TP N°01 :
REGLES DE PROGRAMMATION, ASSERTION

Exercice 1 : Analyse de code

Le code ci-dessous semble correct et pourtant renvoie un message d'erreur lorsqu'on l'exécute.

Q1. Sans l'implémenter dans votre console Python expliquez pourquoi le code ne renvoie pas ce que l'on croit et proposez une modification.

L'erreur provient du fait que la fonction f ne comporte aucun return donc que l'affectation Res = f(L) va provoquer une erreur car il n'y a aucune variable renvoyée par la commande f(L). On peut corriger de deux façons :

En ajoutant un « return » dans f

```
def f(L):
      s = 0
      for t in L:
            s += t
      L. append (s)
      return L

def g(L):
      Res = f(L)
      return Res

L = [1 ,2 ,3]
Res = g(L)
```

On n'ajoute pas de return mais on remplace Res = f(L) par f(L) dans g puis Res=L (L sera quand même modifiée par effet de bord) :

```
def f(L):
      s = 0
      for t in L:
            s += t
      L. append (s)

def g(L):
      f(L)
      Res = L
      return Res

L = [1 ,2 ,3]
Res = g(L)
```

Q2. Compléter ces fonctions avec la signature de type et une documentation. La documentation de la fonction `f` s'affiche avec l'instruction `>>> help(f)`.

```
def f(L:list)-> list:
      """
      Fonction qui calcule la somme des termes d'une liste et ajoute la somme à
la fin de la liste

      Entrée : liste de nombres
      Sortie : liste de nombres

      === jeux de tests ===
      f([1,2,3])
      >>> [1,2,3,6]

      """
      s = 0
      for t in L:
            s += t          # 1 affectation + 1 addition
      L. append (s)
      return L
```

Q3. Déterminer l'ordre de grandeur du nombre d'opérations élémentaires réalisées par les fonctions `f` et `g` en fonction du nombre n d'éléments dans la liste L (on parle de taille d'instance).

Il y a deux opérations élémentaires par itération de boucle « for », or il y a n itérations Soit une complexité de 2n. La complexité temporelle est en $\mathcal{O}$(n).

Exercice 2 : Sans doublon

Q1. Ecrire une fonction `SansDoublons` qui prend en entrée une liste d'éléments L et renvoie une nouvelle liste contenant exactement une seule fois chaque élément de L.

Q2. Ajouter une commande d'assertion qui vérifie que la variable d'entrée est bien une liste.

```
def SansDoublons (L: [any] )-> [any] :
      """
      Renvoie une liste   comportant les mêmes   éléments que la liste   entrée
mais   sans   doublons
      Entrée : une liste d'éléments de type quelconque
      Sortie : liste d'éléments de type quelconque ne contenant pas de doublons

      ===Jeux de tests ===
      SansDoublons([1,1,2,3])
      >>> [1,2,3]

      SansDoublons((1,1,2,3))
      >>> AssertionError

      SansDoublon(['bla',bla','bla'])
      >>>['bla']

      """

      assert type (L) == list

      L2 =[]           # une affectation
      for x in L:      # n itération de la boucle
            if not x in L2:       # un test + not
                  L2.append (x)   # une affectation + une concaténation
      return L2
```

Q3. Pour une liste L de taille *n* fixée, dans quel cas aurait-on la pire complexité en temps ? La meilleure complexité en temps ?

Si une liste L contient n éléments identiques alors la complexité est de 2n + 1. Au contraire si la liste ne contient que des éléments différents la complexité est de 4n+1. Une liste d'éléments identiques constitue donc le meilleur des cas pour une instance de taille n, alors qu'une liste d'éléments différents constitue le pire des cas.
Rq : il est également possible de s'intéresser à la complexité moyenne, mais cela devient rapidement compliqué. En CPGE, on se concentrera uniquement sur la complexité dans le pire des cas pour comparer les algorithmes.

Q4. Déterminer l'ordre de grandeur du nombre d'opérations élémentaires réalisées par la fonction `SansDoublons` en fonction du nombre n d'éléments dans la liste.

La complexité est de l'ordre de n, il s'agit d'une complexité linéaire. (Il est donc peu pertinent dans ce cas de distinguer complexité dans le meilleur des cas et dans le pire des cas car elles sont du même ordre de grandeur).

Exercice 3 : Où sera le lac ?

```
import numpy as np
import matplotlib.pyplot as plt
```

On se trouve dans une vallée où l'altitude au point de coordonnées entières (x ; y) est stockée dans un tableau 2D appelé **alt**. Ainsi, si **alt[3][8]=10** cela signifie qu'au point de coordonnées (3 ; 8), l'altitude est de 10 mètres.
On souhaite trouver un endroit où l'eau formera un petit lac quand il se mettra à pleuvoir. On cherche donc des coordonnées (x; y) telles que l'altitude en (x; y) soit plus petite que celle de ses 8 cases voisines (x-1; y-1), (x; y - 1), (x + 1; y - 1) .. etc.
On suppose que l'on sait que l'on est dans une vallée creuse et qu'un tel point existe. L'algorithme que nous allons utiliser consistera à partir d'un point de la carte, et à avancer tant qu'on peut à chaque pas vers une case voisine d'altitude strictement inferieure à l'altitude de la case où l'on se trouve.

Q1. Créer une fonction **altitude** prenant en argument un entier A et renvoyant un tableau de A lignes et B colonnes tel que :

$$\forall i \in [\![0, A]\!] \quad et \; \forall j \in [\![0, B]\!] \quad , alt[i][j] = i * i + i * j - 60 * i + 1.2 * j * j - 90 * j + 2180$$

```
def altitude(A:int,B:int) -> [[int]]:
    """
    fonction permettant d'obtenir un tableau des altitudes en fonction des coordonnées d'un point sur une carte de dimension données A par B

    entrée : deux entiers A et B correspondants aux dimensions de la carte

    sortie : liste de listes de flottant alt[i][j] correspondant aux altitudes des points de coordonnées (i,j)

    """
    alt=[]
    for i in range (A):
        ligne=[]
        for j in range(B) :
            ligne.append(i*i+i*j-60*i+1.2*j*j-90*j+2180)
        alt.append(ligne)
    return(alt)
```

Q2. Ecrire une liste d'instruction permettant d'obtenir un tableau des altitudes **alt** de 100 par 100.

```
alt = altitude(100,100)

# ou pour obtenir sous forme d'un tableau
alt = np.array(altitude(100,100))
```

Q3. Ecrire une fonction **deplace** qui prend en argument un tableau **alt** contenant les altitudes et deux entiers i et j représentant les coordonnées (i, j) d'un point sur la carte. Cette fonction renvoie :

- Une liste [newI,newJ] contenant les coordonnées d'UNE case voisine de (i, j) dont l'altitude est strictement inférieure à celle de la case (i, j) lorsque c'est possible ;
- -1 si cela n'est pas possible, c'est-à-dire si la case (i; j) est d'altitude minimale parmi ses voisines.

```
def deplace(alt,i,j)->[int]|int:
    """
    fonction permettant de déterminer à partir d'un point de coordonnées (i,j) un point adjacent d'altitude inférieure (s'il existe)

    entrée : tableau des altitudes [[int]] , et des entiers i et j correspondant aux coordonnées du point de départ

    sortie : liste contenant deux entiers correspondant aux coordonnées d'un point adjacent d'altitude inférieure, -1 s'il n'existe pas de point adjacent d'altitude inférieure.

    ====
    jeux de tests
    alt_exemple = [[2,3,3,4],[1,2,3,5],[2,3,3,4],[2,2,3,5]]

    deplace(alt_exemple,2,2)
    >>>[1,1]

    """

    D=[-1,0,1]              # 1 affectation
    for di in D :           # 3 itérations
        for dj in D :       # 3 itérations
            newI = i + di         # 1 affectation + 1 somme
            newJ= j + dj          # 1 affectation + 1 somme
            if alt[newI][newJ] < alt[i][j]:      # 1 test
                return([newI,newJ])
    return(-1)        # return
```

Q4. Ecrire une fonction **futur_lac** reprenant l'algorithme proposé dans l'énoncer pour trouver l'emplacement du futur lac en partant d'une carte **alt** et du point de coordonnées (Di ;Dj).

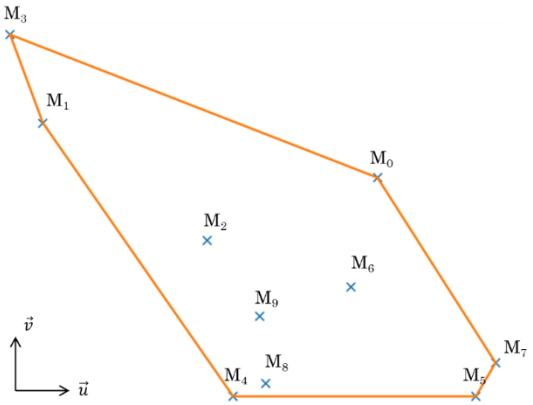
```
def futur_lac(alt:[[int]],Di:int,Dj:int)->(int,int):
    """
    renvoie la position du futur lac à partir du point de départ (Di,Dj)
    utilise la fonction deplace

    renvoie False s'il n'existe pas de point susceptible d'accueillir un lac

    === jeux de tests ===
    alt_exemple = [[2,3,3,4],[1,2,3,5],[2,3,3,4],[2,2,3,5]]

    futur_lac(alt_exemple,3,3)
    >>> [1, 0]

    futur_lac(alt_exemple,0,0)
    >>> [1, 0]

    futur_lac([[1,1,1],[1,1,2],[1,1,1]],0,0)
    >>> False
    """
    #initialisation
    i=Di
    j=Dj
    if deplace(alt,i,j)==-1 :
        return(False)
    while deplace(alt,i,j)!=-1:
        lac = deplace(alt,i,j)
        i=lac[0]
        j=lac[1]
    return (lac)
```

Q5. Trouver l'emplacement d'un futur lac sur la carte **alt**.

```
print(futur_lac(alt,50,50))
>>> [14,32]
```

Exercice 4 : Enveloppe convexe d'une famille de points

Nous considérons une famille $(M_i)_{0\leq i<n}$ de points distincts d'un plan dans le repère orthonormé direct $(O,\vec{u},\vec{v})$. Chaque point est représenté par un tuple de coordonnées (x_i,y_i). La famille de points$(M_i)_{0\leq i<n}$ est représentée par une liste $L = [(x_0,y_0),\dots,(x_{n-1},y_{n-1})]$.



L'enveloppe convexe $\mathcal{C}$ de cette famille est le plus petit polygone convexe contenant tous les points M_i. Le but de cet exercice est de calculer la liste E dite « bien ordonnée » des sommets de $\mathcal{C}$, c'est-à-dire la liste des sommets $\mathcal{S}$ parcourus dans le sens trigonométrique direct, en commençant par le point M_{i0} situé le plus bas parmi ceux situés le plus à gauche. Ainsi avec les points représentés sur la figure ci-dessous, $i_0 = 3$ et nous cherchons à construire la liste E = $[M_3, M_1, M_4, M_5, M_7, M_0]$.

Pour s'épargner le traitement de cas particuliers, nous supposerons que trois points quelconques de la famille $(M_i)_{0\leq i<n}$ ne sont jamais alignés.

Si A, B et C sont trois points non alignés du plan, nous dirons que « (A, B, C) tourne gauche » si l'angle entre $\overrightarrow{AB}$ *et* $\overrightarrow{AC}$ admet une mesure entre $]0,\pi[$.

Q1. Ecrire une fonction `tourne_gauche` qui prend en argument trois points A, B et C et qui renvoie le booléen `True` si (A, B, C) tourne gauche et `False` sinon.

En calculant le produit vectoriel entre deux vecteur $\overrightarrow{AB} = \begin{pmatrix} x_u \\ y_u \\ 0 \end{pmatrix}$ et $\overrightarrow{AC} = \begin{pmatrix} x_v \\ y_v \\ 0 \end{pmatrix}$.

$$\overrightarrow{AB} \wedge \overrightarrow{AC} = \begin{pmatrix} 0 \\ 0 \\ x_u.y_v - y_u.x_v \end{pmatrix} = ||\vec{u}||.||\vec{v}||.\sin(\widehat{\vec{u},\vec{v}})\ \overrightarrow{e_w}$$

Si la valeur du sinus de l'angle est positive alors l'angle appartient à $]0;\pi[$.
On teste donc $x_u.y_v - y_u.x_v > 0$.

```
def tourne_gauche (A:(int,int),B:(int,int),C:(int,int))-> bool :
    """
    Détermine si l'angle formé par les vecteurs AB et AC est compris entre 0 et
pi
    entrée : coordonnées des points non-alignés

    sortie : True si l'angle appartient à l'intervalle ouvert 0,pi
             False sinon

    === jeux de tests ===
    # angle droit entre les vecteurs
    A=(0,0)
    B=(4,2)
    C=(-2,4)
    tourne_gauche(A,B,C)
    >>> True

    #angle nul entre les vecteurs
```

```
    A=(0,0)
    B=(4,2)
    C=(4,2)
    tourne_gauche(A,B,C)
    >>> False

    #angle compris entre 0 et pi entre les vecteurs
    A=(0,0)
    B=(3,1)
    C=(2,5)
    tourne_gauche(A,B,C)
    >>> True
        """
    #condition sur le sinus de l'angle

    return((B[0]-A[0])*(C[1]-A[1]) - (B[1]-A[1])*(C[0]-A[0])>0)
```

Q2. Ecrire une fonction **point_depart** qui, à partir d'une liste L de coordonnées de points renvoie l'indice i_0 du point de départ du tracé de l'enveloppe convexe.

```
def point_depart (L:[tuple])->int:
    """
    à partir d'une liste L de coordonnées de points renvoie l'indice i_0 du point de départ du tracé de l'enveloppe convexe

    entrée : une liste de tuples [(xi,yi)] représentant les coordonnées des points d'un plan

    sortie : indice i0 du point de départ de coordonnées (x0,y0), i.e. le point le plus bas des points les plus à gauche

    === jeux de tests ===

    L_exemple1 = [(1,1),(2,1),(3,1)]
    L_exemple2 = [(1,1),(1,2),(1,3)]
    L_exemple3 = [(1,3),(1,2),(1,1)]

    point_depart(L_exemple1)
    >>> 0
    point_depart(L_exemple2)
    >>> 0
    point_depart(L_exemple3)
    >>> 2

    """
    i0 = 0

    for i in range (1,len(L)):
        if L[i][0]<L[i0][0] or(L[i][0]==L[i0][0] and L[i][1]<L[i0][1]) :
            i0 = i
    return i0
```

Q3. Ecrire une fonction **tri_sommets** qui, appliquées à L et i_0, renvoie la pile* P des points M_i pour $i \neq i_0$, triée dans l'ordre décroissant des angles que font les vecteurs $\vec{u}$ *et* $\overrightarrow{M_{i0}M_i}$. On utilisera la fonction **tourne_gauche**.

```
def tri_sommets (L:[tuple],i0:int)-> list :
    """
    Permet d'obtenir une liste des coordonnées de points Mi triées par valeur
décroissante des angles entre les vecteurs Mi0Mi et le vecteur unitaire u.

    On utilise la fonction tourne_gauche

    entrée
    une liste [(x,y)] contenant les coordonnées des points
    un entier i0 correspondant au point de départ

    sortie : liste de tuples triées

    """
    P= []
    for i in range(len(L)):      # n itérations
        if i!=i0 :               # 1 test
            D=[]                 # 1 affectation
            while P!=[] and tourne_gauche(L[i0],P[-1],L[i]):# max i itérations
                # on dépile jusqu'à trouver la place de L[i] dans P
                D.append(P.pop())                # dépiler + 1 ajout dans D
            P.append(L[i])
            while D!=[]:    # max i itérations
                #on déverse dans P ce que l'on avait versé dans D
                P.append(D.pop())                # dépiler + 1 ajout dans P
    return P
```

Q4. Ecrire une fonction **enveloppe_convexe** qui, appliquée à L, renvoie la liste E en utilisant la méthode suivante :

a. On calcule i_0 puis P

b. On initialise une pile E qui ne contient que M_{i0} au début du calcul

c. On vide P en modifiant E de sorte que chaque tour de boucle, E soit la liste bien ordonnée des sommets de l'enveloppe convexe des points de L qui ne sont pas présents dans P.

```
def enveloppe_convexe (L:[tuple])-> [tuple] :
    """
    Permet d'obtenir la liste des coordonnées des sommets de points constituant
l'enveloppe convexe associées à une famille de points

    entrée : liste [(x,y)] contenant les coordonnées des points d'une famille de
points Mi

    sortie : liste [(xe,ye)] des coordonnées des points de l'enveloppe convexe.

       """
    # étapes (a) et (b)
```

```
    i0 = point_depart(L)
    P = tri_sommets(L,i0)

    #initialisation
    E = [L[i0],P.pop()]

    # étape (c)
    while P!= [] :
        C = P.pop()         # une affectation + dépiler
        B = E[-1]           # une affectation
        A = E[-2]           # une affectation
        while not tourne_gauche(A,B,C) :
            E.pop()         # dépiler
            A,B=E[-2],A     # 2 affectations
        E.append(C)         # un ajout
    return E
```

Q5. Déterminer l'ordre de grandeur du nombre d'opération (complexité) de l'étape (c) en fonction du nombre de points n. Puis déterminer la complexité de la fonction complète `enveloppe_convexe`. Peut-on améliorer cette complexité ?

L'étude de la complexité peut paraitre assez difficile ici car les deux boucles imbriquées sont difficiles à étudier. Cependant en analysant de façon globale on constate que chaque élément est dépilé (pop) une seule fois de la pile P, ce qui représente n opérations élémentaires. Dans le pire des cas, il faudrait y ajouter de l'ordre 4n opérations avec la deuxième boucle while. Mais dans tous les cas on obtient une complexité linéaire pour l'étape (c) ($\mathcal{O}$(n)).

La complexité de la fonction **`tri_sommet`** est de l'ordre de $(1 + \cdots + i + \cdots n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. La complexité est donc de l'ordre de n^2 (quadratique).
La complexité totale est donc également quadratique. Pour améliorer la complexité de la fonction, il faudrait améliorer celle de l'étape de tri. Par exemple, un tri fusion aurait une complexité en $\mathcal{O}(nln(n))$.

Q6. Ecrire une fonction **`tracer`** qui prend en argument un entier n, simule le tirage de points de coordonnées (x_i, y_i), et trace le nuage de ces points ainsi que le contour de leur enveloppe convexe.

```
def tracer (n:int) :
    """
    génère une tirage aléatoire de n points de coordonnées (xi,yi) et trace le
nuage de points avec le contour de l'enveloppe convexe.

    entrée : un entier n

    sortie : la fonction ne retourne rien

    """
    L = [(round(rd.random(),2),round(rd.random(),2)) for i in range (n)]
    print(L)
    x,y = [L[i][0] for i in range (len(L))],[L[i][1] for i in range (len(L))]
```

```
    plt.plot(x,y,"x")


    E = enveloppe_convexe(L)
    E.append(E[0])

    X,Y = [E[i][0] for i in range (len(E))], [E[i][1] for i in range (len(E))]

    plt.plot(X,Y)
    plt.show()
```