Algorithmes opérant par boucles imbriquées

Dans ce chapitre, nous allons étudier plusieurs problèmes nécessitant des boucles imbriquées pour être résolus, c'est-à-dire une boucle à l'intérieur d'une autre boucle.

I. Recherche d'un mot dans un texte

Dans le chapitre sur les chaines de caractères, nous avons étudié et résolu le problème de la recherche d'un caractère dans une chaine de caractères. Ici le problème est plus complexe : on cherche à savoir si un mot (ou une sous-chaine) est présent dans un texte.

L'idée de l'algorithme est la suivante :

- ⋆ On parcourt les caractères du texte et pour chaque caractère, on regarde si le mot commence à cet endroit.
- ⋆ On parcourt alors le mot pour voir si tous les caractères correspondent à partir de là dans le texte :
  - Si c'est le cas, alors le mot est trouvé et on peut renvoyer `True`.
  - Sinon on passe au caractère suivant dans le texte.
- ⋆ Comme souvent, si on n'a pas renvoyé `True` avant la fin de la boucle c'est que le mot est absent du texte. On renvoie alors `False` après la boucle.

Problème technique à gérer : il faut faire attention à ne pas provoquer d'erreurs du type `IndexError`.

```
def recherche_mot(mot, texte):
    """
    Entrées : mot (chaine de caractères)
              texte (chaine de caractères)
    Sortie : True si mot est une suite de caractères consécutifs dans texte
             False sinon
    """
    for i in range(len(texte) - len(mot) + 1):  # mot ne peut pas commencer après cet
                                                # indice sans dépasser le texte
        # Déterminons si mot commence à partir du caractère d'indice i dans texte.
        # Pour cela, on crée une variable booléenne test qui passera à False si on
        # trouve un caractère qui ne correspond pas à celui de mot à la même place.
        test = True
        for j in range(len(mot)):
            if mot[j] != texte[i+j]:    # le caractère ne correspond pas
                test = False
        # À ce stade, la variable test contient l'information recherchée mais si
        # test = False, cela ne veut pas dire que mot n'est pas présent dans le
        # texte : il peut être ailleurs.
        if test:
            return True
    # Si la boucle se termine, alors le mot n'a pas été trouvé dans le texte.
    return False
```

Autre solution plus élégante utilisant le slicing et permettant d'éviter le recours aux boucles imbriquées :

```
def recherche_mot(mot, texte):
    """
    Entrées : mot (chaine de caractères)
              texte (chaine de caractères)
    Sortie : True si mot est une suite de caractères consécutifs dans texte
             False sinon
    """
    for i in range(len(texte) - len(mot)):
        if mot == texte[i:i+len(mot)]:
            return True
    return False
```

II. Recherche des deux valeurs les plus proches dans une liste

Considérons une liste numérique `L`. On souhaite écrire une fonction `ecart_min(L)` qui calcule l'écart minimal entre deux éléments distincts de `L` (les éléments peuvent avoir la même valeur mais doivent être à des endroits différents dans `L`). Par exemple :

```
>>> ecart_min([-13, -10, -19, 11, 20])
3
>>> ecart_min([-17, -13, -19, -11, 11, -13])
0
```

Pour résoudre ce problème, il faut s'inspirer de l'algorithme du calcul du minimum d'une liste : on applique le principe du record à battre. Ainsi :

- ⋆ On initialise le record avec l'écart entre les deux premiers éléments de la liste.
- ⋆ Puis on examine tous les écarts possibles entre deux éléments distincts de `L` et on actualise le record si celui-ci est battu.

```
def ecart_min(L):
    """
    Entrée : L une liste numérique ayant au moins deux éléments
    Sortie : L'écart minimal entre deux éléments distincts de L
    """
    record = abs(L[0] - L[1])

    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and abs(L[i] - L[j]) < record:
                record = abs(L[i] - L[j])

    return record
```

Amélioration du programme en tenant compte du fait qu'on examine deux fois chaque écart :

```
def ecart_min(L):
    """
    Entrée : L une liste numérique ayant au moins deux éléments
    Sortie : L'écart minimal entre deux éléments distincts de L
    """
    record = abs(L[0] - L[1])

    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if abs(L[i] - L[j]) < record:
                record = abs(L[i] - L[j])

    return record
```

III. Tableaux à deux dimensions

Une liste de nombres peut être vue comme un tableau à une dimension car, pour accéder à un élément de la liste, on a besoin de donner un nombre : son indice.

Un tableau à deux dimensions est donc un objet contenant des nombres et pour lequel on a besoin de donner deux nombres pour accéder à un élément.

1) Listes de listes

Une liste de listes de nombres peut être vue comme un tableau à deux dimensions.

Par exemple :

```
L = [[1, 2, 3], [2, -1], [3, 5, 9, 10, 0]]
```

que l'on peut aussi écrire :

```
L = [
  [1, 2, 3],
  [2, -1],
  [3, 5, 9, 10, 0]
]
```

a) Accès aux éléments

Pour accéder à un des nombres qui composent `L`, un seul indice ne suffit pas. En effet :

```
>>> L[0]
[1, 2, 3]
>>> L[1]
[2, -1]
>>> L[2]
[3, 5, 9, 10, 0]
```

puisque `L[i]` donne l'élément d'indice `i` de `L` qui est bien une liste.

Pour obtenir, par exemple, le nombre `10` de la liste `L`, il faut écrire que ce nombre est l'élément d'indice `3` de l'élément d'indice `2` de `L`. Ainsi :

```
>>> L[2][3]
10
```

b) Nombre de lignes, nombre de colonnes

En utilisant le vocabulaire des tableaux, les « lignes » de `L` sont les sous-listes qui la composent. Dans la première ligne (qui est la ligne `L[0]`), il y a 3 nombres donc 3 colonnes.

La fonction `len` appliquée à `L` donne le nombre de lignes. Lorsqu'on l'applique à une ligne, elle nous donne le nombre de colonne de cette ligne :

```
>>> len(L)
3
>>> len(L[0])
3
>>> len(L[1])
2
>>> len(L[2])
5
```

c) Parcours des éléments du tableau

Essayons maintenant d'afficher tous les nombres qui composent le tableau `L` :

Code :

```
for i in range(len(L)):
    for j in range(len(L[i])):
        print(L[i][j])
```

Résultat :

```
1
2
3
2
-1
3
5
9
10
0
```

Remarque : Dans cet exemple, l'utilisation du terme « tableau » pour la liste `L` est un peu abusif : en général, on utilise ce terme uniquement dans le cas où toutes les listes qui composent `L` sont de même taille.

Par exemple :

```
L = [
  [1, 2, 3, 4],
  [-1, 2, -3, 4],
  [0, 1, -1, 2]
]

print(L[1][2])  # Élément à la ligne d'indice 1 et à la colonne d'indice 2
                # donc 2e ligne et 3e colonne

n = len(L)      # Nombre de lignes
p = len(L[0])   # Nombre de colonnes et on a aussi p = len(L[1]) = len(L[2])

for i in range(n):
    for j in range(p):
        print(L[i][j])
```

2) Le module `numpy`

Le module `numpy` permet de travailler avec un nouveau type d'objets : les tableaux (`numpy.ndarray`). Ces tableaux peuvent être de n'importe quelle dimension $d \in \mathbb{N}^*$.

Dans le cas d'un tableau à deux dimensions, un tableau `numpy` se comporte un peu comme une liste de listes de nombres mais avec des syntaxes supplémentaires permettant notamment de faire du calcul matriciel.

Pour utiliser ce module, il faut bien sûr commencer par l'importer :

```
import numpy as np  # On utilise l'importation avec l'alias np (convention classique)
```

On dispose alors des fonctions suivantes :

Fonction	Résultat	Type de retour
`np.array(liste)`	Crée un tableau à partir d'une liste (de listes)	`np.ndarray`
`np.zeros((n_1, n_2, ..., n_d))`	Crée un tableau rempli de zéros de dimension `d` et de taille `(n_1, n_2, ..., n_d)`	`np.ndarray`
`np.ones((n_1, n_2, ..., n_d))`	Crée un tableau rempli de uns de dimension `d` et de taille `(n_1, n_2, ..., n_d)`	`np.ndarray`
`np.shape(tableau)`	Renvoie la taille du tableau	`tuple`

Exemple :

```
>>> L = [
  [1, 2, 3, 4],
  [-1, 2, -3, 4],
  [0, 1, -1, 2]
]
>>> T1 = np.array(L)    # Tableau à 3 lignes et 4 colonnes
>>> np.shape(T1)
(3, 4)
>>> T1
array([[ 1,  2,  3,  4],
       [-1,  2, -3,  4],
       [ 0,  1, -1,  2]])
>>> T2 = np.zeros((2, 4))   # Tableau de zéros à 2 lignes et 4 colonnes
>>> T2
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> T3 = np.ones((3, 2))    # Tableau de uns à 3 lignes et 2 colonnes
>>> T3
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

a) Accès aux éléments

Pour accéder aux éléments d'un tableau à deux dimensions, on peut utiliser la syntaxe :

```
tableau[i, j]
```

où `i` est l'indice de la ligne et `j` l'indice de la colonne de l'élément.

Exemple :

```
>>> T1[0, 0]
1
>>> T1[1, 2]
-3
>>> T1[-1, -1]  # Comme pour les listes, les indices négatifs
2               # peuvent être utilisés
```

Remarque : Il est aussi possible d'utiliser la même syntaxe que pour les listes de listes de nombres : `tableau[i][j]` mais c'est plutôt déconseillé.

En revanche, utiliser la syntaxe `L[i, j]` avec `L` une liste de listes ne fonctionnera pas.

b) Accès à une ligne ou une colonne

Un avantage important des tableaux `numpy` par rapport aux listes de listes est qu'on peut facilement accéder non seulement aux lignes, mais aussi aux colonnes (ce qui est difficile avec des listes de listes).

Pour récupérer la ligne d'indice `i`, on utilise la syntaxe :

```
tableau[i]
# ou bien
tableau[i, :]
```

et pour récupérer la colonne d'indice `j`, on utilise la syntaxe :

```
tableau[:, j]
```

Exemple :

```
>>> T1[1]
array([-1,  2, -3,  4])
>>> T1[2, :]
array([ 0,  1, -1,  2])
>>> T1[:, 2]
array([ 3, -3, -1])
```

Remarque : Plus généralement, la syntaxe suivante permet d'extraire un sous-tableau :

```
tableau[i:j, k:l]
```

Cela renvoie le sous-tableau composé des lignes d'indice `i` inclus à `j` exclus et des colonnes d'indice `k` inclus à `l` exclus (le mécanisme est le même que celui du slicing des listes).

Exemple :

```
>>> T1[0:2, 1:3]
array([[ 2,  3],
       [ 2, -3]])
```

c) Nombre de lignes, nombre de colonnes

La fonction `np.shape(tableau)` renvoie un tuple `(nombre de lignes, nombre de colonnes)`. On peut les récupérer tous les deux simultanément ou bien un par un :

```
n, p = np.shape(tableau)    # n = nbr de lignes, p = nbr de colonnes

taille = np.shape(tableau)
n = taille[0]
p = taille[1]
```

Exemple :

```
>>> n, p = np.shape(T1)
>>> n
3
>>> p
4
>>> taille = np.shape(T2)
>>> taille
(2, 4)
```

```
>>> taille[0]
2
>>> taille[1]
4
```

d) Parcours des éléments du tableau

Pour parcourir tous les éléments du tableau, il faut faire une boucle imbriquée :

```
n, p = np.shape(tableau)
for i in range(n):
    for j in range(p):
        print(tableau[i, j])
```

Exemple :

Code :

```
n, p = np.shape(T1)
for i in range(n):
    for j in range(p):
        print(T1[i, j])
```

Résultat :

```
1
2
3
4
-1
2
-3
4
0
1
-1
2
```

3) Calculs matriciels

Une matrice peut ainsi être représentée :

- ⋆ soit par une liste de listes de nombres
- ⋆ soit par un tableau `numpy` à deux dimensions

Dans cette partie, nous allons utiliser la représentation des matrices par des tableaux `numpy` à deux dimension et nous allons écrire des fonctions pour calculer :

- ⋆ la somme de deux matrices
- ⋆ le produit de deux matrices
- ⋆ la transposée d'une matrice

```
1  def somme(M1, M2):
2      """
3      Entrées : M1 et M2 deux tableaux numpy à deux dimensions
4      Sortie : Somme matricielle M1 + M2
5      """
6      assert np.shape(M1) == np.shape(M2), "Tailles des matrices incompatibles"
7      n, p = np.shape(M1)
8      S = np.zeros((n, p))    # On initialise le résultat par la matrice nulle
9      for i in range(n):
10         for j in range(p):
11             S[i, j] = M1[i, j] + M2[i, j]
12     return S
13
```

```python
def produit(M1, M2):
    """
    Entrées : M1 et M2 deux tableaux numpy à deux dimensions
    Sortie : Produit matriciel M1 x M2
    """
    n, p = np.shape(M1)
    q, r = np.shape(M2)
    assert p == q, "Tailles des matrices incompatibles"
    P = np.zeros((n, r))    # On initialise le résultat par la matrice nulle
    for i in range(n):
        for j in range(r):
            c = 0
            for k in range(p):
                c += M1[i, k] * M2[k, j]
            P[i, j] = c
    return P

def transposee(M):
    """
    Entrée : M tableau numpy à deux dimensions
    Sortie : Transposée matricielle de M
    """
    n, p = np.shape(M)
    T = np.zeros((p, n))    # On initialise le résultat par la matrice nulle
    for i in range(n):
        for j in range(p):
            T[j, i] = M[i, j]
    return T
```

Remarque : Les fonctions que l'on a créées existent déjà dans le module `numpy` soit en tant que fonction ou méthode, soit en tant qu'opération. Ainsi, étant donnés trois tableaux `numpy` à deux dimensions `M1`, `M2` et `M`, on a :

```python
M1 + M2             # Somme matricielle de M1 et M2
np.dot(M1, M2)      # Produit matriciel de M1 et M2
M1.dot(M2)          # Également produit matriciel de M1 et M2
np.transpose(M)     # Transposée matricielle de M
```

Attention : la syntaxe `M1 * M2` existe aussi mais ne correspond pas au produit matriciel. C'est le produit terme à terme.