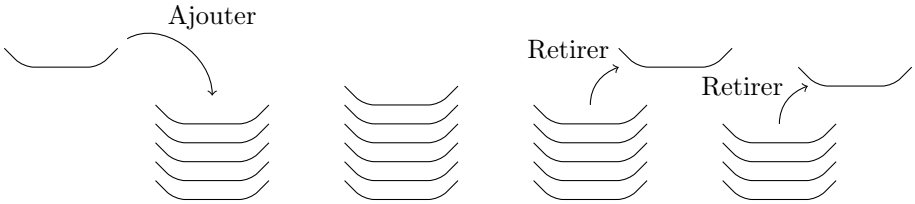
Piles et Files

1 Les piles

1.1 Interface

Une *pile* est une structure de donnée qui permet de stocker des éléments de même type. On l'associe souvent à l'image d'une pile d'assiettes ; chaque nouvelle assiette est ajoutée au-dessus des précédentes, et l'assiette retirée est systématiquement celle du sommet. On qualifie se comportement de «dernier entré, premier sorti» ou encore LIFO (Last In, First Out).

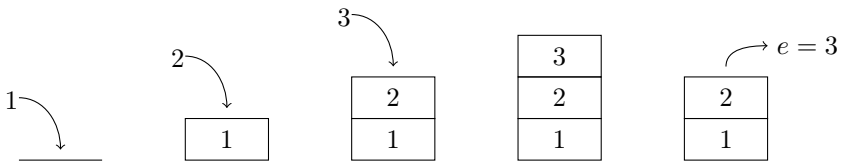


L'opération d'ajout d'un élément au sommet d'une pile est tradionnellement appelée `empiler` (ou `push` en anglais). L'opération de retrait de l'élément au sommet de la pile est appelé `dépiler` (ou `pop` en anglais).
Le deux autres opérations élémentaires sur les piles sont `creer_pile` qui créée une pile vide et `est_vide` qui test si la pile est vide.
L'interface d'une pile est alors la suivante :

fonction	description
`creer_pile()`	crée et renvoie une pile vide
`est_vide(p)`	renvoie `True` si la pile est vide et `False` sinon
`empiler(p, e)`	ajoute `e` au sommet de `p`
`depiler(p)`	retire et renvoie l'élément au sommet de `p` si `p` n'est pas vide, et lève une exception sinon

Si par exemple on exécute les instructions de gauche, on obtient la situation représentée à droite :

```
pile = creer_pile()
empiler(pile, 1)
empiler(pile, 2)
empiler(pile, 3)
e = depiler(pile)
```



Utilisation d'une pile : bouton retour en arrière
Considérons un navigateur web pour lequel on s'intéresse à deux opérations : aller à une nouvelle page et revenir à la page précédente.

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 quitté, le comportement est de la forme « dernier entré, premier sorti », les adresses précédentes peuvent donc être stockées dans une pile.

```
adresse_courante = ""
adresses_precedentes = creer_pile()
```

Lorsqu'on navige vers une nouvelle page, l'adresse courante est ajouté au sommet de la pile, l'adresse cible devient alors la nouvelle adresse courante.

```
def aller_a(adresse_cible):
    empiler(adresses_precedentes, adresse_courante)
    adresse_courante = adresse_cible
```

Pour revenir en arrière, il suffit de revenir à l'adresse au sommet de la pile.

```
def retour():
    if not est_vide(adresses_precedentes):
        adresse_courante = depiler(adresses_precedentes)
```

1.2 A l'aide d'un tableau Python

Les tableaux Python réalisent directement une structure de pile avec leurs opérations `append` et `pop`.

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

La pile vide est alors définie par un `contenu` correspondant au tableau vide.

```
class Pile:
    '''structure de pile'''
    def __init__(self):
        self.contenu = []
```

On peut ainsi tester si la pile est vide en regardant la taille de son `contenu`.

```
    def est_vide(self):
        return len(self.contenu) == 0
```

Les opérations `append` et `pop` réalisent respectivement les opérations `empiler` et `depiler`.

```
    def empiler(self, e):
        self.contenu.append(e)

    def depiler(self):
        if self.est_vide():
            raise IndexError('depiler sur une pile vide')
        else:
            return self.contenu.pop()
```

On résume l'ensemble de la classe dans le programme ci-dessous.

```
1  class Pile:
2      '''structure de pile'''
3      def __init__(self):
4          self.contenu = []
5
6      def est_vide(self):
7          return len(self.contenu) == 0
8
9      def empiler(self, e):
10         self.contenu.append(e)
11
12     def depiler(self):
13         if self.est_vide():
14             raise IndexError('depiler sur une pile vide')
15         else:
16             return self.contenu.pop()
```

1.3 A l'aide d'une liste chaînée

La méthode précédente est raisonnable dans le cadre d'un programme Python, les opérations `append` et `pop` s'éxécutant en moyenne en temps constant. Cette solution ne s'exporte cependant pas à n'importe quel autre langage.

La structure de liste chaînée donne une manière élémentaire de réaliser une pile. Empiler un élément revient à ajouter un élément en tête et dépiler un élément revient à supprimer l'élément de tête.

L'attribut `contenu` est alors initialiser à la liste vide, c'est à dire `None`.

```
class Pile:
    '''structure de pile'''
    def __init__(self):
        self.contenu = None
```

La pile est vide si son contenu est vide.

```
    def est_vide(self):
        return self.contenu is None
```

Empiler un élement est ajouter un élément en tête de la liste chaînée `contenu`.

```
    def empiler(self, e):
        self.contenu = Cellule(e,self.contenu)
```

Dépiler revient à supprimer l'élément en tête de la liste chaînée `contenu` si celle-ci n'est pas vide.

```
    def depiler(self):
        if self.est_vide():
            raise IndexError('depiler sur une pile vide')
        else:
            v = self.contenu.valeur
            self.contenu = self.contenu.suivante
            return v
```

On résume l'ensemble de la classe dans le programme ci-dessous.

```
1  class Cellule:
2      '''une cellule d une liste chainee'''
3      def __init__(self, v, s):
4          self.valeur = v
5          self.suivante = s
6
7  class Pile:
8      '''structure de pile'''
9      def __init__(self):
10         self.contenu = None
11
12     def est_vide(self):
13         return self.contenu is None
14
15     def empiler(self, e):
16         self.contenu = Cellule(e,self.contenu)
17
18     def depiler(self):
19         if self.est_vide():
20             raise IndexError('depiler sur une pile vide')
21         else:
```

```
22            v = self.contenu.valeur
23            self.contenu = self.contenu.suivante
24            return v
```

2 Les files

2.1 Interface

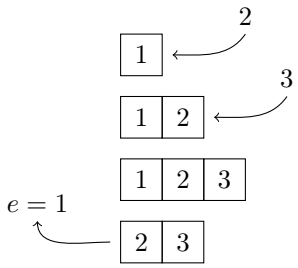
Une *file* est une structure de donnée qui permet de stocker des éléments de même type. On l'associe souvent à l'image d'une file d'attente ; chaque nouvelle personne arrive en fin de file, et la personne suivante a être servi est celle en tête de fil. On qualifie se comportement de «premier entré, premier sorti» ou encore FIFO (Firt In, First Out).

Les deux opérations élémentaires que l'on a besoin pour réaliser cete structure est ajouter un élément en fin de file et retirer l'élément en tête de file. Comme pour les piles, il faut également pouvoir créer une file vide et tester si une file est vide. On obtient alors l'interface suivante.

fonction	description
`creer_file()`	crée et renvoie une file vide
`est_vide(f)`	renvoie `True` si la file est vide et `False` sinon
`ajouter(f, e)`	ajoute `e` à l'arrière de `f`
`retirer(p)`	retire et renvoie l'élément à l'avant de `f` si `f` n'est pas vide, et lève une exception sinon

Si par exemple on exécute les instructions de gauche, on obtient la situation représentée à droite :

```
f = creer_file()
ajouter(f, 1)
ajouter(f, 2)
ajouter(f, 3)
e = retirer(f)
```



2.2 A l'aide d'une liste chaînée

La structure de liste chaînée donne également une manière de réaliser une file. La différence avec la réalisation d'une pile est de pouvoir accéder à la dernière cellule de la liste. Il est alors intéressant de conserver dans notre structure un attribut permettant d'accéder directement à cette dernière cellule.
On peut construire une classe `File` avec deux attributs, l'un appelé `tete` et l'autre appelé `queue`, et désignant respectivement la première cellule et la dernière cellule de la liste chaînée.

```
class File:
    '''structure de file'''
    def __init__(self):
        self.tete = None
        self.queue = None
```

Pour tester si la file est vide, il suffit de tester si un de ces deux attributs est `None`.

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

Pour ajouter un élément en queue de file, il faut créer une nouvelle cellule qui n'a pas de suivant.

```
    def ajouter(self, e):
        c = Cellule(e, None)
```

Cette cellule devient alors la suivante de la queue actuelle et devient la nouvelle queue.

```
        c = Cellule(e, None)
        self.queue.suivante = c
        self.queue = c
```

On a cependant le besoin de traîter le cas particulier où la file est vide et dans ce cas la cellule devient l'unique cellule de la liste et donc celle de tête.

```
    def ajouter(self, e):
        c = Cellule(e, None)
        if self.est_vide():
            self.tete = c
        else:
            self.queue.suivante = c
        self.queue = c
```

Finalement pour retirer un élément, on procède de la même manière que pour une pile, il suffit de penser à redéfinir l'attribut `queue` à `None` lorsque l'opération vide la file.
On obtient alors le programme suivant.

```
class File:
    '''structure de file'''
    def __init__(self):
        self.tete = None
        self.queue = None

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

    def ajouter(self, e):
        c = Cellule(e, None)
        if self.est_vide():
            self.tete = c
        else:
            self.queue.suivante = c
        self.queue = c

    def retirer(self):
        if self.est_vide():
            raise IndexError('retirer sur une file vide')
        else:
            v = self.tete.valeur
            self.tete = self.tete.suivante
            if self.tete is None:
                self.queue = None
            return v
```