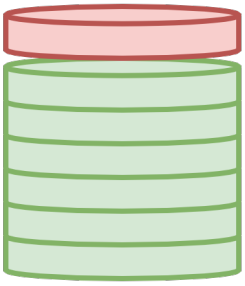
[3]:
```
from doctest import testmod
```

Chap. 6 — Piles et files

6.1 — Introduction

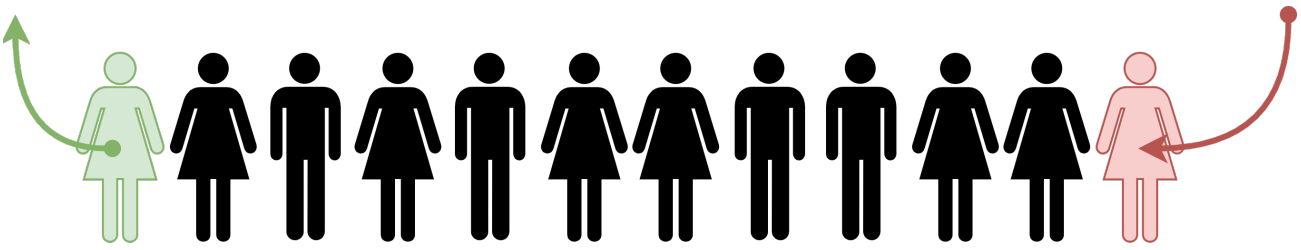
Tout comme les *tableaux*, **Pile** et **File** sont des structures de données qui permettent de (1) **stocker** des ensembles d'objets et (2) **ajouter**/**retirer** des objets un à un.

Dans une **pile** (en anglais *stack*), chaque opération `depile` retire l'élément arrivé le plus récemment. Pour imaginer cette structure, il suffit de penser à une *pile d'assiettes* : on ajoute une assiette sur le sommet et quant on retire une assiette, c'est forcément celle du sommet.



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

Dans une **file** (en anglais *queue*), chaque opération `defile` retire l'élément *qui avait été ajouté* le premier. Pour imaginer cette structure, on pense à une file d'attente dans laquelle (1) les personnes arrivent à tour de rôle, (2) patientent et (3) sont servies dans leur ordre d'arrivé.



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

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.

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.

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

6.3 — Interface et utilisation d'une pile

Détaillons l'interface des piles.

interface	explications 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 booléen indiquant si la pile est vide ou pas
`creer_pile() -> Pile[T]`	aucun paramètre et renvoie une pile vide capable de contenir n'importe quel type d'élément `T`
`empiler(p: Pile[T], e: T) -> None`	ajout de l'élément `e` au sommet de la pile `p` (*push* en anglais), fonction qui prend en paramètre une pile `p` et l'élément `e` de type `T` homogène avec celui des éléments de la pile
`depiler(p: Pile[T]) -> T`	retrait de l'élément au sommet de la pile `p` (*pop* 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

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

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.

6.5 — Réalisation d'une pile avec une liste chaînée

La structure de **liste chaînée** donne une manière élémentaire de réaliser une pile. Empiler un nouvel élément revient à ajouter un nouveau maillon en tête de liste, tandis que dépiler un élément revient à supprimer le maillon de tête.

On peut ainsi construire une classe `Pile` définie par un unique attribut contenu associé à l'ensemble des éléments de la pile, stockés sous la forme d'une liste chaînée.

Implémenter le constructeur de la classe `Pile` qui construit une pile vide en définissant son attribut `contenu` comme la liste vide `None`.

Exemple et test :

```
>>> p = Pile()
>>> print(p.contenu)
None
```

[ ]:
```python
from doctest import testmod


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()
```

Étendre 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
```

[ ]:
```python
# 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 !

class Pile(Pile):
    def est_vide(self) -> bool:
        """
        Est ce que la pile est vide ?

        Returns:
            bool: True si et seulement si la pile est vide

        Exemple et test:
        >>> p = Pile()
        >>> print(p.est_vide())
        True
        >>> p.contenu = Maillon(1, None)
        >>> print(p.est_vide())
        False
        """
        return self.contenu is None


testmod()
```

Implémenter la méthode `empile` dans la classe `Pile`. Pour cela construire une nouvelle liste chaînée dont le premier maillon contient :

- valeur : la valeur à empiler
- suivant : le premier maillon 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 = Pile()
>>> p.empiler(1)
>>> assert not p.est_vide()
>>> print(p.contenu.valeur)
1
>>> p.empiler(2)
>>> print(p.contenu.valeur)
2
```

[ ]:
```python
# 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 !

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()
```

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 avait été prélevée dans le maillon de tête d'origine.

Exemple et tests :

```
>>> p = Pile()
>>> p.empiler(1)
>>> p.empiler(2)
>>> 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
```

[ ]:
```python
# 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 !


class Pile(Pile):
    def depiler(self):
        """
        Dépile la valeur de tête de la pile.

        Raises:
            IndexError: si la pile est vide

        Returns:
            T: valeur de tête de la pile

        Exemple et tests:
```

```
        >>> p = Pile()
        >>> p.empiler(1)
        >>> p.empiler(2)
        >>> 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")

        tete = self.contenu
        valeur_tete = tete.valeur
        maillon_suivant = tete.suivant
        self.contenu = maillon_suivant

        return valeur_tete


testmod()
```

6.6 — Réalisation d'une file avec une liste mutable

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

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 définie, 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.

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
```

[4]:
```
class Maillon:
    """ Maillon d'une liste chaînée """
    def __init__(self, valeur, suivant):
        self.valeur  = valeur
        self.suivant = suivant

class File:
    def __init__(self):
        """
        Exemples et tests:
        >>> f = File()
        >>> print(f.tete)
        None
        >>> print(f.queue)
        None
        """
        self.tete  = None
        self.queue = None


testmod()
```

4]: `TestResults(failed=0, attempted=3)`

La file vide est caractérisée par le fait qu'elle ne contient aucun maillon. 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 deux attributs.

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

Exemples et tests :

```
    >>> f = File()
    >>> print(f.est_vide())
    True
    >>> f.tete = Maillon(1, None)
    >>> print(f.est_vide())
    False
```

[5]:
```
# attention, lorsqu'on implémente une
# classe en une seule fois, il faut écrire
# `class File:`.
# La syntaxe `class File(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 File s'étend elle même !

class File(File):

    def est_vide(self):
        """
        Est ce que la file est vide?

        Returns:
            bool: True ssi 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()
```

5]: `TestResults(failed=0, attempted=4)`

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.

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
```

[6]:
```python
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()
```

6]: `TestResults(failed=0, attempted=8)`

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`.

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

Exemples et tests :

```
>>> f = File()
>>> f.enfiler(1)
>>> f.enfiler(2)
>>> f.enfiler(3)
>>> print( f.defiler() )
1
>>> assert f.queue.valeur == 3
>>> print( f.defiler() )
2
>>> assert f.queue.valeur == 3
>>> print( f.defiler() )
3
>>> assert f.tete  == None
>>> assert f.queue == None
```

```
    >>> f.defiler()
    Traceback (most recent call last):
    IndexError: defiler sur une file vide
```

[8]:
```python
class File(File):
    def defiler(self):
        """
        Supprime et renvoie le premier élément de la file

        Raises:
            IndexError: si la file est vide

        Returns:
            T: valeur du premier élément de la file

        Exemples et tests:
        >>> f = File()
        >>> f.enfiler(1)
        >>> f.enfiler(2)
        >>> f.enfiler(3)
        >>> print( f.defiler() )
        1
        >>> assert f.queue.valeur == 3
        >>> print( f.defiler() )
        2
        >>> assert f.queue.valeur == 3
        >>> print( f.defiler() )
        3
        >>> assert f.tete  == None
        >>> assert f.queue == None
        >>> f.defiler()
        Traceback (most recent call last):
        IndexError: defiler sur une file vide
        """
        if self.est_vide():
            raise IndexError("defiler sur une file vide")

        valeur = self.tete.valeur
        self.tete = self.tete.suivant

        if self.tete is None:
            self.queue = None

        return valeur


testmod()
```

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

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

Les tableaux de Python réalisent également directement une structure de pile, avec leurs opérations `append` et `pop` qui s'exécutent en moyenne en 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
>>> v = p.depiler()
```

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

[ ]:
```python
class Pile:
    """
    Classe Pile
    (implémentée avec les tableaux Python)
    """
    def __init__(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> print(p.contenu)
        []
        """
        self.contenu = []

    def est_vide(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> print(p.est_vide())
        True
        """
        return self.contenu == []

    def empiler(self, valeur):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> p.empiler(1)
        >>> print(p.est_vide())
        False
        >>> print(p.contenu)
        [1]
        >>> p.empiler(2)
        >>> p.empiler(3)
        >>> print(p.contenu)
        [1, 2, 3]
        """
        self.contenu.append(valeur)

    def depiler(self):
        """
        Exemples et tests :
        >>> p = Pile()
        >>> p.empiler(1)
        >>> p.empiler(2)
        >>> p.empiler(3)
        >>> 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()
```