Informatique — MPSI/PCSI, 2024–2025

Correction du TP n° 2

1 Présentation sommaire des structures séquentielles

2 Recherche séquentielle dans un tableau unidimensionnel et variations

Exercice 2.1 (*Des tableaux/listes-jouets*)

1. On notera les différentes façons de définir de nouvelles listes.

```
t0 = [42] * 15                  # À éviter
t00 = [42 for i in range(15)]  # À privilégier
t000 = [42 for _ in range(15)] # Possible également

t1 = [10, 30, 42, 2, 17, 5, 30, -20]

t2 = []
for i in range(-3, 6):
    t2.append(i**2)

t21 = [i**2 for i in range(-3,6)]

t3 = []
for i in range(1000):
    if (i%5) in [0, 2, 4]:
        t3.append(i**3)

t31 = [i**3 for i in range(1000) if (i%5) in [0,2,4]]

t4 = [843.0]
for i in range(20):
    t4.append(t4[i]/3+28)
```

2. Pour faire afficher la longueur de chacune des listes, le plus simple est d'utiliser une boucle. Par exemple :

```
for t in [t0,t00,t00,t1,t2,t21,t3,t31,t4]:
    print(len(t))
```

Sortie obtenue :

```
15
15
15
8
9
9
600
600
21
```

Exercice 2.2

1. La liste est parcourue avec ses indices ou directement par ses éléments. Il suffit ensuite de tester l'appartenance de l'élément `x` au tableau `t` à l'aide d'un branchement conditionnel. Utilisation d'une boucle `for` :

```
def appartient_1(x, t):
    for e in t:
        if e==x:
            return(True)
    # si on arrive là, x n'appartient au tableau t
    return(False)
```

ou d'une boucle `while` :

```
def appartient_2(x, t):
    i = 0
    while i<len(t):
        if t[i]==x:
            return(True)
        i+=1
    return(False)
```

Une autre possibilité... Bien comprendre l'utilisation du `and` (l'ordre des deux conditions est-il indifférent ?) et le `return(i<len(t))`.

```
def appartient_3(x, t):
    i=0
    while i<len(t) and not t[i]==x: # ou t[i] != x à la place de not...
        i+=1
    return(i<len(t))
```

2. Dans le pire des cas (`x` apparaît une seule fois en dernière position), le nombre d'accès au tableau effectués pour tester l'appartenance de `x` à `t` est égal à $2n$ où n est la longueur du tableau (accès au tableau lors de chaque itération de la boucle et lors du test logique). Il s'agit d'une complexité linéaire.

Exercice 2.3

Même principe qu'à l'exercice 2.2, mais il faut, en plus, créer une liste vide et lui ajouter les indices des éléments du tableau égaux à `x`.

```
def positions(x,t):
    '''retourne la liste des positions de x dans t'''
    L = []
    for indice in range(len(t)):
        if t[indice] == x:
            L.append(indice)
    return(L)
```

Exercice 2.4

1. Plusieurs implémentations possibles. Deux exemples parmi d'autres ci-dessous. Attention à l'initialisation de la variable `max`. Le premier élément du tableau `tab[0]` est généralement utilisé. Avec un parcours par valeur (élément) :

```
def maximum(tab):
    """retourne le maximum d'un tableau"""
    if tab == []: #si la liste est vide
        return(None)
    maxi = tab[0]
    for terme in tab[1:]:
        if terme > maxi:
            maxi = terme
    return(maxi)
```

ou avec un parcours par indice :

```
def maximum2(tab):
    """retourne le maximum d'un tableau"""
    if tab == []: #si la liste est vide
```

```
        return(None)
    maxi = tab[0]
    for k in range(1, len(tab)):
        if tab[k] > maxi:
            maxi = tab[k]
    return(m)
```

2. Même si le maximum est « pris » plusieurs fois, la fonction ne renvoie qu'une seule position (la première ou la dernière de façon « naturelle »). Ce comportement peut-être modifié, par exemple en renvoyant la liste des positions où le maximum est atteint et la valeur de ce maximum.

```
def liste_position_maximum(tab):
    """retourne le maximum et la liste des positions où il est atteint"""
    tmaxi, maxi = [0],tab[0]
    for indice in range(1, len(tab)):
        terme = tab[indice]
        if terme > maxi:
            tmaxi, maxi = [indice], terme
        elif terme == maxi:
            tmaxi.append(indice)
    return(tmaxi, maxi)
```

que l'on peut simplifier sachant que la valeur du maximum est celle deu premier élément du tableau `tmaxi`. Il vient alors :

```
def liste_positions_maximum(tab):
    """retourne seulement la liste des positions où il est atteint"""
    tmaxi = [tab[0]]
    for indice in range(1, len(tab)):
        if tab[indice] > tmaxi[0]:
            tmaxi = [indice]
        elif tab[indice] == tmaxi[0]:
            tmaxi.append(indice)
    return(tmaxi)
```

Exercice 2.5

1. Sachant que la somme cumulée des éléments d'un tableau $(t_k)_{0\leqslant k\leqslant n-1}$ est définie par :

$$\forall k \in [\![0, n-1]\!], \quad s_k = \sum_{j=0}^{k} t_j$$

initialisée à $s_0 = t_0$. D'où,

$$\forall k \in [\![1, n-1]\!], \quad s_k = s_{k-1} + t_k$$

En utilisant une seule variable (le tableau `cumul`) pour stocker la somme cumulée au cours du parcours du tableau `tab`, il vient l'implémentation suivante :

```
def sommes_cumulees(tab):
    """retourne les sommes cumulées de tab depuis le premier terme"""
    cumul = [tab[0]]
    for terme in tab[1:]:
        cumul.append(cumul[-1]+terme)
    return(cumul)
```

2. Le nombre d'additions réalisées est ici égal à $n-1$ (complexité linéaire).

Exercice 2.6

1. Une méthode à deux candidats est utilisée, par analogie avec la méthode à un candidat de l'exercice 2.4. Les implémentations suivantes conviennent :

```
def deux_grands(t):
    """retourne les deux + grand éléments d'un tableau de longueur>=2"""
```

```
    a=t[0]# le plus grand
    b=t[1]
    if a>b:
        a,b=b,a
    for e in t[2:]:
        if e>a:
            b=a
            a=e
        elif e>b and e<a: # on a e<=a vrai, mais pas le strict
            b=e
    return(a,b)
```

2. La fonction précédente renvoie bien les deux plus grands éléments du tableau... au sens strict ! Il faut donc préciser si la fonction doit renvoyer les deux plus grands éléments en envisageant les inégalités au sens strict ou au sens large. Les implémentations suivantes conviennent pour obtenir les deux plus grands éléments au sens large.

```
def deux_grands_bis(t):
    """retourne les deux plus grands éléments d'un tableau de longueur >=2
    au sens large"""
    a=t[0]# le plus grand
    b=t[1]
    if a>b:
        a,b=b,a
    for e in t[2:]:
        if e>a:
            b=a
            a=e
        elif e>b:
            b=e
    return(a,b)
```

3 Listes et dictionnaires

Exercice 2.7 (*Dépouillement d'une urne*)

1. Importer la fonction `urne` du fichier `Info-TP02_cadeau.py` qui permet de générer le contenu d'une urne de taille variable, avec un nombre de candidats aléatoires.

```
import random as r

def urne(n):
    cand = ['Aurore', 'Bernard', 'Caroline', 'Denis', 'Eléonore', 'Fanny', 'Grégoire',
    'Henri', 'Isabelle', 'Jade', 'Karim', 'Laurence', 'Marine', 'Nathan', 'Océane',
    'Philippine', 'Quentin', 'Rose', 'Sabine', 'Triphon', 'Ulysse', 'William', 'Yann',
    'Zoé']
    candr = []
    nbcand = r.randint(2,26)
    for i in range(nbcand):
        num = r.randint(0,len(cand)-1)
        candr.append(cand[num])
        cand.pop(num)
    return([candr[r.randint(0,nbcand-1)] for i in range(n)])
```

Exemple de sortie obtenue :

```
>>> u = urne(25)
>>> u
['Grégoire', 'Bernard', 'Rose', 'Grégoire', 'Laurence', 'Denis', 'Aurore', 'Ulysse',
'Bernard', 'Grégoire', 'Bernard', 'Rose', 'Denis', 'Denis', 'Grégoire', 'Aurore',
'Grégoire', 'Laurence', 'Laurence', 'Laurence', 'Grégoire', 'Bernard', 'Denis',
'Bernard', 'Rose']
```

2. Le plus pratique pour dépouiller l'urne est d'utiliser un dictionnaire qui, à chaque candidat, associera son nombre de voix. Notons qu'un des avantages d'un dictionnaire est qu'il n'y a pas besoin de savoir à l'avance qui sont les candidats, ni même combien il y en a. La fonction doit incrémenter la valeur associée (le nombre de voix) à une clef (le nom du ou de la candidat·e) si elle existe déjà, ou de l'initialiser à 1 dans le cas contraire.

```
def depouillement(urne):
    d = {}
    for nom in urne:
        if nom in d: # détermine si le nom du ou de la candidat.e est déjà une clé
du dictionnaire
            d[nom] = d[nom] + 1
        else:
            d[nom] = 1 # Initialisation à 1
    return(d)
```

Résultat obtenu lors de l'exécution sur l'exemple de la question précédente :

```
>>> depouillement(u)
{'Grégoire': 6, 'Bernard': 5, 'Rose': 3, 'Laurence': 4, 'Denis': 4, 'Aurore': 2,
'Ulysse': 1}
```

3. Parcours du dictionnaire par ses clés pour déterminer le vainqueur. Le candidat `vmax` est initialisé à 0. *Remarque :* le gagnant est supposé unique...

```
def vainqueur(D):
    vmax = 0
    for cand in D:
        if D[cand] > vmax:
            vmax = D[cand]
            gagnant = cand
    return(gagnant)
```

4 Algorithmes opérant par boucles imbriquées

Exercice 2.8

Il est possible de parcourir l'ensemble des indices i et j tels que $(i, j) \in [\![1, n]\!]^2$ où n est la longueur de la liste puis de tester les indices qui conviennent. Néanmoins, une écriture judicieuse des boucles permet de s'en affranchir... Pour le reste, il s'agit d'une méthode à un candidat. Attention toutefois de bien initialiser le couple qui convient (et de le faire varier avec le candidat `min`).

```
def proches(t):
    mini = abs(t[1]-t[0])
    c = (1,0)
    for j in range(len(t)-1):
        for i in range(j+1,len(t)):
            test = abs(t[i]-t[j])
            if test <= mini:
                mini = test
                c = (i,j)
    return(c)
```

Exercice 2.9 (*Recherche d'une sous-liste ou d'un sous-mot*)

1. La fonction peut immédiatement renvoyer `False` si la longueur du sous-mot candidat est supérieure à celle du mot entier. Dans le cas contraire, il faut comparer les deux chaînes caractère à caractère à partir d'une position d'une position variable p du mot `m2`. L'index/indice i est incrémenté tant que l'ensemble des caractères de `m1` n'a pas été testé et que les caractères des deux mots coïncident. Dès qu'un caractère ne correspond pas, on peut passer à la position p suivante. Si l'ensemble caractères de `m1` coïncident avec ceux de `m2` (en débutant à la position p), la fonction renvoie `True`. Dans le cas contraire, si cela ne se produit pour aucune position p, la fonction revoie `False`.

```
def ss_mot(m1,m2):
    lg1, lg2 = len(m1),len(m2)
    if lg1 > lg2:
        return(False)
    else :
        for p in range (lg2-lg1+1): # inutile d'aller plus loin (pourquoi ?)
            i = 0
            while i < lg1 and m1[i] == m2[p+i]:
                i += 1
            if i == lg1:
                return(True)
        return(False)
```

2. Tests :

```
print(ss_mot("pou", "pouettagada"))
print(ss_mot("taga", "pouettagada"))
print(ss_mot("da", "pouettagada"))
print(ss_mot("toto", "pouettagada"))
```

```
True
True
True
False
```

3. Test de la fonction avec des listes :

```
print(ss_mot([10,30], t1))
print(ss_mot([0,1,4], t21))
print(ss_mot([17, 5, 30, -20], t1))
print(ss_mot([9,0,6], t2))
```

```
True
True
True
False
```

4. Même principe qu'à l'exercice 2.3

```
def positions_sous_mot(m1,m2):
    lg1, lg2 = len(m1),len(m2)
    if lg1 > lg2:
        return(False)
    else :
        pos = []
        for p in range (lg2-lg1+1): # inutile d'aller plus loin (pourquoi ?)
            i = 0
            while i < lg1 and m1[i] == m2[p+i]:
                i += 1
            if i == lg1:
                pos.append(p)
        return(pos)
```

5. L'adaptation des fonctions précédentes conduit aux implémentations :

```
def ss_mot(m1,m2):
    lg1, lg2 = len(m1),len(m2)
    if lg1 > lg2:
        return(False)
    else :
        for p in range (lg2-lg1+1): # inutile d'aller plus loin (pourquoi ?)
            if m1==m2[p:p+lg1]:
                return(True)
```

```
        return(False)

def positions_sous_mot(m1,m2):
    lg1, lg2 = len(m1),len(m2)
    if lg1 > lg2:
        return(False)
    else :
        pos = []
        for p in range (lg2-lg1+1): # inutile d'aller plus loin (pourquoi ?)
            if m1==m2[p:p+lg1]:
                pos.append(p)
        return(pos)
```

Exercice 2.10 (*Une petite application de la recherche d'un sous-mot...*)

1. L'implémentation suivante convient : *Rappel :* la méthode `append()` ne s'applique pas aux chaînes de caractères.

```
import random as r

def chaine_aleatoire(n):
    A="ACGT"
    s=""
    for k in range(n):
        s+=A[r.randint(0,3)]
    return(s)
```

2. Après implémentation d'une fonction moyenne, pas de problème particulier en utilisant la fonction `position_sous_mot(m1,m2)` implémentée à l'exercice précédent.

```
def moyenne(L):
    s=0
    for e in L:
        s+=e
    return(s/len(L))

print(moyenne([len(positions_sous_mot(chaine_aleatoire(5),chaine_aleatoire(10**4)))
for k in range(100)]))
```

Réponse sous forme d'une autre question : quelle est la probabilité d'obtenir une chaine aléatoire donnée S_1 de 5 caractères, construite à partir d'un alphabet de 4 lettres ? En déduire le nombre moyen théorique de positions où cette chaine doit se trouver dans une chaine S_2 de longueur 10^4 caractères et vérifier...

* *
*