Chapitre 1 : Structures de données avancées

Les Piles ~ Corrigé 1.1

Exercice 1 :

Solution 1 : Mettre en œuvre une pile avec une liste initialisée avec 40 valeurs égales à **None** .

```
1.    #pile.py
2.
3.    NMAX = 40
4.
5.    def creer_pile():
6.        return [None]*NMAX #2
7.
8.    def pile_vide(p):
9.        return p.count(None) == NMAX
10.
11.   def pile_pleine(p):
12.       return p.count(None) == 0
13.
14.   def empiler(p,x):
15.       if not pile_pleine(p):
16.           p[p.index(None)]=x
17.       else:
18.           raise Exception("Pile saturée")
19.
20.   def depiler(p):
21.       assert not pile_vide(p), "Pile vide"
22.       pos =  -1 if pile_pleine(p) else L.index(None)-1
23.       elt = p[pos]
24.       p[pos] = None
25.       return elt
26.
27.   def sommet(p):
28.       assert not pile_vide(p), "Pile vide"
29.       return p[-1] if pile_pleine(p) else p[L.index(None)-1]
30.
31.   def afficher(p):
32.       p1 = copy(p)
33.       while not pile_vide(p1):
34.           print(depiler(p1))
35.
36.   def depilerKelt(p,k):
37.       while not pile_vide(p) and k>0:
38.           depiler(p)
39.           k-=1
40.
41.
42.   def deplacer(p1,p2):
43.       while not pile_vide(p1):
44.           empiler(p2,depiler(p1))
45.
46.
47.
```

```
48.   def inverser_pile(p):
49.       p1,p2 = creer_pile(),creer_pile()
50.       deplacer(p,p1)
51.       deplacer(p1,p2)
52.       deplacer(p2,p)
53.
54.
55.   def depilerJusqua(p,e):
56.       while not est_vide(p) and sommet(p) != e:
57.           depiler(p)
58.
59.   def appartient(p,x):
60.       p1 = p.copy()
61.       depilerJusqua(p1,x)
62.       return False if pile_vide(p1) else True
```

Solution 2 : Mettre en œuvre une pile avec une liste initialement vide.

```
1.    #pile.py
2.
3.    NMAX = 40
4.    from copy import copy,deepcopy
5.
6.    def creer_pile():
7.        return []
8.
9.    def taille(p):
10.       return len(p)
11.
12.   def pile_vide(p):
13.       return taille(p) == 0
14.
15.   def empiler(p,x):
16.       if len(p) < 40:
17.           p.append(x)
18.       else:
19.           #raise : déclencher une erreur
20.           raise Exception ("Erreur: Pile saturée")
21.
22.   def empiler(p, x):
23.       assert taille(p)< NMAX, "Pile pleine"
24.       p.append(x)
25.
26.   def depiler(p):
27.       assert not pile_vide(p), "Erreur : pile vide"
28.       return p.pop() #p.pop(-1)
29.
30.   def sommet(p):
31.       assert not pile_vide(p), "Erreur: Pile vide"
32.       return p[-1]
33.
34.
35.
```

```
36.   #Solution 1
37.   def afficher(p):
38.       p1 = p.copy()
39.       while not pile_vide(p1): print(depiler(p1))
40.
41.   #Solution 2
42.   def afficher(p) :
43.       temp = creer_pile()
44.
45.       while not est_vide(p):
46.           elt = depiler(p)
47.           print(elt)
48.           empiler(temp,elt)
49.
50.       while not est_vide(temp):
51.           empiler(p,depiler(temp))
52.
53.   #Solution 1
54.   def depilerKelt(p,k):
55.       for i in range(k):
56.           if taille(p) > 0:
57.                depiler(p)
58.           else:
59.               break
60.
61.   #Solution 2
62.   def depilerKelt(p,k):
63.       if k>= len(p):
64.           p = creer_pile() #[]
65.       else:
66.           for i in range(k):
67.               depiler(p)
68.
69.
70.   #solution 3
71.   def depilerKelt(p,k):
72.       while not pile_vide(p) and k>0:
73.           depiler(p)
74.           k-=1
75.
76.   #Solution 1
77.   def inverser_pile(p):
78.       p1,p2 = creer_pile(),creer_pile()
79.
80.       while not pile_vide(p):
81.           empiler(p1,depiler(p))
82.
83.       while not pile_vide(p1):
84.           empiler(p2,depiler(p1))
85.
86.       while not pile_vide(p2):
87.           empiler(p,depiler(p2))
88.
89.
90.
91.
```

```
92.   #Solution 2
93.   def deplacer(p1,p2):
94.       while not pile_vide(p1):
95.           empiler(p2,depiler(p1))
96.
97.
98.   def inverser_pile(p):
99.       p1,p2 = creer_pile(),creer_pile()
100.      deplacer(p,p1)
101.      deplacer(p1,p2)
102.      deplacer(p2,p)
103.
104.  #Solution 3
105.  def inverser_pile(p):
106.      p1 = deepcopy(p)
107.      p = creer_pile()
108.      while not pile_vide(p1):
109.          empiler(p,depiler(p1))
110.
111.  #Solution 1
112.  def depilerJusqua(p,e):
113.      while not est_vide(p) and sommet(p) != e:
114.          depiler(p)
115.
116.  #Solution 2
117.  def depilerJusqua(p,x):
118.      if appartient(p,x):
119.          while sommet(p) != x:
120.              depiler(p)
121.      else:
122.          while not pile_vide(p):
123.              depiler(p)
124.
125.  #Solution 1
126.  def appartient(p,elt) :
127.      p1 = creer_pile()
128.      while not est_vide(p) and sommet(p) != elt:
129.          empiler(p1,depiler(p))
130.      if est_vide(p):
131.          resultat = False
132.      else:
133.          resultat = True
134.      while not est_vide(p1):
135.          empiler(p,depiler(p1))
136.      return resultat
137.
138.  #Solution 2
139.  def appartient(p,x):
140.      p1 = copy(p)
141.      while not pile_vide(p1) and sommet(p1)!=x:
142.          depiler(p1)
143.      if pile_vide(p1):
144.          return False
145.      return True
146.
147.
```

```
148.  #Solution 2
149.  def appartient(p,x):
150.      p1 = p.copy()
151.      depilerJusqua(p1,x)
152.      return False if pile_vide(p1) else True
```

Exercice 2 :

```
1.   from pile import *
2.
3.   def conversion(nb10):
4.
5.       res = '0b' if nb10 else '0b0
6.
7.       p = creer_pile()
8.       while nb10 != 0:
9.           empiler(p, nb10%2)
10.          nb10 = nb10 // 2
11.
12.      while not est_vide(p):
13.          res += str(depiler(p))
14.
15.      return res
16.
17.  #test
18.  print(conversion(27))
```

Exercice 3 :

Question 1 : Analyser si une expression arithmétique est bien parenthésée.

```
1.    from pile import *
2.
3.    def analyse_parenthese(ch) :
4.        p = creer_pile()
5.        for c in ch :
6.            if c == "(" :
7.                empiler(p,c)
8.            elif c == ")" :
9.                if est_vide(p):
10.                   print("Il manque une ou des parenthèse(s) ouvrante(s)")
11.                   return False
12.               else :
13.                   depiler(p)
14.       if est_vide(p):
15.           print("Placement des parenthèses correct")
16.           return True
17.       else :
18.           print("Il manque une ou des parenthèses fermantes")
19.           return False
20.
21.
22.   #test
```

```
23.   ch="(3+2))*2"
24.   print(ch)
25.   analyse_parenthese(ch)
26.
27.   ch="((3+2)*2"
28.   print(ch)
29.   analyse_parenthese(ch)
30.
31.   ch="(3+2)*2"
32.   print(ch)
33.   analyse_parenthese(ch)
```

Question 2 : Donner les trois représentations d'une expression arithmétique :

Expression 1 :

La représentation **infixée** : **2 + (3 - (5 + 1) × 2)**

La représentation **préfixée** : **+ 2 - 3 × + 5 1 2**

La représentation **postfixée** : **2 3 5 1 + 2 × - +**

Expression 2 :

La représentation infixée : **/ 2 × - 3 + 4 5 6**

La représentation préfixée : **2 / ((3 - (4 + 5)) × 6)**

La représentation postfixée : **2 3 4 5 + - 6 × /**

Question 3 : Calculer la valeur numérique d'une expression arithmétique.

```
1.    from pile import *
2.    import string
3.
4.    def calc_exp_arith(ch):
5.
6.        p = creer_pile()
7.
8.        for i in range(len(ch)):
9.            if ch[i] in string.digits :
10.               empiler(p, int(ch[i]))
11.           else:
12.               n1 = depiler(p)
13.               n2 = depiler(p)
14.               if   ch[i] == '+' : empiler(p, n1 + n2)
15.               elif ch[i] == '-' : empiler(p, n2 - n1)
16.               elif ch[i] == '*' : empiler(p, n1 * n2)
17.               elif ch[i] == '/' : empiler(p, n2 / n1)
18.
19.       return depiler(p)
20.
21.   #test
22.   print(calc_exp_arith("27*5+"))
```