# Chapitre 2 : Listes

## I La notion de Type

En programmation la notion de "type de donnée"' est un concept important. L'ordinateur n'effectuera pas les mêmes tâches avec un entier, un réel, un mot, un complexe, une matrice, une image...

Tous les objets en Python on un type bien défini, chacun ayant des attributs et des fonctionnalités différents.

Voici quelques types que nous pour la plupart utilisé :

- Pour les lettres, mots, phrases : c'est le type 'string' abrégé en `str`.
- Pour les nombres il y a plusieurs types différents :
  1. Les entiers : 'integer' abrégé en `int`.
  2. Les réels : `float`
  3. Les nombres complexes : `complex`
- Pour vérifier si quelque chose est vrai `True` ou faux `False`, on utilise le type booléen. Ce sont des variables qui ne peuvent prendre que ces deux valeurs : `True` ou `False`.

```
>>> type("bonjour")
<class 'str'>
>>> type(1)
<class 'int'>
>>> type(1.)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
>>> type(2>3)
<class 'bool'>
```

Pour connaitre le type d'un objet on peut utiliser la fonction `type()`

L'objectif de cette section est d'introduire un nouveau type : `list`

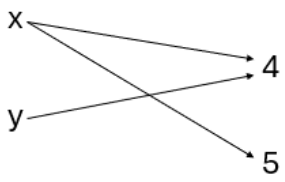
## II Qu'est-ce qu'une liste ?

Une liste, de type `list`, est une structure de données non homogènes : ses éléments peuvent être de types différents, et modifiable : on peut changer ou même supprimer des éléments de la liste. En pratique, une liste est délimitée par des crochets.

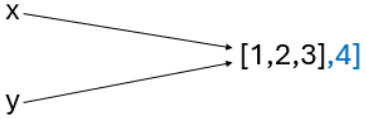
## III Objets immuables et objets mutables

Sur Python, tout est un objet. Certains sont dits "immuables" et d'autres "mutables".

Les objets immuables ne peuvent pas être modifiés. Par exemple le nombre 5 :
Quand on écrit `x=5`, `x` va pointer vers l'objet 5. Si on écrit ensuite `y=x`, on crée alors une autre réfèrence à ce même objet.
Si on écrit par la suite `x=x+1`, on n'a pas modifié l'objet 5 mais simplement changé la référence vers un autre entier. En particulier, cela ne change pas la réfèrence de `y`.




Les objets mutables peuvent en revanche eux être modifiés. Par exemple la liste `[1,2,3]` :
Quand on écrit `x=[1,2,3]`, `x` va pointer vers l'objet `[1,2,3]`. Si on écrit ensuite `y=x`, on crée alors une autre référence à ce même objet. Toutefois si ensuite on écrit `x.append(4)` — ce qui, on le verra plus tard, ajoute 4 à la liste en dernière position — cela modifie l'objet liste lui-même. L'objet a changé, pas les références. En particulier, si on demande la valeur de `y` ensuite, Python donnera aussi `[1,2,3,4]`.



# IV Création d'une liste

Nous avons vu que l'on peut créer une liste manuellement en tapant l'instruction suivante :

```
L = [x1,x2,..., xn]
```

On obtient alors une liste de $n$ éléments, les `xi`, qui peuvent être de types différents.
Par exemple, si on définit `L=[1,2+3j,"texte",[2,1.1]]`, on a une liste de 4 éléments, dont les types sont .....
On voit sur cet exemple que l'on peut même avoir des listes de listes.
La liste vide est définie par `L=` .....
On rappelle que l'instruction `range` permet de créer des listes d'entiers.
On peut créer une liste à l'aide d'opérations sur une autre liste . La syntaxe est la suivante :

```
L = [f(k) for k in liste]
```

On parle parfois de « liste en compréhension »

**Exemple.** Créer une liste `L` contenant les 5 premiers nombres pairs en partant de 0 avec chacune des méthodes précédentes.

— Manuellement .....
— En compréhension .....

# V Accès aux éléments d'une liste

Les listes étant des **types mutables**, on peut également changer ou supprimer des éléments. On va tester les opérations suivantes sur la liste :

```
L = [28,11,14,11]
```

| Instruction Python | Opération effectuée | Exemples |
|---|---|---|
| `L[k]` | | `L[2]` ↓ |
| `L[k]=x` | | `L[0]=4` ↓ |

**Remarque** Pour accéder au dernier élément d'une liste, on peut taper :

..... ou plus simplement .....

**Mise en garde !** ⚠
Pour économiser de la mémoire, `python` ne crée pas de nouvelle liste lors d'une affectation du type `L2=L`, mais donne juste un deuxième nom à la liste `L`, qui est `L2`. Le gros inconvénient est que si l'on modifie `L2`, `L` sera modifié aussi... (c'est valable pour tout les types mutables)

Pour créer une nouvelle liste identique à `L`, il y a deux solutions :

..... et .....

# VI Opérations sur les listes

Voici les opérations sur les listes que vous pouvez être amenés à utiliser :

| Instruction Python | Opération effectuée | Exemples |
|---|---|---|
| `+` | | `[2,4] + [3,2]` ↓ |
| `*` | | `3*[0,1]` ↓ |
| `==` | | `[0,2]==[0,2,2]` ↓ |
| `len` | | `len([0,4])` ↓ |
| `in` | | `[0,1] in [0,1,2]` ↓ |

**Mise en garde !** 

Il faut faire attention que les opérations `+` et `*` ne désignent pas des sommes et multiplications des éléments de la liste.

Par exemple :
`[1,2]+[3,4]` ne renvoie pas `[4,6]`,
mais : .....
et `3*[1,2]` ne renvoie pas `[3,6]`,
mais : .....

On peut également ajouter et enlever des éléments. On va tester les opérations suivantes sur la liste :

```
L = [28,11,14,11]
```

| Instruction Python | Opération effectuée | Exemples |
|---|---|---|
| `L.append(x)` | | `L.append(5)` ↓ |
| `L.insert(i,x)` | | `L.insert(1,5)` ↓ |
| `L.pop(i)` | | `print(L.pop(1))` ↓ |
| `L.index(x)` | | `L.index(11)` ↓ |

**Exemple.** Pour ajouter 3 à la fin de la liste `[3,4,2]` il y a au moins 3 façons :

1. Avec `+` .....
2. Avec `append()` .....
3. Avec `insert` .....

## VII Boucles sur les listes

On peut faire des boucles `for` sur les listes, comme avec la fonction `range(a,b)`. Il faut penser à `range(a, b)` comme une liste particulière de la forme $[a, a+1, a+2, \cdots, b-1]$.

Il y a deux possibilités pour faire des boucles `for` sur une liste :

**PARCOURS PAR ELEMENTS :**
```
for e in L :
    instructions_dependant_de_e
```

**PARCOURS PAR INDICES :**
```
for i in range(len(L)) :
    instructions_dependant_de_i
```

Les deux types de parcours sont utiles voici quelques exemples pour s'en convaincre.
Les deux exemples suivants affichent 1, puis 2, puis 'a'.

**PARCOURS PAR ELEMENTS :**
```
L = [1,2,'a']
for e in L :
    print(e)
```

**PARCOURS PAR INDICES :**
```
L=[1,2,'a']
for i in range(len(L)) :
    print(L[i])
```

Les deux programmes suivants détermine la liste des positions de tous les 3 dans la liste `L`.

**PARCOURS PAR ELEMENTS :**
```
L = [3,4,3,2,3]
P = []
c = 0
for e in L:
    if e == 3:
        P = P+[c]
    c+=1
print(P)
```

**PARCOURS PAR INDICES :**
```
L = [3,4,3,2,3]
P = []
for i in range(len(L)) :
    if L[i] == 3 :
        P.append(i)
print(P)
```

Pour déterminer quel type de parcours il faut savoir si on a besoin de l'indice ou non.

## VIII Exercices

**Exercice 1.** La fonction `randint` de la bibliothèque `random` prend en argument deux entiers $p < q$ et renvoie un entier choisie au hasard dans $[\![p, q]\!]$.

1. Écrire une fonction `Listealeatoire1` qui prend argument un entier $n$ et qui renvoie une liste de longueur $n$ d'entiers aléatoirement choisis dans $[\![0, 10]\!]$ en créant une liste vide `L` et en ajoutant des éléments à l'aide de la méthode `append`
2. Écrire une fonction `Listealeatoire2` faisant la même chose que `Listealeatoire1` mais où la liste est crée en compréhension.

Réponse :

**Exercice 2.**
1. Écrire une fonction `Moyenne1` qui prend en argument une liste `Note` de note et qui calcul la moyenne des notes.
2. Écrire une fonction `Moyenne2` qui prend en argument une liste `Note` de note et une liste de coefficient `Coef` et qui calcul la moyenne pondérée de ces notes.

Réponse :

**Exercice 3.** Écrire une fonction `remplace` qui prend en argument une liste `L`, deux éléments `e1` et `e2` qui renvoie la liste `L` où toutes les occurrences de `e1` sont remplacées par `e2`.

Réponse :

**Exercice 4.** Écrire une fonction `PairImpair` qui prend en argument une liste $L$ et qui renvoie deux listes $M$ et $N$, $M$ contenant tout les éléments d'indice pair et $N$ ceux d'indice impair.

Réponse :

**Exercice 5.** Écrire une fonction `Singletons` qui prend en argument une liste `L` d'entiers et renvoie la liste contenant les mêmes nombres mais au plus une fois et dans le même ordre.

Par exemple, `Singletons([2,1,2,1,3,2,1,4])` renverra la liste `[2,1,3,4]`.

Réponse :