



Python - Listes

[Mise à jour le : 29/2/2024]

- **Source**
 - **Documentation** sur Python.org : [référence du langage](#), [types séquentiels list, tuple, range](#), [compléments sur les listes](#), [fonctions natives](#) (built-in)
- **Mots-clés** : mutable, tableau, méthode, parcours de liste, compréhension de liste

Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

| | | | | |
|--------------|--------------|------------|-----------------|--------------|
| and | continue | finally | is | raise |
| as | def | for | lambda | return |
| assert | del | from | <u>None</u> | <u>True</u> |
| <u>async</u> | elif | global | <u>nonlocal</u> | try |
| <u>await</u> | else | if | not | while |
| break | except | import | or | with |
| class | <u>False</u> | in | pass | yield |

- **Fonctions natives (built-in)**¹⁾ utilisées dans les exemples : **len()**, **list()**, **print()**.

1. Introduction

Les tableaux de Python diffèrent des tableaux que l'on trouve dans les autres langages de programmations par plusieurs aspects :

- ils sont appelés **listes** dans la documentation de Python;
- ils peuvent être **agrandis** ou **rétrécis** du côté des indices les plus grands avec *append* et *pop*;
- accéder à un tableau Python avec un **indice négatif** ne provoque pas nécessairement une erreur. Par exemple `t[-1]` permet d'accéder au dernier élément du tableau `t`, `t[-2]` à l'avant dernier, etc. Pour un tableau de taille n , seul un indice en dehors de l'intervalle **[-n, n-1]** provoquera une **erreur**.

Une liste est une **séquence d'objets hétérogènes** : entiers, réels, caractères, chaînes, etc. Elle stocke des **références** vers les objets. La **taille** de l'objet liste est **indépendante** du type d'objets référencés.

La liste est un **objet mutable** (modifiable **ou** il est stocké). Elle stocke des **références** vers les objets. En général on utilise le terme « **tableau** » lorsque les éléments sont tous de même type.

Exemples

*.py

```
# Deux variables pointent vers le même objet (aliasing)
t1 = ['a', 'b', 'c']
t2 = t1 # Python ne crée pas un autre tableau mais une référence
        (pointeur) vers t1
```



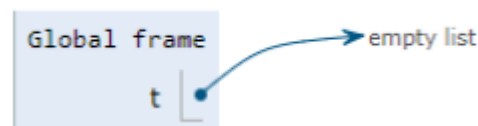
*.py

```
# on modifie l'objet avec la référence t1
t1 += ['d', 'e', 'f'] # les deux variables pointent toujours vers le
                     # même objet
```



2. Construction

Frames Objects



*.py

```
# Liste vide
t = list() # On crée une liste vide
t = []     # Autre méthode pour créer une liste vide (à privilégier)

# Liste non vide
t = [1, 2, 3, 4, 5] # Construction en extension d'une
```

```

liste contenant cinq entiers et
ma_liste = [1, 3.5, "une chaine", []] # autre exemple avec une liste
dans une liste

# Grande liste
t = [0]*1000 # Création d'une liste de 1000 cases initialisées à 0

```



Les **crochets []** délimitent les **listes**.

3. Accès et modification des éléments

Exemple

*.py

```

t = ['c', 'f', 'm']
t[0]           # On accède au premier élément de la liste, ici 'c'
t[2]           # Troisième élément, ici 'm'
t[1] = 'Z'     # On remplace 'f' par 'Z', t renvoie ['c', 'Z',
'm']

```

4. Ajout d'éléments

On dispose de plusieurs méthodes dans la classe **list** (append, extend, insert, etc).

- **Ajout d'un élément à la fin d'une liste, ou d'une liste à la fin d'une autre liste**

On utilise la méthode **append()**.

Exemple

*.py

```
# Ajout d'un élément
impaire = [1,3,5,7]
t4 = impaire.append(9)
print(impaire)      # Résultat : [1, 3, 5, 7, 9]
print t4            # Résultat : aucun, la liste impaire est modifiée
                    # où elle se trouve, append ne renvoie rien !

# Ajout d'une liste
t1 = [1,3,5,7]
t2 = [9,11,13]
t3 = t1.extend(t2)
print(t1)           # Résultat : [1, 3, 5, 7, 9, 11, 13]
print(t3)           # Résultat : aucun, la liste impaire est modifiée
                    # où elle se trouve, extend ne renvoie rien !
```

• Ajout d'un élément dans la liste

On utilise la méthode **insert(position,valeur)** ou une opération de slice **liste[début:fin]** avec début = fin.

Exemple



*.py

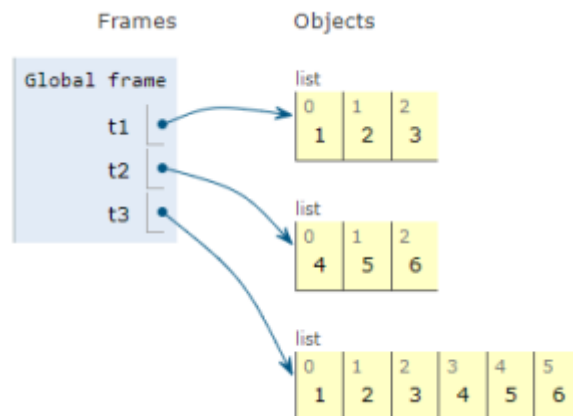
```
impaire = [1,3,5,7,9,13]
impaire.insert(5,11)    # la valeur 11 est placée à la cinquième
                        # position de la liste (début=0!)
print(impaire)          # Résultat : [1, 3, 5, 7, 9, 11, 13]

# Opération de slice
impaire = [1,3,5,7,9,13]
impaire[5:5]=[11]       # Résultat : [1, 3, 5, 7, 9, 11, 13]
```

• Concaténation de listes

On utilise **+** pour assembler (concaténer) des listes. A la différence de **append** et **extend** qui modifient la liste sur laquelle elle travaille, **+** **crée un nouvel objet**.

Exemple



*.py

```
t1 = [1,2,3]
t2 = [4,5,6]
t3 = t1 + t2
print(t3)

# Résultat : [1,2,3,4,5,6]
```

5. Suppression d'éléments

On utilise le mot-clé **del** ou la méthode **remove**.

Exemples

*.py

```
# mot-clé del
# Prend en argument la position de l'élément à supprimer
liste3 = [1, 2, 3, 4, 5, 6]
del(liste3[2])
print(liste3) # Résultat : [1, 2, 4, 5, 6]

# Méthode remove
# Prend en argument la valeur de l'élément à supprimer
t3 = [1, 2, 3, 4, 5, 6]
t3.remove(2)
print(t3) # Résultat : [1, 3, 4, 5, 6]

# Slicing (développé au §7)
t3 = [1, 2, 3, 4, 5, 6]
t3[2:4]=[]
print(t3) # Résultat : [1, 2, 5, 6]
```

La méthode **remove** retire uniquement le premier élément trouvé dans une liste.

6. Parcours des listes

- Utilisation de la boucle **while** ...

*.py

```
# Exemple avec while
adresse = ['Lycée', 'Pierre-Emile', 'Martin', '1 Avenue de Gionne',
          '18000', 'Bourges']
i=0
while i<len(adresse):
    print(adresse[i]+" ", end='') # end='' supprime le retour à la
    ligne après chaque mot
    i+=1

# Résultat
Lycée Pierre-Emile Martin 1 Avenue de Gionne 18000 Bourges
```

- Utilisation de la boucle **for** ...

*.py

```
adresse = ['Lycée', 'Pierre-Emile', 'Martin', '1 Avenue de Gionne',
          '18000', 'Bourges']

# solution 1. Utilisation d'un indice
for i in range(len(adresse)): # renvoie : Lycée Pierre-Emile Martin 1
    Avenue de Gionne 18000 Bourges
    elemt = adresse[i]+' '
    print(elemt,end='')

# solution 2. Itération directe sur les éléments de la liste (plus
simple !)
for elemt in adresse: # elemt va prendre les valeurs successives des
    éléments de ma_liste
    print(elemt+" ", end='')
```

L'**itération directe** sur les éléments est **plus simple** à lire et à écrire mais n'est applicable que lorsqu'il n'est pas nécessaire de connaître l'indice.

- Utilisation de **enumerate**

La fonction native **enumerate()** permet de parcourir un itérable tout en gardant un compte des itérations. `enumerate()` renvoie des **tuples** constitués de la **position d'un élément** dans la liste et de **sa valeur**.

- Syntaxe

- `enumerate(iterable)`
- `enumerate(iterable, start=0)`

*.py

```

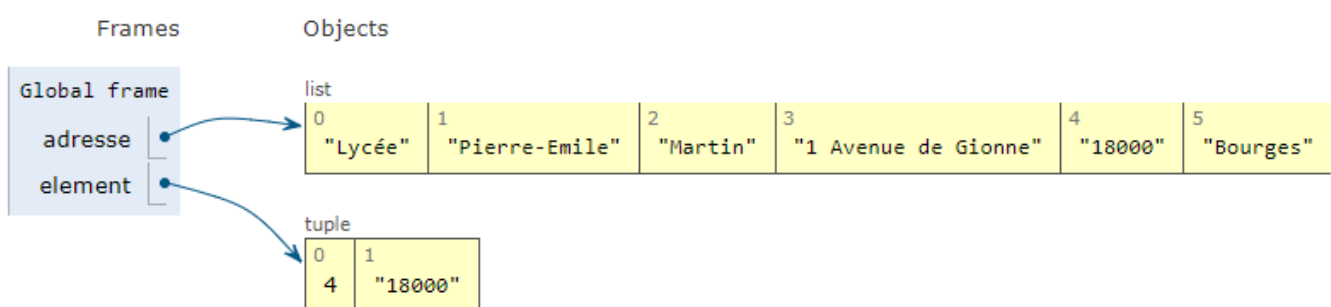
adresse = ['Lycée', 'Pierre-Emile', 'Martin', '1 Avenue de Gionne',
           '18000', 'Bourges']
for element in enumerate(adresse):
    print(element)

# Résultat (figure ci-dessous)
(0, 'Lycée')
(1, 'Pierre-Emile') etc.

# Pour récupérer la position indépendamment de la valeur, on utilise
deux variables
for i, val in enumerate(adresse):
    print(val, "est à la position", i)

# Résultat
# Lycée est à la position 0
# Pierre-Emile est à la position 1 etc.

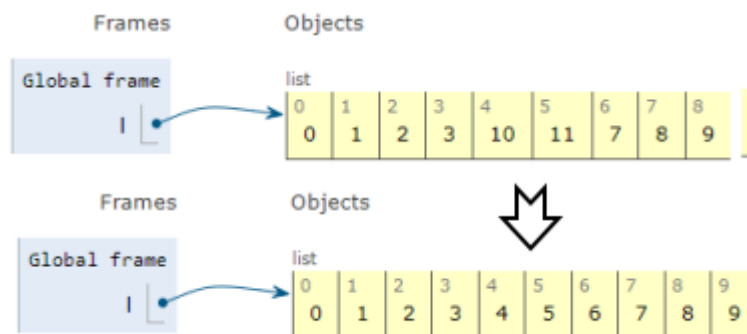
```



7. Affectation sur les slices

L'affectation sur un **slice** permet de **remplacer** ou **d'effacer** des éléments dans une liste.

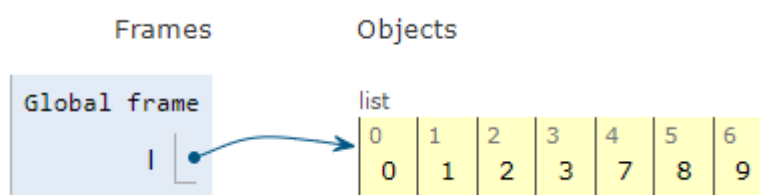
Exemple : remplacement d'objets



*.py

```
l=[0,1,2,3,10,11,7,8,9] # Création d'une liste
l[4:6]=[4,5,6]          # Affectation sur le slice
                          # Les éléments aux positions 4 et 5 sont
                          # supprimés
                          # puis remplacés par les valeurs 4,5,6 (4,5
                          # remplacent 10,11.
                          # 6 est ajouté avant 7. 7,8,9 sont décalés)
                          # Résultat : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Exemple : effacement d'objets



*.py

```
l=[0,1,2,3,10,11,7,8,9] # Création d'une liste
l[4:6]=[]               # Les éléments aux positions 4 et 5 sont
                          # supprimés
                          # et non remplacés
                          # Résultat : [0,1,2,3,7,8,9]
```

8. Compréhensions de liste

Les compréhensions de liste facilitent la **rédaction d'un code très propre** qui se lit presque comme un langage naturel.

Organisation

liste = [opération sur la variable **for** variable **in** liste **if** condition]

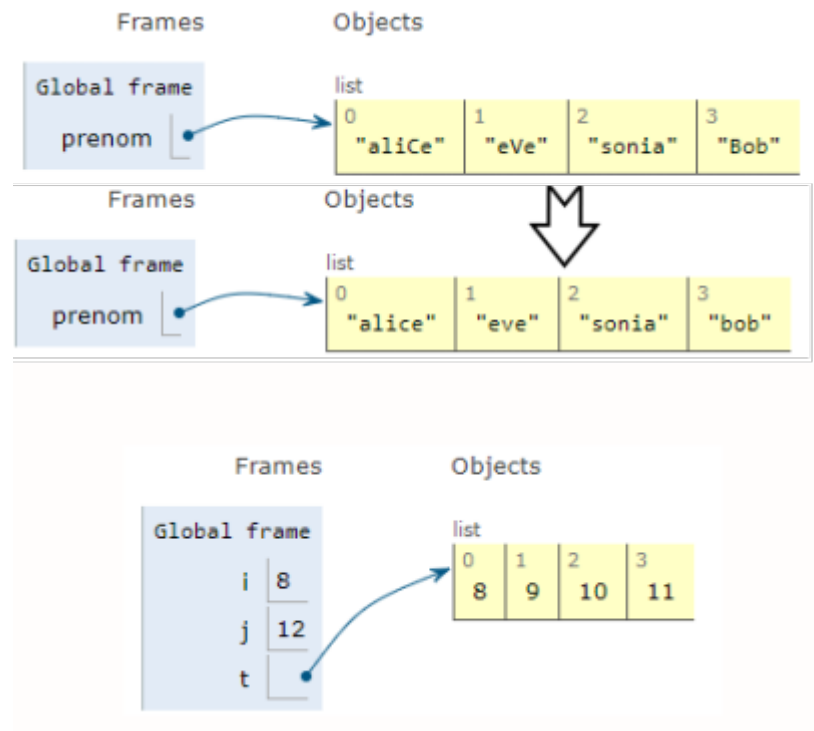
Exemple de création d'une liste par compréhension

*.py

```
# Forme habituelle
t = [0]*100
for i in range(100):
    t[i] = 3*i+1

# Notation par compréhension
t = [3*i+1 for i in range 100]
```

Exemples de traitement de listes par compréhension



*.py

```
prenom = ['aliCe', 'eVe', 'sonia', 'Bob']
chiffres = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# On passe tous les prénoms en minuscule (la liste prenom est modifiée)
prenom = [p.lower() for p in prenom] # Résultat ['alice', 'eve', 'sonia', 'bob']

# On extrait et on affiche les chiffres pairs (une liste est créée à partir du contenu d'une autre)
pair = [x for x in chiffres if x % 2 is 0]
print(pair) # Résultat : [2, 4, 6, 8]

# On extrait et on affiche les chiffres impairs (une liste est créée à partir du contenu d'une autre)
impair = [y for y in chiffres if y not in pair]
print(impair) # Résultat : [1, 3, 5,
```

```
7, 9]
```

```
# Cas particulier : liste contenant de i à j exclus
[x for x in range(i,j)]
# ou
# La fonction prédéfinie list transforme l'ensemble des éléments en un
tableau
list(range(i,j))
```

9. Liste de listes (tableaux à plusieurs dimensions)

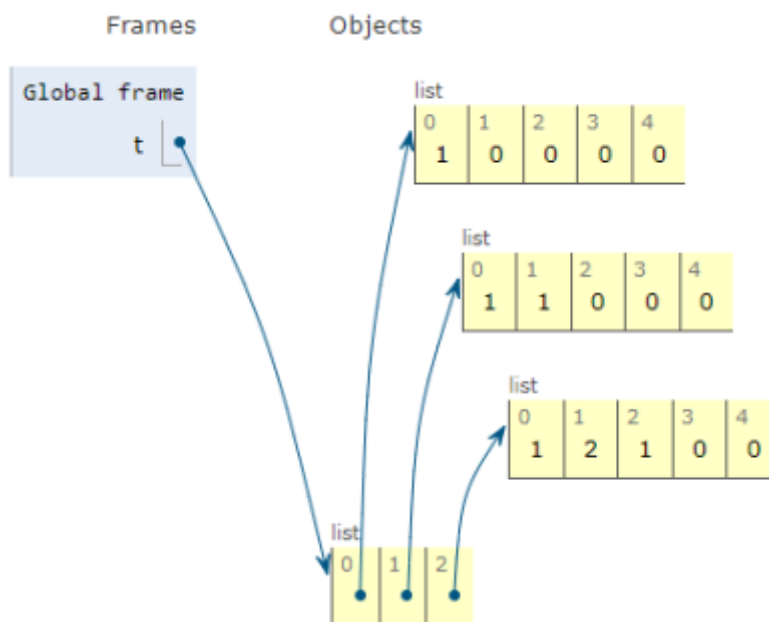
Les listes peuvent contenir des listes.

- **Construction en extension**

| t | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 2 | 1 | 0 | 0 |

*.py

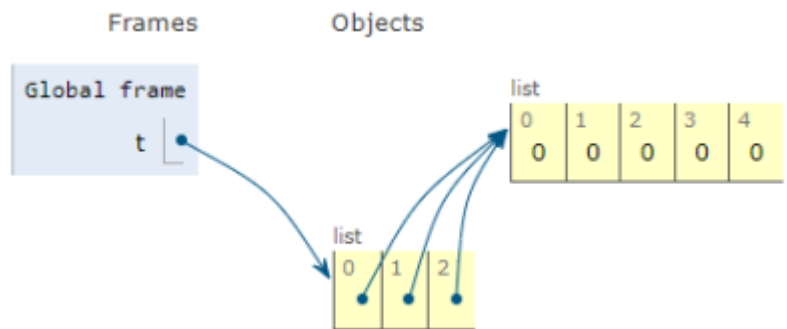
```
t = [[1,0,0,0,0],[1,1,0,0,0],[1,2,1,0,0]]
```



- **Construction par compréhension**

*.py

```
# Tableau 3 x 5
t = [[0]*5 for i in range(3)]
```



Erreur

Il pourrait être tentant d'écrire `t = [[0] * 3]` mais cela **ne donne pas** le résultat ci-dessus mais celui ci-contre.

• Accès à un élément

*.py

```
a = t[2][1] # renvoi 2
```

• Parcours d'un tableau à plusieurs dimensions

*.py

```
# Exemple : somme des éléments d'un tableau 3 x 5
t = [[1,0,0,0,0],[1,1,0,0,0],[1,2,1,0,0]]
# Solution 1 # Définition des intervalles
s=0

for i in range(3):
    for j in range(5):
        s+=t[i][j]
print(s) # donne 7

# solution 2 # Itération directe sur les éléments de la liste (plus simple !)
s=0
for ligne in t:
    for colonne in ligne:
        s+=colonne
print(s) # donne 7
```

10. Transformations

- Transformation d'une chaîne en liste

On utilise la méthode de chaîne **split**.

*.py

```
texte = "Le manuel de Python"
print(texte.split()) # Résultat : ['Le', 'manuel', 'de', 'Python']
                    # Le séparateur est passé en paramètre à split.
                    # Par défaut, il s'agit de la virgule.
```

- Transformation d'une liste en chaîne

On utilise la méthode de list **join**.

*.py

```
liste = ['Le', 'manuel', 'de', 'Python']
texte = " ".join(liste) # les guillemets contiennent le séparateur (si
                        # rien, le séparateur est la virgule)
print(texte)           # Résultat : Le manuel de Python
```

- Transformation d'une liste en paramètre de fonction

Si on dispose d'un tuple ou d'une liste contenant des paramètres, on peut les transformer en paramètre avant de les passer à une fonction.

Exemple

*.py

```
impair = [1,3,5,7,9]
# On imprime la liste
print(impair)           # Résultat : [1,
                        # 3, 5, 7, 9]
# On imprime les paramètres contenus dans la liste !
print(*impair)          # Résultat : 1 3
                        # 5 7 9
```

11. Méthodes et fonctions de la classe list

- Source : w3school.com

11.1 Méthodes

| Méthodes | Paramètres | Effet | Structure |
|----------------|----------------------|--|---|
| append | <i>elem</i> | Ajoute un élément <i>elem</i> à la fin de la liste. | <i>lst.append(elem)</i> |
| clear | | Supprime tous les éléments de la liste. | <i>lst.clear()</i> |
| copy | | Renvoie une copie de la liste. | <i>lst.copy()</i> |
| count | <i>elem</i> | Renvoie le nombre d'éléments avec la valeur spécifiée. | <i>lst.count(elem)</i> |
| extend | <i>iterable</i> | Ajoute une liste à une autre. | <i>lst.extend(iterable)</i> |
| index | <i>elem</i> | Renvoie la position de la première occurrence de la valeur spécifiée | <i>lst.index(elem)</i> |
| insert | <i>pos, elem</i> | Ajoute un élément <i>elem</i> à la position <i>pos</i> . | <i>lst.insert(pos,elem)</i> |
| pop | <i>pos</i> | Supprime l'élément à la position spécifiée. | <i>lst.pop(pos)</i> |
| remove | <i>elem</i> | supprime le premier élément <i>elem</i> trouvé dans la liste | <i>lst.remove(elem)</i> |
| reverse | | Inverse l'ordre dans la liste. | <i>lst.reverse()</i> |
| sort | <i>reverse, func</i> | Trie les éléments d'une liste dans l'ordre croissant ou décroissant. | <i>lst.sort(reverse=True/False, key=myFunc)</i> |

11.2 Fonctions applicables aux listes

| Fonctions | Paramètres | Effet | Structure |
|------------------|-----------------------|---|---------------------------------|
| len | <i>liste</i> | Renvoie le nombre d'éléments de la liste | len(liste) |
| max (min) | <i>liste</i> | Renvoie le plus grand (petit) élément de la liste | max(liste) |
| comp | <i>pos</i> | Compare les éléments de deux listes. Retourne 0 si elles sont égales, -1 si la première est < à la seconde, 1 sinon | |
| sorted | <i>liste</i> | Renvoie une copie triée de la liste. | |
| + | <i>liste X liste</i> | Renvoie une concaténation des deux listes | <i>liste3 = liste1 + liste2</i> |
| * | <i>liste X entier</i> | Renvoie une liste formée de N fois la liste paramètres | <i>liste2 = liste1 * n</i> |

Exemples

[exliste1.py](#)

```
une_liste = ["Journal", 9, 2.714, "pi"]
print(une_liste) # Résultat : ['Journal', 9, 2.714, 'pi']
len(une_liste) # Résultat 4
une_liste.append("fin") # Résultat : ['Journal', 9, 2.714, 'pi', 'fin']
del(une_liste[2]) # Résultat : ['Journal', 9, 'pi', 'fin']
```

Résumé

- Une liste est une séquence **mutable** (modifiable après sa création) pouvant contenir plusieurs objets.
- Une liste se construit avec la syntaxe *nomliste* = [*élément₁*, *élément₂*, *élément_N*].
- On peut insérer des éléments dans une liste à l'aide des méthodes **append**, **insert** et **extends**.
- On peut supprimer des éléments d'une liste avec le mot-clé **del**, la méthode **remove** ou une opération de slice.
- On peut créer des fonctions attendant un nombre inconnu de paramètres en plaçant une * devant le nom du paramètre.
- Les compréhensions de liste permettent de parcourir et filtrer une séquence en renvoyant une nouvelle avec la syntaxe *nouvelle_seq* = [*elem* **for** *elem* **in** *ancienne_seq* **if** *condition*].
- Un tuple est une séquence pouvant contenir des objets. À la différence de la liste, le tuple ne peut pas être modifié une fois créé.



Quiz

- [Python List and Tuples Quiz](#)



Pour aller plus loin ...

- [Python's .append\(\): Add Items to Your Lists in Place](#)
- [Reverse Python Lists: Beyond .reverse\(\) and reversed\(\)](#)
- [Python's filter\(\): Extract Values From Iterables](#)
- [Custom Python Lists: Inheriting From list vs UserList](#)
- [Using the len\(\) Function in Python](#)

¹⁾

Fonctions toujours disponibles.

From:

<http://webge.fr/dokuwiki/> - **WEBGE Wikis**

Permanent link:

<http://webge.fr/dokuwiki/doku.php?id=python:bases:listes&rev=1709192039>

Last update: **2024/02/29 08:33**



