Première NSI - programmation

Travaux dirigés : introduction à Python

qkzk

Pour chaque exercice vous proposerez une expression Python écrite à la main. Si vous disposez d'un poste lors de votre préparation il est utile de vérifier ce que vous avez écrit. Néanmoins, il est préférable de d'abord essayer sur feuille avant de programmer.

Exercice 0 : Lire les extraits de code

Vous en rencontrerez de deux sortes :

1. Du code python extrait d'un code source :

```
a = 3
b = 5
c = a * b
```

2. Du code python extrait de l'interpréteur :

```
>>> a = 3
>>> b = 5
>>> c = a * b
>>> c
15
>>> 2 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Les chevrons `>>>` indiquent une commande tapée et exécutée dans l'interpréteur. Une ligne qui n'en contient pas indique une réponse de l'interpréteur.

La dernière instruction est fausse, elle génère une erreur (on dit aussi qu'elle lève une exception).

Les erreurs Python :

- commencent toujours par `Traceback ...`
- indiquent le fichier et le numéro de ligne où l'erreur est survenue (`<stdin>` pour l'interpréteur),
- sont indiquées par un type d'exception (ici `ZeroDivisionError`)

Questions

1. On a effacé les lignes de sortie de l'interpréteur. Compléter les lignes vides. S'il n'y a rien à écrire, tirez un trait sur la ligne.

```
>>> x = 8

>>> x + 2

>>> x ** 2

>>> x * x

>>> x = x + 1

>>> x
```

Exercice 1 - Opérations

On donne :

```
A=10, B=5, C=2, D=4
```

1. Évaluer les expressions *valides*, rectifier celles qui sont fausses :

   Évaluer = donner la valeur

```
3 * A + 5 * B
3 A - 2 B
A / D
A // D
A += 2
A = 10
A == 12
B == A / 2
B == A // 2
C ** B
C ** -B
```

Les types de base : `int`, `float`, `bool`

Chaque objet python a un **type**. Les types de base que nous rencontrerons souvent sont : `int`, `float`, `bool`.

On accède au type d'un objet avec la fonction `type` :

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> type(2 == 2.0)
<class 'bool'>
>>> type(2) == int
True
```

La fonction `type` est à réserver aux situations extrêmes où on n'a aucune idée du type d'une variable.

On préfère `isinstance` (est une instance de...) qui permet de vérifier un type précis :

`isinstance(obj, type) -> bool`

```
>>> isinstance(2, int)
True
>>> isinstance(2 == 3, bool)
True
```

Exercice 2 : opérations sur les entiers et les flottants

- `int` : *integer*, les entiers, sans valeur maximale. Exemple : `-223`, `2**123`

Il existe 6 opérations de base sur les entiers en python :

operateur	signification
`+`	addition
`-`	soustration
`*`	produit
`//`	division entière
`%`	reste de la division
`**`	exposant

Ainsi :

- `7 // 2` vaut `3` et `7/2` vaut `3.5`.
- `7 % 2` vaut 1 (car $7 = 3 \times 2 + 1$).

1. Proposer une expression donnant le nombre de secondes écoulées entre le premier janvier de cette année à minuit et ce matin, à minuit.
2. Pierre, Paul et Jacques ont acheté par erreur `2 ** 11` gâteaux à la boulangerie. Ils les répartissent équitablement.

   Combien chacun en aura-t-il ? Combien en reste-t-il ?
3. Opération et affectation

   Il est possible d'effectuer à la fois une affectation et une opération en une seule instruction :

   ```
   >>> a = 3
   >>> a
   3
   >>> a += 1 # augmente a de 1, ne renvoie rien
   >>> a
   4
   ```

   On peut utiliser cette notation pour toutes les opérations.

   Question : Evaluer les variables `a, b, c`

   ```
   >>> a = -5
   >>> a *= 2
   >>> a

   >>> b = 11
   >>> b //= 2
   >>> b

   >>> c = 3
   >>> c -= 1
   >>> c
   ```

- `float`: *float*, les nombres à virgule flottante. Exemple : `1.234`, `2.3e4`

  Grosso modo les réels. Retenez pour l'instant qu'il n'y aucun moyen de tester une égalité parfaite avec des flottants.

```
>>> 0.1 + 0.2 == 0.3
False
```

Opérations : les mêmes que pour les entiers, la division réelle `/` en plus.

Attention : lors d'une opération entre un entier et un flottant on obtient toujours un flottant.

```
>>> 3 + 1.0
4.0
```

Question : Évaluer le type de sortie des opérations suivantes :

```
2 + 5
2.0 + 5
3 // 2
3 / 2
4 ** 0.5     # a ** 0.5 = racine carrée de a
```

Exercice 3 - Opérations sur les booléens

Il existe trois opérateurs sur les booléens :

- `not` qui retourne le contraire :

```
>>> not True
False
>>> not False
True
```

- `and` : le "et" logique : `bool_1 and bool_2` est vrai si, et seulement si `bool_1` est vrai ET `bool_2` est vrai.
- `or` : le "ou" logique : `bool_1 or bool_2` est faux si, et seulement si `bool_1` est faux ET `bool_2` est faux.

1. Evaluer les expressions booléennes suivantes :

- `not (1 == 2)`
- `(1 == 2) or (2 ** 2 == 4)`
- `(4 <= 3) or (5 > 2)`

2. `x` est un variable de type `float`. Proposer une expression booléenne permettant de vérifier que :
   1. $x \in [1; 9]$
   2. $x \in]-\infty; 0[\cup]2; 3]$
3. On veut savoir si l'entier n est divisible par trois sans qu'il ne vaille 0. Proposer une expression booléenne qui réponde à la question.

Exercice 4 - affectation

On rappelle le principe de l'affectation

```
x = 2
```

Une fois cette instruction réalisée, `x` est un identifieur qui pointe vers une case mémoire. Celle-ci contient l'entier 2.

Il ne faut pas confondre l'affectation (`=`) et la comparaison "égalité" (`==`)

- Une *comparaison* retourne toujours un *booléen* `True` ou `False`.
- Une *affectation* ne retourne rien.

Vous rencontrerez aussi les affectations multiples :

```
a, b = 2, 3
ma_liste = [1, 2, 3]
x, y, z = ma_liste
```

1. À l'issue de ces affectations que contiennent les variables `a, b, x, y` et `z` ?
2. En deux lignes : affecter à `ma_liste` la liste des entiers pairs entre 2 et 12 inclus et affecter ces entiers aux lettres de l'alphabet.
3. Selon-vous que donnerait la série d'instruction suivante ?

   ```
   ma_liste = [1, 2, 3, 4]
   x, y, z = ma_liste
   ```

4. **Affectations impossibles**. Parmi les affectations suivantes, lesquelles vont générer une erreur ? Lorsque l'affectation est possible, quel est le type de la variable ?

   ```
   a = "22"
   b = 22
   c = a + b
   "d" = 22
   "d" = "22"
   ma_liste = [1, "2", trois]
   ma_liste = ["1", 2, "trois"]
   ma_liste[2] = 9
   ma_liste[3] = 5
   e = ("b" == 22)
   f = True
   ```

Exercice 5 : bloc d'instruction

En Python un **bloc d'instruction :**

1. Commence par une ligne qui se termine par le symbole :
2. Est contenu dans un niveau d'*indentation* (2 ou 4 espaces)
3. Se termine quand le niveau d'indentation décroît.

Conditions : `if elif else`

Une **instruction conditionnelle** :

commence par `if condition:`
se poursuit par une série d'instructions précédées d'une indentation
se poursuit éventuellement par d'autres instructions condtionnelles (`elif condition:`)
se termine éventuellement par sinon... (`else:`)

Ne pas oublier le :

```
age = 23
if age > 18:
    majeur = True
else:
    majeur = False
print(majeur)
```

Le programme ci-dessus comporte une instruction conditionnelle : `if ... else`

- L'instruction `majeur = True` est *indentée* par 4 espaces. Elle n'est exécutée que si la condition `age > 18` est vraie.
- L'instruction `majeur = False` est après `else:`. Elle n'est exécutée que si `age > 18` est faux.
- L'instruction `print(majeur)` n'est pas indentée. Elle est *toujours* exécutée.

- Si d'autres conditions doivent être réalisées, on peut ajouter des instructions `elif condition:`

1. Que verra-t-on à l'écran après avoir exécuté les lignes précédentes ?
2. Écrire un programme Python qui affecte à `nb_pommes` un entier. Ensuite, si le nombre de pommes est pair, vous affichez "divisible par 2". Si ce n'est pas le cas, vous affichez "non divisible par 2".
3. Écrire un programme Python qui affiche la mention obtenue au bac.

```
moyenne = ... # obtenue plus tôt
if ...                          :
```

   On *affiche* du texte avec `print("bonjour")`

   Attention, `"bonjour"` n'est pas une *variable* mais une chaîne de caractères.

4. Rectifier les erreurs d'indentation dans les instructions suivantes :

```
if 1 + 1 == 2:
a = True
else:
    c = 2
        d = 4
            e = 6
    f = 8
```

5. Proposez deux programmes différents qui répondent au problème suivant : Martin peut inviter ses copains s'il a fini ses devoirs et rangé sa chambre.

   On utilisera les variables booléennes `devoir_faits` et `chambre_rangee`

Exercice 6. Boucles non bornées : `while`

Python propose deux boucles : `while` et `for`.

La syntaxe d'un `while` est simple :

```
while condition:
    instruction
```

Tant que `condition` est vraie, on exécute `instruction`

Somme des entiers de 1 à 10 :

```
somme = 0
entier = 1
while entier <= 10:
    somme += entier
    entier += 1
```

Afficher son nom toutes les secondes :

```
from time import sleep
while True:
    print("Robert")
    sleep(1)
```

1. Robert ajoute 50 € à sa cagnotte tous les mois jusqu'à atteindre 1200 €.
   Écrire une boucle `while` Python qui calcule le nombre de mois nécessaires pour qu'il obtienne assez d'argent.
2. Le haricot magique de Jack double de hauteur tous les jours. Il mesure 1 cm le premier jour.
   Écrire une boucle while qui calcule le nombre de jours nécessaires pour qu'il atteigne 1 km de hauteur.
3. Le programme suivant est supposé afficher un point `.` toutes les secondes et s'arrêter après 10 secondes. Malheureusement il entre dans une boucle infinie. Rectifiez le.

   ```
   from time import sleep
   nb_points = 0
   while nb_points < 10:
       print('.')
       sleep(1)        # attend une seconde
   ```

Exercice 7. Boucles bornées : `for`

Une boucle `for` itère sur une *collection* la syntaxe est :

```
for element in collection:
    instruction
```

La variable `element` prend pour valeurs successives chaque objet de `collection`

Exemple : produit des entiers de 3 à 9 :

```
produit = 1
for entier in [3, 4, 5, 6, 7, 8, 9]:
    produit = entier * produit
```

Le même résultat sans devoir décrire toute la liste des entiers :

```
produit = 1
for entier in range(3, 10):
    produit = entier * produit
```

Attention à `range` qui prend 1, 2 ou 3 paramètres :

- `range(10)` : 0, 1, 2, 3, . . . , **9**
- `range(4, 12)` : **4**, 5, 6, . . . , **11**
- `range(1, 10, 2)` : 1, **3**, **5**, 7, 9. Le dernier paramètre est le pas

1. Écrire une boucle `for` qui calcule la somme des entiers pairs plus petits que 100
2. Écrire une boucle `for` qui calcule le produit des entiers dont le reste est 2 dans la division par 3 et qui sont inférieurs à 200.
3. Le programme suivant affiche la table de multiplication par 5.

```
>>> for x in range(3):
...     print('5 *', x, '=', 5 * x)
...
5 * 0 = 0
5 * 1 = 5
5 * 2 = 10
```

1. Modifier le pour qu'il affiche la table complète (de `5 * 0` à `5 * 10`).
2. Modifier le pour qu'on puisse changer le nombre dont on veut la table.
3. Écrire un nouveau programme qui affiche TOUTES les tables.
   Penser à ajouter une ligne de séparation entre les tables (`print()`)

On peut *itérer* (= faire une boucle qui parcourt quelque chose) dans n'importe quelle collection en Python. Par exemple pour afficher une lettre par ligne :

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVXYZ"
for lettre in alphabet:
    print(lettre)
```

4. Affecter à `mes_enfants` les prénoms de vos 6 futurs enfants (j'insiste). Écrire un prénom par ligne.

Lorsqu'on a besoin de connaître *l'indice* d'un élément, il existe deux approches :

```
mes_animaux = ["Lulu", "Lili", "Minouche"]
for i in range(len(mes_animaux)):
    print("Mon animal numero", i + 1, "est", mes_animaux[i])
```

Qui va afficher :

Mon animal numero 1 est Lulu
Mon animal numero 2 est Lili
Mon animal numero 3 est Minouch

On a utilisé la fonction `len(collection) -> int` qui retourne la longueur d'une collection.

Autre approche :

```
mes_animaux = ["Lulu", "Lili", "Minouche"]
for numero, animal in enumerate(mes_animaux):
    print("Mon animal numero", numero + 1, "est", animal)
```

On a utilisé `enumerate` qui parcourt une collection et renvoie à chaque étape un couple avec l'indice et l'élément.

5. Écrire de deux manière la comptine :

   Ma lettre numéro 1 est A
   Ma lettre numéro 2 est B
   Ma lettre numéro 3 est C
   Ma lettre numéro 4 est D
   Ma lettre numéro 5 est E
   ...

Exercice 8 : Un premier types complexe : `list`

Un objet de type `list` Python est une collection d'objets, regroupés dans des `[]`.

```
ma_liste = [1, 2, 3]
ma_liste_vide = []
mes_enfants = ['Rambo 1', 'Rambo 2', 'Rambo 3']
```

On accède à un élément avec son indice : `ma_liste[indice]`. Attention : Python indexe les listes à partir de 0.

Comment s'appelle mon *second* fils, déjà ?

```
>>> mes_enfants[1]
'Rambo 2'
```

Ces objets sont *mutables* == modifiables.

```
>>> mes_enfants.append('Rambo 4') # on ajoute à la fin
>>> mes_enfants
['Rambo 1', 'Rambo 2', 'Rambo 3', 'Rambo 4']
```

On efface un élément avec `del ma_liste[indice]`

```
>>> del mes_enfants[0]      # il ne savait pas tirer à l'arc...
>>> mes_enfants
['Rambo 2', 'Rambo 3', 'Rambo 4']
```

On peut mesurer la longueur d'une liste avec `len(ma_liste)`

```
>>> len(["a", "b", "c"])
3
```

1. On exécute le programme suivant :

   ```
   mes_carres = []
   for i in range(100):
       if i % 2 == 0:
           mes_carres.append(i ** 2)
   ```

   1. Quels sont les premiers et derniers éléments de `mes_carres` ?
   2. Comment accéder à la longueur de la liste `mes_carres` ?
   3. Modifier le code pour déterminer les carrés des entiers divisibles par 3.
   4. Créer la liste de l'énoncé précédent sans utiliser `if`

2. Robert commence un régime. Le mardi et le vendredi, il ne se nourrit plus que de fruits.
   Écrire un programme qui affiche chaque jour de la semaine et le type d'alimentation de Robert :

   Le lundi tu peux manger ce que tu veux,
   Le mardi tu dois manger des fruits,

   On utilisera une liste pour enregistrer les jours `["lundi",...]`

Exercice 9 : Liste et boucle for

Dans cet exercice on répondra d'abord simplement à la question avant de proposer une fonction qui le fasse. On prendra garde aux types des paramètres en entrée et en sortie.

On itère sur une liste avec la syntaxe `for element in ma_liste:`

1. Nous avons prévu d'avoir encore 13 enfants. Compléter la liste de mes enfants à l'aide d'une boucle for.
2. On considère une liste d'entiers `entiers = [1, 2, 3, 4, ... , 1000]`. A l'aide d'une boucle `for` créer la liste des carrés des entiers : `[1, 4, 9, ...]`

Il arrive qu'on ait besoin de modifier un élément d'une liste.

Par exemple : remplacer tous les éléments d'indice *pair* par 0 :

```
for indice in range(len(ma_liste)):
    if indice % 2 == 0: # si l'indice est pair
        ma_liste[indice] = 0 # l'élément est maintenant 0
```

3. Le cinéma n'est plus mon art préféré depuis que j'ai découvert la chaîne youtube de Squeezie (j'ai dû ouvrir google pour taper son pseudo...)

   Modifier les noms de mes enfants pour qu'ils s'appellent `Squeezie 1`, `Squeezie 2` etc.

On peut tester si in élément est dans une liste avec `in`

```
>>> 1 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
```

3. On considère deux variables : `lettres = ['a', 'b', ..., 'z']` et `voyelles = ['a', 'e', 'i', 'o', 'u', 'y']`

   Écrire un programme python qui affiche chaque lettre de l'alphabet avec un commentaire à la manière de :

   ```
   a est une voyelle
   b n'est pas une voyelle
   c n'est pas une voyelle
   ...
   ```

4. Les albums des Beatles sortis au royaume uni sont :

Année	Nom
1963	Please Please Me
1963	With the Beatles
1964	A Hard Day's Night
1964	Beatles for Sale
1965	Help!
1965	Rubber Soul
1966	Revolver
1967	Sgt. Pepper's Lonely Hearts Club Band
1968	The Beatles
1969	Yellow Submarine
1969	Abbey Road
1970	Let It Be

   1. Écrire un programme affichant les titres des albums sortis une année paire.
      On créera deux listes : celle des années et celle des titres.

   Remarque : Il est possible d'itérer dans deux listes à la fois avec `zip` :

   ```
   fruits = ["banane", "fraise", "pastèque"]
   couleurs = ["jaune", "rouge", "verte"]
   for fruit, couleur in zip(fruits, couleurs):
       print(fruit, couleur)
   ```

   Dont l'exécution affiche :

```
banane jaune
fraise rouge
pastèque verte
```

2. Écrire le programme précédent à l'aide de la fonction `zip`
3. Retour sur les albums des Beatles.
   On décide d'enregistrer les albums dans une liste de couples :

   ```
   albums = [(1963, "Please Please me"), (1963, "With the Beatles"), ...]
   ```

   Écrire une boucle qui affiche les années et titres des albums dont le titre contient la lettre "a".

Exercice 10 - Approfondissement sur les `list`

Cette série d'exercices est tirée du cahier d'exercices Python de Tristan LEY.

Partie 1

Créer les listes Python respectant les consignes suivantes, en utilisant impérativement une boucle :

1. Une liste contenant 10 zéros,
2. Une liste contenant les nombres pairs de 0 à 50, une autre contenant les impairs de 0 à 51
3. Une liste contenant 20 nombres tirées aléatoirement entre 1 et 100 *inclus*

```
from random import randint
randint(3, 5) # un entier aléatoire entre 3 et 5 bornes incluses
```

4. Une liste contenant les 15 premiers carrés parfaits :`[1, 4, 9, ...]`
5. une liste de 20 nombres altenant entre 0 et 1 : `[0, 1, 0, 1, 0, ...]`

Rappel : si n est pair, `n % 2` vaut `0`, si n est impair, `n % 2` vaut `1` 6. une liste de 100 nombres alternant entre -1 et 1 : `[-1, 1, -1, 1]` 7. une liste contenant l'ensemble des entiers *relatifs* compris entre -20 et 20.

Partie 2

Reprendre la partie 1 en utilisant des listes par compréhension.

Partie 3

1. Écrire une fonction minimum() qui prend en paramètre une liste PYTHON de nombres et renvoie la valeur minimale de ces nombres.
2. Écrire une fonction `position_minimum()` qui prend en paramètre une liste PYTHON de nombres et renvoie la position de la valeur minimale de ces nombres.
3. Faire de ces deux fonctions une seule, nommée simplement `minimum()`, qui renvoie un tuple dont le premier élément sera la valeur du minimum quand le second donnera sa position dans la liste.

Partie 4

1. Écrire une fonction moyenne() qui prend en paramètre une liste PYTHON de nombres et renvoie la moyenne de ces nombres.

AIDE : on pourra mettre en place une variable nommée moy qui agira comme un accumulateur (au fur et à mesure qu'on parcourra les éléments de la liste, moy se verra ajouter chaque nouvel élément).

```
assert moyenne([3, 5, -9, 2, 4]) == 1, "/!\ moyenne()"
```

Partie 5

1. Écrire une fonction `contient_valeur()` qui prend en paramètres une liste PYTHON et une valeur, et qui renvoie le booléen `True` si la valeur figure dans la liste, `False` sinon.
2. Écrire une fonction `position_valeur()` qui prend en paramètres une liste PYTHON et une valeur, et qui renvoie la position de la valeur dans la liste si elle y figure bien, None sinon.

```
assert contient_valeur([1, 2, 3], 2)
assert not contient_valeur([1, 2, 3], 4)
```

Partie 6

Écrire une fonction `occurrences()` donnant le nombre de fois où un élément figure dans une liste PYTHON.

Les paramètres de cette fonction seront, dans l'ordre, une liste PYTHON, suivie d'une valeur. La valeur de retour sera un entier naturel (nul si l'élément n'apparaît pas dans la liste PYTHON).

```
assert occurences([1, 1, 2, 1, 3], 1) == 3
assert occurences([1, 1, 2, 1, 3], 2) == 1
assert occurences([1, 1, 2, 1, 3], 4) == 0
```

Partie 7

1. Écrire une fonction mediane() calculant la médiane d'une liste PYTHON de valeurs numériques triées par ordre croissant.
2. Adapter la fonction pour qu'elle répondre juste si la liste n'est pas triée.

```
assert mediane([3, 1, 7, 5, 9]) == 5
assert mediane([3, 1, 7, 5]) == 4.0
```

Partie 8

```
1. Créer une fonction `entremele` qui prend deux listes de même longueur en paramètres et renvoie la liste

```python
entremele([1, 2], ["a", "b"])
[1, "a", 2, "b"]
```

Partie 9

Un décalage circulaire est une opération sur une liste PYTHON qui consiste :

- à décaler tous les éléments de la liste d'un cran vers la droite (le dernier venant alors en 1re place);
- ou à décaler tous les éléments de la liste d'un cran vers la gauche (le 1er venant alors en dernière place).

1. Écrire une fonction `decale_droite()` qui prend en argument une liste et renvoie une nouvelle liste contenant les mêmes éléments mais à des positions décalées d'une une placer vers la droite. VALIDATION :
2. Écrire une fonction `decale_gauche()` qui prend en argument une liste et renvoie une nouvelle liste contenant les mêmes éléments mais à des positions décalées d'une place vers la gauche.

VALIDATION :

```
assert decale_droite([1, 2, 3, 4]) == [4, 1, 2, 3], "/!\ decale_droite()"
assert decale_gauche([1, 2, 3, 4]) == [2, 3, 4, 1], "/!\ decale_gauche()"
```
```

Partie 10

Écrire une fonction `valeurs_sup()` qui retourne les éléments d'une liste qui sont supérieurs ou égaux à une valeur donnée.

Cette fonction prendra deux paramètres : une liste PYTHON et une valeur. Elle renverra une liste PYTHON, éventuellement vide. VALIDATION :

```
assert valeurs_sup([1, 2, 3], 1.5) == [2, 3], "/!\ valeurs_sup()"
assert valeurs_sup([1, 2, 3], 4) == [], "/!\ valeurs_sup()"
```

Partie 11

1. Écrire une fonction `permute()` qui prend en argument une liste PYTHON et deux indices `i` et `j`, et renvoie une liste dans laquelle les éléments d'indice `i` et `j` de la liste initiale ont été échangés.
2. Écrire une fonction `renverse()` qui prend en argument une liste PYTHON et renvoie une liste constituée des éléments rangés dans l'ordre inverse.

```
assert permute([4, 0, 2, -1], 0, 3) == [-1, 0, 2, 4], "/!\ permute()"
assert renverse([4, 0, 2, -1]) == [-1, 2, 0, 4], "/!\ renverse()"
```

Partie 12

1. Écrire une fonction `est_croissante()` qui prend en argument une liste PYTHON et renvoie le booléen `True` si cette liste est composée d'éléments rangés par ordre croissant, `False` sinon.
2. Écrire une fonction `est_decroissante()` qui prend en argument une liste PYTHON et renvoie le booléen `True` si cette liste est composée d'éléments rangés par ordre décroissant, `False` sinon.

AIDE : on peut quitter une boucle à l'aide de l'instruction break. D'aucuns considèrent néanmoins cette approche comme inélégante...

```
assert est_croissante([1, 2, 3]), "/!\ est_croissante()"
assert not est_croissante([1, 3, 2]), "/!\ est_croissante()"
assert est_decroissante([3, 2, 1]), "/!\ est_decroissante()"
```

Partie 13

Écrire une fonction `hamming()` qui prend en paramètres deux listes PYTHON de même longueur et renvoie le nombre d'indices en lesquels les deux listes diffèrent (cette valeur est appelée " distance de HAMMING entre les deux listes ").

REMARQUE : la distance de HAMMING, pour simple qu'elle puisse paraître, est une notion tout sauf anecdotique! On l'utilise en informatique, en traitement du signal et dans les télécommunications (elle joue en effet un rôle important en théorie algébrique des codes correcteurs d'erreurs).

```
assert hamming([1, 2, 3, 2, 1], [1, 5, 3, 4, 0]) == 3, "/!\ hamming()"
```

Partie 14 - Vecteurs

Dans le plan un vecteur peut être numériquement représenté par un couple de nombres : ses coordonnées. En sciences on rencontre des vecteurs ayant plus de deux coordonnées. Ils sont représentés de la même manière : par l'ensemble de leurs coordonnées.

Peu importe la valeur n de la dimension, certaines opérations restent similaires :

- l'addition de deux vecteurs se fait coordonnée par coordonnée,
- le produit d'un vecteur par un nombre se fait coordonnée par coordonnée.

On peut étendre ces règles au produit scalaire que vous étudierez cette année en mathématiques.

1. Écrire une fonction `somme_vecteurs` qui prend en paramètres deux listes python de même taille et renvoie leur somme.

```
>>> somme_vecteurs([1, 2, 3], [7, 8, 9])
[8, 10, 11]
```

   Bonus : on pourra renvoyer une erreur `TypeError` avec un message personnalisé lorsque les deux listes n'ont pas la même dimension.

2. Écrire une fonction `produit_vecteur_reel` qui prend en arguments une liste python et un nombre et renvoie le produit de ce vecteur par ce nombre.

```
>>> produit_vecteur_reel([1, 2, 3, 4], 10)
[10, 20, 30, 40]
```

Attention : ces deux fonctions doivent renvoyer le bon résultat quelle que soit les dimensions des listes.

Partie 15

Écrire une fonction `plus_longue_suite_croissante` qui prend en paramètre une liste de nombres et renvoie *la plus longue suite croissante de nombre consécutifs* qu'elle contient. S'il existe plusieurs sous suites croissantes de longueur maximale, la première sera renvoyée.

```
>>> plus_longue_suite_croissante([1, 5, 2, 3, 4, 0, 6])
[2, 3, 4]
>>> plus_longue_suite_croissante([1, 5, 7, 2, 3, 4, 0])
[1, 5, 7]
>>> plus_longue_suite_croissante([5, 4, 3, 2, 1])
[]
```

Exercice 10 - Dictionnaires

On considère le script suivant :

```
dict_eleve = {
    'nom': 'Figny',
    'prénom': 'Jean',
    'age': 16,
    }
```

1. Comment accéder au nom de l'élève ? À son prénom ?
2. Comment obtenir le nombre d'éléments de ce dictionnaire ?
3. Ajouter la moyenne de Jean, qui s'élève à 12.34.
4. On vient de célébrer l'anniversaire de Jean qui a maintenant 17 ans. Changer son age.
5. Jean vient de quitter l'établissement (renvoyé parce qu'il écoute du JuL). Supprimer sa moyenne du dictionnaire.

Exercice 11 - Depuis un dictionnaire vide.

1. Comment créer un dictionnaire vide ? Proposer deux syntaxes différentes.
2. Comment s'assurer qu'un objet enregistré dans une variable `d` est du type dictionnaire ? Proposer deux réponses différentes. Laquelle privilégier ?
3. Créer le dictionnaire `utilisateur` avec les couples clés, valeurs suivants :

clé	valeur
nom	Josephe
prenom	Apolline

clé	valeur
age	22
password	juygvfesw

4. Écrire une fonction qui reçoit une liste de couples clés, valeurs et retourne le dictionnaire correspondant :

   En entrée on reçoit une liste comme ceci :

```
entree = [('nom', 'Josephe'), ('prenom', 'Apolline'), ('age', 22),
          ('password', 'juygvfesw')]
```

   En sortie on veut :

```
sortie = {'nom': 'Josephe',
          'prenom': 'Apolline',
          'age': 22,
          'password': 'juygvfesw'}
```

   Écrire toutes les étapes à la main (création, ajout avec une boucle etc.)

   Remarque : La fonction `dict` transforme ce type de liste, contenant des couples, en un dictionnaire ayant exactement le format souhaité !

```
>>> dict( [(1: 2), (3: 4)] )
{1: 2, 3: 4}
```

Exercice 12 - Itérer sur un dictionnaire.

1. Quelles sont les trois manières d'itérer sur un dictionnaire en Python ?
2. Un site de jeux vidéos enregistre les scores de ses joueurs au jeu Pacman dans un dictionnaire :

```
scores_pacman = {'paul': 34,
                 'honorine': 456,
                 'marcel': 89,
                 'octave': 542,
                 'marine': 12,
                 'mélanie': 134,
                 'patricia': 631}
```

3. Ajouter le score d'Amandine qui a 542 points. Attention à respecter la convention : tous les prénoms sont en minuscule.
4. Créer une fonction qui prend un dictionnaire tel que le précédent et le nom d'un joueur comme `'Amandine'` et retourne son score si le joueur est inscrit ou 0 sinon. À nouveau, attention à la convention d' enregistrement des noms.
5. Créer une fonction permettant d'inscrire un joueur. Elle respecte la signature ci-dessous.

```
incrire_joueur(score_jeu: dict, joueur: str) -> bool
```

   Elle retourne `True` si le joueur n'est pas déjà inscrit, `False` s'il est déjà inscrit. Un nouvel inscrit a un score de 0.

6. Depuis le dictionnaire précédent, créer à la liste des noms de joueurs. Proposer une fonction qui le fasse.
7. Depuis le dictionnaire précédent, calculer le score moyen des inscrits :
   a. En utilisant la fonction `sum`
   b. Sans utiliser la fonction `sum`

8. Créer une fonction qui prend le dictionnaire de joueurs en paramètre et retourne une chaîne de caractères contenant une série de phrases telles que celle ci-dessous, séparées par des retours à la ligne :
`Le score de Paul est 34`

   À nouveau, prenez garde à la façon dont est noté le prénom.

   On pourra utiliser la méthode `upper` des chaînes de caractères :

```
>>> 'aBcD'.upper()
'ABCD'
>>> help(str.upper)
Help on method_descriptor:

upper(self, /)
    Return a copy of the string converted to uppercase.
```

9. La fonction précédente ne convient plus. On souhaite maintenant qu'elle retourne *une liste* de phrases (toujours une phrase par joueur, similaire à la précédente). Adapter votre fonction précédente.

10. Adapter la fonction précédente pour qu'elle retourne la liste des phrases :

    a. Triées par ordre alphabétique du prénom,
    b. Triées par score croissant.

    Trier une liste peut se faire avec la fonction `sorted` ou la méthode `sort` :

```
>>> ma_liste = [3, 2, 1]
>>> sorted(ma_liste)        # retourne une copie triée
[1, 2, 3]
>>> ma_liste
[3, 2, 1]
>>> ma_liste.sort()         # tri en place, ne retourne rien.
>>> ma_liste
[1, 2, 3]
```