Ces exercices visent à développer de l'aisance dans l'écriture de code **Python**. ***Certains sont adaptés à la spécialité mathématiques, quand d'autres en excèdent le niveau d'exigence :*** *la mention « NSI » les identifiera. La mention « Tale » pourra également venir préciser le niveau principalement concerné.*

Dans de nombreux exercices, une ou plusieurs « assertions[1] » sont indiquées : *en copiant-collant ces instructions après la définition de vos fonctions (sautez une ligne pour plus de lisibilité), vous pourrez vérifier que votre code fonctionne comme attendu sur les exemples qu'elles proposent.* ***Attention :*** *il est souvent* ***impossible*** *de tester l'égalité entre quantités numériques : lorsque la situation impliquera des calculs,* ***vous devrez donc chercher à proposer de meilleures assertions***[2] *(moyenne, variance, etc.).*

« Bonus permanent » en NSI : *toujours chercher à déterminer les préconditions que devraient vérifier les paramètres reçus par les fonctions que vous écrirez !*

RAPPELS

- Une liste Python est une **collection** d'objets *numérotés*. On accède à un élément de la collection *via* son **index** (ou indice), **compté à partir de 0**. Ainsi, pour `liste = [10, 20, 30]`, l'instruction `liste[0]` renvoie 10, quand `liste[2]` donne 30. Python calcule la **longueur de la liste** avec l'instruction `len(liste)` (ici c'est 3).
- **Une liste Python est un objet *modifiable*** (on dit aussi *mutable*) : si `liste = [10, 20, 30]`, après l'instruction `liste[1]=0`, la variable `liste` désignera `[10, 0, 30]`.
- **La création d'une liste Python se fait souvent par accumulation :** on part d'une liste vide `liste = []`, et on lui ajoute des éléments, le plus souvent « par la droite » (ou « par la fin »). Il y a trois possibilités :
  - la « méthode » `append()` : après l'instruction `liste.append(2)`, la variable `liste` désignera `[2]` ;
  - la fusion de deux listes : après l'instruction `liste = liste + [4]`, la variable `liste` désignera `[2, 4]` (on peut donc ajouter « par la gauche » des éléments à une liste avec cette technique) ;
  - l'extension par une autre liste : après `liste.extend([6, 8])`, la variable `liste` désigne `[2, 4, 6, 8]`.
- **Python supporte la création de listes *« en compréhension »* :** on peut ainsi écrire une instruction telle que `liste = [ f(i) for i in <ensemble> if test(i) ]`, qui se traduit par *« Python, fabrique la liste des expressions `f(i)`, résultats des transformations de `i` par la fonction `f`, pour `i` parcourant l'objet Python `<ensemble>`, seulement si `i` vérifie la condition `test(i)` »*. Il faut naturellement que les fonctions `f` et `test` ainsi que l'objet `<ensemble>` soient définis au préalable, `test` devant renvoyer un **booléen** (`True` ou `False`).
- **Il y a deux approches pour réaliser un parcours de liste Python :**
  - on commence par énumérer l'ensemble des *indices* (les positions des éléments dans la liste Python) avec une boucle telle que `for i in range(len(liste)):`, puis on obtient la *valeur* mémorisée à la position `i` avec l'instruction `valeur = liste[i]` ;
  - on peut choisir de parcourir directement l'ensemble des *éléments* de la liste Python avec une instruction du type `for valeur in liste:`.

EXERCICE 1 ★☆☆ — *Les bases*

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 nombres impairs de 1 à 51 ;
3. une liste contenant 20 nombres tirés aléatoirement entre 1 et 100 *inclus* (pour cela, vous importerez la fonction `randrange()` de la bibliothèque `random` : `from random import randrange`) ;
4. une liste contenant les 15 premiers carrés parfaits (`[1, 4, 9, 16, 25, ...]`) ;
5. une liste de 20 nombres alternant entre 0 et 1 (`[0, 1, 0, 1, 0, ...]`). Vous utiliserez pour cela la parité de l'indice de la boucle (la variable `i` de l'instruction `for i in range(...)`) ;
6. une liste de 100 nombres alternant entre −1 et 1 (`[1, -1, 1, -1, 1, ...]`). Vous pourrez utiliser *astucieusement* l'indice de la boucle... mais il existe de nombreuses autres approches !
7. une liste contenant l'ensemble des entiers *relatifs* compris entre −20 et 20, **obligatoirement** à l'aide de la boucle `for i in range(1,21)`, et en **n'**utilisant **pas que** la méthode `append()` !

1. Une assertion permet de vérifier la conformité d'une situation : si la vérification échoue, une erreur survient (on dit qu'une *exception est levée*). Par exemple, il est prudent de s'assurer qu'une quantité est non-nulle avant de calculer son inverse...

2. Il suffit de vérifier que la valeur absolue de l'écart entre le résultat théorique et le résultat approché est faible : par exemple `assert abs(ma_fonction(parametre) - valeur_theorique) < 10**-6, "Oups"`, à adapter selon les circonstances.

EXERCICE 2 ★☆☆ — *Pour développer la compréhension*

Reprendre l'exercice 1 en mettant en œuvre la technique des compréhensions de listes.

EXERCICE 3 ★★☆ — *Minimum et localisation*

1. Écrire une fonction `minimum()` qui prend en paramètre une liste Python de nombres et renvoie la valeur minimale de ces nombres. Vous n'utiliserez pas la fonction native de Python !

   Validation :
   ```
   assert minimum([3, 5, -9, -9, 4]) == -9, "/!\ minimum()"
   ```

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.

   Validation :
   ```
   assert position_minimum([3, 5, -9, -9, 4]) == 2,
   ^"/!\ position_minimum()"
   ```

3. Écrire une fonction `positions_minimum()` qui prend en paramètre une liste Python de nombres et renvoie une liste Python contenant **les *positions*** de la valeur minimale de ces nombres (la liste pourra ne contenir qu'un seul élément si le minimum n'apparaît qu'une fois).

   Validation :
   ```
   assert positions_minimum([3, 5, -9,-9, 5]) == [2, 3],
   "/!\ positions_minimum()"
   ```

Aide : on pourra mettre en place une variable nommée `mini` qui contiendra initialement la 1re valeur de la liste (l'initialiser à `0` serait une *très* mauvaise idée : pourquoi ?) et qui sera comparée et modifiée avec chaque autre valeur de la liste Python reçue en paramètre.

Bonus NSI : *faire de ces 3 fonctions une seule, nommée simplement `minimum()`, qui renvoie un tuple dont le 1er élément est la valeur du minimum, le 2nd étant un tuple donnant sa (ses) position(s) dans la liste.*

Remarque : *Python intègre nativement une fonction minimum. Ainsi `min([5, -2, 3])` renverra `-2`.*

EXERCICE 4 ★★☆ — *Maximum et localisation*

1. Écrire une fonction `maximum()` qui prend en paramètre une liste Python de nombres et renvoie la valeur maximale de ces nombres. Vous n'utiliserez pas la fonction native de Python !

   Validation :
   ```
   assert maximum([3, 5, -9, 2, 5]) == 5, "/!\ maximum()"
   ```

2. Écrire une fonction `position_maximum()` qui prend en paramètre une liste Python de nombres et renvoie *position* de la valeur maximale de ces nombres.

   Validation :
   ```
   assert position_maximum([3, 5, -9, 2, 5]) == 2,
   "/!\ position_maximum()"
   ```

3. Écrire une fonction `positions_maximum()` qui prend en paramètre une liste Python de nombres et renvoie une liste Python contenant **les *positions*** de la valeur maximale de ces nombres (la liste pourra ne contenir qu'un seul élément si le maximum n'apparaît qu'une fois).

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

Aide : on pourra mettre en place une variable nommée `maxi` qui contiendra initialement la 1re valeur de la liste (l'initialiser à `0` serait une *très* mauvaise idée : pourquoi ?) et qui sera comparée et modifiée avec chaque autre valeur de la liste Python reçue en paramètre.

Bonus NSI : *faire de ces 3 fonctions une seule, nommée simplement `maximum()`, qui renvoie un tuple dont le 1er élément sera la valeur du minimum, le 2nd étant un tuple donnant sa (ses) position(s) dans la liste.*

Remarque : *Python intègre nativement une fonction maximum. Ainsi `max([5, -2, 3])` renverra `5`.*

EXERCICE 5 ★★☆ — *Moyenne arithmétique*

Écrire une fonction `moyenne()` qui prend en paramètre une liste Python de nombres et renvoie la moyenne arithmétique 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).

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

Remarque : *l'écriture `moy = moy + val` est parfaitement légitime en informatique, même si son pendant mathématique correspond à une équation dépourvue de solution (sauf si `val` vaut 0, bien sûr!). Elle se comprend mieux lorsqu'on la traduit en français, ce qui donne « `moy` prend la valeur `moy + val` ». Dit autrement, la **nouvelle** valeur de la variable `moy` est formée à partir de l'ancienne valeur de `moy`, récupérée par le processeur de l'ordinateur **avant** que le calcul ait lieu, et qu'il peut alors augmenter de la valeur désignée par la variable `val`.*

EXERCICE 6 ★★☆ — *Moyenne arithmétique **pondérée***

Écrire une fonction `moy_pond()` qui prend en paramètre **deux** listes Python de nombres et renvoie la moyenne pondérée des nombres de la première liste pondérée par les nombres de la seconde liste.

Validation :
```
assert moy_pond([10, 16, 5], [1, 0.5, 2]) == 8.0, "/!\ moy_pond()"
```

EXERCICE 7 ★★☆ — *Moyenne géométrique*

Écrire une fonction `moyenne_geom()` qui prend en paramètre une liste Python de nombres et renvoie la **moyenne géométrique** de ces nombres. Par définition, la moyenne géométrique de n nombres, notés x_1, x_2, …, x_n, est $\sqrt[n]{x_1 \times x_2 \times \cdots \times x_n} = (x_1 \times x_2 \times \cdots \times x_n)^{1/n}$.

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` sera multiplié par chaque nouvel élément).

Validation :
```
assert moyenne_geom([2, 4, 8]) == 4, "/!\ moyenne_geom()"
```

Remarques :

- *pour la notion d'*accumulateur*, se reporter à la remarque de l'exercice précédent ;*
- *la notation $\sqrt[n]{x}$ se lit « racine n-ième de x », ce qui équivaut à $x^{1/n}$ ou « x à la puissance 1 sur n » ;*
- *en mathématiques, une suite dite « géométrique » porte ce nom car pour trois termes consécutifs de la suite u_{k-1}, u_k et u_{k+1}, le terme « du milieu » u_k est la **moyenne géométrique** des termes qui « l'entourent » u_{k-1} et u_{k+1}.*

EXERCICE 8 ★★☆ — *Appartenance et localisation*

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.

   Validation :
   ```
   assert contient_valeur([3, 5, 2], 5) == True, "/!\ contient_valeur()"
   assert contient_valeur([3, 5, 2], 0) == False, "/!\ contient_valeur()"
   ```

2. Écrire une fonction `position_valeur()` qui prend en paramètres une liste Python et une valeur, et qui renvoie la ***première** position* de la valeur dans la liste si elle y figure bien, `None` sinon.

   Validation :
   ```
   assert position_valeur([5, 3, 5], 5) == 0, "/!\ position_valeur()"
   assert position_valeur([5, 3, 5], 0) == None, "/!\ position_valeur()"
   ```

3. Écrire une fonction `positions_valeur()` qui prend en paramètres une liste Python et une valeur, et qui renvoie une **liste** Python des positions auxquelles la valeur apparaît dans la liste passée en 1er paramètre (si la valeur en est absente, une liste vide sera retournée).

Validation :
```
assert positions_valeur([3, 5, 5], 5) == [1, 2],
                "/!\ positions_valeur()"
assert positions_valeur([3, 5, 5], 0) == [], "/!\ positions_valeur()"
```

Remarque : *Python sait nativement tester l'appartenance à une liste. Ainsi `3 in [5, -2, 3]` donne `True`. Il peut aussi nativement trouver la position d'un élément : si `liste = [5, -2, 3]` alors `liste.index(3)` renvoie 2. Par contre, une erreur surviendra si on cherche la position d'un élément n'appartenant pas à une liste. Enfin, Python ne sait pas donner les différentes positions auxquelles une valeur apparaît (éventuellement) dans une liste.*

EXERCICE 9 ★☆☆ — *Compter les occurrences*

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).

Validation :
```
assert occurrences([3, 5, -9, 5, 4], 5) == 2, "/!\ occurrences()"
assert occurrences([3, 5, -9, 5, 4], 1) == 0, "/!\ occurrences()"
```

Remarque : *Python sait compter les occurrences d'un élément dans une liste. `[5, 3, 5].count(5)` donne 2.*

EXERCICE 10 ★★☆ — *Entremêler... mais pas les pinceaux!*

1. Écrire une fonction `entremele()` qui prend en argument deux listes Python de même longueur, et renvoie une liste contenant alternativement un élément de la 1re liste suivi d'un élément de la 2nde.

   Validation :
   ```
   assert entremele([1, 2, 3], [5, 6, 7]) == [1, 5, 2, 6, 3, 7],
                   "/!\ entremele()"
   ```

2. Écrire une fonction `entremele2()` qui prend en argument deux listes Python **de longueurs potentiellement différentes** : si une liste est plus longue que l'autre, les valeurs surnuméraires de la plus longue liste sont juste ajoutées à la fin de la liste résultant.

   Validation :
   ```
   assert entremele2([1, 2], [5, 6, 7, 8]) == [1, 5, 2, 6, 7, 8],
                   "/!\ entremele2()"
   ```

EXERCICE 11 ★★☆ — *Tourner en rond... mais avec style!*

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

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_gauche([1, 2, 3]) == [2, 3, 1], "/!\ decale_gauche()"
   ```

Remarque : *le décalage circulaire portant sur des bits est souvent utilisé en cryptographie !*

EXERCICE 12 ★★½ — *Tourner en rond... avec **plus** de style!*

Reprendre l'exercice précédent en ajoutant un paramètre entier **relatif** à la fonction de décalage. Ce paramètre gouvernera le « nombre de crans des décalages » (vers la droite ou vers la gauche, selon son signe).

EXERCICE 13 ★★☆ — *Du binaire au décimal et réciproquement*

RAPPELS

Nous comptons en base 10, avec une numération dite « de position ». Cela signifie que :

- nous utilisons 10 symboles différents : les chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 ;
- lorsque nous avons épuisé ces symboles, nous les *réutilisons* en les *décalant* d'une place vers la gauche : le « un-zéro » (10) qui suit 9 signifie « *lorsqu'on ajoute 1 unité à un lot qui en comporte déjà 9, nous obtenons un paquet qu'on appelle une dizaine* ». L'écriture 10 signifie donc qu'on a 1 dizaine et 0 unité ;
- on peut continuer à ajouter des unités, jusqu'à avoir 1 dizaine et 9 unités, ce que l'on note donc « 19 ». Avec une unité supplémentaire, on forme 2 paquets de dix, soit 2 dizaines, et 0 unité, ce que l'on note « 20 » ;
- on continue ainsi jusqu'à atteindre 9 dizaines et 9 unités : on a alors utilisé toutes les combinaisons possibles de 10 chiffres à placer sur 2 emplacements, et l'ajout d'une unité impose un nouveau regroupement. On introduit alors la position des centaines, à gauche de celle des dizaines, et on écrit « 100 » pour compter 1 centaine, 0 dizaine et 0 unité. Et ainsi de suite...

Un ordinateur compte en binaire, c'est-à-dire en base 2. Cela signifie que :

- il n'utilise que deux états (le courant circule / le courant ne circule pas), que les humains représentent respectivement et par convention avec les chiffres 1 et 0 ;
- pour pouvoir compter au-delà de 1, on doit aussi effectuer des regroupements : au lieu de se faire par puissances de 10, ils se feront par puissances de deux. Le premier pourrait être appelé une « deuzaine » : si j'ajoute 1 unité à 1 unité, j'ai 1 « deuzaine » et 0 unité, ce que l'on note « 10 ». On a donc 10 en binaire qui correspond à 2 en décimal, ce que l'on écrit parfois $10_b = 2_d$. On aura ensuite 11_b qui vaudra donc 1 « deuzaine » et 1 unité, soit 3 en décimal : $11_b = 3_d$. Si l'on ajoute 1 unité à 11_b, on formera 1 « quatraine » (un paquet de $2^2 = 4$ en décimal), et l'on écrira 100_b pour indiquer qu'on a 1 « quatraine », 0 « deuzaine » et 0 unité : on aura donc $100_b = 4_d$. Et ainsi de suite...

L'écriture binaire d'un nombre peut être représentée sous la forme d'une liste Python ne contenant que des 1 et des 0. Ainsi, $123_d = 1111011_b$, que l'on représentera par la liste Python `[1, 1, 1, 1, 0, 1, 1]`.

1. Écrire une fonction `bin2dec()` qui prend en argument une liste Python représentant un nombre binaire et retourne la valeur décimale correspondante.

   Exemple : `bin2dec([1, 0, 0, 1, 1])` calculera donc $2^4 + 2^1 + 2^0$ et renverra 19.

   Aide : la difficulté consiste à calculer la puissance de 2 correspondant à la « bonne » position dans la liste ! Or la liste se lit de la gauche (position 0) vers la droite (position $n-1$, pour une liste de longueur n), tandis que la puissance associée à la position la plus à gauche est 2^{n-1}. Bref : pour une position `k` dans la liste, la puissance de 2 associée a pour exposant `lent(liste)-1-k`.

   Validation :
   ```
   assert bin2dec([1, 0, 0, 1, 1]) == 19, "/!\ bin2dec()"
   ```
2. Écrire la fonction réciproque `dec2bin()` qui prend en argument un nombre entier en écriture décimale et retourne la liste Python correspondant à l'écriture binaire de ce nombre.

   Validation :
   ```
   assert dec2bin(123) == [1, 1, 1, 1, 0, 1, 1], "/!\ dec2bin()"
   ```

Remarque : *deux choses à signaler, pour la culture...*

- *Un chiffre binaire se nomme aussi un bit, terme anglais formé de la contraction de « **bi**nary digi**t** », et qui signifie également « morceau » en français. Le bit le plus à gauche est appelé « bit de poids fort », car il correspond à la puissance de 2 la plus grande, celle qui « pèse » le plus dans la « constitution » du nombre. Le bit le plus à droite est appelé « bit de poids faible », car il correspond à la puissance de 2 la plus faible, celle qui « pèse » le moins dans la « constitution » du nombre.*
- *Le monde se divise en 10 catégories : ceux qui connaissent le binaire, et ceux qui creusent. Creuserez-vous ☺ ?*

EXERCICE 14 ★★½ — *Médiane*

Écrire une fonction `mediane()` calculant la médiane d'une liste Python de valeurs numériques. Le mode de calcul de la médiane d'une série statistique peut notamment être retrouvé sur Wikipedia, exemples à l'appui : https://fr.wikipedia.org/wiki/Médiane_(statistiques)#Mode_de_calcul.

Aide : on doit disposer d'un tableau de valeurs ordonnées, dans l'ordre croissant par exemple. Pour fabriquer un tel tableau *intermédiaire*, utilisez l'instruction `tableau_croissant = sorted(tableau)`. Notez qu'en NSI, découvrir et comprendre des algorithmes qui permettent d'accomplir ce tri est un objectif.

Validation :
```
assert mediane([3, 5, -9, 5, 4]) == 5, "/!\ mediane()"
assert mediane([1, 2, 3, 4]) == 2.5, "/!\ mediane()"
```

EXERCICE 15 ★★☆ — *Variance*

Écrire une fonction `variance()` calculant la variance d'une liste Python de valeurs numériques. On rappelle la formule de la variance d'une série statistique $x_1, x_2, \dots, x_n$: $V = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$ (c'est la moyenne arithmétique des carrés des écarts entre chaque valeur x_i de la série et la moyenne $\overline{x}$ de la série).

Validation :
```
assert variance([1, 2, 3, 4, 5]) == 2, "/!\ variance()"
```

Remarque : *le calcul de la variance d'une série de valeurs pose plus de problèmes qu'on ne l'imagine de prime abord. Les personnes curieuses pourront consulter, sur Wikipedia, la section consacrée au calcul itératif de la variance, puis l'article de la version anglaise intitulé « algorithms for calculating variance ».*

EXERCICE 16 ★☆☆ — *Écart-type*

Écrire une fonction `ecart_type()` calculant l'écart-type d'une liste Python de valeurs numériques. On rappelle que l'écart-type d'une série statistique est la racine carrée de la variance de cette série.

Validation :
```
assert ecart_type([1, 2, 3, 4, 5]) == 2**.5, "/!\ ecart_type()"
```

EXERCICE 17 ★★☆ — *Là-haut*

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()"
```

EXERCICE 18 ★★☆ — *Au-dessus de la moyenne*

Écrire une fonction `valeurs_sup_moy()` qui retourne les éléments d'une liste qui sont supérieurs ou égaux à la moyenne de la liste. Cette fonction prendra un seul paramètre : une liste Python. Elle renverra une liste Python.

Validation :
```
assert valeurs_sup_moy([1, 2, 3]) == [2, 3], "/!\ valeurs_sup_moy()"
```

EXERCICE 19 ★★☆ — *Retournements de situation*

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.

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

2. Écrire une fonction `renverse()` qui prend en argument une liste Python et renvoie une liste constituée des éléments rangés dans l'ordre inverse.

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

EXERCICE 20 ★★☆ — *Les vecteurs, dans le plan, l'espace et au-delà!*

Dans le plan, un vecteur (objet mathématique modélisant un glissement sans rotation, retournement ni déformation) peut être numériquement représenté par un couple de valeurs numériques : ses coordonnées. Dans l'espace, il faudra 3 coordonnées, alors que dans l'espace-temps cher à Albert Einstein (entre autres), il en faudra 4. D'une façon générale, il existe des « espaces » ayant un *très* grand nombre n de dimensions, et les vecteur dans ces espaces ont donc n coordonnées, qu'on peut représenter sous la forme d'une liste Python de n éléments. Peu importe la valeur de n, les règles en vigueur dans le plan restent valables :

- ajouter deux vecteurs revient à ajouter leurs coordonnées, « position après position » ;
- multiplier un vecteur par un réel revient à multiplier chacune des coordonnées par ce réel.

1. Écrire une fonction `somme_vecteurs()` qui prend en argument deux listes Python contenant les coordonnées de deux vecteurs, et renverra les coordonnées du vecteur somme. Cette fonction devra être capable de fonctionner *quel que soit le nombre de coordonnées* commun aux deux vecteurs passés en arguments !

   Validation :
   ```
   assert somme_vecteurs([1, 2, 3, 4], [5, 6, 7, 8]) == [6, 8, 10, 12],
                  "/!\ somme_vecteurs()"
   ```

2. Écrire une fonction `produit_vecteur_reel()` qui prend en argument une liste Python contenant les coordonnées d'un vecteur et un nombre réel. Elle renverra les coordonnées du vecteur obtenu suite au produit par le réel. Cette fonction devra être capable de fonctionner *quel que soit le nombre de coordonnées* du vecteur passé en argument !

   Validation :
   ```
   assert produit_vecteur_reel([1, 2, 3, 4], -2) == [-2, -4, -6, -8],
                  "/!\ produit_vecteur_reel()"
   ```

EXERCICE 21 ★★☆ — *Croissance ou décroissance ?*

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 : dès que 2 éléments s'enchaînent dans un ordre inadéquat, on peut souhaiter interrompre là le parcours de la liste (question d'efficacité !). Pour cela, on utilisera, au choix, soit un `return False` « prématuré », soit l'instruction `break` qui casse la boucle en cours (ces approches sont néanmoins souvent considérées comme étant inélégantes, voire problématiques).

Validation :
```
assert est_croissante([1, 2, 3]) == True, "/!\ est_croissante()"
assert est_croissante([1, 3, 2]) == False, "/!\ est_croissante()"
assert est_decroissante([3, 2, 1]) == True, "/!\ est_decroissante()"
```

Bonus : *pour éviter un `break` que certains informaticiens n'apprécient pas, vous réécrirez le code de ces fonctions à l'aide du boucle `while` dont le test portera sur une variable booléenne qui vaudra `True` tant qu'on n'a pas rencontré de souci, et passera à `False` au premier accroc rencontré.*

EXERCICE 22 ★★★ — *Différences successives*

Écrire une fonction `diff_successives()` qui prend en argument une liste Python de nombres et retourne une liste constituée des différences de chaque paire de nombres consécutifs de la liste.

Exemple : imaginons qu'on dispose dans une liste Python de températures relevées à midi 7 jours d'affilée. Cet exercice nous amène à déterminer les 6 évolutions journalières entre deux températures consécutives. Avec la liste `[15, 20, 22, 18, 18, 20, 17]`, on obtiendra en retour `[5, 2, -4, 0, 2, -3]` (en effet, $20-15=15$, puis $22-20=2$, puis $18-22=-4$, etc.).

Validation :
```
assert diff_successives([15, 20, 22, 18, 18, 20]) == [5, 2, -4, 0, 2],
       "/!\ diff_successives()"
```

EXERCICE 23 ★★★ — *Extraction !*

Écrire une fonction `plus_longue_suite_croissante()` qui retourne une liste comprenant la plus longue suite d'éléments *successifs* qui se suivent dans un ordre croissant. S'il existe plusieurs réponses possibles, la première suite trouvée aura la priorité.

Validation :
```
assert plus_longue_suite_croissante([1, 5, 2, 3, 4, 0, 6]) == [2, 3, 4],
       "/!\ plus_longue_suite_croissante()"
assert plus_longue_suite_croissante([1, 5, 7, 2, 3, 4, 0]) == [1, 5, 7],
       "/!\ plus_longue_suite_croissante()"
assert plus_longue_suite_croissante([5, 4, 3, 2, 1, 0]) == [],
       "/!\ plus_longue_suite_croissante()"
```

EXERCICE 24 ★★☆ — *Distance de Hamming*

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 »).

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

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).*

EXERCICE 25 ★★½ — *Autour de l'ADN*

L'**ADN** est une macromolécule présente dans toutes les cellules des êtres vivants (ainsi que chez de nombreux virus). Elle contient toute l'information génétique (le génome) des êtres vivants, qui gouverne leur développement physique, leur fonctionnement et leur reproduction.

Les molécules d'ADN des cellules vivantes sont formées de deux brins enroulés l'un autour de l'autre en une « double hélice ». Sur chacun de ces deux brins d'ADN, 4 bases azotées différentes s'enchaînent : adénine (`"A"`), thymine (`"T"`), cytosine (`"C"`) et guanine (`"G"`). Ces 4 bases ont des affinités *deux à deux* : l'adénine ne peut se lier qu'avec la thymine, la cytosine ne peut s'apparier qu'avec la guanine. Ce sont ces associations qui sont à l'origine de cette forme si particulière que présente naturellement l'ADN.

1. Écrire une fonction `fabrique_brin_adn()` qui prend comme paramètre un entier `n` et renvoie une liste Python contenant une succession de `n` caractères tirés au hasard parmi `"A"`, `"T"`, `"C"` et `"G"`.

   Aide : vous pourrez utiliser la fonction `choice()` du module `random`.

   Validation : impossible, car la valeur de retour de cette fonction est générée au hasard !
2. Écrire une fonction `deduire_brin_adn()` qui prend comme paramètre une liste Python représentant un brin d'ADN, et renvoie une liste Python représentant le brin d'ADN manquant, nécessaire à la modélisation complète de l'ADN.

Validation :

```
assert deduire_brin_adn(["A", "T", "G", "C"]) == ["T", "A", "C", "G"],
       "/!\ deduire_brin_adn()"
```

3. Écrire une fonction `verifie_adn()` qui prend comme paramètre deux listes Python représentant deux brins d'ADN, et renvoie `True` si ces listes sont correctement appariées, `False` sinon.

   Validation :

```
assert verifie_adn(["A", "T", "G", "A"], ["T", "A", "C", "T"]) == True,
       "/!\ verifie_adn()"
assert verifie_adn(["A", "T", "G", "A"], ["T", "A", "C", "G"]) == False,
       "/!\ verifie_adn()"
```

Remarque : *l'ordinateur à ADN est une des voies non électroniques actuellement explorées pour résoudre des problèmes combinatoires (qui demandent une forte puissance de calcul, qu'on obtient avec l'informatique électronique traditionnelle en multipliant les cœurs des processeurs, les processeurs et/ou les unités centrales... ou en utilisant la physique quantique !). En janvier 2020, un ordinateur à ADN est parvenu à calculer... $\sqrt{900}$! On n'en est clairement qu'aux balbutiements... Quant à l'ordinateur quantique, je vous suggère de regarder cette vidéo ainsi que celle-ci.*