Quelques bonnes pratiques de programmation

1. Introduction

Afin d'uniformiser l'écriture de code en Python, la communauté des développeurs Python recommande un certain nombre de règles afin qu'un code soit lisible. Lisible par quelqu'un d'autre, mais également, et surtout, par soi-même. Lorsque l'on essaie de relire un code écrit « rapidement » il y a un 1 mois, 6 mois ou un an, si le code ne fait que quelques lignes, il se peut que l'on s'y retrouve, mais s'il fait plusieurs dizaines voire centaines de lignes...

Dans ce contexte, le créateur de Python, Guido van Rossum, part d'un constat simple : « code is read much more often than it is written » (« le code est plus souvent lu qu'écrit »).

Afin d'améliorer le langage Python, la communauté qui développe Python publie régulièrement des "Python Enhancement Proposal" (PEP), suivi d'un numéro. Il s'agit de propositions concrètes pour améliorer le code, ajouter de nouvelles fonctionnalités, mais aussi des recommandations sur la manière d'utiliser Python, bien écrire du code, etc...La plus connue de ces publication est la PEP 8 qui consiste en un nombre important de recommandations sur la syntaxe de Python (indentations, import de modules, noms de variables, gestion des espaces, commentaires, etc...

Pour en savoir savoir plus :

- https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/) (documentation originale en anglais)
- https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/ (https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/) (un résumé néanmoins très complet des bonnes pratiques d'utilisation de python).

Dans cette feuille il sera vu comment :

- Spécifier une fonction.
- Utiliser des jeux de tests.

2. Spécifier une fonction

Jusqu'à présent, la description et le comportement attendu des fonctions étaient précisés dans la consigne des exercices. Bien que non obligatoire, une bonne pratique est d'inscrire ces éléments dans le corps de la fonction, sous forme de chaîne de caractère un peu spéciale("docstring" en anglais). C'est ce que l'on appelle, le prototype.

Exercice 1 :

1. Exécuter le code ci-dessous. Que contient l'affichage ?

In [12]:

```
help(bin)
```

```
Help on built-in function bin in module builtins:

bin(number, /)
    Return the binary representation of an integer.

    >>> bin(2796202)
    '0b1010101010101010101010'

Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Réponse :

-

1. Exécuter le code ci-dessous. Que contient l'affichage ?

In [15]:

```
from random import *
help(randint)
```

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Réponse :

-

A retenir :

Spécifier une fonction, c'est préciser :

- Ce que fait la fonction(prototype).
- Ce qu'elle prend en paramètres(les préconditions).
- Ce qu'elle renvoie(les postconditions).

Inscrire ces éléments dans la fonction permet :

- De documenter précisément la fonction.
- De permettre aux autres utilisateurs d'accéder à la documentation de cette fonction à l'aide de la commande `help` .

Un exemple de docstring

In [38]:

```
def distance(a,b):
    """Calcule la distance de deux nombres.
    paramètres :
    ------------
    a et b de type entiers ou float

    return :
    ------------
    dist de type entier ou float

    """

    distance=max(a-b,b-a)
    return dist


help(distance)
```

```
Help on function distance in module __main__:

distance(a, b)
    Calcule la distance de deux nombres.
    paramètres :
    ------------
    a et b de type entiers ou float

    return :
    ------------
    dist de type entier ou float
```

A retenir :

- On écrit le prototype de la fonction sous la ligne qui la définit.
- Ce n'est pas obligatoire mais la bonne pratique est d'utiliser des triples guillemets pour encadrer le ptototype (""").
- D'abord la description, puis les paramètres et leur type(pré-conditions), puis la valeur de retour et son type(post-conditions).
- On utilise des retours à la ligne pour améliorer la lisibilité.

Exercice 2 :

Ecrire le prototype de la fonction ci-dessous.

In [2]:

```
def moyenne(L):
    S=0
    for e in L:
        S=S+e
    return S/len(L)

help(moyenne)
```

Exercice 3 :

Ecrire le prototype de la fonction ci-dessous.

In [33]:

```
def insere(n,L):
    resultat=[]
    i=0
    while L[i]<n:
        resultat.append(L[i])
        i=i+1

    resultat.append(n)

    while i < len(L):
        resultat.append(L[i])
        i=i+1

    return resultat


help(insere)
```

```
Help on function insere in module __main__:

insere(n, L)
```

Exercice 4 :

Compléter la fonction spécifiée ci-dessous

In [42]:

```
def produit_pair(a,b,c):
    """Determine si le produit de a, b et c est pair
    Parametres :
    ------------
    a, b, c de type entier

    Return :
    --------
    1 si le produit est pair.
    0 sinon.

    """
```

3. Le développement par les tests.

Écrire un programme, c'est bien. Écrire un programme juste, c'est mieux. L'idéal serait de pouvoir prouver tous les programmes que l'on écrit, mais outre que ce n'est pas forcément évident, on n'est jamais à l'abri d'une erreur de frappe qui nous fait lire ce que l'on voudrait lire et non ce qui est réellement écrit dans le code. C'est pourquoi il est intéressant (bien que non suffisant en général) de tester les bouts de code que l'on écrit sur des cas particuliers pour lesquels on connait bien le résultat attendu.

Le développement piloté par les tests ou TDD (pour Test Driven Development) est une méthode d'écriture de programme qui met en avant le fait d'écrire d'abord un test pour chaque spécification du programme puis écrire le code qui permettra au programme de passer ce test avec succès.

Plus particulièrement, le TDD présente trois principes:

- Ecrire les tests avant d'écrire le programme.
- Il ne faut tester qu'un point précis du programme.
- Il ne faut écrire que le minimum de code permettant de réussir le test.

Il est bien évidemment impossible d'effectuer une infinité de tests (ce qu'il faudrait faire idéalement pour que le programme fonctionne dans tous les cas).Cela oblige à envisager tous les cas particuliers et les cas "limites" que le programme doit traiter.

Une nouvelle instruction va permettre de réaliser ces tests : `assert`

Assert

Python intègre un mot clef `assert` qui va lever une exception `AssertionError` (il va donc afficher message d'erreur si la condition qui suit est fausse)

In [60]:

```
#cette instruction ne renvoie rien
assert (1 > 0)
```

In [61]:

```
#cette instruction stoppe le programme et lève une erreur
assert( 1 < 0) , "1 est strictement positif voyons..."
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
<ipython-input-61-4c32dceb9a79> in <module>
      1 #cette instruction stoppe le programme et lève une erreur
----> 2 assert( 1 < 0) , "1 est strictement positif voyons..."

AssertionError: 1 est strictement positif voyons...
```

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

Exemple1 :

Dans la suite de Fibonacci, chaque terme est obtenu en additionnant les deux précédents. Les deux premiers termes de cette suite sont 1 et 1. Cette suite commence donc ainsi : $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

On veut écrire la fonction `fibonacci(n)` dont voici les spécifications :

In [62]:

```
def fibonacci(n):
    """
    Liste des n premiers termes de la suite de Fibonacci

    Paramètre :
    -----------
    n entier supérieur à 1, le nombre de termes voulus

    Return :
    --------
    La liste des n termes
    """
```

1. On choisit les tests que doit passer cette fonction :
   - a) Il y a bien n termes dans la liste renvoyée (on doit tester pour quelques valeurs de n)
   - b) `fibonacci(1)` doit renvoyer `[1]`
   - c) `fibonacci(5)` doit renvoyer `[1,1,2,3,5]`
   - d) La relation de récurrence est vérifiée pour tous les termes à partir du troisième.

En éxécutant le code ci-desous, on vérifie bien sûr que tous les tests échouent.

In [70]:

```
# a)
assert(len(fibonacci(10))==10)
assert(len(fibonacci(5))==5)

# b)
assert(fibonacci(1)==[1])

# c)
assert(fibonacci(5)==[1,1,2,3,5])

# d)
L=fibonacci(10)
for i in range (2,len(L)):
    assert(L[i]==L[i-1]+L[i-2])
```

1. On écrit le code de la fonction.

In [73]:

```
def fibonacci(n):
    """
    Liste des n premiers termes de la suite de Fibonacci

    Paramètre :
    -----------
    n entier supérieur à 1, l'indice maximal voulu

    Return :
    --------
    La liste des n termes
    """
    x=1
    y=1
    f=[x]
    i=1
    while i < n:
        x,y=y,x+y
        f.append(x)
        i=i+1

    return f
```

1. On éxécute les tests. S'ils réussissent, rien ne se passe, sinon une erreur est renvoyée, on doit modifier le code de la fonction.

In [74]:

```
# a)
assert(len(fibonacci(10))==10)
assert(len(fibonacci(5))==5)

# b)
assert(fibonacci(1)==[1])

# c)
assert(fibonacci(5)==[1,1,2,3,5])

# d)
L=fibonacci(10)
for i in range (2,len(L)):
    assert(L[i]==L[i-1]+L[i-2])
```

Exercice 4 :

Rappel : Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400.

1. Spécifier la fonction `bissextile(A)` ci-dessous.
2. Ecrire quelques tests pour cette fonction.

In [75]:

```
#1.
def bissextile(A):
    """A compléter"""
    if A % 4 == 0:
        if A % 100 == 0 :
            if A % 400 == 0:
                return True
            else :
                return False
        else :
            return True
    else:
        return False
```

In [ ]:

```
#2. tests
```

Exercice 5 :

Voici une liste de quelque tests. Ecrire une fonction qui passe ces tests(ne pas oublier de la spécifier).

Aide : on pourra utiliser la fonction `ord()` .

In [78]:

```
assert(score('a')==1)
assert(score('z')==26)
assert(score('azerty')==95)
assert(score('python')==98)
```

In [82]:

```
def score(mot):
    """Détermine le score d'un mot
    à l'aide de la valeur de chaque lettre:
    a=1, b=2,...,z=26

    Paramètre :
    -----------
    mot une chaîne de caractères ascii minuscule

    Return :
    --------
    Un entier
    """
```

A retenir :

- Le succès des tests ne garantit pas que la fonction est correcte.
- Il est néanmoins important de bien choisir ces tests:
  - Cas particuliers
  - Exemples courants
  - Valeurs limites
  - etc...

Exemple 2 :

Les fonctions précédentes ont été spécifiées et des tests ont été écrits. Néanmoins, rien ne garantit que les arguments passés dans une fonction au moment de son appel soient corrects (un entier à la place d'un caractère, ou une valeur hors du domaine de validité,...)

Pour remédier à cela, l'instruction `assert` inscrite dans le corps de la fonction permet de garantir les pré-conditions et les posts-conditions.

Reprenons la fonction `fibonacci(n)` . L'entier n doit être supérieur ou égal à 1. Que se passe-t-il si l'on saisit 0 ou une valeur négative, ou encore une chaîne de caractère ?

In [83]:

```
print(fibonacci(0))
print(fibonacci(-1))
print(fibonacci('a'))
```

```
[1]
[1]

---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-83-958e3343714a> in <module>
      1 print(fibonacci(0))
      2 print(fibonacci(-1))
----> 3 print(fibonacci('a'))

<ipython-input-73-2bc8fe2c5e7a> in fibonacci(n)
     15     f=[x]
     16     i=1
---> 17     while i < n:
     18         x,y=y,x+y
     19         f.append(x)

TypeError: '<' not supported between instances of 'int' and 'str'
```

Dans les deux premiers cas, aucune erreur n'est renvoyée mais le résultat de la fonction n'est pas correct. Dans le troisième cas, une erreur est signalée mais il n'est pas toujours évident de comprendre le message.

Pour être plus explicite, on peut utiliser l'instruction `assert` comme suit :

In [91]:

```
def fibonacci(n):
    """
    Liste des n premiers termes de la suite de Fibonacci

    Paramètre :
    -----------
    n entier supérieur à 1, l'indice maximal voulu

    Return :
    --------
    La liste des n termes
    """
    assert(type(n)==type(0)), 'n doit être un entier'
    assert(n>0), 'n doit être strictement positif'

    x=1
    y=1
    f=[x]
    i=1
    while i < n:
        x,y=y,x+y
        f.append(x)
        i=i+1

    return f
```

Voici ce que renvoient désormais les instructions précédentes :

In [93]:

```
print(fibonacci(0))
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
<ipython-input-93-ed8977d052a5> in <module>
----> 1 print(fibonacci(0))

<ipython-input-91-1818b5434d5c> in fibonacci(n)
     12     """
     13     assert(type(n)==type(0)), 'n doit être un entier'
---> 14     assert(n>0), 'n doit être strictement positif'
     15
     16     x=1

AssertionError: n doit être strictement positif
```

In [94]:

```
print(fibonacci(-1))
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
<ipython-input-94-99885d541aa9> in <module>
----> 1 print(fibonacci(-1))

<ipython-input-91-1818b5434d5c> in fibonacci(n)
     12     """
     13     assert(type(n)==type(0)), 'n doit être un entier'
---> 14     assert(n>0), 'n doit être strictement positif'
     15
     16     x=1

AssertionError: n doit être strictement positif
```

In [95]:

```
print(fibonacci('a'))
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
<ipython-input-95-615ed0b15889> in <module>
----> 1 print(fibonacci('a'))

<ipython-input-91-1818b5434d5c> in fibonacci(n)
     11     La liste des n termes
     12     """
---> 13     assert(type(n)==type(0)), 'n doit être un entier'
     14     assert(n>0), 'n doit être strictement positif'
     15

AssertionError: n doit être un entier
```

En cas de valeur incorrecte, un message d'erreur explicite est affiché.

Exercice 6 :

En utilisant l'instruction assert, garantir les préconditions de la fonction ci-dessous et vérifier que des messages d'erreurs explicites s'affichent avec les instructions ci-après.

In [5]:

```
def score(mot):
    """Détermine le score d'un mot
    à l'aide de la valeur de chaque lettre:
    a=1, b=2,...,z=26

    Paramètre :
    -----------
    mot une chaîne de caractères ascii minuscule

    Return :
    --------
    Un entier
    """




    s=0
    for lettre in mot:
        s=s+ord(lettre)-96
    return s
```

In [6]:

```
print(score('Azerty'))
```

```
63
```

In [7]:

```
print(score(95))
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-7-1a14956c143c> in <module>
----> 1 print(score(95))

<ipython-input-5-fa50494187da> in score(mot)
     18
     19     s=0
---> 20     for lettre in mot:
     21         s=s+ord(lettre)-96
     22     return s

TypeError: 'int' object is not iterable
```

Exemple 3:

Il est possible d'intégrer les tests que doit réussir la fonction dans son prototype. Reprenons la fonction `fibonacci(n)` . A la fin de sa description, plaçons-y quelques tests :

In [113]:

```
def fibonacci(n):
    """
    Liste des n premiers termes de la suite de Fibonacci

    Paramètre :
    -----------
    n entier supérieur à 1, l'indice maximal voulu

    Return :
    --------
    La liste des n termes

    >>> fibonacci(5)
    [1, 1, 2, 3, 5]

    >>> len(fibonacci(10))
    10

    """
    assert(type(n)==type(0)), 'n doit être un entier'
    assert(n>0), 'n doit être strictement positif'

    x=1
    y=1
    f=[x]
    i=1
    while i < n:
        x,y=y,x+y
        f.append(x)
        i=i+1

    return f
```

Puis à l'aide du module `doctest`, effectuons ces tests. Les deux tests sont passés avec succès.

In [115]:

```
import doctest
doctest.testmod()
```

Out[115]:

```
TestResults(failed=0, attempted=2)
```

Exercice 7 :

Ecrire une fonction qui inverse l'ordre des caractères d'une chaîne quelconque.La spécifier, garantir ses pré-conditions et inscrire quelques tests dans son prototype. Vérifier ensuite que ces tests passent.

In [8]:

```
#7
def inverse(mot):
    """inverse l'ordre des caractères d'un chaîne
    Paramètre :
    -----------
    mot une chaîne de caractères

    Return :
    --------
    Une chaîne de caractères

    >>> inverse('azerty')
    'ytreza'

    >>> inverse('a')
    'a'

    >>> inverse('')
    ''
    """
```

Conclusion :

- Ce type de rédaction peut paraître long et rébarbatif de prime abord.
- Néanmoins, sur des projets informatiques réalisés en équipe, qui comportent des centaines, voire des milliers de lignes de code, il est primordial d'être organisé et efficace.
- C'est pourquoi ces bonnes pratiques de programmation sont importantes(parmi d'autres...) :
  - Choisir des noms de variables explicites.
  - Commenter ses programmes.
  - Spécifier ses fonctions, dès qu'on les écrit.
  - Disposer d'un jeu de tests.

Pour terminer

Deux modules un peu spéciaux de python...à méditer.

In [ ]:

```
import this
```

In [105]:

```
import antigravity
```

FIN