? À rendre mercredi 17 septembre 2025

Devoir Maison n°1 : Reprise en main de python – *Corrigé*

Remarque

- `range(a,b)` renvoie les entiers compris entre `a` INCLUS et `b` EXCLUS (c'est-à-dire `b-1` INCLUS). `range(0,4)` renvoie donc 0, 1, 2, 3.
- La phrase « pour i allant de 0 à n inclus » se traduit en `for i in range(0,n+1):`
- La phrase « pour i allant de 1 à n inclus » se traduit en `for i in range(1,n+1):`
- `for i in range(1,n):` se traduit en « pour i allant de 1 à n EXCLUS, soit de 1 à $n-1$ INCLUS »
- La structure suivante ne peut JAMAIS être utilisée :

```
for ... : # quoiqu'il arrive ça sort à la 1ère itération
    if .... :
        return ... # fait sortir de la fonction si if vrai dès la 1ère itération
    else :
        return ... # fait sortir de la fonction si if faux dès la 1ère itération
```

- Les programmes doivent être COMMENTÉS : il ne faut pas paraphraser chaque ligne en français, mais expliquer les grandes idées (sauf programmes évidents car très courts).

Exercice n°1 Autour des listes ♪

R1. Écrivez une fonction `testTri(L:list)->bool` qui prend comme argument une liste `L` d'entiers et teste si cette liste est triée en ordre croissant.

Solution:

```
def testTri(L):
    for i in range(len(L)-1):
        if L[i+1]<L[i]:
            return False
    return True
```

On étudie les températures à Valence à 12h00 durant les mois de juillet-août. Pour cela nous disposons de deux listes, la liste `Temp` des 62 températures (flottants) dans l'ordre chronologique du 1er juillet au 31 août, et de la liste `Jour` avec les dates dans l'ordre chronologique (chaînes de caractères au format `"AAAA-MM-JJ"`.

R2. Écrire la fonction `temp_max(L:list)->float` qui prend en entrée une liste de flottants et renvoie la valeur maximale.

Écrire la ligne permettant de récupérer la température maximale à Valence de cet été.

Solution:

```
def temp_max(L):
    M=L[0]
    for i in range(1,len(L)):
        if L[i]>M: # nouveau maximum local
            M=L[i]
    return M
>>> temp_max(Temp)
```

R3. Écrire la fonction `moyenne(L:list)->float` qui prend en entrée une liste de flottants et renvoie la valeur moyenne.

Solution:

```
def moyenne(L):
    assert type(L)==list
    M=0
    for i in range(len(L)):
        M=M+L[i]
    return M/len(L)
```

R4. Écrire la fonction `ecart-type(L:list)->float` qui prend en entrée une liste de flottants et renvoie l'écart-type, défini par $\sigma = \sqrt{\dfrac{1}{N}\sum_{i=0}^{N-1}(x_i - \overline{x})^2}$.
La fonction devra obligatoirement ne faire appel qu'une fois à la fonction `moyenne` et être de complexité temporelle linéaire (en $O(N)$).

Solution:

```
from math import *
def ecart-type(L):
    S=0
    moy=moyenne(L)
    for i in range(len(L)):
        S=S+(L[i]-moy)**2
    return sqrt(S/len(L))
```

R5. Écrire la fonction `jour_min(T:list,J:list)->str` prenant en entrée deux listes `T` et `J` du type de `Temp` et `Jour` et qui renvoie le jour où la température a été minimale.

Solution:

```
def jour_min(T,J):
    imin=0 # rang de la température minimale
    for i in range(len(T)):
        if T[i]<T[imin]: # température plus petite trouvée
            imin=i
    return J[imin]
```

R6. Écrire la fonction `sup(T:list,val:float)->bool` qui teste si au moins un élément de la liste de flottants `T` dépasse la valeur `val`.

Solution:

```
def sup(T,val):
    for i in range(len(T)):
        if T[i]>val:
            return True
    return False
```

R7. Écrire la fonction `liste_sup(T:list,J:list,val:float)->bool` qui renvoie la liste des jours de la liste `J` pour lesquels la température était strictement supérieure à `val`.

Solution:

```
def liste_sup(T,val):
    L=[]
    for i in range(len(T)):
        if T[i]>val:
            L.append(J[i])
    return L
```

R8. Écrire une fonction `taille(L:list[list])->(int,int)` qui prend en argument une liste de listes (un tableau) et renvoie un 2-uplet qui contient en premier élément le nombre de lignes et en deuxième le nombre de colonnes du tableau.
Aucune boucle `for` n'est nécessaire (et donc aucune ne devra apparaître).

Solution:

```
def taille(L):
    assert type(L)==list
    assert len(L)>0
    assert type(L[0])==list
    return (len(L),len(L[0]))
```

R9. Écrire une fonction `recherche2(L:list[list],val)->bool` qui prend en argument une liste de listes (un tableau) d'entiers et un entier `val` et teste si cet entier apparaît dans une des cases du tableau.

Solution:

```
def recherche2(L,val):
    for i in range(len(L)):
        for j in range(len(L[0])):
            if L[i][j]==val: # val trouvé
                return True # il y est
    return False # le tableau a été entièrement parcouru, val n'a pas
   été trouvé
```

Exercice n°2 Poteaux télégraphiques ♪ ♪

Cette partie a pour objectif de choisir où placer des poteaux télégraphiques pour relier le point le plus à gauche d'un paysage unidimensionnel au point le plus à droite. Pour ce faire, une première partie permet l'étude du terrain alors qu'en deuxième partie, l'installation des poteaux télégraphiques est résolue par deux méthodes différentes.
Les fonctions `min`, `max` et `sum` ne pourront pas être utilisées ici.

Partie I Étude du terrain

Pour décrire le terrain, des géographes ont placé des points P_0, P_1, ..., P_n en ligne droite sur ce terrain, convenu que le premier point avait une altitude nulle, puis ont noté les dénivelés (en mètres), c'est à dire les différences d'altitude, entre deux points successifs. Cette succession de mesures est placé dans une liste de flottants, que l'on appelle `denivele`, qui sera passé en argument de chaque fonction. Dans tout le problème, on convient qu'aucun dénivelé mesuré n'est nul.

R1. Écrire une fonction `calcul_altitude` prenant en argument la liste `denivele` et renvoyant une liste de flottants formée par les altitudes des points successivement repérés sur le terrain.
Ainsi, la fonction écrite, appelée avec la liste décrivant le terrain donné en exemple, doit renvoyer :
`[0.0, 100, -100.0, 0.0, 200.0, 300.0, 500.0, 400.0, 450.0, 400.0, 200.0, 300.0]`.

Solution:

```
def calcul_altitude(denivele):
    altitude=[0] # P0 d'altitude nulle
    for i in range(1,len(denivele)):
        altitude.append(altitude[i-1]+denivele[i])
    return altitude
```

R2. Écrire une fonction `calcul_fenetre` qui prend en argument la liste `denivele` et qui renvoie la hauteur maximale `hMax` et la hauteur minimale `hMin` ainsi que les indices `iMax` et `iMin` correspondant.
Ainsi, la fonction `calcul_fenetre` appelée pour notre exemple renvoie les deux tuples $(500.0, 6)$ et $(-100.0, 2)$.

Solution:

```
def calcul_fenetre(denivele):
    hauteurs=calcul_altitude(denivele) # calcul des altitudes
    hMax,hMin,iMax,iMin=hauteurs[0],hauteurs[0],0,0 # initialisation
   des max/min locaux
    for i in range(1,len(hauteurs)):
        if hauteurs[i]>hMax: # nouveau maximum local
            hMax=hauteurs[i]
            iMax=i
        elif hauteurs[i]<hMin: # nouveau minimum local
            hMin=hauteurs[i]
            iMin=i
    return hMax,hMin,iMax,iMin
```

R3. Écrire une fonction `calcul_longueur` prenant en argument la liste `denivele` et renvoyant un flottant égal à la longueur en kilomètres du trajet parcouru « en ligne droite » par un piéton marchant du point P_0 jusqu'au le point P_n, en suivant bien évidemment le relief du terrain. Autrement dit, ce flottant est la longueur de la ligne brisée passant par les points P_0, P_1, ..., P_n dans cet ordre. Les points sont relevés tous les 100 mètres.

Solution: Un segment $[P_k, P_{k+1}]$ est de longueur $\sqrt{100^2 + \text{denivele}[k]^2}$ (d'après le théorème de Pythagore).
Il faut sommer ces distances d'un bout à l'autre (de P_0 à P_n).

```
def calcul_longueur(denivele):
    d=0
    for k in range(len(denivele)):
        d=d+sqrt(100**2 + denivele[k]**2)
    return d*10**-3 # en km
```

Le point d'indice $i \in [\![1, n-1]\!]$ est dit remarquable si et seulement c'est un pic, c'est-à-dire son altitude est plus élevée que celles des points d'indice $i-1$ et $i+1$. On convient de plus que les points d'indice 0 et n sont remarquables.

On appelle alors bassin, toute suite finie de points consécutifs dont le premier et le dernier éléments sont des points remarquables et dont aucun autre point n'est remarquable. Si on reprend le profil de terrain donné au début de cette partie :

— Les points remarquables sont ceux d'indice 0, 1, 6, 8 et 11.

— Les bassins sont formés par les points dont les indices forment les listes [0,1], [1,2,3,4,5,6], [6,7,8] et [8,9,10,11]. La longueur d'un bassin est le nombre d'éléments de la liste des points appartenant au bassin. Dans notre exemple, le bassin le plus long est le deuxième, et sa longueur est 6.

R4. Écrire une fonction `remarquable` prenant en argument la liste `denivele` et l'indice `i` d'un point et renvoyant un booléen égal à `True` si et seulement si le point d'indice i est remarquable.

Solution:

```
def remarquable(denivele,i):
    """
    denivele : liste et i : entier, rang dans la liste, n° du point
    retour : booléen
    """
    if i==0 or i==len(denivele)-1 :
        return True
    elif denivele[i]>=0 and denivele[i+1]<=0 :
        return True
    return False
# plus condensé :
def remarquable(denivele,i):
    return i==0 or i==len(denivele)-1 or (denivele[i]>=0 and denivele[i
   +1]<=0)
```

R5. Écrire une fonction `bassin_max` prenant en argument la liste `denivele` et renvoyant un entier égal à la longueur du plus long bassin du terrain étudié.

Solution:

```
def bassin_max(denivele):
    ideb = 0 # indice début du bassin (0 est un point remarquable)
    maxi = 0 # longueur maximale du bassin
    for i in range(1,len(denivele)):
        if remarquable(denivele,i): # début du bassin suivant / fin du
   bassin précédent
            lg = i-ideb+1 # longueur du bassin en cours de parcours
            if lg > maxi: # longueur sup à la lg max du bassin
   précédent
                maxi = lg # mise à jour de la lg max
            ideb = i # on met à jour l'indice du début du bassin
    return maxi
```

Partie II Installation des poteaux

On souhaite relier le point le plus à gauche au point le plus à droite du terrain. Nous devons donc choisir à quels points parmi les $n+1$ sélectionnés, nous allons planter des poteaux télégraphiques, sachant qu'il y en a évidemment un placé au point d'indice 0, et un placé au point d'indice n.

On suppose que :

— Les poteaux ont tous la même hauteur, qui sera appelée h dans le problème,

— Les fils sont sans poids et tendus, donc relient en ligne droite les sommets de deux poteaux consécutifs.

— Les normes de sécurité imposent que les fils soient en tout point à une distance minimale du sol (mesurée verticalement), que l'on notera Δ et que l'on suppose bien évidemment plus petite que h.

Pour tout $i \in [\![0;n]\!]$, on note h_i, l'altitude du poteau d'indice i.

Pour tout $(i,j) \in [\![0;n]\!]^2$ tel que $i \neq j$, on pose alors : $\alpha_{i,j} = \dfrac{(h_j+\Delta)-(h_i+h)}{j-i}$ et $\beta_{i,j} = \dfrac{h_j-h_i}{j-i}$

Le premier réel $\alpha_{i,j}$ est la pente d'un fil tendu entre le sommet d'un poteau planté au point d'indice i et le point situé à Δ mètres au-dessus du point d'indice j. Le second réel $\beta_{i,j}$ est la pente d'un fil tendu entre les sommets de poteaux plantés respectivement au point d'indice i et au point d'indice j. On admet que le fil tendu entre deux poteaux placés aux points d'indice i et j tel que $j > i$ vérifie les normes de sécurité si $\beta_{i,j} \geq \text{Max}\{\alpha_{i,k}, k \in [\![i+1, j]\!]\}$

R6. Écrire une fonction `correct` prenant en argument la liste `denivele`, la hauteur `h` des poteaux, la distance de sécurité `Delta`, deux indices `i` et `j` de points tels que $j > i$, et renvoyant un booléen égal à `True` si et seulement si un fil tendu entre les poteaux d'indice i et j vérifie les normes de sécurité.

Solution:

```
def correct(denivele,h,Delta,i,j):
    assert j>i
    hauteurs = calcul_altitude(denivele)
    betaij=(hauteurs[j]-hauteurs[i])/(j-i)
    # Recherche de la valeur maximale de alpha_ik
    alpha_max=(hauteurs[i+1]+Delta-(hauteurs[i]+h))/(i+1-i) # 1er max
   local
    for k in range(i+2,j):
        alphaik=(hauteurs[k]+Delta-(hauteurs[i]+h))/(k-i)
        if alphaik>alpha_max: # nouveau max local
            alpha_max=alphaik
    # vérification de la condition
    if betaij>=alpha_max:
        return True
    else:
        return False
```

Considérons une première stratégie pour planter les poteaux, dite algorithme glouton en avant.

Le premier poteau est planté en P_0. Lorsqu'un poteau est placé, pour calculer l'emplacement du poteau suivant, on part du dernier poteau planté et on avance (à droite) avec le fil tendu tant que les normes de sécurité sont respectées et que P_n n'est pas atteint. Un nouveau poteau est alors planté, et on recommence jusqu'à ce que P_n soit atteint.

La figure qui suit illustre la solution produite par cet algorithme, la zone grisée étant celle dans laquelle ne doit passer aucun fil. Les poteaux et les fils en pointillés sont les premiers violant les normes de sécurité lors de la recherche de l'emplacement des poteaux.

R7. Écrire une fonction `glouton_avant` prenant en arguments la liste `denivele`, la hauteur `h` des poteaux, la distance de sécurité `Delta`, et renvoyant la liste des indices des points en lesquels doivent être placés des poteaux lorsque l'on suit la stratégie proposée.
Avec notre exemple, la liste renvoyée est donc [0, 1, 6, 9, 11].

Solution:

```
def glouton_avant(denivele,h,Delta):
    poteaux=[0] # 1er poteau placé en 0
    i=0 # poteau  de départ
    j=1 # poteau d'arrivée
    while i<=len(denivele): # on n'est pas arrivé en Pn
        while j<=len(denivele) and correct(denivele,h,Delta,i,j): #
   tant que la condition de sécurité est vérifiée
            j=j+1 # on avance avec le fil tendu
        # la condition n'est plus vérifiée, on plante le poteau à la
   dernière position où elle était vérifiée
        j=j-1
        poteaux.append(j)
        i=j # on repart du poteau j
        j=j+1  # on tire le fil jusqu'au poteau suivant
    return poteaux
```

L'algorithme présenté a tendance à placer trop de poteaux, en particulier dans les bassins alors qu'il suffirait d'en relier les deux extrémités par un unique fil.

Considérons alors une nouvelle stratégie pour planter les poteaux, dite algorithme glouton en arrière.

Le premier poteau est planté en P_0. Lorsqu'un poteau est placé, pour calculer l'emplacement du poteau suivant, on tend un fil entre le dernier poteau planté et un poteau placé en P_n, puis on recule ce dernier poteau jusqu'à ce que les normes de sécurité soient respectées. Un nouveau poteau est alors planté, et on recommence jusqu'à ce que le dernier point soit atteint.

La figure qui suit illustre la solution produite par cet algorithme, la zone grisée étant celle dans laquelle ne doit passer aucun fil. Les poteaux et les fils en pointillés les derniers violant les normes de sécurité lors de la recherche de l'emplacement des poteaux.

R8. Écrire une fonction `glouton_arriere` prenant en argument la liste `denivele`, la hauteur `h` des poteaux, la distance de sécurité `Delta`, et renvoyant la liste des indices des points en lesquels doivent être placés des poteaux lorsque l'on suit la nouvelle stratégie proposée.
Avec notre exemple, la liste renvoyée est donc `[0, 6, 11]`.

Solution:

```
def glouton_arriere(denivele,h,Delta):
    poteaux=[0] # 1er poteau placé en P0
    i=0
    while i<len(denivele)-1: # tant qu'on n'est pas arrivé en Pn (de
   rang len(denivele)-1 puisque le dernier point)
        j=len(denivele)-1 # on tire le fil depuis Pi jusqu'à Pn
        while j>0 and not correct(denivele,h,Delta,i,j): # tant que la
   condition n'est pas vérifiée
            j=j-1 # on remonte en arrière à la recherche de la position
    la plus à droite qui vérifie la condition
        poteaux.append(j) # on a trouvé la position
        i=j
    return poteaux
```