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)`

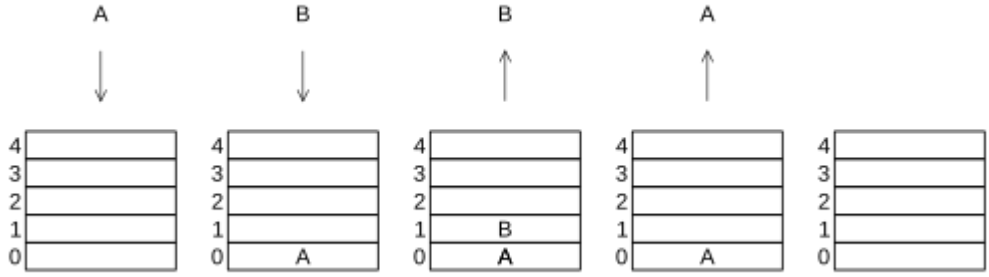
Structures de données linéaires : piles et files

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 piles, les files et les listes. Dans cette première étape, on traitera uniquement piles et files ; on fera les listes après avoir vu les arbres.

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.

Piles

Une pile est une structure linéaire où les insertions et les suppressions se font toutes du même côté, à l'image d'une pile d'assiettes : on rajoute les nouvelles assiettes au sommet de la plie, on prend des assiettes sur le sommet de la pile également. Les piles sont appelées **stack** ou **LIFO** en anglais (last in, first out : le dernier élément rentré est le premier sorti).



Une **interface** (c'est à dire la liste des opérations que l'on peut faire) d'une pile peut être :

Fonction (créerPile)/méthode(les autres)	Description
creer_pile() → Pile	Créer une pile vide
est_pile_vide(*p*) → Booléen	Teste si la pile *p* est vide

Fonction (créerPile)/méthode(les autres)	Description
empliler(*p* , _élément_)	Insère *élément* en tête de *p*
depiler(*p*) → *élément*	Enlève l'*élément* au sommet de la pile *p* et le renvoie
sommet(*p*) → *élément*	Renvoie l'*élément* au sommet de la pile *p*

Exercice 1 sur les piles

Dans quel état se trouve une pile vide après les opérations suivantes

- empiler(1)
- empiler(2)
- dépiler
- empiler(3)
- empiler(4)
- empiler(5)
- dépiler
- dépiler

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** (la signature est l'interface en un peu moins détaillée). Le type de données `Pile` 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.

Exercice 2 sur les piles

Créer le type `Pile` , le tester. Si vous avez implémenté ce type d'une manière différente de

Entrée [ ]:

```
from random import randint

class Pile :
    def __init__(self) :
        self.pile = []

    def est_pile_vide(self):
        return

    def empiler(self , element):
        return

    def depiler(self):
        if self.est_pile_vide():
            raise IndexError("la pile est déjà vide")
        else :
            pass

    def __repr__(self):
        if self.est_pile_vide() :
            return 'Pile vide'
        else:
            return str(self.pile)

def creer_pile():
    # Comme pour les listes, le codage de cette primitive sous for
    return Pile()


# un test parmi d'autres
a = Pile()                  # ou a = creerPile() si on veut vraimer
print("on empile")
for i in range(6):
    a.empiler(randint(1, 20))
    print(a)
print("on dépile")
while not a.est_pile_vide():
    a.depiler()
    print(a)
```

il est légitime de se demander à quoi sert une pile en informatique. On en trouve dans la gestion des modifications de documents dans les traitements de texte. Dans LibreOffice, ctrl-z permet d'annuler la dernière modification du texte, en "dépilant". On peut itérer cette opération. De même dans les navigateurs, le bouton "page précédente" cache une pile conservant les adresses visitées. Plus généralement, on a précédemment mentionné les "piles d'appel", notamment en programmation récursive. C'est bien une structure du type abstrait `pile` qui est utilisée.

Exercice 3 sur les piles : pour ceux qui vont vite

Ecrire une deuxième implémentation de la structure `Pile` . On utilise la classe `Cellule` donnée ci-dessous pour cela. On peut réaliser `Pile` très économiquement à partir de cette dernière : une pile comporte le `haut` de la pile, et la `suite` , qui est soit `None` , on est

alors en bas de la pile, soit une autre cellule (donc une autre pile). On a une construction récursive de la pile (faire un schéma peut aider à comprendre cette construction).
Dans le code proposé ci-dessous, il ne reste que la méthode `depiler` à coder

Entrée [ ]:

```
class Cellule :
    # On reprend la définition de la cellule d'une liste chaînée,
    def __init__(self, haut, suite) :
        self.haut = haut
        self.suite = suite
    def __repr__(self):
        if self.haut is None :
            return 'cellule vide'
        elif self.suite is None :
            return str(self.haut)
        else :
            return str(self.haut) + "-" + str(self.suite)

class Pile :
    def __init__(self) :
        self.pile = None

    def est_pile_vide(self):
        return self.pile is None

    def empiler(self , element):
        self.pile = Cellule(element , self.pile)

    def depiler(self):

    def __repr__(self):
        if self.est_pile_vide() :
            return 'Pile vide'
        elif self.pile.suite is None:
            return repr(self.pile.haut)
        else :
            return repr(self.pile.haut) + '-' + repr(self.pile.sui

def creer_pile():
    return Pile()

ma_poule = creer_pile()
print(ma_poule)
ma_poule.empiler(1)
print(ma_poule)
ma_poule.empiler(2)
print(ma_poule)
ma_poule.depiler()
print(ma_poule)
ma_poule.empiler(3)
print(ma_poule)
ma_poule.empiler(4)
print(ma_poule)
ma_poule.empiler(5)
print(ma_poule)
ma_poule.depiler()
print(ma_poule)
ma_poule.depiler()
```

Encore un exercice (plus amusant... et de type bac)

La notation polonaise inverse (notation postfixe) est une manière de noter les calculs sans utiliser de parenthèses. Cette notation a été utilisée par certaines calculatrices, notamment de Hewlett-Packard.
Exemple : calcul de 7 8 * 2 +

- On lit les deux premiers nombres et l'opérateur, on calcule 7*8 = 56
- On garde le 56 en tête, on lit le nombre et l'opérateur suivant : 56 + 2 = 58 qui est le résultat du calcul

Pour cet exercice, on n'utilisera que des nombre positifs et pas de division (pour éviter la division par 0) . L'évaluation d'une expression est simple et utilise une pile :

- Initialement la pile est vide
- Si on trouve un nombre, on l'empile
- Si on trouve un opérateur, on dépile deux fois pour trouver les deux opérandes (attention à l'ordre pour la soustraction, non symétrique). On effectue l'opération, et on empile le résultat
- Le résultat de l'opération se lit en sommet de pile.

Remarque : pour la programmation, il vaut mieux traiter d'abord le cas des opérateurs puis le cas des nombres dans les si/sinon si/sinon.

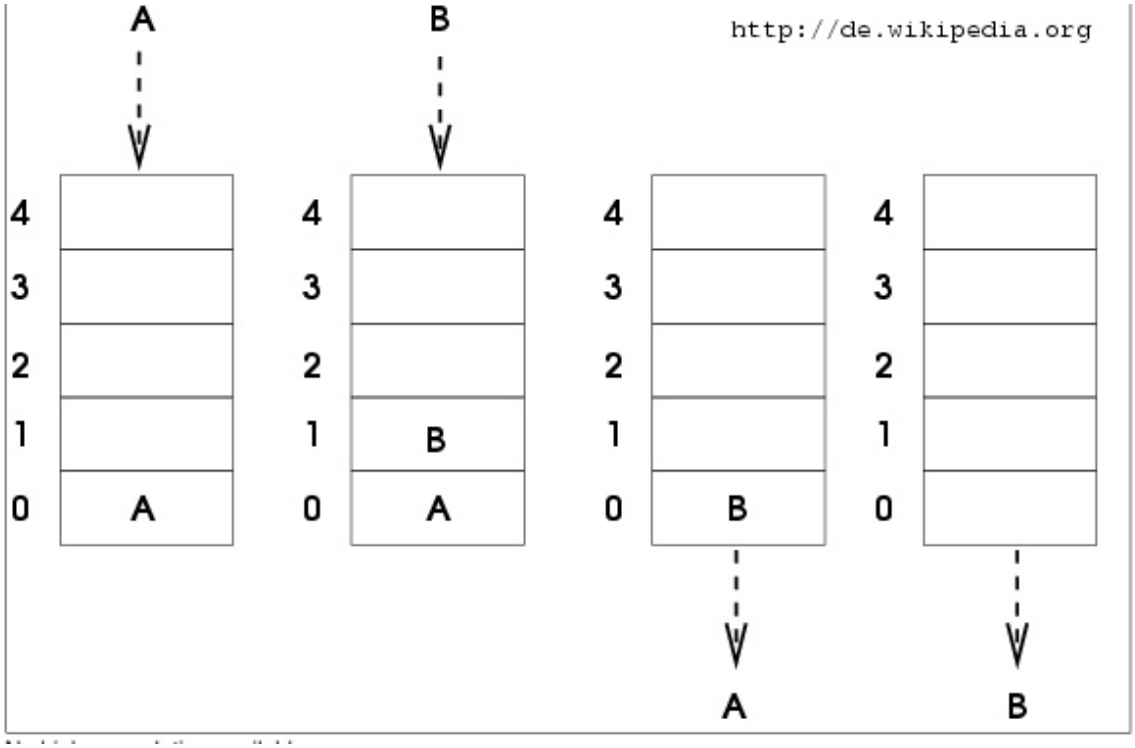
1. Sur papier, donner le résultat de 1 2 3 4 + x 5 x - 7 + . Ecrire ce calcul sous forme infixe (c'est-à-dire de la manière usuelle avec les parenthèses).
2. Ecrire une fonction Python qui, étant donnée une chaîne de caractères exprimant un calcul sous forme postfixe, donne le résultat du calcul, sous les préconditions : nombres positifs, pas de division, expression "bien formée". La tester.
   Rappel : la méthode `split` permet d'obtenir une liste comportant les différents "mots" d'une chaîne de caractères. Utilisée sans arguments, le séparateur est l'espace : `'un deux trois'.split()` renvoie `['un', 'deux', 'trois']` .

Entrée [ ]:

```
1
2 def calcul_postfixe(calcul) :
3     pile = Pile()
4     for char in calcul.split() :
5         #... à compléter (et enlever le pass)
6         pass
7     return
8
9 print(calculPostFixe("1 2 3 4 + * 5 * - 7 + "))
```

Files

Une file est une structure linéaire où les insertions et les suppressions se font à l'opposé l'une de l'autre, à l'image d'une file d'attente : le premier arrivé est le premier servi. Les piles sont appelées *FIFO* ou queue en anglais (first in, first out).



Une **interface** possible d'une file est :

Fonction	Description
creer_file() → Pile	Créer une file vide
est_file_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* aen tête de la file *f* et le renvoie
tete(*f*) → *élément*	Renvoie l'*élément* en tête de la file *f*

Exercices Files 1

1. Dans quel état se trouve une pile vide après les opérations suivantes
   - enfiler(1)
   - enfiler(2)
   - défiler
   - enfiler(3)
   - enfiler(4)
   - enfiler(5)
   - défiler
   - défiler

   Comparer avec l'exercice correspondant sur les piles
2. Créer le type `File` , le tester. Si vous avez implémenté ce type d'une manière différente de votre voisin, vous pouvez tester et comparer l'efficacité de vos implémentations avec `%timeit (mon_test(...))`

Entrée [ ]:

```
from random import randint

class File :
    # Implémentation avec un inconvénient majeur : defiler(self) e
    #est en temps constant, autant liste.pop(0) est en temps linéa
    def __init__(self) :


    def est_file_vide(self):
        return

    def enfiler(self , element):


    def defiler(self):
        if self.est_file_vide():
            raise IndexError("la pile est déjà vide")
        else :
            pass

    def __repr__(self):
        if self.est_file_vide() :
            return 'File vide'
        else:
            return

def creer_file():
    return File()

# un test parmi d'autres
a = creer_file()
print("on enfile")
for i in range(6):
    a.enfiler(randint(1, 20))
    print(a)
print("on défile")
while not a.est_file_vide():
    a.defiler()
    print(a)
```

Exercices Files suite

3 . Ecrire un programme qui permet de retourner une pile, en utilisant uniquement une file (exercice proposé pour le bac).

Commencer par faire un schéma avec la pile que l'on met dans la file, puis la file que l'on remet dans la pile initiale pour imaginer l'algorithme.

On utilisera uniquement les primitives associées aux pile et files (interdiction d'utiliser `pop` , `append` , `len` etc.)

Entrée [ ]:
```
def retourner_pile(mapile):

    return mapile

a = creer_pile()
for i in range(6):
    a.empiler(i)
print(a)
print(retourner_pile(a))
```

On constate avec cette implémentation, très différente a priori de ce que vous avez fait précédemment, que le type abstrait `File` peut être traduit en code de manières très différentes les unes de autres.

Le module deque

Le type `deque` (double ended queue, se lit deck) permet l'implémentation directe d'une file, où les primitives `enfiler` et `defiler` sont en complexité temporelle constante $O(1)$. On donne ci-dessous une exemple de code permettant la création d'une file. Ce type fait partie de la bibliothèque `collections` .
Exercice : reprendre la fonction qui permet d'inverser une pile avec une file, et la réécrire en utilisant `deque` (et éventuellement une pile formée tout simplement à partir d'un tableau dynamique Python, type `list`).

Entrée [ ]:
```
from collections import deque
file = deque()
for i in range(6):
    file.append(randint(1, 20))
    print(file)

for i in range(6):
    file.popleft()
    print(file)
```

Entrée [ ]:
```
# On peut reprendre le code de "Retourner une pile", il fonctionne

def retourner_pile(pile):
    file=deque()


    return pile

poil = []
for i in range(6):
    poil.append(i)
print(poil)
print(retournerPile(poil))
```

Exercice Files suite et fin (encore un exercice de type bac)

4. Implémenter une structure de file à l'aide de deux piles. Pour cela, une des piles est l'entrée, l'autre la sortie. Les deux sont liées. Lorsque l'on ajoute un élément, on l'empile sur l'entrée. Lorsque l'on retire un élément, si la pile de sortie n'est pas vide alors on la

dépile (forcément le premier élément). Si la pile de sortie est vide, alors on retourne la pile d'entrée en la mettant sur la pile de sortie (on transforme au passage un structure LIFO en FIFO, puisqu'on inverse la pile d'entrée). Remarquons au passage que la file est vide si et seulement si les deux piles sont vides.

Entrée [ ]:

```
class Pile :
    def __init__(self) :
        self.pile = []

    def est_pile_vide(self):
        return self.pile == []

    def empiler(self , element):
        self.pile.append(element)

    def depiler(self):
        if self.est_pile_vide():
            raise IndexError("la pile est déjà vide")
        else :
            return self.pile.pop()

    def __repr__(self):
        if self.est_pile_vide() :
            return 'Pile vide'
        else:
            return str(self.pile)

def creerPile():
    return Pile()

class File:
    def __init__(self):
        self.entree = Pile()
        self.sortie = Pile()

    def est_file_vide(self):
        return

    def enfiler(self , element):

    def defiler(self):

    def __repr__(self):
        if self.est_file_vide() :
            return 'File vide'
        else:
            return repr(self.entree) + " - " + repr(self.sortie)

def creerFile():
    return File()

# un test parmi d'autres
a = File()
for i in range(4):
    a.enfiler(randint(1, 20))
    print(a)
for i in range(2):
    print("défiler : ",a.defiler())
    print(a)
for i in range(4):
```

```
    a.enfiler(randint(1,20))
    print(a)

while not (a.estFileVide()):
    print("défiler : ",a.defiler())
    print(a)
```

Exercice complémentaire

Dans cet exercice, la structure de données à utiliser parmi pile et file n'est pas précisée : c'est à vous de la déterminer.

Bon parenthésage. On donne une chaîne de caractères dans laquelle figurent des parenthèes ouvrantes `(` , fermantes `)` , et de même pour les crochets `[` et `]` . Ecrire une fonction qui vérifie le bon parenthésage de l'expression. `([()])` est bien parenthésée, `([()]` et `([(])` ne le sont pas (il manque une `)` dans le premier cas, et dans le deuxième `)` et `]` sont inversés)

Entrée [ ]:
```

```

Sources :

- cours de Clémentine Nebut, Université de Montpellier II
- Wikipedia
- Types de Données et Algorithmes, C. Froidevaux, MC Gaudel, M Soria
- Eléments d'Algorithmique, D. Beauquier, J. Berstel, Ph. Chrétienne
- Document d'accompagnement éduscol Terminale NSI

---

frederic.mandon @ ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France (2015-2019)