Université Paris Dauphine **Licence MIE 1re année**
Algorithmique et programmation 1 **2025-2026**

6. Listes en Python et boucle `for` (cours)

Le type tableau, tel qu'il a été introduit en pseudo-code, n'existe pas en Python. Les listes constituent le type en Python qui permet de représenter une séquence. Ce type est noté `list` en Python. Il a pour particularité de contenir de manière ordonnée plusieurs éléments, ces éléments pouvant être de types différents. C'est un des types de base de Python (tout comme `int`, `float`, `str`...).

6.1 Création de liste en Python, affichage et taille

En Python, on définit une liste avec les crochets `[]` et les éléments sont séparés par une virgule.
On peut créer :

— une liste contenant un nombre donné d'éléments de même type :

```
L1 = [1, 15, -8]
L2 = ["bob", "lulu"]
```

— une liste contenant un nombre donné d'éléments de types différents :

```
L3 = ["Python", 76, 3.8, False, "XXI"]
L4 = ['a', [2, True], 22]
```

— le cas particulier d'une liste d'entiers avec la fonction `range()` :

⇒ `range(n)` crée une séquence de `n` valeurs allant de 0 à `n-1`. Pour obtenir une valeur de type `list` en utilisant la fonction `range()`, il faut convertir explicitement la séquence d'entiers renvoyée par la fonction en une valeur de type `list`.
Par exemple,

```
L5 = list(range(5))
```

va créer une variable `L5` de type `list` de valeur $[0, 1, 2, 3, 4]$.

⇒ `range(m,n)` crée une séquence de `n-m` valeurs allant de `m` à `n-1`.
Par exemple,

```
L6 = list(range(3,7))
```

va créer une variable `L6` de type `list` de valeur $[3, 4, 5, 6]$.

⇒ `range(m,n,p)` crée la séquence des valeurs allant de `m` à `n` non compris en utilisant un pas de `p` > 0 pour passer d'une valeur à l'autre : `m, m+p, m+2p, ..., m+kp` avec `m+kp` le plus grand entier strictement inférieur à `n`.
Par exemple,

```
L7 = list(range(1,9,2))
```

va créer une variable `L7` de type `list` de valeur $[1, 3, 5, 7]$.

⇒ `range(n,m,p)` crée la séquence des valeurs allant de `n` à `m` non compris en utilisant un pas de `p` < 0 pour passer d'une valeur à l'autre : `n, n+p, n+2p, ..., n+kp` avec `n+kp` le plus petit entier strictement supérieur à `m`.

Par exemple,
```
L8 = list(range(9,6,-1))
```
va créer une variable `L8` de type `list` de valeur $[9, 8, 7]$.

— une liste vide avec l'instruction suivante :
```
L9 = []
```

Il est ensuite possible d'afficher la totalité du contenu d'une variable de type liste avec la commande `print()` de la façon suivante :

```
print(L)
```

La taille d'une liste `L` (qui est le nombre d'éléments de cette liste) peut se récupérer à l'aide de la fonction

```
len(L)
```

qui va renvoyer la longueur de la liste `L`.

6.2 Accès aux éléments et modification d'un élément

On accède à un élément de la liste par son indice, tout comme avec les tableaux. Les indices commencent à 0 et finissent à `len(L)-1` et doivent nécessairement être des entiers. Pour accéder à l'élément correspondant à l'indice `i` dans la liste `L`, on fait :

```
L[i]
```

Attention : si on tape cette instruction pour un indice `i` positif en dehors de l'intervalle des indices de la liste (c'est-à-dire $[0, \text{len}(L) - 1]$) alors un message d'erreur apparaît.

Les indices négatifs sont autorisés : si on met `L[-i]`, pour `i` allant de 1 à `len(L)` alors on récupère l'élément de la liste `L` ayant pour indice $\text{len}(L) - i$. Par exemple, `L[-1]` renvoie l'élément en dernière position.

indices négatifs	$-len(L)$	$-len(L)+1$		-2	-1
`L`	élément 1	élément 2	...	élément $n-1$	élément n
indices positifs	0	1		$len(L)-2$	$len(L)-1$

Si une valeur dans une liste est elle-même une séquence, on peut appliquer l'opérateur d'indiçage récursivement :

```
>>> L = ["Bob", 76, [2, 3]]
>>> L[0][2]
  'b'
>>> L[2][1]
  3
>>> L[1][0]
  Traceback (most recent call last):
    File "<console>", line 1, in <module>
  TypeError: 'int' object is not subscriptable
```

À l'opposé des chaînes de caractères qui ne sont pas modifiables, les listes sont **modifiables**. Pour modifier l'élément qui est à l'indice `i` dans une liste, il suffit d'affecter à `L[i]` la nouvelle valeur : `L[i] = expression`.

Exemple
Expliquer ce qui se passe lorsque l'on exécute le programme suivant :

```
L = ["Python", 76, 3.8, False, "XXI"]
print(L[4])
L[4] = 21
print(L)
```

S'il est possible de modifier un élément d'une liste, il n'est pas possible d'ajouter des éléments à une liste de la façon suivante :

```
L = ['a', 'b', 'c']
L[3] = 'd'
```

L'exécution de ces deux instructions déclenche l'erreur suivante :

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 2, in <module>
    L[3] = 'd'
IndexError: list assignment index out of range
```

6.3 Parcours

On peut parcourir les listes avec une boucle `while`.

Exemple

```
L = ["Zoe", "Thomas", "Manon"]
i = 0
while (i < len(L)):
    print('longueur de la chaîne ', L[i], "=", len(L[i]))
    i += 1
```

En Python, une liste est un **itérable**. Un itérable est une séquence de valeurs sur laquelle on peut itérer, c'est-à-dire parcourir, à l'aide d'une nouvelle instruction répétitive en Python, appelée la boucle `for`. Cette boucle `for` généralise la boucle `for` du pseudo-code.

Étant donnée `L` de type `list`, la boucle `for` ci-dessous permet de parcourir les éléments de `L` :

```
for elem in L:
    BlocInstructions
```

Dans cette instruction, la variable `elem` est initialisée avec le premier élément de la liste et l'ensemble des instructions contenues dans `BlocInstructions` est exécuté. La variable `elem` passe ensuite à l'élément suivant de la liste. Si on était arrivé au dernier élément de la liste, alors le programme sort de la boucle, sinon, de nouveau on exécute l'ensemble des instructions contenues dans `BlocInstructions` avec la nouvelle valeur de `elem`.

Exemple

```
L = ["Zoe", "Thomas", "Manon"]
for prenom in L:
    print('longueur de la chaîne', prenom, "=", len(prenom))
```

Il est important de remarquer que le type des valeurs référencées par `elem` peut varier puisqu'une liste peut contenir des éléments de types différents.

Exemple

```
L = [True, "Bob", 5.8]
for elem in L:
    print("le type de l'élément", elem, "est", type(elem))
```

La boucle `for` permet donc de parcourir n'importe quel itérable. Or, une chaîne de caractères est également un itérable. C'est pour cette raison qu'on peut utiliser la boucle `for` pour parcourir les caractères d'une chaîne !

Itérer sur les éléments d'une chaîne avec boucle `for`
Étant donnée `ch` de type `str`, l'instruction de la boucle `for` sur les éléments de la chaîne est :

```
for car in ch:
    BlocInstructions
```

Dans cette instruction, la variable `car` est initialisée avec le premier caractère de la chaîne et l'ensemble des instructions contenues dans `BlocInstructions` est exécuté. La variable `car` passe ensuite au caractère suivant de la chaîne. Si on était arrivé au dernier caractère, alors le programme sort de la boucle, sinon, de nouveau on exécute l'ensemble des instructions contenues dans `BlocInstructions` avec la nouvelle valeur de `car`.

Exemple

```
ch = "abcd"
for car in ch:
    print(car)
```

La fonction `range()` renvoie aussi un itérable. On peut alors itérer sur les indice d'une liste `L` en utilisant la boucle `for` et la fonction `range()` de la façon suivante :

```
for i in range(len(L)):
    BlocInstructions
```

Rappelons que `range(len(L))` crée une séquence d'entiers de 0 à $len(L) - 1$ correspondant donc à la séquence des indices de `L`. Dans cette boucle `for`, la variable `i` est initialisée au premier élément de cette séquence (donc 0, le premier indice) et l'ensemble des instructions de `BlocInstructions` est exécuté. À l'itération suivante, la variable `i` prend la valeur de l'élément suivant de la séquence (à savoir 1) et l'ensemble des instructions contenues dans `BlocInstructions` est exécuté, et ainsi de suite. La dernière itération correspond au dernier élément de la séquence ($len(L)-1$) et une fois exécutée, on sort de la boucle. Cette version est similaire à la boucle `for` du pseudo-code.

Exemple : reprenons le premier exemple écrit avec une boucle `while` et utilisons la boucle `for` pour obtenir le même résultat :

```
L = ["Zoe", "Thomas", "Manon"]
for i in range(len(L)):
    print('longueur de la chaîne', L[i], "=", len(L[i]))
```

Remarque : Nous pouvons aussi parcourir une liste L de la fin vers le début (du dernier élément vers le premier) en utilisant la boucle `for` et la fonction `range()` de la façon suivante :

```
for i in range(len(L)-1,-1,-1):
     BlocInstructions
```

Exemple :

```
L = ["Zoe", "Thomas", "Manon"]
for i in range(len(L)-1,-1,-1):
    print('longueur de la chaîne', L[i], "=", len(L[i]))
```

génère l'affichage suivant :

```
longueur de la chaîne Manon = 5
longueur de la chaîne Thomas = 5
longueur de la chaîne Zoe = 3
```

Itérer sur les indice d'une chaîne
De la même façon, on peut itérer sur les indices d'une chaîne `ch` :

```
for i in range(len(ch)):
    BlocInstructions
```

Exemple

```
ch = "abcd"
for i in range(len(ch)):
    print(ch[i])
```

6.4 Méthodes de liste

Nous avons déjà vu des fonctions définies sur des listes, comme `len(L)`. Il existe de plus de nombreuses méthodes de liste en Python. Tout comme les chaînes de caractères, une méthode de liste s'appelle sur une liste existante sous la forme :

```
    liste.Nom_Methode([argument])
```

[argument] signifie que les arguments sont optionnels.

Nous allons ici vous présenter les principales méthodes, que vous aurez le droit d'utiliser explicitement lors des examens, sauf indication contraire. Seules les méthodes présentées en cours pourront être utilisées. Pour connaître l'ensemble des méthodes Python associées au type `list` faire `dir(list)`, et pour connaître le détail d'une méthode spécifique faire :
`help(list.Nom_Methode)`.

Étant données une variable `L` de type `list`, une variable `ele` de type quelconque et une variable `indice` de type `int` :

6.4.1 Méthodes ne renvoyant rien

— `L.append(ele)` : ajoute (en dernière position) dans la liste `L` l'élément `ele`.
```
>>> L = [1,2,3]
>>> L.append("Python")
>>> print(L)
  [1, 2, 3, 'Python']
```

— `L.remove(ele)` : supprime l'élément `ele` dans la liste `L`. Si l'élément apparaît plusieurs fois dans la liste, c'est la première occurrence qui est supprimée et si l'élément n'apparaît pas, alors la liste n'est pas modifiée et un message d'erreur est affiché. Après cette exécution, les indices des éléments restants ont changé (ceux après la première occurrence supprimée).

— `L.insert(indice,ele)` : insère l'élément `ele` à l'indice donné.

— `L.reverse()` : modifie la liste `L` en inversant l'ordre des éléments.

— `L.sort()` : modifie la liste `L` en la triant par ordre croissant (si tous les éléments sont de même type).
```
>>> L = [9,5,1,7]
>>> L.sort()
>>> print(L)
  [1, 5, 7, 9]
```

— `L.extend(L2)` : modifie la liste `L` en lui concaténant la liste `L2`.
```
>>> L = [1, 2, 3]
>>> L2 = [4, 5]
>>> L.extend(L2)
>>> print(L)
  [1, 2, 3, 4, 5]
```

— `random.shuffle(L)` : méthode du module `random` qui réarrange aléatoirement les éléments de la liste `L` selon une distribution uniforme sur toutes les permutations possibles.

Attention
Contrairement aux chaînes, les méthodes de listes modifient la liste elle-même sans en renvoyer de nouvelle. Illustrons-le à travers le code qui suit.

```
>>> liste1 = [3, 1, 8]
>>> liste2 = liste1.append(7)
>>> print(liste1)
```

```
    [3,1,8,7]
>>> print(liste2)
    None
```

L'application de la méthode `append()` sur `liste1` ne renvoie rien, représenté en Python par le mot-clé `None`. Donc la variable `liste2` est initialisée à cette valeur ; par contre `liste1` a elle bien été modifiée.

6.4.2 Méthodes et fonctions renvoyant une valeur

- `L.pop()` : supprime le dernier élément de la liste `L` et renvoie cet élément supprimé. Si la liste `L` était vide, alors elle affiche un message d'erreur. On peut également préciser l'indice de l'élément que l'on veut supprimer en exécutant `L.pop(indice)`.
- `L.count(ele)` : compte le nombre d'occurrences de l'élément `ele`.
- `L.index(ele)` : renvoie l'indice de `ele` dans la liste `L`. Si `ele` apparaît plusieurs fois dans la liste, elle renvoie le premier indice et si `ele` n'apparaît pas dans la liste alors elle affiche un message d'erreur.
- `max(L)` : renvoie l'élément le plus grand de `L`. La liste `L` ne doit pas être vide, et ses éléments doivent être comparables entre eux.
- `min(L)` : renvoie l'élément le plus petit dans la liste `L`. La liste `L` ne doit pas être vide, et ses éléments doivent être comparables entre eux.
- `random.choice(L)` : méthode du module `random` qui renvoie un élément choisi aléatoirement et uniformément parmi les éléments de la liste `L`.

6.5 La compréhension de listes

Pour créer une liste, on doit donc utiliser une méthode de list et une instruction répétitive. Par exemple :

```
n = 10
nouvelleListe = []
for i in range(10):
    nouvelleListe.append(int(input()))
```

Il existe une méthode alternative en Python pour créer une liste : c'est la **compréhension de listes**. La syntaxe est la suivante

```
nouvelleListe = [expression for ele in iterable if condition]
```

La partie conditionnelle introduite avec le mot-clé `if` est optionnelle.

Pour notre premier exemple cela donne :

```
nouvelleListe = [int(input()) for i in range(10)]
```

Exemples :

```
>>> a = [1,4,2,7,1,9,0,3]
>>> l1 = [x for x in a if x>5]
>>> l1
[7, 9]
>>> l2 = [a+b for a in "abc" for b in "de"]
>>> l2
['ad', 'ae', 'bd', 'be', 'cd', 'ce']
```

Pour créer une liste de listes initialisée avec des zéros, on peut écrire

```
mat=[[0 for i in range(10)] for j in range(10)]
```