 Sur le site [e-nsi.gitlab.io/pratique/n1](https://e-nsi.gitlab.io/pratique/N1/) faire les 10 premiers exercices

1. [Remplacer une valeur](https://e-nsi.gitlab.io/pratique/N1/100-remplacer/sujet/)
2. [Maximum](https://e-nsi.gitlab.io/pratique/N1/110-maximum_nombres/sujet/)
3. [Indice première occurrence](https://e-nsi.gitlab.io/pratique/N1/115-ind_prem_occ/sujet/)
4. [Dénivelé positif](https://e-nsi.gitlab.io/pratique/N1/116-denivele_positif/sujet/)
5. [Dernière occurrence](https://e-nsi.gitlab.io/pratique/N1/117-derniere_occurence/sujet/)
6. [Moyenne simple](https://e-nsi.gitlab.io/pratique/N1/118-moyenne/sujet/)
7. [Recherche d'indices](https://e-nsi.gitlab.io/pratique/N1/119-recherche_indices/sujet/)
8. [Soleil couchant](https://e-nsi.gitlab.io/pratique/N1/120-soleil_couchant/sujet/)
9. [Premier minimum local](https://e-nsi.gitlab.io/pratique/N1/122-premier_minimum/sujet/)
10. [Nombres puis double](https://e-nsi.gitlab.io/pratique/N1/128-nb_puis_double/sujet/)

# Remplacer une valeur

Écrire la fonction remplacer prenant en argument :

* une liste d'entiers valeurs
* un entier valeur\_cible
* un entier nouvelle\_valeur

et renvoyant une **nouvelle** liste contenant les mêmes valeurs que valeurs, dans le même ordre, sauf valeur\_cible qui a été remplacé par nouvelle\_valeur.

**La liste passée en paramètre ne doit pas être modifiée**.

def remplacer(valeurs, valeur\_cible, nouvelle\_valeur):

    ...

# Tests

# 1er test

valeurs = [3, 8, 7]

assert remplacer([3, 8, 7], 3, 0) == [0, 8, 7]

assert valeurs == [3, 8, 7]

# 2nd test

valeurs = [3, 8, 3, 5]

assert remplacer([3, 8, 3, 5], 3, 0) == [0, 8, 0, 5]

assert valeurs == [3, 8, 3, 5]

*Réponse*

def remplacer(valeurs, valeur\_cible, nouvelle\_valeur):

    liste = list(valeurs)

    for i in range(len(liste)):

        if liste[i] == valeur\_cible:

            liste[i] = nouvelle\_valeur

    return liste

# Maximum

Écrire une fonction maximum :

* prenant en paramètre une liste **non vide** de nombres : nombres
* renvoyant le plus grand élément de cette liste.

Chacun des nombres utilisés est de type int ou float.

 On interdit ici d'utiliser max, ainsi que sort ou sorted.

def maximum(nombres):

    ...

# Tests

assert maximum([98, 12, 104, 23, 131, 9]) == 131

assert maximum([-27, 24, -3, 15]) == 24

*Réponse*

def maximum(nombres):

    maxi = nombres[0]

    for i in range(1, len(nombres)):

        if nombres[i] > maxi:

            maxi = nombres[i]

    return maxi

# Indice de la première occurrence

Écrire une fonction indice qui prend en paramètres element un nombre entier, tableau un tableau de nombres entiers, et qui renvoie l'indice de la **première** occurrence de element dans tableau.

La fonction devra renvoyer None si element est absent de tableau.

On n'utilisera pas ni la fonction index, ni la fonction find.

def indice(element, tableau):

    ...

# tests

assert indice(1, [10, 12, 1, 56]) == 2

assert indice(1, [1, 50, 1]) == 0

assert indice(15, [8, 9, 10, 15]) == 3

assert indice(1, [2, 3, 4]) is None

*Réponse*

def indice(element, tableau):

    retour = None

    for i in range(len(tableau)):

        if tableau[i] == element and retour == None:

            retour = i

    return retour

# tests

assert indice(1, [10, 12, 1, 56]) == 2

assert indice(1, [1, 50, 1]) == 0

assert indice(15, [8, 9, 10, 15]) == 3

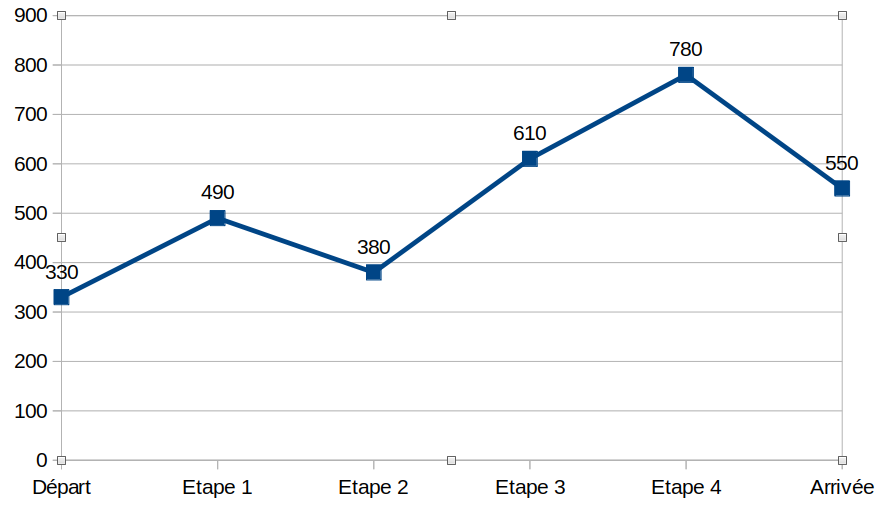
assert indice(1, [2, 3, 4]) is None

print(indice(41, [2,35,42,39,41]))

print(indice(48, [2,35,42,39,41]))

# Calcul du dénivelé cumulé positif d'une course de montagne

Le dénivelé cumulé positif d'une course de montagne est la somme totale des dénivelés de l'ensemble des ascensions durant la course.



Sur l'exemple ci-dessus :

* la course commence par une ascension de dénivelé positif 160 (490−330)
* entre l'étape 2 et l'étape 3, le dénivelé positif est de 230 (610−380)
* entre l'étape 3 et l'étape 4, le dénivelé positif est de 170 (780−610)
* les autres parties de la course sont des descentes

Le dénivelé cumulé positif total de cette course est donc 160+230+170=560

Écrire une fonction denivele\_positif qui prend en argument la liste non vide des altitudes atteintes à la fin de chaque ascension et de chaque descente pendant la course et qui renvoie le dénivelé cumulé positif de cette course.

def denivele\_positif(altitudes):

    ...

# tests

assert denivele\_positif([330, 490, 380, 610, 780, 550]) == 560

assert denivele\_positif([200, 300, 100]) == 100

*Réponse*

def denivele\_positif(altitudes):

    cumul = 0

    for i in range(len(altitudes)-1):

        delta = altitudes[i+1] - altitudes[i]

        if delta > 0 :

            cumul = cumul + delta

    return cumul

# Dernière occurrence

Programmer la fonction derniere\_occurrence, prenant en paramètre un tableau non vide d'entiers et un entier cible, et qui renvoie l'indice de la **dernière** occurrence de cible.

Si l'élément n'est pas présent, la fonction renvoie la longueur du tableau.

On n'utilisera pas la fonction index

def derniere\_occurrence(tableau, cible):

    ...

# tests

assert derniere\_occurrence([5, 3], 1) == 2

assert derniere\_occurrence([2, 4], 2) == 0

assert derniere\_occurrence([2, 3, 5, 2, 4], 2) == 3

*Réponse*

def derniere\_occurrence(tableau, cible):

    do = len(tableau)

    for i in range(len(tableau)):

        if tableau[i] == cible:

            do = i

    return do

# Moyenne simple

Écrire une fonction moyenne prenant en paramètre un tableau non vide d'entiers et qui renvoie la moyenne des valeurs du tableau.

Dans cet exercice, on n'utilisera pas la fonction prédéfinie sum ni aucune autre fonction de quelque module que se soit calculant la moyenne.

def moyenne(valeurs):

    ...

# tests

def sont\_proches(x, y):

    return abs(x - y) < 10\*\*-6

assert sont\_proches(moyenne([10, 20, 30, 40, 60, 110]), 45.0)

assert sont\_proches(moyenne([1, 3]), 2.0)

assert sont\_proches(moyenne([44, 51, 12, 72, 65, 34]), 46.333333333333336)

*Réponse*

def moyenne(valeurs):

    somme = 0

    n = len(valeurs)

    for element in valeurs:

        somme = somme + element

    moyenne = somme/n

    return moyenne

# Recherche des positions d'un élément dans un tableau

Écrire une fonction indices qui prend en paramètres un entier element et un tableau d'entiers et qui renvoie la liste croissante des indices de element dans le tableau entiers.

Cette liste sera donc vide [] si element n'apparait pas dans entiers.

On n'utilisera ni la méthode index, ni la méthode max.

def indices(element, entiers):

    ...

# Tests

assert indices(3, [3, 2, 1, 3, 2, 1]) == [0, 3]

assert indices(4, [1, 2, 3]) == []

assert indices(1, [1, 1, 1, 1]) == [0, 1, 2, 3]

assert indices(5, [0, 0, 5]) == [2]

*Réponse*

def indices(element, entiers):

    liste = []

    for i in range(len(entiers)):

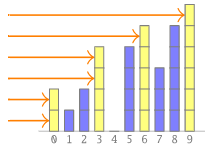
        if entiers[i] == element:

            liste.append(i)

    return liste

# Soleil couchant sur les bâtiments[⚓︎](https://e-nsi.gitlab.io/pratique/N1/120-soleil_couchant/sujet/#soleil-couchant-sur-les-batiments)

Lorsque des bâtiments sont alignés, ils se font de l'ombre les uns les autres. Dans cet exercice, nous sommes au soleil couchant, les rayons du soleil sont donc supposés horizontaux.



Le schéma ci-dessus illustre un soleil couchant qui éclaire 9 bâtiments, les rayons du soleil sont représentés par des flèches horizontales.

* Les bâtiments aux indices 0 et 3 reçoivent des rayons de soleil alors que les bâtiments aux indices 1 et 2 sont masqués.
* Les **4** bâtiments aux indices [0, 3, 6, 9] reçoivent des rayons de soleil sur au moins un étage et sont donc éclairés, alors que les autres ne le sont pas.
* Il n'y a pas de bâtiment à l'indice 4.

Écrire une fonction nb\_batiments\_eclaires qui prend en argument la liste hauteurs des bâtiments et qui renvoie le nombre de bâtiments éclairés.

* La hauteur des bâtiments (en nombre d'étages) est donnée par une liste d'entiers positifs. Une hauteur de zéro étage signifie l'absence de bâtiment.
  + Pour l'exemple ci-dessus, cette liste est [2, 1, 2, 4, 0, 4, 5, 3, 5, 6].

def nb\_batiments\_eclaires(hauteurs):

    ...

# Tests

assert 4 == nb\_batiments\_eclaires([2, 1, 2, 4, 0, 4, 5, 3, 5, 6])

assert 1 == nb\_batiments\_eclaires([0, 3, 1, 2])

*Réponse*

def nb\_batiments\_eclaires(hauteurs):

    nombre\_eclaires = 0

    hauteur\_max = 0

    for i in range(len(hauteurs)):

        if hauteurs[i] > hauteur\_max:

            nombre\_eclaires = nombre\_eclaires + 1

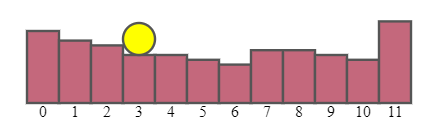
            hauteur\_max = hauteurs[i]

    return nombre\_eclaires

# Le premier minimum local

Alors qu'elle joue sur un chemin dallé, Élodie laisse rouler une balle. En observant les dalles devant elle, elle se rend compte que certaines dalles sont plus basses que les précédentes, d'autres plus hautes.

Elle se pose la question suivante : "Où va s'arrêter la balle ?"



On donne les hauteurs des dalles dans le chemin sous forme d'une liste de nombres entiers positifs. Cette liste compte au minimum deux valeurs.

On garantit que la hauteur de la dernière dalle est strictement supérieure celles de toutes les autres.

Par exemple :

Dans l'exemple précédent, illustré par la figure, la balle s'arrête sur la dalle d'indice 6. En effet, la balle s'arrête sur **la première dalle dont la hauteur est strictement inférieure à celle de la suivante**.

On signale que lorsque deux dalles consécutives sont à la même hauteur, la balle continue de rouler.

Écrire la fonction indice\_arret :

qui prend en argument la liste des hauteurs des dalles (hauteurs),

et qui renvoie l'indice de la dalle sur laquelle s'arrête la bille. La balle est initialement sur la dalle d'indice 0.

def indice\_arret(hauteurs):

    ...

# Tests

hauteurs = [3, 2, 5]

assert indice\_arret(hauteurs) == 1

hauteurs = [3, 5]

assert indice\_arret(hauteurs) == 0

hauteurs = [10, 8, 7, 5, 5, 4, 3, 6, 6, 5, 4, 12]

assert indice\_arret(hauteurs) == 6

*Réponse*

def indice\_arret(hauteurs):

    indice\_arret = 0

    for i in range(1,len(hauteurs)-1):

        if hauteurs[i+1] > hauteurs[i] and indice\_arret == 0:

            indice\_arret = i

    return indice\_arret

# Double du précédent dans un tableau

Écrire une fonction nombres\_puis\_double qui prend en paramètre un tableau de nombres entiers, et qui renvoie la liste (éventuellement vide) des couples d'entiers (a, b) qu'il peut y avoir dans le tableau tel que b suit a et b = 2 \* a.

def nombres\_puis\_double(valeurs):

    ...

# tests

assert nombres\_puis\_double([1, 4, 2, 5]) == []

assert nombres\_puis\_double([1, 3, 6, 7]) == [(3, 6)]

assert nombres\_puis\_double([7, 1, 2, 5, 3, 6]) == [(1, 2), (3, 6)]

assert nombres\_puis\_double(

    [5, 1, 2, 4, 8, -5, -10, 7]) == [(1, 2), (2, 4), (4, 8), (-5, -10)]

*Réponse*

def nombres\_puis\_double(valeurs):

    liste = []

    for i in range(len(valeurs)-1):

        if valeurs[i+1] == 2\*valeurs[i]:

            a = valeurs[i]

            b = valeurs[i+1]

            liste.append((a,b))

    return liste