- Une liste Python est une **collection** d'objets *numérotés*. On accède à un élément de la collection *via* son **index** (ou indice), **compté à partir de 0**. Ainsi, pour `liste = [10, 20, 30]`, l'instruction `liste[0]` renvoie 10, `liste[1]` donne 20 et `liste[2]` donne 30.
- *Une liste Python est assez éloignée du type abstrait de données « liste », tel que défini par les chercheurs en informatique. Une liste Python se rapproche davantage d'un* « tableau dynamique » *(comme pour le type abstrait de données « tableau », on peut accéder* directement *à n'importe quel élément d'une liste Python sans devoir parcourir tous les éléments qui le précèdent ; en revanche, un « tableau abstrait » est de* taille fixe, *alors qu'une liste Python peut être étendue, augmentée à volonté).*
- On obtient la **longueur d'une liste** avec l'instruction `len(liste)`. Avec la définition précédente de la variable `liste`, `len(liste)` renvoie la valeur 3.
- **Vérifier qu'un élément appartient à une liste** implique le mot-clef `in`. Ainsi, `2 in [0, 1, 2]` renverra `True`.
- **Une liste Python est un objet *modifiable*** (on dit aussi *mutable*) : si `liste = [10, 20, 30]`, après l'instruction `liste[1]=0`, la variable `liste` désignera `[10, 0, 30]`.

  ⚠ *Cela implique que lorsqu'une fonction a une liste comme paramètre,* ***il est possible de modifier la liste directement depuis le code de la fonction.*** *Si cela n'est pas souhaité, il faut alors* ***commencer par recopier le contenu de la liste*** *(voir plus loin) dans une variable locale, déclarée dans le corps de la fonction.*
- **La création d'une liste Python se fait souvent par accumulation :** on part d'une liste vide `liste = []`, et on lui ajoute des éléments, le plus souvent « par la droite » (ou « par la fin »). Il y a trois possibilités :
  - la « méthode » `append()` : après l'instruction `liste.append(2)`, la variable `liste` désignera `[2]` ;
  - la fusion de deux listes : après l'instruction `liste = liste + [4]`, la variable `liste` désignera `[2, 4]` (on peut donc ajouter « par la gauche » des éléments à une liste avec cette technique) ;
  - l'extension par une autre liste : après `liste.extend([6, 8])`, la variable `liste` désigne `[2, 4, 6, 8]`.
- **Python supporte la création de listes *« en compréhension »* :** on peut ainsi écrire une instruction telle que `liste = [ f(i) for i in <ensemble> if test(i) ]`, qui pourrait se traduire par *« Python, fabrique-moi la liste des expressions `f(i)`, résultats des transformations de `i` par la fonction `f`, pour `i` parcourant l'objet Python `<ensemble>`, seulement si `i` vérifie la condition `test(i)` »*.

  Il faut naturellement que les fonctions `f` et `test` ainsi que l'objet `<ensemble>` soient définis au préalable, `test` devant renvoyer un **booléen** (`True` ou `False`).
- **Il y a deux approches principales pour réaliser un parcours de liste Python :**
  - on commence par énumérer l'ensemble des *indices* (les positions des éléments dans la liste Python) avec une boucle telle que `for i in range(len(liste)):`, puis on obtient la *valeur* mémorisée à la position `i` avec l'instruction `valeur = liste[i]` ;
  - on peut choisir de parcourir directement l'ensemble des *éléments* de la liste Python avec une instruction du type `for valeur in liste:` ;
  - on peut parcourir une liste en ayant connaissance *à la fois* de la position et de la valeur située à cette position : `for pos, val in enumerate([10, 20, 30]): print(pos, "->", val)`.
- ⚠ **Copier une liste** n'est pas aussi simple qu'on peut le penser, en Python. En effet, les variables Python n'ont pas « réellement » de contenu : ce sont des *références* vers des *objets*. On ne peut donc pas se contenter d'écrire `liste2 = liste1`, car `liste1` et `liste2` désigneraient la *même* liste (modifier `liste1` ou `liste2` revient au même !). **On peut faire** `liste2 = liste1[:]` **ou** `liste2 = [x for x in liste1]`.
- ⚠ **Cette technique n'est pas suffisante pour copier une liste de listes** (une liste dont chaque élément est lui-même une liste). Ainsi, avec `liste1 = [[10, 20], [30, 40]]` et `liste2 = liste1`, `liste1[0][1]` désigne 20 mais `liste2[0][1] = 0` modifiera *aussi* `liste1`. Pour éviter cette situation, il faut utiliser la commande `deepcopy` du module `copy` : `import copy ; liste3 = copy.deepcopy(liste1)`.
- **Quelques autres fonctionnalités des listes Python**, qu'il peut être utile de connaître.
  - L'instruction `liste.pop(i)` **renvoie et supprime** l'élément situé à la position `i` de `liste`. Sans paramètre, `liste.pop()` renvoie et supprime le **dernier** élément de `liste`.
  - L'instruction `liste.insert(i, valeur)` insère `valeur` à la position `i` de la liste (la valeur présente auparavant à la position `i` et les suivantes se trouvent décalées vers la *droite*).
  - Effacer l'élément situé à la position `i` de `liste` se fait avec l'instruction `del liste[i]`.

- Effacer le **premier** élément `valeur` *qui sera rencontré en parcourant* `liste` (à partir de la position `0`) se fait avec l'instruction `liste.remove(valeur)`.
- Trier une liste se fait à l'aide de l'instruction `liste.sort()`. **La liste est modifiée en place !**
- « Renverser » une liste se fait **en place** à l'aide de l'instruction `liste.reverse()` : si `liste = [1, 2, 3]`, après l'instruction `liste.reverse()`, la valeur de `liste` sera `[3, 2, 1]`.
- Extraire une sous-liste se fait à l'aide de la technique du *« slicing »* (explicitement hors-programme) : l'instruction `liste[a:b:c]` renverra une nouvelle liste formée à partir des éléments de `liste` situés à partir de la position `a` incluse jusqu'à la position `b` **exclue**, par sauts de `c` en `c`. Dit autrement :
  - `a` désigne la position initiale (celle à laquelle l'extraction débutera ; `a` vaut `0` s'il n'est pas précisé) ;
  - `b` désigne la position **qui suit** celle ou s'achèvera l'extraction (non précisé, `b` vaut `len(liste)`) ;
  - `c` désigne un *décalage* entre chaque extraction de valeur (`c` vaut `1` s'il n'est pas précisé) ;
  - si `a`, `b` ou `c` ne sont pas définis, il faut tout de même laisser les symboles « : » utiles ! Ainsi, `liste[1::2]` extrait les valeurs d'une liste présentes aux positions impaires.

  Remarque *et* exemple : les paramètres `a`, `b` et `c` peuvent prendre des valeurs négatives (en maintenant l'ordre `a < b`). Une position négative est comptée à partir de la fin de la liste : `liste[-1]` renvoie le *dernier* élément de `liste`, `liste[-len(liste)]` renvoie le *premier* élément de `liste`. Ainsi, on fabrique une *nouvelle* liste en renversant l'ordre des éléments de `liste` avec l'instruction `liste2 = liste[::-1]`).
- **Une caractéristique importante des listes Python**, qu'il est utile de connaître : l'accès, l'ajout (à droite) ou la modification d'un élément s'effectue *en temps constant*. Cela signifie que, *quelle que soit la taille de la liste*, récupérer ou modifier n'importe laquelle de ses valeurs prendra quasiment le même temps (très court). On dit que la complexité en temps est en $O(1)$.

  L'insertion ou la suppression d'un élément à une position donnée, ou encore le parcours d'une liste s'effectuent en $O(n)$, ce qui signifie que le temps de parcours est proportionnel à n, qui désigne par convention la taille de la liste (c'est-à-dire son nombre d'éléments).

Exemples de compréhensions de listes

▶ Liste des multiples de 3 inférieurs strictement à 100 :

```
multiples_de_3 = [3*i for i in range(33)]
```

▶ Liste des multiples de 6 construite à partir de la liste précédente :

```
multiples_de_6 = [2*j for j in multiples_de_3]
```

▶ Liste des multiples de 6 inférieurs à 70 et qui ne sont pas multiples de 5 :

```
def test(n):                    # n%5 donne le reste de la division entière de n par 5
    return (n < 70) and (n%5 != 0)      # si n%5 vaut 0, alors n est multiple de 5
multiples_de_6_mais_pas_de_5 = [2*j for j in multiples_de_3 if test(2*j)]
print(multiples_de_6_mais_pas_de_5)     # Donne [6, 12, 18, 24, 36, 42, 48, 54, 66]
```

▶ Une fonction qui renvoie la liste des diviseurs d'un entier :

```
def diviseurs(n):
    return [k for k in range(1, n+1) if n%k == 0]
print(diviseurs(1431))                  # Affichera [1, 3, 9, 27, 53, 159, 477, 1431]
```

*Notez que `range(1, n+1)` est nécessaire, autrement `k` prendrait des valeurs entières comprises entre `0` et `n-1` (le deuxième paramètre d'une instruction `range` étant toujours **exclu**).*

▶ Une liste de listes, donnant les résultats des tables de multiplications :

```
tables_mul = [[i*j for j in range(1, 11)] for i in range(1, 11)]
```

▶ Calculer la somme des multiples de 5 compris entre 120 et 610 :

```
sum([i for i in range(120, 611) if i%5 == 0])
```

On parcourt tous les entiers compris entre 120 et 610 (611 exclus), en ne conservant pour les additionner que ceux qui sont multiples de 5 (c'est-à-dire ceux dont le reste de la division euclidienne par 5 est nul).