LISTES PYTHON

À la fin de ce chapitre, je sais :

- ☞ créer une liste simplement ou en compréhension
- ☞ manipuler une liste pour ajouter, ôter ou sélectionner des éléments
- ☞ tester l'appartenance d'un élément à une liste
- ☞ utiliser la concaténation et le tronçonnage sur une liste
- ☞ itérer sur les éléments d'une liste avec une boucle for
- ☞ coder les algorithmes incontournables (count, max, min, sum, avg)

La liste Python est une collection très [1] pratique. Elle est à la base de quasiment tous les algorithmes que l'on demande d'implémenter lors des épreuves de concours. C'est une liste **muable** implémentée par un tableau dynamique.

A Constructeurs de listes

Cette section s'intéresse à la manière dont on peut créer des listes en Python.

a Crochets

À tout seigneur tout honneur, les crochets sont la voie royale pour créer une liste.

```
L = []  # Empty list
print(L, type(L)) # [] <class 'list'>
L = [1, 2, 3]
print(L) # [1, 2, 3]
L = [True, False,  False]
print(L) # [True, False, False]
L = ["cours", "td", "tp"]
print(L) # ['cours', 'td', 'tp']
L = [[2.3, 4.7], [5.9, 7.1], [1.8, 3.9]]  # Nested List !
print(L) # [[2.3, 4.7], [5.9, 7.1], [1.8, 3.9]]
```

L'exemple précédent montre que :

1. trop?

- On peut créer des listes de n'importe quel type de donnée. Une liste est un conteneur avant toute chose.
- On peut créer des listes imbriquées (nested list), c'est à dire des listes de listes. Ces dernières sont très appréciées par les créateurs d'épreuves de concours.

b Constructeur `list`

La fonction `list()`[2] permet également de créer une liste à partir de n'importe quel objet itérable. Elle s'utilise le plus souvent pour convertir un autre objet itérable[3] en liste.

```
L = list()  # Empty list
print(L, type(L)) # [] <class 'list'>
L = list("Coucou !")
print(L) # ['C', 'o', 'u', 'c', 'o', 'u', ' ', '!']
L = list(range(10))
print(L) #  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

---

c Construire une liste en compréhension

Tout comme les ensembles en mathématiques, les listes peuvent être construites à partir d'une description compréhensible des éléments de la liste. Cette méthode de création de liste est à rapprocher du paradigme fonctionnel.

```
L = [ i for i in range(10) ]
print(L) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
L = [ i for i in range(10) if i %2 == 0]
print(L) # [0, 2, 4, 6, 8]
```

P Cette méthode de création de liste est puissante mais est très délicate à manipuler. C'est pourquoi il est préférable de ne l'utiliser que si on est vraiment sûr de soi, sinon c'est une perte de points assurée au concours. On peut l'éviter avec une boucle `for` et un `append` et ainsi assurer des points.

STOP Le paragraphe précédent est important pour l'épreuve d'informatique!

B Opérations sur une liste

a Longueur d'une liste

La fonction `len` renvoie la longueur d'une séquence. Elle s'utilise donc aussi pour les listes.

```
L = [ 1, 2, 3 ]
print(len(L)) # 3
```

---

2. On appelle cette fonction le **constructeur** des objets de type `list`.
3. un dictionnaire ou une chaîne de caractères par exemple

P En Python, on peut tester si une liste est vide avec la fonction `len`.

```
L = [ 1, 2, 3]
while len(L) > 0:
    print(L.pop()) # 3 2 1
```

---

b Appartenance à une liste

Les mots-clefs `in` et `not in` permettent de tester l'appartenance à une liste et renvoient les booléens correspondants.

```
L = [ 1, 2, 3]
print(2 in L) # True
print(42 in L) # False
print(42 not in L) # True
```

---

c Ajouter un élément à une liste

La méthode `append` de la classe `list` permet d'ajouter un élément à la fin d'une liste. Cette méthode modifie la liste.

```
L = [ 1, 2, 3]
L.append(4)
print(L) # [1, 2, 3, 4]
L.append(5)
print(L) # [1, 2, 3, 4, 5]
```

---

d Retirer le dernier élément d'une liste

La méthode `pop` de la classe `list` permet de retirer le dernier élément ajouté.

```
L = [ 1, 2, 3 ]
L.pop()
print(L) # [1, 2 ]
```

C Concaténation et démultiplication de listes

L'opérateur + permet de concaténer une liste à une autre liste pour en créer une nouvelle, c'est à dire de créer une nouvelle liste en fusionnant leurs éléments.

```
L = [1, 2, 3]
M = [4, 5, 6]
N = L + M
print(N) # [1, 2, 3, 4, 5, 6]
```

L'opérateur $*$ permet de démultiplier un élément dans une liste.

```
L = [1] * 10
print(L) # [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Il existe également l'opérateur de concaténation et d'affectation `+=` ainsi que l'opérateur de démultiplication et d'affectation `*=`.

```
L = [1, 2]
L += [3]
L *= 4
print(L) # [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

D Des listes indiçables

L'intérêt des listes Python est qu'elles permettent d'accéder à un élément particulier en temps constant[4]. Pour cela, il suffit de connaître l'indice de l'élément dans la liste. Naturellement, comme expliqué dans le cours précédent :

- le premier élément d'une liste est l'élément d'indice `0`,
- le dernier élément d'une liste est l'élément d'indice `len(L)-1`.

Tout comme les chaînes de caractères, les listes autorisent également les indices négatifs pour identifier un élément à partir de la fin de la chaîne.

```
L = [ 1, 2, 3 ]
print(L[0]) # 1
print(L[len(L) - 1])  # 3
print(L[-2]) # 2
```

E Des listes itérables

Une liste Python est itérable, c'est à dire qu'elle peut être l'objet d'une itération via une boucle `for`.

```
L = [ 1, 2, 3 ]
for elem in L:
    print(elem)     # 1 2 3   elem
```

Naturellement, on peut parcourir une liste d'après ses indices :

```
L = [ 1, 2, 3 ]
for i in range(len(L)):
    print(L[i])       # 1 2 3
```

L'intérêt de la première syntaxe est qu'on ne peut pas se tromper sur les indices. Le premier inconvénient est que l'on ne peut pas modifier l'élément `elem`. Le second inconvénient est qu'on ne dispose pas l'indice alors qu'on pourrait en avoir besoin...

4. C'est parce qu'elles sont implémentées par des tableaux dynamiques.

On choisira donc l'une ou l'autre syntaxe selon qu'il est nécessaire ou pas de disposer de l'indice ou de modifier les éléments.

F Des listes tronçonnables

Vocabulary 1 — Slicing ⇜⇝ Tronçonnage

Tout comme les chaînes de caractères, les listes sont tronçonnables.

```
L =  [1, 2, 3, 4, 5, 6]
print(L[1:3])   # slicing start stop ⟶ [2, 3]
print(L[-3:-1]) # negative slicing ⟶ [4, 5]
print(L[::-1])  # reverse list ⟶ [6, 5, 4, 3, 2, 1]
print(L[0:-1:2])# slicing start stop step  ⟶ [1, 3, 5]
```

G Les algorithmes simples mais incontournables

Les listes Python sont très souvent utilisées pour agréger des éléments différents d'un même type. Lors d'un traitement automatisé des données, il est naturel de vouloir :

- compter les éléments d'un certain type,
- trouver le maximum ou le minimum des éléments s'il y en a un, ou trouver l'indice de cet extremum,
- calculer une somme, une moyenne ou un écart type, si les éléments de la liste sont numériques.

Le code **??** implémente ces algorithmes incontournables en Python.

Code 1 – Algorithmes simples et incontournables

```
def count_elem_if(L, value):
    c = 0
    for elem in L:
        if elem == value:
            c += 1
    return c


def max_elem(L):
    if len(L) > 0:
        m = L[0]
        for elem in L:
            if elem > m:
                m = elem
        return m
    else:
        return None
```

```
def min_elem(L):
    if len(L) > 0:
        m = L[0]
        for elem in L:
            if elem < m:
                m = elem
        return m
    else:
        return None


def sum_elem(L):
    acc = 0
    for elem in L:
        acc += elem
    return acc


def avg_elem(L):
    if len(L) > 0:
        acc = 0
        for elem in L:
            acc += elem
        return acc / len(L)
    else:
        return None

if __name__=="__main__":
    L = ["words", "letters", "words", "sentences", "words"]
    print(count_elem_if(L, "words"))  # 3
    L = [ 13, 19, -7, 23, -29, 5, -3, 41]
    print(max_elem(L), min_elem(L), sum_elem(L), avg_elem(L))  # 41 -29 62 7.75
    L = []
    print(max_elem(L), min_elem(L), sum_elem(L), avg_elem(L))  # None None 0 None
    L = list("Anticonstitutionnellement")
    print(max_elem(L), min_elem(L))  # u A
```

P En Python, il existe les fonctions :

- `sum` pour calculer la somme des éléments d'une liste,
- `max` et `min` pour trouver le maximum et le minimum d'une liste.

```
L =[1, 2, 3]
print(sum(L), max(L), min(L)) # 6, 3, 1
```

Avant de les utiliser, lisez attentivement le sujet de l'épreuve. Si elles ne sont pas autorisées explicitement, ne pas les utiliser.

STOP Le paragraphe précédent est important pour l'épreuve d'informatique!

H Listes, copies et références

En manipulant les listes, il faut rester vigilant lorsqu'on souhaite partager des éléments ou copier des éléments. Il faut avoir en mémoire la règle d'or Python [5].

P L'affection simple d'une liste à une autre ne recopie pas les éléments de la liste mais copie la référence de la liste. Sur l'exemple suivant, M et L désigne le même objet en mémoire. Si on modifie l'un, on modifie l'autre.

```
L = [1, 2, 3]
M = L
print(id(M) == id(L)) # True
M[0] = 42
print(L) # [42, 2, 3]
```

Il n'y a donc pas recopie des éléments dans ce cas.

Par contre, l'instruction suivante permet de recopier une liste, c'est à dire dupliquer ses éléments en mémoire. On dispose alors de deux références vers deux objets différents. L'un ne modifie pas l'autre.

```
L = [1, 2, 3]
M = L[:]
print(id(M)== id(L)) # False
M[0] = 42
print(L) # [1, 2, 3]
```

I Listes, paramètres et fonctions

Cette section s'intéresse au cas où une liste est un paramètre d'une fonction. Que se passe-t-il dans ce cas ?

La liste étant une séquence muable, lorsqu'une fonction utilise une liste qu'elle a reçue en paramètre, les opérations sont directement effectuées sur la liste en mémoire comme le montre le code **??**. Ceci explique pourquoi on n'a pas besoin de renvoyer une liste modifiée par une fonction si celle-ci a été transmise en paramètre. **C'est ce qu'on appelle un passage par référence d'un type muable.**

Comme le montre l'exemple **??**, pour une variable immuable comme un `int`, le passage en paramètre n'accorde pas plus de droits sur l'objet : celui-ci est toujours immuable. Donc, d'autres références sont crées pour enregistrer les calculs. Si la dernière référence créée contenant le résultat n'est pas renvoyée par la fonction en utilisant `return`, le calcul est perdu.

On fera donc attention à utiliser `return` lorsqu'il le faut !

Code 2 – Passage en paramètre d'un type muable et d'un type immuable à une fonction

```
def f(L):
```

5. Toute variable Python est une référence vers un objet en mémoire.

```
    print("inside f start :", id(L), L)
    for i in range(len(L)):
        L[i] = L[i] + 100
    print("inside f end :", id(L), L)


def g(a):
    print("inside g start :", id(a), a)
    for i in range(3):
        a = 10 * a
    print("inside g end : ", id(a), a)
    # return a # ---> on aurait dû décommenter cette ligne !


# TESTS
M = [1, 2, 3]
f(M)
print("main M", id(M), M)

b = 2
g(b)
print("main b", id(b), b)
```

Voici le résultat de l'exécution de ce code :

```
inside f start : 4374619648 [1, 2, 3]
inside f end : 4374619648 [101, 102, 103]
main M 4374619648 [101, 102, 103]
inside g start : 4373872912 2
inside g end :  4373887856 2000
main b 4373872912 2
```

J Tuples

Les tuples Python sont des séquences immuables que l'on construit avec les parenthèses `()`. On peut les manipuler comme des listes : ils sont indiçables, itérables et tronçonnables. Il ne faut juste pas tenter de modifier leurs éléments car cela n'est pas possible !

```
T = (1, 2, 3, 4, 5)

for elem in T:
    print(elem)

for i in range(len(T)):
    print(T[i])

print(T[-1])
print(T[::-1])
print(T[1:4:2])
```