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 parcourt la liste, si on tombe sur un nombre on l'ajoute à la pile et si on trouve une opération, on extrait les deux éléments 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

```
TAILLE_PILE   = -1
DÉBUT_PILE    = -2
CAPACITÉ_PILE = -3

def indice_haut_pile(pile):
    DÉBUT  = DÉBUT_PILE
    TAILLE = TAILLE_PILE
    return pile[DÉBUT] + pile[TAILLE]
```

On va chercher à implémenter une pile bornée à partir d'un tableau (une liste de taille fixe). La capacité maximale de la pile, l'indice de début ainsi que le nombre effectif d'éléments de la pile seront stockés dans les trois dernières cases.

	11	22		···	4	1	2

Le tableau ci-dessus, représente une pile d'une capacité de 4, contenant 2 éléments (11 et 22) à partir de l'indice 1. Ci-dessous, un exemple simple d'exécution :

- On commence par initialiser une pile de capacité maximale 4
- On empile 9 à l'indice 0
- On empile 8 à l'indice 1
- On dépile (ce qui renvoie 8)

				···	4	0	0
9				···	4	0	1
9	8			···	4	0	2
9				···	4	0	1

1. En utilisant la fonction et les variables globales de l'encadré en haut à gauche écrire les fonctions : `initialiser(taille_tableau, capacité)`, `est_vide(pile)`, `est_pleine(pile)`, `push(pile, x)` et `pop(pile)`. On soulèvera des erreurs lorsque nécessaire.

```
def nouvelle_pile(taille_tab, capacité):
    pile =  [0] * taille_tab
    pile[TAILLE_PILE]   = 0
    pile[DÉBUT_PILE]    = 0
    pile[CAPACITÉ_PILE] = capacité
    return pile

def est_vide(pile):
    return pile[TAILLE_PILE] == 0

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

def push(pile, x):
    if est_pleine(pile):
        raise ValueError("Pile pleine")
    i = indice_haut_pile(pile)
    pile[i] = x
    pile[TAILLE_PILE] = pile[TAILLE_PILE] + 1


def pop(pile):
    if est_vide(pile):
        raise ValueError("Pile Vide")
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    i = indice_haut_pile(pile)
    return pile[i]
```

2. Écrire une fonction `réduire_naïf(pile)` pour pouvoir continuer à ajouter des éléments à la pile lorsque cette dernière est pleine. On supprimera l'élement le plus ancien de la pile en décalant tous les éléments vers la gauche. Modifier la fonction `push` pour réduire la pile si nécessaire avant d'ajouter le nouvel élément.

9	8	7	6	···	4	0	4	⟶ réduire_naïf ⟶	8	7	6		···	4	0	3

```python
def réduire_naïf(pile):
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    haut = indice_haut_pile(pile)
    for i in range(haut): # haut exclu
        pile[i] = pile[i+1]

def push(pile, x):
    if est_pleine(pile):
        réduire_naïf(pile)
    i = indice_haut_pile(pile)
    pile[i] = x
    pile[TAILLE_PILE] = pile[TAILLE_PILE] + 1
```

3. Comme la fonction `réduire_naïf` est coûteuse, nous allons plutôt faire glisser la pile en incrémentant l'indice de début. Écrire la fonction `réduire` implémentant cette nouvelle stratégie. On supposera le tableau assez grand pour que le haut de pile n'atteigne pas les trois cases finales.

—	9	8	7	6			···	4	0	4	La pile est pleine
—		8	7	6			···	4	1	3	Après réduction
—		8	7	6	5		···	4	1	4	Après le push de la valeur 5
—			7	6	5	4	···	4	2	4	Après le push de la valeur 4 (et réduction)

```python
def réduire(pile):
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    pile[DÉBUT_PILE]  = pile[DÉBUT_PILE]  + 1


def push(pile, x):
    if est_pleine(pile):
        réduire(pile) # c'est le seul changement
    i = indice_haut_pile(pile)
    pile[i] = x
    pile[TAILLE_PILE] = pile[TAILLE_PILE] + 1
```

4. Pour rester borné, nous allons réutiliser les cases du début de tableau une fois l'indice correspondant à la capacité atteint. Modifier les fonctions `indice_haut_pile` et `réduire` pour que les indices de début et de haut de pile reviennent à zéro, de manière cyclique, une fois la capacité maximale atteinte.

—	9	8	7	6			···	4	0	4	La pile est pleine	9 8 7 6
—		8	7	6			···	4	1	3	Après réduction	8 7 6
—	5	8	7	6			···	4	1	4	Après le push de la valeur 5	8 7 6 5
—	5	4	7	6			···	4	2	4	Après le push 4 (et réduction)	7 6 5 4

```python
# Il suffit de travailler pour l'indice de début modulo n avec n = capacité
def réduire(pile):
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    n = pile[CAPACITÉ_PILE]
    pile[DÉBUT_PILE]  = (pile[DÉBUT_PILE]  + 1)%n

def indice_haut_pile(pile):
    n = pile[CAPACITÉ_PILE]
    return (pile[DÉBUT_PILE] + pile[TAILLE_PILE])%n
```

5. Comment choisir `taille_tableau` en fonction de `capacité` ? Modifier le code pour optimiser la taille.

Il faut capacité + 3 cases pour gérer la pile. En pratique, on pourrait même gagner une case supplémentaire en remarquant que la capacité peut se déduire de la longueur du tableau.

Il suffit de remplacer chaque occurences de `PILE[CAPACITÉ_PILE]` par `len(pile) -2`

```
def nouvelle_pile(capacité):
    pile =  [0] * (capacité+2)
    pile[TAILLE_PILE]   = 0
    pile[DÉBUT_PILE]    = 0
    return pile
```

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

   Lors de l'ajout, la fonction `réduire_naïf` déplacera 99 éléments vers la gauche et modifiera la taille de la pile : ce qui donne 100 opérations.

   La fonction `réduire` modifie simplement le début et la taille de la pile : 2 opérations.

   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
DÉBUT_PILE    = -2

def nouvelle_pile(capacité):
    pile =  [0] * (capacité+2)
    pile[TAILLE_PILE]   = 0
    pile[DÉBUT_PILE]    = 0
    return pile

def est_vide(pile):
    return pile[TAILLE_PILE] == 0

def est_pleine(pile):
    return  pile[TAILLE_PILE] == len(pile) - 2

def réduire(pile):
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    n = len(pile) - 2
    pile[DÉBUT_PILE]  = (pile[DÉBUT_PILE]  + 1)%n

def indice_haut_pile(pile):
    n = len(pile) - 2
    return (pile[DÉBUT_PILE] + pile[TAILLE_PILE])%n

def pop(pile):
    if est_vide(pile):
        raise ValueError("Pile Vide")
    pile[TAILLE_PILE] = pile[TAILLE_PILE] - 1
    i = indice_haut_pile(pile)
    return pile[i]

def push(pile, x):
    if est_pleine(pile):
        réduire(pile)
    i = indice_haut_pile(pile)
    pile[i] = x
    pile[TAILLE_PILE] = pile[TAILLE_PILE] + 1
```

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 $S = 15$.

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
```