

Plan

I.	Introduction	1
II.	Idées générales	1
III.	Idées en pratique sur Python	1
IV.	Commentaires.....	2
V.	Dans les fonctions	2
VI.	Exercices.....	3
VII.	Écrire les tests soi même	5
VIII.	Assert	5
IX.	Doctest	7

I - Mise au point de programme : spécification

I. Introduction

En tant que codeur débutant, il est indispensable de prendre les bonnes habitudes dès le départ. De nombreuses conventions se sont imposées au fil du temps devant la complexité, la taille croissante du code mais aussi parce que le partage du travail devient de plus en plus indispensable. L'accroissement de la mémoire a aussi permis d'améliorer la lisibilité du code. Et oui, il fut un temps où chaque octet était important et une variable à une lettre moins couteuse qu'une de 20, où un commentaire prenait trop de place dans le fichier...Aujourd'hui, la longueur du nom d'une variable ou d'un commentaire ne doit plus vous effrayer.

Votre code doit être facile à lire par une autre personne. Et cette autre personne peut très bien être-vous 2 ans plus tard....on est parfois surpris de ne pas comprendre ce que l'on a bien pu faire!

II. Idées générales

Voici quelques conventions propres à Python mais qui fonctionnent pour tous les langages. Elles sont disponibles sur [le site de python](#). Allez voir !

Je les résume en français : **beau est mieux que laid**, **explicite c'est mieux qu'implicite** (oui, le truc qu'on ne sait plus pourquoi...), **simple mieux que complexe**, **complexe mieux que compliqué**, éviter si possible trop d'imbrications, aérer, etc....Je vais isoler la dernière idée : maintenant c'est mieux que jamais mais jamais e:t parfois préférable à immédiatement.

→ Oui, bien souvent, on veut tellement finir et tester notre bébé que l'on en oublie d'être propre et judicieux, mieux vaut faire une pause.

III. Idées en pratique sur Python

Elles sont disponibles sur [le site de python](#). Allez voir !

Je les résume au maximum. Pour bien comprendre l'intérêt de tout cela, gardez à l'esprit que vous passerez plus de temps à relire votre code qu'à l'écrire!

- ✓ En python ne mélangez jamais tabulations et espaces
- ✓ Une ligne de code ne devrait pas dépasser 79 caractères.
- ✓ Encoder en Utf-8 (voir chapitre Représentation d'un texte...lorsqu'il sera disponible)
- ✓ Importer les bibliothèques en début de programme sur des lignes séparées. ([Pour en savoir plus sur les bibliothèques.](#))

Il y a de nombreuses conventions sur les espaces. Globalement on met un (et un seul) espace avant et après les affectation, comparaisons, booléens et opérations et pas d'espace pour le reste (, {, [...

IV. Commentaires

Les commentaires doivent être des phrases complètes de préférence en anglais. Je ne vais pas vous obliger à suivre cette règle mais soyez cohérents. Je vois très régulièrement des codes en Franglais. Des variables en français, d'autre en anglais et de même pour les noms de fonctions. Faites un choix une fois pour toute (sauf pour les commentaires pour l'instant).

En Python les commentaires commencent par un #. Nous allons donc adopter pour cette année: commentaires en français et au choix noms de fonctions ET variables dans une même langue (français ou anglais).

➤ Pensez à mettre à jour vos commentaires si vous modifiez le code !

Commenter ce n'est pas commenter chaque ligne mais plutôt indiquer **les grandes étapes** lorsque le code s'allonge et **expliquer certaines lignes** qui vous paraissent techniques.

V. Dans les fonctions

Nous verrons en terminale la programmation orienté objet. Pour différencier les objets des fonctions, variables et méthodes (des objets) on utilise deux conventions de nommage différentes

nom_de_ma_fonction : pour les fonctions, variables et méthodes tout en minuscule avec _ pour séparer les mots.

MaClasse : pour le nom des classes (en terminale). Majuscule pour chaque nouveau mot et mots collés. Norme appelée CamelCase, utilisé dans bien des langages.

Les constantes sont entièrement en majuscule : NOM_DE_MA_CONSTANTE. Une constante est une variable à laquelle on donne une valeur qui ne changera pas dans tout le programme. Par exemple, on code un jeu et au début, on préfère s'en tenir à deux joueurs. On peut créer NB_MAX_JOUEURS=2 et on utilisera cette variable dans tout le programme sans jamais la changer. Imaginons que le code permette de jouer à trois joueurs. Il suffit de changer la valeur de la constante NB_MAX_JOUEURS=3.

Intéressons-nous maintenant à la documentation des fonctions. Lorsque vous créez une fonction, vous devez la **documenter**. C'est le rôle du **docstring**. Le docstring se place juste après la création de la fonction par def. Il commence et termine par trois guillemets ".

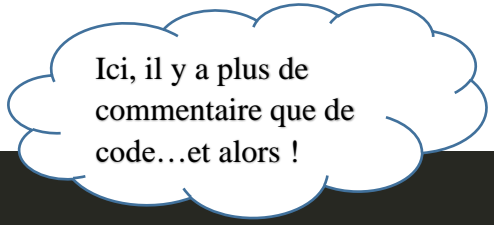
Votre docstring doit décrire le rôle de la fonction, puis les paramètres passés en arguments (type et rôle), ainsi que le type de ce qui est retourné

Ce docstring, peut être lu en tapant :

nom_de_ma_fonction.__doc__ (deux underscores de chaque côté).

Exemple :

```
def mettre_au_carre(x):  
    """renvoie le carré de x  
  
    Paramètres d'entrée : x -> (int, float ou decimal) : un nombre quelconque  
    Paramètre de sortie : Un nombre -> (int, float ou decimal)  
    """  
    return(x*x)
```



Ici, il y a plus de
commentaire que de
code...et alors !

A FAIRE :

Testez la docstring sous spyder. Si on tape `print(mettre_au_carre.__doc__)` ou `help(nom_de_la_fonction)` on lira 'renvoie lequelconque'

VI. Exercices

1. La mesure d'un angle peut être donnée en degrés ou en radian. L'unité de calcul naturelle pour les angles est le radian. La bibliothèque math ne sait donc calculer le cosinus que d'un angle donné en radian. `ma_fonction1`, convertit des degrés en radians car π radians correspondent à 180° . Modifier les noms des fonctions, variables et créez le docstring selon les normes énoncées plus haut.

```
import math
def ma_fonction1(x):
    return(x*math.pi/180)

def ma_fonction2(x):
    return(math.cos(ma_fonction1(x)))
```

Petite remarque, on a importé le module math de la bibliothèque standard (la bibliothèque de Python). Ci-dessus, on l'a juste préparé et on appelle les fonctions qu'il contient au fur et à mesure des besoins (exemple `math.cos`). Ci-dessous, j'importe directement ce que je vais utiliser. Les fonctions sont chargées en mémoire et donc directement connues dans mon programme (je tape `cos` directement, plus `math.cos`). A noter si je remplace par `from math import *`, alors je charge dans la mémoire de mon programme toutes les fonctions qui existent dans math...donc je surcharge inutilement.

```
from math import pi,cos
#from math import *

def ma_fonction1(x):
    return(x*pi/180)

def ma_fonction2(x):
    return(cos(ma_fonction1(x)))
```

2. Comprendre ce que fait le code suivant et corriger tous les problèmes de spécification.

```
def fonction(a,b):
    return(a*100/b)
```

3. Comprendre ce que fait le code suivant et corriger tous les problèmes de spécification

```
import math

def fonction1(a,b,c,d):
    return(a+b+c+d)

def fonction2(a,b,c,d):
    return(fonction1(a,b,c,d)/4)
```

4. Comprendre ce que fait le code suivant et corriger tous les problèmes de spécification

Que dois-je taper pour avoir l'aire d'un rectangle de côté 3 et 4, agrandi 10 fois?

Que dois-je taper pour que le programme affiche le volume d'un pavé droit de mesure 5 ; 7 ; 6 cm, agrandi 3 fois?

```
import math

def fonction1(a,b):
    coefficient=10
    c=coefficient*coefficient*a*b
    print("L'aire d'un rectangle de mesure",a,"et",b,"qui subit un agrandissement de coefficient",coefficient,"est",c)

#volume d'un parallélépipède de côté a,b,c qui subit un agrandissement de coefficient d
def fonction2(a,b,c,d):
    return(a*b*c*d*d*d)
```

5. Comprendre ce que fait le code suivant et corriger tous les problèmes de spécification

```
def fonction1(a):
    print("1 - jouer à la bataille navale")
    print("2 - jouer au puissance 4")
    print("3 - quitter")
    a=input("Taper votre choix 1 ou 2 ou 3:")
    while a!="1" or a!="2" or a!="3":
        a=input("Taper votre choix :")
    return(a)
```

A FAIRE : Vous devez être capable **de programmer** une fonction à partir de sa spécification.

```
def encoder(chaine):
    '''
    Encode une chaîne de caractère en son équivalent ASCII

    Paramètres d'entrée : chaine de caractère -> char
    Paramètre de sortie : encodage ASCII -> int
    '''
    # votre code ici
```

II - Mise au point de programme : Les tests

De toute évidence, le code qu'on écrit n'a aucune assurance de fonctionner si on ne le teste pas...

Ce qui doit être Tester :

- Cas correct
- Cas faux
- Cas limite
- Cas hors limite.

Exemple : avec la fonction qui teste si des nombres sont en ordres croissants.

```
def croissant(a,b,b):
    """Fonction de test de la croissance d'une suite de nombre
    Paramètres d'entrée : a,b,c -> int ou float
    Paramètres de sortie : retourne -> valeur booléenne
```

```
"""
    (True si a,b,c sont tels que a<b<c ; False sinon)
"""
Return a<b<c
# Jeu d'essai
assert(croissant(1,5,9)==True)
assert(croissant(1,5,2)==False)
assert(croissant(7,3,4)==False)
assert(croissant(9,7,5)==False)
assert(croissant(5,5,5)==False)
```

Il existe plusieurs approches en Python pour s'assurer qu'un code fonctionne.

VII. Écrire les tests soi même

Le moyen le plus simple consiste à écrire un jeu de test après `if __name__=="__main__":`

```
def ma_fonction(n):
    ...

if __name__ == '__main__':
    print(ma_fonction(5))
```

C'est généralement ce qu'on fait quand on développe. Ces tests doivent couvrir tous les cas possibles et être compréhensibles.

Selon les contextes (devoir, projet, développement en cours...) on peut les laisser ou les effacer.

Il est préférable de les remplacer par de vrais tests... → A EVITER

VIII. Assert

Python intègre un mot clef `assert` qui va lever une exception `AssertionError` si la condition qui suit est fausse:

```
>>> assert 1 == 1 # ne fait rien

>>> assert 1 == 2 # plante le programme
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

C'est le moyen le plus efficace et rapide de tester un programme ou une fonction.

Il ne faut pas intégrer les assertions à la fonction elle-même. Il est préférable de les intégrer à des fonctions de tests indépendantes du programme.

Un exemple

Prenons la fonction qui affiche la suite de Fibonacci. Elle utilise une structure que vous ne connaissez pas encore la list, mais ce n'est pas le sujet ici. On ne va faire que l'utiliser.

```
def fibonacci(n):
    """
    Liste des termes de la suite de Fibonacci de l'indice 0 à l'indice n inclus

    Paramètres d'entrée : n->int : l'indice maximal voulu
    Paramètres de sortie : suite_fibonacci -> list : la liste des termes
    """
```

```
'''
if type(n) != int or n < 0:
    return None
x = 1
y = 1
suite_fibonacci = [x]
indice = 0
while indice < n:
    x, y = y, x + y
    suite_fibonacci.append(x)
    indice += 1
return suite_fibonacci
```

la **suite de Fibonacci** est une [suite d'entiers](#) dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence par les termes 0 et 1

On peut tester plusieurs choses :

- La taille de la liste : $n + 1$
- Différents résultats : 0, 1, 5 etc.
- Les éléments de la liste sont des entiers
- La propriété de Fibonacci : $un + un + 1 = un + 2$
- La sortie dans les cas impossibles : paramètre négatif, paramètre non entier

```
def tester_fibonacci():
    '''
    Teste certaines propriétés de la fonction Fibonacci

    Paramètres de sortie : None
    Conditions d'utilisation : lève une exception AssertionError
                             si la fonction est mal programmée
    '''
    fib_10 = fibonacci(10)

    # longueur de la liste
    assert len(fib_10) == 11

    # différents résultats
    assert fibonacci(0) == [1]
    assert fibonacci(1) == [1, 1]
    assert fibonacci(5) == [1, 1, 2, 3, 5, 8]

    # ses éléments sont entiers
    for terme in fib_10:
        assert type(terme) == int

    # La propriété de récurrence
    assert fib_10[-3] + fib_10[-2] == fib_10[-1]

    # Valeur de retour dans les cas impossibles
    assert fibonacci(-1) == None
    assert fibonacci('a') == None
    assert fibonacci(3.14) == None
```

IX. Doctest

Notre : La rédaction de la Docstring est libre. Les deux notations suivantes sont correctes. Prenez celle qui vous paraît naturelle.

```
'''
    Teste certaines propriétés de la fonction Fibonacci

    Paramètres de sortie : None
    Conditions d'utilisation : lève une exception AssertionError
                            si la fonction est mal programmée
'''
"""
Calcule produit de a et b
@param a: (number, str, list) premier facteur
@param b: (number, str, list) second facteur
@return: (number, str, list) le produit
"""
```

Mais revenons à l'utilisation de Doctest, qui s'intègre naturellement à la docstring. Python permet grâce au module `doctest` d'intégrer les tests à la documentation. Il est parfois délicat de tester certaines fonctions, en particulier les affichages. Pour les fonctions qui réalisent des calculs cela reste pratique.

Un exemple :

```
def multiple (a, b):
    """
    Calcule produit de a et b
    @param a: (number, str, list) premier facteur
    @param b: (number, str, list) second facteur
    @return: (number, str, list) le produit

    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
    """
    return a * b

if __name__ == '__main__':
    import doctest
    doctest.testmod() # s'il ne se passe rien, les test sont justes
```

Quand on exécute le programme, il ne se passe rien.

Un exemple qui échoue :

```
def Fonction_mal_testee():
    """
    Simple fonction qui echoue
    >>> Fonction_mal_testee()
    3
    """
    return 2

if __name__ == "__main__":
    import doctest
    doctest.testmod() # s'il ne se passe rien, les tests sont justes.
```

Voici la sortie d'un exemple qui échoue

```
>>> python3 2_tester_doctest.py
*****
File "/home/quentin/realiser_des_tests/2_tester_doctest.py", line 5,
in __main__.Fonction_mal_testee
Failed example:
    Fonction_mal_testee()
Expected:
    3
Got:
    2
***
*****
1 items had failures:
  1 of 1 in __main__.Fonction_mal_testee
***Test Failed*** 1 failures.
```

Pour aller plus loin :

<http://www.test-recette.fr/recette/aspects-organisationnels.html>

<http://sametmax.com/les-docstrings/>

A FAIRE : Reprenez les 5 exercices de la partie I et ajoutez alternativement des tests avec la méthode Assert et la méthode doctest.