Le style

Qu'est ce que le style ?

Style : '' *Manière particulière d'utiliser le langage pour s'exprimer ou pour écrire.* ''

En programmation, le style concerne à la fois :

- **la lisibilité**
- **la maintenabilité**

Même un code *'jetable'* doit être **lisible et correct**. Un code bien écrit est plus facile à déboguer

Le style en tant que priorité pour cette année

Pour cette année, nous mettons l'accent sur

- la correction des programme
- **le style**

L'efficacité n'est pas le point primordiale. Préoccupez vous en uniquement lorsque cela pose un problème. **Les réponses justes mais lentes, sont meilleures que les réponses rapides mais fausses.**

Pourquoi c'est important ?

Exemple de code :

```
def k(i: int) -> int:
  # rxkfghh =rxkfgh2 sauf si i=1
   rxkfghh="gh"
   if i==1:
       p=7
       a=p+2
       rxkfgh=1
       rxkfghh="temporaire"
      #print(rxkfghh, rxkfgh)
   else:
        rxkfghh2="tempoaussi"
        rxkfgh = i*k(i-1)
        #print(rxkfghh, rxkfgh)
   return rxkfgh

print(k(5))
```

Que retourne cette fonction ?

```
print(k(5))
```

```
120 # factorielle de 5 (=1*2*3*4*5)
```

Les points clé du style

Est ce que le code est lisible et maintenable ?

En vert : Ce qui est vérifié par Pylint

- De bons identifiants ?
- Une bonne 'disposition'
  - Espace, longueur des lignes <=80
- Bonne mise en place des fonctions
  - Clarté
  - Évitement des répétitions
- Est ce que la complexité a été maîtrisée ?
  - Pas plus de 4 niveau d'indentation (ex fonction, 2 boucles et un if)
  - Les fonctions ne font pas plus de 40 lignes en incluant les commentaires
- Une docstring
  - pour le programme entier
  - pour chaque fonction

Guides de style

- **Les règles vérifiées par pylint**
- **Les règles officielles de Python :**
  - Le standard PEP-8
    - un nombre important de recommandations sur la syntaxe de Python.
  - PEP 20 -- The Zen of Python
    - réflexion philosophique avec des phrases simples qui devraient guider tout programmeur.
    - accessible sous la forme d'un « œuf de Pâques » (easter egg en anglais) `import this`
- **Un guide de style du cours**
- Un cours en ligne : openclassroom

Conventions de nommage générale

Des noms qui donnent du sens

- Toutes les entités doivent recevoir des noms qui communiquent le rôle de l'entité au lecteur :
  - `cout`
  - `duree_en_seconde,`
  - `change_type_image()`
  - `...`
- **C'est la règle de nommage la plus importante.**

- Les identifiants à 1 caractère ne doivent pas être utilisés, sauf pour les situations suivantes:

  - Les variables temporaires génériques de types de base où il existe une convention bien établie, par exemple, un caractère `c`, une string `s`, et les entiers `i` et j.
  - Les variables représentant des mesures physiques communes classiquement représentées avec un seul caractère, par exemple, `x` et `y` pour les coordonnées, `t` pour le temps, v pour la vitesse.

Nom du langage python

Ne pas utiliser les noms réservés du langage tels que `list, tuple, len, dict,` etc

- Les noms intégrés ont une signification bien définie.
- En vous les attribuant
  - vous rendez confus le lecteur,
  - vous supprimez aussi la fonctionnalité de votre propre programme.

Nommage des variables et des fonctions

Les noms des variables

- **Utiliser uniquement des minuscules**
  - Séparer les mots par des underscores ('_')
  - Respecter ces règles même pour les acronymes.
  - Exemple :
    - utiliser `est_html_compatible`
    - et non `est_HTML_compatible`
- **Utiliser des noms plus longs et plus significatifs pour les variables à portée étendue,** c'est à dire ceux qui sont présents sur de nombreuses lignes de code.
- **Éviter les abréviations**
- **Jamais un seul caractères,** excepté :
  - `i, j, k,` pour des des indices
  - `c,` pour un caractère générique
  - `s,` pour une chaîne de caractère générique

Les noms des fonctions

- Comme les noms des variables
  - c'est à dire, des minuscule avec des underscores
  - et compréhensible
- Ne jamais redéfinir des fonctions déjà existantes comme par exemple :
  - `len, list …`
- Distinguer les fonctions
  - qui retournent des valeurs (fonctions réelles)
  - et les fonctions qui font certaines choses (procédures)
- Pour les fonction (réelles), donner un nom de fonction
  - pour ce qu'elle renvoie
  - et non pour ce qu'elle fait :
    - `moyenne()`
    - et non `calcule_moyenne()`
- Pour les procédure, donner le nom de fonction pour ce que la fonction fait de manière explicite:
  - `affiche_moyenne()`

Conventions de nommage spécifiques

Pluriels

La forme plurielle doit être utilisée pour les noms représentant une collection d'objets.

```
points, valeurs
```

➢ Cela améliore la lisibilité car le nom donne à l'utilisateur un indice immédiat du type de la variable.

Éléments d'une collection

la forme singulière du nom de la collection doit être utilisée pour ses éléments.

```
for point in points:
    ...
for valeur in valeurs:
    ...
```

➢ Cela améliore la lisibilité en établissant de manière explicite la relation entre la collection et ses éléments.

Index des itérateurs

Les indices génériques devraient être nommés `i, j, k` etc.

```
for i in range(len(nums)):
  ...
```

➢ La notation provient d'une convention en mathématiques.

➢ Les variables `j` et `k` devraient être utilisées uniquement pour les boucles imbriquées.

Nom des booléens

Le préfixe `est` devrait être utilisé pour les variables et les méthodes booléennes.

```
est_visible, est_fini, est_trouve, est_ouvert...
```

➢ C'est une pratique commune dans la communauté

➢ Utiliser le préfixe `est` permet de résoudre le problème du choix des mauvais noms de variable booléennes tels que `statut` ou `drapeau`.

Mise en page et commentaires

Quantité d'indentation

L'indentation de base doit être de 4 espaces

Les lignes

- Séparer les fonctions par 2 ou 3 lignes vides
- Au plus 80 caractères par ligne
  - Occasionnellement, vous pouvez en avoir 100
- Ne pas écrire plusieurs instruction sur une ligne

Les commentaires

- Chaque programme doit avoir une docstring au début
  - indiquant son rôle, l'auteur et la date
- Chaque fonction doit avoir une docstring au début qui indique ce qu'elle fait
- Ailleurs utiliser les commentaires avec parcimonie
  - Un bon code doit être lisible sans commentaire
  - Ne commentez pas un mauvais code pour l'expliquer, **changez le !**

Espace blanc

Ne lésinez pas sur les espaces blancs.

- Les opérateurs conventionnels doivent être entourés d'un caractère d'espace.
- Les virgules doivent être suivies d'un espace blanc.

```
a = (b + c) * d
```

Plutôt que

```
a=(b+c)*d
```

```
fait_des_choses(a, b, c, d)
```

Plutôt que

```
fait_des_choses(a,b,c,d)
```

```
n = 25 * (nb_chars  -  1)
```

Plutôt que

```
n=25*(nb_chars - 1)
```

- Fait ressortir les composants individuels des déclarations.
- Améliore la lisibilité.
- Voir la PEP-8 pour plus de détails sur les espaces blancs.

Exemples : Erreur de style

Exemple 1 : Calcul de périmètre

Ne pas faire cela

```
import math

def calcule_perimetre(r) :
    distance=2*r
    perim=math.pi*distance
    return round(perim, 5)
```

Mais plutôt faire

```
import math

NBCHIFFRES = 5

def perimetre_cercle(un_rayon):
    """
    Calcule le périmètre d'un
    cercle à partir de son rayon.
    """
    diametre = 2 * un_rayon
    perimetre = math.pi * diametre
    return round(perimetre, NBCHIFFRES)
```

Exemple 2 : Nombre de 'e'

Ne pas faire cela

```
def compte_nombre_e(list):
    "affiche le nombre de e
      dans une chaîne "
    nb=0
    for carac in list:
        if carac=='e':
            nb +=1
    print(nb)
```

Mais plutôt faire

```
def affiche_nombre_e(chaine):
    """affiche le nombre de e
        dans une chaîne """
    compteur = 0
    for c in chaine:
        if c == 'e':
            compteur += 1
    print(compteur)
```

Exemple 3 : Code obscur

"Écrire clairement, ne pas être trop génial"

Que fait la fonction suivante ?

```
def intelligent(n):
    for i in range (1, n*(n+1) + 1 ):
        print(i %(n+1)and "*" or '\n',end = "")
```

- C'est incompréhensible!
- Visez la clarté, pas la taille minimale

"Cette version est bien plus lisible/maintenable"

```
def affiche_carre(n):
    """affiche un carré de taille n de "*" """
    for ligne in range (n):
        print(n * "*")
```

```
intelligent(3)
```

→

```
***
***
***
```

Exemple 4 : Évitez les répétitions

La mauvaise façon

```
def affiche_monnaie(n_centimes):
    """affiche les pièces correspondants  à un nombre de
centimes.
    précondition :  le nombre n_centimes est divisible par 10"""

    n_2euros = n_centimes // 200
    if n_2euros > 0:
        print('{} pièce(s) de 2 euro'.format(n_2euros))
        n_centimes = n_centimes - n_2euros *200

    n_euros = n_centimes // 100
    if n_euros > 0:
        print('{} pièce(s) de 1 euro'.format(n_euros))

    if n_centimes >= 50:
        print('1 pièce de 50 centimes')
        n_centimes = n_centimes - 50

    if n_centimes >= 20:
        print('1 pièce de 20 centimes')
        n_centimes = n_centimes - 20

    if n_centimes >= 10:
        print('1 pièce de 10 centimes')
```

```
affiche_monnaie(280)
```



```
1 pièce(s) de 2 euro
1 pièce de 50 centimes
1 pièce de 20 centimes
1 pièce de 00 centimes
```

Exemple 4 : Évitez les répétitions

Utiliser des listes de données pour éviter les répétitions

```
def affiche_monnaie2(n_centimes):
 """affiche les pièces correspondants  à un nombre de centimes.
 précondition :  le nombre n_centimes est divisible par 10"""

 pieces = [(200, '2-euros'),(100, '1-euros'),
           (50, '50 centimes'), (20, '20 centimes'),
           (10, '10 centimes') ]

 for piece_valeur, piece_nom in pieces: # boucle les pièces
   nombre_piece = n_centimes // piece_valeur
   if nombre_piece > 0:
     print('{} piece(s) de {}'.format(nombre_piece, piece_nom))
     n_centimes = n_centimes % piece_valeur
```

Une bien meilleure façon

```
affiche_monnaie2(280)
```

→

```
1 pièce(s) de 2 euro
1 pièce(s) de 50 centimes
1 pièce(s) de 20 centimes
1 pièce(s) de 00 centimes
```

Exemple 5 : Gestion des cas *'spéciaux'*

Assurez-vous que les cas spéciaux sont vraiment spéciaux

Pas comme cela

```
def affiche_tout(liste_elements):
 """Affiche tous les elements d'une liste
   (qui peut être vide)"""

 if len(liste_elements) > 0:
     for element in liste_elements:
           print(element)
```

```
affiche_tout([3,2,1]
```



```
3
2
1
```

Mais comme ceci

```
def affiche_tout(liste_elements):
 """Affiche tous les elements d'une liste
   (qui peut être vide)"""

 for element in liste_elements:
     print(element)
```