TNSI	Représentation des données Listes, piles, files	Cours – **Listes, piles et files**

Objectifs :

- ⇨ Découvrir les concepts de listes chaînées, piles et files
- ⇨ Mettre en œuvre des listes, piles et files en python
- ⇨ Connaître des cas d'utilisation de ces structures de données

I – Structures de données linéaires

Les **structures de données linéaires** sont des suites séquentielles d'éléments ordonnés e_1, e_2, …, e_n.

Les tableaux python sont donc des structures linéaires, tandis que les dictionnaires ne le sont pas.

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.

Une structure est dite **homogène** si toutes les données qu'elle contient sont du même type.

Dans la plupart des langages de programmation, les tableaux comme les listes doivent être homogène, mais python permet au contraire de gérer sans difficulté des structures de données hétérogène.

On va s'intéresser à trois types de structures linéaires : les listes, les piles et les files.

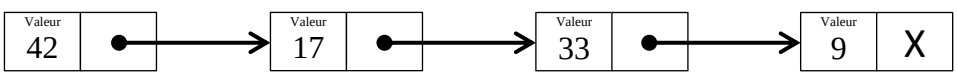
II – Les listes chaînées

Les tableaux de python (la classe `list`) sont des structures très souples qui permettent l'ajout d'éléments en fin de liste de manière assez efficace. Par contre ils ne sont pas du tout adaptés à un ajout en début ou milieu de liste. En effet, pour ajouter un élément en début de liste il faut décaler toute la liste, ce qui est une opération coûteuse (complexité linéaire).
Il existe des structures de données qui permettent un ajout n'importe où dans la liste en temps constant : les listes chaînées.

Une **liste chaînée** est une structure linéaire où les éléments sont reliés les uns aux autres, mais ne se trouvent pas forcément dans des zones de mémoire contigües.

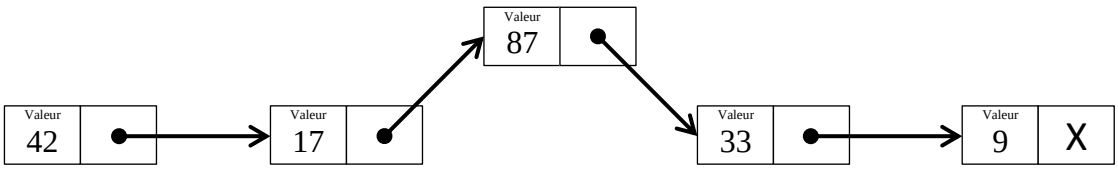
1) Principe

L'idée est que chaque élément d'une liste chaînée (chaque « maillon » ou cellule (*cell* en anglais)) embarque la valeur à représenter [1]ainsi qu'un pointeur sur le maillon suivant :



[1] Cette valeur peut correspondre à n'importe quel objet python : un entier, une chaîne de caractère, un booléen, un tableau, un dictionnaire, ou même une classe qu'on vient de programmer …

Il est alors assez simple d'insérer un élément dans la liste :



Les listes chaînées supportent en général les opérations suivantes :

- Création d'une liste vide
- Insertion d'un élément en tête de liste
- Savoir si la liste est vide
- Rechercher un élément dans la liste
- Concaténer deux listes.

Chaque élément de la liste est appelé « Cellule » (« cell » en anglais) ou maillon.

2) Implémentation en python

Pour représenter une liste, il faut partir de l'élément de base qui est le maillon. Celui-ci contient deux informations : la valeur et la référence (on dit aussi le *pointeur*) vers le maillon suivant.

Il paraît donc naturel de représenter un maillon par une classe qui aurait deux attributs : valeur et suivant.

On note que cette classe est intrinsèquement récursive puisque ses attributs font référence à elle-même (le champ `suivant` est du type `Maillon`).

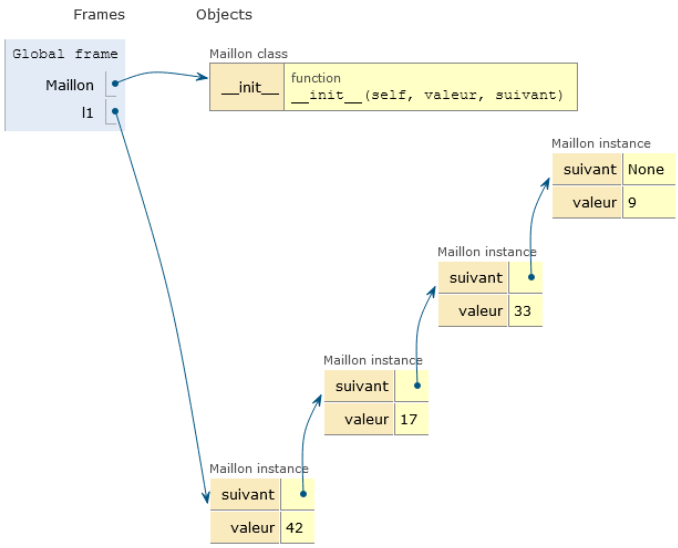
Maillon
Attributs `valeur` (objet de type quelconque) `suivant` : Maillon
Méthodes `__init__(valeur, suivant)`

```
class Maillon():
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant
```

Une liste est alors simplement une référence vers le premier maillon de la chaîne.
La création d'une liste chaînée peut se faire par :

```
l1 = Maillon(42, Maillon(17, Maillon(33, Maillon(9,None))))
```

On peut visualiser la situation en mémoire en exécutant ce code dans python tutor :



Voyons maintenant comment implémenter les différentes fonctionnalités que doit fournir une liste chaînée.

a. Création d'une liste vide

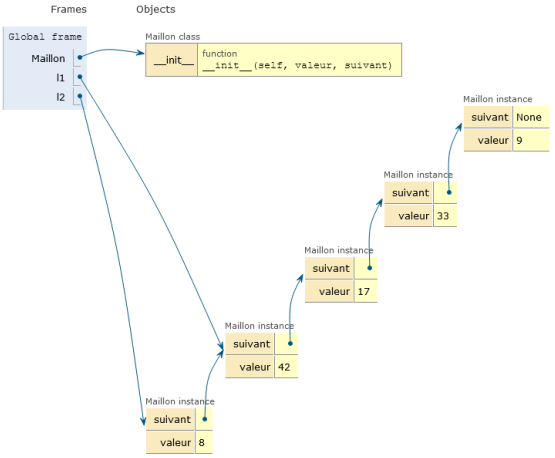
Une liste vide ne possède aucun maillon. La liste vide ne pointe donc sur aucun maillon, donc sur None.

```
def liste_vide():
    return None

lv = liste_vide()
```

b. Insertion d'un élément en tête de liste

On peut créer une deuxième liste par ajout d'un maillon en tête de la précédente :

```
def ajout_en_tete(liste, valeur):
    return Maillon(valeur,liste)

l2 = ajout_en_tete(8, l1)
```



c. Savoir si la liste est vide

Il suffit de tester si la liste est à None.

```
def est_vide(liste):
    return liste is None

print(est_vide(l1))    # Renvoie False
print(est_vide(lv))    # Renvoie True
```

d. Rechercher un élément dans la liste

Pour cela il nous faut parcourir la liste jusqu'à trouver l'élément ou à atteindre la fin. Ce parcours peut se faire sous forme récursive ou itérative.

```
def recherche_element_rec(maillon, valeur_cherchee):
    if maillon is None:
        return None
    elif maillon.valeur == valeur_cherchee:
        return maillon
    else:
        return recherche_element_rec(maillon.suivant, valeur_cherchee)

m1 = recherche_element_rec(l1, 25)  # Renvoie None
m2 = recherche_element_rec(l1, 33)  # Renvoie l'objet maillon contenant 33
```

et en itératif :

```
def recherche_element(liste, valeur_cherchee):
    maillon_actuel = liste
    while maillon_actuel is not None:
        if maillon_actuel.valeur == valeur_cherchee:
            return maillon_actuel
        maillon_actuel = maillon_actuel.suivant
    return None
```

Application 1 :

1) Ecrire une fonction `longueur(liste)` qui renvoie un entier égal à la longueur de la liste donnée en argument. Quelle est la complexité de cette fonction ?
2) Ecrire une fonction `nieme_element(n, liste)` qui renvoie la valeur du n-ième élément de la liste ou lève une exception « `IndexError` » si l'indice n'est pas valide. Quelle est la complexité de cette fonction ?

e. Concaténer deux listes.

Le plus simple est de le faire de manière récursive, le cas de base étant le cas où la première liste est vide.

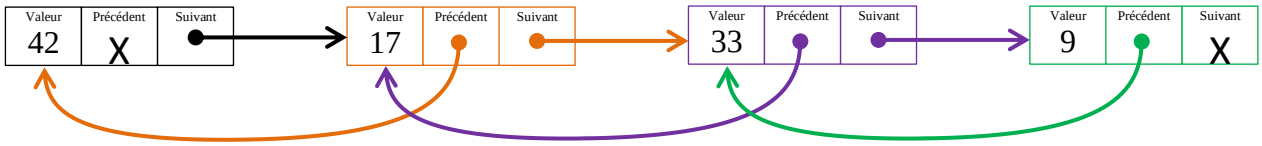
```
def concatener(l1, l2):
    if est_vide(l1):
        return l2
    return Maillon(l1.valeur,concatener(l1.suivant,l2))

l1 = Maillon(6, Maillon(7, Maillon(8, None)))
l2 = Maillon(9, Maillon(10, Maillon(11, Maillon(12, None))))
l3 = concatener(l1, l2)
```

3) Remarques

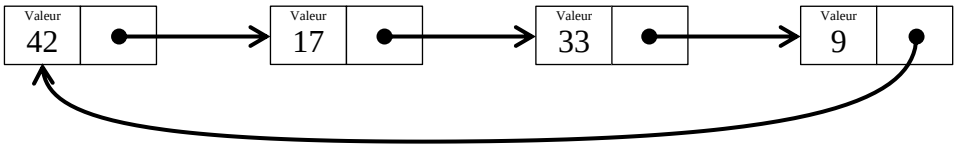
On a étudié ici le cas de listes chaînées non mutables. En effet, nos fonctions (`ajout_en_tete`, `concatener`) renvoient à chaque fois une nouvelle liste chaînée, sans modifier l'ancienne. C'est le même comportement que les chaînes de caractère (`str`) en python qui sont non-mutables. On peut bien évidemment réaliser des fonctions où les listes chaînées seraient mutables. La concaténation par exemple est bien plus simple si on se contente de relier le dernier maillon de `l1` avec le premier de `l2` (cette opération modifie `l1` mais pas `l2`). Avec des listes mutables, il faut cependant être bien plus vigilant car une fonction peut avoir pour effet de bord de modifier une liste.

4) Variantes

Les listes que nous venons de voir sont dites « simplement chaînées ». Il existe des listes « doublement chaînées » où chaque maillon est doublement lié : à la fois au maillon suivant, mais aussi au maillon précédent. Ceci permet de parcourir la liste de façon simple dans les deux sens et simplifie également l'ajout d'un élément dans la chaîne avant un maillon donné.



On peut aussi utiliser des listes cycliques, où le dernier maillon pointe sur le premier. Les fonctions de parcours de la liste doivent alors être adaptées pour éviter les boucles infinies.



III - Les piles

Une **pile** est une structure linéaire où les insertions et les suppressions se font toutes du même côté.

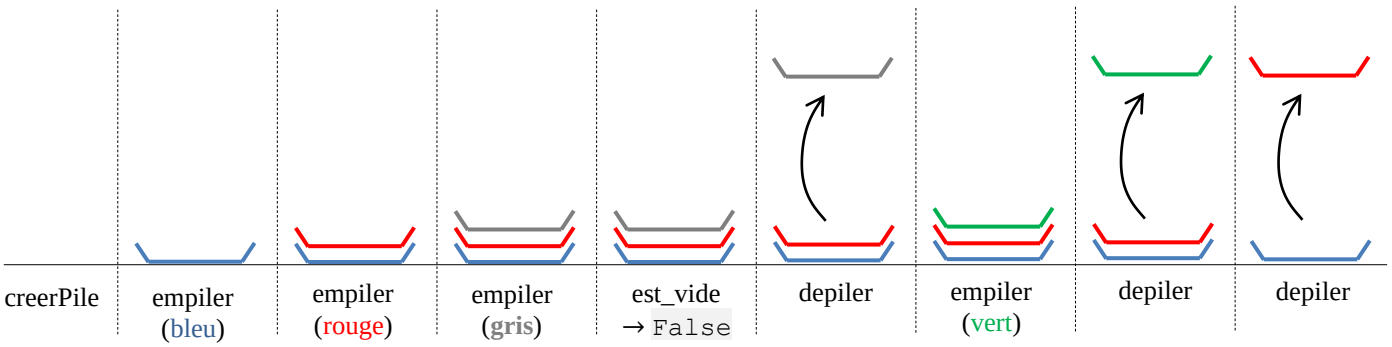
Ces structures sont comme une pile de livre : on récupère en premier le livre que l'on a déposé en dernier. Pour récupérer le premier livre, il faudra retirer (« dépiler ») tous les autres. On parle aussi de listes du type **LIFO** (**L**ast **I**n, **F**irst **O**ut) ou **stack** en anglais.

1) Principe

L'interface minimale d'une pile pourrait être la suivante :

Fonction	**Description**
creer_Pile() → Pile	Créer une pile vide
est_vide(*p*) → Booléen	Teste si la pile *p* est vide
empliler(*p* , *élément*)	Insère *élément* en tête de *p*
depiler(*p*) → *élément*	Enlève l'*élément* au sommet de la pile *p* et le renvoie

Voici ce que pourrait donner une utilisation d'une pile (ici matérialisée avec des assiettes) :



Les opérations principales sur une pile sont donc l'empilement (*push* en anglais) et le dépilement (*pull*). On ajoute souvent une méthode sommet (*peek* en anglais) pour récupérer l'élément au sommet de la pile sans la modifier (sans dépiler donc).
On ne considèrera ici que des piles sans limite de capacité, mais en réalité la plupart des piles ont une limite au nombre d'éléments pouvant être empilés (si on essaye d'ajouter des éléments au-delà de la limite, on obtient une erreur « Stack Overflow ».

Application 2 :
Dans quel état se trouve une pile initialement vide après les opérations suivantes ?

```
empiler(1)
empiler(2)
depiler()
empiler(3)
empiler(4)
empiler(5)
depiler()
depiler()
```

2) Implémentation avec une classe python

On cherche à créer une classe python qui implémentera l'interface minimale décrite plus haut.

Application 3 :

1) Quelle structure de donnée pourrait-on utiliser pour créer une pile en python ? Essayer de trouver plusieurs possibilités et donner les avantages et inconvénients de chacune.
2) Programmer la classe `Pile` en python dont le constructeur réalise la fonction `creer_Pile` (crée une pile vide), puis implémenter les méthodes `est_vide`, `empiler`, `depiler` et `__repr__` pour afficher le contenu de la pile (sans la modifier). La méthode `__repr__` peut se contenter d'afficher un tableau contenant les éléments et on peut si on le souhaite écrire une méthode `__str__` qui affichera la pile de manière plus explicite (avec un élément par ligne comme sur la pile).
3) Rajouter une méthode `sommet` qui renvoie l'élément au sommet de la pile sans le dépiler

3) Utilisation des piles

Les piles sont souvent utilisées en programmation, par exemple pour les appels de fonctions. A chaque fois qu'une fonction est appelée, on place la ligne où doit reprendre le programme après l'appel de la fonction sur la pile, puis on exécute la fonction et quand on en sort on reprend à l'endroit indiqué par la pile d'appel.

De même les fonctions Annuler (Ctrl-Z) et Refaire (Ctrl-Y) présentes dans de nombreux logiciels utilisent une pile des opérations pour pouvoir les annuler dans l'ordre inverse où elles ont été effectuées.
L'historique de navigation avec les boutons suivants et précédents (flèches vers la droite et la gauche) fonctionne aussi grâce à une structure de type pile.

IV - Les files

Une **file** est une structure linéaire où les insertions et les suppressions se font à l'opposé l'une de l'autre

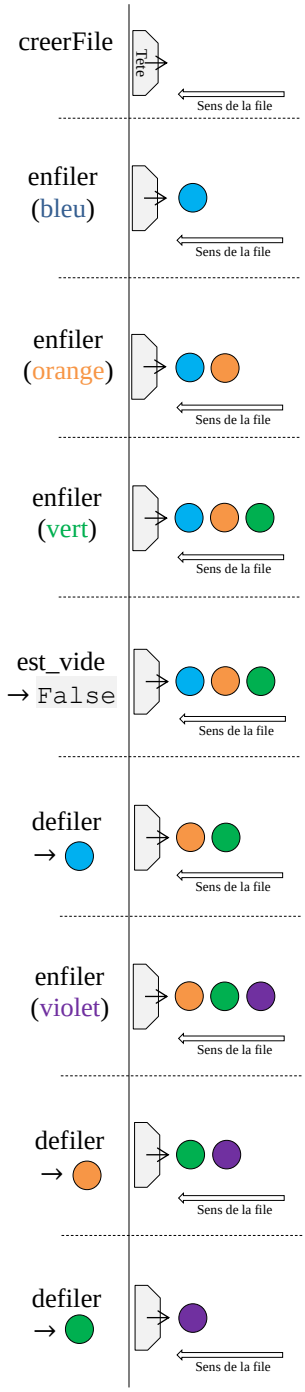
Elles sont à l'image d'une file d'attente : le premier arrivé est le premier servi. Les files sont des listes de type **FIFO** (**F**irst **I**n, **F**irst **O**ut) ou **queue** en anglais.

1) Principe

L'interface minimale d'une file pourrait être la suivante :

Fonction	**Description**
creer_File() → File	Créer une file vide
est_vide(*f*) → Booléen	Teste si la file *f* est vide
enfiler(*f* , *élément*)	Insère *élément* en queue de *f*
defiler(*f*) → *élément*	Enlève l'*élément* à la tête de la file *f* et le renvoie

Ici les opérations de base sont donc enfiler (*enqueue* en anglais) et défiler (*dequeue*).



Application 4 :

Dans quel état se trouve une file initialement vide après les opérations suivantes ?

enfiler(1)
enfiler(2)
defiler()
enfiler(3)
enfiler(4)
enfiler(5)
defiler()
defiler()

Comparer au résultat obtenu dans l'application 2 avec une pile.

2) Implémentation avec une classe python

On cherche à créer une classe python qui implémenta l'interface minimale décrite plus haut.

Application 5 :

1) Quelle structure de donnée pourrait-on utiliser pour créer une file en python ? Essayer de trouver plusieurs possibilités et donner les avantages et inconvénients de chacune.
2) Programmer la classe `File` en python dont le constructeur réalise la fonction `creer_File` (crée une file vide), puis implémenter les méthodes `est_vide`, `enfiler` et `defiler`. Rajouter une méthode `__repr__` (voire `__str__`) afin de pouvoir tester facilement les autres méthodes.

3) Utilisation des files

On utilise des files à chaque fois qu'on doit gérer des attentes. Par exemple le système d'exploitation gère une liste des processus dont il se sert pour déterminer à quel processus il donne la main à tour de rôle (puis quand le processus n'a plus la main on le replace en queue de file).

Exercices – **Listes, piles et files**

Exercices sur les listes chaînées

Exercice 1 : Classe ListeChainee

Définir une classe `ListeChainee` qui permet de gérer les listes chaînées en fournissant toutes les méthodes vues dans le cours.

1) Sur quelle classe doit s'appuyer la classe `ListeChainee` ? Doit-elle en hériter ?
2) Créer le constructeur de cette classe qui doit créer une liste vide.
3) Ajouter les méthodes vues dans le cours (vous pouvez reprendre le code en l'adaptant) : `ajout_en_tete`, `est_vide`, `recherche_element`, `longueur`, `nieme_element`, `concatener`.
4) On souhaite qu'un utilisateur puisse écrire `len(l1)` plutôt que `l1.longueur()` pour obtenir la longueur de la liste. Quelle méthode magique doit-on utiliser ? Modifier le code en conséquence.
5) Même question pour pouvoir concaténer les listes en écrivant `l = l1 + l2`.
6) La méthode magique `__getitem__(self, n)` permet d'écrire `l1[2]` plutôt que `l1.nieme_element(2)`. Modifier le code pour pouvoir obtenir ce comportement.
7) Ajouter une méthode `tableau` qui renvoie un tableau contenant toutes les valeurs de la liste (vous pouvez écrire une fonction récursive ou itérative).
8) Ajouter une méthode `__repr__` qui renvoie une chaîne de caractère représentant la liste comme un tableau (ex : `[42, 13, 33]`).
9) Ajouter une méthode `__eq__` pour tester l'égalité de deux listes. Deux listes sont égales si elles contiennent les mêmes valeurs dans le même ordre.

Bonus :

- Ajouter dans le constructeur de la classe la possibilité de fournir un paramètre optionnel représentant un tableau de valeur permettant de créer une liste avec les valeurs du tableau.
- Ajouter une méthode `__str__` pour renvoyer une représentation du type `"92 => -8 => 31 => 14"`.
- Déterminer la longueur de la liste a un coût linéaire. Pour éviter ce problème, on se propose de rajouter un attribut `taille` à notre classe `ListeChainee`. Cet attribut sera initialisé à zéro pour une liste vide et mis à jour par chaque fonction qui modifie la longueur. La fonction longueur n'aura alors plus qu'à donner la valeur du champ `taille` (donc exécution en temps constant). Implémenter ces modifications.
- Ecrire une méthode `supprime_en_tete` qui supprime le premier élément de la liste et renvoie sa valeur.

Exercice 2 : multiplie_par

On suppose que la file est constituée de nombre (ou du moins d'objets pour lesquels la multiplication est définie). Modifier la classe `ListeChainee` en écrivant une méthode `multiplie_par(n)` qui multiplie tous les éléments de la liste par le nombre n fourni en argument.

Exercice 3 : double

Modifier la classe `ListeChainee` en écrivant une méthode `double` qui duplique tous les éléments de la liste. Ainsi la liste A => B => C donnera la liste A => A => B => B => C => C.

Exercice 4 : maxi

On suppose que la file est constituée de nombre (ou du moins d'objets pour lesquels la méthode `__lt__` est définie). Ecrire la méthode `maxi` qui renvoie le plus grand élément de la liste chainée. (On ne traitera pas le cas de la liste vide).

Exercice 5 : nb_occurences

Ecrire la méthode `nb_occurences(element)` qui prend en paramètre un élément et qui renvoie le nombre d'occurrences de cet élément dans la liste chainée.

Exercice 6 : inverse

Ecrire une méthode `inverse` qui inverse l'ordre des éléments de la liste. Ainsi A => B => C deviendra C => B => A.
Bonus pour les plus rapides : Trouver une méthode qui ne nécessite pas la création d'une nouvelle liste, mais réarrange les maillons de la chaîne existante.

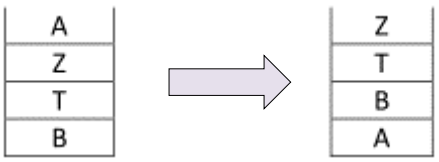
Exercices sur les piles
Les exercices suivants doivent être traités en utilisant uniquement des piles à l'exclusion de toute autre structure linéaire. On pourra utiliser n'importe quelle implémentation de pile vues dans les applications du cours.

Exercice 7 : inverse, le retour

Ecrire une fonction inverse(p) qui renvoie l'inverse de la pile p.

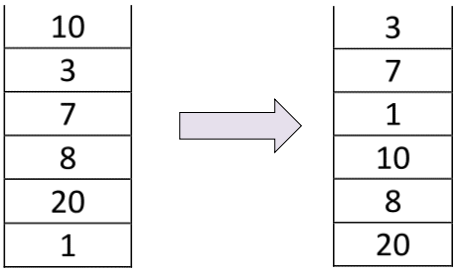
Exercice 8 : décalage vers le haut

Ecrire une fonction `decalage_haut(p)` qui renvoie une pile où tous les éléments de la pile p sont décalées d'un cran vers le haut (et l'élément le plus en haut de la pile se retrouve tout en bas de la pile (voir exemple ci-contre).



Exercice 9 : pairs_en_bas

On suppose que la pile est constituée d'entiers naturels. Ecrire une fonction `pairs_en_bas(p)` qui renvoie une pile correspondant à la pile p où tous les éléments pairs sont en bas (en gardant l'ordre donné au départ).
Voir l'exemple ci-contre.



Exercices sur les files
Les exercices suivants doivent être traités en utilisant uniquement des files à l'exclusion de toute autre structure linéaire. On pourra utiliser n'importe quelle implémentation de file vues dans les applications du cours. On procédera par ajout de méthodes à la classe File (qui modifient donc l'objet file). On peut soit recopier la classe, soit créer une nouvelle classe héritant de la classe File et qui implémente une méthode supplémentaire.

Exercice 10 : double, plusieurs versions

On suppose que la file est constituée de nombre (ou du moins d'objets pour lesquels la multiplication est définie).

1) Ecrire une méthode `double_v1` qui multiplie par 2 tous les éléments de la file.
2) Ecrire une méthode `double_v2` qui duplique tous les éléments de la file. A <= B <= C donnera la liste A <= A <= B <= B <= C <= C.
3) Ecrire une méthode `double_v3` qui double la taille de la liste en la dédupliquant. A <= B <= C donnera la liste A <= B <= C <= A <= B <= C.

Exercice 11 : passer_devant_tout_le_monde

Ecrire la méthode `passer_devant_tout_le_monde` qui doit faire comme `enfiler` mais au lieu de faire rentrer l'élément à la dernière place on le fait entrer devant tout le monde (en tête de file).

Exercice 12 : defiler_par_derriere

Ecrire la méthode `defiler_par_derriere` qui doit faire comme `defiler` mais au lieu de retirer le bon élément, elle retire (et renvoie) l'élément qui est le dernier de la file.

Exercice 13 : inverse… pour les files

Ecrire la méthode `inverse` qui inverse les éléments de la file. A <= B <= C <= D donnera la liste D <= C <= B <= A.
Aide : On pourra utiliser `defiler_par_derriere()`…

Pour les exercices suivants, aucune structure de donnée n'est imposée. C'est à vous de voir laquelle serait la plus adaptée.

Exercice 14 : Chaînes bien parenthésées

On dit qu'une chaîne de caractère est bien parenthésée lorsque chaque parenthèse ouvrante est associée à une unique fermante du même type (« `()` » ou « `[]` ») et réciproquement.
Ecrire une fonction `bien_parenthesee(c)` qui renvoie `True` si la chaîne `c` est bien parenthésée (en prenant en compte les parenthèses rondes et les crochets) et `False` sinon.
Ex : `bien_parenthesee("(4+5)*[(2*3+4)/5]")` renvoie `True`,
`bien_parenthesee("(4+5)*((2*3+4]/5]")` renvoie `False`
`bien_parenthesee("(4+(5)*7-)[2(*3+4)]/5")` renvoie `True`
`bien_parenthesee("(3-[5)*7+4]")` renvoie `False`

Exercice 15 : Problème de Flavius Josèphe

Flavius Josèphe est un historiographe romain juif du 1er siècle, dont l'œuvre historique est sujette à caution. Il a donné la première version du problème suivant : "41 soldats juifs, cernés par des soldats romains, décident de former un cercle. Un premier soldat est choisi au hasard et est exécuté, le troisième à partir de sa gauche est ensuite exécuté. Tant qu'il y a des soldats, la sélection continue en en tuant un sur deux. Le but est de trouver à quel endroit doit se tenir un soldat pour être le dernier. Josèphe, peu enthousiaste à l'idée de mourir, parvint à trouver l'endroit où se tenir. Quel est-il ?"
Question : Ecrire un programme python pour répondre à ce problème.
Variante : 42 soldats juifs, deux survivants, et les romains en tuent un sur trois.

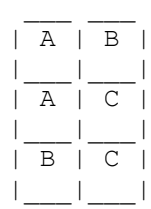
Exercice 16 : Optimisation des containers

Des camions convoient des containers dédiés à l'exportation vers un terminal portuaire.
Les containers seront stockés temporairement dans le terminal en attendant d'être chargés sur les bateaux qui arrivent. Pour économiser l'espace, les containers sont empilés les uns sur les autres dans le terminal.
Il y a différents bateaux qui ont chacun une destination différente. Chaque container est à charger sur un bateau précis et chaque bateau peut avoir un ou plusieurs containers.
Les opérateurs du terminal doivent prendre dans le stockage temporaire les bons containers pour les charger sur le bon bateau. Si le bon container est en haut de la pile, il sera plus facile de le récupérer et de le charger dans le bateau. Mais si le container que l'on veut se trouve sous d'autres containers, il faudra d'abord retirer et mettre de côté ces containers avant d'accéder au container voulu, ce qui occasionnera des délais un coût supplémentaire.

Pour gérer au mieux le terminal des containers, vous avez besoin de planifier leur stockage.

Il y a potentiellement 26 bateaux différents notés de A à Z partant chacun vers une destination différente.
Vous connaissez le planning d'arrivé des bateaux : le bateau A arrive en premier, le B en deuxième, suivi du C, … Les bateaux arrivent toujours dans l'ordre alphabétique.
Vous connaissez également l'ordre d'arrivée des camions qui amènent les containers. Tous les containers arrivent AVANT le premier bateau.
Vous devez planifier la façon dont les containers seront stockés quand les camions arriveront de manière à minimiser l'espace occupé au sol par les containers et permettre ensuite un chargement efficace des bateaux.

Exemple :
Six camions arrivent avec des containers dans l'ordre suivant : BACCBA
B est le container pour le bateau B. Ce container arrive en premier
Il est suivi d'un container A pour le bateau A, puis de deux containers C pour le bateau C, …
On peut stocker ces containers en deux piles représentées ci-contre.
Quand le bateau A arrivera, on chargera les deux caisses du haut de la première pile.
Quand B sera à quai, on chargera la dernière caisse de la première pile, puis la première de la deuxième pile.
Enfin quand le bateau C sera là, on y installera les deux derniers containers de la deuxième pile.
Le processus complet utilisera seulement 2 piles de containers.
Notez que nous ne pouvons pas trier les containers avant de les stocker car ce serait trop long et coûteux en pratique.



Maintenant c'est votre tour. Vous devez écrire un programme qui connaissant l'ordre d'arrivée des camions détermine le nombre et la constitution des piles pour assurer le chargement le plus efficace possible.