## jupytext: cell\_metadata\_filter: -all formats: md:myst text\_representation: extension: .md format\_name: myst format\_version: '0.8' jupytext\_version: 1.5.0 kernelspec: display\_name: Python 3 language: python name: python3

# Les listes

Une liste est une collection ordonnée de valeurs.

Dans une liste, chaque valeur occupe une position bien définie que l'on repère par un entier appelé \*\*indice\*\*.

La première valeur est associée à l'indice 0, la seconde à l'indice 1, \_etc.\_

Une liste a une longueur (\_i.e.\_ un nombre d'éléments) finie, ainsi la liste vide a pour longueur 0.

En Python, il n'est pas nécessaire que tous les éléments d'une liste soient du même type, même si dans les exemples que nous considérerons, ce sera souvent le cas.

On peut trouver des informations précieuses sur le sujet des listes dans l'aide en ligne de Python disponible à l'adresse : <https://docs.python.org/3/tutorial/datastructures.html>.

## Avant-propos : listes et itérables

Dans la suite, nous parlerons de listes, qui est un type de données bien spécifique en Python. Toutefois, une grande partie de notre propos pourra se transposer à l'ensemble des itérables en Python (c'est-à-dire l'ensemble des objets Python dont on peut parcourir les éléments un à un).

Il existe toutefois une différence majeure entre listes et itérables : nous verrons dans la suite de ce chapitre que l'on peut accéder au $i$-ème élément d'une liste simplement, alors que ce n'est généralement pas possible pour un itérable (pour ce dernier, il faudra parcourir l'ensemble de ses éléments et s'arrêter lorsque l'on est effectivement rendu au $i$-ème).

Toutefois, si l'on a un itérable iterable, il est possible de le transformer en liste simplement à l'aide de la fonction list :

liste = list(iterable)

(creation-liste)=

## Création de liste

Pour créer une liste contenant des éléments définis (par exemple la liste contenant les entiers 1, 5 et 7), il est possible d'utiliser la syntaxe suivante :

liste = [1, 5, 7]

De la même façon, on peut créer une liste vide (ne contenant aucun élément) :

liste = []

print(len(liste))

On voit ici la fonction len qui retourne la taille d'une liste passée en argument (ici 0 puisque la liste est vide).

Toutefois, lorsque l'on souhaite créer des listes longues (par exemple la liste des 1000 premiers entiers), cette méthode est peu pratique. Heureusement, il existe des fonctions qui permettent de créer de telles listes. Par exemple, la fonction range(a, b) retourne un itérable contenant les entiers de a (inclus) à b (exclu) :

it = range(1, 10) # it = [1, 2, 3, ..., 9]

it = range(10) # it = [0, 1, 2, ..., 9]

it = range(0, 10, 2) # it = [0, 2, 4, ..., 8]

On remarque que, si l'on ne donne qu'un argument à la fonction range, l'itérable retourné débute à l'entier 0. Si, au contraire, on passe un troisième argument à la fonction range, cet argument correspond au pas utilisé entre deux éléments successifs.

## Accès aux éléments d'une liste

Pour accéder au $i$-ème élément d'une liste, on utilise la syntaxe :

liste[i]

Attention, toutefois, le premier indice d'une liste est 0, on a donc :

liste = [1, 5, 7]

print(liste[1])

print(liste[0])

On peut également accéder au dernier élément d'une liste en demandant l'élément d'indice -1 :

liste = [1, 5, 7]

print(liste[-1])

print(liste[-2])

print(liste[-3])

De la même façon, on peut accéder au deuxième élément en partant de la fin *via* l'indice -2, *etc.*

Ainsi, pour une liste de taille $n$, les valeurs d'indice valides sont les entiers compris entre $-n$ et $n - 1$ (inclus).

Il est également à noter que l'accès aux éléments d'une liste peut se faire en lecture (lire l'élément stocké à l'indice i) comme en écriture (modifier l'élément stocké à l'indice i) :

liste = [1, 5, 7]

print(liste[1])

liste[1] = 2

print(liste)

Enfin, on peut accéder à une sous-partie d'une liste à l'aide de la syntaxe liste[d:f] où d est l'indice de début et f est l'indice de fin (exclu). Ainsi, on a :

liste = [1, 5, 7, 8, 0, 9, 8]

print(liste[2:4])

Lorsque l'on utilise cette syntaxe, si l'on omet l'indice de début, la sélection commence au début de la liste et si l'on omet l'indice de fin, elle s'étend jusqu'à la fin de la liste :

liste = [1, 5, 7, 8, 0, 9, 8]

print(liste[:3])

print(liste[5:])

(parcours-liste)=

## Parcours d'une liste

Lorsque l'on parcourt une liste, on peut vouloir accéder :

* aux éléments stockés dans la liste uniquement ;
* aux indices de la liste uniquement (même si c'est rare) ;
* aux indices de la listes et aux éléments associés.

Ces trois cas de figure impliquent trois parcours de liste différents, décrits dans ce qui suit.

**Attention.** Quel que soit le parcours de liste utilisé, il est fortement déconseillé de supprimer ou d'insérer des éléments dans une liste pendant le parcours de celle-ci.

### Parcours des éléments

Pour parcourir les éléments d'une liste, on utilise une boucle for :

liste = [1, 5, 7]

for elem in liste:

print(elem)

Dans cet exemple, la variable elem va prendre successivement pour valeur chacun des éléments de la liste.

### Parcours par indices

Pour avoir accès aux indices (positifs) de la liste, on devra utiliser un subterfuge. On sait que les indices d'une liste sont les entiers compris entre 0 (inclus) et la taille de la liste (exclu). On va donc utiliser la fonction range pour cela :

liste = [1, 5, 7]

n = len(liste) # n = 3 ici

for i in range(n):

print(i, liste[i])

### Parcours par éléments et indices

Dans certains cas, enfin, on a besoin de manipuler simultanément les indices d'une liste et les éléments associés. Cela se fait à l'aide de la fonction enumerate :

liste = [1, 5, 7]

for i, elem in enumerate(liste):

print(i, elem)

On a donc ici une boucle for pour laquelle, à chaque itération, on met à jour les variables i (qui contient l'indice courant) et elem (qui contient l'élément se trouvant à l'indice i dans la liste liste).

Pour tous ces parcours de listes, il est conseillé d'utiliser des noms de variables pertinents, afin de limiter les confusions dans la nature des éléments manipulés. Par exemple, on pourra utiliser i ou j pour noter des indices, mais on préfèrera elem ou val pour désigner les éléments de la liste.

(ex4.1)=

### Exercice

Écrivez une fonction en Python qui permette de calculer l'argmax d'une liste, c'est-à-dire l'indice auquel est stockée la valeur maximale de la liste.

Si cette valeur maximale est présente plusieurs fois dans la liste, on retournera l'indice de sa première occurrence.

:class: tip, dropdown

```python

def argmax(liste):

i\_max = None

# On initialise elem\_max à une valeur

# qui n'est clairement pas le max

if len(liste) > 0:

elem\_max = liste[0] - 1

for i, elem in enumerate(liste):

if elem > elem\_max:

i\_max = i

elem\_max = elem

return i\_max

print(argmax([1, 6, 2, 4]))

```

## Manipulations de listes

Nous présentons dans ce qui suit les opérations élémentaires de manipulation de listes.

### Insertion d'élément

Pour insérer un nouvel élément dans une liste, on peut :

* rajouter un élément à la fin de la liste à l'aide de la méthode append ;
* insérer un élément à l'indice i de la liste à l'aide de la méthode insert.

Comme vous pouvez le remarquer, il est ici question de méthodes et non plus de fonctions. Pour l'instant, sachez que les méthodes sont des fonctions spécifiques à certains objets, comme les listes par exemple. L'appel de ces méthodes est un peu particulier, comme vous pouvez le remarquer dans ce qui suit :

liste = [1, 5, 7]

liste.append(2)

print(liste)

liste.insert(2, 0) # insère la valeur 0 à l'indice 2

print(liste)

### Suppression d'élément

Si l'on souhaite, maintenant, supprimer un élément dans une liste, deux cas de figures peuvent se présenter. On peut souhaiter :

* supprimer l'élément situé à l'indice i dans la liste, à l'aide de l'instruction del ;
* supprimer la première occurrence d'une valeur donnée dans la liste à l'aide de la méthode remove.

liste = [1, 5, 7]

del liste[1] # l'élément d'indice 1 est le deuxième élément de la liste !

print(liste)

liste = [7, 1, 5, 1]

liste.remove(1) # supprime la première occurrence de 1 dans la liste

print(liste)

### Recherche d'élément

Pour trouver l'indice de la première occurrence d'une valeur dans une liste, on utilisera la méthode index :

liste = [1, 5, 7]

print(liste.index(7))

Si l'on ne cherche pas à connaître la position d'une valeur dans une liste mais simplement à savoir si une valeur est présente dans la liste, on peut utiliser le mot-clé in :

liste = [1, 5, 7]

if 5 in liste:

print("5 est dans liste")

### Création de listes composites

On peut également concaténer deux listes (c'est-à-dire mettre bout à bout leur contenu) à l'aide de l'opérateur + :

liste1 = [1, 5, 7]

liste2 = [3, 4]

liste = liste1 + liste2

print(liste)

Dans le même esprit, l'opérateur \* peut aussi être utilisé pour des listes :

liste1 = [1, 5]

liste2 = 3 \* liste1

print(liste2)

Bien entendu, vu le sens de cet opérateur, on ne peut multiplier une liste que par un entier.

### Tri de liste

Enfin, on peut trier les éléments contenus dans une liste à l'aide de la fonction sorted :

liste = [4, 5, 2]

liste2 = sorted(liste)

print(liste2)

Il est à noter que l'on peut trier une liste dès lors que celle-ci contient des éléments du même type (ou de types assimilables, par exemple des valeurs numériques, entières ou flottantes) à partir du moment où une relation d'ordre est définie sur ce type. On peut donc par exemple trier des listes de chaînes de caractères :

liste = ["a", "zzz", "c"]

print(sorted(liste))

Sachant que les émoticones sont des caractères comme les autres, on peut ainsi (enfin) obtenir une réponse à un problème vieux comme le monde :

liste = ["🐔", "🥚"]

print(sorted(liste))

### Exercices

:name: ex4.2

Écrivez une fonction qui prenne deux listes en entrée et retourne l'intersection des deux listes (c'est-à-dire une liste contenant tous les éléments présents dans les deux listes).

:class: tip, dropdown

```python

def intersection(liste1, liste2):

liste\_intersection = []

for elem in liste1:

if elem in liste2 and not elem in liste\_intersection:

liste\_intersection.append(elem)

return liste\_intersection

print(intersection([1, 6, 2, 4], [2, 7, 6]))

```

:name: ex4.3

Écrivez une fonction qui prenne deux listes en entrée et retourne l'union des deux listes (c'est-à-dire une liste contenant tous les éléments présents dans au moins une des deux listes) sans doublon.

:class: tip, dropdown

```python

def union\_sans\_doublon(liste1, liste2):

liste\_union = []

for elem in liste1 + liste2:

if elem not in liste\_union:

liste\_union.append(elem)

return liste\_union

print(union\_sans\_doublon([1, 6, 2, 4], [2, 7, 6, 2]))

```

## Copie de liste

Pour la plupart des variables, en Python, la copie ne pose pas de problème :

a = 12

b = a

a = 5

print(a, b)

Cela ne se passe pas de la même façon pour les listes. En effet, si liste est une liste, lorsque l'on écrit :

liste2 = liste

on ne recopie pas le contenu de liste dans liste2, mais on crée une variable liste2 qui va "pointer" vers la même position dans la mémoire de votre ordinateur que liste. La différence peut sembler mince, mais cela signifie que si l'on modifie liste même après l'instruction liste2 = liste, la modification sera répercutée sur liste2 :

liste = [1, 5, 7]

liste2 = liste

liste[1] = 2

print(liste, liste2)

Lorsque l'on souhaite éviter ce comportement, il faut effectuer une copie explicite de liste, à l'aide par exemple de la fonction list :

liste = [1, 5, 7]

liste2 = list(liste)

liste[1] = 2

print(liste, liste2)

## Bonus : listes en compréhension

Il est possible de créer des listes en filtrant et/ou modifiant certains éléments d'autres listes ou itérables. Supposons par exemple que l'on souhaite créer la liste des carrés des 10 premiers entiers naturels. Le code qui suit présente deux façons équivalentes de créer une telle liste :

# Façon "classique"

liste = []

for i in range(10):

liste.append(i \*\* 2)

print(liste)

# En utilisant les listes en compréhension

liste = [i \*\* 2 for i in range(10)]

print(liste)

On remarque que la syntaxe de liste en compréhension est plus compacte. On peut également appliquer un filtre sur les éléments de la liste de départ (ici range(10)) à considérer à l'aide du mot-clé if :

liste = [i \*\* 2 for i in range(10) if i % 2 == 0]

print(liste)

Ici, on n'a considéré que les entiers pairs.

## Super Bonus : le parcours simultané de plusieurs listes

Il peut arriver que l'on ait à parcourir simultanément plusieurs listes. Par exemple, supposons que l'on ait une liste stockant les prénoms d'enfants d'une classe et une autre liste contenant leurs noms. Si l'on veut afficher les noms et prénoms de ces enfants, on peut effectuer un parcours par indice :

prenoms = ["Jeanne", "Anne", "Camille"]

noms = ["Papin", "Bakayoko", "Drogba"]

for i in range(len(prenoms)):

print(prenoms[i], noms[i])

On peut aussi se dire que, en toute logique, on n'a pas réellement besoin ici d'accéder aux indices des éléments et que l'on voudrait juste effectuer un parcours simultané des deux listes. C'est ce que permet la fonction zip() :

prenoms = ["Jeanne", "Anne", "Camille"]

noms = ["Papin", "Bakayoko", "Drogba"]

for p, n in zip(prenoms, noms):

print(p, n)

On peut même parcourir plus de deux listes simultanément :

prenoms = ["Jeanne", "Anne", "Camille"]

noms = ["Papin", "Bakayoko", "Drogba"]

ages = [12, 11, 12]

for p, n, a in zip(prenoms, noms, ages):

print(p, n, a)

Bien entendu, pour pouvoir utiliser zip(), il faut que les listes soient de même taille.

## Liste des exercices de ce chapitre

1. [Argmax](about:blank)
2. [Intersection de listes](about:blank)
3. [Union de listes](about:blank)