LISTE

Les listes font parties des quatre grandes **structures de données** qui existent en Python en plus des **sets**, des **tuples** et des **dictionnaires**.

C'est un objet très pratique, aussi bien pour les développeurs débutants que pour ceux plus expérimentés.

Une liste est simplement une structure de données **muable** et **ordonnée** dans laquelle tu peux stocker **n'importe quel type d'objet**.

Chaque objet contenu dans une liste est appelé un **élément.**

Pour créer une liste, tu dois placer ces éléments entre des crochets et les séparer par une virgule.

On accède aux éléments d'une liste grâce à leur **indice**, c'est-à-dire leur position dans la liste.

Enfin, et c'est vraiment le plus intéressant, une liste est une **séquence de données !** Cela signifie que tu peux **itérer** sur cette séquence, avec une boucle `for` par exemple.

Une liste est ordonnée

C'est important que tu retiennes bien qu'une liste est ordonnée car ce n'est pas le cas de toutes les structures de données en Python.

Dans tous les cas, cela signifie simplement que les éléments de ta liste seront toujours dans l'ordre que tu auras défini dans ton code.

Il n'y a pas d'histoire d'ordre alphabétique ou quoi que ce soit, c'est toi qui décides !

```
# liste vide
villes = []

# liste avec trois items de type str
villes = ['Paris', 'Lille', 'Lyon']
print(villes)
```

▶ Exécuter

N'importe quel type de données

Comme je te le disais, tu peux stocker vraiment n'importe quoi dans une liste ! Des nombres, des booléens, des chaînes de caractères et pourquoi pas d'autres listes. Tout ça au sein d'une seule et même liste.

Alors c'est vrai que dans la pratique, on préfère souvent stocker des éléments d'un même type comme des prénoms, des noms de villes, ou des montants mais sache qu'il n'y a aucune limite.

```
# liste avec trois éléments de type str
villes = ['Paris', 'Lille', 'Lyon']

# liste avec cinq éléments de type int
prix = [3, 10, 25, 40, 100]

# liste avec plusieurs éléments de types différents
liste_de_tout_et_rien = [5, 'Docstring', True, 9.5, 4, 'Python', ['

# liste qui contient un dictionnaire
adresse = [
    {
        'rue': 'rue du Serpent',
        'numero': 6,
        'ville': 'Lille'
    }
]
```

Les indices

Dans une liste chaque élément est accessible grâce à sa position.

On appelle cette position un indice (index en anglais).

On continue avec les villes :

```
villes = ['Paris', 'Lille', 'Lyon']

print(villes[0])  # 'Paris'
print(villes[1])  # 'Lille'
print(villes[2])  # 'Lyon'
```

▶ Exécuter

On reviendra un peu plus tard sur cette notion d'indice car on peut l'utiliser de plusieurs façons.

☢ Attention: Pour rappel, le premier élément d'une liste se trouve toujours en position zéro ! On aura donc toujours un décalage de 1. Si vous souhaitez accéder au 2e élément d'une liste, il faudra utiliser l'indice 1 et ainsi de suite.

Muable

Quand on parle d'objet muable, cela veut dire qu'un objet peut être modifié même après son assignation.

C'est-à-dire qu'on va pouvoir ajouter, modifier ou supprimer des éléments à la volée.

```
villes = ['Paris', 'Lile', 'Lyon']

# J'ai mal orthographié exprès un élément, on va modifier ça :
villes[1] = 'Lille'
print(villes) # ['Paris', 'Lille', 'Lyon']

# Et si on ajoutait une ville ?
villes.append('Strasbourg')
print(villes) # ['Paris', 'Lille', 'Lyon', 'Strasbourg']

```

▶ Exécuter

Itérable

Et oui, tu peux itérer sur une liste car c'est une **séquence de données** au même titre qu'une chaîne de caractères. Ça devient très facile d'appliquer une logique à tous les éléments de ta liste !

```
villes = ['Lille', 'Lyon', 'Strasbourg']

for ville in villes:
    print(ville.upper())
```

▶ Exécuter

Quand doit-on utiliser une liste ?

C'est une question que tu es en droit de te poser au vu des nombreuses possibilités offertes par Python.

Je vais te donner une astuce personnelle que j'utilise à chaque fois que j'ai besoin de créer une structure de données :

- Est-ce que j'ai besoin d'associer une valeur à une clé ? Oui ?→ **Dictionnaire**
- Est-ce que j'ai besoin de modifier ma structure de données ? Non → **Tuple**
- Est-ce que j'ai besoin de garder les éléments dans le même ordre ? Non → **Set**
- Pour toutes les autres possibilités → **Liste**

Avec ce raisonnement, tu devrais t'en sortir dans la majorité des cas.

De toute façon, sache que c'est assez simple de passer d'une structure à l'autre grâce aux fonctions de conversion de type de Python.

Par exemple il est facile de retirer tous les doublons d'une liste en passant par un set :

```
villes = ['Paris', 'Lille', 'Lyon', 'Paris', 'Strasbourg']
villes = list(set(villes))
print(villes) # ['Lyon', 'Lille', 'Strasbourg', 'Paris']
```

▶ Exécuter

J'utilise la fonction `set()` pour convertir ma liste `villes` d'origine en `set`, ce qui va avoir pour effet de retirer tous les doublons car les `set` ne les acceptent pas. Ensuite je n'ai plus qu'à reconvertir ce set en liste pour la récupérer sans les doublons.

☢️ Attention: Comme la liste a été convertie en set, il se peut que les éléments ne soient plus dans le même ordre que celui d'origine.

Ce que tu dois absolument savoir faire avec

Je vais commencer par les choses les plus évidentes puis on ira vers des opérations plus complexes 🙂

Créer une liste

Il existe deux méthodes pour créer une liste vide :

- Avec les crochets → `[]`
- Avec le constructeur → `list()`

Avec les crochets

```
liste_vide = []
```

Avec le constructeur

```
liste_vide = list()
```

Généralement, on préfère utiliser **les crochets** pour créer une liste.

C'est plus lisible et c'est un petit peu plus rapide ! Bon, on parle de millièmes de secondes de différence, ça n'a aucun impact mais ça reste un argument. Utilise seulement le constructeur `list()` quand tu as besoin de convertir un autre objet en liste :

```
site = 'Docstring'
site_to_liste = list(site)
print(site_to_liste)  # ['D', 'o', 'c', 's', 't', 'r', 'i', 'n', 'g'
```

▶ Exécuter

```
['D', 'o', 'c', 's', 't', 'r', 'i', 'n', 'g']
```

Accéder à des éléments

- `liste[indice]` → Retourne l'élément associé à l'indice
- `liste[début:fin:pas]` → Retourne le ou les éléments en fonction de l'intervalle précisé

`liste[indice]` → Retourne l'élément associé à l'indice

Pour accéder à un élément dans une liste, on utilise la notation entre crochets `[ ]` et l'indice de l'élément.

```
villes = ['Paris', 'Lille', 'Lyon']

print(villes[1])   # 'Lille'
print(villes[-1])  # 'Lyon'
print(villes[3])   # IndexOutOfRange Error
```

▶ Exécuter

```
Lille
Lyon
IndexError: list index out of range on line 5
```

Tu peux également récupérer des éléments en partant de la fin avec les indices -1, -2 et

ainsi de suite.

`liste[début:fin:pas]` → Retourne le ou les éléments en fonction de l'intervalle précisé

Dans ce cas, on doit utiliser le découpage par tranche *ou* slicing* : `[début:fin:pas]`

Cela te permet de créer un intervalle de sélection dans ta liste et de préciser éventuellement un pas si tu souhaites récupérer un élément sur deux par exemple.

Jouons avec ça :

```
villes = ['Paris', 'Lille', 'Lyon']

print(villes[0:2])    # ['Paris', 'Lille']
print(villes[0::2])   # ['Paris', 'Lyon']
print(villes[0:])     # ['Paris', 'Lille', 'Lyon']
print(villes[:])      # ['Paris', 'Lille', 'Lyon']
print(villes[:-1])    # ['Paris', 'Lille']
print(villes[::-1])   # ['Lyon', 'Lille', 'Paris']
```

▶ Exécuter

Tu peux utiliser cette technique pour sélectionner des items évidemment mais aussi pour copier proprement une liste ou encore pour inverser l'ordre des éléments à l'intérieur d'une liste, c'est très pratique !

Fais attention quand tu utilises le *slicing* car l'indice de début est **inclusif** tandis que l'indice de fin est **exclusif.**

Ajouter des éléments

Pour ajouter des éléments dans une liste, Python dispose de plusieurs méthodes :

- `.append(item)` → Ajoute un item à la fin de ta liste
- `.insert(index, item)` → Ajoute un item à la position indiquée en paramètre
- `.extend(iterable)` → Ajouter tous les items de la collection dans ta liste

`.append(item)` → Ajoute un élément à la fin de ta liste

C'est la méthode la plus connue et celle qui est le plus souvent utilisée :

```
villes = ['Paris', 'Lille', 'Lyon']
villes.append('Strasbourg')
print(villes)
```

▶ Exécuter

`.insert(index, item)` → Ajoute un item à la position indiquée en paramètre

Beaucoup moins utilisée mais qui peut s'avérer utile.

```
villes = ['Paris', 'Lille', 'Lyon']
villes.insert(1, 'Strasbourg')
print(villes)  # ['Paris', 'Strasbourg', 'Lille', 'Lyon']
```

▶ Exécuter

```
['Paris', 'Strasbourg', 'Lille', 'Lyon']
```

Tu peux là aussi utiliser un indice négatif, pour insérer un élément en partant de la fin de la liste :

```
villes = ['Paris', 'Lille', 'Lyon']
villes.insert(-1, 'Strasbourg')
villes.insert(-2, 'Nice')
print(villes)
```

▶ Exécuter

```
['Paris', 'Lille', 'Nice', 'Strasbourg', 'Lyon']
```

`.extend(iterable)` → Ajouter tous les items de la collection dans ta liste

```
villes = ['Paris', 'Lille', 'Lyon']
autres_villes = ['Strasbourg', 'Marseille']
villes.extend(autres_villes)
print(villes)  # ['Paris', 'Lille', 'Lyon', 'Strasbourg', 'Marseille
```

▶ Exécuter

```
['Paris', 'Lille', 'Lyon', 'Strasbourg', 'Marseille']
```

Modifier des éléments

Une liste est un objet muable, ce qui signifie qu'on peut la modifier sans problème !

- `liste[indice] = value` → Assigne une nouvelle valeur
- `liste[début:fin] = [value1, value2, ...]` → Assigne des nouvelles valeur sur l'intervalle donné

`liste[indice] = value` → Assigne une nouvelle valeur

```
1 villes = ['Paris', 'Lille', 'Lyon']
2 villes[1] = 'Strasbourg'
3 print(villes)  # ['Paris', 'Strasbourg', 'Lyon']
```

▶ Exécuter

```
['Paris', 'Strasbourg', 'Lyon']
```

`liste[début:fin] = [value1, value2, ...]` → Assigne des nouvelles valeur sur l'intervalle donné

```
1 villes = ['Paris', 'Lille', 'Lyon']
2 villes[1:] = ['Strasbourg', 'Rennes']
3 print(villes)  # ['Paris', 'Strasbourg', 'Rennes']
```

▶ Exécuter

```
['Paris', 'Strasbourg', 'Rennes']
```

Supprimer des éléments

Il existe trois façons de supprimer un élément :

- `del liste[indice]` → Supprime un ou plusieurs éléments d'une liste. Peut aussi être utilisé pour **détruire** complètement une liste.
- `liste.pop(indice)` → Retire un élément de la liste. Si tu ne précises pas d'indice, cela retire automatiquement le dernier élément. Et surtout, cela permet de récupérer l'élément supprimé.
- `liste.remove(item)` → Supprime un élément de la liste.

`del liste[indice]` → Supprime un ou plusieurs éléments d'une liste. Peut aussi être utilisé

pour détruire complètement une liste.

```
villes = ['Paris', 'Lille', 'Lyon']

del villes[0]
print(villes)  # ['Lille', 'Lyon']
```

▶ Exécuter

```
villes = ['Paris', 'Lille', 'Lyon']

del villes
print(villes)  # NameError
```

▶ Exécuter

`liste.pop(index)` → Retire un item de la liste. Si tu ne précises pas d'index, cela retire automatiquement le dernier élément. Et surtout, cela permet de récupérer cet élément.

```
villes = ['Paris', 'Lille', 'Lyon']

ma_ville = villes.pop(1)
print(villes)    # ['Paris', 'Lyon']
print(ma_ville)  # Lille
```

▶ Exécuter

`liste.remove(item)` → Supprime un élément de la liste.

```
villes = ['Paris', 'Lille', 'Lyon']

villes.remove('Paris')
print(villes)  # ['Lille', 'Lyon']
```

▶ Exécuter

```
villes = ['Paris', 'Lille', 'Lyon']

villes.remove('Marseille')
print(villes)  # ValueError
```

▶ Exécuter

LE SITE

Accueil

Tableau de bord

Articles

Podcast

Nouveautés

Formules d'abonnement

APPRENTISSAGE

Parcours

Formations

Le glossaire

Exercices

Quiz

Projets

À PROPOS

Termes et conditions

CGV

Nous contacter

Politique des cookies

INSCRIS-TOI À LA NEWSLETTER

Pour recevoir les dernières nouveautés du site et de Python ainsi que des astuces et liens intéressants.

Prénom...

elon@musk.com

S'abonner