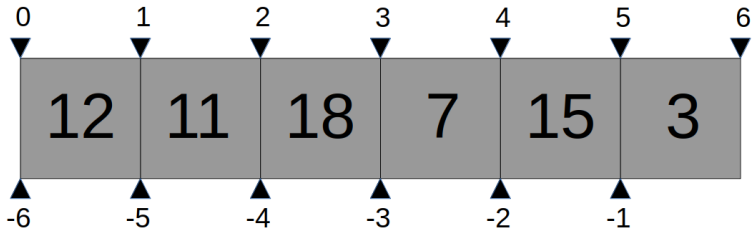
TP 4 : listes et boucles for

Listes en Python (suite)

On peut extraire une partie de la liste (ce qui donnera à nouveau une liste) en déclarant l'indice de début souhaité et l'indice de fin souhaité. J'appellerai ceci le « saucissonnage de liste »pour des raisons qui seront un peu plus claires ensuite. Pour extraire les éléments numéros trois quatre et cinq, on essaie naturellement de donner en indice de départ l'indice 2 (puisque les indices commencent à 0, l'indice 2 correspond bien au 3ème élément) et en indice de fin l'indice 4 (qui correspond bien au cinquième élément) :

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[2:4]
[18, 7]
```

À nouveau une surprise : nous n'obtenons pas le résultat attendu. En fait, il faut penser aux éléments comme des tranches de saucisson, et aux indices comme l'endroit où l'on doit mettre un coup de couteau pour obtenir la tranche souhaitée. Le petit schéma suivant représente ceci clairement.



On peut même se servir d'indices négatifs. Les résultats suivants doivent devenir clairs.

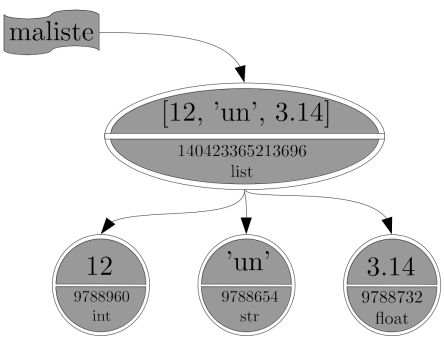
```
>>> liste[2:]
[18, 7, 15, 3]
>>> liste[:2]
[12, 11]
>>> liste[0:len(liste)]
[12, 11, 18, 7, 15, 3]
>>> liste[:]    # meme resultat
[12, 11, 18, 7, 15, 3]
>>> liste[2:5]
[18, 7, 15]
```

```
>>> liste[2:7]
[18, 7, 15, 3]
>>> liste[-2:-4]
[]
>>> liste[-4:-2]
[18, 7]
>>> liste[len(liste)-1]
3
>>> liste[-1]   # meme resultat
3
```

Important à noter : les dépassements d'indice sont tolérés lorsqu'on utilise le saucissonnage . Notez l'apparition de la liste vide `[]`, qui nous sera bien souvent utile. Notez finalement que l'indice -1 est l'indice du dernier élément, ce qui permet parfois d'éviter d'avoir recours au calcul de la longueur de la liste par `len`.

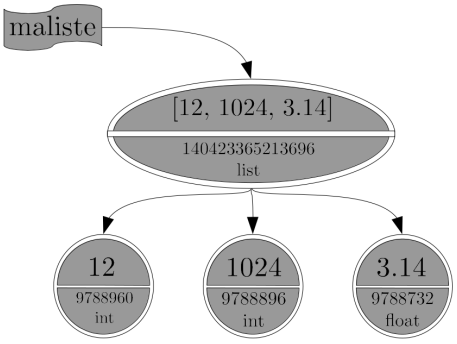
Intéressons nous maintenant à la position en mémoire des éléments d'une liste.

```
>>> maliste = [12, 'un', 3.14]
>>> maliste, id(maliste), type(maliste)
([12, 'un', 3.14], 140423365213696, list)
>>> maliste[0], id(maliste[0]), type(maliste[0])
(12, 9788960, int)
>>> maliste[1], id(maliste[1]), type(maliste[1])
('un', 9788654, str)
>>> maliste[2], id(maliste[2]), type(maliste[2])
(3.14, 9788732, float)
```



Une liste est en fait un carnet d'adresses : elle contient les emplacements mémoire des différents éléments de la liste. Une propriété remarquable des listes, propriété que ne possèdent pas les `tuple`, est le fait que l'on peut modifier un ou plusieurs éléments de la liste sans modifier son emplacement mémoire et donc son identifiant. Voici un exemple :

```
>>> maliste[1] = 2**10
>>> maliste, id(maliste), type(maliste)
([12, 1024, 3.14], 140423365213696, list)
>>> maliste[0], id(maliste[0]), type(maliste[0])
(12, 9788960, int)
>>> maliste[1], id(maliste[1]), type(maliste[1])
(1024, 9788896, str)
>>> maliste[2], id(maliste[2]), type(maliste[2])
(3.14, 9788732, float)
```



La liste n'a pas changé d'identifiant, on a simplement modifié sa **valeur** : le contenu de la liste est toujours à la même adresse dans la mémoire. Les concepteurs de Python on prévu ce comportement afin d'éviter que de grandes listes soient déplacées dans la mémoire à chaque modification d'un seul élément. On dit que le type `list` est **mutable**. Le type `tuple` n'est quand à lui pas mutable. Voyez par exemple ce que donne une tentative de modification d'un élément d'un `tuple` :

```
>>> a = (1,2,3,4)
>>> a, id(a), type(a)
((1, 2, 3, 4), 139650946155440, <class 'tuple'>)
>>> a[2] = 34
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
```

Une modification d'un objet **non mutable** aura pour conséquence la désignation d'un nouvel emplacement mémoire. Voyez par exemple dans l'exemple précédent le changement d'identifiant de `maliste[1]`. En effet cet objet est de type `int` donc non mutable. La modification de sa valeur entraîne la modification de son emplacement mémoire.

Les techniques de saucissonnage permettent de modifier les listes à souhait.

```
>>> liste = [12, 11, 18, 7, 5, 13]
>>> liste[2:5] = ['a', 'b', 'c']
>>> liste
```

Le problème des copies de listes

On peut vouloir créer une copie d'une liste. Mais cela nous amène à quelques surprises.

```
>>> liste = ['HEC', 'ESSEC', 'ESCP']
>>> copie = liste    # creation d'une copie
>>> liste[0] = 'EDHEC'   #modification de l'original
>>> liste, copie
>>> copie[1] = 'AUDENCIA'
>>> liste, copie
```

Une modification de la liste originale se propage à la copie, et une modification de la liste copie se propage à l'originale. Ce n'est pas conforme à nos habitudes Python : une affectation est une affectation une fois pour toute pas une identification comme en mathématiques. L'explication se trouve du côté des identifiants (c'est même ce problème de copies de listes qui nous oblige à nous pencher ainsi en détail sur ces questions d'identifiants).

Lors de l'affectation `copie = liste`, c'est l'adresse mémoire de la liste d'origine qui est stockée dans la nouvelle variable `copie`. Cette nouvelle variable pointe donc au même endroit dans la mémoire, et toute

modification d'une liste affecte l'autre liste au même moment. On dit que l'on a créé un **alias** de liste. Ce mécanisme a été mis en place afin d'éviter d'avoir à souvent recopier des grandes listes dans la mémoire, ce qui est très coûteux en temps de calcul. Pour tenter d'éviter ce problème, on peut extraire d'abord **tous les éléments** de la liste grâce à l'instruction `liste[:]` puis on affecte le résultat dans la nouvelle variable `copie`.

```
>>> copie = liste[:]
>>> copie[1]='nouveau'
>>> liste, copie
>>> liste[1] = 'new'
>>> liste, copie
```

Le résultat semble satisfaisant, mais en réalité ne l'est pas. On dit qu'on a créé une **copie superficielle** (en anglais : « shallow copy ») comme nous pouvons le voir sur les exemples suivants :

```
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x                  # creation d'un alias
>>> z = x[:]               # creation d'une copie superficielle
>>> x[0] = 'c'      # affectera x et y mais pas z
>>> z[2][2] = 765    # affecte z ET x ET y !!
>>> x, y, z
```

Le problème est le même que dans la copie superficielle : puisque le troisième élément de la liste `x` est une liste, donc une adresse, c'est cette adresse qui est copiée dans la copie superficielle `z`. Et donc une modification d'un élément de cette liste via `z` affectera bien `x` aussi. Pour éviter ce problème, on réalise des **copies profondes** (en anglais « deep copy »). On procède ainsi :

```
>>> import copy
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x                  # creation d'un alias
>>> z = x[:]               # creation d'une copie superficielle
>>> w = copy.deepcopy(x)  # copie profonde
>>> x[0] = 'c'      # affectera x et y mais pas z ni w
>>> z[2][2] = 765    # affecte x,y,z mais pas w
>>> x, y, z, w
```

Quelques outils pour les listes

Pour ajouter un élément à une liste, on utilise la méthode `append`. On procède ainsi :

```
>>> malist = [1, 2, 45, 'ok', 0.69]
>>> malist, id(malist)
>>> malist.append("toto")
>>> malist, id(malist)
```

C'est la première fois que nous rencontrons cette opération « point ». Certains objets Python (comme les listes) possèdent ainsi des **méthodes**. La méthode s'appelle **append**, on l'applique à l'objet `malist` avec comme paramètre effectif la chaîne de caractères `"toto"`. L'objet `malist` est alors modifié. Mais vous pouvez remarquer que son identifiant mémoire n'a pas changé (toujours pour économiser des déplacements en mémoire). Voici quelques méthodes utiles sur les listes (dans cet exemple la liste que l'on cherche à modifier s'appelle `malist`).

méthode	**effet**
`malist.append(x)`	ajoute l'élément `x` en fin de liste
`malist.extend(L)`	ajoute les éléments de `L` en fin de liste
`malist.insert(i, x)`	ajoute l'élément `x` en position i
`malist.remove(x)`	supprime la première occurence de `x`
`malist.pop(i)`	supprime l'élément d'indice i et le renvoie
`malist.index(x)`	renvoie l'indice de la première occurence de `x`
`malist.count(x)`	renvoie le nombre d'occurences de `x`
`malist.sort()`	modifie la liste en la triant
`malist.reverse()`	modifie la liste en inversant l'ordre de ses éléments

La méthode `sort` modifie la liste. Si l'on veut obtenir une nouvelle liste, triée, sans modification de l'ancienne, on peut avoir recours à la fonction **sorted**. Comparez les exécutions de ces deux codes.

```
>>> maliste = [78, 12, 0, 57]
>>> sorted(maliste)
>>> maliste
```

```
>>> maliste = [78, 12, 0, 57]
>>> maliste.sort()
>>> maliste
```

Finalement, nous avons déjà rencontré la fonction **len** qui renvoie la longueur d'une liste. On peut aussi tester l'appartenance d'un élément à une liste grâce à l'opérateur **in**.

```
>>> maliste = [78, 12, 0, 57]
>>> len(maliste)
>>> 5 in maliste
>>> 0 in maliste
```

Boucles for

Si l'on sait à l'avance combien de fois on doit répéter un bloc d'instructions, on peut se servir d'un compteur de boucle qui permet de sortir de la boucle `while`. Par exemple si l'on veut afficher 5 fois le même texte on procède ainsi

```
k = 1
while (k <= 5):
    print('Ce texte sera affiche 5 fois (ici k =', k, ')',sep='')
    k += 1
```

Python (et la plupart des langages de programmation) possèdent une instruction qui permet d'écrire le même type de code de manière beaucoup plus compacte. L'exemple précédent s'écrirait ainsi :

```
for k in range(5):
    print('Ce texte sera affiche 5 fois (ici k =', k, ')',sep='')
```

Après l'instruction **for**, on écrit le nom du compteur de boucle que l'on souhaite utiliser puis **in range(n)**, où n est le nombre de fois que l'on souhaite exécuter le bloc d'instructions. On termine par deux points pour signaler l'ouverture d'un bloc indenté. Au premier passage dans la boucle, ce compteur est initialisé. À chaque passage dans la boucle il est incrémenté, par défaut de 1. L'instruction **range** précise toutes les valeurs que prendront le compteur de boucle au fur et à mesure des passages dans la boucle. On voit sur l'exemple précédent que `range(5)` fait prendre au compteur toutes les valeurs de 0 à 4. La variable compteur (`k` dans l'exemple ci-dessus) n'est pas détruite à la fin de la boucle. On peut le vérifier avec ce petit exemple, qui affiche les quinze premiers multiples de 19 :

```
>>> for k in range(15):
....     print(k*19, end = ', ')
....
>>> k, type(k)
14, int
```

On voit aussi qu'avec l'instruction **range**, le compteur est un entier. Dans un souci d'optimisation de la mémoire, l'instruction **range** ne créée pas une liste d'entiers dans laquelle l'instruction `for` ira piocher au fur et à mesure. Elle les créée les uns après les autres, afin de ne pas surcharger la mémoire. L'instruction `range(n)` distribue les entiers de $[\![0; n-1]\!]$. L'instruction `range(p,n)` distribue les entiers de $[\![p\ ;\ n-1]\!]$, et finalement l'instruction `range(p, n, k)` parcourt les entiers de $[\![p\ ;\ n-1]\!]$ mais en sautant de k en k. Un point important : `range(p, n, k)` n'est pas une liste. Mais on peut créer une liste d'entiers en utilisant l'instruction **list**. Essayez les instructions suivantes

```
>>> type(range(5))
>>> x=range(5)
>>> x.append(15)
>>> type(list(range(5)))
>>> list(range(10))
>>> list(range(1, 19))
>>> list(range(0, 20, 5))
>>> list(range(0, 10, 3))
>>> list(range(0, -6, -1))
>>> list(range(1, 0))
>>> list(range(0))
>>> list(range(0, 1, -1))
```

Dans l'instruction `for`, on peut utiliser d'autres objets que `range`. Il faut que l'objet en question soit d'un type qu'on appelle séquentiel. Les types séquentiels que nous avons rencontrés sont les suivants : les listes (`list`), les tuples (`tuple`) et les chaînes de caractères (`str`). Par exemple on peut faire un boucle for dans laquelle on parcourt tous les éléments d'une chaîne de caractères :

```
>>> for i in 'longue chaîne de caractères':
....     print(i, end = ' -> ')
....
>>> for el in ['salut', 17, 18.15]:
....     print(el)
....
```

Finalement pour créer des listes, Python nous fournit un outil très pratique, très similaire aux mathématiques, qui définit des listes en compréhension (comme dans les ensembles mathématiques : l'ensemble des $f(x)$ avec $x \in E$). La syntaxe pour définir une telle liste est la suivante :

```
[ f(x) for x in liste ]
```

à rapprocher de la syntaxe mathématique

$$\{f(x)\ ,\ x \in E\}$$

Voici quelques exemples :

```
>>> liste = [2, 4, 6, 8, 10]
>>> [3*x for x in liste]
[6, 12, 18, 24, 30]
>>> [[x, x**3] for x in liste]
[[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
>>> [3*x for x in liste if x > 5]
[18, 24, 30]
>>> [3*x for x in liste if x**2 < 50]
[6, 12, 18]
>>> liste2 = list(range(3))
>>> [x**y for x in liste for y in liste2]
```

Exercice 1.

1. Écrire une fonction `somme(liste)` qui renvoie la somme des éléments de la liste d'entiers `liste`.
2. Créez en une seule ligne une liste nommée `carres` qui contient les carrés des 100 premiers entiers. Utilisez la pour vérifier votre résultat, en comparant avec `sum(carres)`.
3. Écrire de même une fonction `maximum` qui renvoie le maximum d'une liste d'entiers.
   On pourra poser une variable `m` prenant pour valeur le premier élément de la liste, puis parcourir la liste en modifiant `m` au fur et à mesure si l'on rencontre des éléments plus grands que `m` dans la liste.
4. (a) Créez en une seule ligne une liste de 100 réels uniformément répartis entre 0 et 3, nommée `antecedents`.
   (b) Créez en une seule ligne une liste nommee `images` qui contient les images par $x \mapsto x^2/(x^3+1)$ des éléments de `antecedents`.
   (c) Comparez le maximum de cette liste trouvé via votre fonction et via `max(images)`.

Exercice 2.

1. Codez une fonction `mon_count(liste, x)` qui renvoie le nombre d'occurences de `x` dans la liste `liste`, sans utiliser `count`.
2. Codez une fonction `mon_pop(liste, i)` qui renvoie la liste `liste` dans laquelle l'élément d'indice `i` a été effacé, sans utiliser `pop`.
3. Codez une fonction `mon_index(liste, x)` qui renvoie l'indice de la première occurence de `x` dans la liste `liste`, sans utiliser `index`. La fonction renverra -1 si aucune occurence n'a été trouvée.
4. Codez une fonction `mon_reverse` qui prend une liste en argument et renvoie cette liste mais écrite dans l'ordre inverse. On n'utilisera pas la fonction `reverse`.