TP n° 2 – Structures de données séquentielles et applications

D'après des TP de Frédéric Junier et des documents eduscol

1 Présentation sommaire des structures séquentielles

1.1 Quelques définitions

> Mémo – **Objets itérables, indexables**
> **Un (objet) itérable** est un objet dont on peut parcourir les valeurs, à l'aide d'une boucle `for` par exemple.
> **Un (objet) indexable** est un objet qui implémente la méthode `__getitem__()`. En d'autres termes, il décrit des objets qui sont des "conteneurs", ce qui signifie qu'ils contiennent d'autres objets.
> **En python, cela inclut les listes, les tuples, les chaînes de caractère et les dictionnaires**.

1.2 Tableau

1.2.1 Qu'est-ce qu'un tableau ?

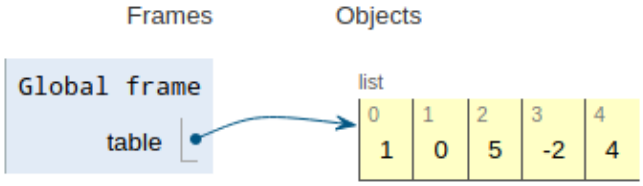
Un tableau est un type de donnée permettant de stocker plusieurs valeurs de manière séquentielle et d'y accéder à l'aide d'une seule variable.

La particularité d'une structure de tableau est que le contenu de la i-ème case peut être lu ou modifié en temps constant, c'est-à-dire indépendant de i.

Le moyen le plus simple de créer un tableau en python est d'énumérer l'ensemble de ses valeurs (séparées par des virgules) :

```
>>> table = [1, 0, 5, -2, 4,]
>>> table
[1, 0, 5, -2, 4]
```

On vient ici de créer une variable table qui est un tableau (toute énumération de valeurs entre crochets `[ ]` sera interprétée comme un tableau par python).
Voyez ce qu'il se passe en sous-main à l'aide de Python tutor :



Une zone de mémoire a été réservée pour stocker les 5 données de notre tableau. Cette zone est pointée par la variable `table`. Cette représentation est extrêmement importante : en soit, une variable ne contient pas de valeur en python, elle est une référence (on dit un **pointeur** en informatique) vers une zone de la mémoire dans laquelle la ou les valeurs sont stockées. *Remarque :* Cela aura une grande importance lorsque ce réseau de pointeurs sera plus compliqué.

Le langage python autorise à stocker dans un tableau des données de type quelconque, sans aucune cohérence : on peut mettre des entiers, des nombres à virgule flottante, des chaînes de caractères, ... dans un même tableau :

```
["pomme", 12, 3.14, "pêche"]
```

1.2.2 Indiçage des listes

Indiçage positif : les éléments d'un tableau sont indexés à partir de 0 : le premier élément aura pour indice 0, le second aura pour indice 1, etc.

Attention : les indices d'un tableau de n éléments commencent à 0 et se terminent à $n-1$.

Pour accéder à la valeur d'indice `i` du tableau `table`, on utilise la syntaxe `table[i]`.

Indiçage négatif : les éléments d'un tableau sont **également** indexés à partir de -1 : le **dernier** élément aura pour indice -1, l'avant-dernier aura pour indice -2, etc.

Les indices d'un tableau de n éléments commencent donc à -1 et se terminent à $-n$.

```
>>> t = ["pomme", 12, 3.14, "pêche"]
>>> t[1]
12
>>> t[-1]
'pêche'
>>>> t[-4]
'pomme'
```

1.2.3 Concaténation

Les tableaux supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication :

```
>>> ani1 = ['girafe','tigre']
>>> ani2 = ['singe','souris']
>>> ani1 + ani2
['girafe','tigre','singe','souris']
>>> ani1 * 3
['girafe','tigre', 'girafe','tigre', 'girafe','tigre']
```

Petite subtilité : l'opération de concaténation renvoie un nouveau tableau.

1.2.4 Slicing (ou coupe)

Un autre avantage des tableaux est la possibilité d'en « sélectionner » une partie. On dit alors qu'on récupère une **coupe** (slicing) du tableau.

La syntaxe `t[d:f:p]` permet ainsi de récupérer les k éléments du tableau d'indice d à f exclus par pas de p, avec k vérifiant $d + (k-1)p < f \leqslant d + kp$.

Lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole « : », Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:2]
[0, 1]
>>> x[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1:6:3]
[1,4]
```

1.2.5 Modifier le contenu d'un tableau

En python, un tableau n'est pas une structure figée en mémoire. On peut notamment :

— modifier une des valeurs stockée dans le tableau (ce qui revient en général à la remplacer par une autre) ;
— ajouter ou supprimer des éléments, à la fin, au début, voire même au milieu d'un tableau.

Pour modifier un élément dans un tableau, on utilise la syntaxe suivante :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0] = 'lion'
>>> animaux
['lion', 'tigre', 'singe', 'souris']
```

1.2.6 Les méthodes append() et pop()

Une **méthode** peut-être définie comme une fonction qui « appartient à » un objet. En l'occurrence, l'objet est un tableau.

La méthode `append()` permet d'ajouter un élément à la fin de la liste.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux.append('lion')
>>> animaux
['girafe', 'tigre', 'singe', 'souris', 'lion']
```

La méthode `pop()` (sans argument) supprime l'élément de plus grand indice (le dernier élément du tableau), et renvoie sa valeur.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux.pop()
'souris'
>>> animaux
['girafe', 'tigre', 'singe']
```

Attention, `pop()` utilisé sur un tableau vide `[]` renverra une erreur puisqu'il n'y a pas d'élément à supprimer.

1.3 Tuples (n-uplets) et chaînes de caractère

1.3.1 Définitions

Un **tuple** est une liste qui ne peut plus être modifiée. Pour créer un tuple, la syntaxe suivante peut-être utilisée : `mon_tuple = (1, "ok", "olivier")`. Les parenthèses ne sont pas obligatoires mais facilitent la lisibilité du code.
Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières. Elles peuvent être définies de la façon suivante : `animaux = "girafe tigre"`

1.3.2 Points communs et différences

Les tableaux, tuples et chaînes de caractères ont des points communs : elles sont indicées de la même façon, il est possible d'accéder aux données de la même façon (lecture d'un élément, slicing, etc.). À l'inverse :

- **Les tuples et les chaînes de caractères ne sont pas modifiables**.
- Les méthodes `append` et `pop` n'agissent que sur les listes.

1.4 Dictionnaires

1.4.1 Description du type

Décrire un type revient essentiellement à donner les opérations permises, voici donc les opérations permises par un dictionnaire. Nous fixons deux ensembles C et V . L'ensemble C sera appelé l'ensemble des « clefs » et V l'ensemble des « valeurs ». Un dictionnaire permet d'associer à un élément de C un élément de V. Autrement dit, un dictionnaire est une fonction (au sens mathématique) de C dans V .

Selon le point de vue adopté ici, une clef peut ne pas avoir de valeur associée, autrement dit un dictionnaire n'est pas forcément une application de C vers V.

Les opérations permises par le type des dictionnaires sont :

- Créer un dictionnaire vide.
- Ajouter une association : étant donné $c \in C$ et $v \in V$, décider que v sera associée à c. Si une valeur était déjà associée à c, la nouvelle écrase l'ancienne.

— Lire la valeur associée à une clef : étant donné $c \in C$, renvoyer la valeur associée à c. Si aucune valeur n'est associée à c, une erreur est déclenchée.

De plus, un dictionnaire doit permettre de réaliser ces trois opérations de base efficacement. Selon les implémentations, le temps d'exécution sera en $O(\log(n))$, où n est le nombre de données enregistrées, voire en $O(1)$. À ces trois opérations, on rajoute souvent une opération permettant de supprimer une entrée, et une permettant de savoir si une clef est présente dans un dictionnaire. En outre, on peut aussi ajouter une opération permettant de renvoyer toutes les clefs d'un dictionnaire, ce qui permettra de parcourir toutes les données du dictionnaire. Ceci dit, si le but principal est de parcourir toutes les données enregistrées, un simple tableau ferait tout aussi bien l'affaire. L'intérêt du dictionnaire est de pouvoir trouver (ou tester la présence) une clef précise très rapidement.

1.4.2 En Python

Le type des dictionnaires est présent nativement dans Python. Voici la syntaxe des opérations de base ci-dessus, où l'on a au besoin $c \in C$ et $v \in V$:

- Créer un dictionnaire vide : `d = {}`.
- Associer `v` à la clé `c` : `d[c] = v`.
- Renvoyer la valeur associée à la clé `c` : `d[c]`.
- Voir si `c` est une clef de `d` : `c in d`.
- Supprimer l'entrée correspondant à `c` dans `d` : `del d[c]`. Déclenche une erreur si la clef `c` n'est pas présente dans `d`. On peut trouver cette syntaxe troublante... Un `del(d, c)` aurait été plus clair. On peut préférer utiliser `d.pop(c)` qui en plus renvoie l'élément supprimé.
- La méthode `keys()` renvoie l'ensemble des clefs de `d`. On peut donc parcourir un dictionnaire à l'aide d'une boucle de la forme `for c in d.keys():`.

> Mémo – **Structures séquentielles**
>
> - Les itérables indexables de type `list`, `tuple` ou `str` se parcourent soit avec ses indices, soit par éléments, soit les deux combinés.
> - Un dictionnaire (dernier type d'itérable indexable) se parcourt soit avec ses clés, soit avec ses valeurs, soit les deux combinées.
> - Une coupe des éléments de `L` d'indices `a` à `b` se note `L[a:b+1]`.
> - La méthode `append()` permet d'ajouter un élément.
> - La méthode `pop()` agit sur place et renvoie la valeur supprimée.
> - L'opération `=` recopie les adresses des pointeurs, pas les éléments.

2 Recherche séquentielle dans un tableau unidimensionnel et variations

> Mémo – **Tableau (unidimensionnel)**
> Dans la suite, un tableau unidimensionnel désignera, pour simplifier une séquence 1D de type `list`', `tuple` ou `str`.
> Les méthodes `append()` et `pop()` n'agissent que sur les listes.

Exercice 2.1 (*Des tableaux/listes-jouets*)

1. Créer les listes suivantes, qui pourront servir dans la suite du TP pour vérifier les fonctions qui auront été écrites :

```
t0 = [42] * 15                # À éviter
t00 = [42 for i in range(15)]  # À privilégier
t000 = [42 for _ in range(15)] # Possible également
t1 = [10, 30, 42, 2, 17, 5, 30, -20]
t2 = []
for i in range(-3, 6):
    t2.append(i**2)
t21 = [i**2 for i in range(-3,6)]
```

```
t3 = []
for i in range(1000):
    if (i%5) in [0, 2, 4]:
        t3.append(i**3)
t31 = [i**3 for i in range(1000) if (i%5) in [0,2,4]]

t4 = [843.0]
for i in range(20):
    t4.append(t4[i]/3+28)
```

Ces définitions seront écrites dans le fichier de script. Après exécution (F5), on vérifiera les valeurs de ces listes dans l'interpréteur et/ou l'explorateur de variables.

2. Quelles sont les longueurs de ces listes ?

> MÉMO – **Longueur**
> La fonction `len()` renvoie le nombre des éléments (ou la longueur) d'un objet. En cas d'absence d'argument ou d'argument invalide, l'exception « TypeError » est générée.

Exercice 2.2

1. Écrire une fonction testant l'appartenance d'un objet à un tableau. Proposer un code avec une boucle `for` et un autre avec une boucle `while`.

```
def appartient(x, t):
    ....

>>> appartient(42, [5, 12, 17, 42,])
True
>>> appartient(24, [5, 12, 17, 42])
False
```

2. Quel est, dans le pire des cas (à préciser !), le nombre maximum d'accès au tableau effectués pour tester l'appartenance de x à T ? *(Par accès au tableau, on entend : consultation/modification via `... = t[i]` ou `t[i] = ...` ou itération d'une boucle de la forme `for y in t: ...`)*

> MÉMO – **Parcours d'un tableau**
>
> avec ses indices :
> ```
> for i in range(len(L)):
>     print(L[i])
> ```
> par éléments :
> ```
> for e in L:
>     print(e)
> ```
> les deux simultanément :
> ```
> for i,e in enumerate(L):
>     print(i,e) # e=L[i]
> ```

Exercice 2.3

Écrire une fonction renvoyant la liste (éventuellement vide) des positions d'un objet dans un tableau.

```
def positions(x, t):
    ...

>>> positions(42, [12, 17, 42, 5])
[2]
>>> positions(24, [12, 17, 42, 5])
[]
>>> positions(42, [42, 12, 17, 42, 5])
[0, 3]
```

Exercice 2.4

1. Pour un tableau **dont les éléments sont des nombres entiers**, écrire une fonction revoyant le *maximum* des éléments de ce tableau. De même, écrire une fonction renvoyant la *position* de ce maximum.

```
>>> maximum(t1), position_maximum(t1)
42, 2
```

2. Que se passe-t-il lorsque le maximum est pris plusieurs fois ? Comment changer ce comportement ? Conclure quand à l'importance de la précision des énoncés et des mots utilisés : *la* et *une* ont des sens différents...

Exercice 2.5

1. Écrire une fonction prenant en entrée un tableau, et renvoyant le tableau des sommes cumulées (depuis le premier terme) :

```
>>> sommes_cumulees(t2)
[9, 13, 14, 14, 15, 19, 28, 44, 69]
```

2. Évaluer le nombre d'additions réalisées, en fonction de la longueur n du tableau. Si ce nombre est de l'ordre de n^2, essayer de réécrire la fonction pour arriver à un nombre de l'ordre de n.

Exercice 2.6

1. Écrire une fonction prenant en entrée un tableau possédant au moins deux éléments, et renvoyant les deux plus grands éléments de ce tableau.

```
>>> deux_grands(t1)
42, 30
```

2. Cette fonction était mal spécifiée : que se passe-t-il si le plus grand élément est présent deux fois ? Réécrire la fonction pour que le comportement soit différent !

3 Listes et dictionnaires

Exercice 2.7 (*Dépouillement d'une urne*)

À l'issue d'une élection à scrutin uninominal, on récupère un tableau contenant tous les noms inscrits sur les bulletins trouvés dans l'urne. Par exemple :
`urne = ["Maurice", "Roger", "Maurice", "Marie", "Marie", "Jeanne", ...]`.
Le but de cet exercice est de déterminer le vainqueur de l'élection, en un seul parcours de l'urne. Le plus pratique est d'utiliser un dictionnaire qui à chaque nom associera son nombre de voix. Notons qu'un des avantages d'un dictionnaire est qu'il n'y a pas besoin de savoir à l'avance qui sont les candidats, ni même combien il y en a.

MÉMO – **Parcours d'un dictionnaire**

avec ses clés :

```
for cle in D:
    print(cle)
```

par ses valeurs :

```
for v in D.values():
    print(v)
```

les deux simultanément :

```
for cle,v in D.items():
    print(cle,v)
```

1. Importer la fonction `urne` du fichier `Info-TP02_cadeau.py` qui permet de générer le contenu d'une urne de taille variable, avec un nombre de candidats aléatoires.
2. Écrire une fonction `depouillement(urne)` qui fournit le résultat du dépouillement, c'est-à-dire le nombre de voix pour chaque candidat.
3. Écrire une fonction permettant de déterminer le vainqueur de l'élection.

4 Algorithmes opérant par boucles imbriquées

Exercice 2.8
Écrire une fonction prenant en entrée un tableau t non vide, dont les éléments sont des nombres entiers, renvoyant un couple (i, j) tel que $i > j$ et $|t_i - t_j|$ est minimal. Quel est, dans le pire des cas (à préciser !), le nombre maximum d'accès au tableau effectués ?

```
def proches(t):
    ....

>>> t1
[10, 30, 42, 2, 17, 5, 30, -20]
>>> proches(t1)
(6, 1)
```

Exercice 2.9 (*Recherche d'une sous-liste ou d'un sous-mot*)
La liste $[1, 3, 5]$ peut être vue comme une sous-liste de la liste $[12, -15, 1, 3, 5, 19, 23]$ (ici, par sous-liste, on entend « en un seul morceau » : $[1, 3, 5]$ n'est pas vue comme une sous-liste de $[1, 2, 3, 4, 5]$). De même le mot `tag` est un sous-mot de `pouettagada`.
On cherche, dans cet exercice, à déterminer si une liste t_1 (respectivement un mot m_1) est une sous-liste (respectivement un sous-mot) d'une liste t_2 (respectivement un mot m_2). Les manipulations de listes et de chaînes de caractères étant très proches [1], les programmes doivent normalement fonctionner indifféremment sur les listes et les chaînes de caractères.
Si on reprend l'exemple `m1 = "tag"` et `m2 = "pouettagada"`, alors `m2[5:8] == m1`. Une façon assez simple de répondre au problème serait alors :

Entrées : m_1, m_2
$lg_1, lg_2 \leftarrow |m_1|, |m_2|$ # les longueurs
pour d *allant de* 0 *à* $lg_2 - lg_1$ **faire**
  si $m_2[d : d + lg_1] = m_1$ **alors**
    Résultat : `True`
Résultat : `False`

Dans un premier temps, on va s'interdire le *slicing* : seules les comparaisons « lettre-à-lettre » sont autorisées.

1. Écrire une fonction `ss_mot` prenant en entrée deux chaînes de caractères et retournant `True` ou `False` selon que la première est ou n'est pas un sous-mot de la seconde.
2. Tester cette fonction judicieusement (ce qui doit inclure au moins 4 cas : absence, présence au bord gauche, au bord droit, et au milieu), sur des listes ou chaînes de caractère.
3. Vérifier que cette fonction permet également de déterminer si une liste t_1 est une sous-liste d'une liste t_2.
4. En s'appuyant sur la fonction `ss_mot`, écrire une nouvelle fonction prenant en entrée deux chaînes de caractères et retournant la liste (éventuellement vide) des positions dans m_2 où on trouve le mot m_1. La fonction Tester sur différents exemples.

   ```
   >>> positions_sous_mot("tag", "pouettagada")
   [5]
   >>> positions_sous_mot("plouf", "pouettagada")
   []
   >>> positions_sous_mot("ta", "taratata")
   [0, 4, 6]
   ```

5. Réécrire les fonctions précédentes en utilisant le *slicing* (une coupe des éléments d'une liste ou d'une chaîne de caractères)

1. Indexation, longueur via `len`, slicing...

Exercice 2.10 (*Une petite application de la recherche d'un sous-mot...*)

1. Écrire une fonction `chaine_aleatoire` prenant en entrée un entier naturel n et retournant une chaîne aléatoire de longueur n sur l'alphabet $\{A, C, G, T\}$. La tester.
   Indication : on pourra utiliser la fonction `randint()` du module `random`[2].
2. Créer une première chaîne aléatoire m_1 de longueur 5 (avec uniquement les lettres $\{A, C, G, T\}$) et une deuxième de même type mais de longueur 10^4. Déterminer le nombre de positions de m_2 auxquelles on trouve m_1. Faire la moyenne sur une centaine d'exemples. Commenter.

* *
*

2. Pour la syntaxe détaillée, voir par exemple : https://www.w3schools.com/python/ref_random_randint.asp