Rappel sur les listes, effets de bords

Question 1

In [30]:
```
def foo_0(x):
    x += 1
    return x
```

In [31]:
```
x = 3
y = foo_0(x)
print(x, y)
```

```
3 4
```

In [8]:
```
def foo_1(t):
    t[0] = t[0] + 1
```

In [11]:
```
t = [1, 2, 7]
foo_1(t)
print(t)
```

```
[2, 2, 7]
```

In [32]:
```
def foo_1_bis(x):
    x += 1
```

In [33]:
```
x = 3
foo_1_bis(x)
print(x)
```

```
3
```

In [34]:
```
def foo_2(t):
    t.append(0)
```

In [35]:
```
t = [1, 2, 7]
foo_2(t)
print(t)
```

```
[1, 2, 7, 0]
```

In [41]:
```
def foo_2_bis(t):
    t = t + [0]
```


In [42]:
```
t = [1, 2, 7]
foo_2_bis(t)
print(t)
```

```
[1, 2, 7]
```

In [51]:
```
def foo_2_doublebis(t):
    t += [0]
```

In [52]:
```
t = [1, 2, 7]
foo_2_doublebis(t)
print(t)
```

```
[1, 2, 7, 0]
```

In [53]:
```
def foo_3(t):
    t_local = t
    t_local[0] = t_local[0] + 1
```

In [54]:
```
t = [1, 2, 7]
foo_3(t)
print(t)
```

```
[2, 2, 7]
```

In [55]:
```
def foo_3_bis(t):
    t_local = [x for x in t]
    t_local[0] = t_local[0] + 1
```

In [56]:
```
t = [1, 2, 7]
foo_3_bis(t)
print(t)
```

```
[1, 2, 7]
```

En conclusion foo_1, foo_2, foo_2_doublebis et foo_3 agissent par effet de bord sur t.

- foo_0 est une fonction pure. Elle ne modifie pas l'entier x car le type int est immutable. L'instruction x += 1 est équivalente à l'instruction x = x + 1. L'execution de x + 1 crée une nouvelle valeur et la variable locale x est associée à cette nouvelle valeur. Par contre, cela n'affecte pas la valeur du x en dehors de la fonction.
- foo_1 agit par effet de bord car ce n'est pas la valeur de t qui est passée à la fonction, mais son identifiant. Comme la liste est un objet de type mutable contrairement à tous les autres types vus précédemment (bool, int, float, complex, tuple), cette fonction agit par effet de bord.
- foo_1_bis n'a aucun effet de bord. En effet, le type int étant immutable, l'instruction x += 1 est équivalente à x = x + 1 et cette instruction associe une nouvelle valeur à la variable locale x. Elle ne change pas la valeur associée à la variable x lors de l'appel de la fonction.
- foo_2 agit par effet de bord
- foo_2_bis n'agit par par effet de bord. En effet t + [0] crée une nouvelle liste obtenue par concatenation de la liste t et de la liste [0]. Cette nouvelle liste est ensuite associée à la variable locale t. En conséquence, la valeur de t lors de l'appel est inchangée.
- foo_2_doublebis agit par effet de bord. C'est une dez bizareries et un des dangers de x += a qui n'est pas équivalent à x = x + a lorsque x est de type mutable. La première instruction effectue une mutation de la valeur tandis que la seconde instruction crée une nouvelle valeur.
- foo_3 agit par effet de bord. En effet, l'instruction t_local = t ne fait pas une copie de la liste t. Cette instruction associe un deuxième nom à la valeur associée à t. Tout changement fait sur une des variables aura donc un effet sur l'autre. On dit dans ce cas qu'il y a aliasing.
- foo_3_bis n'agit pas par effet de bord sur t car t_local contient une copie du tableau t.

Question 2

In [57]:
```
def stop_zero_1(t, eps):
    ans = []
    for x in t:
        if x == 0:
            ans.append(eps)
        else:
            ans.append(x)
    return ans
```

In [58]:
```
t = [2, 0, 3]
stop_zero_1(t, 1.0e-16)
```

Out[58]:
```
[2, 1e-16, 3]
```


In [60]:
```
def stop_zero_2(t, eps):
    for i in range(len(t)):
        if t[i] == 0:
            t[i] = eps
```

In [63]:
```
t = [2, 0, 3]
stop_zero_2(t, 1.0e-16)
print(t)
```

```
[2, 1e-16, 3]
```

Question 3

In [64]:
```
def moyenne(t):
    ans = 0
    for x in t:
        ans += x
    ans /= len(t)
    return ans
```

In [65]:
```
t = [2, 0, 3]
moyenne(t)
```

Out[65]:
```
1.6666666666666667
```

In [68]:
```
import numpy as np

def ecart_type(t):
    m = moyenne(t)
    t_local = [(x - m)**2 for x in t]
    variance = moyenne(t_local)
    return np.sqrt(variance)
```

In [69]:
```
t = [2, 0, 3]
ecart_type(t)
```

Out[69]:
```
1.247219128924647
```

Algorithme de compréssion RLE

Question 1

Le codage RLE de la liste [4,4,1,1,1,1,1,3,3,3,3,6,6,6,6] est [(4,2),(1,5),(3,4),(6,4)]. Le codage RLE de la liste [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1] est [(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1),(0,1),(1,1)]. Le codage de la première liste prends moins de mémoire que la liste. Cependant, le codage de la seconde liste prend malheureusement plus de mémoire.

La liste dont le codage est [(3,5),(4,2),(1,3),(3,1)] est [3,3,3,3,3,3,4,4,1,1,1,3]

Question 2

In [71]:
```
def decode_rle(c):
    ans = []
    for x, n in c:
        ans += [x for k in range(n)]
    return ans
```

In [72]:
```
decode_rle([(3, 5), (4, 2), (1, 3), (3, 1)])
```

Out[72]:
```
[3, 3, 3, 3, 3, 4, 4, 1, 1, 1, 3]
```

Question 3

In [89]:
```
def encode_rle(t):
    ans = []
    # L'indice courant du tableau
    k = 0
    # Un faux dernier element qui est égal au premier élément de la
    dernier_element = t[0]
    # Le nombre d'éléments égaux consécutifs dans la séquence
    n = 0
    while k <= len(t):
        if k == len(t) or dernier_element != t[k]:
            ans.append((dernier_element, n))
            if k < len(t):
                dernier_element = t[k]
                n = 1
        else:
            n += 1
        k += 1
    return ans
```

In [94]:
```
encode_rle([1, 1, 1, 2, 2, 3, 3, 3, 3])
```

Out[94]:
```
[(1, 3), (2, 2), (3, 4)]
```

In [95]:
```
decode_rle(encode_rle([1, 1, 1, 2, 2, 3, 3, 3, 3]))
```

Out[95]:
```
[1, 1, 1, 2, 2, 3, 3, 3, 3]
```


Question 4

Pour que ce type de codage ait de l'intérêt pour la compression de données, il faut qu'il y ait de nombreuses valeurs les unes à la suite des autres qui soient égales. Ce genre de compression peut être utile pour compresser des images de dessins ayant de grands aplats de couleur. Cet algorithme est notamment utilisé pour la compression d'images gif ou png.

Manipulation de parties - Représentation par des listes

Question 1

In [97]:
```
def liste_vers_ensemble(t):
    ans = []
    for x in t:
        if x not in ans:
            ans.append(x)
    return ans
```

In [98]:
```
liste_vers_ensemble([2, 3, 2, 7])
```

Out[98]:
```
[2, 3, 7]
```

Question 2

In [101]:
```
def est_present(A, x):
    g = 0
    d = len(A) - 1
    while g <= d:
        m = (g + d) // 2
        if x < A[m]:
            d = m - 1
        elif x > A[m]:
            g = m + 1
        else:
            return True
    return False
```

Question 3

In [103]:
```
def enigme(A, B):
    res = []
    for a in A:
        if not (a in B):
            res.append(a)
    return res
```

Question 4

In [106]:
```
def intersection(t1, t2):
    ans = []
    i1 = 0
    i2 = 0
    while i1 < len(t1) and i2 < len(t2):
        if t1[i1] == t2[i2]:
            ans.append(t1[i1])
            i1 += 1
            i2 += 1
        elif t1[i1] < t2[i2]:
            i1 += 1
        else:
            i2 += 1
    return ans
```

In [108]:
```
intersection([1, 3, 4, 9], [3, 7])
```

Out[108]: [3]

Manipulation de parties - Représentation binaire

Question 1

In [110]:
```
def barre(t):
    return [1 - x for x in t]
```

In [111]:
```
barre([1, 0, 1, 1, 0])
```

Out[111]: [0, 1, 0, 0, 1]

Question 2

In [112]:
```
def intersection(t1, t2):
    n = len(t1)
    ans = [-1 for k in range(n)]
    for k in range(n):
        if t1[k] == 1 and t2[k] == 1:
            ans[k] = 1
        else:
            ans[k] = 0
    return ans
```

In [113]:
```
intersection([0, 1, 0], [1, 1, 0])
```

Out[113]: [0, 1, 0]

In [114]:
```
def union(t1, t2):
    n = len(t1)
    ans = [-1 for k in range(n)]
    for k in range(n):
        if t1[k] == 1 or t2[k] == 1:
            ans[k] = 1
        else:
            ans[k] = 0
    return ans
```

In [115]:
```
def delta(t1, t2):
    n = len(t1)
    ans = [-1 for k in range(n)]
    for k in range(n):
        if (t1[k] == 1 and t2[k] == 0) or (t1[k] == 0 and t2[k] ==
            ans[k] = 1
        else:
            ans[k] = 0
    return ans
```

Manipulation de parties - Représentation par un entier

Question 1

In [118]:
```
bin(319)
```

Out[118]: '0b100111111'

In [119]:
```
bin(124)
```

Out[119]: '0b1111100'

Les parties associées sont donc respectivement {a, b, c, d, e, f} et {c, d, e, f, g}

Question 2

In [124]:
```
def entier_vers_liste(ma):
    ans = [0 for k in range(n)]
    k = 0
    while ma != 0:
        ans[k] = ma % 2
        ma //= 2
        k += 1
    return ans
```

In [125]:
```
n = 8
entier_vers_liste(7)
```

Out[125]:
```
[1, 1, 1, 0, 0, 0, 0, 0]
```

Question 3

In [126]:
```
def liste_vers_entier(t):
    m = 0
    puissance = 1
    for k in range(len(t)):
        m += t[k] * puissance
        puissance *= 2
    return m
```

In [128]:
```
liste_vers_entier([1, 0, 1])
```

Out[128]:
```
5
```

In [129]:
```
def intersection_entier(ma, mb):
    ta = entier_vers_liste(ma)
    tb = entier_vers_liste(mb)
    t = intersection(ta, tb)
    return liste_vers_entier(t)
```

In [130]:
```
intersection_entier(7, 12)
```

Out[130]:
```
4
```

Question 4

In [170]:
```
def plus_un(t):
    ans = [x for x in t]
    k = 0
    while k < len(t) and ans[k] == 1:
        ans[k] = 0
        k += 1
    k = k % n
    ans[k] = 1
    return ans
```

In [177]:
```
n = 8
plus_un([1, 1, 0, 0, 0, 0, 0, 0])
```

Out[177]:
```
[0, 0, 1, 0, 0, 0, 0, 0]
```

In [178]:
```
def parcourt():
    t = [0 for k in range(n)]
    for k in range(2**n):
        print(t)
        t = plus_un(t)
```

In [179]:
```
n = 2
parcourt()
```

```
[0, 0]
[1, 0]
[0, 1]
[1, 1]
```

In [188]:
```
def parties():
    t = [0 for k in range(n)]
    for k in range(2**n):
        print("{", end="")
        for i in range(n):
            if t[i] == 1:
                print(E[i], end="")
        print("}")
        t = plus_un(t)
```

In [192]:
```
n = 4
E = ["a", "b", "c", "d"]
parties()
```

```
{}
{a}
{b}
{ab}
{c}
{ac}
{bc}
{abc}
{d}
{ad}
{bd}
{abd}
{cd}
{acd}
{bcd}
{abcd}
```

In [ ]: