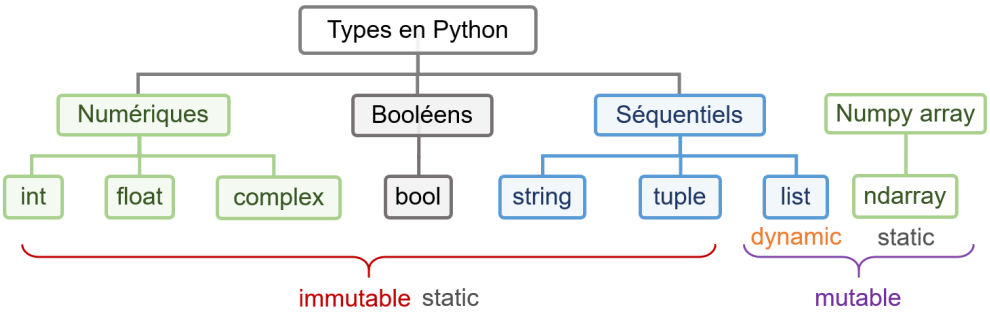
1 Tableaux et listes

De manière générale une *structure de données* spécifie la façon d'**organiser les données** d'un problème à résoudre et le format de stockage de ces données (= la manière dont la machine stocke l'information en RAM) afin de permettre un accès efficace en lecture et/ou en modification de ces données. Une structure de donnée est dite

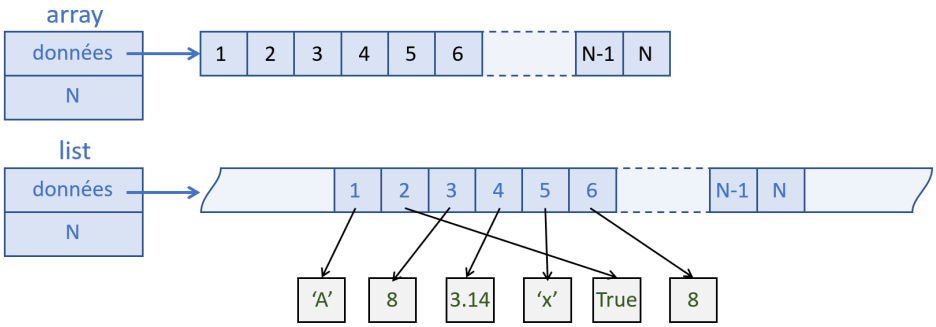
- **statique** lorsque la quantité de mémoire qui lui est allouée est fixée au moment de sa création et ne peut plus être modifiée ensuite,
- **dynamique** lorsqu'au contraire, la quantité de mémoire allouée à la structure de donnée est susceptible d'être modifiée (augmentée ou diminuée) au cours du déroulement de l'algorithme,
- **mutable** lorsque son contenu peut être modifié (= les données peuvent être lues et écrites),
- **non mutable** (immuable ou *immutable* en anglais) lorsque son contenu ne peut pas être modifié (= les données peuvent seulement être lues).

Parmi les différents types de données en Python, il faut distinguer les types de base (int, float, bool, complex) et les *conteneurs* (tuple, list ou ndarray) qui sont des données qui peuvent contenir d'autres données. Les `list` et les tableaux Numpy (`ndarray`) sont *mutable* alors que les autres types (int, float, bool, complex et tuple) ne le sont pas. La figure ci-dessous représente les principaux types Python que nous utiliserons :



1.1 Comment choisir entre une liste Python ou un tableau Numpy ?

Dans un tableau Numpy (`ndarray`), tous les éléments sont de **même type**. Dans une liste Python (`list`), ils peuvent être **de types différents**. Mais cela ne constitue pas la seule différence entre ces types d'objet. La manière précise dont Python arrange la mémoire pour stocker les données d'une liste est complexe et sort du cadre de ce cours. La figure ce-dessous illustre la manière dont la mémoire est organisée pour stocker les données d'un tableau et d'une liste [1].



Les éléments d'un tableau Numpy sont stockés successivement dans la mémoire. Chaque élément occupe le même nombre de cases de mémoire. Les éléments d'une liste sont indexés par l'intermédiaire d'un tableau de références. On peut ainsi échanger la place de deux éléments d'une liste très rapidement : il suffit d'inverser les adresses mémoires grâce à l'instruction `L[i], L[j] = L[j], L[i]`.

Statique *vs* dynamique ?

- la classe `list` est une structure de donnée dynamique. Cela signifie que sa taille peut être modifiée (augmentée par la méthode `.append()` ou diminuée par la méthode `.pop()` par exemple),

1. Les premiers termes de cette exemple de liste sont `['A',True, 8, 3.14, 'x', 8, ...]`.

- la classe `ndarray` est une structure de donnée statique. Cela signifie que la taille d'un tableau Numpy est fixée dès sa création et **ne peut plus est changée**.

Il sera donc plus contraignant d'utiliser un tableau Numpy (`ndarray`) qu'une liste car non seulement le nombre d'éléments doit être connu dès la création du tableau mais les éléments doivent également être du même type (int, float, bool,...). Par contre, le tableau s'avère plus performant en terme de vitesse d'exécution que la liste car les données sont rangées de manière beaucoup plus simple dans la mémoire RAM (cf fig. 1.1). De plus, le tableau Numpy permet une syntaxe nettement plus compacte pour les opérations mathématiques : l'instruction `t*=2` multiplie toutes les valeurs du `ndarray t` par 2 : cette opération **cache évidemment une boucle** sur les `N` éléments du tableau qui est effectuée par le processeur.

A retenir

- Pour stocker des éléments de manière séquentielle, possédant éventuellement des types différents, la classe `list` est particulièrement bien adaptée.
- Pour stocker un nombre fixe de valeurs numériques, la classe `ndarray` est à privilégier.

1.2 Modification d'objet *vs* création d'un nouvel objet

Considérons l'exemple suivant dans lequel l'instruction `id(var)` permet d'afficher l'adresse mémoire de l'objet désigné par la variable `var`.

```
#import numpy as np
M=np.array([[1,2,3],[4,5,6]]) # création d'une matrice de 2 lignes et 3 colonnes
print('AVANT     addr = ',id(M),'\n M = \n', M )
M[0,2]=0 # mise à zéro de l'élément de la ligne 1 et de la colonne 3
print('APRES (1) addr = ',id(M),'\n M = \n', M )
M=M[:,1:2] # sélection de la 2ème colonne
print('APRES (2) addr = ',id(M),'\n M = \n', M )
```

Le résultat affiché est

```
AVANT     addr = 2352859285424
M =
[[1 2 3]
[4 5 6]]
APRES (1) addr = 2352859285424
M =
[[1 2 0]
[4 5 6]]
APRES (2) addr = 2352845227712
M =
[[2]
[5]]
```

On constate que l'adresse de la variable `M` n'est pas modifiée par l'instruction `M[0,2]=0`. Elle est par contre modifiée par l'instruction `M=M[:,1:2]` bien que l'objet `M` soit de type *mutable* : cela signifie qu'un nouvel objet a été créé par cette instruction.

Attention, les instructions `L=L[1:5]` ou `t=t[1:5]` ne modifient pas la liste `L` ou le tableau `t` mais créent un nouvel objet construit à partir de l'ancien puis fait pointer la référence `L` ou `t` vers ce nouvel objet. L'ancien objet `list` est conservé dans la mémoire s'il possède encore une référence qui pointe sur lui (cf exemples ci-après).

1.3 Exemples de manipulation de listes en Python

Décortiquons quelques exemples afin de bien comprendre la manière dont les données sont organisées dans la mémoire et les règles quant à leur modifications.

Exercice 1 : prédire ce qui est affiché lors de l'exécution des scripts suivants.

Exemple 1 : copie de liste

```
1 L1=['A','B','C']
2 L2=L1
3 print(L1.pop())
4 print(L1)
5 print(L2)
6 L1+=['X', 'Y']
7 print(L1,L2)
8 L1=L1[1:4]
9 print(L1,L2)
```

Le programme affiche

```
C
['A', 'B']
['A', 'B']
['A', 'B', 'X', 'Y'] ['A', 'B', 'X', 'Y']
['B', 'X', 'Y'] ['A', 'B', 'X', 'Y']
```

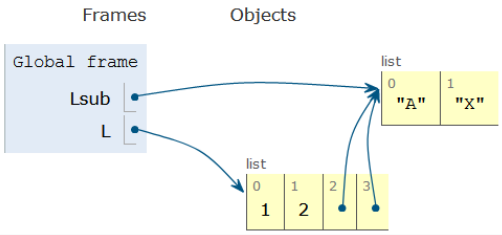
Exemple 2 : listes et sous-listes

```
1 Lsub=['A','B']
2 L=[1,2,Lsub,Lsub]
3 print(L)
4 Lsub.pop()
5 print(L)
6 L[2].append('X')
7 print(Lsub,L)
```

Le programme affiche

```
[1, 2, ['A', 'B'], ['A', 'B']]
[1, 2, ['A'], ['A']]
['A', 'X'] [1, 2, ['A', 'X'], ['A', 'X']]
```

A la fin de l'exécution, la structure de la mémoire est la suivante (cf PythonTutor) :



Exemple 3 : mauvaise manière d'initialiser à zéro une liste de listes

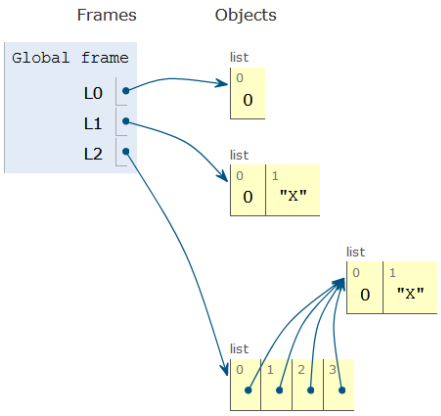
On souhaite créer une liste de listes pour l'utiliser comme un tableau 2D de n lignes et m colonnes et accéder ainsi aux éléments avec l'instruction `L[i][j]` avec `i` $= 0 \dots n-1$ et `j` $= 0 \dots m-1$.

```
1 L0=[0]
2 L1=[0]*2
3 L2=[[0]*2]*4 # liste de 4 sous-listes
4 L1[1]='X'
5 print(L1)
6 L2[2][1]='X' # 2ème élément de la 3ème sous
      liste
7 print(L2)
```

Le programme affiche

```
[0, 'X']
[[0, 'X'], [0, 'X'], [0, 'X'], [0, 'X']]
```

A la fin de l'exécution, la structure de la mémoire est la suivante :



Exemple 4 : manière correcte d'initialiser à zéro une liste de listes

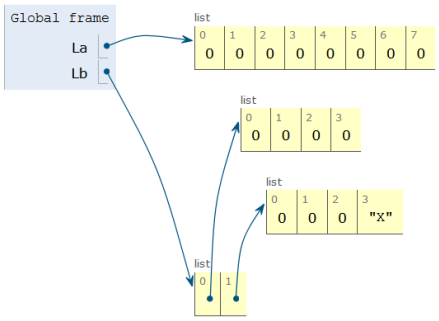
Observer l'imbrication des crochets `[` et `]` dans les listes en compréhension `[expr for.. in range(..)]`.

```
A=[0 for j in range(4) for i in range(2)]
B=[[0 for j in range(4)] for i in range(2)]
print(A)
B[1][3]='X' # 2ème ligne, 4ème colonnes
print(B) #  liste de 2 sous-listes
```

Le programme affiche

```
[0, 0, 0, 0, 0, 0, 0, 0]
[[0, 0, 0, 0], [0, 0, 0, 'X']]
```

A la fin de l'exécution, la structure de la mémoire est la suivante :



Exemple 5 : ajout d'éléments dans une liste *vs* concaténation de listes

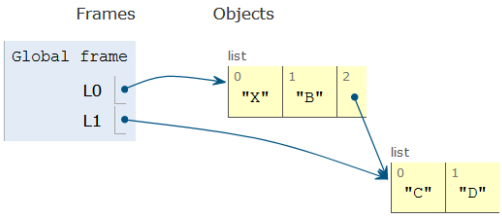
```
L0=['A','B']
L1=['C','D']
L0+=[L1] # équivalent à L0.append(L1)
L0[0]='X'
print(L0)
```

Le programme affiche

```
['X', 'B', ['C', 'D']]
```

Ici la liste L1 a été ajouté comme un 3ème élément de la liste L0.

A la fin de l'exécution, la structure de la mémoire est la suivante :



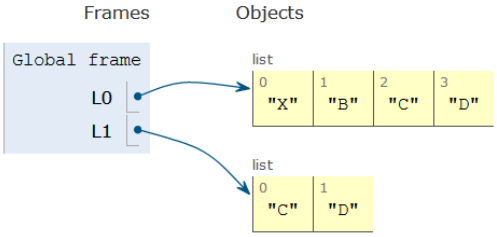

```
L0=['A','B']
L1=['C','D']
L0+=L1 # équivalent à L0.extend(L1)
L0[0]='X'
print(L0)
```

Le programme affiche

```
['X', 'B', 'C', 'D']
```

Dans ce cas, la liste L0 a été concaténée avec la liste L1.

A la fin de l'exécution, la structure de la mémoire est la suivante :



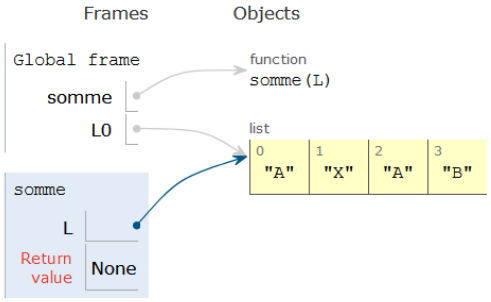
Exemple 6 : listes et fonction

```
def somme(L):
    L+=L0 # concaténation de la liste L
    L0[1]='X' # modification du second élément de la liste
L0=['A','B']
somme(L0)
print(L0)
```

Le programme affiche

```
['A', 'X', 'A', 'B']
```

Interprétation : lorsque le processeur entre dans la fonction `somme`, la liste L et la liste L0 font référence au même objet. L'instruction `L+=L0` étend (concatène) la liste L avec elle même. Les éléments de la liste L0 sont donc recopiés et ajoutés à la liste L (qui n'est autre que L0). Cette opération est parfaitement licite. Si on avait remplacé l'instruction `L+=L0` par `L+=[L0]`, on aurait ajouté au dernier élément de la liste une liste qui est L0, c'est-à-dire elle-même. Cela est également parfaitement licite en Python. Voici l'état de la mémoire juste avant de sortir de la fonction :



Cet exemple montre qu'en manipulant des listes dans des fonctions il est facile de parvenir à un programme dont la compréhension est très difficile : l'exemple proposé montre exactement ce qu'il ne faut pas faire si l'on souhaite que d'autres utilisateurs puissent suivre la logique du programme. Prendre garde également à bien distinguer l'opération de concaténation (=`.extend()`) de l'opération d'ajout d'élément (=`.append()`).

Exemple 7 : fonction, objets mutables (tableaux Numpy `ndarray`) et non mutables (`int`)

```
import numpy as np
T1=np.array([1,2,3,4])
T2=np.array([1,2,3,4])
def add1(T,x):
    T+=x
def add2(T,x):
    T=T+x
add1(T1,2)
add2(T2,3)
print(T1,T2)
```

Le programme affiche

```
[3 4 5 6] [1 2 3 4]
```

Interprétation : lorsque la fonction `add1` s'exécute, la variable `T` fait référence au même objet que `T1` (= un tableau Numpy). S'agissant d'un objet *mutable*, l'instruction `T+=x` modifie ce tableau Numpy `T` ajoutant `x` à chacun de ses termes. L'objet n'étant pas recréé, les valeurs modifiées demeurent accessibles par la variable `T1` du script d'appel au sortir de la fonction.

Lorsque la fonction `add2` s'exécute, les variables `T` et `T2` font toujours référence toutes les deux au même objet. En revanche, l'instruction `T=T+x` ne modifie pas l'objet référencé par `T` mais

- crée un nouvel objet en dupliquant `T`
- ajoute `x` à chaque de ses éléments (fonction addition du tableau Numpy)
- puis fait référencer la variable `T` vers ce nouvel objet.

Ainsi, ce nouvel objet qui convient la somme des termes, est référencé par `T`, mais désigne un objet différent de celui qui est référencé par `T2`. Lorsque l'on quitte la fonction `add2`, la référence `T` est détruite, la somme est donc définitivement perdue : la fonction `add2` n'a servi à rien !

```python
def add1(T,x):
    T+=x
a=5
add1(a,2)
print(a)
```

Le programme affiche

```
5
```

Interprétation : cette fois les variables `a` et `T` font référence à un objet `int` qui est *non mutable*. Ainsi, l'instruction `T+=x` **ne peut pas modifier** l'objet T : un nouvel objet est donc créé tout comme si on avait écrit `T=T+x`. Cet objet est donc perdu lorsque l'on quitte la fonction car la variable `a` fait référence à un objet différent.

Les « gros » objets (listes, tableaux) sont généralement *mutable* de manière à éviter au maximum qu'on les recopie dans leur intégralité lorsque l'on effectue des opérations élémentaires sur ces objets.

2 Les piles LIFO

2.1 Notion de pile

Définition 2.1 : Pile LIFO *Last In First Out*

Une *pile LIFO* est un conteneur dans lequel on peut uniquement :

- savoir s'il est vide,
- ajouter un élément (**empiler**), opération appelée *push*,
- consulter le dernier élément ajouté (on dit qu'il est placé « en haut de la pile »),
- retirer ce dernier élément (**dépiler**), opération appelée *pull*.

On parle de pile *Last In First Out* car le dernier élément déposé est le premier qui peut être retiré [a].

a. Il existe également les piles FIFO *First In First Out* mais ne sont pas au programme.

Un objet de type *pile* possède donc

- **deux méthodes de lecture** pour :
  - renvoyer un booléen pour savoir si la pile est vide,
  - renvoyer la valeur du dernier élément = celui du *haut de la pile*,
- et **deux méthodes d'écriture**, c'est à dire modifiant l'objet *pile* pour :
  - enlever le dernier élément = celui du *haut de la pile*,
  - ajouter un élément au *sommet de la pile*.

On peut évidemment faire l'analogie avec une pile d'assiettes (ou de livres, cf fig. 1) : on ne voit que l'assiette du dessus, on ne peut ajouter une nouvelle assiette que par le dessus et on ne peut que retirer l'assiette du dessus en premier. Si on a besoin d'accéder aux autres assiettes, il faut commencer par retirer celles qui sont au-dessus.

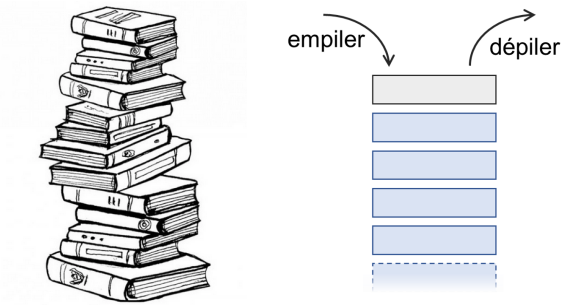


FIGURE 1 – Illustration de la notion de pile LIFO.

La notion de pile est fondamentale en informatique et possède de nombreuses applications, par exemple la fonction « annuler la dernière action » dans les logiciels d'édition (chaque nouvelle action étant empilée, elle peut être dépilée pour annuler la dernière action). Nous verrons également l'intérêt des piles pour les fonctions récursives (chapitre suivant). En terme de complexité temporelle, chacune des 4 opérations sur une pile est considérée comme ayant un coût unitaire.

2.2 Implémentation des piles en Python

En python, pour implémenter les piles, nous utiliserons un objet de type *list* en **nous limitant aux 5 cinq instructions suivantes** :

- `pile1 = []` ou `pile1 = list()` pour créer une pile (*i.e.* une liste vide),
- `len(pile1) == 0` pour savoir si la pile `pile1` est vide,
- `pile1.append(x)` pour empiler un nouvel élément sur la pile `pile1`,
- `pile1.pop()` pour enlever et retourner l'élément sur *le haut de la pile* `pile1`,
- `pile1[-1]` pour renvoyer la valeur de l'élément sur le haut de la pile `pile1`.

Note : L'instruction `pile1.pop()` permet de dépiler le dernier élément. On peut également utiliser `x=pile1.pop()` pour dépiler le dernier élément et stocker sa valeur dans `x`.

Exercice 2 : Permuter les deux éléments au sommet de la pile

Écrire une fonction Python `permuter(p)` qui permute les deux éléments du *haut* de la pile `p`, ces éléments étant supposé exister.

Solution : comme pour une pile d'assiettes, il est nécessaire de retirer d'abord celle du haut de la pile, de la déposer quelque part. On prend ensuite la seconde assiette que l'on met également de côté, puis on les remet dans le bon ordre.

```
def permuter(p):
        x = p.pop()  #  dépiler et stocker le dernier élément
        y = p.pop()  # dépiler et stocker le suivant
        p.append(x) # empiler celui qui était le dernier
        p.append(y) # empiler celui qui était en avant-dernière position
```

Pourquoi n'y a-t-il pas d'instruction `return` ?

2.3 Exemples d'application des piles

Nous allons traiter deux exemples d'application des piles.

2.3.1 Expression bien parenthésée

Soit une chaîne de caractères constituée de parenthèses ouvrantes `(` et fermantes `)`. Par exemple : `'(3+(2+a*(1-x)+(2*Log(x))/(1+x)))'`.

On souhaite construire un vérificateur de parenthésage en utilisant un objet *pile*. Compléter le squelette de la fonction `bienParenthese(s)` donné ci-dessous, qui prend en argument une chaîne de caractères `s` et la parcourt de gauche à droite de manière à vérifier qu'elle est bien parenthésée. La fonction renvoie True s'il n'y a pas d'erreurs dans le parenthésage de cette expression et False sinon. On rappelle qu'en Python, les éléments d'une chaîne de caractères `s` peut être parcourus à l'aide d'une boucle `for` par l'instruction `for car in s :`.

```
# Parenthésage
def bienParenthese(s):
    pile=[]
    for car in (s):
        if car=='(': # ouvrant
            # à compléter

        if car==')': # fermant
            # à compléter

    return # à compléter
```

Solution : l'idée est d'empiler les parenthèses ouvrantes à chaque fois que l'on en rencontre une. Lorsque l'on rencontre une parenthèse fermante, on vérifie qu'une parenthèse ouvrante lui correspond dans la pile puis

on la dépile. A la fin du parcours de la chaîne, on vérifie que la pile est bien vidée, c'est-à-dire qu'il n'y a pas d'ouvrantes en trop.

```python
def bienParenthese(s):
    pile=[]
    for car in (s):
        if car=='(': # ouvrant
            pile.append(car) # on empile la parenthèse ouvrante
        if car==')': # fermant
            if pile==[]: # pile vide
                return False # Erreur
            pile.pop()
    return len(pile)==0 # La pile doit être vidée sinon Erreur
```

Exercice 3 : Parenthésage avec « ( ) », « { } » et « [ ] »

Modifier la fonction précédente de manière à inclure également les symboles crochets [ ] et accolades { }. A chaque symbole « ouvrant » doit correspondant le bon symbole « fermant » et les symboles ne doivent pas se croiser.

Solution : tout comme précédemment, on empile les symboles « ouvrants » que l'on rencontre. Mais cette fois, lorsque l'on rencontre un symbole « fermant », il ne suffit plus de vérifier que la pile n'est pas vide avant de dépiler. Il faut également vérifier que l'élément placé sur le *haut de la pile* est bien l'« ouvrant » correspondant à ce « fermant ».

```python
def bienParenthese2(s):
    pile=[]
    for car in (s):
        if car=='(' or car =='{' or car =='[': # ouvrant
            pile.append(car) # on empile l'ouvrant

        if car==')':
            if (len(pile)==0) or pile[-1]!='(':  # ()?
                return False
            pile.pop()
        if car==']':
            if (len(pile)==0) or pile[-1]!='[':  # []?
                return False
            pile.pop()
        if car=='}':
            if (len(pile)==0) or pile[-1]!='{': # {}?
                return False
            pile.pop()
    return len(pile)==0 # la pile doit être vidée sinon Erreur
```

Exercice 4 : Parenthésage, généralisation*

On considère cette fois que les symboles ouvrants sont données dans une liste Python de caractères `ouvrants = ['(','{','[','<', ... ]` et que les fermants correspondants sont donnés dans la liste `fermants = [')','}',']','>', ... ]` ordonnée dans le même ordre, c'est-à-dire que le k-ième élément de la liste des ouvrants correspond au k-ième élément de la liste des fermants. Définir une fonction `bienParenthese3(s,ouvrants,fermants)` qui vérifie le parenthésage de la chaîne `s` pour les symboles `ouvrants` et `fermants` fournis en argument. [a]

a. Indication : pour tester qu'un caractère `c` appartienne à une liste `L` on peut utiliser le test
`if c in L :`
On peut également parcourir la liste `L` avec une boucle `for` utilisant un indice numérique :
`for k in range(len(L)):`
`if c == L[k]:`
On obtient en plus l'indice `k` de l'élément.

Solution : on teste la correspondance entre le l'ouvrant et le fermant en récupérant l'indice k du fermant. L'algorithme est le suivant.

- Tout caractère qui est un ouvrant est empilé.
- Dès qu'un caractère fermant est trouvé, on vérifie que la liste ne soit pas vide (sinon on renvoit Faux).
- puis un vérifie que le caractère placé en haut de la pile est bien l'ouvrant correspond au fermant en question. Dans ce cas on dépile l'ouvrant. Sinon on renvoit Faux.

- Enfin, lorsque la boucle est entièrement parcourue, on vérifie que la pile est bien vide.

Le code est le suivant :

```
# Parenthésage
def bienParenthese3(s,ouvrants,fermants):
    pile=[]
    for car in (s):
        if car in ouvrants : # ouvrant
            pile.append(car) # on empile l'ouvrant
        for k in range(len(fermants)) : # recherche du fermant
            if car == fermants[k] : # car est le k-ième fermant
                if len(pile) == 0 : # liste vide ?
                    return False  # alors erreur
                if pile[-1] == ouvrants[k]: # le haut de pile correspond avec le fermant ?
                    pile.pop() # alors on dépile
                else:
                    return False # sinon erreur
    return len(pile) == 0 # la pile doit être vidée sinon Erreur
```

Le tester avec les instructions suivantes :

```
exp='AB12bCDDAaddcba'
o=['A','B','C','D']
f=['a','b','c','d']
bienParenthese3(exp,o,f)
```

2.3.2 Evaluation d'une expression en notation polonaise inversée

On souhaite réaliser une fonction qui évalue une expression arithmétique écrite en notation polonaise inversée (NPI). La *notation polonaise inversée*, également appelée *notation post-fixée*, permet d'écrire de façon non ambiguë des formules arithmétiques sans utiliser de parenthèses.

Au lieu d'écrire les opérandes autour de l'opérateur (comme $4+5$), on écrit d'abord les opérandes, puis l'opérateur (`[ 4 , 5 , "+" ]`).

Par exemple, l'expression $3 \times (4+7)$ peut s'écrire :
`[ 3 , 4 , 7 , "+" , "*" ]`
ou `[ 4 , 7 , "+" , 3 , "*" ]`
De même l'expression

$$A = ((23-4) \times 4 + 11/3$$

peut s'écrire :
`[ 24 , 4 , "-" , 4 , "*" , 11 , 3 , "/" , "+" ]`

Exercice 5 : Évaluation d'une expression arithmétique écrite en notation polonaise inversée

En utilisant une pile, écrire une fonction `evalueNPI(expr)` qui permet d'évaluer une expression en notation polonaise inversée tout en parcourant l'expression fournie en entrée. On utilisera pour l'entrée une liste d'éléments étant :

- ou bien des nombres (entiers ou flottants) pour les opérandes,
- bien des caractères "+", "-", "*", "/" pour les opérateurs [a].

L'expression `expr` fournie en entrée sera supposée correcte et on ne demande pas au programme de le vérifier.

a. L'idée est de distinguer le cas où l'élément de la liste parcouru est un nombre (= un opérande) ou un opérateur binaire (= qui agit sur deux opérandes).

Solution : on parcourt linéairement la liste entrée à l'aide d'une boucle `for`. Chaque opérande rencontré est empilé. Dans le cas où l'élément rencontré est un opérateur, on va chercher les deux derniers opérandes, on effectue l'opération demandée et on empile le résultat. En fin de parcourt, on vide la pile et renvoie le dernier élément qui constitue l'évaluation de l'expression.

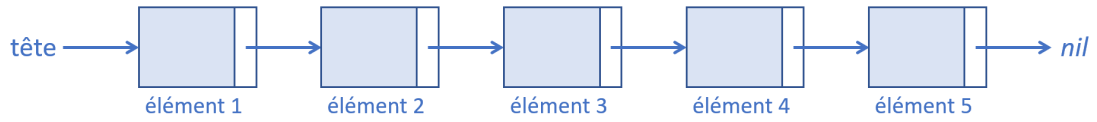
```
def evaluateNPI(expr):
    pile=[]
    Operateurs=["+","-","*","/"]
    for el in expr:
        # el est un opérateur
        if el in Operateurs:
            b=pile.pop() # operande 2
            a=pile.pop() # operande 1
```

```
            if el=="+":
                pile.append(a+b)
            elif el=="-":
                pile.append(a-b)
            elif el=="*":
                pile.append(a*b)
            elif el=="/":
                pile.append(a/b)
        else:
            pile.append(el)
    return pile.pop()
```

3 Performances des structures de données linéaires

3.1 Tableau, liste Python et liste chaînée

Hormis la structure de pile, il existe en informatique de nombreuses autres structures de données (dont la présentation exhaustive est hors-programme). Nous avons déjà évoqué le tableau et la liste Python (objet `list`). Mentionnons la liste chaînée qui permet de ranger des éléments dont le nombre est susceptible de varier. Ci-dessous est donné le schéma d'une liste chaînée : chaque élément donne l'accès au suivant.



Une **liste chaînée** est une suite ordonnée d'éléments $(a_0, a_1, \dots a_{n-1})$. Le premier élément a_0 est la tête de liste. L'accès aux éléments se fait en parcourant les éléments de manière séquentielle : pour lire la valeur de a_k, on doit accéder à a_0, puis à a_1, ... , puis à a_{k-1} et enfin à a_k. Chaque élément contient une référence qui permet d'accéder à l'élément suivant (c'est l'« adresse » de l'élément suivant).

3.2 Comparatif de performances

Les performances des structures de données sont quantifiées par le coût de certaines opérations courantes :

- rechercher si x est un élément de la structure de données (test d'appartenance),
- renvoyer le maximum (en supposant l'existence d'une relation d'ordre totale sur les éléments),
- insérer un élément à une position quelconque,
- accéder à un élément en position quelconque,
- insérer, accéder à un élément en position finale (ou en tête de liste chaînée),

Les différentes structures de données ne sont pas équivalentes en terme d'efficacité pour ces opérations. Par exemple, l'accès à un élément quelconque d'un **tableau** se fait à coût constant (en $\mathcal{O}(1)$). En revanche, pour accéder au k-ième élément d'une **liste chaînée** de n éléments, il faut parcourir les k premiers. Ainsi, l'accès à un élément quelconque requiert en moyenne un coût temporel de l'ordre du nombre d'éléments, soit en $\mathcal{O}(n)$. L'insertion d'une élément dans un tableau à une place arbitraire est mal aisée car on est contraint de décaler tous les éléments suivants d'une case. En revanche, l'ajout d'un élément dans une liste chaînée se fait à coût constant. La **liste Python** `list` (cf fig. 1.1) combine certains avantages de la liste chaînée et du tableau. Avec une structure de **pile**, il est possible de lire la valeur d'un élément quelconque, insérer ou supprimer un élément située à une position quelconque, mais cela requiert de dépiler tous les éléments qui sont placés au dessus (puis de les rempiler éventuellement). Le coût temporel de ces opérations est donc en $\mathcal{O}(n)$.

opérations	tableau	liste chaînée	liste Python	pile
accès arbitraire	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
ajout, suppression	non prévu	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
ajout en tête (ou en queue)	non prévu	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
test d'appartenance	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
renvoyer le max (le min)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

En utilisant des structures de données *non linéaires* (arbre, graphe), il est possible d'optimiser les opérations de recherche de maximum, de test d'appartenance, d'insertion, etc... Cela est hors-programme.