Piles

1 Piles

1.1 Description de la structure de donnée

1.1.1 Structure de données ??

Une structure de donnée est la description théorique de ce que l'on appelle souvent type en python. Pour décrire une structure de données il faut préciser plusieurs choses :

— Quelles données pouvons-nous stocker ?
— Quelles opérations pouvons-nous effectuer sur la structure ?

Voici quelques exemples tirés de Python :

— Les chaînes de caractères (**str**) qui stockent des caractères. On peut accéder à un caractère, concaténer les chaînes...
— Les tableaux (**array**) numpy qui sont de taille fixe et ne contiennent que des données d'un même type (pas question de stocker des chaînes de caractère à côté d'entiers)
— Les listes (**list**)
— ...

1.1.2 Piles

La pile est une manière particulière de représenter des données en informatique. Le concept est assez simple et repose sur le principe du "dernier entré, premier servi" (ou LIFO pour Last In First Out).

— on peut "pousser" des données arbitraires sur la pile (on dit empiler, ou push en anglais)
— à tout moment, on peut retirer un object du haut de la pile (si elle n'est pas vide) : on dépile (ou pop en anglais)
— on peut savoir si la pile est vide ou non (reste-t-il un élément au sommet ?)
— Et c'est tout : pas d'accès à un élément quelconque, au nombre d'élément(s) ou autre.

En pratique, il nous faudra aussi pouvoir construire une pile vide. Cette description ne dépend pas du langage utilisé pour programmer.

1.2 Exemples

1.2.1 Descriptions

De nombreuses situations de la vie courante font intervenir des piles.

— La plonge dans un restaurant : les serveurs ammènent des assiettes sales (qu'ils empilent d'ailleurs...) et le plongeur dépile les assiettes une par une pour les laver.
— La fonction annuler (CTRL-Z) de beaucoup de logiciels
— L'historique d'un navigateur internet (ou tout autre bouton "précédent").

1.2.2 Pourquoi utiliser des piles

Plusieurs raisons peuvent nous mener à choisir les piles pour nos algorithmes - la nature même du problème - la simplicité de la structure qui permet d'espérer une meilleure efficacité en terme de temps de calcul et d'occupation mémoire - au plus bas niveau, les processeurs utilisent une ou des piles pour gérer les appels de fonction.

1.3 Algorithmes

1.3.1 Navigation

Prennons pour commencer la fonction "page précédente" d'un navigateur.

— p est une pile vide au départ
— A chaque fois que l'on quitte une page : empiler l'addresse de la page quittée dans p

— A chaque appuis sur le bouton "page précédente" : Si la pile n'est pas vide, dépiler une addresse puis charger la page visée.

Remarquez que le principe est le même pour la fonction "annuler" qui dépile la dernière action puis l'annule.

1.3.2 Manipulations classiques

Les manipulations suivantes sont très classiques :
Pour vider une pile p :

```
import pile

p = pile.Pile()  # p est maintenant une pile

while not p.estvide():
    sommet = p.depile()
    # faire des choses de sommet, voir même re-remplir la pile...
```

Utilisation classique d'une pile : remplir une pile puis la vider

```
p = Pile()
while condition:
    ...
    p.empile(untruc)
    ...
while not p.estvide():
    s = p.depile()
    ...
```

Exercice Créer une fonction qui calcule la longueur d'une pile.

```
def longueur(p):

```

1.3.3 Notation polonaise inversée

Il s'agit de représenter les opérations arithmétiques (+, -,... et les fonctions usuelles) en notation post-fixée, ie. en notant l'opérateur après le(s) chiffres.

Par exemple $3+5$ se note 3 5+.

Donnons la représentation de l'opération

$$(5+7)\times 3-13\times 9+\ln(5)$$

On obtient :

$$5\,7\,+\,3\,\times 13\,9\,\times\,-5\,\ln\,+$$

La lecture de ce genre d'expressions (utilisée sur d'anciennes calculatrices par exemple) est très simple. - Si on lit un nombre on l'empile - Sinon on dépile suffisement de nombres pour pouvoir calculer le résultat de l'opération et on empile ce résultat

A la fin de la lecture, un seul nombre doit rester dans la pile, c'est le résultat cherché.

```
import math

p = pile.Pile() # création d'une pile vide
p.empile(5)
p.empile(7)
print("au bout de 2 lectures ", p) # pour voir le résultat
# simulation de la lecture de +
a = p.depile()
b = p.depile()
p.empile(a + b)
# une fois traitée l'opération, on continue la lecture
p.empile(3)
# lecture de *
a = p.depile()
b = p.depile()
p.empile(a * b)
# poursuite de la lecture...
p.empile(13)
```

```
    p.empile(9)
    # lecture de *
    a = p.depile()
    b = p.depile()
    p.empile(a * b)
    # lecture de -
    # attention, on a empilé le membre de gauche en premier
    a = p.depile()
    b = p.depile()  # c'est lui le premier
    p.empile(b - a)

    p.empile(5)
    # lecture de ln
    a = p.depile()
    p.empile(math.log(a))
    # lecture de +
    a = p.depile()
    b = p.depile()
    p.empile(a + b)

    # on doit avoir une pile ne contenant qu'un élément
    # et cet élément est la réponse
    print(p.depile(), (5+7) * 3 - 13 * 9 + math.log(5))
```

```
('au bout de 2 lectures ', Pile([5, 7]))
(-79.3905620875659, -79.3905620875659)
```

1.3.4 Puissances entières

Le problème à résoudre est simple :

— Données : $a \in \mathbb{R}$ et $n \in \mathbb{N}$
— Sortie : a^n

Prennons un exemple On veut calculer 5^{20}.

Première remarque : $5^{20} = (5^{10})^2$, c'est à dire que si on sait calculer 5^3 il nous suffit d'une seule multiplication pour conclure.

Poursuivons. $5^{20} = ((5^5)^2)^2$. Par contre pour 5^5 l'astuce ne fonctionne plus. Qu'à cela ne tienne, $5^5 = 5 \times 5^4$.

On obtient $5^{20} = ((5 \times (5^2)^2)^2)^2$.

Comptons le nombre d'opérations : 4 mises au carrés, 1 multiplication. Seulement 5 multiplications contre 20 précédemment.

```
    def puissance_rec(a, n):
        if n == 0:
            return 1
        if n % 2 == 0:
            return puissance_rec(a, n //2)**2
        return a * puissance_rec(a, n - 1)
```

```
    %timeit puissance_rec(433, 123456)
```

```
10 loops, best of 3: 39.4 ms per loop
```

Le problème de l'approche précédente est qu'elle suppose de connaître la fin de la décomposition pour pouvoir commencer les calculs. Heureusement que l'on dispose des piles pour se souvenir des opérations à effectuer pour reconstruire le résultat.

```
    import pile

    CARRE = 0
    MULTIPLIE = 1

    def puissance_pile(a,n):
        """
        Calcule et retourne a^n

        >> puissance_pile(2,5)
        32
        >> puissance_pile(15, 0)
        1
```

```
    """
    # principe : on trouve d'abord les opérations à effectuer
    # que l'on empile une par une.
    # Puis on dépile et effectue les opérations.
    p = pile.Pile()  # création d'une nouvelle pile.
    e = n
    while e > 0:
        #n est pair, on mettra au carré, opération CARRE
        if e % 2 == 0:
            p.empile(CARRE)
            e = e // 2
        else:
            p.empile(MULTIPLIE)
            e = e - 1
    res = 1
    while not p.estvide():
        op = p.depile()
        if op == CARRE:
            res = res * res
        else:
            res = res * a
    return res
```

```
puissance_pile(2, 5), puissance_pile(10, 10)
```

```
(32, 10000000000)
```

```
%timeit puissance_pile(433, 123456)
```

```
10 loops, best of 3: 39.6 ms per loop
```

L'utilisation classique de telles fonctions se fait avec des exposants gigantesques, mais on ne retient que le reste de la division euclidienne par un certain module à chaque opérations.

Un exemple de telles fonctions est implémenté dans le module puissances. Voici les performances obtenues

```
import puissances as puiss
import random

n = random.randint(10**50, 10**51 - 1)
n
```

```
40323256184022433661999327658583008612409105364059 5L
```

```
%timeit puiss.puissance_rec(433, n)
```

```
1000 loops, best of 3: 862 µs per loop
```

```
%timeit puiss.puissance_pile(433, n)
```

```
1000 loops, best of 3: 818 µs per loop
```

```
%timeit puiss.puissance_python(433, n)
```

```
1000 loops, best of 3: 633 µs per loop
```

1.3.5 Hanoï, le retour

Utilisons des piles pour résoudre le problème des tours de Hanoï. Le principe général est le même que pour les puissances : il faut empiler les opérations à faire.

Ici, les opérations sont encodés par 4 données : - les 3 numéros de barres - le nombre de palet à déplacer

Au moment de dépiler, on sait traiter immédiatement le cas où il y a un seul palet à déplacer, sinon on empile les problèmes...

```
def deplace(dep, arr):
    print("Déplacer de %i vers %i." % (dep, arr))
```

```
    def hanoi_pile(n):
```

```
    hanoi_pile(4)
```

```
Déplacer de 0 vers 1.
Déplacer de 0 vers 2.
Déplacer de 1 vers 2.
Déplacer de 0 vers 1.
Déplacer de 2 vers 0.
Déplacer de 2 vers 1.
Déplacer de 0 vers 1.
Déplacer de 0 vers 2.
Déplacer de 1 vers 2.
Déplacer de 1 vers 0.
Déplacer de 2 vers 0.
Déplacer de 1 vers 2.
Déplacer de 0 vers 1.
Déplacer de 0 vers 2.
Déplacer de 1 vers 2.
```

2 Classes et objets

2.1 Principe et utilisation

Un objet (en fait toute donnée Python, mais passons) est une donnée qui possède des attributs et des méthodes. Ce sont respectivement des données supplémentaires attachées à cet objet ou des fonctions qui lui sont propres. Une classe représente le type de l'objet. on dit qu'un objet est une instance d'une classe

Un exemple étant plus parlant qu'un long discours, en voici

```
    l = list((1,2,3))
    l.append(1) # append est une méthode de l'objet l
    print("l est une instance de la classe", type(l))
    # __doc__ est un attribut de la liste l
    print("La documentation d'une liste est : ")
    print(l.__doc__)

    p = pile.Pile()
    print("p est une instance de la classe Pile", type(p))

    s = 'ab gdf'
    s.split(' ')
```

```
('l est une instance de la classe', <type 'list'>)
La documentation d'une liste est :
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
('p est une instance de la classe Pile', <type 'instance'>)


    ['ab', 'gdf']
```

Le but est maintenant de voir comment on peut implémenter une telle class Pile.

```
    class PileVideError(Exception): # voici un premier exemple de class
        pass


    class Pile():
        """
        Structure de données de pile

        Appeler Pile() pour créer une pile vide. Les méthodes sont
            - empile
            - depile
            - estvide
            - sommet
        """
```

```
    # une première méthode, qui construit les attributs de chaqu
    def __init__(self): # self se réfère à l'instance courante
        self._p = [] # _p est un attribut de chaque instance.
        # au départ la pile est vide.

    # chaque méthode est en fait une fonction.
    # par convention, on nomme le premier argument self,
    # et il fait référence (étiquette)
    # à l'instance qui utilise la méthode en question.
    def estvide(self):
        """
        Teste si cette pile est vide ou non
        """
        return len(self._p) == 0

    def sommet(self):
        """
        Retourne le sommet de cette pile sans dépiler,
        renvoie une erreur si la pile est vide
        """
        if self.estvide():
            raise PileVideError("La pile est vide !")
        else:
            return self._p[-1]

    def empile(self, elt):
        """
        Empile des données dans cette pile

        elt : l'élément à empiler

        Retourne cette pile
        """
        self._p.append(elt)
        return self
```

```
    def depile(self):
        """
        Dépile un élément. Renvoie une erreur si la pile est vide
        """
        if self.estvide():
            raise PileVideError("La pile est vide !")
        else:
            return self._p.pop()

#ces deux méthodes servent juste à l'affichage dans la console.
    def __str__(self):
        return str(self._p)

    def __repr__(self):
        return "Pile(" + str(self) + ")"
```

```
p = Pile()
p.empile(-4)
p, p._p
```

```
(Pile([-4]), [-4])
```

```
p2 = Pile()
p2.empile(-4)
# vérifions que chaque instance possède des attributs propres
id(p._p), id(p2._p)
```

```
(139779846554760, 139779846263008)
```

```
p._p == p2._p
```

```
True
```