**Remarque préliminaire** : Jupyter est parfois capricieux pour le téléchargement des images. Si les images n'apparaissent pas dans le notebook, chargez les dans le même dossier que le notebook. Les adresses se trouvent en double cliquant dans les cellules de texte (là où il y a précisé "image", c'est qu'il y a une image normalement...). Puis changez le code comme ceci : `![Image : listes](http://www.maths-info-lycee.fr/images/arbre1.jpg)` devient `![Image : listes](imagearbre1.jpg)` ou même `![](imagearbre1.jpg)`

# Structures de données linéaires

Les structures de données linéaires sont des suites d'éléments $e_1, e_2, \dots, e_n$. Dans une structure linéaire, on traite les données séquentiellement, c'est-à-dire les unes après les autres. De plus on doit pouvoir ajouter et supprimer des éléments.
On va s'intéresser à trois types de structures linéaires : les listes, les piles et les files.

**Compléter ce cours/TD** au fur et à mesure que vous le faites. Pour écrire dans une cellule, double-cliquez dedans. Une fois le TD fait, **imprimez**-le : pour réviser, la mémorisation se fait mieux avec un cours papier que sur écran.

## Les listes

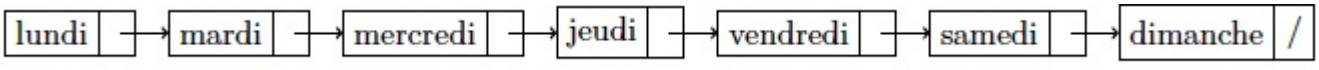
Une première *remarque* fondamentale : on ne parle pas du type `list` vu en Python en première. Le type `list` de Python est en fait un tableau dynamique.

Une liste en informatique est une suite d'éléments. Cette suite est finie, et peut être vide. Chaque élément de la suite est repérée par son indice : la liste est ordonnée par l'indice (et non par la valeur de l'élément).

Deuxième *remarque* : une liste informatique n'est pas non plus ce qu'on appelle une liste dans le langage courant. Quand on a une liste de courses, on ne suit pas l'ordre de la liste pour faire ses courses. Et on ne met pas deux fois le même ingrédient (sauf étourderie). Dans un style plus littéraire, vous pouvez lire les notes de chevet (枕草子, Makura no sōshi) de Sei Shōnagon, écrites vers 990. Ce sont des listes poétiques : "Choses dont on néglige souvent la fin", "Choses que l'on méprise", "Choses qui font battre le cœur", "Fleurs des arbres", "Cascades"...

*Exemple* à compléter : On donne la liste `L1` des jours de la semaine :

```
L1 = [lundi , mardi, mercredi , jeudi , vendredi , samedi , dimanche]
```




`L1` a pour tête ... *à compléter* ... et est suivie de la liste `L2 =` ... *à compléter*...
`L2` a pour tête ... *à compléter* ... et est suivie de la liste `L3 =` ... *à compléter* ...
Donner la dernière étape de cette décomposition:

*Appeler éventuellement le professeur pour vérification*

Une défintion simple du type `Liste` utilise les primitives suivantes (ce sont entre autre les méthodes de la classe, mais pas forcément) :

- construction d'une liste vide : `creerListe()`
- test de vacuité d'une liste : `estVide(liste)`
- Ajouter un élément en tête de la liste : `cons(élément, liste)` . C'est en fait le constructeur "historique" du type `liste` .
- Renvoyer le premier élément de la liste sans le supprimer : `donnee(liste)` . Renvoie la "tête" de liste
- Renvoyer la liste suivante, éventuellement vide, obtenue à partir de la liste initiale en supprimant son premier élément. : `suivant(liste)` . Renvoie la "queue" de la liste.

Une liste `L` peut s'écrire `L = cons(donnee(L) , suivant(L))` . On remarque que cette définition est récursive ; `suivant(L)` pouvant être une liste.

**Exercices (sur papier ou à compléter dans la cellule) :**

On utilisera uniquement les fonctions primitives définies ci-dessus

1. Créer la liste de vos quatre films préférés. les films doivent être dans l'ordre de vos préférences. Essayez d'écrire une seule ligne.
2. On donne une liste L1. Ecrire un algorithme en langage naturel renvoyant la liste L2, qui est dans l'ordre inverse de L1.

## Une première implémentation

En terminale NSI, on travaillera essentiellement sur les listes chaînées. Les listes chainées sont composées de maillons, et sont définies récursivement. Un maillon est composé d'un élément de tête, et... d'un autre maillon, qui contient les éléments suivants, éléments de queue. En fin de liste le maillon suivant est `None` . Les maillons sont aussi appelés cellules, notamment en anglais ( `cell` ).

On utilisera les conventions suivantes (qui ne sont pas universelles, d'autres versions peuvent exister) :

- la liste vide est le maillon de tête `None` et de queue `None` ;
- une liste de longueur 1 est composée d'un unique maillon de tête différente de `None` , et de queue `None` ;
- il est impossible d'avoir un maillon de tête `None` et de queue différente de `None` . Une implémentation est donnée ci-dessous, basée sur ces cellules (tête , queue). L'unique attribut de cette classe est :
- `cellule` : la liste proprement dite Les attributs de la classe `Cell` sont :
- `tete` : l'élément de tête de la liste (éventuellement `None` )
- `queue` : la liste composant la deuxième partie de `Cell` (éventuellement `None` )

Compléter le code en rajoutant les primitives `longueurListe` qui, comme son nom l'indique, renvoie la longueur de la liste ; et `listeElements` , qui, comme son nom l'indique moins clairement, renvoie un tableau dynamique Python (type `list` ) comportant les éléments de la liste, dans l'ordre où la tête de la liste a pour indice 0 dans le tableau dynamique.

In [ ]:
```
class Cellule :
    def __init__(self, tete = None, queue = None) :
        # Admirez le  joli booléen dans l'assertion
        assert (queue is None) or (tete is not None), 'construction
impossible'
        # On peut aussi rajouter une assertion pour vérifier que qu
eue
        # est soit un maillon, soit None
        self.tete = tete
        self.queue = queue

    def est_vide(self):
        return self.tete is None # and self.queue is None est inuti
le vu le constructeur

    def donnee(self):
        # Méthode classique, mais qui n'apporte rien de plus que se
lf.tete !
        assert not(self.est_vide()) , 'Listevide'
        return self.tete

    def suivant(self):
        # Méthode classique, mais qui n'apporte rien de plus que se
lf.queue !
        assert not(self.est_vide()) , 'Listevide'
        return self.queue

    def longueur_liste_rec(self):
        # Algortihme à coder :
        # si la liste est vide on renvoie 0, sinon on renvoie 1 + l
a longueur de la queue

        return

    def longueur_liste_iter(self):
        # Algorithme :
        # Après initialisation de la longueur à 0, tant qu'il reste
des élements dans self, on
        # ajoute 1 et on remplace self par sa queue
        long = 0
        return long

    def liste_elements_rec(self) :
        # On reprend l'algorithme récursif du calcul de la longueur
:
        # en cas de liste vide on renvoie [], sinon on renvoie la l
iste composée de la
        # tête à laquelle on ajoute la liste des éléments de la que
ue
        # Remarque : les listes s'aditionnent [1] + [2, 3] = [1, 2,
3]
        # Remarque : on peut aussi programmer avec la méthode appen
d
        return
```

```
    def liste_elements_iter(self) :
        # Algorithme : on se base sur le calcul de la longueur en r
écursif
        return

    def appartient_iter(self, element):
        return

    def appartient_rec(self, element) :
        return

    def suppression_elt_iter(self, element) :
        if not self.appartient_iter(element) :
            # On écrit une fonction préliminaire qui teste si l'élé
ment est dans la liste
            return False
        else :
            # Il faut garder en mémoire le liens  de la cellule pré
cédente à la cellule courante
            # pour pouvoir "couper et recoller" ce lien vers la cel
lule suivante
            # On passe de :
            #               précédente -> courante -> suivante
            # à :
            #               précédente -> suivante
            return True

    def suppression_elt_rec(self, element) :
    # On écrit une fonction préliminaire qui teste si l'élément est
dans la liste
        if not self.appartient_rec(self, element) :
            return False
        else :
            self.suppression_elt_rec_2(self, element)
            return True

    def suppression_elt_rec_2(self, element) :
        # partie récursive proprement dite
        # Reprendre la méthode itérative
        return None

    def __repr__(self):
        if self.tete is None :
            return '()'
        else:
            # la 1ère possibilité met l'aspect récursif en avant
            # la 2ème possibilité mes l'aspect chaîné en avant
            return '(' + str(self.donnee()) + repr(self.suivant()).
replace('None','()') + ')'
            # return  str(self.donnee) + '->' + repr(self.suivant)

def cons(tete, queue) :
    # Ici la primitive n'est pas une méthode mais une fonction. Don
t on
    # peut constater la totale inutilité, puisqu'elle tient sur une
ligne...
```

```
    return Cellule(tete, queue)



nil = Cellule(None, None)       # notation "nil" historique
print("liste nil : ",print(nil)," de longueur : ",nil.longueur_list
e_iter(),". Test pour vide :",nil.est_vide())

print("la liste nil a pour tête ", nil.tete , " et pour queue ",ni
l.queue)

liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)
print("liste :" , liste," de tête ",liste.donnee()," et de queue ",
liste.suivant())
# Donner l'instruction pour trouver 2, l'écrire à la place de None
print("Pour trouver 2, on a utilisé l'instruction ... " , None)
print("la longueur de la liste est : ",liste.longueur_liste_rec())
"""
print("Conversion en tableau dynamique (itératif) :", liste.liste_e
lements_iter())
print("Conversion en tableau dynamique :(récursif)", liste.liste_el
ements_iter())
print ("3 est dans ",liste," : ",liste.appartient_iter(3), liste.ap
partient_rec(3))
print ("7 est dans ",liste," : ",liste.appartient_iter(7), liste.ap
partient_rec(7))
"""

"""
# Suppression d'un élément : penser à tester les cas "limite"
liste.supprimer_elt_rec(3)
liste.supprimer_elt_rec(4)
liste.supprimer_elt_rec(0)
print(liste)

liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)

liste.supprimer_elt_iter(3)
liste.supprimer_elt_iter(4)
liste.supprimer_elt_iter(0)
print(liste)
"""

"""
# Egalité de listes
liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)
print(listeb , "est égale à ",liste," : ", liste.est_egale(listeb))
listeb = cons(4,listeb)
print(listeb , "est égale à ",liste," : ", liste.est_egale(listeb))
"""
```

```
        print()
```

**Exercices**

1. Donner la complexité dans le pire des cas des méthodes `est_vide()`, `longueur()`, `liste_elements()`, aussi bien pour les méthodes récursives qu'itératives.
2. Ecrire une méthode `appartient(element)` qui renvoie `None` si l'élément n'appartient pas à la liste, et son indice sinon. Complément pour ceux qui vont vite : en déduire une méthode `suppr_element(element)` qui supprime la première occurence d'un élément donné dans une liste.

*Questions complémentaires éventuelles*

1. Ecrire une méthode `est_egale(liste2)` qui teste si la liste2 est égale à la liste appelant la méthode.
2. Ecrire une méthode qui inverse la liste. En version un peu plus facile, on peut se contenter de renvoyer la liste inversée, plutôt que de modifier l'objet.
3. Ecrire une méthode `derniere(liste)` qui renvoie la dernière cellule de la liste. En déduire une méthode `concatener(liste2)` qui concatène la liste2 en fin de la liste appelant la méthode.

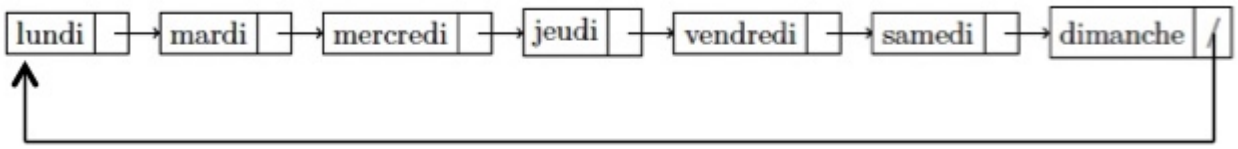
## Autres versions des listes

On peut créer d'autres versions des listes:

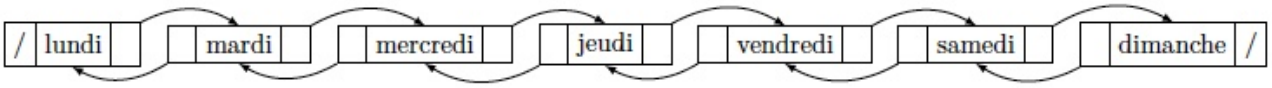
- Listes basées sur des tableaux. On perd l'intérêt des listes, qui est d'insérer facilement un élément

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| lundi | mardi | mercredi | jeudi | vendredi | samedi | dimanche |

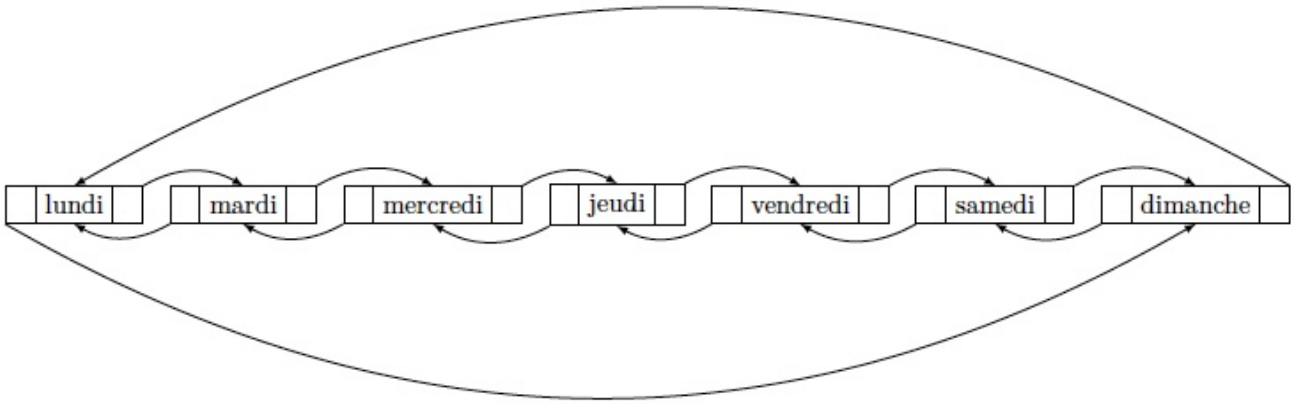
- Listes circulaires. Permet de boucler en fin de liste sur le premier élément




- Listes doublement chaînées. Permet de connaître non seulement l'élément suivant dans la liste, mais aussi le précédent




- Listes doublement chaînées circulaires



*Rappel sur une remarque importante* : le type abstrait `Liste` n'est pas le type `list` de Python. Les listes de Python sont basées sur des tableaux, et mélangent des accès de type fonctions ( `del(ma_liste[3]` ), des accès de type objet ( `ma_liste.append('truc')` ), et des accès plus étranges ( `machin in ma_liste` , `ma_liste[3:6]` )

### Une deuxième implémentation

Quand on passe de l'itératif au récursif pour l'implémentation des méthodes `longueur_liste()` et `liste_elements()` , cela change-t-il l'usage de la classe ?

Réponse

# Types abstraits

Comme vous venez de le voir juste ci-dessus, la manière dont est programmée la classe n'influe pas sur son usage. Plus précisément, l'**implémentation** de la classe ne joue pas sur sa **signature**. Le type de données `Liste` peut être définit de manière **abstraite**. Par exemple, pour utiliser des flottants en Python, vous n'av(i)ez pas besoin de connaître la représentation sous la forme mantisse-exposant, forme que l'on a vue dans le cours sur le codage.
On définit un type abstrait par sa **signature** : nom des opérations, type des arguments, type du retour des opérations.
Autant l'implémentation d'un type abstrait ne joue pas sur sa signature, par définition même, autant elle peut jouer sur la complexité des opérations du type.

# Des primitives différentes pour le type liste

Le type `Liste` n'est pas fixé dans le marbre. On peut proposer des primitives plus nombreuses et plus riches.

On propose ici les fonctions primitives sur les listes suivantes :

- construction d'une liste. La liste peut être vide, ou bien on peut la construire à partir d'un élément de tête et d'une autre liste. On appelle cette fonction : `creerListe(e = Aucun , liste = Aucun)`
- test de vacuité d'une liste : `est_vide(liste)`
- Obtention de la longueur de la liste : `longueur(liste)`
- Accéder au k-*ième* élément de la liste : `lire(liste , k)`
- Supprimer le k-*ième* élément de la liste : `supprimer(liste , k)` . Cette méthode renvoie une nouvelle liste.
- Insérer un élément en k-*ième* position dans la liste : `inserer(liste , k)` . Cette méthode renvoie une nouvelle liste.
- _Les trois primitives précédentes seront implémentées sur une méthode `get_maillon(liste , k)` _ Cette méthode est donnée ci-dessous en itératif. C'est un bon exercice que de la programmer également en récursif, et de voir que le fonctionnement des primitives n'en change pas pour autant

Exercice : programmer les méthodes `lire` et `insérer` , dans l'implémentation suivante du type `liste` .
Et pour ceux qui vont vite : programmer `supprimer` ,

In [ ]:
```
class Cellule :
    def __init__(self, tete = None, queue = None) :
        # Admirez le  joli booléen dans l'assertion
        assert (queue is None) or (tete is not None), 'construction
impossible'
        # On peut aussi rajouter une assertion pour vérifier que qu
eue
        # est soit un maillon, soit None
        self.tete = tete
        self.queue = queue

    def estVide(self):
        return self.tete is None # and self.queue is None est inuti
le vu le constructeur

    def donnee(self):
        assert not(self.estVide()) , 'Listevide'
        return self.tete

    def suivant(self):
        assert not(self.estVide()) , 'Listevide'
        return self.queue


    def longueur_liste(self):
        long = 0
        while not self.estVide():
            long = long + 1
            self = self.suivant()
        return long


    def get_maillon(self, i):
        # Version itérative
        if i >= self.longueur_liste() :
            raise IndexError('Index trop grand')
        else :
            while i > 0:
                i = i - 1
                self = self.suivant()
            return self

    def get_maillon_rec(self, i):

            return

    def lire(self , i) :
            return

    def inserer(self , i, element) :
        return

    def supprimer(self , i) :
        return
```

```
    def __repr__(self):
        if self.tete is None :
            return '()'
        else:
            # la 1ère possibilité met l'aspect récursif en avant
            # la 2ème possibilité mes l'aspect chaiîné en avant
            return '(' + str(self.donnee()) + repr(self.suivant()).
replace('None','()') + ')'
            # return  str(self.donnee) + '->' + repr(self.suivant)

maliste = Cellule()
print(maliste, maliste.estVide(), maliste.longueur_liste())
for i in range(5 , -1 , -1) :
    maliste = Cellule("film"+str(i),maliste)
    print("affichage : ",maliste, "de longueur ",maliste.longueur_l
iste())

# TESTS nombreux et multiples ! Il en manque d'importants d'ailleur
s, à compléter...
print("lecture des éléments d'indices 1 et 5 :",maliste.lire(1),mal
iste.lire(5))
print()
print("Insertion et suppression en itératif")
i = 5
print("get maillon d'indice ",i,)
print(maliste.get_maillon(i))
maliste = maliste.inserer(i , "film3b")
print("affichage insertion : ",maliste, "de longueur ",maliste.long
ueur_liste())
maliste = maliste.supprimer(i)
print("affichage suppression : ",maliste, "de longueur ",maliste.lo
ngueur_liste())
maliste = maliste.supprimer(0)
print("affichage suppression indice 0: ",maliste, "de longueur ",ma
liste.longueur_liste())
maliste = maliste.supprimer(maliste.longueurListe() - 1)
print("affichage suppression dernier indice : ",maliste, "de longue
ur ",maliste.longueur_liste())
liste_quasi_vide = Cellule("rien", Cellule())
print(liste_quasi_vide, "longueur : ", liste_quasi_vide.longueur_li
ste())
liste_quasi_vide = liste_quasi_vide.supprimer(0)
print("affichage suppression indice 0: ",liste_quasi_vide, "de long
ueur ",liste_quasi_vide.longueur_liste())
```

## Une troisième implémentation pour le type Liste

On donne ci-dessous une implémentation à base de tableaux dynamiques (le fameux type `list` de Python). La tête de la liste sera l'élément d'indice 0, la queue toute la suite.
Comme vous pouvez le constater ci-dessous, on ne construit pas de classe : les primitives sont traduites en fonctions.