Chapitre 10 : Méthodes de programmation, Commentaires

I Commentaires et signature

I.1 Commentaires

Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent. Un commentaire en Python débute par le hashtag #.

```
for l in L:
    print(l)    # affiche tous les éléments de la liste L
```

Lorsqu'un programme comporte un grand nombre de lignes de code, il est absolument indispensable de placer des commentaires pour pouvoir s'y retrouver.

Un commentaire peut éclairer le rôle un bloc d'instruction ou parfois même une seule instruction. Cela économise beaucoup de temps et d'énergie à la relecture d'un algorithme.

I.2 Signature d'une fonction

Notion de cours (Signature).

La **signature** (ou **en-tête** ou **spécification**) d'une fonction est un bloc de commentaires décrivant le rôle de la fonction, le type de chaque argument en entrée et en sortie. A la différence du corps de la fonction, la signature ne produit aucun résultat. Elle est destinée à l'utilisateur et n'est pas utilisée par l'interpreteur Python.

En Python, la signature se délimite entre """ et """.

```
def val_abs(x):
    """ calcule la valeur absolue du flottant x
    entrée : x (float)
    sortie : float
    """
    if x < 0:
        return
    else:
        return
```

Noter que pour préciser la signature d'une fonction, on peut aussi donner les deux lignes suivantes

```
def val_abs(x: float) -> float:
    """ calcule la valeur absolue du flottant x """
```

La signature permet de renseigner précisément sur l'action d'une fonction. On peut afficher la signature d'une fonction à l'aide de la commande `help(nom_fonction)`.

```
>>> help(val_abs)
Help on function val_abs in module __main__:

val_abs(x:float) -> float
    calcule la valeur absolue du flottant x
    entrée : x (float)
    sortie : float
```

I.2.1 Exercice : calcul d'un minimum

Compléter le corps des fonctions suivantes :

```
def min_deux(a: int, b: int) -> int:
    """ renvoie le minimum entre a et b
    -> version avec 1 return
    """
```

```
def min_deux_2(a: int, b: int) -> int:
    """ renvoie le minimum entre a et b
    -> version avec 2 return
    """
```

```
def min_trois(a:int,b:int,c:int)->int:
    """ renvoie le minimum entre a, b et c
    -> version avec 4 return sans appel
       à min_deux
    """
```

```
def min_trois_2(a:int,b:int,c:int)->int:
    """ renvoie le minimum entre a, b et c
    -> version avec 2 appels à min_deux
    """
```

I.3 Assertion

Certaines fonctions nécessitent une ou plusieurs préconditions sur les arguments afin de donner le résultat attendu et/ou éviter une erreur d'exécution.

```
def division(x: int, y: int) -> int:
    return x // y
```

```
>>> division(2, 0)

ZeroDivisionError: integer division or
modulo by zero
```

Pour éviter que l'utilisateur fournisse un argument en dehors du domaine d'utilisation, on peut utiliser l'instruction `assert e` où `e` est une expression booléenne. Si `e` est fausse alors le **programme s'arrête et alerte** l'utilisateur.

```
def division(x: int, y: int) -> int:
    assert y != 0
    return x // y
```

```
>>> division(2, 0)

AssertionError
```

On peut même alerter l'utilisateur sur l'erreur qu'il a commise en le précisant en commentaire.

```python
def division(x: int, y: int) -> int:
    assert y != 0, 'division par 0 impossible'
    return x // y
```

```
>>> division(2, 0)
AssertionError: division par 0 impossible
```

II Exercices

Exercice 1
La fonction `moyennes` prend en paramètre une liste de nombres qui est non vide.

```python
def moyennes(liste):
  n=len(liste)
  # initialisation d'une liste lisse avec la première valeur de la liste
  lisse=liste[0]
  for i in range(n-1):
    # moyenne de deux éléments consécutifs
    m=(liste[i]+liste[i+1]/2)
    lisse.append(m)
  return lisse
```

1. Utiliser le commentaire pour corriger le code de la fonction
2. Ecrire la signature de la fonction de deux manières différentes
3. Utiliser une ou plusieurs assertions pour vérifier que l'utilisateur entre bien une liste non vide.

Exercice 2
La fonction `lissage` prend en paramètre une liste de nombres qui est non vide.

```python
def lissage(liste):
  n=len(liste)
  # initialisation d'une liste lisse avec la première valeur de la liste
  lisse=[liste[0]]
  for i in range(1,n-1):
    # ajoute à la liste lisse la moyenne de trois éléments consécutifs
    lisse.append(sum(liste[i-1:i+1])/3)
  lisse.append(liste[n-1])
  return lisse
```

1. Utiliser le commentaire pour corriger le code de la fonction
2. Ecrire la signature de la fonction de deux manières différentes
3. Utiliser une ou plusieurs assertions pour vérifier que l'utilisateur entre bien une liste non vide.

Exercice 3
1. Ecrire une fonction `ajoute` qui prend en paramètres deux listes de nombres de même longueur et renvoie une liste construite en effectuant la somme terme à terme des éléments des deux listes.
2. Ecrire la signature de cette fonction
3. Utiliser l'instruction `assert` pour s'assurer que l'utilisateur entre bien deux listes de la même taille.