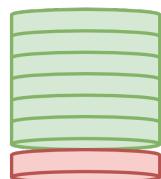


Dans une **file** (en anglais *queue*), chaque opération dépend de l'élément qui arrive et (3) sont servies dans leur ordre d'arrivée.
D'abord dans laquelle (1) les personnes arrivent à tour de rôle, (2) patientent avant d'être ajouté le premier. Pour imaginer cette structure, on pense à une file d'assiettes : on ajoute une assiette sur le sommet et quand on retire une assiette, le plus récemment. Pour imaginer cette structure, il suffit de penser à une pile d'assiettes : on ajoute une assiette sur le sommet et quand on retire une assiette, c'est forcément celle du sommet.



Dernier entré, premier sorti (en anglais **LIFO** pour last in, first out)

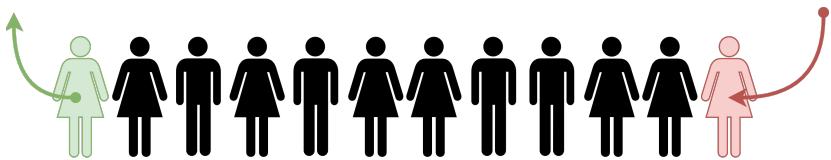
Dans une **pile** (en anglais *stack*), chaque opération dépend de l'élément qui arrive le plus récemment. Pour imaginer cette structure, il suffit de penser à une pile d'assiettes : on ajoute une assiette sur le sommet et quand on retire une assiette, tout comme les tableaux, **Pile et File** sont des structures de données qui permettent de (1) stocker des ensembles d'objets et (2) ajouter/retirer des objets un à un.

Tout comme les tableaux, **Pile et File** sont des structures de données qui permettent de (1) stocker des ensembles d'objets et (2) ajouter/retirer des objets un à un.

6.1 – Introduction

Chap. 6 – Piles et files

[1]: from doctest import testmod



Premier arrivé, premier sorti (en anglais **FIFO** pour *first in, first out*)

6.2 – Interface commune aux piles et aux files

Classiquement, chacune de ces deux structures a une **interface** proposant au minimum les quatre opérations suivantes :

pile	file	opérations
Pile()	File()	créer une structure initialement vide
est_vide()	est_vide()	tester si une structure est vide
empile()	enfile()	ajouter un élément à une structure
depile()	defile()	retirer et obtenir un élément d'une structure

REMARQUE

Comme pour les tableaux et les listes chaînées, on préconisé pour les piles et les files une **structures homogènes**. C'est-à-dire que tous les éléments stockés aient le même type.

REMARQUE

Dans ce cours, nos structures de pile et de file seront considérées **mutables** : chaque opération d'ajout ou de retrait d'un élément **modifie la pile ou la file** à laquelle elle s'applique.



6.3 – Interface et utilisation d'une pile

Mais il aurait été tout à fait possible d'en décider autrement.

Détailons l'interface des piles.

Interface et commentaires

Pile[T]

Le type des piles contenant des éléments de type T. Par exemple T peut être int pour les nombres entiers ou encore str pour les chaînes de caractères

est_vide(p: Pile[T]) -> bool prend en paramètre une pile p et renvoie un boolean indiquant si la pile est vide ou pas aucun paramètre et renvoie une pile vide capable de contenir n'importe quel type d'élément T.

empiler(p: Pile[T], e: T) -> Note ajoute de l'élément e au sommet de la pile p (push en anglais) qui prend en paramètre la pile p et renvoie l'élément qui en a été retiré. On suppose que la pile est non vide et une exception est levée le cas échéant

depiler(p: Pile[T]) -> T retiret de l'élément au sommet de la pile p (pop en anglais) qui prend en paramètre une pile p et renvoie l'élément de la pile qui suit de l'élément de la pile avec l'élément de type T homogène avec l'élément qui suit de la pile.

2] : TestResults(failed=0, attempted=47)

```
    return self.sortie.empiler(valuer)
    if self.sortie.est_vide():
        raise IndexError("depiler sur file vide")
    return self.sortie.depiler()
```

testmo()

```
self.sortie.empiler(valuer)
if self.sortie.est_vide():
    raise IndexError("depiler sur file vide")
```

```
return self.sortie.depiler()
```

testmo()

Exemple

Exemple d'utilisation des piles : Considérons un navigateur web dans lequel on s'intéresse à deux opérations : **aller** à une nouvelle page et **revenir** à la page précédente. On veut que le bouton de retour en arrière permette de remonter une à une les pages précédentes, et ce jusqu'au début de la navigation.

En plus de l'adresse courante, qui peut être stockée dans une variable à part, il nous faut donc conserver l'ensemble des pages précédentes auxquelles il est possible de revenir. *Puisque le retour en arrière se fait vers la dernière page qui a été quittée, la discipline **dernier entré, premier sorti** des piles est exactement ce dont nous avons besoin pour cet ensemble.*

6.4 – Interface et utilisation d'une file

Comme pour les piles, on note `File[T]` le type des files contenant des éléments de type `T`.

interface	explications et commentaires
<code>File[T]</code>	le type des files contenant des éléments de type <code>T</code>
<code>creer_file() -> File[T]</code>	créer une file vide
<code>est_vide(f: File[T]) -> bool</code>	renvoie <code>True</code> si <code>f</code> est vide et <code>False</code> sinon
<code>enfiler(f: File[T], e) -> None</code>	ajoute l'élément <code>e</code> à la fin de la file <code>f</code>
<code>defiler(f: File[T]) -> T</code>	retirer et renvoyer l'élément situé au début de la file <code>f</code>

```
>>> f = File()
>>> print(type(f.entre)
<class '__main__.Pile'>
>>> print(type(f.sortie))
<class '__main__.Pile'>
"""
self.entre = Pile()
self.sortie = Pile()

def est_vide(self):
"""
Exemples et tests :
>>> f = File()
>>> print(f.est_vide())
True
>>> f.entre.empiler(1)
>>> print(f.est_vide())
False
>>> f.entre.depiler()
1
>>> f.sortie.empiler(1)
>>> print(f.est_vide())
False
"""
return self.entre.est_vide() and self.sortie.est_vide()

def enfiler(self, valeur):
"""
Exemples et tests
>>> f = File()
>>> f.enfiler(1)
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> f.entre.depiler()
3
>>> f.entre.depiler()
2
>>> f.entre.depiler()
1
"""
self.entre.empiler(valeur)

def defiler(self):
"""
Exemples et tests :
>>> f = File()
>>> f.enfiler(1)
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> f.defiler()
1
>>> f.defiler()
2
>>> f.defiler()
3
>>> f.defiler()
Traceback (most recent call last):
IndexError: defiler sur file vide
"""
if self.sortie.est_vide():
    while not self.entre.est_vide():
        valeur = self.entre.depiler()
```

Exemple d'utilisation des files : Considérons le jeu de cartes de la bataille. Chaque joueur possède un paquet de cartes et pose à chaque manche la carte prise **sur le dessus du paquet**. Le vainqueur de la manche récupère alors les cartes posées, pour les placer **au-dessous de son paquet**.

En plus des cartes posées au centre de la table nous avons besoin de conserver en mémoire le paquet de cartes de chaque joueur. Puisque les cartes sont remises dans un paquet à une extrémité et prélevées à l'autre, la discipline **premier entré, premier sorti** des files est exactement ce dont nous avons besoin pour chacun de ces ensembles.

La structure de liste chaînée donne une manière élégante de réaliser une pile.

Empiler un nouvel élément revient à ajouter un nouveau mailon en tête de liste, tandis que dépiler un élément revient à supprimer le mailon de tête.

On peut ainsi construire une classe File qui définit par un unique attribut contenu en définissant son attribut contenu comme la liste vide None.

ACTIVITE 1

```
[2]: from doctest import testmod
>>> print(p)
None
```

Exemple et test :

```
>>> p = File()
implémenter le constructeur de la classe File qui construit une pile vide
en définissant son attribut contenu comme la liste vide None.
```

6.5 – Réalisation d'une pile avec une liste chaînée

Exemples et tests :

```
[12]: class File:
    def __init__(self):
        self.liste = []
    def entrée(self, élément):
        self.liste.append(élément)
    def sortie(self):
        if len(self.liste) > 0:
            return self.liste.pop(0)
        else:
            print("la pile est vide")
    def empile(self, élément):
        self.liste.insert(0, élément)
    def est_vide(self):
        return len(self.liste) == 0
    def __str__(self):
        return str(self.liste)

f = File()
f.entrée(1)
f.entrée(2)
f.entrée(3)
f.sortie()
print(f.est_vide())
False
f.sortie.empile(1)
f.sortie.empile(2)
f.sortie.empile(3)
f.sortie()
print(f.est_vide())
True
f.est_vide()
```

IndexError : défiler sur file vide

```
Traceback (most recent call last):
 3     f.defiler()
 2     f.defiler()
 1     f.defiler()
>>> f.defiler()
IndexError: défiler sur file vide
```

```

class Maillon:
    """ Maillon d'une liste chaînée """
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class Pile:
    """
    Encapsulation des piles à l'aide des Maillons de listes chaînées.
    """
    def __init__(self):
        """
        Constructeur de Pile.

        Exemple et tests:
        >>> p = Pile()
        >>> print(p.contenu)
        None
        """
        self.contenu = None

# tests de la classe
testmod()

```

2] : TestResults(failed=0, attempted=2)

ACTIVITÉ 2

Etendre la classe Pile en implémentant la méthode `est_vide`.

Exemples et tests :

```

>>> p = Pile()
>>> print(p.est_vide())
True
>>> p.contenu = Maillon(1, None)
>>> print(p.est_vide())
False

```

[3]: # attention, lorsqu'on implémente une
classe en une seule fois, il faut écrire
'class Pile:'.
La syntaxe 'class Pile(Ma_Classe):' est utilisée
dans ce notebook (et dans les juges en lignes)
pour étendre la classe existante Ma_Classe et lui
ajouter de nouvelles méthodes.

Ici, la classe Pile s'étend elle même !

à un tous les éléments de la pile d'entrée sur la pile de sortie. En effet, le premier élément prélevé sur la pile d'entrée est le dernier entrant (discipline LIFO de la pile utilisée), c'est-à-dire celui qui devra sortir de la file après tous les autres (discipline FIFO de la file que l'on veut réaliser), et il sortira bien le dernier puisqu'il sera ajouté le premier sur la pile de sortie (discipline LIFO de la pile utilisée).

- On peut alors maintenant dépiler le premier élément de la nouvelle pile de sortie, qui, s'il y a eu retournement, était auparavant le dernier élément de la pile d'entrée. À moins que cette pile soit encore vide (ce qui signifierait que les deux piles étaient vides), et donc la file est également vide.



ACTIVITÉ 10

Implémenter la classe File en utilisant deux pile.

Exemples et tests :

```

>>> f = File()
>>> print(type(f.entre'e))
<class '__main__.Pile'>
>>> print(type(f.sortie))
<class '__main__.Pile'>

```

```

>>> f = File()
>>> print(f.est_vide())
True
>>> f.entre'e.empiler(1)
>>> print(f.est_vide())
False
>>> f.entre'e.depiler()
1

```

Implémenter la méthode empiler dans la classe File. Pour cela construire une nouvelle liste chaînée dont le premier mailion contient :
— valeur : la valeur à empiler
— suivant : le premier mailion de la liste d'origine de la pile.
Puis mettre à jour le contenu de la pile avec cette nouvelle liste.
Exemple et tests :

```
>>> p = File()  
>>> p.empiler(1)  
>>> p.empiler(2)  
>>> p.est_vide()  
>>> p.est_vide()  
>>> p.valeur()  
>>> p.empiler(3)  
>>> p.valeur()  
>>> p.empiler(4)  
>>> p.valeur()  
>>> p.empiler(5)  
>>> p.valeur()
```

ACTIVITÉ 3

[3] : TestResults (failed=0, attempt=4)

```

    bool: True si et seulement si la pile est vide
    Returns:
        Est ce que la pile est vide ?
    def est_vide(self) -> bool:
        return self.contentu is None
    testsmod():
        """ Tests pour la méthode est_vide() """
        p = Pile()
        print(p.est_vide())
        assert p.est_vide() == True
        p.push(1)
        print(p.est_vide())
        assert p.est_vide() == False
        p.pop()
        print(p.est_vide())
        assert p.est_vide() == True

```

Quand une pile ainsi créée est vide lorsque ces deux piles sont toutes les deux vides. Ajouter un nouvel élément consiste simplement à empiler cet élément sur la pile d'entrée.

Retirer un élément est l'opération la plus délicate.

1. Pour commencer, deux cas de figures :

- Si la pile de sortie **n'est pas vide** (cas simple), il suffit de dépiler son premier élément.
- Si la pile de sortie **est vide** (cas délicat), il faut alors commencer par retourner la pile d'entrée pour la mettre à la place de la pile de sortie. On peut réaliser cette opération intermédiaire en transférer un

- Une file réalisée ainsi est caractérisée par deux attributs entrée et sortie ;
- On peut donc définir une nouvelle version de la classe File utilisant ce principe.

- toute carte remise dans la réserve est ajoutée à l'autre pile (la défausse)
- S'ajoute un mécanisme liant les deux paquets : une fois la pioche vide on retourne la défausse pour en faire une nouvelle pioche, laissant à la place une défausse vide.
- Cette gestion des cartes correspond à une structure de file : une fois la pioche initiale vidée, les cartes seront piochées précisément dans l'ordre dans lequel elles ont été défaussées.
- La première défaussee sera la première piochée (FIFO).

```
>>> print(p.contenu.valeur)
2
```

```
>>> v = p.depiler()
>>> print(v)
3
>>> v = p.depiler()
>>> print(v)
2
>>> v = p.depiler()
>>> print(v)
1
>>> v = p.depiler()
Traceback (most recent call last):
IndexError: depiler sur une pile vide
"""

if self.est_vide():
    raise IndexError("depiler sur une pile vide")
return self.contenu.pop()

testmod()
```

0] : TestResults(failed=0, attempted=34)

6.8 – Réaliser une file avec deux piles

Une réalisation radicalement différente de cette même structure de file consiste à utiliser **deux piles**.

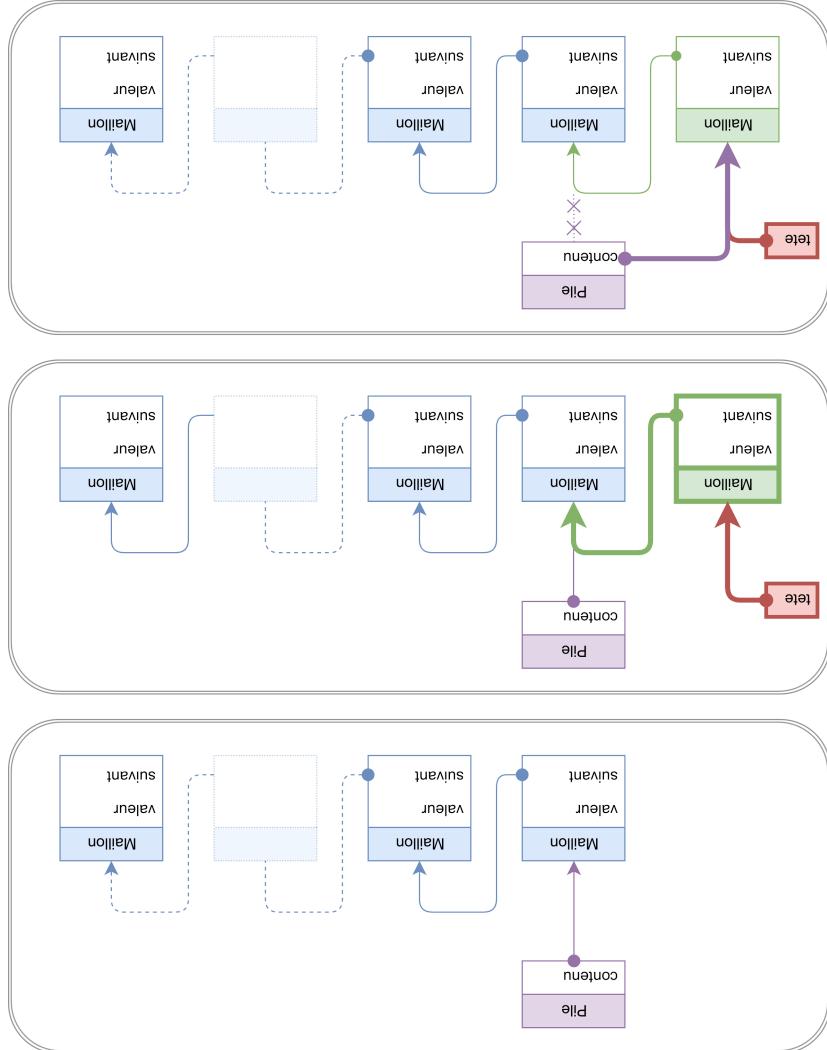
Exemple

Illustrons cette construction par un exemple.

On prend pour cela modèle sur un jeu de cartes où l'on disposerait d'une pioche, au sommet de laquelle on prend des cartes (disposées face cachée), et d'une défausse, au sommet de laquelle on en repose (disposées face visible).

Chacun de ces deux paquets de cartes est une **pile**, et ces deux paquets forment ensemble la réserve de cartes. On a ensuite la discipline suivante :

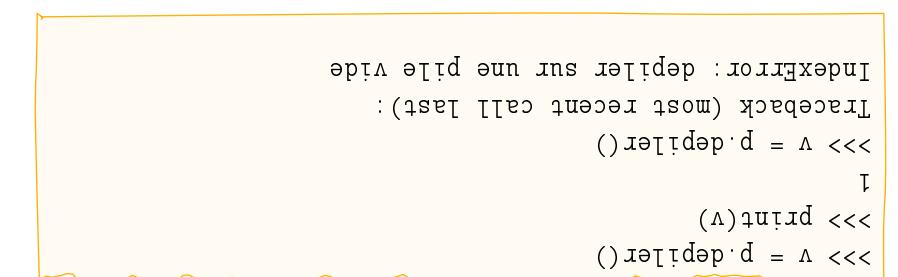
- toute carte prise dans la réserve est retirée dans l'une de ces piles (la pioche),



```
# La syntaxe classe_Pile(Ma_Classe) : est utilisée
# class en une seule fois, il faut écrire
# attention, lorsqu'on importe une
[4]: # attention, lorsqu'on importe une
```

```
[10]: class Pile:
    def __init__(self):
        self.contentu = []
    def est_vide(self):
        return self.contentu == []
    def printt(p):
        print(p.contentu)
    def examples_etc_tests():
        p = Pile()
        p.printt(p.est_vide())
        p = Pile()
        p.printt(p.est_vide())
        p = Pile()
        p.printt(p.est_vide())
        p = Pile()
        p.printt(p.est_vide())
        p = Pile()
        p.printt(p.est_vide())
    def __len__(self):
        return len(self.contentu)
    def empiler(self, valeur):
        self.contentu.append(valeur)
    def depiler(self):
        if len(self.contentu) > 0:
            return self.contentu.pop(0)
        else:
            raise IndexError("depiler sur une pile vide")
    def empiler(self, valeur):
        self.contentu.append(valeur)
    def examples_etc_tests():
        p = Pile()
        p.empiler(1)
        p.empiler(2)
        p.empiler(3)
        p.printt(p.contentu)
        p = Pile()
        p.empiler(1)
        p.empiler(2)
        p.empiler(3)
        p.printt(p.contentu)
        p = Pile()
        p.empiler(1)
        p.empiler(2)
        p.empiler(3)
        p.printt(p.contentu)
        p = Pile()
        p.empiler(1)
        p.empiler(2)
        p.empiler(3)
        p.printt(p.contentu)
```

IndexError: depiler sur une pile vide
Traceback (most recent call last):
1 >>> V = p.depiler()
2 >>> print(V)
3 >>> V = p.depiler()



```
# dans ce notebook (et dans les juges en lignes)
# pour étendre la classe existante Ma_Classe et lui
# ajouter de nouvelles méthodes.
#
# Ici, la classe Pile s'étend elle même !
class Pile(Pile):
    def empiler(self, valeur):
        """
        Empile valeur dans la pile courante.

        Args:
            valeur (T): valeur à empiler

        Exemple et tests:
        >>> p = Pile()
        >>> p.empiler(1)
        >>> assert not p.est_vide()
        >>> print(p.contenu.valeur)
        1
        >>> p.empiler(2)
        >>> print(p.contenu.valeur)
        2
        """
        tete_courante = self.contenu
        tete_nouvelle = Maillon(valeur, tete_courante)
        self.contenu = tete_nouvelle

        # version courte ;
        # self.contenu = Maillon(valeur, self.contenu)

    testmod()
```

4] : TestResults(failed=0, attempted=6)

ACTIVITÉ 4

Pour finir, implémenter `depiler` afin de récupérer la valeur au sommet de la pile.

Si la pile est vide, lever une exception indiquant : "IndexError: depiler sur une pile vide".

Sinon, il faut récupérer la valeur du premier maillon puis retirer ce maillon de la liste chaînée. Pour cela, le nouveau maillon de tête doit être le maillon suivant du maillon supprimé.

Enfin, après la mise à jour de la liste chaînée, il faut renvoyer la valeur qui

temps constant. Cette richesse des tableaux redimensionnables propre au langage Python peut donc donner une définition en apparence très simple à une autre version de la classe Pile.

Implémenter la classe Pile en utilisant la structure list pour y stocker les valeurs.

Exemples et tests :

```
>>> p = Pile()
>>> print(p.contenu)
[]

>>> print(p.est_vide())
True

>>> p.empiler(1)
>>> print(p.est_vide())
False
>>> print(p.contenu)
[1]
>>> p.empiler(2)
>>> p.empiler(3)
>>> print(p.contenu)
[1, 2, 3]

>>> v = p.depiler()
>>> print(v)
3
>>> v = p.depiler()
>>> print(v)
2
```

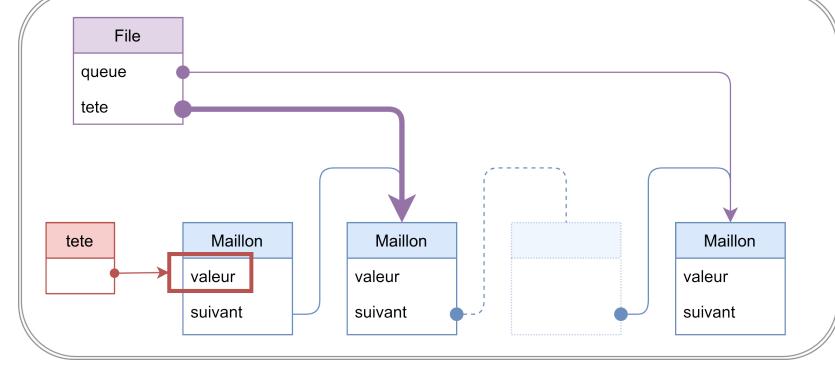
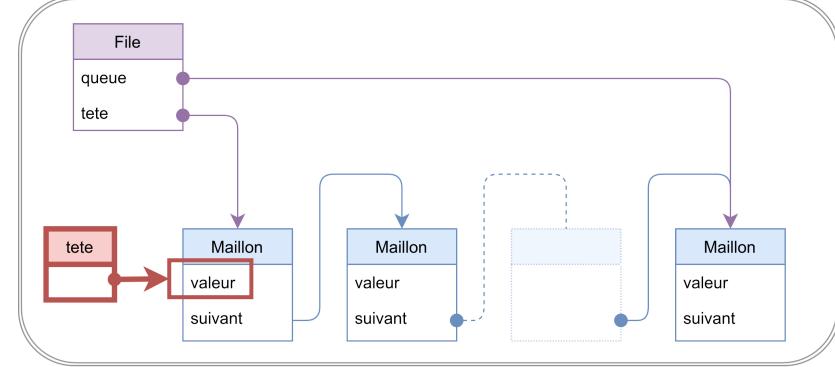
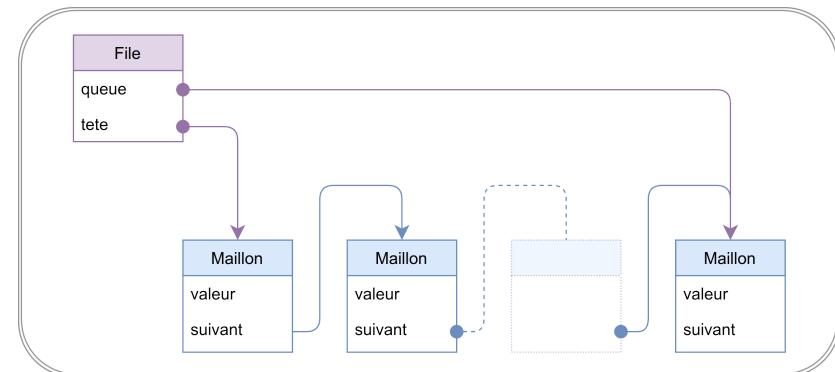
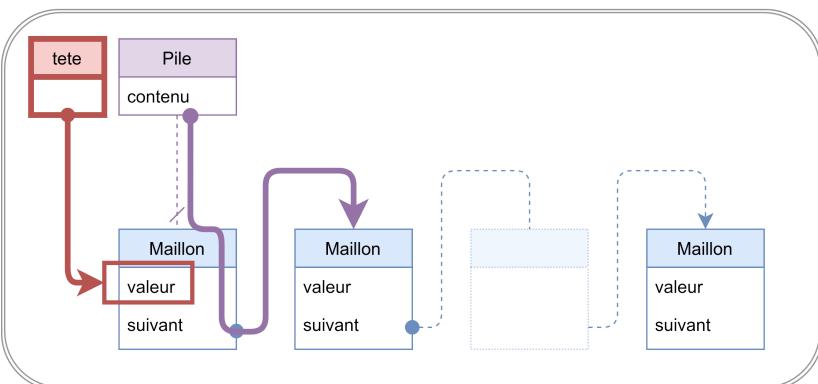
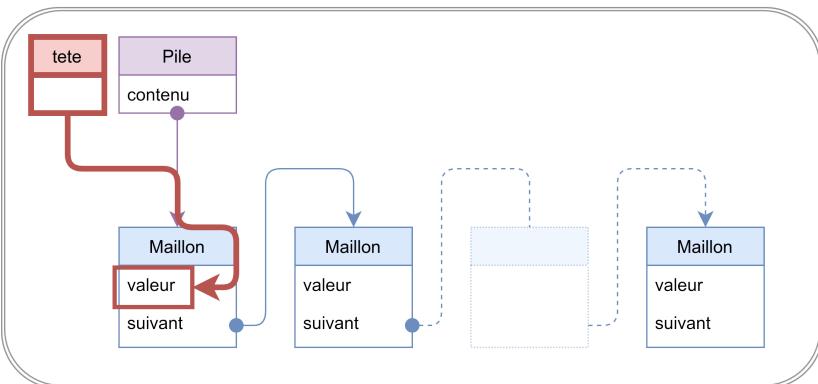
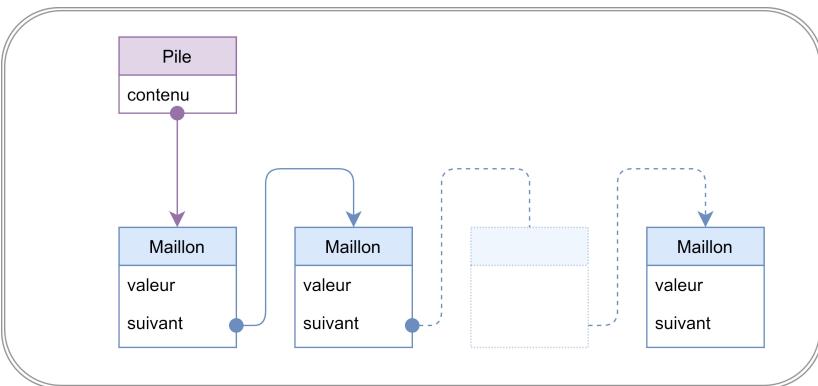
```
Example et tests :  
avait ete prelevee dans le mailloin de tete d'origine.  
    <>>> p = PiLe()  
    <>>> p.empile(1)  
    <>>> p.empile(2)  
    <>>> p.depile()  
    <>>> p.print()  
    2  
    <>>> v = p.depile()  
    <>>> p.print(v)  
    <>>> v = p.depile()  
    <>>> p.print()  
    1  
    <>>> v = p.depile()  
    <>>> p.print(v)  
    <>>> v = p.depile()  
    <>>> p.print(v)  
    Traceback (most recent call last):  
    Line 1:  depile sur une file vide
```

Les tableaux de Python réalisent également directement une structure de pile, avec leurs opérations append et pop qui s'exécutent en moyenne en

ACTIVITÉ 9

6.7 – Réalisation d'une pile avec les tableaux de Python

9] : TestResults(failed=0, attempted=20)



```

    print(f.defiler())
    f.entier(1)
    f.entier(2)
    f.entier(3)
    print(f.defiler())
    assert f.queue == 3
    print(f.defiler())
    f.defiler()
    assert f.queue == None
    print(f.defiler())
    f.defiler()
    Traceback (most recent call last):
      File <fichier sur une pile vide>, line 1
        raise IndexError("depiler sur une pile vide")
    IndexError: depiler sur une pile vide
    
```

La structure de liste chaînée donne également une manière de réaliser une file, à condition de considérer la variante des listes chaînées mutables.

6.6 – Réalisation d'une file avec une liste mutable

5] : TestResults(failed=0, attempted=8)

```

    tests()
    return valeur_tete

    self.__contenu = mailлон_sуivant
    mailлон_sуivant = tete.sуivant
    valeur_tete = tete.valeur
    tete = self.__contenu

    raise IndexError("depiler sur une pile vide")
  File <fichier sur une pile vide>, line 1
    if self.__est_vide():
      raise IndexError("depiler sur une pile vide")
    else:
      tete = self.__depiler()
      if self.__est_vide():
        raise IndexError("depiler sur une pile vide")
      else:
        tete = self.__depiler()
        if self.__est_vide():
          raise IndexError("depiler sur une pile vide")
        else:
          tete = self.__depiler()
          if self.__est_vide():
            raise IndexError("depiler sur une pile vide")
            print(f.defiler())
            f.defiler()
            assert f.tete == None
            print(f.defiler())
            f.defiler()
            Traceback (most recent call last):
              File <fichier sur une pile vide>, line 1
                raise IndexError("depiler sur une pile vide")
            IndexError: depiler sur une pile vide
            
```

En effet, on peut retirer l'élément de tête en retirant le maillon de tête. MAIS, l'ajout d'un nouvel élément à l'arrière de la file revient à ajouter un nouveau maillon en queue de liste. **Une mutation intervient** à cet endroit : alors que le maillon qui était le dernier de la liste chaînée avant l'ajout n'avait pas de suivant défini, il a comme suivant après l'ajout le nouveau maillon créé pour le nouvel élément.

Autre différence avec la structure de pile, il faut accéder *efficacement* au dernier maillon.

Pour cela, le plus intéressant est de conserver dans notre structure de donnée un attribut permettant d'accéder directement au dernier maillon.

ACTIVITÉ 5

On peut ainsi construire une classe `File` dont le constructeur définit deux attributs, l'un appelé `tete` et l'autre appelé `queue`, et désignant respectivement le premier maillon et le dernier maillon de la liste chaînée utilisée pour stocker les éléments.

Implémenter le constructeur de la classe `File` qui définit les deux attributs `tete` et `queue` et les initialise à `None`.

Exemples et tests :

```
>>> f = File()
>>> print(f.tete)
None
>>> print(f.queue)
None
```

```
[6]: class Maillon:
    """ Maillon d'une liste chaînée """
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class File:
    def __init__(self):
```

```
[8]: class File(File):
    def enfiler(self, valeur):
        """
        Ajoute la valeur à la fin de la file

        Args:
            valeur (T): valeur à ajouter

        Exemples et tests:
        >>> f = File()
        >>> f.enfiler(1)
        >>> print(f.tete.valeur)
        1
        >>> print(f.queue.valeur)
        1
        >>> f.enfiler(2)
        >>> f.enfiler(3)
        >>> print(f.tete.valeur)
        1
        >>> print(f.queue.valeur)
        3
        """
        nouveau = Maillon(valeur, None)

        if self.est_vide():
            self.tete = nouveau
        else:
            self.queue.suivant = nouveau
            self.queue = nouveau

    testmod()
```

8] : TestResults(failed=0, attempted=16)

Pour retirer un élément il s'agit de supprimer le premier maillon de la file, exactement comme il avait été fait lors de l'utilisation d'une liste chaînée pour réaliser une pile. Cependant, si le maillon retiré est le dernier, on veut également redéfinir l'attribut `self.queue` à `None`, afin de maintenir notre invariant qu'une file vide a ses deux attributs qui valent `None`.

ACTIVITÉ 8

Implémenter la méthode `defiler` qui retire le premier maillon de la file et renvoie la valeur de ce maillon.

Exemples et tests :

La file vide est caractérisée par le fait qu'elle ne contient aucun million.
En conséquence, sa tête et sa queue sont indéfinies. En outre, l'un comme l'autre ne peut valoir None que dans ce cas. Pour tester la vacuité de la file, il suffit donc de consulter l'un des attributs.

Implémenter la méthode `est_vide()` qui renvoie True si et seulement si l'attribut tête vaut None.

Exemples et tests :

```

if __name__ == "__main__":
    f = File()
    print(f.est_vide())
    f.append("un")
    print(f.est_vide())
    f.pop()
    print(f.est_vide())
    f.pop()
    print(f.est_vide())
    f.pop()
    print(f.est_vide())

```

ACTIVITÉ 6

6] : `TestResults(failed=0, attempted=1)`

```

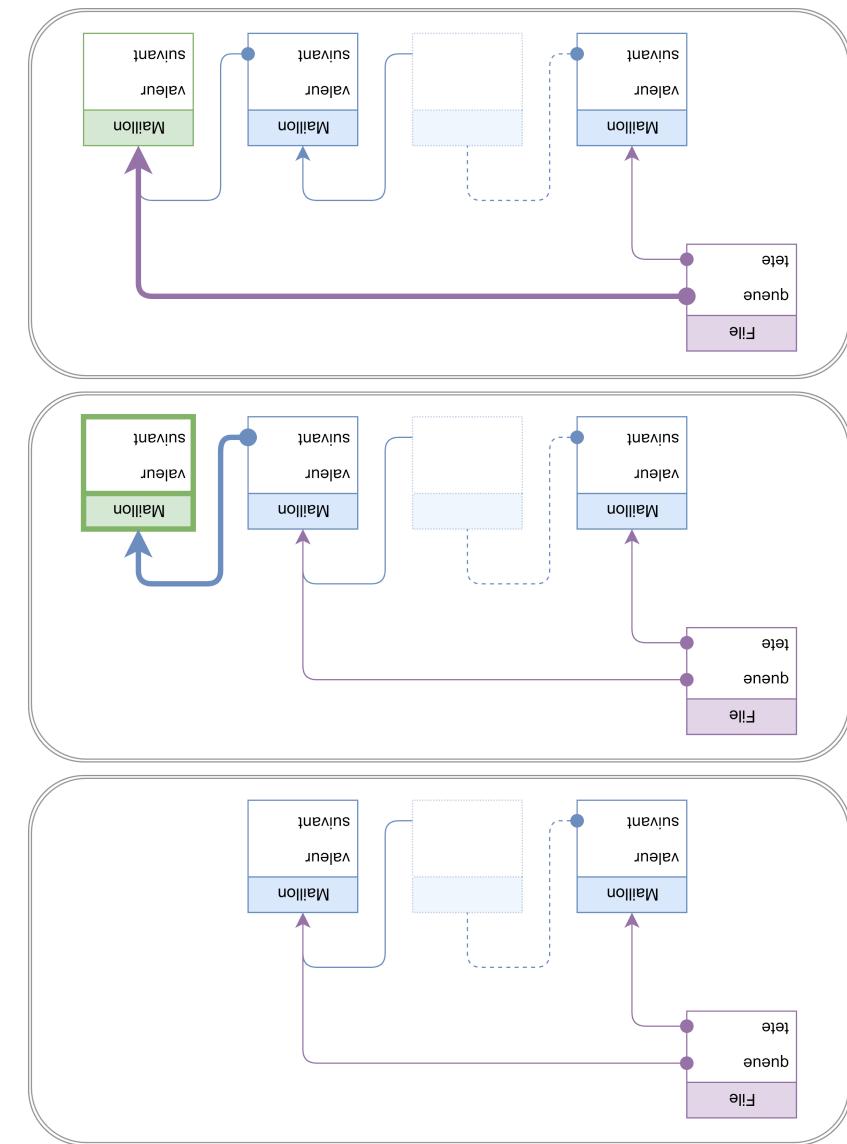
class File:
    def __init__(self):
        self.queue = None
        self.tete = None

    def append(self, valeur):
        if self.tete is None:
            self.tete = self.queue = Million(valeur)
        else:
            self.queue.suivant = Million(valeur)
            self.queue = self.queue.suivant

    def pop(self):
        if self.tete is None:
            raise IndexError("File est vide")
        valeur = self.tete.valeur
        self.tete = self.tete.suivant
        if self.tete is None:
            self.queue = None
        return valeur

    def est_vide(self):
        return self.tete is None

```



```

class File(File):
    def est_vide(self):
        """
        Est ce que la file est vide?
        Returns:
            bool: Truessi la file est vide
        Exemples et tests:
        >>> f = File()
        >>> print(f.est_vide())
        True
        >>> f.tete = Maillon(1, None)
        >>> print(f.est_vide())
        False
        """
        return self.tete is None

testmod()

```

7] : TestResults(failed=0, attempted=12)

L'ajout d'un nouvel élément à l'arrière de la file demande de créer un nouveau maillon. Ce maillon prend la dernière place, et n'a donc pas de maillon suivant.

Ce maillon est alors définie comme suivant le maillon de queue actuel.

On a cependant besoin de traiter le cas particulier où il n'existe pas de maillon de queue, qui correspond à une file initialement vide. Dans ce cas le nouveau maillon devient l'unique maillon de la file, et donc son maillon de tête.

Pour finir, dans tous les cas, notre nouveau maillon devient en outre le nouveau maillon de de queue de la file.

ACTIVITÉ 7

En suivant l'algorithme décrit ci-dessus, implémenter la méthode `enfiler` qui admet comme argument une valeur à ajouter en queue de file.

Exemples et tests :

```

>>> f = File()
>>> f.enfiler(1)
>>> print(f.tete.valeur)
1
>>> print(f.queue.valeur)
1
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> print(f.tete.valeur)
1
>>> print(f.queue.valeur)
3

```