

TP03

November 8, 2021

1 Premiers exemples de boucles imbriquées

On a vu dans les TP précédents comment on pouvait utiliser des boucles pour effectuer un traitement. La plupart du temps les boucles ont une structure simple, elles consistent soit à parcourir un tableau ou un range et à effectuer un traitement pour chacun de ses éléments

```
s = 0
for x in tableau:
    print(x)
    s = s + x
```

soit à répéter un traitement tant qu'une condition n'est pas remplie :

```
q = 0
while x >= y:
    q = q + 1
    x = x - y
```

Dans ce TP, on va rajouter un peu de complexité en considérant des boucles imbriquées, c'est-à-dire que dans le corps d'une boucle on pourra effectuer une autre boucle. Pour ce TP, on n'ira pas plus loin que deux boucles imbriquées. Prenons un exemple simple. Si on veut calculer $\sum_{i=1}^n \sum_{j=1}^n i + j$ on pourra écrire :

```
s = 0
for i in range(1, n+1):
    for j in range(1, n+1):
        s = s + i + j
```

On sait qu'une boucle

```
for i in range(1, 4+1):
    s = s + i
```

revient à effectuer pour les 4 valeurs de i le corps :

```
s = s + 1
s = s + 2
s = s + 3
s = s + 4
```

Ici,

```

for i in range(1, 4+1):
    for j in range(1, 3+1):
        s = s + i + j

```

revient donc à écrire :

```

for j in range(1, 3+1):
    s = s + 1 + j
for j in range(1, 3+1):
    s = s + 2 + j
for j in range(1, 3+1):
    s = s + 3 + j
for j in range(1, 3+1):
    s = s + 4 + j

```

mais chacune de ces boucles peut elle-même être déroulée et correspond à effectuer :

s = + 1 + 1	s = s + 1 + 2	s = s + 1 + 3
s = + 2 + 1	s = s + 2 + 2	s = s + 2 + 3
s = + 3 + 1	s = s + 3 + 2	s = s + 3 + 3
s = + 4 + 1	s = s + 4 + 2	s = s + 4 + 3

On a placé ici les instructions en ligne également pour faire ressortir le fait que ces deux boucles imbriquées correspondent naturellement à une multiplication des cas. Ainsi deux boucles imbriquées de n itérations effectuant un traitement en temps constant seront dites en temps quadratique, ou encore en n^2 .

Pour la question suivante, on demande de programmer les sommes :

- $S_1 = \sum_{i=1}^n \sum_{j=1}^n ij$
- $S_2 = \sum_{i=1}^n \prod_{j=1}^n i + j$
- $S_3 = \sum_{1 \leq i \leq j \leq n} ij$

```

[1]: def somme_1(n):
      # BEGIN SOLUTION
      s = 0
      for i in range(1,n+1):
          for j in range(1,n+1):
              s = s + i * j
      return s
      # END SOLUTION

```

```

[2]: from civ import *
      init()
      test(somme_1, 4, 100)
      test(somme_1, 0, 0)
      test(somme_1, 1, 1)
      valide()

```

OK!: somme_1(4) vaut 100

OK!: somme_1(0) vaut 0

OK!: somme_1(1) vaut 1
Tous les tests sont valides

```
[3]: def somme_2(n):  
    # BEGIN SOLUTION  
    s = 0  
    for i in range(1,n+1):  
        p = 1  
        for j in range(1,n+1):  
            p = p * (i + j)  
        s = s + p  
    return s  
    # END SOLUTION
```

```
[4]: from civ import *  
init()  
test(somme_2, 4, 3000)  
test(somme_2, 0, 0)  
test(somme_2, 1, 2)  
valide()
```

OK!: somme_2(4) vaut 3000
OK!: somme_2(0) vaut 0
OK!: somme_2(1) vaut 2
Tous les tests sont valides

```
[5]: def somme_3(n):  
    # BEGIN SOLUTION  
    s = 0  
    for i in range(1,n+1):  
        for j in range(i,n+1):  
            s = s + i * j  
    return s  
    # END SOLUTION
```

```
[9]: from civ import *  
init()  
test(somme_3, 4, 65)  
test(somme_3, 0, 0)  
test(somme_3, 1, 1)  
valide()
```

OK!: somme_3(4) vaut 65
OK!: somme_3(0) vaut 0
OK!: somme_3(1) vaut 1
Tous les tests sont valides

2 Recherche de motif dans un texte

On va maintenant étudier un problème très classique dans l'étude des textes : la recherche de motif. On considère un texte sous la forme d'une chaîne de caractères s , ce qui ne va rien changer pour nous par rapport à un tableau car on ne va faire que lire, et on cherche la présence d'un motif, ou mot, m dans cette chaîne.

L'algorithme qu'on va employer est celui de la recherche naïve : à chaque position dans s on regarde si on arrive à retrouver le motif m . Quand on l'a trouvé, on s'arrête avec un succès. Sinon, on le cherche à la position suivante.

Voici un exemple en cherchant "toto" dans "totatotou" :

indices	0	1	2	3	4	5	6	7	8
s	t	o	t	a	t	o	t	o	u
recherche à l'indice 0	t	o	t	o					
à l'indice 1		t	o	t	o				
à l'indice 2			t	o	t	o			
à l'indice 3				t	o	t	o		
à l'indice 4					t	o	t	o	
					motif trouvé à l'indice 4				

```
[6]: def recherche_naive(motif, chaine):  
    '''Recherche le motif dans la chaîne et renvoie l'indice de  
    la première occurrence du motif s'il est présent ou None sinon.'''  
    # BEGIN SOLUTION  
    n, p = len(chaine), len(motif)  
    for i in range(n-p+1):  
        present = True  
        for j in range(p):  
            if chaine[i+j] != motif[j]:  
                present = False  
                break  
        if present:  
            return i  
    return None  
    # END SOLUTION
```

```
[7]: from civ import *  
s = "totatotou"  
init()  
test(recherche_naive, "toto", s, 4)  
test(recherche_naive, "to", s, 0)  
test(recherche_naive, "totou", s, 4)
```

```
test(recherche_naive, "titi", s, None)
test(recherche_naive, s+"_", s, None)
valide()
```

OK!: recherche_naive('toto', 'totatotou') vaut 4
 OK!: recherche_naive('to', 'totatotou') vaut 0
 OK!: recherche_naive('totou', 'totatotou') vaut 4
 OK!: recherche_naive('titi', 'totatotou') vaut None
 OK!: recherche_naive('totatotou_', 'totatotou') vaut None
 Tous les tests sont valides

3 Recherche des deux valeurs les plus proches

On considère un tableau t et on cherche $i \neq j$ tels que $\text{abs}(t[i]-t[j])$ soit minimale. C'est à dire qu'on cherche les deux valeurs les plus proches dans le tableau. On rappelle en effet que $|x - y|$ est la distance dans $\mathbb{R}$ entre x et y .

```
[8]: def recherche_proches(t):
    '''Renvoie (i,j) où  $i < j$  tels que  $\text{abs}(t[i]-t[j])$  soit minimale. Si jamais
    →  $t$  contient
    moins de deux éléments, on renverra un erreur avec raise ValueError
    S'il y a plusieurs solutions, on renverra celle où  $i$  est minimal.
    '''
    # BEGIN SOLUTION
    if len(t) < 2:
        raise ValueError
    m = abs(t[1]-t[0])
    mc = (0, 1)
    for i in range(len(t)):
        for j in range(i+1, len(t)):
            v = abs(t[i]-t[j])
            if v < m:
                m = v
                mc = (i, j)
    return mc
    # END SOLUTION
```

```
[9]: from civ import *
init()
test(recherche_proches, [0,3,2,1], (0,3))
test(recherche_proches, [0,10,9,8], (1,2))
test_erreur(recherche_proches, [0], ValueError)
valide()
```

OK!: recherche_proches([0, 3, 2, 1]) vaut (0, 3)
 OK!: recherche_proches([0, 10, 9, 8]) vaut (1, 2)
 OK!: recherche_proches([0]) a bien fait l'erreur ValueError

Tous les tests sont valides

4 Tri à bulles

On va étudier ici un algorithme de tri très simple : le tri à bulles. Ce tri consiste à identifier des éléments consécutifs dans un tableau qui ne sont pas rangés et à les inverser. C'est-à-dire que si $t[i] > t[i+1]$ on échange les valeurs dans $t[i]$ et $t[i+1]$.

Remarque à ce propos, on a vu comment faire un tel échange avec une variable temporaire, mais en Python, il y a un mécanisme d'affectation des couples qui permet de le faire en une instruction $t[i], t[i+1] = t[i+1], t[i]$.

Résoudre une telle inversion peut être vue comme une bulle qui remonterait progressivement dans un verre d'une boisson gazeuse. C'est cela qui a donné son nom à ce tri. Pour trier, on va donc réaliser des inversions en faisant progresser l'indice i . A la suite d'un balayage on n'a aucune raison d'avoir trié le tableau, on va donc itérer cela jusqu'à ce que le tableau soit trié.

Voici en animation un tri par tri à bulle. Les éléments du tableau sont représentés par des points du plan de coordonnées $(i, t[i])$.

Une première approche est de faire un balayage de plus et si on ne trouve rien à inverser, c'est que le tableau est trié.

```
[10]: def tri_a_bulles_test_inversion(t):  
    '''trie t, attention, cette fonction ne renvoie rien'''  
    # BEGIN SOLUTION  
    a_inverse = True  
    while a_inverse:  
        a_inverse = False  
        for i in range(len(t)-1):  
            if t[i] > t[i+1]:  
                a_inverse = True  
                t[i], t[i+1] = t[i+1], t[i]  
    # END SOLUTION
```

```
[11]: from civ import *  
init()  
def est_trie(t):  
    ts = t[:]  
    ts.sort()  
    if ts != t:  
        print('Echec', t, '!=', ts)  
        return False  
    else:  
        return True  
test_condition(tri_a_bulles_test_inversion, est_trie, [4,5,6,1,3])  
test_condition(tri_a_bulles_test_inversion, est_trie, [5,4,3,2])  
test_condition(tri_a_bulles_test_inversion, est_trie, ['z','a','c'])  
valide()
```

OK!: tri_a_bulles_test_inversion([4, 5, 6, 1, 3]) a fait ce qu'on attendait
OK!: tri_a_bulles_test_inversion([5, 4, 3, 2]) a fait ce qu'on attendait
OK!: tri_a_bulles_test_inversion(['z', 'a', 'c']) a fait ce qu'on attendait
Tous les tests sont valides

On dit qu'un couple d'indices (i, j) où $i < j$ est une **inversion** pour t si $t[i] > t[j]$.

Montrer qu'à l'issue du premier balayage, le dernier élément (le maximum) est nécessairement à la bonne position. Que peut-on dire à l'issue de k balayages ? En déduire une version sans **while** et juste avec deux **for** imbriqués du tri à bulles.

```
[12]: def tri_a_bulles(t):  
    '''trie t, attention, cette fonction ne renvoie rien'''  
    # BEGIN SOLUTION  
    for k in reversed(range(len(t))):  
        a_inverse = False  
        for i in range(k):  
            if t[i] > t[i+1]:  
                a_inverse = True  
                t[i], t[i+1] = t[i+1], t[i]  
    # END SOLUTION
```

```
[13]: from civ import *  
init()  
def est_trie(t):  
    ts = t[:]  
    ts.sort()  
    if ts != t:  
        print('Echec', t, '!=', ts)  
        return False  
    else:  
        return True  
test_condition(tri_a_bulles, est_trie, [4,5,6,1,3])  
test_condition(tri_a_bulles, est_trie, [5,4,3,2])  
test_condition(tri_a_bulles, est_trie, ['z','a','c'])  
valide()
```

OK!: tri_a_bulles([4, 5, 6, 1, 3]) a fait ce qu'on attendait
OK!: tri_a_bulles([5, 4, 3, 2]) a fait ce qu'on attendait
OK!: tri_a_bulles(['z', 'a', 'c']) a fait ce qu'on attendait
Tous les tests sont valides

```
[ ]:
```