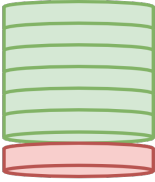


Chap. 6 – Piles et files

6.1 – Introduction

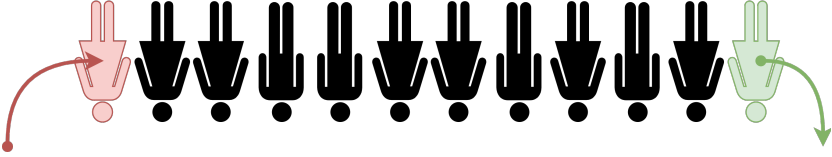
Tout comme les *tableaux*, **Pile** et **File** sont des structures de données qui permettent de (1) **stocker** des ensembles d'objets et (2) **ajouter/retirer** des objets un à un.

Dans une **pile** (en anglais *stack*), chaque opération de retrait s'effectue sur 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 de retrait s'effectue sur 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'arrivée.



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 préconise 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 <code>int</code> pour les nombres entiers ou encore <code>str</code> pour les chaînes de caractères



```
>>> 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()

6.6 – Réalisation d'une file avec deux piles

interface

explications et commentaires

est_vide(p: Pile[T]) -> bool

prend en paramètre une pile p et renvoie un booléen 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

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

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

empiler(p: Pile[T], e: T) -> T

depiler(p: Pile[T]) -> T

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.

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

```
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.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)
        >>> print(p.est_vide())
        False
        >>> print(p.contenu)
        [1]
        >>> p.empiler(2)
        >>> p.empiler(3)
        >>> print(p.contenu)
        [1, 2, 3]
        """
        u = self.contenu.pop()
        >>> print(u)
        3
        >>> u = p.depiler()
        >>> print(u)
        3
        >>> u = p.depiler()
        >>> print(u)
        2
        >>> u = p.depiler()
        >>> print(u)
        1
        >>> p = Pile()
        >>> print(u)
        2
        """
```



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>

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 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)
```

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
```

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

```
[ ] : class Pile(Pile):
    def depiler(self):
        """
        Dépile la valeur de tête de la pile.
        Raises:
            IndexError: si la pile est vide
        Returns:
            l: valeur de tête de la pile
        Exemple et tests:
        >>> p = Pile()
        >>> p.empler(1)
        >>> p.empler(2)
        >>> v = p.depiler()
        2
        >>> print(v)
        2
        >>> v = p.depiler()
        1
        >>> 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
        maillon_suivant = tete.valeur
        self.contenu = maillon_suivant
        return valeur_tete

testmod()
```

6.6 – Réalisation d'une file avec une liste (mutable)

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

Les tableaux de Python réalisent également 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 Py-



```
>>> print(p.est_vide())
```

```
False
```

```
[ ]: 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 `empiler` 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
```

```
[ ]: 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()
```