Chapitre II

Structures de données en Python

Table des matières

Partie A : Introduction **2**
1. Définitions . . . 2
2. Les tableaux et les listes chaînées . . . 2
   a) Les tableaux . . . 2
   b) Les listes chaînées . . . 3

Partie B : Les listes en Python **4**
1. Définir une liste Python et accéder à ses éléments . . . 4
   a) Définir une liste . . . 4
   b) Accéder aux éléments d'une liste . . . 5
   c) Exercices . . . 6
2. Manipulation et opérations sur les listes . . . 7
   a) Concaténer des listes . . . 7
   b) Modifier un élément d'une liste . . . 7
   c) Copier une liste . . . 7
   d) Supprimer un élément . . . 8
   e) Insertion d'un élément dans une liste . . . 9
   f) Autres méthodes . . . 9
3. La mise en pratique ! . . . 12

Partie C : Matrices et array Numpy **19**
1. Les matrices ou tableaux multi-dimensionnels . . . 19
   a) Création et manipulation d'une matrice . . . 19
2. Les array Numpy . . . 21
   a) Création d'un array . . . 21
   b) Manipulation d'un array . . . 22
   c) Exercices de création d'arrays . . . 23
   d) Opérations sur les array . . . 25
   e) On passe à la pratique ! . . . 26

Partie D : Autres structures de données en Python **29**
1. Les tuples . . . 29
2. Les dictionnaires . . . 30
   a) Créer un dictionnaire . . . 31
   b) Accéder aux éléments d'un dictionnaire . . . 31
   c) Exemple d'utilisation d'un dictionnaire . . . 32

Partie A

Introduction

Lors de l'étude de la rapidité d'exécution d'un algorithme (il s'agit de la théorie de la *complexité temporelle*), les calculs se basent sur certaines opérations élémentaires que l'on s'est fixées, le plus souvent sur le choix des opérations élémentaires les plus présentes ou les plus représentatives de l'algorithme. Il est bien-sûr légitime de se demander si certaines opérations élémentaires sont plus coûteuses en temps ou en mémoire que d'autres. De même, lorsqu'on stocke en mémoire un ensemble cohérent de données (comme une liste en Python par exemple) l'accès à une donnée ou l'écriture d'une donnée dans cet ensemble a un coût différent selon sa "situation" dans l'ensemble.

Ainsi, on va étudier la notion informatique de **structure de données** qui permet de spécifier, selon les besoins du programmeur, le coût de chaque opérations élémentaires sur l'ensemble des données de la structure telle que l'accès ou l'écriture d'une donnée.

1. Définitions

En informatique, une **structure de données** est une façon de représenter en mémoire un ensemble de données en spécifiant :

— la manière d'attribuer une certaine quantité de mémoire à cette structure ;

— la manière d'accéder à ce qu'elle contient.

Plus précisément, on dit que :

— La structure de données est **statique** si la quantité de mémoire attribuée à la structure lors de sa création est fixe et ne peut être modifiée (par exemple, la classe `str` ou `tuple` en Python).

— Dans le cas contraire, on dit que la structure de données est **dynamique** (par exemple, la classe `list` en Python) ;

De plus, une structure de données est dite **mutable** si on peut modifier les données contenues dans la structure après sa création : les classes `tuple` et `str` ne sont pas mutable en Python tandis que la classe `list` l'est.

Nous verrons dans la suites exactement à quoi coorexpondent les classes `list` et `tuple`

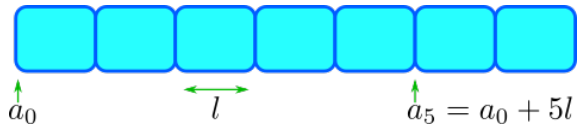
Voici quelques exemples de structures de données usuelles en informatique :

- Les structures de données linéaires représentent les données sous forme de suites finies : chaque élément dans une structure linéaire, à part le dernier, possède un successeur ; il existe plusieurs types de structures linéaires dont les listes, les tableaux, les piles et les files
- les tableaux multidimensionnels (matrices) ;
- Les structures d'arbres (penser à la structure arborescente des fichiers en mémoire) ;
- Les structures de graphes ou structures relationnelles (penser aux bases de données).

2. Les tableaux et les listes chaînées

a. Les tableaux

Un tableau est une structure de données linéaires dans laquelle on stocke une suite de données de même type à des emplacements mémoire consécutifs. Du fait que les données partagent le même type, chaque emplacement dans la tableau a une taille constante l.

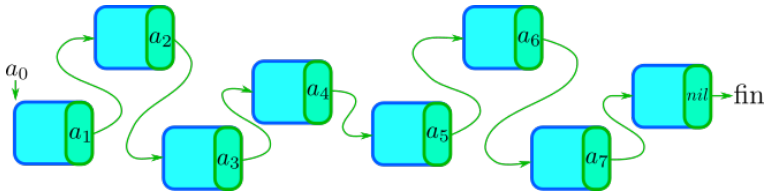


Ainsi, à la création du tableau, l'adresse a_0 du début de l'emplacement d'indice 0 est fixé, puis on accède à l'adresse de l'emplacement d'indice k en calculant $a_k = a_0 + kl$. L'accès à un emplacement se fait alors en temps constant : on obtient toujours l'adresse d'un emplacement du tableau en 2 opérations élémentaires (une addition, une multiplication).

Le désavantage de cette structure est qu'elle est statique : on ne peut pas ajouter d'emplacement à la fin du tableau car l'espace mémoire après le dernier emplacement n'est pas forcément libre.

b. Les listes chaînées

Une liste chaînée est une structure de données linéaires dans laquelle on stocke une suite de données de même type, et à chaque emplacement d'une donnée, on associe un pointeur contenant l'adresse en mémoire de l'emplacement suivant.



Contrairement à la structure de tableau, on ne peut pas connaître à l'avance l'adresse de l'emplacement de donnée d'indice k d'une liste chaînée. Pour l'obtenir, il faut parcourir les k emplacements précédents en partant de l'emplacement d'indice 0. Ainsi, l'accès à un emplacement dans une liste chaînée se fait en temps linéaire : en effet, il faut faire k opérations élémentaires pour obtenir l'adresse de l'emplacement d'indice k (k lectures).

L'avantage de cette structure par rapport à celle de tableau est qu'elle est dynamique : on peut ajouter un emplacement supplémentaire en modifiant un pointeur !

Partie B

Les listes en Python

Malgré son nom, la classe `list` n'est pas une liste chaînée : il s'agit d'une structure de données linéaire plus élaborée qui permet de combiner les avantages des deux classes décrites plus haut. À savoir que cette structure est dynamique et que le temps d'accès aux emplacements est constant. Un autre de ses avantages est qu'on peut stocker différents types de données parmi les emplacements d'une même liste.

1. Définir une liste Python et accéder à ses éléments

a. Définir une liste

Une liste (de type `list`) en Python est donc une suite d'objets de type quelconque (par exemple `int`, `complex`, `str` ou même `list` !).

Pour définir un tel objet, on délimite la suite d'objets par des crochets ouvrant `[` et fermant `]` et on sépare chacun des objets par une virgule `,`.

Essayons sur un exemple :

```
>>>L=[3,12,'coucou',2+5j,6.2,'liste',8]
```

Comme dit précédemment, une liste peut contenir des listes ! Au passage, on peut définir une liste ne contenant rien : il suffit d'indiquer la fermeture immédiatement après l'ouverture de la liste.

```
>>>M=[[1,1],[3],[3.1,3.4]] #Une liste de listes !

>>>N=[] #Une liste vide
```

On peut appeler les variables `L`,`M` et `N` pour voir que Python comprend bien les listes comme on lui transmet :

```
>>>L
>>>M
>>>N
```

On peut également définir une liste par compréhension de la même façon que les ensembles en mathématiques (enfin presque) : voici des exemples d'ensembles définis en compréhension et l'instruction qui permet de définir le liste qui leur correspond.

Définir une liste par compréhension :

- $\{k^2 \mid k \in [\![0,5]\!]\}$ $\rightsquigarrow$ `[k**2 for k in range(6)]`
- $\{t \in [\![-6,6]\!] \mid t^4 \leq 70\}$ $\rightsquigarrow$ `[t for t in range(-6,7) if t**4<=70]`

Exercice 1.

Donner les instructions permettant de définir les listes "correspondants" aux ensembles suivants :

1. $\{k + \frac{1}{k} \mid k \in [\![1, 50]\!]\}$
2. $\{n \in [\![0, 100]\!] \mid 5 \text{ divise } n^3 - 2n - 1\}$
3. $\{1, 2, 4, 8, ..., 1024, 2048, 4096, 8192, 16384\}$.
4. $2\pi\mathbb{Z} \cap [-10, 100]$.
5. $[\![-50, 0]\!] \cup [\![50, 100]\!]$.

Correction.

1. `[k+1/k for k in range(1,51)]`
2. `[n for t in range(101) if (n**3-2*n-1)%5==0]`
3. `[2**i for i in range(15)]`
4. `[2*k*pi for t in range(-1000,1000) if 2*k*pi>=-10 and 2*k*pi<=100]`
5. `[m for m in range(-50,100) if n<=0 or n>=50]`

b. Accéder aux éléments d'une liste

Pour accéder aux éléments, on utilise leurs indice que l'ont désigne, comme on l'a vu pour les chaînes de caractères, immédiatement après le nom de la liste et entouré par des crochets. **ATTENTION :** Le **1er** élément d'une liste a pour indice **0**, le 2eme $\rightsquigarrow$ 1, etc...

Faisons un essai avec les listes qu'on a définies précédemment :

```
>>>L[0]
>>>L[2]
>>>M[2]
>>>M[2][0]
```

Les autres techniques que l'on a apprises pour l'accès aux chaînes de caractères sont aussi valables pour les listes : accès par indices négatifs, slicing et longueur de liste via la fonction `len`

```
>>>L[-2] #Renvoie l'avant dernier élément de L
>>>len(L) #Renvoie la longueur de L
>>>L[2:4] #Renvoie les éléments d'indice 2 à !! 3 !! (=4-1)
>>>M[2][0]
```

On peut alors parcourir une liste grâce à une boucle `for`. Il faut bien noter que `range(len(L))` énumère tous les indices de la liste `L` : les nombres de `0` à `len(L)-1`.
On rappelle bien sûr que `range(k)` est une énumération des nombres de 0 à $\mathbf{k-1}$.
Ici, on **affiche dans le shell** les éléments de `L` un par un :

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

Une liste étant un objet énumérable pour Python (comme les chaînes de caractères), on peut la parcourir directement :

```
>>>for element in L:
...    print(element)
```

Dernier point de ce paragraphe, on peut créer une liste de nombre grâce à la fonction `range` et la fonction `list`. Le résultat de l'appel d'une fonction `range` **N'EST PAS** une liste pour Python ; pour transformer une énumération obtenue avec `range`, il faut lui appliquer la fonction `list` Voyons ceci sur quelques exemples :

```
>>>list(range(5))
>>>list(range(4,13))
>>>list(range(3,-1))
>>>list(range(1,30,2))
>>>list(range(10,0,-1))
```

c. Exercices

Exercice 2.

1. Définir une liste `I` contenant dont les éléments sont les chaînes de caractères suivantes (n'oubliez pas les espaces) `'je '`, `'déteste'`, `'suis fan de '` et `"l'Informatique"`
2. En utilisant l'accès aux éléments de `I` par leurs indices et grâce à la concaténation des chaînes de caractères, écrire la phrase `"je déteste l'Informatique"` puis la phrase `"je suis fan de l'Informatique"`.
3. Créer une liste `N` contenant les nombres allant de 1 à 82 de 3 en 3 i.e. $1, 4, 7, ..., 79, 82$
4. **Afficher** un par un chacun des nombres de cette liste.
5. Donner l'instruction qui permet de définir la liste `C` représentant l'ensemble suivant - ne pas oublier d'importer le module `math` ou le module `numpy` pour le cosinus (`cos`) :

$$\{x \in [\![0, 100]\!] \mid \cos(x) \geq 0\}$$

```
#Q.1
I=['je ','déteste ','suis fan de ',"l'Informatique"]

#Q.2
I[0]+I[1]+I[3]
I[0]+I[2]+I[3]

#Q.3
N=list(range(1,83,3))
```

```
10
11 #Q.4
12 for i in range(len(N)):
13     print(N[i])
14
15 #Q.5
16 from math import cos
17 C=[x for x in range(101) if cos(x)>=0]
```

2. Manipulation et opérations sur les listes

a. Concaténer des listes

Première opération, comme pour les chaînes de caractères : **la concaténation**. Grâce à l'opérateur `+`, on peut concaténer deux listes, c'est-à-dire les mettre bout à bout pour former une deuxième liste.
Observons le résultat de l'instruction suivante :

```
>>>[3,4,5]+[1,1,1,1,2]
```

On peut également dupliquer (c'est-à-dire concaténer un certain nombre de fois) une liste autant que l'on veut pour former une nouvelle liste grâce à l'opérateur * : Pour `L` une liste et `k` un entier, `L*k` renvoie la concaténation de `k` fois la liste `L` i.e. `L+L+...+L`.
Voyons ceci sur un exemple :

```
>>>[1,6,8]*4
```

b. Modifier un élément d'une liste

Si `L` est une liste, alors `L[k]=x` remplace l'élément d'indice `k` par `x`.

```
>>>L=[1,2,3,4]
>>>L[2]=15
>>>L
```

On peut remplacer une portion entière de liste grâce au slicing. Attention la portion à remplacer et la nouvelle liste qui la remplace doivent avoir la même taille !

```
>>>L=list(range(11)) #liste des nombres de 0 à 10
>>>L[2:5]=[12,34,61] #remplace [2,3,4] par [12,34,61]
>>>L
```

c. Copier une liste

ATTENTION ici ! La copie d'une liste est délicate. Autant pour copier une variable, rien de plus simple : si je veux copier la variable `x` dans une variable `y`, il me suffit de faire `y=x`. Puis si je modifie la valeur de `y`, cela n'affecte pas `x`.

ATTENTION !!!!! En Python, l'instruction `M=L` ne fait pas une simple copie de la liste `L`, elle crée un *alias* de cette liste vers la liste `M` c'est-à-dire que toute modification de `M` affecte `L` et inversement !! Testons dans le shell :

```
>>>L=[1,2,3,4]
>>>M=L
>>>L[0]=4
>>>L
>>>M

>>>M[1]=1
>>>L
>>>M
```

Ainsi pour créer une copie indépendante d'une liste `L`, on doit utiliser la **méthode** `copy()` de l'objet `L`. Observons sur un exemple :

```
>>>L=[1,2,3,4]
>>>M=L.copy()
>>>L[0]=4
>>>L
>>>M
```

Aparté : La méthode `copy()` ne fonctionne pas pour les sous-listes d'une liste... ainsi, si on modifie les sous-listes d'une liste dans un original, elles le seront aussi dans la copie... même avec la méthode `copy`. Pour pallier à ce problème, on utilise le module `copy` et la **fonction** `deepcopy` :

```
>>>from copy import deepcopy
>>>L=[[1,2],[3,4]]
>>>M=deepcopy(L)
>>>L[0][1]=4
>>>L
>>>M
```

d. Supprimer un élément

Pour supprimer un élément d'une liste, on utilise la fonction **`del`**. Cela fonctionne également avec une portion de liste.
Testons :

```
>>>L=[1,5,7,7,8,9,12,12]
>>>del L[-1]
>>>L

>>>del L[2:4]
>>>L
```

Bien sûr, la longueur de la liste diminue après l'utilisation de `del`

```
>>>L=[1,5,7]
>>>len(L)

>>>del L[0]
>>>len(L)
```

e. Insertion d'un élément dans une liste

L'insertion d'un élément dans une liste peut se faire grâce à deux **méthodes** de la classe `list` :

— La méthode `append(x)` qui ajoute l'élément `x` à la fin de la liste ;

— La méthode `insert(i,x)` qui insère l'élément `x` à l'index `i` (et donc décale l'indice des éléments suivants).

Observons :

```
>>>L=['aa','ee','ii','oo']
>>>L.append('uu')
>>>L

>>>L.insert(2,'eeeiii')
>>>L
```

f. Autres méthodes

Voici quelques autres méthodes intéressantes pour interagir avec les listes :

— La méthode `remove(x)` supprime la première occurrence de `x` de la liste ;

— La méthode `pop(k)` supprime l'élément d'indice `i` et **retourne l'élément** en question ;

— La méthode `reverse()` inverse l'ordre des éléments de la liste ;

— La méthode `sort()` trie dans l'ordre croissant (si c'est possible) la liste ;

Exemples

```
>>>L=['z','r','t','a','b','g','d','p','k','h']
>>>L.remove('b')
>>>L
```

```
>>>L.pop(6)
>>>L

>>>L.reverse()
>>>L

>>>L.sort()
>>>L
```

Exercice 3.

1. Créer une liste `L` qui contient 250 chiffre 0 puis 342 chiffres 1 (la liste doit ressembler à `[000...00111...11]`).
2. Créer une liste `N` qui contient les nombres de 2 à 16. Remplacer le nombre 6 (chercher son indice) de la liste par 1000.
3. En utilisant la liste `N` modifiée dans la question précédente et une boucle `for`, remplacer tous les éléments de `N` par leur carré.
4. Ajouter dans `N` le nombre 33 entre les nombres 4 et 9.
5. Trier la liste `N` dans l'ordre décroissant.

```python
#Q.1
L=[0]*250+[1]*342

#Q.2
N=list(range(2,17))

N[4]=1000

#Q.3
N=[n**2 for n in N]

#Q.4
N.insert(1,33)

#Q.5
N.sort().reverse()
```

— La méthode `count(x)` compte le nombre de fois qu'apparaît `x` dans la liste ;
— La méthode `index(x)` retourne l'index de la première occurrence de `x` dans la liste ;
— La fonction `min(L)` renvoie le minimum des éléments de la liste `x`
— La fonction `max(L)` renvoie le maximum des éléments de la liste `x`
— La fonction `sum(L)` renvoie la somme des éléments de la liste `x`

— La fonction `sorted(L)` renvoie une copie triée dans l'ordre croissant de la liste `L` (contrairement à la méthode `sort()`, cette fonction ne modifie pas `L`)

```
>>>L=[3,6,3,4,1,2,2,99,4,45,2,1,2]
>>>L.count(2)

>>>L.index(1)

>>>min(L)
>>>max(L)
>>>sum(L)

>>>M=sorted(L)
>>>L
>>>M
```

Et pour finir ; pour appliquer une fonction à tous les éléments d'un tableau, on peut utiliser la méthode vue pour la définition d'une liste par compréhension. Voyons ceci sur un exemple : j'ai une liste de nombres et je veux mettre tous les nombres de la liste à la puissance 3 puis leur ajouter 12, voici comment faire :

```
>>>L=[3,6,3,4,1,2,2,99,4,45,2,1,2]

>>>def f(x):
...    return x**3+12

>>>M=[f(elem) for elem in L]
>>>M
```

Exercice 4.

On cherche à estimer les racines du polynômes :

$$x^3 - 33,94x^2 + 141,548875x + 49,622625$$

1. Créer une liste `I` contenant les nombres de -30 à 30 avec un pas de 0.01
2. Créer une liste `M` contenant les éléments de `L` auxquels on a appliqué la fonction $f : x \mapsto x^3 - 33,94x^2 + 141,548875x + 49,622625$
3. Créer une liste `S` qui contient les indices i de `M` qui vérifient :

$$(M[i] \leq 0 \text{ et } M[i+1] > 0) \text{ ou } (M[i] \geq 0 \text{ et } M[i+1] < 0)$$

4. En déduire une approximation des racines du polynôme.

```python
#Q.1
I=[e/100 for e in list(range(-3000,3001))]

#Q.2
def f(x):
    return x**3-33.94*x**2+141.5885*x+49.622625

M=[f(e) for e in I]

#Q.3
S=[i for i in range(len(M)-1) if (M[i]>=0.0 and M[i+1]<0.0) or (M[i]<=0.0 and M[i+1]>0.0)]

#Q.4
ind=[[I[i],I[i+1]] for i in S]
```

3. La mise en pratique !

Voici une liste d'exercices afin de mieux comprendre et manipuler les listes en Python :

Exercice 5.

1. Écrire une fonction `somme(L)` qui prend pour argument une liste `L` de nombres et qui *renvoie* la somme des éléments de `L`. Si la liste `L` est vide, le résultat de `somme(L)` devra être 0.
2.
3. Écrire une fonction `produit(L)` qui prend pour argument une liste `L` de nombres et qui *renvoie* le produit des éléments de `L`. Si la liste `L` est vide, le résultat de `produit(L)` devra être 1.
4. Que calcule la fonction `mystere(n)` suivante ?

```python
def mystere(n):
    return produit(list(range(1,n+1)))
```

5. Écrire une fonction `moyenne(L)` qui prend pour argument une liste `L` de nombres et qui *renvoie* la moyenne des éléments de `L`.
6. Écrire une fonction `variance(L)` qui prend pour argument une liste `L` de nombres et qui *renvoie* la variance des éléments de `L`.

   La variance V d'une liste $x_1, ..., x_n$ de nombres de moyenne m est définie par :

$$V = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - m)^2.$$

Correction.

1.

```
def somme(L):
    S=0
    for element in L:
        S+=element
    return S
```

2.

```
def produit(L):
    P=1
    for element in L:
        P*=element
    return P
```

3. Cette fonction, pour un nombre n en argument, renvoie la valeur $n!$.

4.

```
def moyenne(L):
    if len(L)==0:
        return 0
    else:
        return somme(L)/len(L)
```

5.

```
def moyenne(L):
    if len(L)==0:
        return 0
    else:
        return somme(L)/len(L)
```

6.

```python
def variance(L):
    m=moyenne(L)
    n=len(L)
    S=0
    if n==0:
        return 0
    else:
        for i in range(n):
           S+=(L[i]-m)**2
        return S/n
```

Exercice 6.

Écrire une fonction `separation(L,chaine)` qui prend pour argument une liste de chaînes de caractères `L` et une chaîne de caractères `chaine` et qui renvoie une liste `[M,N]` de deux listes : la liste `M` doit contenir tous les mots situés avant (dans l'ordre lexicographique) `chaine` inclus et la liste `N`, les mots situés après.

Correction.

```python
def separation(L,chaine):
    M=[e for e in L if e>chaine]
    N=[e for e in L if e<=chaine]
    return [M,N]
```

Exercice 7. *Liste des termes d'une suite*

Soit (u_n) la suite récurrente :

$$\begin{cases} u_0 \in [-1, +\infty[\\ u_{n+1} = \sqrt{u_n + 1} \end{cases}$$

1. Justifier que cette suite est bien définie.
2. Écrire une fonction `liste_termes(n,initial)` qui renvoie la liste des n premiers termes $u_0, ..., u_{n-1}$ de la suite (u_n) ayant pour terme initial $u_0 =$`initial`.
3. De même, écrire une fonction qui renvoie la liste des termes de la suite récurrente

$$\begin{cases} v_1 = 1 \\ v_{n+1} = \frac{n^2}{(n+1)^2} v_n \end{cases}$$

puis calculer la somme des n premiers termes de la suite pour $n = 100$, 1000, 10000, 100000, 1000000. Que constate-t-on ?

Correction.

1.
2.

```
from math import sqrt

def liste_termes(n,initial):
    L=[initial]
    for i in range(1,n):
        L.append(sqrt(L[i-1]+1))
    return L
```

3.

```
def liste_termes2(n):
    L=[1]
    for i in range(1,n):
        L.append(L[i-1]*((i)**2)/(i+1)**2)
    return L


for i in range(2,7):
    sum(liste_termes2(10**i))
```

Exercice 8. *Crible d'Érathostène*

On cherche dans cet exercice à écrire une fonction qui renvoie la liste des nombres premiers plus petits qu'un certain nombre `n`. Pour cela, on va utiliser la méthode du crible d'Érathostène :

- on part de la liste des nombres allant de 2 à n ;
- on supprime de cette liste tous les nombres divisibles par 2 ;
- dans cette nouvelle liste, on supprime tous les nombres divisibles par 3 ;
- 4 ayant été enlevé, on passe au nombres divisibles par 5 ;
- etc...

Écrire une fonction `crible(n)` qui renvoie la liste des nombres premiers compris entre 0 et n grâce à la méthode du crible d'Érathostène.

Combien y-a-t-il de nombres premiers entre 0 et 100 ? 1000 ? 10000 ?

Correction.

```
def crible(n):
    P=list(range(2,n+1))
    i=0
    while i<len(P):
        for k in P[i+1:]:
            if k%P[i]==0:
                P.remove(k)
        i+=1
    return P


for i in range(2,5):
    len(crible(10*i))
```

Exercice 9.

Importer la fonction `randint` du module `random` que nous avons étudiée dans le chapitre précédent i.e.

```
from random import randint
```

1. Écrire une fonction `ZeroUn(n)` qui renvoie une liste de longueur `n` et dont chaque élément est un entier tiré aléatoirement et uniformément parmi 0 et 1.
2. Écrire une fonction `MaxBandeDeZero(L)` qui détermine la taille de la plus grande plage de 0 consécutifs dans la liste `L` constituée de 0 et de 1.
3. Écrire une fonction qui permet de déterminer la moyenne de la taille de la plus grande plage de 0 consécutifs parmi un échantillon de `N` listes de longueur `n` contenant de 0 et des 1 tirés aléatoirement.

   Conjecturer les valeurs des moyennes théoriques de la plus grande plage de 0 pour des listes de taille 10, 50, 100, 250 et 500.

Correction.

1.

```python
def ZeroUn(n):
    return [randint(2) for i in range(n)]

#Remarque : en fait, la fonction randint intègre directement un argument qui permet de générer une liste de nombres tirés aléatoirement de taille voulue :
# l'instruction randint(2, size=100) renvoie une liste de 100 nombres tirés aléatoirement entre 0 et 1.
#On aurait donc pu écrire :

#def ZeroUn(n):
#    return randint(2, size=n)
```

2.

```python
def MaxBandeDeZero(L):
    maxi = 0
    maxiencours = 0
    for x in L:
        if x==1:
            maxiencours=0
        else:
            maxiencours+=1
        if maxi < maxiencours:
            maxi=maxiencours
    return maxi
```

3.

```python
def MoyenneBande(n,N):
    return moyenne([MaxBandeDeZero(ZeroUn(n)) for i in range(N)])
```

On peut conjecturer les valeurs théoriques suivantes (mais ce ne sont que des approximations !) :
— $2,8$ pour $n = 10$;
— 5 pour $n = 50$;
— 6 pour $n = 100$;
— $7,3$ pour $n = 250$;
— $8,3$ pour $n = 500$;

Exercice 10.

Le tri par sélection est un algorithme de tri : il permet de trier une liste de nombres du plus petit au plus grand (par ordre croissant). Voilà une description d'une étape à répéter de notre

algorithme de tri

— on cherche le plus petit élément du tableau
— on l'échange avec le premier élément du tableau
— on recommence le processus avec le sous-tableau restant (i.e. le tableau moins le premier élément)

1. Écrire une fonction `indice_minimum(L,k)` qui prend pour arguments une liste `L` de nombres et un indice `k` et qui renvoie l'indice du minimum des éléments `L[k]`,...,`L[n-1]` (où `n` est la longueur du tableau).
2. Écrire une fonction `tri_selection(L)` qui prend pour argument une liste `L` de nombres et renvoie la liste triée grâce à l'algorithme décrit plus haut (et qui bien sûr utilise la fonction `indice_minimum`!).

Correction.

```
def indice_minimum(L,k):
    i,m = k,L[k]
    for j in range(k+1,len(L)):
        if L[j]<m:
            i,m= j,L[j]
    return i

def tri_selection(L):
    for k in range(len(L)-1):
        i = indice_minimum(L,k)
        if i!= k:
            L[i],L[k] = L[k],L[i]
    return L
```

Partie C

Matrices et array Numpy

Dans cette partie nous allons manipuler les tableaux bi-dimentionnel, plus communément appelés *matrices* en Mathématiques. En informatique, notamment dans sa partie dédiée au traitement d'images, les matrices sont très utiles pour modéliser une image numérique : chaque coefficient de la matrice encode un pixel de l'image.

Dans un premier temps, nous allons, grâce à la classe `list` de Python, créer notre propres matrices sous forme de listes de listes et quelques fonctions qui leur sont dédiées.
Nous verrons ensuite la modélisation des matrices de la classe `array` de Numpy.

1. Les matrices ou tableaux multi-dimensionnels

Dans cette partie, nous allons nous-mêmes créer une pseudo-classe (nous verrons les chapitre consacré à la P.O.O. comment créer proprement une classe en Python) pour modéliser nos matrices et définir des fonctions utiles comme l'addition ou la multiplication de matrices par exemple.

a. Création et manipulation d'une matrice

Pour la modélisation matricielle en Python, nous allons considérer une matrice `M` comme une liste de listes toutes de même taille :

```
1 #création d'une matrice de taille 2x3
2 M = [
3 [1,2,3],
4 [3,4,5]
5 ]
```

Il faut bien garder à l'esprit que nous modélisons les matrices : les sous-listes de `M` correspondent donc à ses lignes ; elles **doivent** donc toutes avoir **la même taille** (qui correspond donc au nombre de colonnes).

Grâce à ce que nous connaissons déjà sur le comportement et la syntaxe des listes, on peut accéder aux éléments de la matrice `M`, connaître son nombre de lignes ou de colonnes. **Attention !** il ne faut pas oublier que le premier élément d'un liste en Python est d'**indice 0** !

```
>>>M[0][1] #Retourne la valeur du coefficient de la première ligne et deuxième colonne
     de M
2

>>>len(M) #retourne le nombre de lignes
2

>>>len(M[0]) #retourne le nombre d'éléments de la première ligne et donc le nombre de
    colonnes de M
3
```

```
>>>M[1] #retourne la deuxième ligne de la matrice
[3,4,5]
```

Attention au comportement des listes !

On rappelle que le type `list` en Python a un comportement spécifique concernant les copies : étant donnée une liste `L`, il faut utiliser la fonction `copy` ou même `deepcopy` du module `copy` si `L` est une liste de listes pour créer une copie indépendante de `L`.

```
>>>L=[[1,2],[3,4]]
>>>K=L #crée non pas une copie mais un alias de L
>>>K[0][1]=6 #modifie K et L !!!

>>>L[0][1] #vérification
6

>>>from copy import deepcopy
>>>L=[[1,2],[3,4]]
>>>J=deepcopy(L) #crée un copie de L indépendante de L
>>>J[0][1]=6 #modifie J mais pas L

>>>L[0][1] #vérification
2
```

Dans la question 1. de l'exercice suivant, on peut utiliser la syntaxe `[0]*5` pour créer la liste `[0,0,0,0,0]`. On aurait alors envie, pour créer une matrice `M` de taille 4×5 remplie de zéros, d'écrire :

```
>>>M=[[0]*5]*4
```

Au premier abord, cela semble créer ce qu'on attend :

```
>>>M
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Mais NON ! la syntaxe précédente présente le défaut de créer des alias de listes (comme dans l'exemple précédent de `K` par rapport à `L`) et pas des copies indépendantes. Voyons cela :

```
>>>M[0][4]=3
>>>M
[[0, 0, 0, 0, 3], [0, 0, 0, 0, 3], [0, 0, 0, 0, 3], [0, 0, 0, 0, 3]]
```

Aïe, comme chacune des sous-listes sont des alias, elles sont toutes modifiées lorsqu'on en modifie

une !

Pour pallier ce problème, on utilisera la syntaxe `[[0]*3 for k in range(n)]` pour créer une liste de `n` copies indépendantes de `[0]*3` (car elle est recalculée `n` fois et pas simplement dupliquée `n` fois comme avec la syntaxe précédente).

2. Les array Numpy

Nous avons vu comment modéliser les matrices mathématiques grâce aux listes (de type **`list`**) natives de Python.
Les "**`list`**" de Python sont très pratiques : elle peuvent contenir n'importe quel type d'objets et on peut leur ajouter/supprimer des éléments très facilement.
En contrepartie, le temps d'accès aux éléments n'est pas le plus efficace possible.

Or, on remarque que nos modélisations de matrices n'ont pas forcément besoin de toutes les propriétés offertes par le type **`list`**, en effet :

- tous les éléments d'une matrice ont la même nature ;
- une matrice a une taille "constante" ;

Ainsi, afin de gagner en rapidité lors de l'accès aux éléments, il est plus pertinent de modéliser une matrice par une structure en tableau (multidimensionnel) qui a l'avantage de permettre l'accès à ses éléments à coût constant, mais qui ne peut contenir que des éléments de même nature et qui est de taille fixée à la création (ce qui ne nous gênera pas ici !).

Pour modéliser nos matrices par des tableaux, nous allons utiliser le module `numpy` qui propose un nouveau type d'objet, le type `ndarray`, que nous appellerons plus simplement "*array*" (tableau en anglais). Les array proposent exactement la structure décrite en introduction et sont donc idéaux pour modéliser nos matrices !

Tout d'abord, importons le module `numpy` avec son surnom "habituel" !

```
>>>import numpy as np
```

a. Création d'un array

Pour créer un array, le module `numpy` fournit la fonction `array(M)` qui prend un argument obligatoire `M` qui est une liste de listes où toutes les sous-listes de `M` ont la même taille et contiennent des éléments de même type (en tenant compte des conversions implicites de Python, par ex : **`int`** vers **`float`** vers **`complex`**).

Voici quelques exemples de création d'array :

```
>>>M=[[1,2,3],[4,5,6]] #on crée une "matrice" M de taille 2x3
>>>A=np.array(M) #on la convertit en array numpy

>>>B=np.array([[2,1],[1.2,np.pi],[0,0]]) #on crée directement un array en lui donnant
    une matrice explicite
#on note que puisque certains coefficients sont de type float, les coefficients de
    type int sont convertis en float lors de la création de l'array
```

La fonction `array(M,dtype)` prend également un argument facultatif `dtype` qui permet de préciser le type des données que l'on souhaite mettre dans l'array. Par exemple :

```
>>>M=[[1,2,3],[4,5,6]] #on crée une "matrice" M de taille 2x3
>>>A=np.array(M,dtype=float) #les élements de A seront convertis en float même s'ils
    étaient tous de type int au départ.
```

Les valeurs possibles de `dtype` sont :

- **bool** (True/False ou 1/0) ;
- **float** ;
- **complex** ou plus intéressant encore :
- (u)**int8**/(u)**int16**/(u)**int32**/(u)**int64**=**int** qui permettent de préciser sur combien de bits (8, 16 ,32 ou 64 seront codés les entiers relatifs (naturels en rajoutant le u) de l'array. Cela nous permettra notamment, lorsque nous manipulerons des images, en niveau de gris par exemple, de les rendre plus "légères" en mémoire en utilisant `dtype=uint8` (entiers de 0 à 255).

On a également des fonctions permettant de créer des array particuliers :

- `zeros((n,m),dtype=type)` renvoie un array de `n` lignes, `m` colonnes rempli de 0 (convertis dans le type indiqué).
  Remarque : l'argument obligatoire est un **tuple** donc les parenthèses sont nécessaires ; on peut également créer des array remplis de zéros uni/tri/quadri/etc-dimensionnels avec un **tuple** adéquat.

  Testez les instructions suivantes :

  ```
  >>>np.zeros((2,2))

  >>>np.zeros((9),dtype=int)

  >>>np.zeros((2,5,3))
  ```

- `ones(tuple,dtype)` se comporte comme `zeros` mais renvoie un array remplis de 1.
- `eye(n)` renvoie la matrice identité de taille `nxn`
- `diag(L)` renvoie la matrice diagonale de taille **len**(L)xlen(L) dont les coefficients diagonaux sont les éléments de `L` (si ses éléments sont de même type bien-sûr).

b. Manipulation d'un array

Pour récupérer des éléments d'un array, tout ce que l'on connaît sur les listes s'applique, par exemple :

```
>>>A=np.array([
[1,2,5],
[3,4,0],
```

```
],dtype=float) #on crée un array A de flottants

>>>A[0] #renvoyer la première ligne
[1.,2.,5.]
>>>A[1][2] #coefficients de 2ème ligne, 3ème colonne
0.
```

Mais on a également quelques raccourcis syntaxiques bien pratiques par rapport au listes simples :

Si `A` est un array à 2 dimensions :

— `A[i,j]` renvoie l'élément de $i+1$ème ligne et $j+1$ème colonne (équivalent `A[i][j]`)

```
>>>A[1,2] #équivalent à A[1][2]
0.
```

— Le slicing des listes est généralisé aux array, notamment pour les colonnes :
`A[:,j]` renvoie le $j+1$ème colonne de `A` (ce qui était impossible à faire aussi simplement avec une matrice modélisée par une liste de listes)

```
>>>A[:,1]
array([2.,4.])
```

c. Exercices de création d'arrays

Exercice 11.

Modéliser les matrices suivantes avec des array en utilisant les outils de création et de manipulation précédents :

1. $\begin{pmatrix} 1 & 0 & \dots & 0 & 1 \\ 0 & 1 & \ddots & & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & & \ddots & 1 & 0 \\ 1 & 0 & \dots & 0 & 1 \end{pmatrix}$ avec 15 lignes et 15 colonnes
2. $\begin{pmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 2 & \dots & 29 & 30 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix}$ avec 30 lignes et 30 colonnes

3. $\begin{pmatrix} 3 & 0 & \dots & 0 & 0 \\ 0 & 6 & \ddots & & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & & \ddots & 87 & 0 \\ 0 & 0 & \dots & 0 & 90 \end{pmatrix}$

Correction.

```python
#Q1
A1=np.eye(15)
A1[0,14]=1
A1[14,0]=1

#Q2
A2=np.zeros((30,30))
A2[1,:]=[i for i in range(1,31)]

#Q3
A3=np.eye(30) #on remarque qu'il y a 30 éléments sur la diagonale !
for i in range(30)
    A3[i,i]=3*(i+1)
```

Exercice 12.

1. Définir une fonction `base(n)` qui prend en argument un entier naturel non nul `n` et qui renvoie la liste des array correspondant aux n^2 matrices avec n lignes et n colonnes quasiment remplies de 0 mais avec un 1 à chaque place possible.
2. Définir une fonction `matrice_suite(n)` qui prend en argument un entier naturel non nul `n` et qui renvoie l'array correspondant à la matrice avec n lignes et n colonnes telle que :

$$A = \begin{pmatrix} 1 & 2 & \dots & n-1 & n \\ n+1 & n+2 & \dots & 2n-1 & 2n \\ \vdots & \vdots & & \vdots & \vdots \\ ? & ? & \dots & n^2-1 & n^2 \end{pmatrix}$$

Correction.

```python
#Q1
def base(n):
    Liste_matrices=[]
    for i in range(n):
        for j in range(n):
            E=np.zeros((n,n))
            E[i,j]=1
            Liste_matrices.append(E)
    return Liste_matrices

#Q2
def matrice_suite(n):
    A=np.zeros((n,n))
    for i in range(n):
        A[i,:]=[j+1 for j in range(n)]
    return A
```

d. Opérations sur les array

Dans la partie précédente, nous avons codé plusieurs opérations matriciels pour notre modélisation en liste de listes. Grâce aux array, la syntaxe de ces opérations est naturelle et simple :

On considère `A,B` des array de mêmes dimensions et `c` un nombre (int, float ou complex).

- `A+B` renvoie l'array somme coefficient par coefficient de `A` et `B`
- `A*B` renvoie l'array produit coefficient par coefficient de `A` et `B`
  Attention ! Ce N'est PAS le produit matriciel et l'instruction `A**n` renvoie `A*...*A` (`n` termes)
- `c*A` renvoie l'array des coefficients de `A` tous multipliés par `c`

Si les array `A,B` sont **bi-dimensionnels** et donc modélisent des matrices, on a les outils suivants :

- `dot(A,B)` renvoie l'array correspondant au **produit matriciel** de `A` par `B` s'il est possible ; de plus `matrix_power(A,n)` renvoie A^n.
- `A.T` renvoie l'array correspondant à **la transposée** de `A`

Toujours pour des matrices modélisées par des array, on a accès au déterminant et à l'inverse (si elle existe) à partir d'un sous-module de `numpy` : la bibliothèque `numpy.linalg`
Je vous invite à aller faire un tour sur la page de la sous-module numpy.linalg pour connaître toutes ses spécificités.

```
import numpy.linalg as lin
```

— `det(A)` renvoie le **déterminant** de `A`
— `inv(A)` renvoie l'array correspondant à **l'inverse** de `A` si elle existe

e. On passe à la pratique !

Exercice 13.

Donner les instructions les plus courtes possibles permettant de renvoyer le résultat demandé - les matrices seront bien-sûr modélisée par des array !

1. Créer une matrice de taille 12×20 remplie de 7
2. Créer une matrice triangulaire supérieure de taille 10×10 dont tous les termes sont nuls exceptés les 9 coefficients de la surdiagonale qui sont égaux à 1.
   Calculer ses puissances $2, 3, ..., 9, 10$. Que dire de cette matrice ?
3. Dans cette question, on importera le module `random` (que vous surnommerez `rd` et sa fonction `random()` (faites des tests, que renvoie-elle ?)
   a) définir une fonction `mat_aleatoire()` qui renvoie une matrice de taille 2×2 dont chaque coefficient est un nombre flottant tiré "aléatoirement" entre 0 et 1.
   b) définir une matrice `A=mat_aleatoire()` puis renvoyer :
      — sa deuxième colonne,
      — sa transposée,
      — son déterminant puis son inverse si elle existe
   c) Faire plusieurs essai de calcul de déterminant avec des matrices "tirée aléatoirement" avec la fonction `mat_aleatoire()`. Qu'en pensez-vous ?

Correction.

```
#Q1
A=7*np.ones((12,20))

#Q2
B=np.zeros((10,10))
for i in range(9):
    B[i,i+1]=1

import numpy.linalg as lin
for k in range(2,11):
    print(lin.matrix_power(B,k))
#B**10 est la matrice nulle !
```

```python
#Q3
import random as rd
#la fonction rd.random() renvoie un nombre "aléatoire" entre 0 et 1 exclu
#Q3.a)
def mat_aleatoire():
    A=np.zeros((2,2),dtype='float')
    A[0,0],A[0,1],A[1,0],A[1,1]=rd.random(),rd.random(),rd.random(),rd.random()
    return A
#Q3.b)
A=mat_aleatoire()
A[:,1]
A.T
lin.det(A)
if lin.det(A)!=0:
    lin.inv(A)
#Q3.c)
L=[lin.det(mat_aleatoire()) for i in range(10000)]
Z=[d for d in L if (d<=0.00000001 and d>=-0.00000001)]
#on compte le nombre de déterminants très proches de zéros
#pas zéro pile du fait des "erreurs" d'opérations sur les float
>>>len(Z)
0
#on peut aussi prouver que la proba que det(A)=0 est nulle :)
```

Exercice 14. *Matrice de Vandermonde*

On appelle **matrice de Vandermonde**, une matrice $V = V(\alpha_1, ..., \alpha_n)$ de la forme :

$$V = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \dots & \alpha_n^{n-1} \end{pmatrix}$$

où $n \in \mathbb{N}^*$ et $\alpha_1, ...\alpha_n \in \mathbb{R}$.

1. Définir une fonction `Vandermonde(L)` qui prend en argument une liste non vide `L=[a1,...,an]` et qui renvoie la matrice (sous forme d'array) de Vandermonde V(`a1,...,an`).
2. Calculer grâce à Python, la valeur de

   $$\det(V(1, 2, 3, 4))$$

   et renvoyer son inverse.
3. Déterminer mathématiquement une formule pour $\det(V(\alpha_1, ..., \alpha_n))$ et retrouver que $\det(V(1, 2, 3, 4))$ vaut bien la valeur annoncée par Python !

Correction.

```
def Vandermonde(L):
    n=len(L)
    A=ones((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j+1:]*=L[i]
    return A
```

Partie D

Autres structures de données en Python

1. Les tuples

Le type `tuple` en Python dont on a beaucoup parlé sans vraiment voir ce que c'est, peut être apparenté au type `list` mais les objets de ce type sont non mutables : on ne peut pas rajouter d'élement dans un `tuple` et on ne peut pas mofifier les éléments dans un `tuple`. Ainsi, sa taille en mémoire est fixe à la création de celui-ci. Il permet donc une meilleure gestion de la mémoire et un accès plus rapide que pour une `list`.
Un `tuple` se définit de la même façon de base qu'une `list` avec des parenthèses à la place des crochets - ou même sans parenthèse si le contexte le permet !

```
>>>t=(1,2,3,3.14)
>>>t
(1, 2, 3, 3.14)
>>>u=3,2,1
>>>u
(3,2,1)
```

Dans quel cas ne peut-on pas se passer de parenthèses ? Pour la création d'un tuple avec un seul élement ; dans ce cas, la syntaxe Python veut qu'on place une virgule après l'élement en question (pour ne pas confondre avec les parenthèses d'une expression mathématique) :

```
>>>t1=(1,)
>>>t
(1,)
```

La récupération des éléments d'un `tuple` se fait de la même manière que pour une liste :

```
>>>t=1,2,3,3.14
>>>t[0]
1
>>>t[3]
3.14
>>>t[1:3]
(2,3)
```

On peut également concaténer des `tuple` comme on le ferait pour des listes :

```
>>>t=1,2,3,3.14
>>>u=3,2,1
>>>t+u
(1,2,3,3.14,3,2,1)
```

```
>>>u+('a',)
(3,2,1,'a')
```

Comme on l'a vu dans la première partie sur les variables, les **tuple** permettent d'affecter des valeurs à plusieurs variables dans la même instruction, et de pouvoir échanger les valeurs de plusieurs variables avec une syntaxe épurée :

```
>>>a,b,c=1,2,3
>>>a
1
>>>b
2
>>>c
3
>>>a,b,c=c,a,b
>>>a
3
>>>b
1
>>>c
2
```

Une fonction peut également retourner plusieurs valeurs de façon syntaxiquement simple grâce aux **tuple**

```
def div_euclidienne(a,b):
    q,r=0,a
    while r>=b:
        q,r=q+1,r-b
    return q,r

>>>div_euclidienne(11,4)
(2,3)
```

Attention ! Comme indiqué dans l'introduction, les **tuple** sont non-mutables *(ou immuables)*, ainsi, on ne peut ni modifier, ni supprimer, ni ajouter des éléments à un **tuple** ; faites le test !

```
>>>t=(1,2,3,4)
>>>t[0]=2
TypeError: 'tuple' object does not support item assignment
```

2. Les dictionnaires

Les **dictionnaires** sont des objets de type **dict** et sont très similaires aux listes. La grande différence tient au fait que les éléments d'un dictionnaire sont non pas repérés par des indices mais par des **clés**

c'est-à-dire (presque) n'importe quel objet Python : chaque valeur du dictionnaire est identifié par un une chaîne de caractère, un complexe, un float, etc...

a. Créer un dictionnaire

Pour créer un dictionnaire, on utilise à la place des crochets d'une liste, des accolades ouvrantes `{` et fermantes `}`.
Ensuite, on indique chaque clé suivi de sa valeur associée séparés par `:` (deux points) ; puis les paires clé :valeur sont séparées par des virgules :

```
>>>di={'clé': 3.5, 'key': 'voilà', 5: 6}
>>>di
{'clé': 3.5, 'key': 'voilà', 5: 6}
```

On peut également créer un dictionnaire à partir d'un dictionnaire vide et en indiquant les valeurs correspondantes aux clés voulues a posteriori :

```
>>>d={} #dictionnaire vide
>>>d['maths']=1
>>>d['info']=2
>>>d
{'maths': 1, 'info': 2}
```

b. Accéder aux éléments d'un dictionnaire

Quand on connait les clés d'un dictionnaire, on emploie une syntaxe similaire à celle des listes en indiquant bien-sûr, à la place de l'indice de l'éléments, sa clé :

```
>>>d={'maths': 1, 'info': 2}
>>>d['maths']
1
>>>d['info']
2
```

Si on ne les connait pas, on peut récupérer, sous forme d'itérable (à l'instar de ce que renvoie la fonction **`range`**) : toutes les clés, ou tous les éléments, ou tous les couples (clé,élément) sour forme de **`tuple`**.
De même que pour le retour d'une fonction **`range`**, on peut utiliser la fonction **`list`** pour convertir ces itérables en liste ou bien-sûr les utiliser directement dans une boucle **`for`**

Récupération des clés

```
>>>di={'clé': 3.5, 'key': 'voilà', 5: 6}
>>>di.keys()
dict_keys(['clé', 'key', 5]) #itérable
>>>list(di.keys())
['clé', 'key', 5]
```

Récupération des éléments

```
>>>di={'clé': 3.5, 'key': 'voilà', 5: 6}
>>>di.values()
dict_values([3.5, 'voilà', 6]) #itérable
>>>list(di.values())
[3.5, 'voilà', 6]
```

Récupération des couples (clé,élement)

```
>>>di={'clé': 3.5, 'key': 'voilà', 5: 6}
>>>di.items()
dict_items([('clé', 3.5), ('key', 'voilà'), (5, 6)]) #itérable
>>>list(di.items())
[('clé', 3.5), ('key', 'voilà'), (5, 6)]
```

Il existe beaucoup d'autres fonctions et méthodes associées à la classe **dict** que je vous laisse lire dans la documentation officielle. Après la lecture du paragraphe sur les dictionnaires, je vous invite à vous rendre au tout début de cette page web et de parcourir le sommaire afin de revoir les notions que nous avons vues dans ce chapitre et le précédent (de manière plus théorique bien-sûr) et de découvrir de nouvelles structures (les **set** par exemple) et bien d'autres spécificités de Python.

c. Exemple d'utilisation d'un dictionnaire

On va utiliser un dictionnaire pour modéliser la caisse d'un magasin : chaque clé vaudra la valeur des billets/pièces potentiellement présents dans la caisse *(pour simplifier, on ne considéra pas les centimes et le plus gros billets sera de* 100€*)* et l'élément de chaque clé `n` correspondra au nombre de billets/pièces de `n`€ dans la caisse. Exemple :

Modélisation de caisses

```
>>>caisseA={100:0, 50:0, 20:0, 10:0, 5:0, 2:0, 1:0} #caisse vide
>>>caisseB={100:2, 50:5, 20:1, 10:12, 5:3, 2:0, 1:27}
```

Exercice 15.

Étant donnée deux caisses modélisées comme précédemment et affectées à des variables `caisseA` et `caisseB` ; donner une ou plusieurs instructions qui permettent de :

1. Déterminer le nombre de billets de 20€ dans la caisse `caisseA`.
2. Ajouter 5 pièces de 2€ dans la caisse `caisseA`.
3. Retirer 6€ dans la caisse `caisseA` **si c'est possible !!** (et on le sait pas a priori).
4. Fusionner deux caisses `caisseA` et `caisseB` dans une nouvelle caisse `caisseC`.
5. Déterminer le montant totale de la caisse `caisseA` (sans connaissance a priori des valeurs des pièces et billets !)

Correction.

1. `caisseA[20]`.
2. `caisseA[2]+=5`.
3.

```
if caisseA[1]>=6:
    caisseA[1]-=6
elif caisseA[1]>=4:
    if caisseA[2]>=1:
        caisseA[1]-=4
        caisseA[2]-=1
    elif caisseA[5]>=1:
        caisseA[1]-=1
        caisseA[5]-=1
elif caisseA[2]>=3:
    caisseA[2]-=3
else:
    print("Impossible d'enlever 6 euros dans la caisse !")
```

4.

```
caisseC=dict()
for cle in caisseA.keys():
    caisseC[cle]=caisseA[cle]+caisseB[cle]
```

5.

```
Somme=0
for cle in caisseA.keys():
    Somme+=caisseA[cle]*cle
```

Nous verrons dans un prochain T.P. la notion d'algorithme **glouton** qui nous permettra de produire un algorithme de rendu de monnaie d'une caisse (supposée "pleine à ras-bord").