C. Charignon

Table des matières

I Cours **2**

1 Introduction **2**
1.1 Structures de données 2
1.2 Mise au point : objets mutables 2
1.3 Gestion de la récursivité en mémoire 3

2 Les piles **3**
2.1 Présentation du type 3
2.2 En Python 3

3 Exemple important : notation polonaise inversée **4**
3.1 Présentation du principe 4
3.2 Programmation 4

II Exercices **5**

Première partie
Cours

1 Introduction

1.1 Structures de données

En informatique, lorsqu'on se lance dans un algorithme, un point important de réflexion est de choisir la structure de donnée la mieux adaptée. En effet, il existe de nombreuses manières d'enregistrer des données ; chacune a ses avantages et inconvénients. Typiquement, les différentes structures de données existantes permettent ou non de réaliser les opérations suivantes, et si oui plus ou moins rapidement :

- insérer un nouvel élément ;
- supprimer un élément ;
- remplacer un élément par un autre ;
- rechercher si un élément est présent ;
- rechercher le maximum ou le minimum.

Imaginons par exemple que nous voulions ranger notre collection de disques audio. Voici plusieurs méthode possibles, indiquez pour chacune des 5 opérations ci-dessus laquelle est la plus pratique (pour le dernier point, disons que rechercher le maximum ou le minimum signifie rechercher le disque le plus ancien ou le plus récent) :

- En vrac étalés par terre ;
- Dans une pile, sans ordre ;
- Dans une pile, trié par ordre alphabétique des artistes ;
- Dans une pile, trié par date de parution ;
- Dans différents tiroirs, un par style musical ;
- Dans différents tiroirs, selon l'ordre alphabétique.

La politique de Python dans ce domaine est simple : utiliser le type `list` pour presque tout ! De fait, ce type a été programmé pour que toutes les opérations ci-dessus soient possibles. En contrepartie, elles sont plus lentes qu'elle ne le seraient dans d'autres langages.

Vous avez utilisé en premier année le type `array` fourni par la bibliothèque `numpy`. Dans un objet de type `array`, on ne peut pas insérer ou supprimer d'élément. En contrepartie, les autres opérations sont beaucoup plus rapides.

Le seul nouveau type présent au programme est le type des « piles » que nous allons maintenant décrire.

Avant ça, clarifions un point : lorsque nousvoudrons décrire un type de données, nous donnerons

- les opérations élémentaires possibles ;
- et leur coût.

1.2 Mise au point : objets mutables

Le type `list` de Python, tout comme le type `array` de numpy sont modifiables, ce qui signifie qu'on peut modifier le contenu d'une variable de type `list` ou `array`.

Au contraire, les types élémentaires de booléens, flottants, entiers, chaînes de caractères sont des types persistants, on ne peut donc pas les modifier. Lorsque j'effectue :

```
1 x=1
2 x=x+2
```

je n'ai pas modifié le contenu de `x`, mais j'ai créé une nouvelle variable, que j'ai encore appelée x et qui a écrasé l'ancienne.

(Faire des dessins de pointeurs pour expliquer la différence en mémoire)

Il ne s'agit pas uniquement d'une question de point vue : la différence entre une variable modifiable ou persistante a des conséquences très concrètes. (faire des exemples)

- Une variable mutable peut être modifiée par un programme. Un tel programme ne renvoie rien (pas de return), on dit qu'il a uniquement des effets de bords. Certains appellent un tel programme une "procédure" au lieu d'une "fonction".
- Une variable mutable est copiée de manière superficielle. Si on veut effectuer une vraie copie, on peut utiliser la fonction `deepcopy` de la bibliothèque `copy`.

1.3 Gestion de la récursivité en mémoire

Voyons, schématiquement, ce qui se passe en mémoire lors de l'exécution d'un programme un peu compliqué qui appelle de nombreux autres programmes (typiquement une fonction récursive). Prenons l'exemple du calcul de puissances selon l'algorithme des puissances divisées.

Remarque : Toute fonction récursive peut être traduite en une fonction impérative utilisant une pile. Il suffit essentiellement de gérer soi-même la pile des appels.

cf exercice : 7

2 Les piles

2.1 Présentation du type

Une pile est une structure de données toute simple, qui permet uniquement deux opérations élémentaires :

- ajouter un élément (en anglais : « push ») ;
- et retirer un élément (« pop »).

L'élément qu'on peut récupérer est le dernier qu'on a mis. En anglais on dit que c'est une structure « LIFO » : Last In First Out. On pourrait aussi dire FILO d'ailleurs : First In Last Out.

On comprend l'utilisation du terme « pile ». Il faut s'imaginer que chaque élément ajouté est placé au sommet d'une pile, et que c'est uniquement l'élément du sommet qui est facile d'accès.

Aux deux opérations élémentaires ci-dessus, il convient des rajouter les opérations suivantes, nécessaire pour tout type de données mutable :

- créer une pile vide ;
- tester si une pile est vide.

Remarque : En particulier, on n'a pas accès à la taille de la pile. Si on veut parcourir une pile du début à la fin on devra utiliser une boucle de type « tant que la pile est non vide ». Mais en pratique on n'a pas envie de le faire : si le but était de parcourir les données du début à la fin, il fallait les mettre dans un tableau.

Comme on l'a vu en introduction, l'ordinateur utilise en permanence une pile : la pile des appels. C'est là qu'il enregistre tous les calculs de fonction qu'il devra effectuer. Cette pile pourrait s'appeler « à faire ».

cf exercice : ??

2.2 En Python

On a dit que la philosophie de Python est d'utiliser le type `list` comme type à tout faire. C'est pourquoi les créateurs de Python ont programmé sur les `list` les méthodes `pop` et `append` pour retirer ou ajouter un élément en extrémité, ce qui permet d'utiliser le type `list` comme une pile.

Dans les dessins qu'on a pu faire jusqu'ici, on représentait le haut de la pile vers le haut du tableau. Si on utilise une `list` Python, il faudra s'imaginer que le sommet de la pile est à droite. Soit vous renversez votre écran d'un quart de tour dans le sens trigonométrique, soit vous imaginez que la gravité va vers la gauche, soit vous travaillez vos exercices allongé à la romaine.

Remarque : Ainsi en pratique, dans les exercices sur les piles en Python, nous utiliserons le type `list`. Cependant, certains sujet de concours, lorsqu'ils veulent insister sur l'utilisation des piles précisent par exemple :

« On suppose connues quatre fonctions `push`, `pop`, `newStack` et `isEmpty` qui implémentent les opérations élémentaires sur les piles. Il est demandé de manipuler les piles uniquement via ces quatre fonctions. »

Bien que ce soit strictement inutile, on peut redéfinir les fonctions de base des piles :

```python
def nv_pile():
    return []

def estVide(p):
    return len(p)==0

def empile(x,p):
    p.append(x)

def dépile(p):
    return p.pop()
```

3 Exemple important : notation polonaise inversée

Exemple figurant au programme officiel des cpge.

3.1 Présentation du principe

Les premières calculatrices utilisaient souvent des piles (les HP ont continué un certain temps). En effet, il existe un moyen très pratique de noter des calculs qui ne nécessite pas de parenthèses, et qui pourra être traduit très facilement pour un ordinateur, c'est la notation polonaise inversée. Elle consiste à noter un opérateur *après* ses arguments. Par exemple $1 + 2$ sera noté $12+$. On dit que l'opérateur est alors "postfixe", alors qu'en notation mathématique traditionnelle, les opérateurs sont le plus souvent "infixes".

Pour être plus précis, un opérateur dans une formule postfixée est appliqué aux deux sous-formules valides qui le précèdent.

Exemple : que signifient les calculs suivants ?

- $123\times+$
- $12+3\times$
- $123-+$
- $1234-++\times$

Une fois qu'on a convaincu l'utilisateur d'utiliser la notation polonaise inversée, il devient facile [1] de programmer la calculette.

On part d'une pile vide, et on lit les instructions tapées par l'utilisateur dans l'ordre. Quand on lit un nombre, on l'empile. Quand on lit un opérateur unaire (qui prend un seul argument : par exemple une fonction cos, sin, etc.. ou le signe $-$) on dépile le nombre au sommet de la pile, on lui applique l'opérateur, et on le rempile. Enfin, si on rencontre un opérateur binaire ($+, \times$, etc...) on dépile les deux nombres au sommet de la pile, on leur applique l'opérateur, on rempile le résultat.

Si l'utilisateur n'a pas fait d'erreur de syntaxe, à la fin la pile contiendra un seul élément : le résultat voulu.

Notons que ceci ne demande qu'une seule lecture de la suite d'instruction : le temps d'exécution est linéaire (à condition que les opérateurs utilisées s'exécutent tous en temps borné bien sûr).

3.2 Programmation

On commence par une fonction annexe pour évaluer une opération :

```python
def applique(o,a,b):
    """ Entrée : o une chaîne de caractères qui représente une opération
                 a et b deux nombres
        Sortie : a o b """
    if o=="+":
        return a+b
    elif o=="-":
        return a-b
    elif o=="*":
```

1. tout est relatif

```
        return a*b
    elif o=="/":
        return a/b
```

Et voici la fonction principale :

```
def évaluation(formule):
    """ Entrée : une formule algébrique postfixée.
        Sortie : le résultat de son évaluation """

    p = nouvelle_pile()

    #On lit simplement la formule de gauche à droite
    for x in formule :
        if type(x)==int:
            empile(x,p)
        else:
            b=depile(p)
            a=depile(p)
            #Il faut appliquer l'opération x à a et b
            empile( applique(x, a, b), p)

    return depile(p)
```

Deuxième partie

Exercices

Exercices : piles

Exercice 1. ** Manipulation de piles abstraites

On suppose qu'on dispose d'un type `pile` qu'on peut manipuler via les fonctions `empile`, `depile`, `nouvellePile` et `estVide`.

Écrire les fonctions suivantes :

1. `regardeSommet` qui renvoie le premier élément sans modifier la pile.
2. `echangeSommets` qui échange les deux premiers éléments de la pile.
3. `extrait` qui prend en entrée une pile p et un élément x et qui enlève tous les x présent dans p.
4. `longueur` qui renvoie la longueur de la pile. Attention : la pile devra être revenu à son état initial à la fin.

Exercice 2. ** Correction de copies

Un professeur corrige une pile de copies exercice par exercice. Ainsi il corrige d'abord le premier exercice de chaque copie de la pile, la reposant ensuite sur une deuxième pile. Une fois la pile finie, il prend la deuxième pile et recommence avec l'exercice 2. Ainsi de suite jusqu'à épuisement des exercices.

1. Écrire l'algorithme en pseudo-code (c'est-à-dire en français).
2. Programmer cette méthode en Python. On prendra en entrée une pile, et le nombre d'exercices. Pour simuler la correction d'une copie on écrira un message «Correction de l'exercice k» où k est le numéro de l'exercice en cours.

Exercice 3. **! Parenthésage

Le but est de prendre en entrée une chaîne de caractères et de déterminer si elle est bien parenthésée.

1. Pour commencer on ne prend en compte que les parenthèses classiques `"("` et `")"`. Il n'y a pas besoin de pile, un simple entier contenant le nombre de parenthèses ouvertes mais pas encore fermées rencontrées suffira.
2. Maintenant, on suppose qu'il y a différent type de parenthèses (parenthèses, accolades, crochets, guillemets...). Par exemple, la chaîne suivante est mal parenthésée : `"{bla (blabla} blu)"`.
   (a) On va utiliser une « liste d'association » pour enregistrer la parenthèse fermante qui correspond à chaque parenthèse ouvrante. Ce sera une liste de couples de la forme *(parenthèse ouvrante, parenthèse fermante)*. Par exemple `parenthèses= [ ("(", ")"), ( "{","}"), ("[", "]")]`. Cette liste d'association pourra être placée en variable globale.
   Écrire une fonction prenant en entrée une parenthèse ouvrante et renvoyant la parenthèse fermante correspondante.
   (b) Passer au programme principal. On propose d'utiliser une pile de caractères. Chaque parenthèse ouvrante sera empilée. Et lorsqu'on rencontre une parenthèse fermante, on dépile et on vérifie que le type de parenthèse ouvrante dépilé est le bon.

Exercice 4. * Calcul booléen

1. Adapter la fonction qui évalue une expression écrite en notation polonaise inversée pour traiter des booléens, et les opérations « et », « ou », « non » et « implique ».
2. À présent rajouter également les opérateurs $<$ et $>$ qui prennent deux entiers et renvoient un booléen. Par exemple pour `2 3 < 5 7 > ou`, la fonction renverra `Vrai`.

Exercice 5. ** Passage à l'écriture normale

Le but est de transformer une expression écrite en notation polonaise inversée en une écriture classique. Par exemple `2 2 3 + *` sera transformé en la chaîne de caractères `"2 * (2 + 3)"` .

Le principe est simplement de procéder comme lorsqu'on effectue le calcul, sauf qu'au lieu d'effectuer les opérations, on créera la chaîne de caractère correspondante. On utilisera donc une pile de chaînes de caractères pour enregistrer les calculs intermédiaires.

On rappelle le fonctionnement de la méthode `format` : par exemple pour créer la chaîne de caractères `"(2+3)"`, on peut taper `"({}+{})".format(2,3)`.

Exercice 6. ** Algorithme de Horner

Un polynôme sera représenté par la liste de ses coefficients, par exemple `[1,5,2,3]` représente le polynôme $1+5X+2X^2+3X^3$. Le but est décrire une fonction d'évaluation, qui prendra en entrée un polynôme P et un nombre x et qui calculera $P(x)$.

Pour chacun des algorithme suivants, donner la complexité. Programmer le dernier.

1. Algorithme naïf : pour tout $k \in \mathbb{N}$, x^k est calculé par k multiplications.

2. En utilisant l'exponentiation rapide pour les calculs de puissance.
3. Par l'algorithme de Horner, basé sur la remarque suivante : soit $P \in \mathbb{R}[X]$, n son degré et $(a_0, ..., a_n)$ ses coefficients, alors :
$$P = a_0 + \left(a_1 + a_2X + ... a_nX^{n-1}\right) \times X.$$
Pour cet algorithme, il sera plus pratique de ranger les coefficients dans une pile, le coefficient constant étant au sommet et le coefficient de plus haut degré au fond. Ainsi, P sera représenté par `[an,...,a1,a0]` et le polynôme $a_1 + a_2X + ... a_nX^{n-1}$ n'est autre que celui représenté par la pile `[an,...,a1]`...

Exercice 7. * ! Dérécursivation**

On reprend le problème des tours de Hanoï. Le but est d'écrire une fonction non récursive pour afficher les opérations à faire pour le résoudre.

On utilisera une pile `aFaire` contenant les opérations restant à faire. L'opération déplacer n disque de la tige i vers la tige j sera représentée par le triplet (n, i, j). Ainsi, initialement, la pile contiendra $(n, 0, 2)$.

On utilisera une boucle "tant que la pile est non vide". A chaque étape, on regardera la tâche suivante à effectuer. S'il s'agit de déplacer un disque, on affichera directement le déplacement à effectuer. Sinon, on remplacera dans la pile la tâche de déplacer k disque par une suite de tâches déplaçant $k-1$ disques.

On pourra écrire une fonction `tigeRestante` qui étant donnés i et j éléments de $[\![0, 2]\!]$ détermine k tel que $k \in [\![0, 2]\!] \setminus \{i, j\}$.

Exercice 8. * Autre exemple de dérécursification**

Écrire une version non récursive du tri rapide. On utilisera une pile (donc en pratique un tableau python) pour contenir les prochaines opérations à faire.

Quelques indications

1 1.
2. Mettre dans une autre pile tous les éléments à garder, puis les remettre dans la pile de départ.

6 Il est sans doute plus pratique ici d'utiliser une fonction récursive.

8 On utilisera une pile de couples : un couple `(d, f)` signifie que la prochaine plage à trier est `t[d:f]`.

Quelques solutions

1

3

4

5

6

7

8