()`, `min()`, `max()`, `index()`, `pop()`, `insert()`

Il est interdit d'utiliser les fonctions `sum()`, `min()`, `max()`, `index()`, `pop()` et `insert()` dans vos codes en première année (1A) et pour l'IE1 de deuxième année (2A). Nous avons choisi cette règle car nous voulons que vous acquériez l'algorithmique qui se cache derrière ces fonctions, en écrivant vous-même les boucles et les conditions correspondantes.

En deuxième année (2A), nous supposons que les bases d'algorithmiques sont connues et vous pouvez donc utiliser ces fonctions.

3.11 Utilisation du **in** ailleurs que dans un **for**

Il est interdit d'utiliser l'opérateur **in** ailleurs que dans les instructions **for in** et **for in range** en première année (1A). Nous avons choisi cette règle car l'utilisation de **in** cache souvent une boucle **for** ou **while** que vous devez savoir écrire par vous même. En deuxième année (2a), nous supposons que cela est acquis et vous pouvez donc utiliser cet opérateur.

Exemple :

```
# A ne pas faire en 1A
liste = [1, 5, 3, 8, 10]
if 5 in liste:
    print("trouve")
```

3.12 Slicing

Il est interdit d'utiliser le Slicing en première année (1A) au FIMI. Nous avons choisi cette règle car il masque la manipulation des indices et le déroulement des boucles, ce qui empêche de bien comprendre comment parcourir ou extraire des éléments pas à pas.

En deuxième année (2A), on suppose que ces mécanismes sont acquis et cette pratique devient autorisée, car elle permet alors d'écrire un code plus concis.

```
# A ne pas faire en 1A
liste = [10, 20, 30, 40, 50]
sous_liste = liste[1:4]

# A ne pas faire en 1A
liste = [1, 2, 3, 4, 5]
inverse = liste[::-1]
```

3.13 **return** dans une boucle

Il est interdit d'utiliser un **return** à l'intérieur d'une boucle dans vos codes au FIMI. Nous avons choisi cette règle car placer un **return** au milieu d'une boucle change complètement le flot d'exécution classique et perturbe le comportement normal des boucles, qui doivent pouvoir suivre un parcours séquentiel et prévisible du début à la fin.

3.14 Plusieurs **return** dans une fonction

Il est interdit de définir une fonction avec **plusieurs return** au premier semestre de la première année (1A). Chaque fonction doit avoir un unique point de sortie. Cette contrainte est relaxée au deuxième semestre (ISN2).

Exemple :

```
# Bonne pratique
def pair1(x) :
    if x % 2 == 0 :
        res = True
    else :
        res = False
    return res

# A ne pas faire au S1
def pair2(x) :
    if x % 2 == 0 :
        return True
    else :
        return False
```
