4
Piles

1. Un type de données est **abstrait** lorsqu'il est défini par la manière de s'en servir : on sait seulement quelles sont les **valeurs** possibles et les **opérations** possibles avec ce type de données.
On impose parfois aussi des limitations sur la **complexité en temps** (nombre de calculs à effectuer pour réaliser une opération) et sur la **complexité en espace** (quantité de mémoire occupée pour représenter une valeur en machine).

2. La notion de *type abstrait de données* s'oppose ainsi à celle de **structure de données**, qui décrit la représentation effective de données au sein d'un langage de programmation.
On résume cette opposition en présentant le *type abstrait de données* comme le point de vue de l'utilisateur et la *structure de données* comme le point de vue de l'implémenteur.

I
Notion de pile

3. Valeurs
La notion de **pile** est *type de données abstrait* qui représente un ensemble de données de types quelconque. Les données d'une pile n'ont pas nécessairement toutes le même type.

4. Création
On peut créer une pile de deux manières.
4.1 On peut d'abord créer une pile vide, puis ajouter des éléments un par un à cette pile.
4.2 On peut aussi constituer en pile les éléments d'une liste comme si on avait créé une pile vide puis ajouté, un par un et dans l'ordre de la liste, les éléments de la liste à la pile.

Autres opérations

5. État de la pile
On dispose de fonctions permettant de vérifier si une pile est *vide* ou non et de calculer la *hauteur* d'une pile, c'est-à-dire le nombre d'éléments qui la composent.

6. Modifications de la pile
Les opérations qui modifient une pile expliquent le nom de pile.
6.1 Les entrées qu'on ajoute et qu'on enlève à une pile suivent le principe

Dernier arrivé, premier servi

qui est le principe à l'œuvre pour faire la vaisselle : la dernière assiette posée sur la pile sera aussi la première assiette lavée.
6.2 En pratique, on dispose d'une fonction pour ajouter un élément supplémentaire au sommet de la pile ; d'une fonction pour connaître la valeur de l'élément au sommet de la pile (c'est-à-dire le dernier élément ajouté) et d'une fonction pour retirer de la pile l'élément situé au sommet.
Cette dernière fonction doit retourner la valeur de l'élément retiré.

II

Exemple d'implémentation

7. On suppose que le code pour implémenter la notion de pile est écrit dans le fichier `piles.py`, fichier placé dans le répertoire `pyzo201x/lib/python3.y` (je ne suis pas sûr que ce soit une bonne chose de procéder ainsi). Pour utiliser ce module, il faut l'importer comme n'importe quel autre module.

```
import piles as pl
```

8. Création
On crée une pile avec les éléments d'une liste `L` avec `pl.Pile(L)`. En particulier, on crée une pile vide avec `pl.Pile([])`.

9. Opérations
On dispose d'une pile `p`.

9.1 État
La fonction `isVoid()` retourne un booléen, égal à `True` si, et seulement si, la pile `p` est vide.
Le nombre d'éléments de la pile est donné par `p.getHeight()`.
La valeur de l'élément situé au sommet de la pile est donnée par `p.getTop()`.

9.2 Modifications : empiler et dépiler
On pose un élément `x` au sommet de la pile `p` avec `p.push(x)`.
L'instruction `p.pop()` enlève et retourne l'élément placé au sommet de la pile `p`.

9.3 Affichage
Pour afficher le contenu d'une pile, il faut la convertir en chaîne de caractères. La commande `p.str()` crée une chaîne de caractères qui contient les éléments de la liste `p` séparés par des points. On peut modifier le séparateur et, par exemple, remplacer les points par des points-virgules au moyen de la commande `p.str(';')`.
L'élément situé au sommet de la pile est alors le dernier caractère de la chaîne de caractères.

III

Exemples d'utilisation

Q 1. *Les deux fonctions suivantes modifient le contenu d'une pile et ne retournent rien (c'est-à-dire qu'elles retournent* `None` *toutes les deux).*

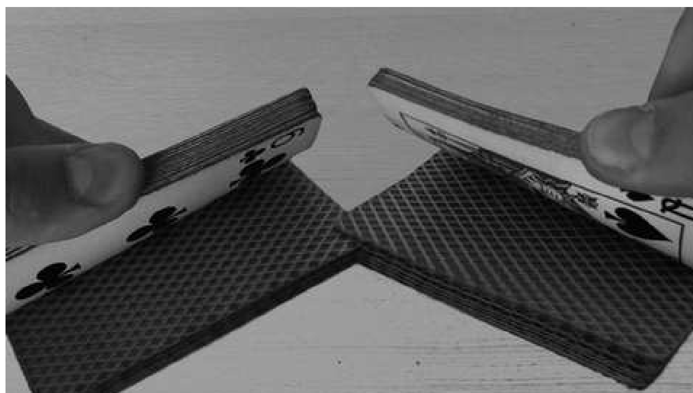
Q 1.a *Écrire une fonction* `dup` *qui duplique l'élément situé au sommet d'une pile non vide. Cette fonction transforme par exemple la pile* `'1.2.3.4.2.1.3'` *en* `'1.2.3.4.2.1.3.3'`.

Q 1.b *Écrire une fonction* `swap` *qui permute les deux éléments situés au sommet d'une pile de hauteur supérieure à 2. Cette fonction transforme par exemple la pile* `'1.2.3.4.2.1.3'` *en* `'1.2.3.4.2.3.1'`.

Q 2. *Par convention, le sommet de la pile est le 0-ième élément et le dernier élément d'une pile de hauteur h est le $(h-1)$-ième élément de cette pile.*
Écrire une fonction d'argument n qui donne la valeur du n-ième élément d'une pile. Si n est supérieur ou égal à la hauteur de la pile, la fonction devra retourner `None`*. On utilisera une pile auxiliaire.*

Q 3. *Écrire une fonction* `permuter` *qui enlève l'élément situé au sommet de la pile et le place tout en bas de la pile (ce qui réalise une permutation circulaire des éléments de la pile). On utilisera une pile auxiliaire.*

Q 4.a *On dispose d'un jeu de cartes : on le coupe en deux tas et on mélange ces deux tas comme dans les films — un tas dans chaque main et on enchevêtre les cartes des deux tas.*



Comment réaliser une opération analogue avec les éléments de deux piles ?

Q 4.b *On dispose de deux piles triées par ordre croissant (la valeur au sommet de chaque pile est la plus grande valeur de la pile). Comment fondre ces deux piles en une seule pile triée par ordre croissant ?*

Q 5. *Écrire une fonction prenant comme argument une chaîne de caractères* `expr` *et retournant un booléen, égal à* `True` *si, et seulement si, les éventuelles parenthèses figurant dans cette chaîne de caractères sont convenablement placées.*

10. Notation polonaise inversée

La **notation polonaise inversée** permet de formuler des expressions algébriques sans faire usage de parenthèses :

- Par appui sur la touche *Entrée* [$\hookleftarrow$], les opérandes sont placés les uns à la suite des autres dans une pile ;
- L'application d'une opération dépile un certain nombre d'opérandes (un seul opérande dans le cas du changement de signe, de l'application d'une fonction telle que cos, exp, $\sqrt{\cdot}$... deux opérandes dans le cas de l'addition, de la multiplication, de la soustraction, de la division...) et empile le résultat calculé.

Exemples de calculs algébriques

Expression	Notation polonaise inversée
$2+3$	$2 \hookleftarrow 3 \hookleftarrow +$
$5-2$	$5 \hookleftarrow 2 \hookleftarrow -$
$5/2$	$5 \hookleftarrow 2 \hookleftarrow /$
$(2-5)\times 7$	$2 \hookleftarrow 5 \hookleftarrow -7 \hookleftarrow *$
$(5+2)/(4-3)$	$5 \hookleftarrow 2 \hookleftarrow +4 \hookleftarrow 3 \hookleftarrow -\ /$
$((5+2)-4)/3$	$5 \hookleftarrow 2 \hookleftarrow +4 \hookleftarrow -\ 3 \hookleftarrow /$

Q 6. *Écrire une fonction* `npi(args)` *prenant pour arguments une liste* `args` *contenant les opérandes et les opérations dans l'ordre où ils sont saisis en notation polonaise inverse et qui retourne le résultat. Ainsi,*

```
npi([5,2,7,-10,'-','*','/'])
```

devra interpréter son argument comme l'opération

$$\frac{5}{2\times(7-(-10))} = \frac{5}{34}$$

et retourner 0.1470588235...

On se limitera aux quatre opérations algébriques : $+$, $-$, $\times$ *et* $/$.

Réponses aux questions

III Exemples d'utilisation

R 1.a

```
def dup(p):
  if p.getHeight()>0:  # si la pile n'est pas vide
    x = p.getTop()     # on récupère (sans dépiler) la valeur du sommet
    p.push(x)          # et on l'empile
  return None
```

R 1.b Partant de la pile $p = p_0 p_1 \cdots p_{N-1} p_N$, on dépile la valeur au sommet qu'on affecte à la variable x, après quoi

$$p = p_0 p_1 \cdots p_{N-1} \quad \text{et} \quad x = p_N.$$

On dépile la nouvelle valeur au sommet qu'on affecte à la variable y, après quoi

$$p = p_0 p_1 \cdots p_{N-2}, \quad x = p_N \quad \text{et} \quad y = p_{N-1}.$$

On empile alors la valeur de x pour obtenir

$$p = p_0 p_1 \cdots p_{N-2} p_N$$

puis on empile la valeur de y pour obtenir

$$p = p_0 p_1 \cdots p_{N-2} p_N p_{N-1}.$$

On a bien permuté les deux valeurs au sommet de la pile.

```
def swap(p):
  if p.getHeight()>1: # s'il y a au moins 2 éléments dans la pile
    x = p.pop()
    y = p.pop()
    p.push(x)
    p.push(y)
  return None
```

R 2. On ne peut accéder directement qu'à la valeur située au sommet d'une pile. Pour accéder à la n-ième valeur, il faut donc commencer par dépiler les n valeurs qui sont situés au-dessus d'elle dans la pile. On passe donc de

$$p = p_0 p_1 \cdots p_{N-1} p_N \qquad q = \varnothing$$

à

$$p = p_0 p_1 \cdots p_{N-n} \qquad q = p_N p_{N-1} \cdots p_{N-(n-1)} \qquad x = p_{N-n}$$

Une fois la valeur connue et affectée à x, il faut rempiler ces n valeurs pour remettre la pile p dans l'état où on l'a trouvée en arrivant.

```
def n_ieme_element(p, n):
  if (n<p.getHeight()): # s'il y a au moins n éléments dans la pile p
    q = pl.Pile([])
    # on dépile les n éléments au sommet de p
    # pour les empiler dans une pile q
    for i in range(n):
      q.push(p.pop())
    # on récupère (sans dépiler) la valeur qui apparaît alors au sommet de p
    x = p.getTop()
    # on empile les éléments de q dans la pile p
    for i in range(n):
      p.push(q.pop())
    return x
  else:
    return None
```

R 3. On commence par dépiler la valeur située au sommet de p, qu'on affecte à la variable s. On dépile ensuite les autres éléments de p, qu'on empile au fur et à mesure dans une pile auxiliaire q. On passe donc de

$$p = p_0 p_1 \cdots p_{N-1} p_N \qquad q = \varnothing$$

à

$$p = \varnothing \qquad s = p_N \qquad q = p_{N-1} \cdots p_1 p_0.$$

Lorsque p est vide, on empile d'abord la valeur de s, puis on dépile les éléments de q pour les empiler dans p. On arrive ainsi à

$$p = p_N p_0 p_1 \cdots p_{N-1} \qquad q = \varnothing.$$

```
def permuter(p):
  s = p.pop()
  q = pl.Pile([])
  while (p.getHeight()>0):
    q.push(p.pop())
  p.push(s)
  while (q.getHeight()>0):
    p.push(q.pop())
  return None
```

R 4.a À chaque étape, on choisit aléatoirement le tas dans lequel on va prendre une carte.

```
def choix():
  from random import randint
  return randint(0,1)

def battage(p, q):
  r = pl.Pile([])
  # Tant qu'aucune pile n'est vide,
  while ((p.getHeight()>0) and (q.getHeight()>0)):
  # on en choisit une au hasard pour dépiler.
    if choix():
      r.push(p.pop())
    else:
      r.push(q.pop())
  # L'une des deux piles est maintenant vide.
  # On vide la pile p (si ce n'est pas déjà fait).
  while (p.getHeight()>0):
    r.push(p.pop())
  # On vide la pile q (si ce n'est pas déjà fait).
  while (q.getHeight()>0):
    r.push(q.pop())
  # On remet les valeurs à l'endroit
  R = pl.Pile([])
  while not(r.isVoid()):
    R.push(r.pop())
  return R
```

Il faut penser à remettre les valeurs à l'endroit avant de retourner la pile ainsi créée. La structure de pile fait que les valeurs situées en haut des piles p et q sont les premières à être dépilées pour être empilées dans r. De ce fait, ces valeurs se retrouvent *au fond* de la pile r, alors qu'elles devraient se trouver *au sommet* pour reproduire le mélange des deux paquets de cartes.

R 4.b C'est le même principe mais cette fois, on ne choisit pas au hasard l'élément qu'on va dépiler : on dépile toujours la plus grande des deux valeurs au sommet.

```
def fusion(p,q):
  r = pl.Pile([])
  while ((p.getHeight()>0) and (q.getHeight()>0)):
  # On choisit la plus grande valeur pour dépiler.
    if (p.getTop()>q.getTop()):
      r.push(p.pop())
    else:
      r.push(q.pop())
  # L'une des deux piles est maintenant vide, on vide donc l'autre pile
  while (p.getHeight()>0):
    r.push(p.pop())
  while (q.getHeight()>0):
    r.push(q.pop())
  # On remet les valeurs à l'endroit (c'est-à-dire dans l'ordre croissant)
  R = pl.Pile([])
  while not(r.isVoid()):
    R.push(r.pop())
  return R
```

R 5. Une chaîne de caractères est bien parenthésée lorsque chaque parenthèse fermante ")" est précédée, de plus ou moins loin, par une parenthèse ouvrante "(" et que toute parenthèse ouvrante est suivie, de plus ou moins loin, par une parenthèse fermante.
Le principe de l'algorithme est le suivant :
– On empile chaque parenthèse ouvrante ;
– À chaque parenthèse fermante, on dépile une parenthèse ouvrante.
Il y a deux types de mauvais parenthésage :
– Si une parenthèse se ferme sans avoir été ouverte au préalable, on va essayer de dépiler dans une pile vide ;
– Si une parenthèse ouverte n'est pas refermée, il restera une parenthèse ouvrante dans la pile à la fin de l'analyse.

```
def bienParenthesee(expr):
  p = Pile([])
  for x in expr:
    if (x=='('):
      p.push(x)
    if (x==')'):
      if (p.isVoid()):
        return False # Plus de parenthèses fermées que de parenthèses ouvertes !
      else:
        p.pop()
  # On vérifie qu'on a bien fermé toutes les parenthèses ouvertes.
  return (p.isVoid())
```

R 6. On parcourt la liste des arguments :

- Si on trouve un nombre, on le place sur la pile ;
- Si on trouve une opération, on dépile les deux premiers éléments et on empile le résultat de l'opération.

```
def npi(args):
  p = Pile([])
  for x in args:
    if (x in ['+','-','*','/']):
    # On dépile deux éléments pour effectuer une opération binaire
      a = p.pop() # dernier opérande saisi
      b = p.pop() # avant-dernier opérande saisi
      if (x=='+'):
        p.push(b+a)
      if (x=='*'):
        p.push(b*a)
      if (x=='-'):
        p.push(b-a)
      if (x=='/'):
        p.push(b/a)
    else: # On empile un opérande supplémentaire
      p.push(x)
  return p.pop()
```