Algorithmique et programmation 1

L1 Mathématiques - L1 Informatique

Semestre 1

Chapitre 5 - Les listes en Python : le type `list`

Introduction

Savez-vous résoudre les problèmes suivants ?

- Lire n nombres, puis les afficher dans l'ordre inverse
- Lire n nombres, puis les afficher dans l'ordre croissant
- Lire n nombres différents (et vérifier qu'ils le sont)
- Écrire une fonction qui renvoie les n premiers nombres premiers
- Stocker tous les $P(x)$ pour x variant de a à b par pas de $0,01$
- Écrire une fonction qui renvoie le minimum **et** le maximum d'une suite de nombre.

Nous allons voir un nouveau type de données pour résoudre ce genre de problèmes : le type `list`

Objectif : désigner avec une seule variable une collection de valeurs
Liste : suite **indexée** (numérotée) d'objets quelconques (type `list` en python)

- Élements "rangés" dans des "cases" numérotées de 0 à $n-1$
- En mémoire : tableau à n cases, chacune contenant une référence (*"flèche"*) vers un objet
- Peut contenir des objets de plusieurs types différents
- **Mutable** : peut être modifiée, agrandie, raccourcie...

Création et affichage

Création : suite entre `[` et `]` d'expressions séparées par `,`

In [1]:

```
lst = [3, 'toto', 4.5, False, None]
print(lst)
```

```
[3, 'toto', 4.5, False, None]
```

Liste vide `[]` : liste ne contenant aucun objet

In [2]:

```
lst = []
print(lst)
```

```
[]
```

Une liste peut contenir d'autres listes !

In [3]:

```
lst = ['test', [1, [2], 3]]
print(lst)
```

```
['test', [1, [2], 3]]
```

Exemple (Python tutor)

(exemple filé à exploiter tout au long du cours)

Opérations et fonctions de base

Longueur d'une liste

La longueur d'une liste (le nombre d'éléments qu'elle contient) s'obtient par la fonction `len`.

In [4]:

```
lst = [3, 'toto', 4.5, False, None]
print(len(lst))
```

```
5
```

In [5]:

```
print(len([]))
```

```
0
```

Accès aux éléments

Les éléments d'une liste à n éléments sont numérotés de 0 à $n-1$.
Le numéro d'un élément est appelé son *indice*.
L'accès à un élément donné s'appelle l'**indexation**.

In [6]:

```
lst = [3, 'toto', 4.5]
print(lst[1])
```

```
toto
```

Attention !

- Le premier élément d'une liste est l'élément d'indice `0` !
- Si la liste a `n` éléments, il n'existe pas d'élément d'indice `n` !
- L'accès à un indice supérieur ou égal à la taille de la liste provoque une erreur !

In [7]:

```
lst = [3, 'toto', 4.5]
print(lst[3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[7], line 2
      1 lst = [3, 'toto', 4.5]
----> 2 print(lst[3])

IndexError: list index out of range
```

Exercice : Écrire une fonction qui affiche tous les éléments d'une liste (un par ligne)

In [8]:

```
def affiche_elements(lst):
    ...

lst = [3, 'toto', 4.5]
affiche_elements(lst)
```

In [9]:

```
def affiche_elements(lst):
    i = 0
    while i < len(lst):
        print(lst[i])
        i = i + 1


lst = [3, 'toto', 4.5]
affiche_elements(lst)
```

```
3
toto
4.5
```

Modification d'un élément

On peut modifier le i-ème élément de `lst` à l'aide d'une affectation :

In [10]:

```
lst = [3, 'toto', 4.5, False, None]
print(lst[2])
lst[2] = 'titi'
print(lst)
```

```
4.5
[3, 'toto', 'titi', False, None]
```

Attention, ceci ne crée pas une nouvelle liste mais modifie la liste sur place !

In [11]:

```
lst = [3, 'toto', 4.5, False, None]
lst_bis = lst
lst[2] = 'titi'
lst_bis
```

Out[11]:

```
[3, 'toto', 'titi', False, None]
```

Concaténation et répétition

Comme pour les chaînes de caractères (`str`) on peut utiliser les opérateurs `+` pour fabriquer la concaténation de deux listes et `*` pour répéter une liste.

In [12]:

```
[3, 'toto', 4.5] + [False, None]
```

Out[12]:

```
[3, 'toto', 4.5, False, None]
```

In [13]:

```
[] + [3, 'toto', 4.5] + []
```

Out[13]:

```
[3, 'toto', 4.5]
```

In [14]:

```
3 * ['a', 'b']
```

Out[14]:

```
['a', 'b', 'a', 'b', 'a', 'b']
```

In [15]:

```
[0] * 13
```

Out[15]:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

On peut utiliser ces opérateurs pour recopier une liste. Comparer :

In [16]:

```
lst = [3, 'toto', 4.5]
lst2 = lst
lst3 = lst + []
lst4 = lst * 1
```

Test d'appartenance

Exercice : Écrire une fonction recevant une liste et une valeur, et renvoyant `True` si la valeur apparaît dans la liste (`False` sinon)

In [17]:

```
def appartient(lst, val):
    ...
```

In [18]:

```python
def appartient(lst, val):
    i = 0
    while i < len(lst):
        if lst[i] == val:
            return True
        i += 1
    return False

lst = ['Hildegarde', 'Cunégonde', 'Médor']

print(appartient(lst, 'Cunégonde'))

if appartient(lst, 'Médor'):
    print('Bon chien !')
```

```
True
Bon chien !
```

Remarque : Cette fonctionnalité existe déjà en Python :

- `val in lst` vaut `True` si `val` apparaît dans `lst`, `False` sinon
- Réciproquement, on peut écrire `val not in lst`

In [19]:

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
'Cunégonde' in lst
```

Out[19]:

```
True
```

In [20]:

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
'Rex' not in lst
```

Out[20]:

```
True
```

In [21]:

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
if 'Médor' in lst:
    print('Bon chien !')
```

```
Bon chien !
```

Minimum et maximum d'une liste

Exercice : Écrire une fonction recevant une liste **non vide d'éléments comparables entre eux** et renvoyant la valeur du plus petit élément qui apparaît dans la liste

In [22]:

```
def minimum(lst):
    ...
```

Même question pour le plus grand élément

In [23]:

```
def maximum(lst):
    ...
```

In [24]:

```
def minimum(lst):
    res = lst[0]  # plante si lst == []
    i = 1
    while i < len(lst):
        if lst[i] < res:
            res = lst[i]
        i += 1
    return res
```

In [25]:

```
def maximum(lst):
    res = lst[0]
    i = 1
    while i < len(lst):
        if lst[i] > res:
            res = lst[i]
        i += 1
    return res
```

In [26]:

```
lst = [4, 6.6, 2, -7, 13, -6, 0]
print(minimum(lst), maximum(lst))
```

```
-7 13
```

Ces fonctionnalités existent déjà en Python : fonctions `min` et `max`.

In [27]:

```
lst = [4, 6.6, 2, -7, 13, -6, 0]
print(min(lst), max(lst))
```

```
-7 13
```

Attention : pour que cela fonctionne il faut que tous les éléments soient comparables !

In [28]:

```
lst = [3, 'toto', 4.5, False, None]
print(min(lst))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 2
      1 lst = [3, 'toto', 4.5, False, None]
----> 2 print(min(lst))

TypeError: '<' not supported between instances of 'str' and 'int'
```

Une erreur se produit aussi si l'on appelle ces fonctions sur une liste vide :

In [29]:

```
min([])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[29], line 1
----> 1 min([])

ValueError: min() iterable argument is empty
```

Somme des éléments d'une liste

Exercice : Écrire une fonction `somme(lst)` qui renvoie la somme des éléments d'une liste dont tous les éléments sont des nombres.

In [30]:

```
def somme(lst):
    ...
```

In [31]:

```
def somme(lst):
    res = 0
    i = 0
    while i < len(lst):
        res += lst[i]
        i += 1
    return res

print(somme([1, 2, 3]))
```

```
6
```

La fonction prédéfinie `sum` de Python réalise également ce calcul :

In [32]:

```
sum([1, 2, 3])
```

Out[32]:

```
6
```

Attention : la fonction `sum` provoque une erreur si un élément de la liste n'est pas un nombre !

In [33]:

```
sum([3, 'toto', 4.5, False, None])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 1
----> 1 sum([3, 'toto', 4.5, False, None])

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Manipulations plus complexes : méthodes

On va maintenant énumérer un certain nombre de méthodes prédéfinies sur les listes, permettant des modifications plus complexes. Pour plus de détails, on pourra consulter la **documentation en ligne**.

Agrandir ou rétrécir une liste

Plusieurs instructions ont un effet sur la taille de la liste :

- L'instruction `lst.append(elem)` ajoute l'élément `elem` à la fin de la liste `lst`
- L'instruction `lst.pop()` supprime le dernier élément de `lst` et renvoie sa valeur
- L'instruction `lst.pop(i)` supprime l'élément d'indice `i` de `lst` et renvoie sa valeur

*Les fonctions `append` et `pop` sont appelées **méthodes**, ou fonctions s'appliquant à un objet (nous en verrons d'autres dans les cours suivants)*

Attention, ces instructions ne créent pas une nouvelle liste mais modifient la liste sur place !

Attention, ne pas confondre `x = lst[2]` et `x = lst.pop(2)` !

In [34]:

```
lst = [3, 'toto', 4.5, False, None]
lst_bis = lst

lst.append(1)
print(lst_bis)

elem = lst_bis.pop(2)
print(elem)

print(lst)
```

```
[3, 'toto', 4.5, False, None, 1]
4.5
[3, 'toto', False, None, 1]
```

Exercice : écrire une fonction recevant deux listes et ajoutant tous les éléments de la seconde à la fin de la première

In [35]:

```
def etend_liste(une_liste, autre_liste):
    ...
```

- Attention, pas de `return` : `une_liste` doit être modifiée sur place !
- Attention, on ne doit pas modifier `autre_liste` !

In [36]:

```
def etend_liste(une_liste, autre_liste):
    i = 0
    while i < len(autre_liste):
        une_liste.append(autre_liste[i])
        i += 1

lst = [3, 'toto', 4.5]
lst2 = [False, None]

print(lst)
print(lst2)
etend_liste(lst, lst2)
print(lst)
print(lst2)
```

```
[3, 'toto', 4.5]
[False, None]
[3, 'toto', 4.5, False, None]
[False, None]
```

- Cette fonctionnalité existe déjà en Python : `une_liste.extend(autre_liste)`

In [37]:

```
lst = [3, 'toto', 4.5]
lst_bis = lst
lst2 = [False, None]

lst.extend(lst2)
print(lst)
print(lst_bis)
```

```
[3, 'toto', 4.5, False, None]
[3, 'toto', 4.5, False, None]
```

In [38]:

```
lst = [3, 'toto', 4.5]
lst_bis = lst
lst2 = [False, None]

lst = lst + lst2
print(lst)
print(lst_bis)
```

```
[3, 'toto', 4.5, False, None]
[3, 'toto', 4.5]
```

Exercice : Écrire une fonction recevant deux listes en argument et renvoyant une nouvelle liste contenant tous les éléments de la première suivis de tous les éléments de la seconde, à la manière de l'opérateur `+`.

In [39]:

```
def concatene(lst1, lst2):
    ...
```

In [40]:

```
def concatene(lst1, lst2):
    res = []
    i = 0
    while i < len(lst1):
        res.append(lst1[i])
        i += 1
    i = 0
    while i < len(lst2):
        res.append(lst2[i])
        i += 1
    return res
```

In [41]:

```
def concatene(lst1, lst2):
    res = []
    res.extend(lst1)
    res.extend(lst2)
    return res
```

In [42]:

```
concatene([3, 1, 67], [2, 4])
```

Out[42]:

```
[3, 1, 67, 2, 4]
```

Exercice : Écrire une fonction recevant une liste `lst` et un entier `n` en argument et renvoyant une nouvelle liste contenant les éléments de `lst` répétés `n` fois à la manière de l'opérateur `*`.

In [43]:

```
def repete(lst, n):
    ...
```

In [44]:

```
def repete(lst, n):
    res = []
    i = 0
    while i < n:
        j = 0
        while j < len(lst):
            res.append(lst[j])
            j += 1
        i += 1
    return res
```

In [45]:

```
def repete(lst, n):
    res = []
    i = 0
    while i < n:
        res.extend(lst)
        i += 1
    return res
```

In [46]:

```
repete([4, 5, 6], 3)
```

Out[46]:

```
[4, 5, 6, 4, 5, 6, 4, 5, 6]
```

Rechercher la position d'un élément

Exercice : Écrire une fonction renvoyant le plus petit indice où apparaît un élément `x` dans une liste `lst` (on renverra `None` si `x` n'apparaît pas dans la liste)

In [47]:

```
def chercher(lst, x):
    ...
```

In [48]:

```python
def chercher(lst, val):
    i = 0
    while i < len(lst):
        if lst[i] == val:
            return i
        i += 1
    return None

lst = [3, 'toto', 4.5, False, None, 4.5]
indice = chercher(lst, 4.5)
print(indice)
indiceNone = chercher(lst, 'AP1')
print(indiceNone)
```

```
2
None
```

Cette fonction existe déjà en Python : méthode `index`

In [49]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.index(4.5)
```

Out[49]:

```
2
```

Attention, cette méthode provoque une erreur si l'élément à retirer n'est pas dans la liste !

In [50]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.index(3.5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[50], line 2
      1 lst = [3, 'toto', 4.5, False, None, 4.5]
----> 2 lst.index(3.5)

ValueError: 3.5 is not in list
```

Compter le nombre d'occurrences d'un élément

Exercice : Écrire une fonction renvoyant le nombre de fois où apparaît un élément `x` dans une liste `lst`

In [51]:

```
def compter(lst, x):
    ...
```

In [52]:

```
def compter(lst, x):
    cpt = 0
    i = 0
    while i < len(lst):
        if lst[i] == x:
            cpt += 1
        i += 1
    return cpt

lst = [3, 'toto', 4.5, False, None, 4.5]
print(compter(lst, 4.5))
```

```
2
```

Cette fonction existe déjà en Python : méthode `count`

In [53]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.count(4.5)
```

Out[53]:

```
2
```

Vider entièrement une liste

Exercice : Écrire une fonction supprimant tous les éléments de la liste `lst`

In [54]:

```
def vider(lst, x):
    ...
```

In [55]:

```
def vider(lst):
    while len(lst) > 0:
        lst.pop()

lst = [3, 'toto', 4.5, False, None, 4.5]
print(lst)
vider(lst)
print(lst)
```

```
[3, 'toto', 4.5, False, None, 4.5]
[]
```

Cette fonction existe déjà en Python : méthode `clear`

In [56]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.clear()
print(lst)
```

```
[]
```

Renverser une liste

Exercice : Écrire une fonction renversant l'ordre des éléments d'une liste `lst`

In [57]:

```
def renverser(lst, x):
    ...
```

In [58]:

```
def renverser(lst):
    i = 0
    j = len(lst) - 1
    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i += 1
        j -= 1

lst = [3, 'toto', 4.5, False, None, 4.5]
renverser(lst)
print(lst)
```

```
[4.5, None, False, 4.5, 'toto', 3]
```

Cette fonction existe déjà en Python : méthode `reverse`

In [59]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.reverse()
print(lst)
```

```
[4.5, None, False, 4.5, 'toto', 3]
```

Retirer un élément donné

Exercice : Écrire une fonction retirant la première occurrence d'un élément `x` dans une liste `lst` (on ne fera rien si la liste ne contient pas `x`)

In [60]:

```
def retirer(lst, x):
    ...
```

In [61]:

```
def retirer(lst, x):
    i = chercher(lst, x)
    if i is not None:
        while i < len(lst)-1:
            lst[i] = lst[i+1]
            i += 1
        lst.pop()

lst = [3, 'toto', 4.5, False, None, 4.5]
print(lst)
retirer(lst, 4.5)
print(lst)
```

```
[3, 'toto', 4.5, False, None, 4.5]
[3, 'toto', False, None, 4.5]
```

In [62]:

```
def retirer2(lst, x): # version utilisant pop(i)
    i = chercher(lst, x)
    if i is not None:
        lst.pop(i)

lst = [3, 'toto', 4.5, False, None, 4.5]
print(lst)
retirer2(lst, 4.5)
print(lst)
```

```
[3, 'toto', 4.5, False, None, 4.5]
[3, 'toto', False, None, 4.5]
```

Cette fonction existe déjà en Python : méthode `remove`

In [63]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.remove(4.5)
print(lst)
```

```
[3, 'toto', False, None, 4.5]
```

Attention, cette méthode provoque une erreur si l'élément à retirer n'est pas dans la liste !

In [64]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.remove(3.5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[64], line 2
      1 lst = [3, 'toto', 4.5, False, None, 4.5]
----> 2 lst.remove(3.5)

ValueError: list.remove(x): x not in list
```

Ajouter un élément donné à un indice donné

Exercice : Écrire une fonction qui insère un élément `x` à la position `i` de la liste `lst` (si `i` est trop grand, la fonction insère `x` à la fin de `lst` ; si `i` est trop petit, elle insère `x` au début de `lst`)

In [65]:

```
def ajouter(lst, i, x):
    ...
```

In [66]:

```
def ajouter(lst, i, x):
    # determine une valeur correcte pour i
    i = max(0, min(i, len(lst)))
    # on ajoute une case vide à la fin
    lst.append(None)
    # decalage des cases sur la droite à partir de i
    j = len(lst) - 1
    while j > i:
        lst[j] = lst[j-1]
        j -= 1
    lst[i] = x # on insère x à l'indice i

lst = [3, 'toto', 4.5, False, None, 4.5]
print(lst)
ajouter(lst, 3, 79) # indice ok
print(lst)
ajouter(lst, -6, "petit") # indice trop petit
print(lst)
ajouter(lst, 16, "grand") # indice trop grand
print(lst)
```

```
[3, 'toto', 4.5, False, None, 4.5]
[3, 'toto', 4.5, 79, False, None, 4.5]
['petit', 3, 'toto', 4.5, 79, False, None, 4.5]
['petit', 3, 'toto', 4.5, 79, False, None, 4.5, 'grand']
```

Cette fonction existe déjà en Python : méthode `insert`

Attention, cette méthode insère l'élément à la fin si l'indice est supérieur à `len(lst)` , et au début si l'indice est négatif !

In [67]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
print(lst)
lst.insert(3, 79)
print(lst)
```

```
[3, 'toto', 4.5, False, None, 4.5]
[3, 'toto', 4.5, 79, False, None, 4.5]
```

In [68]:

```
lst.insert(-8, "petit")
print(lst)
lst.insert(16, "grand")
print(lst)
```

```
['petit', 3, 'toto', 4.5, 79, False, None, 4.5]
['petit', 3, 'toto', 4.5, 79, False, None, 4.5, 'grand']
```

Trier une liste

Enfin, il est possible de trier le contenu d'une liste avec la méthode `sort`. Programmer ce genre de fonctions fait partie des objectifs du semestre 2.

In [69]:

```
lst = [4, 6.6, 2, -7, 13, -6, 0]
lst.sort()
print(lst)
```

```
[-7, -6, 0, 2, 4, 6.6, 13]
```

Attention, cette méthode ne fonctionne pas si les éléments ne sont pas tous comparables !

In [70]:

```
lst = [3, 'toto', 4.5, False, None, 4.5]
lst.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[70], line 2
      1 lst = [3, 'toto', 4.5, False, None, 4.5]
----> 2 lst.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

Notez que la méthode `sort` modifie définitivement la liste (une nouvelle liste n'est pas créée). Il existe aussi une fonction permettant de fabriquer une copie triée d'une liste : la fonction `sorted`.

In [71]:

```
lst = [4, 6.6, 2, -7, 13, -6, 0]
print(sorted(lst))
print(lst)
```

```
[-7, -6, 0, 2, 4, 6.6, 13]
[4, 6.6, 2, -7, 13, -6, 0]
```

Récapitulatif

Opérateurs sur les listes

opérateur	effet
`lst[i]` (dans une expression)	élément d'indice `i` de `lst`
`lst[i] = expr`	modifie l'élément d'indice `i` de `lst`
`lst1 + lst2`	concaténation (nouvelle liste)
`lst * n`	répétition (nouvelle liste)
`x in lst`	`True` si `x` apparaît dans `lst`
`x not in lst`	`True` si `x` n'apparaît pas dans `lst`

$\star$: erreur si `i` n'est pas un indice correct de `lst`

Fonctions sur les listes

fonction	effet
`len(lst)`	renvoie la longueur de `lst`
`min(lst)`	renvoie le plus petit élément de `lst` ⋆♡
`max(lst)`	renvoie le plus grand élément de `lst` ⋆♡
`sum(lst)`	renvoie la somme des éléments de `lst` ♣
`sorted(lst)`	renvoie une copie triée de `lst` ♡

⋆ : erreur si `lst` est vide

♡ : erreur si `lst` contient des éléments incomparables

♣ : ne fonctionne que sur des listes de *nombres*

Méthodes qui modifient la liste

méthode	effet
`lst.append(x)`	ajoute `x` à la fin de `lst`
`lst.extend(lst2)`	ajoute les éléments de `lst2` à la fin de `lst`
`lst.insert(i, x)`	ajoute `x` à l'indice `i` dans `lst`
`lst.remove(x)`	retire la première occurrence de `x` de `lst`⋆
`lst.pop()`	retire et renvoie le dernier élément de `lst`♣
`lst.pop(i)`	retire et renvoie l'élément d'indice `i` de `lst`♡
`lst.clear()`	vide la liste
`lst.sort()`	trie la liste♢
`lst.reverse()`	renverse la liste

⋆ : erreur si `lst` ne contient pas `x`
♣ : erreur si `lst` est vide
♡ : erreur si l'indice `i` n'existe pas dans `lst`
♢ : erreur si `lst` contient des éléments incomparables

Méthodes qui ne modifient pas la liste

méthode	effet
`lst.index(x)`	renvoie l'indice de la première occurrence de `x` dans `lst`⋆
`lst.count(x)`	renvoie le nombre d'occurrences de `x` dans `lst`⋆
`lst.copy()`	renvoie une copie (superficielle !) de `lst`♣

⋆ : erreur si `lst` ne contient pas `x`

♣ : *superficielle* veut dire que les éléments ne sont pas recopiés

Manipulation de sous-listes : les *slices* (tranches)

La connaissance de cette notion n'est pas exigible à l'examen.

Accès à une slice

- Syntaxe : `lst[i, j]` construit une liste contenant les éléments d'indices `i` à `j-1` de `lst`
- Attention, l'élément d'indice `i` est **inclus** mais celui d'indice `j` est **exclu** !

In [72]:

```
lst = [3, 'toto', 4.5]
print(lst[1:len(lst)])
print(lst[0:1])
```

```
['toto', 4.5]
[3]
```

- Si `i` est omis, il prend la valeur par défaut `0`
- Si `j` est omis, il prend la valeur par défaut `len(lst)`

In [73]:

```
lst = [3, 'toto', 4.5]
print(lst[:len(lst)-1])
print(lst[:])  # cette instruction crée une copie de lst !
```

```
[3, 'toto']
[3, 'toto', 4.5]
```

Affectation de slice

On peut aussi utiliser la syntaxe des tranches pour modifier en une seule fois une partie de la liste

In [74]:

```
lst = [3, 'toto', 4.5]
lst[1:3] = ["riri", "fifi", "loulou"]
print(lst)
```

```
[3, 'riri', 'fifi', 'loulou']
```

Tranches avec intervalle

Il est possible de compléter la notation des tranches en spécifiant un "pas" `k` : `lst[i:j:k]` . Dans ce cas, on sélectionne uniquement les éléments d'indices `i` , `i+k` , `i+2*k` , etc. en s'arrêtant à l'indice `j` (exclu).

In [75]:

```
lst = [0, 1, 2, 3, 4, 5]
print(lst[1:6:2])
```

```
[1, 3, 5]
```

Un exemple un peu étrange :

In [76]:

```
lst = [0, 1, 2, 3, 4, 5]
print(lst[6:1:-1])
```

```
[5, 4, 3, 2]
```

Indices négatifs

La connaissance de cette notion n'est pas exigible à l'examen.
Il est possible d'utiliser des indices négatifs pour accéder aux éléments d'une liste. Dans ce cas, les éléments sont numérotés à partir de la droite, en commençant par l'indice `-1` et jusqu'à l'indice `-len(lst)` :

In [77]:

```
lst = [3, 'toto', 4.5]
print(lst[-1])
print(lst[-3])
```

```
4.5
3
```

On voit que `lst[-1]` est une autre façon de désigner l'élément `lst[len(lst)-1]`, et `lst[-len(lst)]` désigne `lst[0]`. Tenter d'accéder à un indice plus petit provoque une erreur :

In [78]:

```
lst = [3, 'toto', 4.5]
print(lst[-4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[78], line 2
      1 lst = [3, 'toto', 4.5]
----> 2 print(lst[-4])

IndexError: list index out of range
```

Ces indices négatifs sont utiles comme raccourcis d'écriture dans certains cas particuliers, par exemple pour construire la copie d'une liste privée de son dernier élément :

In [79]:

```
lst = [3, 'toto', 4.5]
print(lst[:-1])
```

```
[3, 'toto']
```