Correction du devoir surveillé 2

Applications directes du cours

1. Traçons la courbe de la fonction $f : x \longmapsto x + \sin(x)$ sur l'intervalle $[-7; 7]$, avec mille points.

   Pour cela, commençons par coder la fonction f.

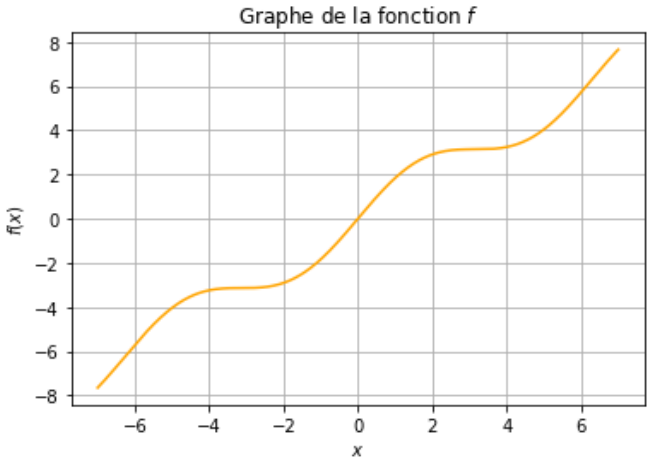
```
import numpy as np

def f(x):
    return x + np.sin(x)
```

   On crée alors une liste de mille abscisses de $[-7; 7]$ régulièrement espacées, ainsi que la liste de leurs images par f, que l'on placera en ordonnées.

```
a = -7
b = 7
h = (b - a) / 999
    # On subdivise l'intervalle [-7 ; 7] en 999 sous-segments.

abscisses = [a + k * h for k in range(1000)]
    # On crée la liste de tous les nombres a + k * h,
    # pour k allant de 0 à 999.
    # Le premier élément de la liste est donc a + 0 * h = a
    # et le dernier élément de la liste est a + 999 * h = b.
    # La liste contient bien mille points.

ordonnees = [f(x) for x in abscisses]
```

   Traçons alors la courbe de la fonction f.

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(abscisses, ordonnees)
plt.show()
```



2. Pour déterminer si un objet x est un élément d'une liste L, on parcourt un à un les éléments de la liste L.
   Dès que l'on trouve l'élément x, on arrête immédiatement la recherche.
   Si on arrive à la fin du parcours sans avoir trouvé x, alors la liste L ne contient pas x.

```
def recherche(L, x):
    """
    Entrées : une liste L et un objet x.
    Sortie : True si x appartient à L, False sinon.
    """
    for elem in L:      # Pour elem parcourant un à un
                        # les éléments de la liste L,
        if elem == x:       # si l'élément elem vaut x,
            return True         # alors on renvoie True
                                # et la fonction s'arrête.
    return False        # Si on arrive à la fin du parcours,
                        # c'est que la fonction ne s'est pas
                        # arrêtée avant, donc que x n'est pas
                        # un élément de la liste L.
```

3. Écrivons une fonction qui prend en argument une liste de nombres et qui la trie dans l'ordre croissant, en utilisant l'algorithme de tri par insertion, avec des échanges successifs.

   Commençons par définir une fonction d'échange.

```
def echange(L, i, j):
    """
    Entrées : une liste L et deux indices i et j de la liste L.
    Échange les éléments d'indices i et j dans la liste L.
    Sortie : aucune, la liste L est modifiée en place.
    """
    aux = L[i]
    L[i] = L[j]
    L[j] = aux
```

   Implémentons alors le tri par insertion.

```
def tri_insertion(L):
    """
    Entrée : une liste L de nombres.
    Sortie : la liste L triée, avec l'algorithme de tri par insertion.
    """
    n = len(L)
    for k in range(1, n):
    # Pour k allant de 1 à n - 1,
        # on insère l'élément L[k] dans le début de la liste
        # de la manière suivante.

        pos = k # Position à laquelle se trouve
                # l'élément que l'on cherche à insérer
                # (initialement, il est à l'indice k).
```

```
        while (pos > 0) and (L[pos] < L[pos - 1]):
            echange(L, pos, pos - 1)
            pos = pos - 1
        # Tant que l'élément n'est pas en première position
        # et tant que l'élément d'indice pos est inférieur
        # à l'élément précédent dans la liste,
            # on échange les éléments d'indices pos et pos - 1.
            # L'élément qui nous intéresse se trouve
            # désormais à l'indice pos - 1.
    return L
```

4. Considérons la suite $(u_k)_{k\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = 5 \\ \forall k \in \mathbb{N}, u_{k+1} = \sqrt{2+u_k}. \end{cases}$$

Pour calculer son terme de rang n, on procède de la manière suivante.

- On commence par créer une variable `terme` qui vaut initialement u_0, c'est-à-dire 5.
- Puis on fait une boucle, à chaque tour de laquelle on applique à la variable `terme` la formule de récurrence qui définit la suite.
  La boucle fera n tours :
  - ⋆ après le premier tour de boucle, la variable `terme` vaut u_1 ;
  - ⋆ après le deuxième tour de boucle, la variable `terme` vaut u_2 ;
  - ⋆ …
  - ⋆ après n tours de boucle, la variable `terme` vaut u_n.
- À l'issue de la boucle, on renvoie la variable `terme`, qui vaut u_n.

```
import numpy as np

def suite(n):
    """
    Entrée : un entier naturel n.
    Sortie : le terme u_n de rang n.
    """
    terme = 5                       # Avant la boucle, terme vaut u_0.
    for i in range(1, n + 1):       # Pour i allant de 1 à n,
                                    # c'est-à-dire n fois,
        terme = np.sqrt(2 + terme)  # on applique la formule de
                                    # récurrence à la variable terme,
                                    # donc terme prend pour valeur u_i.
    return terme                    # Après la boucle, terme vaut u_n.
```

Exercice — Pile pile face (d'après un oral Agro-Véto 2024)
Soit $p \in [0;1]$. On dispose d'une pièce de monnaie qui tombe sur pile avec probabilité p.
Dans la suite, on codera le côté pile par 1 et le côté face par 0.

1. (a) Comme la pièce tombe sur pile avec probabilité p, la variable aléatoire qui donne le numéro (0 ou 1) associé au résultat d'un lancer de la pièce suit une loi de Bernoulli de paramètre p. On la simule de la manière suivante.

```
import numpy.random as nr

def piece(p):
    """
    Entrée : un flottant p compris entre 0 et 1.
    Sortie : 1 (pile) avec probabilité p,
             0 (face) avec probabilité 1 - p.
    """
    u = nr.rand()    # On tire un flottant aléatoire dans [0 ; 1[,
                     # selon la loi uniforme.
    if u < p:        # Si u < p, ce qui arrive avec proba p,
        return 1         # on renvoie 1 : la pièce tombe sur pile.
    else:            # Sinon (donc avec proba 1 - p),
        return 0         # on renvoie 0 : la pièce tombe sur face.
```

   (b) La fonction suivante prend en argument la probabilité p et un entier N, et crée une liste contenant le résultat de N lancers de la pièce indépendants.

```
def lancers_successifs(p, N):
    """
    Entrées : - un flottant p compris entre 0 et 1,
              - un entier N.
    Sortie : liste des résultats de N lancers de la pièce.
    """
    return [piece(p) for lancer in range(N)]
        # Pour lancer allant de 0 à N - 1, donc N fois,
        # on tire la pièce.
```

2. (a) Pour simuler des lancers de la pièce et déterminer le nombre de lancers nécessaires à l'obtention de la configuration pile-pile-face, on commence par effectuer trois lancers de la pièce, à l'aide de la fonction précédente, et on stocke les résultats dans une liste.

   Alors tant que les trois derniers lancers n'ont pas donné pile-pile-face, c'est-à-dire tant que les trois derniers éléments de la liste des résultats ne valent pas 1, 1 puis 0, on effectue un lancer supplémentaire de la pièce : on ajoute le résultat du lancer à la fin de la liste et on incrémente le compteur de lancers.

   À la sortie de la boucle, c'est-à-dire dès que l'on a obtenu pile-pile-face, on renvoie le compteur de lancers.

```
def pile_pile_face(p):
    """
    Entrée : un flottant p compris entre 0 et 1.
    Sortie : le nombre de lancers nécessaires
```

```
            à obtenir pile-pile-face.
    """
    L = lancers(3)  # Liste des résultats obtenus.
    n = 3           # Nombre de lancers effectués.
    while not((L[n - 3] == 1) and (L[n - 2] == 1) and (L[n - 1] == 0)):
                    # Tant que les trois derniers lancers
                    # ne donnent pas pile pile face,
                        # on fait un lancer supplémentaire.
        L.append(piece(p))
        n = n + 1
    return n
```

(b) Pour estimer le nombre moyen de lancers nécessaire à l'obtention de la configuration pile-pile-face, on répète un grand nombre de fois l'expérience et on moyenne les résultats obtenus par la fonction précédente sur ce grand nombre d'expériences.

```
def temps_moyen(p, N):
    """
    Entrées : - un flottant p compris entre 0 et 1,
              - un entier N (grand).
    Sortie : moyenne sur N expériences du nombre
             de lancers nécessaires à obtenir
             pile-pile-face.
    """
    S = 0
    for experience in range(N): # On fait N tours de boucle :
                                # on fait N expériences.
        S = S + pile_pile_face(p)   # On ajoute à S le nombre
                                    # de lancers nécessaires à
                                    # obtenir pile-pile-face.
    return S / N
```

Problème 1 — The Matrix Recoded

Partie I : Opérations matricielles

1. La fonction suivante calcule la somme des deux matrices passées en arguments.

```
def somme_matricielle(A, B):
    """
    Entrées : deux matrices A et B de même taille,
              représentées comme listes de listes.
    Sortie : la matrice A + B.
    """
    nA = len(A)      # Nombre de lignes de la matrice A.
    pA = len(A[0])   # Nombre d'éléments de la première ligne de A,
                     # càd nombre de colonnes de la matrice A.
    nB = len(B)      # Nombre de lignes de la matrice B.
    pB = len(B[0])   # Nombre de colonnes de la matrice B.
```

```
    # On teste si les matrices A et B ont bien même taille.
    if nA != nB or pA != pB:
        return "Erreur, les matrices doivent avoir même taille !"
    else: # Dans ce cas, A et B ont bien la même taille.
        S = [pA * [0] for i in range(nA)]
        # S est initialement la matrice nulle
        # à nA lignes et pA colonnes,
        # et sera la somme des matrices A et B.
        for i in range(nA): # i parcourt les indices
                            # des lignes de A et B.
            for j in range(pA): # j parcourt les indices
                                # des colonnes de A et B.
                S[i][j] = A[i][j] + B[i][j]
                # Le coefficient à la ligne i et à la colonne j
                # dans la matrice S devient la somme des coefficients
                # à la ligne i et la colonne j dans A et B.
        return S
```

2. La fonction suivante calcule le produit des deux matrices passées en arguments.

```
def produit(A, B):
    """
    Entrées : deux matrices A et B.
    Sortie : le produit matriciel AB s'il est bien défini,
             un message d'erreur sinon.
    """
    nA = len(A)      # Nombre de lignes de A.
    pA = len(A[0])   # Nombre de colonnes de A.
    nB = len(B)      # Nombre de lignes de B.
    pB = len(B[0])   # Nombre de colonnes de B.
    if pA != nB:
        return "Le produit n'est pas défini."
    else:
        P = [pB * [0] for ligne in range(nA)]
        # P sera le produit AB des deux matrices,
        # P possède nA lignes et pB colonnes.
        for i in range(nA):
        # i parcourt les indices de lignes de P.
            for j in range(pB):
            # j parcourt les indices de colonnes de P.

                # On calcule le coefficient du produit AB
                # à la ligne i et à la colonne j,
                # qui est initialisé à 0.

                for k in range(pA):
                # k parcourt les indices de colonnes de A,
                # qui sont aussi les indices de lignes de B.
                    P[i][j] += A[i][k] * B[k][j]

        return P
```

Partie II : Matrices magiques (d'après un oral Agro-Véto 2025)

On dit qu'une matrice M est une **matrice magique** lorsque la somme des éléments de chaque ligne et de chaque colonne de M est la même.

3. (a) La fonction suivante prend en arguments une matrice M et un indice i de ligne de M, et renvoie la somme des éléments de la ligne i de la matrice M, c'est-à-dire la somme des éléments de la liste $M[i]$.

```
def somme_ligne(M, i):
    """
    Entrées : une matrice M et un entier i.
    Sortie : la somme des éléments de la ligne i de la matrice M.
    """
    S = 0
    p = len(M[0])   # Nombre de colonnes de M.
    for j in range(p):
    # Pour j allant de 0 à p - 1,
    # càd pour j parcourant les indices de colonnes de M,
        S += M[i][j]
        # on ajoute à S l'élément d'indice j de M[i],
        # càd le coefficient de M à la ligne i et à la colonne j.
    return S
```

   (b) La fonction suivante prend en arguments une matrice M et un indice j de colonne de M, et renvoie la somme des éléments de la colonne j de la matrice M.

```
def somme_colonne(M, j):
    """
    Entrées : une matrice M et un entier j.
    Sortie : la somme des éléments de la colonne j de la matrice M.
    """
    S = 0
    n = len(M)  # Nombre de lignes de M.
    for i in range(n):
    # Pour i allant de 0 à n - 1,
    # càd pour i parcourant les indices de lignes de M,
        S += M[i][j]
        # on ajoute S à le coefficient de M
        # à ligne i et à la colonne j.
    return S
```

4. Pour déterminer si une matrice M est magique, on commence par calculer la somme des éléments de la première ligne de M, qui sera notre somme de référence.

   On compare ensuite la somme de chaque ligne et la somme de chaque colonne avec cette somme de référence.

   Si l'on rencontre une ligne ou une colonne dont la somme des éléments n'est pas égale à la première somme calculée, la matrice n'est pas magique et on renvoie `False` immédiatement.

   Si la fonction ne s'est pas arrêtée avant la fin du parcours, alors la somme sur toutes les lignes et sur toutes les colonnes de M est la même, donc la matrice M est magique, et on renvoie `True`.

```
def magique(M):
    """
    Entrée : une matrice M.
    Sortie : True si la matrice M est magique, False sinon.
    """
    n = len(M)      # Nombre de lignes de M.
    p = len(M[0])   # Nombre de colonnes de M.
    somme_ref = somme_ligne(M, 0)
    for i in range(n):
    # Pour i allant de 0 à n - 1,
    # c'est-à-dire pour i parcourant les indices des lignes de M.
        if somme_ligne(M, i) != somme_ref :
            return False
    for j in range(p):
    # Pour j allant de 0 à p - 1,
    # c'est-à-dire pour j parcourant les indices des colonnes de M.
        if somme_colonne(M, j) != somme_ref:
            return False
    return True
```

Problème 2 — Chiffrement de César

Dans ce problème, on se propose tout d'abord d'implémenter le cryptage et le décryptage d'un message selon un codage substitutif quelconque. On implémentera ensuite le chiffrement de César, puis on en proposera une méthode de déchiffrement.

On supposera que les textes à crypter sont tous composés de lettres majuscules non accentuées et d'espaces, sans ponctuation.

Partie I : Substitutions monoalphabétiques

Un **chiffrement par substitution monoalphabétique** est une méthode de cryptage très simple qui consiste à remplacer chaque lettre de l'alphabet par une autre.

La permutation des lettres utilisée pour un tel chiffrement se représente comme un dictionnaire, dont les clés sont les lettres de l'alphabet et les valeurs les lettres correspondantes après substitution.

1. Pour crypter une chaine de caractères à l'aide d'un dictionnaire de codage, on commence par initialiser le message crypté à la chaine vide.

   On parcourt ensuite chaque caractère de la chaine et on concatène au message codé la valeur (c'est-à-dire la lettre) associée au caractère dans le dictionnaire.

```
def cryptage(Dico, message):
    """
    Entrées :
        - un dictionnaire de codage Dico ;
        - une chaine de caractères message.
    Sortie : la chaine de caractères message cryptée
             avec le codage donné par le dictionnaire Dico.
```

```
    """
    message_code = ""
    for lettre in message:
    # lettre parcourt tous les caractères de la chaine message.
        lettre_codee = Dico[lettre]
        message_code += lettre_codee
    return message_code
```

2. (a) Le dictionnaire réciproque d'un dictionnaire de codage `Dico` est le dictionnaire dans lequel on échange les rôles des clés et des valeurs, c'est-à-dire le dictionnaire dont les clés sont les valeurs de `Dico` et les valeurs sont les clés de `Dico` correspondantes.

   Pour le construire, on commence par initialiser le dictionnaire réciproque au dictionnaire vide.

   Puis pour chaque lettre, c'est-à-dire pour chaque clé du dictionnaire `Dico`, on ajoute au dictionnaire réciproque une nouvelle entrée qui associe à la lettre cryptée (la valeur associée à la lettre dans `Dico`) la lettre initiale (la clé dans `Dico`).

```
def reciproque(Dico):
    """
    Entrée : un dictionnaire de codage Dico.
    Sortie : son dictionnaire réciproque (de décodage).
    """
    Dicodage = {}
    for cle in Dico:
    # Pour chaque clé cle du dictionnaire Dico,
        Dicodage[Dico[cle]] = cle
        # on crée une entrée dans Dicodage,
        # de clé Dico[cle] (la valeur associée à cle dans Dico)
        # et de valeur cle.
    return Dicodage
```

   (b) Pour décrypter un message codé en connaissant le dictionnaire de codage, il suffit alors d'appliquer la fonction `cryptage` avec le dictionnaire réciproque du dictionnaire de codage.

```
def decryptage(Dico, message_code):
    """
    Entrées :
        - un dictionnaire de codage Dico ;
        - une chaine de caractères message_code.
    Sortie : la chaine message_code décryptée.
    """
    Dicodage = reciproque(Dico)
    return cryptage(Dicodage, message_code)
```

Partie II : Chiffrement par décalage

3. Commençons par définir une chaine de caractères contenant toutes les lettres de l'alphabet, dans l'ordre alphabétique :

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Pour décaler une lettre de d places dans l'ordre alphabétique, on procède de la manière suivante. Si la lettre est d'indice k dans cette chaine `ALPHABET`, la lettre décalée est celle d'indice $k + d$ modulo 26 : lorsque l'on décale la lettre `"Z"` (d'indice 25) d'une place, on revient à la lettre `"A"` (d'indice $0 = (25 + 1) \bmod 26$).

La fonction suivante renvoie alors le dictionnaire associé au chiffrement de César pour un décalage du nombre de places d passé en argument.

```
def DiCaesar(d):
    """
    Entrée : un entier d.
    Sortie : le dictionnaire de codage associé au
             chiffrement de César pour décalage de d lettres
             dans l'ordre alphabétique.
    """
    Dicalage = {' ' : ' '}
    # Les espaces sont inchangées par ce chiffrement.
    for k in range(26):      # k va de 0 à 25.
        lettre = ALPHABET[k]
        lettre_codee = ALPHABET[(k + d) % 26]
        Dicalage[lettre] = lettre_codee
    return Dicalage
```

Si l'on est destinataire d'un message codé, alors on connait très probablement le dictionnaire de codage, donc il est facile de décrypter le message (avec les fonctions de la partie précédente).

Cependant, lorsque l'on intercepte un message codé, il est rare que l'on dispose également du dictionnaire de codage. Dans le cas du chiffrement de César, pour trouver rapidement — c'est-à-dire plus rapidement qu'en testant toutes les possibilités — le nombre d de décalages effectués, on peut utiliser une analyse fréquentielle du texte à décrypter, c'est-à-dire trouver la fréquence d'apparition de chacune des lettres.

En français, la lettre la plus fréquente (dans un texte suffisamment long) est la lettre `"E"`, la deuxième plus fréquente est la lettre `"A"`.[1] Pour décoder un message, on pourra alors tester en priorité le décalage pour lequel la lettre la plus fréquente du message codé représente la lettre `"E"`.

4. La fonction suivante prend en arguments une chaine de caractères et un caractère, et renvoie le nombre d'occurrences du caractère dans la chaine.

```
def occurrences(chaine, caractere):
    """
    Entrées :
        - une chaine de caractères chaine ;
        - un caractère caractere.
    Sortie : le nombre d'occurrences de caractère
             dans chaine.
    """
```

1. En latin, ce sont les lettres `"I"` et `"E"`.

```
    nombre_occ = 0
    for lettre in chaine:
    # lettre parcourt les caractères de la chaine.
        if lettre == caractere:
            nombre_occ += 1
    return nombre_occ
```

5. (a) La fonction suivante prend en argument une chaine de caractères et renvoie le dictionnaire dont les clés sont les caractères de la chaine et les valeurs leur nombre d'occurrences dans la chaine.

   On le construit en compréhension, les nombres d'occurrences étant calculés à partir de la fonction de la question précédente.

```
def dicoccurrences(chaine):
    """
    Entrée : une chaine de caractères chaine.
    Sortie : le dictionnaire dont les clés sont
             les caractères de chaine et les valeurs
             leur nombre d'occurrences dans chaine.
    """
    return {car : occurrences(chaine, car) for car in chaine}
```

   (b) Pour trouver dans une chaine de caractères la lettre différente de l'espace la plus fréquente, on commence par construire le dictionnaire de la question précédente, puis on trouve en la clé dont la valeur associée est maximale.

```
def lettre_frequente(chaine):
    """
    Entrée : une chaine de caractères.
    Sortie : le caractère le plus fréquent dans chaine
             qui n'est pas une espace.
    """
    Dico = dicoccurrences(chaine)
    maxi = 0    # Nombre d'occurrences maximal.
    for lettre in Dico:
    # lettre parcourt les clés du dictionnaire.
        if lettre != " " and Dico[lettre] > maxi:
            lettre_gagnante = lettre
            # lettre_gagnante est la clé rencontrée
            # ayant la plus grande valeur associée
            # dans le dictionnaire Dico.
            maxi = Dico[lettre]
    return lettre_gagnante
```

6. Commençons par écrire une fonction qui prend en entrée une lettre de l'alphabet et qui renvoie le nombre de places qui séparent la lettre `"E"` de la lettre, c'est-à-dire la valeur de d telle que le chiffrement de César avec un décalage de d places envoie la lettre `"E"` sur la lettre considérée.

```
def decalage_E(lettre):
    """
```

```
    Entrée : une lettre de l'alphabet lettre.
    Sortie : le nombre de places qui séparent "E"
             de la lettre dans l'ordre alphabétique.
    """
    # On cherche l'indice de la lettre dans l'alphabet.
    for k in range(26):
        if ALPHABET[k] == lettre:
            return k - 4
            # "E" est la lettre d'indice 4 dans l'alphabet.
```

Pour décrypter un message codé par le chiffrement de César, on commence alors par trouver la lettre la plus fréquente dans le message codé.

Sous l'hypothèse que la lettre la plus fréquente dans le message initial est le `"E"`, la lettre trouvée est l'image de `"E"` par le chiffrement de César. On retrouve de combien de places le `"E"` a été ainsi décalé à l'aide de la fonction précédente. On en déduit alors le dictionnaire de codage utilisé pour le chiffrement, à l'aide de la question 3.

Enfin, on décrypte le message codé à l'aide de la question 2.

```
def dechiffrement_en_E_majeur(message_code):
    """
    Entrée : une chaine de caractère message_code.
    Sortie : la chaine message_code décryptée
             sous l'hypothèse que la lettre
             la plus fréquente est "E".
    """
    lettre = lettre_frequente(message_code)
    d = decalage_E(lettre)
    return decryptage(DiCaesar(d), message_code)
```