Séance 5 : Listes, tuples et gestion mémoire

L1 – Université Côte d'Azur

Exercice 1 — Occurrence (⋆)

Pour chacune des fonctions, écrivez au moins trois tests avec `assert`.

1. En utilisant une boucle `for`, définissez une fonction `apparaît(x, L)` qui renvoie `True` si le nombre `x` apparaît dans la liste `L` et `False` sinon.
2. Comment auriez-vous fait avec le mot-clé `in` ?
3. Définissez la fonction `contient(L1, L2)` qui renvoie `True` si tous les nombres de la liste de nombres `L1` apparaissent dans `L2` et `False` sinon.
4. Définissez la fonction `commun(L1, L2)` qui renvoie le premier nombre de `L1` qui apparait aussi dans `L2`. Si un tel nombre n'existe pas, `commun(L1, L2)` renvoie `False`.

Exercice 2 — Produit scalaire (⋆)

Dans cet exercice, on identifie les tuples avec les coordonnées dans $\mathbb{R}^n$ muni de sa base orthonormale canonique.

1. Définissez une fonction `produit_scalaire(v1, v2)` qui prend en argument deux vecteurs représentés par des tuples de nombres de même longueur et qui renvoie leur produit scalaire. Par exemple, `produit_scalaire((1,2,3), (4,5,6))` renvoie 32 (= 4 + 10 + 18).
2. Déduisez-en une fonction `norme(v)` qui prend en argument un vecteur représenté par un tuple de nombres et qui renvoie sa norme. Par exemple, `norme((3,4))` renvoie `5.0`.

Exercice 3 — Produit externe (⋆)

Définissez une fonction `produit_externe(k, L)` qui prend en argument un nombre `k` et un vecteur $\vec{v}$ représenté par une **liste** de nombres, et qui renvoie la liste représentant le vecteur $k \cdot \vec{v}$, **sans modifier** `L`. Proposez une première solution avec la méthode `append` et une autre solution avec une liste définie par compréhension.

Exercice 4 — Substitution (⋆)

Définissez une *procédure* `remplace(x, y, L)` sans valeur de retour qui a pour effet de remplacer dans `L` tous les `x` par `y`. Par exemple, on aura au toplevel :

```
>>> L = [1, 2, 4, 1, 3]
>>> remplace(1, 5, L)
>>> L
[5, 2, 4, 5, 3]
```

```
def remplace(x,y,L):
    for i in range(len(L)):
        if L[i] == x:
            L[i] = y
```

Exercice 5 — Reconnaître une permutation (⋆⋆)

Définissez la fonction `est_permutation(L)` qui prend en argument une liste `L` de n entiers et qui renvoie `True` si `L` contient une et une seule fois tous les nombres de 0 à $n-1$. Par exemple, `est_permutation([0,2,1])` renvoie `True` mais `est_permutation([0,1,0])` et `est_permutation([1,3,2])` renvoient `False`. Cherchez d'abord une solution quelconque, puis ajoutez la contrainte de parcourir une seule fois `L` (complexité linéaire).

Une première idée consiste à vérifier que tous les nombres de L sont uniques et bien compris entre 0 et `Len(L)-1`.

```
def unique(x,L):
    déjà_trouvé = False
    for e in L:
        if e==x and déjà_trouvé:
            return False
        elif e==x:
            déjà_trouvé = True
    return déjà_trouvé

def est_permutation(L):
    n=len(L)
    for e in L:
        if not( 0<=e and e<n and unique(e,L)):
            return False
    return True
```

Une méthode plus rapide consiste à vérifier que l'on trouve tous les nombre de 0 à `len(L)-1` au moins une fois. Forcément, il ne peut pas y avoir d'autres valeurs.

```
def est_permutation(L):
    for i in range(len(L)):
        if i not in L: # if not(i in L):
            return False
    return True
```

La méthode la plus efficace consite à faire un seul parcours de la liste et d'utiliser une liste pour se rappeler si on ait déjà tombé sur un nombre. Si on parcours les `Len(L)` de la liste en constatant qu'elles sont toutes uniques et comprise entre 0 et `Len(L)-1` c'est forcément que l'on a une permutation des éléments entre 0 et `Len(L)-1`.

```
def est_permutation(L):
    n = len(L)
    # Pour l'instant on en a trouvé aucun
    trouvés = [False]*n
    for e in L:
        if not(0<=e and e<n):
            return False
        elif trouvés[e]: # si on l'a déjà rencontré
            return False
        else:
            trouvés[e]=True
    return True
```

Exercice 6 — Crible d'Ératosthène (⋆⋆)

Écrivez une fonction `ératosthène(n)` qui renvoie la liste des nombres premiers inférieurs ou égaux `n` en appliquant l'algorithme du crible d'Ératosthène (demandez si nécessaire des explications à votre enseignant ou Wikipédia).

L'idée est de commencer avec un tableau L contenant tous les nombres de 0 à n. Ainsi on a pour tout k, L[k]==k. Dans le crible d'Érathostène, on barre les nombre non premiers. L'équivalent de barrer sera ici de remplacer k par False. Si à la fin L[k]==k, alors k est premier. Si par contre L[k]==False, k ne l'est pas.

*On fait varier i de 2 à $\sqrt{n}$. On continue le calcul tant que i*i≤n. Si L[i]!= False cela signifie que i est premier (sinon, i aurait été barré précédemment) . Il reste à barrer les multiples de i : 2i, 3i, 4i, etc. C'est un range de pas i et dont la première valeur est 2i.*

À la fin on récupère tous les nombres de L différents de False. Ce sont les nombres premiers.

```
def Eratosthène(n):
    # On initialise L à [0..n]
    L=[]
    for i in range(n+1):
        L.append(i)
    # Par définition 0 et 1 ne sont pas premiers
    L[0] = False
    L[1] = False
    # On parcours les entiers de i à racine de n.
    i=2
    while i*i <= n:
        # Si i est premier
        # On barre tous les multiples de i : 2*i, 3*i, etc.
        if L[i] != False:
            for k in range(2*i,n+1,i):
                L[k] = False
        i = i+1
    # Pour chaque indice i :
    # si L[k] = i -> k est premier
    # Si L[k] = False -> k n'est pas premier
    return [ k for k in L if k != False]
```

Exercice 7 — Les neurones de la lecture (⋆⋆−⋆ ⋆ ⋆)

1. Définissez une fonction `mélange(m)` qui renvoie un mot obtenu à partir de `m` en changeant aléatoirement l'ordre de ces lettres à l'exception de la première et de la dernière lettre. Par exemple, `mélange('cerise')` pourra renvoyer `'creise'` ou `'cierse'`. *Indication* : Vous pouvez utiliser `list` pour convertir une chaîne de caractères en liste et écrire vous-même la fonction inverse `liste_vers_chaîne(L)`. La fonction `random.shuffle` permet de mélanger les éléments d'une liste.

```
from random import shuffle

def liste_vers_chaîne(L):
    s=""
    for e in L:
        s = s+e
    return s

def mélange(mot):
    n=len(mot)
    if  n<=2: # si le mot est très court, rien à faire !
        return mot
    else:
        centre=list(mot[1:n-1]) # On aurait pu créer la liste avec une boucle
        shuffle(centre) # on mélange
        return  mot[0] + liste_vers_chaîne(centre) + mot[n-1]
```

2. Définissez une fonction `liste_des_mots(m)` qui prend en argument un texte composé de mots et de ponctuation (`',.:; !?`) et qui renvoie la liste des mots et ponctuations qui constituent ce texte. Par exemple,

```
>>> liste_des_mots('Un: deux? Trois!')
['Un', ':', ' ', 'deux', '?', ' ', 'Trois', '!']
```

```python
def liste_des_mots(m):
    ponctuation= " :,.;!?'"
    liste_mots = []
    mot_en_cours = ''
    for c in m:
        if c in ponctuation:
            # Si deux ponctuations se suivent on n'ajoute pas de mot vide.
            if mot_en_cours != '':
                liste_mots.append(mot_en_cours)
            # Dans tous les cas
            liste_mots.append(c)
            mot_en_cours = ''
        else:
            mot_en_cours = mot_en_cours + c
    # On est arrivé à la fin.
    # On ajoute le dernier mot à la liste
    if mot_en_cours != '':
        liste_mots.append(mot_en_cours)
    return liste_mots
```

3. Définissez une fonction `mélange_texte(s)` qui renvoie ce même texte où chaque mot a été mélangé comme à la question 1.

```python
def mélange_mots(m):
    mots_normaux = liste_des_mots(m)
    mots_mélangés = []
    for mot in mots_normaux:
        mots_mélangés.append(mélange(mot))
    return liste_vers_chaîne(mots_mélangés)
```

4. Mélangez l'article 1 de la déclaration universelle des droits de l'homme. Confirmez-vous ce que rapportent certains sites webs sur la capacité du cerveau humain à remettre dans le bon ordre les lettres en lecture rapide ?

Exercice 8 — Statistiques (⋆ ⋆ ⋆)

Vous êtes libre pour gérer vous-même l'affichage (taille des marges, couleurs utilisés, etc) tant que le résultat est satisfaisant à vos yeux exigeants. Seule condition : le code doit fonctionner quelque soit la taille de la fenêtre.

1. Créer une fonction `effectif(L)` qui renvoie une liste de la forme `[(xmin,ymin), … ,(xmax,ymax)]`. Chaque couple `(x,y)` obtenu correspond à un élément `x` de `L` et à son nombre d'appartions `y` dans la liste `L`. Les valeurs `xmin` et `xmax` sont respectivement le minimum et le maximum de `L`.

```
>>> effectif([10,5,5,10,7,10,10,5])
[(5, 3), (6, 0), (7, 1), (8, 0), (9, 0), (10, 4)]
```

```python
def effectifs(L):
    Xmin=min(L)
    Xmax=max(L)
    E = [ (i,0) for i in range(Xmin,Xmax+1) ]
    for e in L:
        (ee,xx) = E[e-Xmin] # ee=e
        E[e-Xmin] = (ee,xx+1)
    return E
```

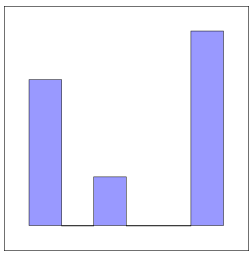
2. Avec Tk, faites un programme qui affiche l'histogramme correspondant à l'effectif. On prendra soin de n'afficher que les valeurs comprises entre le minimum et le maximum de `L` comme dans le graphisme ci-dessous correspondant à l'exemple précédant : `[10,5,5,10,7,10,10,5]`.

```
Titre = "Afficheur statistique"
Hauteur = 500
Largeur = Hauteur
marge=Hauteur/10
root = tk.Tk()
root.title(Titre)
Dessin = tk.Canvas(root,height=Hauteur,width=Largeur,bg="white")
Dessin.pack()
# Quelques variables globales
eff=effectifs(L)
Xmin=eff[0][0]
Xmax=eff[-1][0]
Ymax=max([y for (x,y) in eff])
Δx = (Largeur-2*marge)/(Xmax-Xmin+1)
Δy = (Hauteur-2*marge)/Ymax

def convertir_coordonnée(x,y):
    X = marge+x*Δx
    Y = Hauteur  - (marge+y*Δy)
    return (X,Y)

def afficher_rectangle(i,j):
    p=convertir_coordonnée(i,0)
    q=convertir_coordonnée(i+1,j)
    Dessin.create_rectangle(p,q,fill="#9999FF")

for x in range(Xmin,Xmax+1):
    (i,j)=eff[x-Xmin]
    afficher_rectangle(i-Xmin,j)
```

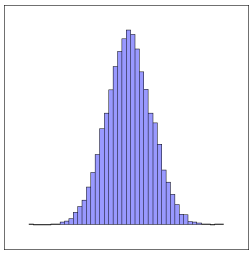


3. Écrivez une fonction `hasard(n)` qui simule n lancers de pièce et qui compte le nombre de piles.

```
# À mettre au début du fichier
from random import randint

def hasard():
    i=0
    for j in range(100):
        i+=randint(0,1)
    return i
```

4. Utilisez cette fonction pour générer une liste de 1000 valeurs aléatoires, puis, affichez son histogramme. Vous devez obtenir une courbe en cloche.



```
L = [ hasard() for i in range(10000) ]
```