Entrée[1]:
```
1 import numpy as np
```

 # TP Python 5 : focus sur les listes 

Partie I : Introduction

I.1) : Manipuler les listes

- Lorsqu'on a besoin de garder et d'utiliser plusieurs variables au cours d'un programme, on utiliser des **listes** pour les stocker.
- Une liste est une collection d'objets de différents types (variables numérique, chaînes de caractères, booléens, listes, etc.).
- Une liste est une variable que l'on peut nommer comme on le souhaite.
- Une liste a une taille **variable**!

Dans les exemples ci-dessous, la variable liste est nommée `L` :

Description	Code
Liste vide	`L = []`
Liste contenant des objets de types différents	`L = [3, "lundi", 5.6, True]` `L = ["mardi", 4.5, [7, 8, 9], 45.1]`
Premier objet de la liste ou objet de rang 0	`L[0]`
k+1-ième objet de la liste ou objet de rang k	`L[k]`
Dernier objet de la liste	`L[-1]`
Longueur ou nombre d'objets d'une liste	`len(L)`
Ajouter un objet en fin de liste	`L.append(objet)`
Insérer un objet au rang k	`L.insert(rang, objet)`
Concaténer (rassembler) deux listes	`L1 + L2`

Remarque:

- Les listes contiennent des objets de types différents, contrairement aux chaînes de caractères qui ne contiennent que des objets de type chaîne.
- La boucle bornée **for** *element* **in liste** possède la même syntaxe que pour la chaîne de caractères. La variable *element* va prendre les valeurs successives des objets de la liste (cf. parcourir une liste).

Exemple :

```
Liste1 = []
Nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Mois = ["janvier", "février", "mars", "avril", "mai", "j
uin", "juillet", "août", "septembre", "octobre", "novemb
re","décembre"]
```

- Une liste est donc une variable dans laquelle les données sont placées entre crochet.
  - La première liste créée s'appelle **Liste1** et est **vide**.
  - La deuxième liste s'appelle **Nombres** et comprend 12 élèments, les nombres de 0 à 12.
  - La troisième liste s'appelle **Mois** et comprend 12 élèments, des chaînes de caractères, les douze mois de l'année.
- Nous avons créé ces listes à l'aide d'une saisie manuelle de chacun de ses éléments, on dit que la **liste a été créée en extension**

I.2) Accéder aux éléments d'une liste

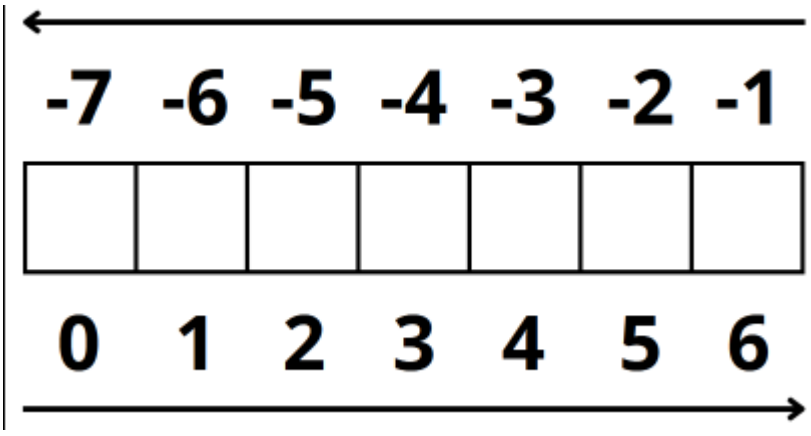
Avec les listes définies dans l'exemple précédent :

- `Nombres[0]` renvoie 1, car 1 est le premier élément de la liste **Nombres**
- `Nombres[3]` renvoie 4, car 4 est le quatrième élément de la liste **Nombres**

Attention ! Il y a un décalage car on commence au rang 0

- `Nombres[-1]` renvoie 12, car 12 est le dernier élément de la liste **Nombres**
- `Nombres[-2]` renvoie 11, car 11 est l'avant dernier élément de la liste **Nombres**

Illustration du principe de lecture des éléments :



- **Dans le sens de lecture :** le premier élément est accessible avec l'indice 0.
- **Dans le sens inverse de lecture :** le premier élément est accessible avec l'indice -1.

I.3) : Créer une liste par ajouts successifs

- La méthode `append` permet d'**ajouter un nouvel élément en fin de liste**.
- Elle s'utilise selon la syntaxe : `liste.append(nouvel_element)`

Exemple :

```
Nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Nombres.append(13)
```

On vient d'ajouter le nouvel élément : 13 à la fin de la liste.

Ainsi, le liste devient :

```
Nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Remarques :

- Une liste est ordonnée : `[1, 2, 3, 4, 5]` $\neq$ `[2, 1, 3, 4, 5]`
- Une liste peut contenir plusieurs fois le même élément : `[1, 2, 2, 2, 3]` $\neq$ `[1, 2, 3]`

À vous de faire :

Saisir dans la console la liste : `Nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`

À l'aide d'une boucle, compléter la liste des entiers jusqu'à 31.

```
Entrée[12]: 1
```

```
Sortie[12]: ['chat', 2, 2]
```

→Cliquez-ici pour la réponse←

```
Entrée[ ]: 1
```

À vous de faire :

En partant d'une liste vide, créer, par ajout successifs, la suite des nombres pairs de 0 à 100.

```
Entrée[ ]: 1
```

→Cliquez-ici pour la réponse←

Entrée[ ]:
```
1
```

Partie II : Différentes méthodes et opérateurs sur les listes

II.1) Quelques méthodes

- La méthode `extend` permet d'**ajouter à une liste, tous les éléments d'une autre liste**.
  - Elle s'utilise selon la syntaxe : `liste.extend(liste2)`

- La méthode `insert` permet d'**insérer un objet v à l'indice i**.
  - Elle s'utilise selon la syntaxe : `liste.insert(i,v)`

- La méthode `pop` permet de **supprimer l'élément d'indice i de la liste (par défaut le dernier) et retourne la valeur de l'élément supprimé**.
  - Elle s'utilise selon la syntaxe : `liste.pop(i)`

- La méthode `remove` permet de **supprimer la première valeur v dans s**.
  - Elle s'utilise selon la syntaxe : `liste.remove(v)`

- La méthode `reverse` permet de **renverser l'ordre des éléments de la liste**.
  - Elle s'utilise selon la syntaxe : `liste.reverse()`

Exemples :

- Avec `liste2 = [51, 91]` et `liste2.extend([24, 33])`, on obtient `[51, 91, 24, 33]`.
- Ensuite avec `liste2.insert(0,55)`, on obtient `[55, 51, 91, 24, 33]`.
- Puis, avec `liste2.pop(0)`, on obtient `[51, 91, 24, 33]`.
- Puis, avec `liste2.pop()`, on obtient `[51, 91, 24]`.
- Puis, avec `liste2.extend([1,2,1])`, on obtient `[51, 91, 24, 1, 2, 1]`.
- Puis, avec `liste2.remove(1)`, on obtient `[51, 91, 24, 2, 1]`.
- Et enfin, avec `liste2.reverse()`, on obtient `[1, 2, 24, 91, 51]`.

II.1) Quelques opérateurs

- L'opérateur `len` permet de **connaître la longueur de la liste**.
  - Elle s'utilise selon la syntaxe : `len(liste)`

- L'opérateur `del` permet de **supprimer un ou des éléments de la liste**.
  - Elle s'utilise selon la syntaxe : `del liste[i]` ou `del liste[i:j]`

- L'opérateur `+` permet d'**ajouter un élément à une la liste**.
  - Elle s'utilise selon la syntaxe : `liste = liste + [v]`

Exemples :

- Avec `liste3 = ["haricot", "bettrave", "navet", "courgette", "asperge"]` .
- Avec `len(liste3)` , renvoie `5` .
- Avec `del liste3[2]` , on obtient `['haricot', 'bettrave', 'courgette', 'asperge']` .
- Puis, avec `del liste3[1:3]` , on obtient `['haricot', 'asperge']` .
- Et enfin, avec `liste3 = liste3+[1,2]` , on obtient `['haricot', 'asperge', 1, 2]` .

II.3) Parcourir les listes

Voici deux rédactions d'une fonction Somme.

Ces fonctions ont pour paramètre une liste L de nombres et elles renvoient pour résultat la somme des nombres de L.

```
def Somme1(L):
    longueur = len(L)
    S = 0
    for i in range (0,longueur):
        S += L[i]
    return S

def Somme2(L):
    S = 0
    for i in L:
        S += i
    return S
```

Les deux fonctions renvoient le même résultat. Cela permet d'illustrer le fait qu'on peut faire une boucle directement avec les éléments d'une liste.

Partie III : Liste en compréhension

III.1) Sans condition

En Python, la notion de liste de compréhension (ou compréhension de listes) représente une manière originale et très puissante de générer des listes.

La syntaxe de base comprend au moins une boucle `for` placée entre crochets, précédée d'une variable (qui peut ou non être la variable d'itération).

Exemples :

- `[i for i in range(10)]` renvoie la liste `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `[2 for i in range(5)]` renvoie la liste `[2, 2, 2, 2, 2]`

À vous de faire :

Créer une fonction `Ecarts` ayant pour paramètre une liste `L` de nombres et un nombre `e`.

Elle renvoie la liste des écarts `abs(x-e)` entre les nombres `x` de `L` et le nombre `e`.

Entrée[ ]:
```
1
```

→Cliquez-ici pour la réponse←

III.2) Avec condition

On peut aussi ajouter des conditions lors de la création d'une liste en compréhension.

Exemple :

`[i for i in range(100) if i % 2 !=0]` renvoie `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]`

Cette liste affiche les nombres impairs compris entre 0 et 100.

À vous de faire :

Créer une fonction `Proches` ayant pour paramètre une liste `L` de nombres et deux nombres `e` et `r`.

Elle renvoie la liste des éléments `x` de `L` tels que l'écart entre `x` et `e` soit inférieur ou égal à `r`.

Entrée[ ]:
```
1
```

→Cliquez-ici pour la réponse←

Partie IV : Exercices

Exercice 1 :
Considérons la suite $(p_n)_{n\in\mathbb{N}}$ définie par, pour tout entier naturel n :

$$\begin{cases} p_0 = 0.3 \\ p_{n+1} = 0.3 + 0.7p_n^2 \end{cases}.$$

Écrire une fonction qui renvoie la liste des n premiers termes de la suite.

Entrée[ ]:
```
1
```

Exercice 2 :
Considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par, pour tout entier naturel n :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{u_n}{1+u_n} \end{cases}.$$

Écrire une fonction qui renvoie la liste des n premiers termes de la suite.

Entrée[ ]:
```
1
```

Exercice 3 :

1. a. Écrire dans l'éditeur l'instruction :

   ```
   L=[2*i+1 for i in range(10)]
   ```

1. b. Quel est le contenu de la liste L ?
2. Créer la liste appelée `cubes` , des cubes des cinq premiers nombres impairs positifs.

Entrée[ ]:
```
1
```

Entrée[ ]:
```
1
```

Exercice 4 :

1. Écrire une fonction `produit` qui calcule le produit des éléments d'une liste de nombres donnée.
2. Écrire une fonction `moyenne` qui calcule la moyenne d'une liste L1 de valeurs dont les effectifs sont dans une liste L2.

Remarque : on peut définir au préalable une fonction somme pour compléter plus facilement la deuxième fonction.

Entrée[ ]:
```
1
```

Exercice 5 :
Soit L une liste d'entiers naturels. Écrire une fonction `pair(L)` pour qu'elle renvoie la liste P des entiers naturels pairs de L.

Entrée[ ]:
```
1
```