Listes - collectes de données

La liste : list

- La liste (ou list) en python est une variable dans laquelle on peut mettre plusieurs variables (à valeurs multiples) : C'est une collection d'éléments, mais chaque élément est un scalaire. C'est un objet.
- Déclaration :

nomList=[v1, v2,……]

exp : numbers = [10, 5, 7, 2, 1] , numbers est une liste de 5 éléments ; long=5

- Les éléments d'une liste peuvent avoir différents types : des nombres entiers, des flottants, autres listes.
- Les éléments d'une liste sont toujours numérotés à partir de zéro

L'indexation de liste

La valeur entre crochets qui sélectionne un élément de la liste est appelée **index** , tandis que l'opération de sélection d'un élément dans la liste est appelée **indexation** : c'est l'accès aux éléments de la liste.

Exp1:
numbers[2]==➔7

Exp2 :

```
numbers = [10, 5, 7, 2, 1]
print("liste initiale : ", numbers)
numbers[0] = 111
numbers[1] = numbers[4]
print("nouvelle liste :", numbers)
```

Exécution :

liste initiale : [10, 5, 7, 2, 1]

après modification :

nouvelle liste : [111, 1, 7, 2, 1]

La fonction len () :

La longueur d'une liste peut varier pendant l'exécution. De nouveaux éléments peuvent être ajoutés à la liste, tandis que d'autres peuvent en être supprimés. Cela signifie que la liste est une entité très dynamique.

La fonction « len » prend le nom de la liste comme argument et retourne le nombre d'éléments actuellement stockés dans la liste (en d'autres termes - la longueur de la liste).

Exp :

```
numbers = [10, 5, 7, 2, 1]
print(len(numbers)) ==========➔ 5
```

L’instruction del :

N'importe quel élément de la liste peut être supprimé à tout moment - cela se fait avec une instruction nommée del (supprimer).
Remarque: c'est une instruction, pas une fonction.

Exp :

```
numbers = [10, 5, 7, 2, 1]
print("liste initiale : ", numbers)
numbers[0] = 111
numbers[1] = numbers[4]
print("nouvelle liste :", numbers)
print(len(numbers))
del numbers[1]
print(numbers)

print(len(numbers))

print(numbers[4]) #erreur
```

Exécution :

```
liste initiale :  [10, 5, 7, 2, 1]

nouvelle liste : [111, 1, 7, 2, 1]



5

[111, 7, 2, 1]

4

Traceback (most recent call last):

  File "<string>", line 10, in <module>

IndexError: list index out of range
```

L’indexation négative :

Les indices négatifs sont légaux Cela peut sembler étrange, mais les indices négatifs sont légaux et peuvent être très utiles. Un élément avec un indice égal à -1 est le dernier de la liste.

Exp: numbers = [10, 5, 7, 2, 1] print(numbers) print(numbers[-1]) print(numbers[-2]) print(numbers[-3]) print(numbers[-4]) print(numbers[-5])	Exécution : [10, 5, 7, 2, 1] 1 2 7 5 10

LAB: Les bases des listes

Il y avait une fois un chapeau. Le chapeau ne contenait pas de lapin, mais une liste de cinq numéros : 1, 2, 3, 4 et 5.
Votre tâche consiste à:

- écrire une ligne de code qui invite l'utilisateur à remplacer le numéro du milieu de la liste par un nombre entier entré par l'utilisateur.
- écrire une ligne de code qui supprime le dernier élément de la liste.
- écrire une ligne de code qui imprime la longueur de la liste existante.

Exp de solution : hatList = [1, 2, 3, 4, 5] print(hatList) lg=len(hatList) print("lg =", lg) milieu=lg//2 hatList[milieu]=int(input("entrer un nombre : ")) print(hatList) print(len(hatList)) del hatList[-1] print(hatList) print(len(hatList))	Exécution : [1, 2, 3, 4, 5] lg = 5 entrer un nombre : 100 [1, 2, 100, 4, 5] 5 [1, 2, 100, 4] 4

Fonctions	Méthodes
• Une fonction n'appartient à aucune donnée - elle obtient des données, elle peut créer de nouvelles données et elle (généralement) produit un résultat. • Une fonction appartient à tout le code. En général, un appel de fonction typique peut ressembler à ceci: result = function(arg) La fonction prend un argument, fait quelque chose et renvoie un résultat.	• Une méthode est un type spécifique de fonction - elle se comporte comme une fonction et ressemble à une fonction, mais diffère dans la façon dont elle agit et dans son style d'invocation. • Une méthode fait tout cela, mais est également capable de changer l'état d'une entité sélectionnée. • Une méthode appartient aux données pour lesquelles elle fonctionne. Cela signifie également que l'invocation d'une méthode nécessite une spécification des données à partir desquelles la méthode est invoquée. Un appel de méthode typique ressemble généralement à ceci: result = data.method(arg) Remarque: le nom de la méthode est précédé du nom des données propriétaires de la méthode. Ensuite, vous ajoutez un point « . » suivi du nom de la méthode et d'une paire de parenthèses entourant les arguments . La méthode se comportera comme une fonction, mais peut faire quelque chose de plus - elle peut changer l'état interne des données à partir desquelles elle a été invoquée.

Listes de Méthodes

Les listes sont des objets qui possèdent quelques méthodes très pratiques pour les manipuler:

Append()

Grace à cette méthode un nouvel élément est collé à la fin de la liste existante :

MyList.append(x) : ajoute l'occurrence ***x*** en fin de liste.

La longueur de la liste augmente alors d'une unité.

Exp :	Exécution :
MyList = [5, 0, 3, 9, 4, 3, 7] print(MyList) print("nb élément : ", len(MyList)) MyList.append(15) print(MyList) print("nb élément après append: ", len(MyList))	[5, 0, 3, 9, 4, 3, 7] nb élément : 7 [5, 0, 3, 9, 4, 3, 7, 15] nb élément après append: 8

Exp :	Exécution :
myList = [] print("lg de liste vide : ", len(myList)) for i in range(5): myList.append(i + 1) print(myList) print("lg de liste après remplissage : ", len(myList))	lg de liste vide : 0 [1, 2, 3, 4, 5] lg de liste après remplissage : 5

insert()

La méthode insert() est un peu plus intelligente ; elle peut ajouter un nouvel élément à n'importe quel endroit de la liste , pas seulement à la fin.

MyList.insert(i,x) : insère l'occurrence x à la position i.

Remarques :

Il faut deux arguments:

- Le premier indique l'emplacement requis de l'élément à insérer; Alors tous les éléments existants qui occupent des emplacements à droite du nouvel élément (y compris celui à la position indiquée) sont décalés vers la droite, afin de faire de la place pour le nouvel élément;
- Le second est l'élément à insérer.

La longueur de la liste augmente alors d'une unité comme dans le cas de la méthode append()

Exp :	Exécution :
numbers = [111, 7, 2, 1]	
print(numbers)	[111, 7, 2, 1]
print("avant insert ; lg= ",len(numbers))	avant insert ; lg= 4
numbers.insert(0, 222)	
print(numbers)	[222, 111, 7, 2, 1]
print("après insert; lg= ",len(numbers))	après insert; lg= 5

Exp :	Exécution :
myList = []	
print("lg de liste vide : ", len(myList))	lg de liste vide : 0
for i in range(10):	
myList.insert(i,2*i)	
print(myList)	[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
print("lg de liste après remplissage : ", len(myList))	lg de liste après remplissage : 10

Exp :
 	
myList = []
 	
print("lg de liste vide : ", len(myList))
 	
for i in range(10):
 	
  myList.insert(0,2*i) // liste inversée
 	
print(myList)
 	
print("lg de liste après remplissage : ", len(myList))
	Exécution :
	
	lg de liste vide : 0
	
	[18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
	
	lg de liste après remplissage : 10

Utilisations de liste	
Exp : calcul de la somme des éléments d'une liste En utilisant la boucle for : myList = [10, 1, 8, 3, 5] total = 0 print(myList) for i in range(len(myList)): total += myList[i] print("La somme des éléments de la liste :", total)	Exécution : [10, 1, 8, 3, 5] La somme des éléments de la liste : 27
Exp : calcul de la somme des éléments d'une liste En utilisant in : myList = [10, 1, 8, 3, 5] print(myList) total = 0	Exécution : [10, 1, 8, 3, 5]

for i in myList: total += i print("La somme des éléments de la liste :", total)	La somme des éléments de la liste : 27
Exp : permutation des éléments d'une liste myList = [10, 1, 8, 3, 5] print("liste initial : ", myList) myList[0], myList[4] = myList[4], myList[0] myList[1], myList[3] = myList[3], myList[1] print("liste finale, après permutation : ",myList)	Exécution : liste initial : [10, 1, 8, 3, 5] liste finale, après permutation : [5, 3, 8, 1, 10]
Exp : permutation (bis) myList = [10, 1, 8, 3, 5,20,14,9] print(myList) lg = len(myList) print("lg = ",lg) for i in range(lg // 2): myList[i], myList[lg - i - 1] = myList[lg - i - 1], myList[i] print(myList)	Exécution : [10, 1, 8, 3, 5, 20, 14, 9] lg = 8 [9, 14, 20, 5, 3, 8, 1, 10]

LAB: Les bases des listes - les Beatles

• étape 1: créez une liste vide nommée beatles;

• étape 2: utiliser la méthode append() pour ajouter les membres suivants de la bande à la liste: "John Lennon", "Paul McCartney", et "George Harrison";

• étape 3: utilisez la boucle for et la méthode append() pour inviter l'utilisateur à ajouter les membres suivants de la bande à la liste:, "Stu Sutcliffe" et "Pete Best";

• étape 4: utilisez l'instruction del pour supprimer "Stu Sutcliffe" et "Pete Best" de la liste;

• étape 5: utilisez la méthode insert() pour ajouter "Ringo Starr" au début de la liste.

Exemple de solution :

```
#Etape1 :

Beatles = [ ]

print("Etape1 : ",Beatles)

#Etape2 :

Beatles.append("John Lennon")

Beatles.append("Paul McCartney")

Beatles.append("George Harrison")

print("Etape2 : ",Beatles)

#Etape3 :

for members in range(2):

    Beatles.append(input("entrer un membre : "))

print("Etape3 : ",Beatles)

#Etape4 :
```

```
del Beatles[-1]

del Beatles[-1]

print("Etape4 : ",Beatles)

#Etape5 :

Beatles.insert(0,"Ringo Starr")

print("Etape5 : ",Beatles)
```

Exécution :

```
La liste dans Etape1 :  []

La liste dans Etape2 :  ['John Lennon', 'Paul McCartney', 'George Harrison']

Entrer un membre : Stu Sutcliffe

Entrer un membre : Pete Best

La liste dans Etape3 :  ['John Lennon', 'Paul McCartney', 'George Harrison', 'Stu Sutcliffe', 'Pete Best']

La liste dans Etape4 :  ['John Lennon', 'Paul McCartney', 'George Harrison']

La liste dans Etape5 :  ['Ringo Starr', 'John Lennon', 'Paul McCartney', 'George Harrison']
```

Points clés à retenir

1. La **liste est un type de données** en Python utilisé pour **stocker plusieurs objets**. Il s'agit d'une collection **ordonnée et modifiable** d'éléments séparés par des virgules entre crochets, par exemple:

```
myList = [1, None, True, "I am a string", 256, 0]
```

2. Les listes peuvent être **indexées et mises à jour**, par exemple:

```
myList = [1, None, True, 'I am a string', 256, 0]

print(myList[3])   # outputs: I am a string

print(myList[-1])  # outputs: 0

myList[1] = '?'

print(myList)  # outputs: [1, '?', True, 'I am a string', 256, 0]

myList.insert(0, "first")

myList.append("last")

print(myList)    # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']
```

3. Les listes peuvent être **imbriquées**,

par exemple: `myList = [1, 'a', ["list", 64, [0, 1], False]]`.

Vous en apprendrez plus sur l'imbrication dans le module 3.1.7 - pour le moment, nous voulons juste que vous soyez conscient que quelque chose comme cela est également possible.

4. Les éléments de liste et les listes peuvent être **supprimés**, par exemple:

```
myList = [1, 2, 3, 4]

del myList[2]

print(myList)        # outputs: [1, 2, 4]

del myList   # supprime toute la liste
print(myList) =➔ provoque une erreur, car l'objet liste est détruite
```

Encore une fois, vous en apprendrez plus à ce sujet dans le module 3.1.6 - ne vous inquiétez pas. Pour le moment, essayez simplement de tester le code ci-dessus et vérifiez comment sa modification affecte la sortie.

5. Les listes peuvent être **itérées** en utilisant la boucle `for`, par exemple:

```
myList = ["white", "purple", "blue", "yellow", "green"]
```

```
for color in myList:
    print(color)
```

6. La fonction `len()` peut être utilisée pour **vérifier la longueur de la liste** , par exemple:

```
myList = ["white", "purple", "blue", "yellow", "green"]
print(len(myList))          # outputs 5
del myList[2]
print(len(myList))          # outputs 4
```

7. Une typique **fonction** ressemble comme suit invocation: `result = function(arg)`, tandis qu'un typique **méthode** ressemble invocation comme ceci: `result = data.method(arg)`.

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, 2, 3, 4, 5]
lst.insert(1, 6)
del lst[0]
lst.append(1)
print(lst)
```

Vérifier

[6, 2, 3, 4, 5, 1]

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, 2, 3, 4, 5]
lst2 = []
add = 0
```

```
for number in lst:
    add += number
    lst2.append(add)

print(lst2)
```

Vérifier

[1, 3, 6, 10, 15]

Exercice 3

Que se passe-t-il lorsque vous exécutez l'extrait de code suivant?

```
lst = []
del lst
print(lst)
```

Vérifier

NameError: name 'lst' is not defined

Exercice 4

Quelle est la sortie de l'extrait de code suivant?

```
lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))
```

Vérifier

[2, 3]

3

Listes de Méthodes (suite)

La méthode sort() :

Python a ses propres mécanismes de trie. La méthode sort() permet de trier les éléments d'une liste dans l'ordre alphanumérique croissant sur place.

Exp :	Exécution:
myList = [8, 10, 6, 2, 4] print("avant le trie: ",myList) myList.sort() print("après le trie : ",myList)	avant le trie : [8, 10, 6, 2, 4] après le trie : [2, 4, 6, 8, 10]

Exp :	Exécution:
myList = ["b","10","Z","a","z","q"] print("avant le trie : ",myList) myList.sort() print("après le trie : ",myList)	avant le trie : ['b', '10', 'Z', 'a', 'z', 'q'] après le trie : ['10', 'Z', 'a', 'b', 'q', 'z']

Remarque : La fonction sorted(liste)

Cette fonction permet de trier une liste sans la modifier

Exp :	Exécution :
li = [3, 1, 4, 2, 5] print ("liste avant sorted ",li) print ("les éléments triés avec la fonction sorted:",sorted(li)) print ("liste après sorted ",li)	liste avant sorted [3, 1, 4, 2, 5] les éléments triés avec la fonction sorted: [1, 2, 3, 4, 5] liste après sorted [3, 1, 4, 2, 5] NB : la liste non modifiée

La méthode reverse() :

La méthode reverse() permet d'inverser les éléments d'une liste sur place.	
Exp : lst = [5, 3, 1, 2, 4] print("avant :", lst) lst.reverse() print("après :",lst)	Exécution: avant : [5, 3, 1, 2, 4] après : [4, 2, 1, 3, 5]
Exp : myList = ["b","10","Z","a","z","q"] print("avant : ",myList) myList.reverse() print("après : ",myList)	Exécution: avant : ['b', '10', 'Z', 'a', 'z', 'q'] après : ['q', 'z', 'a', 'Z', '10', 'b']

Points clés à retenir

1. Vous pouvez utiliser la méthode `sort()`pour trier les éléments d'une liste, par exemple:

```
lst = [5, 3, 1, 2, 4]
print(lst)
lst.sort()
print(lst) # outputs: [1, 2, 3, 4, 5]
```

2. Il existe également une méthode de liste appelée `reverse()`, que vous pouvez utiliser pour inverser la liste, par exemple:

```
lst = [15, 3,0, 2, 34]

print(lst)

lst.reverse()

print(lst) # outputs: [34, 2, 0, 3, 15]
```

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
lst = ["D", "F", "A", "Z"]

lst.sort()

print(lst)
```

Vérifier

```
['A', 'D', 'F', 'Z']
```

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
a = 3

b = 1

c = 2


lst = [a, c, b]

lst.sort()

print(lst)
```

Vérifier

```
[1, 2, 3]
```

Exercice 3

Quelle est la sortie de l'extrait de code suivant?

```
a = "A"
b = "B"
c = "C"
d = " "
lst = [a, b, c, d]
lst.reverse()
print(lst)
```

Vérifier

```
[' ', 'C', 'B', 'A']
```

<table>
<tr><td colspan="2">Listes de Méthodes (suite)</td></tr>
<tr><td colspan="2">La méthode extend(l) :</td></tr>
<tr><td colspan="2">La méthode extend(l) permet de mettre bout à bout deux listes. C’est un agrandissement d’une liste par une autre liste passée en paramètre.</td></tr>
<tr><td>Exp :

lst1 = [1, 2, 3, 4]

print("liste1 : ", lst1)

lst2 = [4, 5, 1, 0]

print("liste2 : ", lst2)

lst1.extend(lst2)

print ("liste1 résultat :",lst1)</td><td>Exécution:

liste1 : [1, 2, 3, 4]

liste2 : [4, 5, 1, 0]

liste1 résultat : [1, 2, 3, 4, 4, 5, 1, 0]</td></tr>
<tr><td colspan="2">La méthode count(p) :</td></tr>
</table>

La méthode count() permet de compter le nombre d'occurrences d'une

valeur dans une listes passée en paramètre.

Exp :	Exécution:
liste = ["a","a","a","b","c","c"]	
print("le nb de a =",liste.count("a"))	le nb de a = 3
print("le nb de c =",liste.count("c"))	le nb de c = 2

La méthode index(p)

La méthode index(p) permet de connaitre la position de l'élément passé en paramètre.

Exp :	Exécution :
liste = ["a","a","a","b","c","c"]	['a', 'a', 'a', 'b', 'c', 'c']
print(liste)	l'index de b est : 3
print("l'index de b est :",liste.index("b"))	l'index de a est : 0
print("l'index de a est :",liste.index("a"))	#envoie la première position trouvée

La méthode remove(p)

La méthode remove(p) permet de retirer d'une liste la première valeur trouvée qui est égale au paramètre. Cela permet de supprimer une valeur dont on ne veut pas dans la liste.

Exp :	Exécution :
liste = ["a", "b", "c"]	
print("avant suppression : ", liste)	avant suppression : ['a', 'b', 'c']
liste.remove("a")	
print("après suppression : ",liste)	après suppression : ['b', 'c']

Remarque :

rappel de l'instruction del qui permet de supprimer un élément de la liste connaissant son index

Exp :	Exécution :
liste = ["a", "b", "c"] print("avant suppression : ", liste) del liste[1] print("après suppression : ",liste)	avant suppression : ['a', 'b', 'c'] après suppression : ['a', 'c']

La méthode pop()

La méthode pop() permet de retirer le dernier élément d'une liste.

Exp :	Exécution :
liste = ["a", "b", "c"] print("avant suppression : ", liste) liste.pop() print("après suppression : ",liste)	avant suppression : ['a', 'b', 'c'] après suppression : ['a', 'b']

Opérations sur les listes & tranches

L'affectation

L'affectation : list2 = list1 copie le nom du tableau, pas son contenu.
En effet, les deux noms (list1 et list2) identifient le même emplacement dans la mémoire de l'ordinateur.
La modification de l'un d'eux affecte l'autre, et vice versa.

Exp :	Exécution :
list1 = [1] print("list1 : ",list1) list2 = list1	list1 : [1]

print("list2 après affectation (list2 = list1): ",list2) list1[0] = 2 print("list1 après modification (list1[0] = 2) : ",list1) print("list2 après modification : ",list2)	list2 après affectation (list2 = list1): [1] list1 après modification (list1[0] = 2) : [2] list2 après modification : [2]

Les tranches

Une tranche est un élément de la syntaxe Python qui permet de faire une toute nouvelle copie d'une liste ou de parties d'une liste

La tranche [:]

Cette partie discrète du code décrit comme [:] est capable de produire une toute nouvelle liste.
La modification de l'une n'affecte pas l'autre, car ce sont 2 objets différents.

Exp : list1 = [1] print("list1 : ",list1) list2 = list1[:] print("list2 après affectation (list2 = list1): ",list2) list1[0] = 2 print("list1 après modification (list1[0] = 2) : ",list1) print("list2 après modification : ",list2)	Exécution : list1 : [1] list2 après affectation (list2 = list1[:]): [1] list1 après modification (list1[0] = 2) : [2] list2 après modification : [1]

La tranche [début:fin]

Cette tranche crée une nouvelle liste (cible), en prenant des éléments de la liste source ; les éléments des indices du début à (fin – 1).
Remarque :

- Pas à fin mais à (fin – 1).
- Un élément d'indice égal à fin est le premier élément qui ne participe pas au découpage.
- L'utilisation de valeurs négatives pour le début et la fin est possible (tout comme dans l'indexation).
- La modification de l'une n'affecte pas l'autre, car ce sont 2 objets différents.
- La nouvelle liste aura (fin – début) éléments.

Exp :	Exécution :

list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[1:3] print("list1 après affectation (list2 = list1[1:3]) : ",list1) print("list2 après affectation : ",list2)	list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[1:3]) : [10, 8, 6, 4, 2] list2 après affectation : [8, 6]
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[2:2] print("list1 après affectation (list2 = list1[-1:1]) : ",list1) print("list2 après affectation : ",list2)	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[-1:1]) : [10, 8, 6, 4, 2] list2 après affectation : []
Tranches avec indices négatifs	
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[1:-1] print("list1 après affectation (list2 = list1[1:-1]) : ",list1) print("list2 après affectation : ",list2) Rq : l'élément d'indice -1 est exclu	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[1:-1]) : [10, 8, 6, 4, 2] list2 après affectation : [8, 6, 4]
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[-1:1] print("list1 après affectation (list2 = list1[-1:1]) : ",list1) print("list2 après affectation : ",list2)	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[-1:1]) : [10, 8, 6, 4, 2] list2 après affectation : []
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[-3:-1] print("list1 après affectation (list2 = list1[--3:-1]) : ",list1)	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[-3:-1]) : [10, 8, 6, 4, 2]

print("list2 après affectation : ",list2)	list2 après affectation : [6, 4]
La tranche [:fin]	
Si la tranche ne contient pas l'indice de début, alors elle commence à l'élément d'index 0 (par défaut). Remarque : myList[:fin] est un équivalent à myList[0:fin]	
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[:3] print("list1 après affectation (list2 = list1[:3]) : ",list1) print("list2 après affectation : ",list2)	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[:3]) : [10, 8, 6, 4, 2] list2 après affectation : [10, 8, 6]
La tranche [début:]	
Si la tranche ne contient pas l'indice de fin, alors elle commence à l'élément d'index début et elle se termine len(liste). Remarque : myList[début:] est un équivalent à myList[début:len(myList)]	
Exp : list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) list2 = list1[2:] print("list1 après affectation (list2 = list1[2:]) : ",list1) print("list2 après affectation : ",list2) Remarque : le dernier élément est inclus	Exécution : list1 : [10, 8, 6, 4, 2] list1 après affectation (list2 = list1[2:]) : [10, 8, 6, 4, 2] list2 après affectation : [6, 4, 2]
Supprimer La tranche [:]	
L'instruction del est capable de supprimer plus qu'un simple élément de liste à la fois ; elle peut également supprimer des tranches .	

Exp :	Exécution :
list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) del list1[1:3] print("list1 après suppression list1[1:3]) : ",list1)	list1 : [10, 8, 6, 4, 2] list1 après suppression list1[1:3]) : [10, 4, 2]
Exp :	Exécution :
list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) del list1[:] print("list1 après suppression list1[1:3]) : ",list1) Remarque : del list1[:] supprime tous les éléments, la liste existe mais vide	list1 : [10, 8, 6, 4, 2] list1 après suppression list1[1:3]) : []
Exp :	Exécution :
list1 = [10, 8, 6, 4, 2] print("list1 : ",list1) del list1 print("list1 après suppression list1[1:3]) : ",list1) Remarque : del list1 supprime la liste elle-même , du coup l'objet liste n'existe plus	list1 : [10, 8, 6, 4, 2] Traceback (most recent call last): File "<string>", line 6, in <module> NameError: name 'list1' is not defined

Opérations sur les listes : in & not in	
Python propose deux opérateurs très puissants, capables de parcourir la liste afin de vérifier si une valeur spécifique est stockée dans la liste ou non.	
Exp :	Exécution :
myList = [0, 3, 12, 8, 2] print(5 in myList) print(5 not in myList) print(12 in myList)	False True True

Points clés à retenir

1. Si vous avez une liste l1, l'affectation suivante : l2 = l1ne fait pas de copie de la liste l1, mais crée les variables l1 et l2 pointent vers une seule et même liste en mémoire.

Par exemple:

```
vehiclesOne = ['car', 'bicycle', 'motor']

print(vehiclesOne)          # outputs: ['car', 'bicycle', 'motor']

vehiclesTwo = vehiclesOne

del vehiclesOne[0]          # deletes 'car'

print(vehiclesTwo)          # outputs: ['bicycle', 'motor']
```

2. Si vous souhaitez copier une liste ou une partie de la liste, vous pouvez le faire en effectuant un découpage :

```
colors = ['red', 'green', 'orange']

print(colors)                    ----------→ ['red', 'green', 'orange']

copyWholeColors = colors[:]

print(copyWholeColors)           ----------→ ['red', 'green', 'orange']

copyPartColors = colors[0:2]

print(copyPartColors) )          ----------→ ['red', 'green']
```

3. Vous pouvez également utiliser **des indices négatifs** pour effectuer des tranches. Par exemple:

```
sampleList = ["A", "B", "C", "D", "E"]

newList = sampleList[2:-1]

print(newList)          # outputs: ['C', 'D']
```

4. Les paramètres start et end sont **facultatifs** lors de l'exécution d'une tranche:, list[start:end] par exemple:

```
myList = [1, 2, 3, 4, 5]

sliceOne = myList[2: ]

sliceTwo = myList[ :2]

sliceThree = myList[-2: ]


print(sliceOne) # outputs: [3, 4, 5]

print(sliceTwo) # outputs: [1, 2]

print(sliceThree) # outputs: [4, 5]
```

5. Vous pouvez **supprimer des tranches** à l'aide de l'instruction del:

```
myList = [1, 2, 3, 4, 5]

del myList[0:2]

print(myList) # outputs: [3, 4, 5]

del myList[:]

print(myList) # deletes the list content, outputs: []
```

6. Vous pouvez tester si certains éléments **existent dans une liste ou ne pas** utiliser les mots `in`- clés et `not in`, par exemple:

```
myList = ["A", "B", 1, 2]

print("A" in myList) # outputs: True

print("C" not in myList) # outputs: True

print(2 not in myList) # outputs: False
```

Exercice 1

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]
```

```
l2 = l1
l3 = l2

del l1[0]
del l2[0]

print(l3)
```

Vérifier

```
['C']
```

Exercice 2

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]
l2 = l1
l3 = l2
print("l1 :",l1)
print("l2 :",l2)
print("l3 :",l3)
del l1[0]
del l2
print("l1 :",l1)
#print("l2 :",l2)  #erreur
print("l3 :",l3)
```

Vérifier

l1 : ['A', 'B', 'C']
l2 : ['A', 'B', 'C']
l3 : ['A', 'B', 'C']
l1 : ['B', 'C']
l3 : ['B', 'C']

Exercice 3

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]
l2 = l1
l3 = l2

del l1[0]
del l2[:]

print(l3)
```

Vérifier

```
[]
```

Exercice 4

Quelle est la sortie de l'extrait de code suivant?

```
l1 = ["A", "B", "C"]
l2 = l1[:]
l3 = l2[:]

del l1[0]
del l2[0]

print(l3)
```

Vérifier

```
['A', 'B', 'C']
```

Exercice 5

Insérez `in` ou `not in` au lieu de `???` pour que le code génère le résultat attendu.

```
myList = [1, 2, "in", True, "ABC"]
```

```
print(1 ??? myList) # outputs True
print("A" ??? myList) # outputs True
print(3 ??? myList) # outputs True
print(False ??? myList) # outputs False
```

Vérifier

```
myList = [1, 2, "in", True, "ABC"]

print(1 in myList) # outputs True

print("A" not in myList) # outputs True

print(3 not in myList) # outputs True

print(False in myList) # outputs False
```

Exemples de remplissage de listes simples où listes dans une autre liste	
Exp: r = [] for i in range(8): r.append(2*i) print(r)	Exécution : [0, 2, 4, 6, 8, 10, 12, 14]
Exp: r=[2*i for i in range(8)] print(r)	Exécution : [0, 2, 4, 6, 8, 10, 12, 14]
Exp: List_impair = [x for x in range(10) if x % 2 != 0] print(List_impair) Exp: List_pair = [x for x in range(10) if x % 2 == 0] print(List_pair)	Exécution : [1, 3, 5, 7, 9] [0, 2, 4, 6, 8]
Exp :	Exécution :

tab = [[i*j for i in range(4)]for j in range(4)] print(tab)	[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6], [0, 3, 6, 9]]

Points clés à retenir

1. La **compréhension des listes** vous permet de créer de nouvelles listes à partir des listes existantes de manière concise et élégante. La syntaxe d'une compréhension de liste se présente comme suit:

```
[expression for element in list if conditional]
```

qui est en fait un équivalent du code suivant:

```
for element in list:
    if conditional:
        expression
```

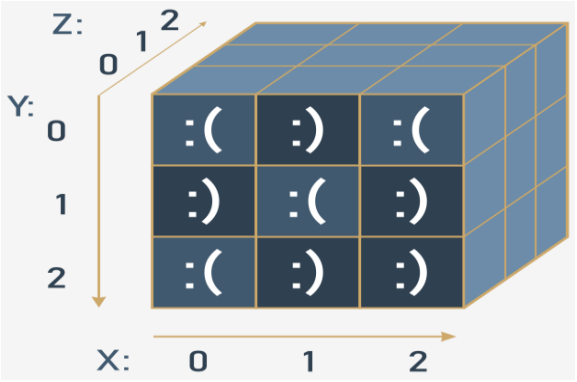
Voici un exemple de compréhension de liste - le code crée une liste de cinq éléments remplie avec les cinq premiers nombres naturels élevés à la puissance de 3:

```
cubed = [num ** 3 for num in range(5)]
print(cubed) # outputs: [0, 1, 8, 27, 64]
```

2. Vous pouvez utiliser **des listes imbriquées** en Python pour créer des **matrices** (c'est-à-dire des listes bidimensionnelles). Par exemple:


```
# A four-column/four-row table - a two dimensional array (4x4)

table = [[":(", ":)", ":(", ":)"],
         [":)", ":(", ":)", ":)"],
         [":(", ":)", ":)", ":("],
         [":)", ":)", ":)", ":("]]


print(table)

print(table[0][0]) # outputs: ':('

print(table[0][3]) # outputs: ':)'
```

3. Vous pouvez imbriquer autant de listes que vous le souhaitez dans les listes et créer ainsi des listes à n dimensions, par exemple des tableaux à trois, quatre ou même soixante-quatre dimensions. Par exemple:




```
# Cube - a three-dimensional array (3x3x3)

cube = [[[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':(', 'x', 'x']],

        [[':)', 'x', 'x'],
         [':(', 'x', 'x'],
         [':)', 'x', 'x']],

        [[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':)', 'x', 'x']]]

print(cube)
print(cube[0][0][0]) # outputs: ':('
print(cube[2][2][0]) # outputs: ':)'
```