TP : piles et files avec `deque`

1 Piles

1.1 Tutoriel

On utilise dans ce TP la classe *deque* qui implémente les piles comme des listes doublement chaînées. Le constructeur *deque* crée une liste vide. L'ajout d'un élément au sommet se fait avec *append*, le dépilement avec *pop* :

[1]:
```
from collections import deque
myStack = deque()#création d'une pile vide

myStack.append('a')#ajout d'un élément
myStack.append('b')#ajout d'un élément
myStack.append('c')#ajout d'un elt

print(myStack)
```

[2]:
```
for i in range(3):
    e = myStack.pop()#dépiler
    print(e, end=" ")
print("\n",myStack)
```

```
c b a
 deque([])
```

Dépiler une liste vide soulève une exception *IndexError*. Voici comment on récupère l'exception pour poursuivre le traitement :

[3]:
```
try:#essai de dépilement
    myStack.pop()
except IndexError as ie:
    print(ie)#affiche le msg d'erreur (facultatif)
    print("On peut continuer à travailler,\n\
    l'exception est traitée")
```

```
pop from an empty deque
On peut continuer à travailler,
    l'exception est traitée
```

1.2 Exo : pile vide

A l'aide des seules primitives *append* (pour empiler), *pop* (pour dépiler) et du traitement de l'exception *IndexError*, écrire la fonction :

```
def isempty(pile):
```

qui indique si la pile en argument est vide. Bien sûr, après l'appel de la fonction la pile doit avoir le même contenu qu'avant.

[5]:
```
p1, p2 = deque(), deque()
p1.append(1)
print(isempty(p1), isempty(p2))
print(p1,p2)
```

```
False True
deque([1]) deque([])
```

1.3 Exo : consulter le sommet

A l'aide des seules primitives *append* (pour empiler) et *pop* (pour dépiler), écrire la fonction :

```
def peek(pile):
```

qui retourne le sommet de la pile en argument lorsqu'elle n'est pas vide. Sinon une exception *IndexError* est soulevée. Bien sûr, après l'appel de la fonction la pile doit avoir le même contenu qu'avant.

[7]:
```
p = deque()
p.append(10); p.append(20)
print(peek(p))
print(p)
```

```
20
deque([10, 20])
```

1.4 Exo : renversement

A l'aide des seules primitives *append* (pour empiler),*pop* (pour dépiler) et des fonctions *isempty* et *peek* écrire la procédure :

```
def rev(pile):
```

qui renverse la pile. On n'utilisera pas de liste mais des piles auxiliaires.

[9]:
```
p = deque()
for i in range(3):
    p.append(i*10)
print(p,id(p))
rev(p)
print(p,id(p))
```

```
deque([0, 10, 20]) 140602171339392
deque([20, 10, 0]) 140602171339392
```

1.5 Exo : rotation

A l'aide des seules primitives *append* (pour empiler),*pop* (pour dépiler) et des fonctions *isempty* et *peek* écrire la procédure :

```
def rotation(pile):
```

qui échange le sommet et la base de la pile. On n'utilisera pas de liste mais des piles auxiliaires.

[11]:
```
p = deque()
for i in range(4):
    p.append(i*10)
print(p,id(p))
rotation(p)
print(p,id(p))
```

```
deque([0, 10, 20, 30]) 140602171393872
deque([30, 10, 20, 0]) 140602171393872
```

1.6 Exo : permutations encodables

Dans cet exercice, les *permutations* sont implantées par des tuples.

On dit qu'une permutation $(a_1, a_2, \dots a_n)$ de $(1, 2, \dots n)$ peut être *engendrée par une pile* lorsque il est possible, à partir de la séquence d'entrée $(1, 2, \dots, n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1, a_2, \dots a_n)$ en utilisant les opérations suivantes :

- empiler l'élément suivant de la séquence d'entrée
- ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2, 3, 1)$ est engendrée par la suite d'opérations : $EEDEDD$.

1.6.1 Q1

Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?

$(3, 1, 2)$; $(3, 4, 2, 1)$; $(4, 5, 3, 7, 2, 1, 6)$ et $(3, 5, 7, 6, 8, 4, 9, 2, 10, 1)$

1.6.2 Q3

Montrer que s'il existe un triplet $(i, j, k) \in [1, n]^3_{\mathbb{N}}$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation $(a_1, a_2, \dots a_n)$ n'est pas engendrable par une pile.

1.6.3 Q4

Utilisant ce qui précède, écrire une fonction

```
def encodable(p):
```

qui prend en paramètre un tuple d'entiers de 1 à n représentant une permutation (supposée bien formée) à afficher et déterminant si cette dernière peut être engendrée par une pile.

Cette fonction gère une pile *deque* contenant des entiers et retourne une liste :

- la liste vide si la permutation n'est pas encodable
- sinon, une liste de $2n$ caractères E et D indiquant la séquence d'encodage.

[13]:
```
encoder((2,3,1))
```

[13]:
```
['E', 'E', 'D', 'E', 'D', 'D']
```

[14]:
```
encoder((3,1,2))
```

[14]:
```
[]
```

1.6.4 Q5

Expliquer comment toute permutation peut être engendrée à l'aide de deux piles (mais toujours en respectant l'ordre de la séquence d'entrée), et rédiger la fonction

```
def encoder2(p)
```

correspondante.

Il s'agit donc d'introduire une seconde pile pour stocker les éléments qui ne doivent pas être affiché tout de suite.

[16]:
```
encoder2((3,1,2))
```

[16]:
```
['E', 'E', 'E', 'D', 'e', 'D', 'd', 'D']
```

[28]:
```
for e in encoder2((4, 5, 3, 7, 2, 1, 6)):
    print(e, end = " ; ")
```

```
E ; E ; E ; E ; D ; E ; D ; D ; E ; E ; D ; e ; D ; d ; e ; D ; d ; D ;
```

2 Files

2.1 Tutoriel

On utilise encore la classe *deque* et son constructeur. On ajoute un élément à la fin de la file (donc à droite) par *append* comme pour les piles.

En revanche, pour défiler un élément, on supprime le plus ancien item de le file, c'est à dire le plus à gauche. On utilise alors la méthode *popleft*.

[18]:
```
q = deque()#une file vide
for i in range(5):
    q.append(i)#ajouter $i$ à la finde la file
print(q)
```

```
deque([0, 1, 2, 3, 4])
```

[19]:
```
for i in range(5):
    e = q.popleft()#retirer le premier élément
    print(e)
```

```
print(q)
```

```
0
1
2
3
4
deque([])
```

[20]:
```
#tester la vacuité de la file
isempty(q)
```

[20]:
```
True
```

[21]:
```
q = deque([1,2,3])
#ajouter un élément en début de file (quelle hérésie !!)
q.appendleft(0)
q
```

[21]:
```
deque([0, 1, 2, 3])
```

2.1.1 Exercice : consulter le début de file

Ecrire la fonction

```
def first(q):
```

qui retourne le plus ancien élément de la file q (si il existe) et soulève une exception *IndexError* sinon.

La file n'est pas modifiée.

[23]:
```
q=deque([5,1,6])
first(q),q
```

[23]:
```
(5, deque([5, 1, 6]))
```

2.2 Exercice : nombres de Hamming

Les *nombres de Hamming* sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5 :1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30...

Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (songez que le 1999e entier de Hammingest égal à 8 100 000 000 et le 2000e à 8 153 726 976: il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).

On adopte la démarche suivante : on utilise trois files f_2, f_3, f_5 contenant initialement le nombre 1. La file f_i est censée ne contenir que le nombre 1 ou des multiples de i.

- on détermine le plus petit des trois têtes de file, que l'on note k et que l'on stocke dans une liste résultat

- on retire cet élément des files où il se trouve
- on insère en queue des files f_2, f_3 et f_5 les entiers $2k$, $3k$ et $5k$.

Cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.

Dans ce qui suit seules les méthodes *append* et *popleft* des files et les fonctions *isempty* et *first* sont autorisées.

2.2.1 Q1

Rédiger une fonction

```
def hamming(n):
```

qui retourne la liste des n premiers nombres de Hamming.

[25]:
```
hamming(20)
```

[25]:
```
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24]
```

2.2.2 Q2

L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files.

Par exemple, à la fin de la constitution de la liste des 5 premiers nombres de Hamming, les 3 files contiennent respectivement :

```
deque([6, 8, 10])
deque([6, 9, 12, 15])
deque([10, 15, 20, 25])
```

Modifier votre fonction pour que les files soient sans intersections et justifier la démarche employée.

[27]:
```
hamming2(20)
```

[27]:
```
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24]
```

[ ]:
```

```