Séance 7 : Modules et types abstraits

L1 – Université Côte d'Azur

Exercice 1 — Parcours suffixe et évaluation avec une pile

Le but de cet exercice est d'évaluer des expressions arithmétiques automatiquement. Certes, Python sait déjà le faire, mais il peut être intéressant de savoir le programmer soi-même.

Il est assez difficile d'évaluer une expression représentée par une chaîne (par exemple `"20-(4+5 * 3)"`). Il faut gérer les priorités et les parenthèses et savoir dans quel ordre faire les calculs. Il existe cependant une écriture, la fameuse *notation polonaise inversée* (NPI), très simple à calculer avec une pile. C'était la notation utilisée par les calculatrices HP [1] lors de la jeunesse de mon père (ce qui ne nous rajeunit pas…). Cette écriture permet de représenter une formule sans aucunes parenthèses et où les calculs s'effectuent dans l'ordre de la lecture. Par exemple la formule précédente deviendra « 20 4 5 3 × + − » (on fait d'abord la multiplication, puis l'addition, en enfin, on termine par la différence.

L'algorithme d'exécution est le suivant : on parcours la liste, si on tombe sur un nombre on l'ajoute à la pile et si on trouve une opération, on extrait les deux élèments du haut de pile, on leur applique l'opération et on ajoute le résultat à la fin de la pile.

1. Rappelez rapidement les principales fonctions qui définissent l'interface d'une pile.
2. Appliquer à la main l'algorithme précédent à la liste `[1, 2, '-', 3, 4, '+', '*']`. À quelle expression mathématique cela correspond-il ?
3. Quelle liste correspond à la formule $\frac{3 \times (2+5)}{7}$?
4. Écrire une fonction `calculer(liste)` qui à partir d'une liste de nombres et de chaînes (uniquement une de ces quatre : `"+"`, `"-"`, `"*"` et `"/"`) exécute le calcul correspondant.

Exercice 2 — Doubler les valeurs d'une matrice

On représente une matrice par la liste de ses lignes (qui sont elles-mêmes représentées par des listes).

Par exemple, `M = [ [1,2,3], [4,5,6] ]` représente la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.

On rappelle que vous devez utiliser les deux fonctions vues en cours (sans avoir à les réécrire, même si vous devez savoir le faire) :

- `dimensions(M)` qui renvoie un couple `(n,m)` correspondant au nombre de lignes et de colonnes.
- `matrice_nulle(n,m)` qui renvoie une matrice de dimensions $n \times m$ et ne contenant que des zéros.

1. Quelle instruction permet de remplacer le `3` par un `0` ?
2. Si `M` est une matrice définie dans Python, que donne le calcul `2 * M` dans la console Python ?
3. Écrivez une fonction `doubler(M)` qui *modifie* la matrice `M` et double chacun de ses coefficients.
4. Écrivez une fonction `double(M)` qui *renvoie* une nouvelle matrice correspondant à la matrice `M` dans laquelle les coefficients ont été doublés.
5. Écriver une fonction `est_le_double(M1,M2)` qui renvoie `True` si `M2` vaut le double de `M1` et `False` sinon. On ne créera pas de nouvelles matrices.

1. https://fr.wikipedia.org/wiki/Calculatrices_HP

Exercice 3 — Implémenter une pile bornée

```python
TAILLE_PILE = -1
HAUT_PILE   = -2
DÉBUT_PILE  = -3 # À partir de la question 3

def incrémenter(pile, indice):
    pile[indice] =  pile[indice] + 1

def décrémenter(pile, indice):
    pile[indice] =  pile[indice] - 1
```

On va chercher à implémenter un pile bornée à partir d'un tableau (c'est à dire une liste dont on ne changera pas la taille). On considère les conventions suivantes : la dernière case du tableau correspond aux dimensions maximales de la pile et l'avant dernière case correspond à la première case libre de la pile. On stockera les valeurs de la pile au début du tableau. Pour simplifier la lecture et l'écriture du code, on définit les variables globales ci-contre. Ci-dessous, une exemple simple d'exécution :

- On commence par initialiser une pile de taille 4

				···	0	4

- On empile 9 à l'indice 0

9				···	1	4

- On empile 8 à l'indice 1

9	8			···	2	4

- On dépile (ce qui renvoie 8)

9				···	1	4

1. Écrire les fonctions `initialiser(taille_tab, taille_pile)`, `est_vide(pile)`, `est_pleine(pile)`, `push(pile, x)` et `pop(pile)`. On soulèvera une erreur lors de l'ajout d'un élèment à une pile pleine.

```python
def nouvelle_pile(taille_tampon, taille_pile):
    pile =  [0] * taille_tampon
    pile[HAUT_PILE]   = 0
    pile[DÉBUT_PILE]  = 0  # Pour la question 3
    pile[TAILLE_PILE] = taille_pile
    return pile

def est_vide(pile):
    return pile[HAUT_PILE] == pile[DÉBUT_PILE]

def est_pleine(pile):
    return  pile[HAUT_PILE] == pile[TAILLE_PILE]

def push(pile, x):
    if est_pleine(pile):
        raise ValueError("Pile pleine")
    i = pile[HAUT_PILE]
    pile[i] = x
    incrémenter(pile, HAUT_PILE)

def pop(pile):
    if est_vide(pile):
        raise ValueError("Pile Vide")
    décrémenter(pile, HAUT_PILE)
    i = pile[HAUT_PILE]
    return pile[i]
```

2. Pour pouvoir continuer à ajouter des élèments à la pile lorsqu'elle est pleine, écrire une fonction `réduire_v1(pile)` qui supprime l'élement le plus ancien de la pile en décalant tous les élèments vers la gauche. Modifier la fonction `push` pour réduire la pile si nécessaire avant d'ajouter le nouvel élèment.

9	8	7	6	···	4	4	⟶ réduire_v1 ⟶	8	7	6		···	3	4

```python
def réduire_v1(pile):
    décrémenter(pile, HAUT_PILE)
    haut = pile[HAUT_PILE]
    for i in range(haut): # haut exclu
        pile[i] = pile[i+1]

def push(pile, x):
    if est_pleine(pile):
        réduire_v1(pile)
    i = pile[HAUT_PILE]
    pile[i] = x
    incrémenter(pile, HAUT_PILE)
```

3. Comme la fonction `réduire_v1` est coûteuse, nous allons ajouter une nouvelle information à la fin de la pile : l'indice de début de pile. Ainsi, pour réduire, il suffira d'incrémenter l'indice du début de pile. Écrire la fonction `réduire` implémentant cette nouvelle stratégie. Modifier `est_plein` en conséquent.

–	9	8	7	6			···	0	4	4	La pile est pleine
–		8	7	6			···	1	4	4	Après réduction
–		8	7	6	5		···	1	5	4	Après le push de la valeur 5
–			7	6	5	4	···	2	6	4	Après le push de la valeur 4 (et réduction)

```python
def réduire(pile):
    if est_vide(pile):
        raise ValueError("On ne peut réduire une pile vide")
    incrémenter(pile, DÉBUT_PILE)

def est_pleine(pile):
    haut  = pile[HAUT_PILE]
    début = pile[DÉBUT_PILE]
    taille = pile[TAILLE_PILE]
    return  haut -  début == taille
```

4. Modifier les fonctions `incrémenter` et `est_pleine` pour que l'indice revienne à zéro une fois la capacité maximale de la pile atteinte (on laissera une case vide de marge entre la fin et le début de pile).

–	9	8	7	6			···	0	4	4	La pile est pleine	9 8 7 6
–		8	7	6			···	1	4	4	Après réduction	8 7 6
–		8	7	6	5		···	1	5	4	Après le push de la valeur 5	8 7 6 5
–	4		7	6	5		···	2	0	4	Après le push 4 (et réduction)	7 6 5 4

```python
# Il suffit de travailler modulo n avec n = taille + 1
def incrémenter(pile, indice):
    n = pile[TAILLE_PILE] + 1
    pile[indice] = (pile[indice] + 1)%n

def est_pleine(pile):
    haut  = pile[HAUT_PILE]
    début = pile[DÉBUT_PILE]
    n = pile[TAILLE_PILE] + 1
    return (1 + haut -  début) % n == 0 # car (1+taille)%n == 0

```

5. Pourquoi avoir laissé une case vide ? Quelle est la valeur minimale de `taille_tab` en fonction de `taille_pile` ?

Sans le +1 dans la formule précédente, `est_plein` se résumerait à `haut==début` et serait indistinguable de `est_vide`. Il faut donc une case inutilisée pour différencier les deux cas.

En tout il faut `taille_pile + 1` pour stocker la pile et sa case inutilisée ainsi que trois cases supplémentaires pour stocker les trois attibuts de la pile, sa taille, son début et sa fin ce qui donne : `taille_tab` $\geq$ `taille_pile + 4`.

En pratique, on pourrait même se passer de `taille_tab` en prenant la plus petite taille possible pour le tableau : `taille_pile + 3` (et non plus 4 car on se passerai alors de la dernière case du tableau ; la valeur qui y était stockée pouvant être trouvée par la formule `len(pile) - 3`)

6. Dans une pile pleine de 100 éléments, combien d'opérations (écritures dans le tableau) utilisera t'on avec `réduire_v1` lors de l'ajout d'un nouvel élément ? Même question avec `réduire`.

   À chaque ajout, la fonction `réduire_v1` qui déplace 99 éléments vers la gauche, modifie le haut de la pile : ce qui donne 100 opérations.

   La fonction `réduire` modifie simplement le bas de la pile : 1 opération.

   La première fonction est dite en $O(n)$ ou plus simplement linéaire (si la taille double, sa complexité double aussi). La seconde fonction est en $O(1)$ ou plus simplement constante (le nombre d'opération ne dépend pas de la taille de la pile)

   La version finale du code se trouve ci-après (en commentaire les modifications à apporter si on souhaite se passer de la dernière case du tableau.

```python
TAILLE_PILE = -1 # À supprimer
HAUT_PILE   = -2 # ou -1 si on supprime TAILLE_PILE
DÉBUT_PILE  = -3 # ou -2 si on supprime TAILLE_PILE

def nouvelle_pile(taille_pile):
    pile =  [0] * (taille_pile + 4) # ou [0] * (taille_pile + 3)
    pile[HAUT_PILE]   = 0
    pile[DÉBUT_PILE]  = 0
    pile[TAILLE_PILE] = taille_pile # À supprimer
    return pile

def incrémenter(pile, indice):
    n = pile[TAILLE_PILE] + 1 # ou n = len(pile) - 3 + 1 = len(pile)-2
    pile[indice] = (pile[indice] + 1)%n

def decrémenter(pile, indice):
    n = pile[TAILLE_PILE] - 1 # ou n = len(pile)-2
    pile[indice] = (pile[indice] - 1)%n

def est_vide(pile):
    return pile[HAUT_PILE] == pile[DÉBUT_PILE]

def est_pleine(pile):
    n = pile[TAILLE_PILE] + 1 # ou n = len(pile)-2
    return (1 + pile[HAUT_PILE] -  pile[DÉBUT_PILE]) % n == 0

def réduire(pile):
    if est_vide(pile):
        raise ValueError("On ne peut réduire une pile vide")
    incrémenter(pile, DÉBUT_PILE)

def push(pile, x):
    if est_pleine(pile):
        réduire(pile)
    i = pile[HAUT_PILE]
    pile[i] = x
    incrémenter(pile, HAUT_PILE)

def pop(pile):
    if est_vide(pile):
        raise ValueError("Pile vide")
    décrémenter(pile, HAUT_PILE)
    i = pile[HAUT_PILE]
    return pile[i]
```

Exercice 4 — Carré magique

Un carré magique est une matrice $n \times n$ et telle que la somme de chaque ligne, de chaque colonne, et des deux diagonales est une constante S. Par exemple, pour $n = 3$, on a le carré magique ci-contre de somme constante $S = 15$.
On représente une matrice par une liste de listes. Par exemple, le carré magique ci-contre correspond à `cm = [ [2,7,6] , [9,5,1] , [4,3,8] ]`

2	7	6
9	5	1
4	3	8

1. Écrivez une fonction `est_carré(M)` qui prend en argument une liste de listes `M` et qui renvoie `True` si `M` est une matrice carrée et `False` sinon.

```python
def est_carré(M):
    if M==[]:
        return False
    n = len(M)
    for ligne in M:
        if len(ligne) != n:
            return False
    return True
```

2. Écrivez une fonction `est_magique(M)` qui prend en argument une liste de listes `M` et qui renvoie `True` si `M` est un carré magique et `False` sinon.
   Pour augmenter la lisibilité du code, on écrit des fonctions permettant d'accéder aux lignes, au colonnes et aux diagonales.

```python
def ligne(M,i):
    return M[i]

def colonne(M, j):
    (n,m) = dimensions(M)
    return [ M[i][j] for i in range(m)]

# Renvoie la diagonale de la matrice M. k=0 ou 1 suivant si on veut la
# diagonale de gauche à droite ou de droite à gauche

def diagonale(M,k):
    (n,m) = dimensions(M)
    if k==0:
        return [ M[i][i] for i in range(n)]
    else:
        return [ M[n-1-i][i] for i in range(n)]
```

```python
def est_magique(M):
    if not est_carré(M):
        return False
    (n,m) = dimensions(M) # Forcément n=m
    S = sum(ligne(M,0)) # Vous savez écrire la fonction sum !
    # vérification des lignes
    for i in range(n):
        if sum(ligne(M,i)) != S:
            return False
    # vérification des colonnes
    for j in range(n):
        if sum(colonne(M,j)) != S:
            return False
    # vérification des diagonales
    for k in range(2): # k=0 ou 1
        if sum(diagonale(M,k)) != S:
            return False
    # Si tous les tests ont été passé avec succés
    return True
```