Numérique et sciences informatiques
Classe de première

D Algorithmique
1 Parcours séquentiel d'un tableau

Dans votre dossier « Programmation »

* créez un dossier « Tableaux »
* Dans ce dossier, créez un fichier « ManipTableaux.py »
* Dans ce fichier écrire l'instruction d'importation de la fonction `randint` depuis le module `random` :

```
from random import randint
```

I Quelques fonctions pour la manipulation d'un tableau

I.1 Créer un tableau comportant un nombre donné d'entiers choisis aléatoirement

Nous allons créer une fonction `tabAlea(n, a, b)` qui renvoie un tableau comportant un nombre `n` d'entiers choisis aléatoirement entre deux entiers `a` et `b` compris. Pour cela, nous pouvons utiliser la technique de la compréhension de liste :

```
def tabAlea(n, a, b) :
    return [randint(a, b) for i in range(n)]
```

I.2 Le mot clé `in`

• Pour savoir si une valeur donnée `val` se trouve dans une variable `tab` de type **`list`** donnée on peut utiliser le mot clé **`in`** selon la syntaxe suivante :

```
val in tab
```

Cette expression renvoie :

* **`True`** si la valeur de la variable `val` fait partie des valeurs du tableau ;
* **`False`** sinon.

• On peut aussi utiliser le mot-clé **`in`** de la même façon :

```
val in iter
```

si la variable `iter` est d'un type 'itérable' (c'est-à-dire un type que l'on peut « parcourir »). Les types 'itérables' que nous avons rencontrés jusqu'ici sont les type **`str`**, **`list`** et **`tuple`**. Nous en verrons d'autres par la suite. Dans le cas où la variable `iter` est de type **`str`**, `val` doit aussi être de type **`str`** sinon l'expression générera un message d'erreur.

• *Exemples :*

```
>>> '89' in "7895"
True
>>> '79' in "7895"
False
>>> 5 in 7895
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    5 in 7895
TypeError: argument of type 'int' is not iterable
>>> 5 in "7895"
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    5 in "7895"
TypeError: 'in <string>' requires string as left operand, not int
```

I.3 Rechercher une valeur dans un tableau

Nous allons écrire une fonction `chercher(val, tab)` qui permet de déterminer le plus petit indice où apparaît la valeur `val` dans un tableau `tab` *si cette valeur est dans le tableau*. Si la valeur `val` ne se trouve pas dans le tableau `tab`, la fonction renverra `None`.

Pour obtenir ce résultat, le programme consiste à parcourir un à un tous les éléments du tableau et renvoyer l'indice correspondant dès que l'élément choisi possède la valeur recherchée. Ici, on parcourra les indices dans le sens croissant donc la fonction renverra le plus petit indice possible lorsque la valeur recherchée se trouve dans le tableau.

```
def chercher(val, tab) :
    for i in range(len(tab)) :
        if tab[i] == val :
            return i
    return None
```

On peut remarquer que ce programme traite aussi correctement le cas d'un tableau vide.

I.4 Rechercher le nombre d'occurrences d'une valeur dans un tableau

Pour déterminer le nombre d'apparitions (on parle de nombre d' « occurrences ») d'une valeur `val` dans un tableau `tab` on peut écrire la fonction `nbOccur(tab, val)` suivante :

```
def nbOccur(tab, val) :
    cpt = 0
    for e in tab :
        if e == val :
            cpt += 1
    return cpt
```

On utilise ici un compteur `cpt` qui s'incrémente de 1 à chaque fois que l'on rencontre dans le tableau une occurrence de la valeur recherchée.

I.5 Déterminer la moyenne des éléments d'un tableau

On suppose que le tableau à étudier ne contient que des nombres. On peut alors rechercher la moyenne des nombres contenus dans ce tableau. Pour cela on peut écrire la fonction `moyenneTab(tab)` ci-dessous. Cette fonction renvoie `None` si le tableau est vide.

```
def moyenneTab(tab) :
    lg = len(tab)
    if lg == 0 :
        return None
    s = 0
    for e in tab :
        s += e
    return s/lg
```

I.6 Déterminer le minimum ou le maximum d'un tableau

On veut créer une fonction `minTab(tab)` qui prend en argument un tableau de nombres `tab` et qui renvoie un tuple à deux places dont les éléments sont :

* le plus petit indice associé à la valeur minimum du tableau ;
* cette valeur minimum.

On peut écrire le programme suivant :

```
def minTab(tab) :
```

```
    indMin = 0
    mini = tab[0]
    for i in range(1, len(tab)) :
        if tab[i] < mini :
            indMin = i
            minimum = t[i]
    return (indMin, minimum)
```

A faire vous-même

1. Recopiez et testez cette fonction.
2. Écrire la fonction `maxTab(tab)` qui renvoie sous la forme d'un tuple *l'indice et la valeur maximum* d'un tableau.
3. Essayer ces deux fonctions avec le tableau suivant qui ne contient que des valeurs de type **`str`** :
`semaine=['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']`
Que constate-t-on et pourquoi ?
4. Écrire une autre fonction `minTab2(tab, g, d)` qui comporte trois arguments suivants :

* `tab` désigne le tableau à étudier ;
* `g` désigne l'indice à partir duquel on commence à rechercher le minimum ;
* `d` désigne l'indice jusqu'auquel on arrête de rechercher le minimum.

Il s'agit donc d'obtenir un tuple à deux places qui donnent les mêmes informations pour le *minimum local* relativement à la tranche du tableau `tab` compris entre l'indice `g` et l'indice `d`.

II. La notion de correction et de coût d'un algorithme

II.1 Correction et coût d'un algorithme

Lorsque l'on écrit un algorithme, on doit avant tout se demander s'il résout bien le problème que l'on s'est posé. On dit que l'on se pose la question de **la correction** de l'algorithme. Pour chaque algorithme, on doit être capable de produire un raisonnement qui prouve que le résultat obtenu par l'exécution du programme correspond bien à ce qui est attendu.

Un autre problème consiste à estimer le nombre d'opération effectuées, et donc le temps d'exécution du programme *en fonction de* la taille des données de départ. Cette estimation est indépendante des caractéristiques de l'ordinateur sur lequel est le programme est exécuté et du moment où le programme est exécuté. On dit que l'on cherche à évaluer **le coût** ou **la complexité** de l'algorithme.

II.2 La complexité des algorithmes présentés

Pour les quatre fonctions que nous venons de voir, il est facile de voir que le nombre d'opérations effectuées et donc le temps d'exécution sont *proportionnels à la taille du tableau traité*. On dit que la complexité de ces algorithmes est **linéaire** et qu'elle est en $O(n)$.

Il existe cependant une différence entre l'algorithme de recherche (présenté en I.3) et les trois suivants : le premier peut s'achever très vite si on trouve la valeur recherchée dès le le début du tableau alors qu'il est nécessaire de parcourir tout le tableau pour achever les trois suivants. Dans le premier cas, le parcours complet du tableau n'est nécessaire que *dans le pire des cas* : c'est-à-dire le cas où l'élément recherché se trouve à la fin du tableau ou même ne s'y trouve pas du tout.