Piles

Informatique #03

Exercice 1 — les fonctions de base

Dans tout cet exercice, on manipulera des piles à capacité bornée implémentées à l'aide de listes de longueur `c = 10`.

- **Nouvelle pile –** Écrire une fonction `creer_pile()` qui renvoie une pile vide.
- **Pile vide –** Écrire une fonction `est_vide(p)` qui détecte la présence d'une pile vide.
- **Taille –** Écrire une fonction `taille(p)` qui renvoie la taille de la pile `p`.
- **Empiler –** Écrire une fonction `empiler(p, e)` qui empile l'élément `e` dans la liste `p`.
- **Dépiler –** Écrire une fonction `depiler(p)` qui dépile le sommet de la pile `p` et qui renvoie le sommet en question.
- **Égalité entre deux piles –** Écrire une fonction `egalite(p1, p2)` qui retourne `True` si les deux piles sont identiques, `False` sinon.
- **Couper le sommet d'une pile –** Écrire une fonction `decouper(p, i)` qui retourne une nouvelle pile contenant la liste des i-èmes derniers éléments de `p`.
- **Empiler deux piles –** Écrire une fonction `empiler_les_piles(p1, p2)` qui entasse la pile `p2` à la pile `p1`.
- **Échanger deux éléments d'une pile** (♣) **–** Écrire une fonction `echanger(p, i, j)` qui échange les i-ième et j-ième éléments d'une pile.

```
In [2]: def creer_pile(c):
            p = (c+1)*[None]
            p[0] = 0
            return p
In [3]: print(creer_pile(c))
[0, None, None, None, None, None, None, None, None, None]
In [4]: def est_vide(p):
            return p[0] == 0
In [5]: est_vide([0, None, None])
Out[5]: True
In [6]: def taille(p):
            return p[0]
In [7]: def empiler(p, e):
            n = p[0]
            p[n+1] = e
            p[0] += 1
            return p
In [8]: pile = creer_pile(c)
        empiler(pile, 'a')
        empiler(pile, 'b')
        empiler(pile, 'c')
        print(pile)
[3, 'a', 'b', 'c', None, None, None, None, None, None, None]
In [9]: def depiler(p):
            n = p[0]
            assert n > 0, 'La pile est vide !'
            if n != 0:
                p[0] = n-1
                return p[n]
In [10]: depiler(pile)
Out[10]: 'c'
In [11]: print(pile)
[2, 'a', 'b', 'c', None, None, None, None, None, None, None]
In [12]: def egalite(p1, p2):
             if p1[0] != p2[0]:
                 return False
             for i in range(p1[0]):
                 if depiler(p1) != depiler(p2):
                     return False
             return True
In [13]: egalite([3, 1, 2, 3], [3, 1, 2, 3])
Out[13]: True
In [14]: egalite([3, 1, 2, 3], [3, 2, 2, 3])
Out[14]: False
```

```
In [15]: def decouper(p, i):
             assert i <= p[0], "la pile n'est pas assez longue"
             p2 = creer_pile(c)
             for k in range(i):
                 x = depiler(p)
                 empiler(p2, x)
             return p2
In [16]: pile = [6, 'a', 'b', 'c', 'd', 'e', 'f']
         pile2 = decouper(pile, 3)
         print(pile)
         print(pile2)
[3, 'a', 'b', 'c', 'd', 'e', 'f']
[3, 'f', 'e', 'd', None, None, None, None, None, None, None]
In [17]: def empiler_les_piles(p1, p2):
             for i in range(p2[0]):
                 empiler(p1, depiler(p2))
In [18]: print(pile)
         print(pile2)
         empiler_les_piles(pile, pile2)
         print(pile)
[3, 'a', 'b', 'c', 'd', 'e', 'f']
[3, 'f', 'e', 'd', None, None, None, None, None, None, None]
[6, 'a', 'b', 'c', 'd', 'e', 'f']
In [19]: def echanger(pile, i, j):
             if i > j:
                 i, j = j, i
             pile2 = creer_pile(c)
             for k in range(pile[0]-j):
                 empiler(pile2, depiler(pile))
             x1 = depiler(pile)
             pile3 = creer_pile(c)
             for k in range(pile[0]-i):
                 empiler(pile3, depiler(pile))
             x2 = depiler(pile)
             empiler(pile, x1)
             for k in range(pile3[0]):
                 empiler(pile, depiler(pile3))
             empiler(pile, x2)
             for k in range(pile2[0]):
                 empiler(pile, depiler(pile2))
```

Exercice 2 — battre les cartes

Écrire une fonction `melanger(p1, p2)` qui mélange les éléments de `p1` et `p2` dans une troisième pile de la façon suivante : tant qu'une pile (au moins) n'est pas vide, on retire aléatoirement un élément au sommet d'une des deux piles et on l'empile sur la pile résultat.

```
In [21]: from random import *

         def melanger(p1, p2):
             p3 = creer_pile(c)
             n = p1[0] + p2[0]
             while p3[0] != n:
                 k = randint(1,2)
                 if k == 1 and p1[0] != 0:
                     empiler(p3, depiler(p1))
                 if k == 2 and p2[0] != 0:
                     empiler(p3, depiler(p2))
             return p3
In [22]: pile1 = [5, 1, 2, 3, 4, 5]
         pile2 = [4, 'a', 'b', 'c', 'd']
         melanger(pile1, pile2)
Out[22]: [9, 'd', 5, 'c', 'b', 4, 3, 2, 'a', 1, None]
```

Exercice 3 — conversion base 10 / base 2

Écrire une fonction `conversion(nb10)` qui retourne la représentation en base 2 du nombre `nb10` représenté en base 10 à l'aide de piles.

```
In [23]: def conversion(nb10):
             p = creer_pile(c)
             while nb10 != 0:
                 empiler(p, nb10%2)
                 nb10 = nb10 // 2
             res = ''
             for i in range(p[0]):
                 res += str(depiler(p))
             return int(res)
In [24]: conversion(27)
Out[24]: 11011
```

Exercice 4 — mots bien parenthésés (♣)

1. Écrire une fonction `parenthese(mot)` qui vérifie que la chaîne de caractères `mot`constituée de caractères (et) est bien parenthésée.
2. Généraliser la fonction précédente en une fonction `parenthese_g(mot)` qui vérifier le bon parenthésage d'une chaîne de caractères constituée de parenthèses, d'accolades et de crochets. On pourra écrire une fonction intermédiaire `inv(symbole)` qui prend en argument une parenthèse ouvrante, une accolade ouvrante ou un crochet ouvrant et qui retourne le symbole fermant correspondant.

```
In [25]: def parenthese(mot):
             p = creer_pile(c)
             for i in range(len(mot)):
                 if mot[i] == '(':
                     empiler(p, '(')
                 else:
                     if est_vide(p):
                         return False
                     depiler(p)
             return est_vide(p)
In [26]: parenthese('(()(()')
Out[26]: True

In [27]: def inv(symbole):
             s1 = '({['
             s2 = ')}]'
             return s2[s1.index(symbole)]

     def parentheseg(mot):
             p = creer_pile(c)
             for i in range(len(mot)):
                 if mot[i] in '({[':
                     empiler(p, mot[i])
                 else:
                     if est_vide(p):
                         return False
                     a = depiler(p)
                     if mot[i] != inv(a):
                         return False
             return est_vide(p)
```

```
In [28]: parentheseg('[()](){(()()}')
Out[28]: True
In [29]: parentheseg('[()(){()]()}')
Out[29]: False
```

Exercice 5 — évaluer une expression en notation polonaise inversée

Écrire une fonction `eval_polonaise(exp)` qui permet d'évaluer une expression en notation polonaise inversée, ceci à l'aide d'une pile. (cf. les explications au tableau)

```
In [30]: def eval_polonaise(exp):
             p = creer_pile(c)
             for e in exp:
                 if e == '+' or e == '*':
                     x = int(depiler(p))
                     y = int(depiler(p))
                     if e == '+':
                         z = x + y
                     else:
                         z = x * y
                     empiler(p, z)
                 else:
                     empiler(p, e)
             return depiler(p)
In [31]: eval_polonaise('2365*+*')
Out[31]: 66
```