Chapitre VII

Piles

Figure VII.1 – "On prend celle du dessus ou on en ajoute une ... rien d'autre!!!"

Sommaire

1 Structure de piles

1.1 Pourquoi les piles

L'usage de l'outil informatique conduit à stocker de très nombreuses informations en mémoire. L'objet stocké en mémoire peut être de différente nature (type) : `int`, `str`, `list`, `numpy.ndarray`, et accompagné des différentes opérations qu'il supporte, il constitue alors **une structure de données**.

La structure de données que nous avons manipulé le plus fréquemment jusqu'alors est la **liste**. Elle présente de grands avantages de souplesse, en particulier son caractère dynamique :

- insertion/suppression d'éléments **de tout rang** dans la liste (caractère **mutable**)
- nombreuses **méthodes Python** applicables : `.remove`, `.reverse`, `.sort`, `.append`, `.extend`, `.insert`

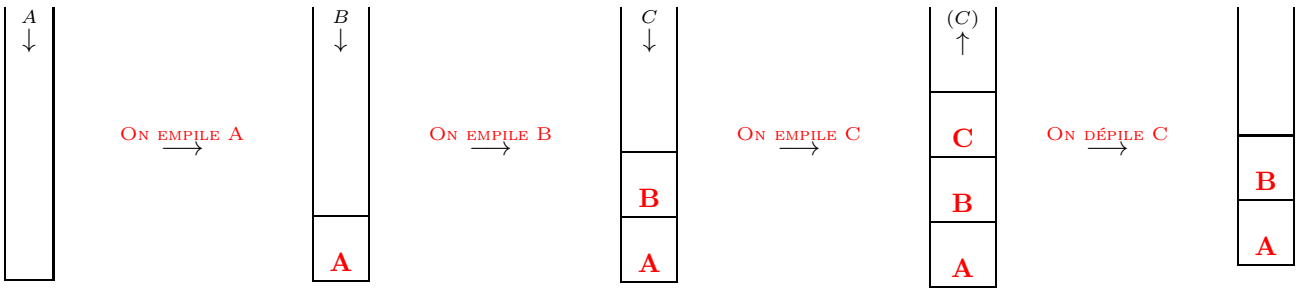
Nous allons maintenant introduire une nouvelle structure de données appelée **Pile** qui se révèle particulièrement adaptée dans la programmation de certaines méthodes algorithmiques.

Définition - (1.1) - 1:

On appelle **Pile** *une structure de données de forme identique à celle d'une* **liste**, *mais dont les opérations de type diffèrent de celles de cette dernière. Dans une pile :*

- *les données sont empilées les unes sur les autres, telle une pile d'assiettes !*
- *on ne peut accéder qu'au* **dernier élément de la pile**, *i.e. le sommet de la pile que l'on peut :*
  - ▶ *retirer de la pile, on* **dépile**
  - ▶ *consulter*
  - ▶ *sur lequel on peut ajouter un élément on* **empile**

Exemple : on veut mettre A,B,C dans une pile. Les opérations successives sont les suivantes : on empile A, puis on empile B, et enfin on empile C ; on peut ensuite "dépiler", mais forcé de commencer par C !!!. Schématiquement, cela donne :



Propriété - (1.1) - 1:

*Les piles sont des structures de type LIFO pour "***L***ast* **I***n,* **F***irst* **O***ut", c'est à dire "dernier entré, premier sorti".*

La `pile` n'étant pas un type natif de Python, nous devrons définir les opérations fondamentales suivantes par fonctions :

- `creer_pile(c)` : créé et renvoie une nouvelle pile de taille c, initialement vide.
- `empile(p,e)` : empile l'élément e sur le sommet de la pile p et renvoie p.
- `depile(p)` : dépile l'élément situé sur le sommet de la pile p et le renvoie.

On peut enrichir la structure de données en ajoutant les fonctions suivantes, non essentielles, mais parfois utiles :

- `vide_pile`(p) : renvoie l'état vide ou non de la pile sous forme du booléen `True` ou `False`.
- `taille_pile(p)` : revoie la taille de la pile sous forme d'un entier.
- `sommet_pile(p)` : renvoie la valeur du sommet de la pile.

EXERCICE N°1: *Donner la suite des opérations effectuées dans la manipulation de pile décrite sur le schéma ci-dessus.*

On retiendra finalement que les structures piles sont adaptées dans les cas suivants :

- stockage d'un nombre de données inconnu à l'avance (empilement possible "à l'infini")
- nécessité d'accéder uniquement au dernier élément empilé.

1.2 Construction d'une structure de piles

a - Piles bornées - définitions des fonctions de pile

Pour définir une pile bornée :

- on utilise une liste dont on déclare la taille N en choisissant celle-ci suffisamment grande pour pouvoir réaliser toutes les manipulations souhaitées.

- on stocke dans le premier élément de pile, et donc à l'indice 0 de la liste, le nombre d'éléments contenus dans la pile.

Les commandes Python à définir sont les suivantes :

Listing VII.1 –

```
def creer_pile(c):
    p=(c+1) *[None
    ]\\
    p[0]=0 #
    initialisation
    de la pile
    return p
```

Listing VII.2 –

```
def empile(p,e):
    n=p[0]
    assert n<len(p)
    -1 # on verifie
     la taille de
    pile
    p[0]=n+1
    p[n+1]=e
    return p
```

Listing VII.3 –

```
def pile_vide(p):
    return
    taille_pile(p)
    ==0
```

Listing VII.4 –

```
def taille_pile(p):
    return p[0]
```

Listing VII.5 –

```
def depile(p):
    n=p[0]
    p[0]=n-1
    assert n>0 #on
    verifie que la
    pile n'est pas
    vide
    return p[n]

```

Listing VII.6 –

```
def sommet_pile(p):
    assert
    taille_pile(p)
    >0 #verifie que
     la pile n'est
    pas vide
    return p[p[0]]
```

b - Piles non bornées - méthodes natives `.pop` et `.append`

L'utilisation des piles bornées est assujettie à une contrainte de taille, puisque le nombre maximum d'éléments qu'il est possible d'empiler doit être connu à l'avance.

On peut très facilement s'affranchir de cette limitation en définissant une autre structure de pile dite **non bornées**. Elle consiste simplement à créer une liste vide en guise de pile, et à en contrôler les éléments entrant et sortant à l'aide des méthodes `.pop` et `.append`.

Les commandes Python pour cette nouvelle structure sont alors :

Listing VII.7 –

```
def creer_pile():
    return list()
```

Listing VII.8 –

```
def empile(p,e):
    return p.append
    (e)
```

Listing VII.9 –

```
def pile_vide(p):
    return
    taille_pile(p)
    ==0
```

Listing VII.10 –

```
def taille_pile(p):
    return len(p)
```

Listing VII.11 –

```
def depile(p):
    assert len(p)>0
    return p.pop()

```

Listing VII.12 –

```
def sommet_pile(p):
    assert
    taille_pile(p)
    >0
    return p[-1]
```

2 Applications

2.1 Analyse du parenthésage

Le principe du «bon parenthésage» est de s'assurer qu'à toute parenthèse ouverte dans un texte est associée une parenthèse fermée. Ceci est particulièrement important dans les langages de programmation pour lesquels le bon parenthésage est une condition requise de bonne syntaxe.

> REMARQUE - (2.1) - 1:
>
> - La notion de parenthésage peut naturellement être étendue au cas des balises exploitées dans les langages XML ou HTML, qui une fois ouvertes doivent impérativement être fermées.
> - De nombreux logiciels de saisie informatique comme les programmes de traitement de texte, les environnements de développement de code, ou encore les tableurs utilisent des routines «temps réel» de vérification du parenthésage.

On peut par exemple analyser le bon parenthésage dans le cas d'une expression mathématique (ce contrôle est réalisé dans les formules des cellules d'un tableur par exemple). L'expression :

$$((3+2)\times 8)/(2+3) \tag{VII.1}$$

est bien parenthésée
alors que l'expression :

$$((3+2\times 8)/(2+3 \tag{VII.2}$$

ne l'est pas.

Nous allons voir que la structure de pile est un type particulièrement adapté à la vérification d'un bon parenthésage.

Pour cela nous allons créer une pile et parcourir un à un les caractères de l'expression à valider , de gauche à droite ; la règle est la suivante :

- Toute parenthèse ouvrante donne lieu à un empilement.
- Toute parenthèse fermante donne lieu à un dépilement.

Ainsi :
un bon parenthésage conduit à une pile vide en fin de lecture de l'expression (le programme devra renvoyer `True`), et non vide sinon (le programme devra renvoyer `False`).

> REMARQUE - (2.1) - 2: La taille de la pile ne pouvant être connue à l'avance, on utilisera préférentiellement une pile non bornée.

Un code d'évaluation du bon parenthésage de l'expression `expr` est par exemple :

Listing VII.13 –

```
def parenthesage(expr):
    p=creer_pile()
    for e in expr:
        if e=="(":
            empile(p,e)
        if e==")":
            if pile_vide(p):
                return False
            else:
                depile(p)
    return pile_vide(p)
```

Si l'on évalue évalue le parenthésage des deux expressions précédentes VII.1 et VII.2, la sortie de l'interpréteur donne :

```
>>> expr_juste="((3+ 2)*8)/(2+3)"
>>> expr_fausse="((3+ 2* 8)/(2+3"
>>> print parenthesage(expr_juste)
True
>>> print parenthesage(expr_fausse)
False
```

EXERCICE N°2: *Adapter le code précédent pour contrôler le bon parenthésage d'expressions comportant trois types de parenthèses :* (), [], *et* {} *? Par exemple :*

$$[(3+2)*8]/(2+3) \tag{VII.3}$$

et

$$[(3+2]*8/(2+3)) \tag{VII.4}$$

2.2 Evaluation par notation polonaise inversée (NPI) ou R*everse* P*olish* N*otation* (RPN)

L'évaluation d'une expression arithmétique nécessite de connaître les opérateurs fondamentaux $+, -, *, /$ et également les règles de proriétés relatives à ces derniers (multiplication et division prioritaires sur addition et soustraction). Le recours aux parenthèses est alors parfois requis. Par exemple :

$$(1+2)\times 3 \tag{VII.5}$$

$$1+2\times 3 \tag{VII.6}$$

ne donne évidemment pas le même résultat.

En outre, les choses se compliquent nettement si l'on veut faire évaluer ces expressions par une machine.

Pour la première, le calcul de $1+2$ nécessite de connaître 2 avant de pouvoir l'additionner à 1.
Pour la seconde, c'est encore pire, puisque les trois opérandes doivent être entrées en machine pour réaliser d'abord la multiplication de 2 par 3, puis l'addition du résultat avec 1.

Par conséquent, une lecture séquentielle «brute» de gauche à droite des expressions arithmétiques ainsi écrites n'est pas envisageable pour un calculateur informatique, qui doit d'abord connaitre toutes les opérandes avant d'appliquer dessus les opérateurs. Notre notation "habituelle" des expressions arithmétiques, appelée `notation infixe`, doit donc être revue dans le cas d'une évaluation informatique.

La solution a été apportée par le Polonais Jan Lukasiewicz en 1920 avec la **notation polonaise** ou notation préfixée, puis reprise dans les années 1950 par l'informaticien australien Charles Leonard Hamblin dans une forme dite **postfixée** appelée **N**otation **P**olonaise **I**nversée (NPI ou RPN en anglais). Le fabriquant de calculateur Hewlett-Packard a très largement diffusé cette méthode dès les années 60 en l'implémentant comme interface de saisie dans tous ses calculateurs financiers et scientifiques (HP9100,HP35,HP28,HP48,HP49, HP50, et plus récemment la HP-Prime.)

Le principe est d'entrer dans un premier temps les opérandes en machine, puis d'appliquer dans un second temps les opérateurs dessus.

$$\Longrightarrow$$

la NPI s'affranchit totalement de l'usage des parenthèses !!!

Par exemple, l'expression 2.2 est entrée en machine de la façon suivante :

$$1\ 2\ +\ 3\ *$$

Dans la mesure où chaque opérateur ne prend systématiquement que deux opérandes, **l'écriture NPI est sans ambiguité !!**

Là-encore l'usage d'une pile se révèle parfaitement adapté à l'exploitation de la NPI[1].

Le principe d'évaluation est le suivant :

- **On parcourt l'expression de gauche à droite (comme en notation `infixe`), mais...**
- **si l'on rencontre une opérande, on l'empile.**
- **si l'on rencontre un opérateur on depile deux opérandes, on applique l'opérateur dessus, et on empile le résultat obtenu.**

On propose l'implémentation suivante :
- On crée d'abord une pile non bornée pour les données.
- On parcourt l'expression de gauche à droite pour alimenter la pile ou si un opérateur est lu : on dépile 2 opérandes, on leur applique l'opérateur et on empile le résultat.
  En python cela donne :

Listing VII.14 –

```
p=creer_pile()
def evalNPI(exprNPI):
    listeNPI=exprNPI.split() #permet la conversion de la chaine NPI en liste
    for e in listeNPI:
        if e!="*" and e!="/" and e!="+" and e!="-":
            empile(p,int(e))
```

1. c'est précisément comme cela que fonctionne les calculateurs HP.

```
            print p #affiche la pile dès qu'une opérande est ajoutée
        if e=="+":
            op1=depile(p)
            op2=depile(p)
            empile(p,op1+op2)
        if e=="-":
            op1=depile(p)
            op2=depile(p)
            empile(p,op1-op2)
        if e=="*":
            op1=depile(p)
            op2=depile(p)
            empile(p,op1*op2)
        if e=="/":
            op1=depile(p)
            op2=depile(p)
            empile(p,op1/op2)
    return sommet_pile(p)
```

Par exemple, l'évaluation de "1 2 + 3 *" donne :

```
>>> print evalNPI("1 2 * 3 +")
[1]
[1, 2]
[2, 3]
5
```

3 Pour aller plus loin : déclaration d'une classe "Pile"

3.1 Principe des classes

a - Programmation orientée objet et classe

L'une des points forts de Python est qu'il est `orienté objet`. Nous avons déjà très largement exploité cette caractéristique en simple utilisateur lorsque nous appliquions des méthodes, par exemple `.pop()`, `.append(arg)`, `.delete(arg)` etc..., à des variables.

Les méthodes sont des fonctions contenues dans un objet de type `class`. Ainsi, définir un objet par une classe permet de fournir à l'interpréteur, à la fois les propriétés de type, ainsi que toutes les opérations envisageables sur la classe.

La suite propose dans un premier temps d'expliquer sommairement **le principe de déclaration d'une classe** et son utilisation, puis dans une seconde partie d'implémenter la classe `Pile`.

b - Syntaxe de déclaration des classes

Le code de déclaration d'une classe possède toujours la même structure :

Listing VII.15 –

```
class Maclass:
""" on va définir la nouvelle classe Maclass"""
    def __init__(self,arg1=val_init1, arg2=val_init2 ...):
        self._argappel1,self._argappel2.....=arg1,arg2....
    def methode1(self):
        return fct1(self._argappel1,self._argappel2....)
    def methode2(self):
```

```
        return fct2(self.arg1,self.arg2....)
    etc......
```

COMMENTAIRES :

- La première ligne "ouvre" la définition d'une nouvelle classe, ici `Maclass`
- La seconde ligne commente notre affaire!!!
- Les troisième et quatrième lignes permettent de déclarer une `méthode` particulière et **toujours présente** dans une définition de `class` ; on l'appelle `le constructeur` car elle créée et initialise l'objet de la classe. On notera l'emploi systématique du mot `self` désignant l'objet non encore appelé (on dira "non instancié"). On initialise ensuite les **attributs** `_argappel1`, `_argappel2`.... de l'objet qui sera instancié : c'est fait dans la quatrième ligne qui affecte à `self._argappel1`, `self._argappel2`,... les valeurs initialisées en préambule de la classe.

  Attention : on notera la présence d'un caractère "underscore" `_` devant les attributs d'objet après `self.`
- Les déclaration de `methodes` suivantes sont semblables à des définitions de fonctions et permettent de renvoyer une valeur lorsque lorsqu'elle sont appelées.

  Leur syntaxe est celle employée habituellement pour appliquer une méthode à un objet de la classe :

Listing VII.16 –

```
var=Maclass(arg1,arg2,....)
print var._argappel1,var._argappel2 #cas particulier: fait appel au
constructeur pour renvoyer les valeurs d'attributs de l'objet instancié.
print var.methode1() #applique la méthode 1 à l'objet var instancié
print var.methode2() #applique la méthode 2 à l'objet var instancié
```

Par exemple, dans la dernière ligne, l'argument `self` prend la valeur `Maclass var` lorsque `var.methode2()` est executée.

3.2 Un exemple d'implémentation : la classe pile

Les piles, non natives en Python, peuvent être très facilement implémentées à l'aide d'une classe qui définira alors toutes les méthodes nécessaires à leur manipulation.

On propose ci-dessous un exemple d'implémentation complètes du type `pile` à l'aide de la classe `Pile` :

Listing VII.17 –

```
class Pile:
    def __init__(self,pilevide=[]):
        self.pile=pilevide #initialise la pile vide

    #fonction d'empilement
    def empile(self,e):
        self.pile.append(e)

    #fonction de dépilement
    def depile(self):
        if len(self.pile)==0:
            raise ValueError ("La pile est vide!!!")
        return self.pile.pop()

    #fonction pilevide:
```

```
16      def pile_vide(self):
17          return len(self.pile)==0
18
19      #fonction taille_pile
20      def taille_pile(self):
21          return len(self.pile)
22
23      #fonction sommet_pile
24      def sommet_pile(self):
25          if len(self.pile)==0:
26              raise ValueError ("La pile est vide!!!")
27          return self.pile[-1]
```

Et voici ce que ça donne dans la pratique :

```
>>> p=Pile()
>>> print p.taille_pile()
0

>>> print p.pile_vide()
True
>>> p.empile("A")
>>> print p.pile_vide()
False
>>> p.empile("B")
>>> p.empile("C")
>>> print p.taille_pile()
3
>>> print p.sommet_pile()
C
>>> p.depile()
>>> print p.sommet_pile()
B
>>> print p.taille_pile()
2
```