# QUELQUES REMARQUES SUR L'ÉPREUVE ITC CCINP PC 2024

## 1 Sur les fonctions, les listes et les dictionnaires

Il était demandé à plusieurs reprises dans l'énoncé de veiller à ce qu'une fonction modifie (ou non) un de ces arguments. On revient sur ce point.

### Comportement des listes

Le point de départ est le fait que certaines structures de données (par exemple les listes et les dictionnaires) sont modifiables en python ("mutable"). Ce n'est pas le cas des chaînes de caractères. Par exemple si on définit une chaîne `chaine` par l'instruction `chaine = 'abc'`, l'instruction `chaine[1] = 'e'` n'est pas valable. On peut avoir l'impression de modifier `chaine` en écrivant `chaine = chaine + 'd'`, mais en fait une nouvelle chaîne (portant le même nom) est créée.
Pour les listes par contre, si l'on crée une liste par l'instruction `l = [0,1]`, puis l'on exécute la commande `l.append(2)` ou bien `l[1] = 3`, on ne crée pas une nouvelle liste, mais on modifie la liste existante. Les dictionnaires on la même propriété.
Une autre particularité des listes est que si l'on déclare une première liste `l1 = [0,1]` puis qu'on exécute la commande `l2 = l1`, il y a une seule liste stockée en mémoire : si on modifie l'une, on modifie l'autre.
Tester :

```
l1 = [0,1]
l2 = l1
l2[0] = 3
print(l1)
```

Pour remédier à cela, on peut faire des copies terme à terme : `l3 = l1[:]`, `l4 = [a for a in l1]` ou utiliser la méthode `copy` (ce qui était suggéré dans l'énoncé). Tester :

```
l1 = [0,1]
l3 = l1[:] ; l4 = [a for a in l1] ; l5 = l1.copy()
l3[0] = 3 ; print(l3,l1)
l4[0] = 3 ; print(l4,l1)
l5[0] = 3 ; print(l5,l1)
```

On retiendra le fait que l'on nne crée jamais une liste par la commande `l_new = l`.

**Exercice 1** – Cela pose également problème lorsque l'on veut créer un tableau de taille $n \times n$ contenant des 0. Tester :

```
ligne = 4*[0]
tableau = 4*[ligne] ; print(tableau)
tableau[0][0] = 1 ; print(tableau)
```

Écrire une suite d'instructions qui crée un tableau (de même taille) en évitant le problème identifié ci-dessus. Tester. **Corrigé:**

```
tableau = []
for i in range(4):
    tableau.append([0 for j in range(4)]) # 4*[0] aurait convenu
print(tableau)
tableau[0][0] = 1 ; print(tableau)
```

ou bien

```
tableau = [[0 for j in range(4)] for i in range(4)] ; print(tableau)
tableau[0][0] = 1 ; print(tableau)
```

## Fonctions et listes

Le comportement des listes rentrées en paramètre d'une fonction est lié à ce qui précède. Quand une liste est un argument d'une fonction, les modifications effectuées lors de l'exécution de la fonction subsistent en dehors de la fonction ; on dit que la liste est passée "en référence " à la fonction.
Prenons des exemples pour fixer les idées. Tester :

```
def ajoutUn(x):
    x = x+1
    return x

x = 0
print(ajoutUn(x))
print(x)
```

La valeur de `x` n'a pas été modifiée. C'est sa valeur qui a été transmise à la fonction.
Remarque : la variable `x` créée à l'intérieur de la fonction est locale. La formulation de cette fonction est maladroite (voire trompeuse).
Observons maintenant ce qui se passe avec les listes :

```
def echange(l):
    l[0] , l[1] = l[1] , l[0]
    return l

l1 = [0,1]
print(echange(l1))
print(l1)
```

La liste `l1` a été modifiée : c'est sa référence, et non sa valeur, qui a été passée à la fonction.
On peut même remarquer que l'instruction `return` est superflue (voire trompeuse) : l'objectif de cette fonction est de modifier son entrée. On aurait dû écrire :

```
def echange2(l):
    l[0] , l[1] = l[1] , l[0]
```

Cette dernière fonction retourne `None`. Une telle fonction est parfois qualifiée (en référence à d'autres langages où la distinction est plus nette) de procédure ; on lira également que cette fonction agit par effet de bord.
Cette caractéristique peut paraître bizarre d'un première abord, mais dans de nombreuses situations on ne souhaite pas créer de nouvelle liste (penser aux tris par exemple) mais faire évoluer la liste d'entrée. Si l'on veut créer une nouvelle liste, il faut le faire explicitement.

**Exercice 2** Écrire une fonction `echange3(l)` qui renvoie, sans modifier son entrée `l`, une liste similaire à `l` dans laquelle les deux premiers termes ont été échangés. On utilisera la méthode `copy` associée aux listes. **Corrigé:**

```
def echange3(l):
    l_new = l.copy()
    l_new[0] , l_new[1] = l_new[1] , l_new[0]
    return l_new

l1 = [0,1]
print(echange3(l1))
print(l1)
```

On peut également dans une fonction combiner le fait de retourner une certaine valeur et de modifier un paramètre d'entrée (d'où le terme "effet de bord") :

**Exercice 3** – Écrire une fonction `echangeMaxMin(l)` qui prend en paramètre une liste d'entiers `l`, qui renvoie la différence entre le maximum et le minimum des valeurs de `l`, et qui échange ces deux valeur au sein de la liste. On supposera que la liste est de longueur au moins 2 et que tous ses éléments sont distincts.
Par exemple, si `l1 = [1,0,4,2]`, l'exécution de la commande `echangeMaxMin(l1)` devra renvoyer `4` et modifier `l1` de sorte que celle-ci soit égale à `[1,4,0,2]`.
Écrire les instructions permettant de tester tout cela. **Corrigé:**

```
def echangeMaxMin(l):
    ind_min ,val_min = 0 , l[0]
    ind_max ,val_max = 0 , l[0]
    for i in range(1,len(l)):
        if l[i] < val_min:
            ind_min ,val_min = i , l[i]
        if l[i] > val_max:
            ind_max ,val_max = i , l[i]
    l[ind_min] , l[ind_max] = l[ind_max] , l[ind_min]
    return val_max - val_min

l1 = [1,4,0,2] ; print(l1)
sortie = echangeMaxMin(l1)
print(f'valeur de la différence : {sortie}')
print(f'nouvelle valeur de la liste l1 : {l1}')
```

## Avec les dictionnaires

**Exercice 4** – Deux joueurs jouent au jeu suivant : ils lancent chacun un dé équilibré et ajoutent le résultat à leur score.
Rappel : on peut obtenir un nombre aléatoire selon une loi uniforme sur $[[a, b]]$ par la commande `rd.randint(a,b)` (après l'import : `import random as rd`).
On modélise un état du jeu par un dictionnaire `dicoJeu` ayant deux clefs : `dicoJeu['nbTours']` contient le nombre de tours joués et `dicoJeu['score']` contient le score sous la forme d'une liste à deux éléments (le premier éléments est le score du joueur 1 et le second le score du joueur 2).

1. Écrire une fonction `initJeu()` qui renvoie un dictionnaire représentant l'état initial du jeu.
2. Écrire une fonction `tourJeu(dicoJeu)` qui fait évoluer dictionnaire `dicoJeu` selon un tour du jeu.
3. Écrire une fonction `tourJeuBis(dicoJeu)` qui renvoie un dictionnaire représentant l'état du jeu après un tour de jeu à partir de l'état représenté par `dicoJeu`. Cette fonction ne devra pas modifier son entrée.
4. Écrire une suite d'instruction qui simule une partie durant 10 tours et qui affiche le nom du gagnant.

**Corrigé:**

```
import random as rd

def initJeu():
    dicoJeu = {}
    dicoJeu['nbTours'] = 0
    dicoJeu['score'] = [0,0]
    return dicoJeu

def tourJeu(dicoJeu):
    dicoJeu['nbTours'] += 1
    de1 , de2 = rd.randint(1,6) , rd.randint(1,6)
    dicoJeu['score'][0] += de1
    dicoJeu['score'][1] += de2

dicoJeu =  initJeu()
for i in range(10) :
    tourJeu(dicoJeu)

print(dicoJeu)
if dicoJeu['score'][0] > dicoJeu['score'][1]:
    print('le joueur 1 a gagné')
elif dicoJeu['score'][0] < dicoJeu['score'][1]:
    print('le joueur 2 a gagné')
else:
    print('égalité')


# question 3
def tourJeuBis(dicoJeu):
    dicoJeu_new = {}
    dicoJeu_new['nbTours'] = dicoJeu['nbTours'] + 1
    dicoJeu_new['score'] = dicoJeu['score'].copy()
    de1 , de2 = rd.randint(1,6) , rd.randint(1,6)
    dicoJeu_new['score'][0] += de1
    dicoJeu_new['score'][1] += de2
    return dicoJeu_new
```

# 2 Sur SQL

## 2.1 Avec la BDD du DS

On reprend la base de données du DS.
Écrire les requètes SQL permettant d'obtenir les informations suivantes.

1. Déterminer les noms des joueurs ayant joué une partie le 10 janvier 2024 (date représentée par la,chaîne de caractères '10-01-2024') et commencé celle-ci. **Corrigé:**

```
SELECT DISTINCT J.nom
FROM Joueur AS J JOIN Partie AS P
                  ON J.id_joueur = P.id_joueur1
```

2. Déterminer les noms des joueurs ayant joué une partie le 10 janvier 2024 (date représentée par la,chaîne de caractères '10-01-2024'). **Corrigé:**

```
SELECT DISTINCT J.nom
FROM Joueur AS J JOIN Partie AS P
                  ON J.id_joueur = P.id_joueur1
UNION
SELECT DISTINCT J.nom
FROM Joueur AS J JOIN Partie AS P
                  ON J.id_joueur = P.id_joueur2
```

3. Déterminer pour chaque joueur (identifié par son identifiant) le nombre de parties qu'il commencées et gagnées. **Corrigé:**

```
SELECT  id_joueur1 , COUNT(*)
FROM Joueur
WHERE resultat = 1
GROUP BY id_joueur1
```

4. Déterminer les joueurs (leurs identifiants) dont le nombre de parties commencées et gagnées est supérieur à 10. **Corrigé:**

```
SELECT  id_joueur1 , COUNT(*)
FROM Joueur
WHERE resultat = 1
GROUP BY id_joueur1
HAVING COUNT(*) > 10
```

## 2.2 Avec la BDD Commerce

On reprend la base de données ayant servi de support au cours.

### Autour de la question sur le pourcentage

1. Déterminer le pourcentage de commandes payées. **Corrigé:**

```
SELECT 100*AVG(Paye)
FROM commandes
```

ou bien

```
SELECT 100*SUM(Paye)/COUNT(*)
FROM commandes
```

2. Déterminer le nombre de commande effectuées le '12/05/2015'. **Corrigé:**

```
SELECT COUNT(*)
FROM commandes
WHERE date ='12/05/2015'
```

3. Déterminer le nombre de commande effectuées le '12/05/2015' et qui sont payées. **Corrigé:**

```
SELECT COUNT(*)
FROM commandes
WHERE date ='12/05/2015'  and paye = 1
```

4. Déterminer le pourcentage des commandes effectuées le '12/05/2015' qui sont payées. **Corrigé:**

```
SELECT 100*(SELECT COUNT(*)
FROM commandes
WHERE date ='12/05/2015' and paye =1)/
(SELECT COUNT(*)
FROM commandes
WHERE date ='12/05/2015' )
```

### Classement

1. Déterminer les trois pizzas les plus chères (classées en ordre descendant) **Corrigé:**

```
SELECT nom , prix
FROM pizzas
ORDER BY prix DESC LIMIT 3
```

2. Déterminer la troisième pizza la plus chère **Corrigé:**

```
SELECT nom , prix
FROM pizzas
ORDER BY prix DESC LIMIT 1 OFFSET 2
```

### Autour de la question nécessitant un GROUP BY

1. Déterminer pour chaque pizza le nombre de fois où elle a été commandée. **Corrigé:**

```
SELECT piz.nom  , COUNT(*)
FROM pizzas AS piz JOIN commandes AS Com
                   ON piz.cle = com.pizza
GROUP BY piz.cle
```

2. Déterminer les pizzas ayant été commandées plus de 10 fois, ordonnées par nombre de commandes décroissant. **Corrigé:**

```
SELECT piz.nom  , COUNT(*)
FROM pizzas AS piz JOIN commandes AS Com
                   ON piz.cle = com.pizza
GROUP BY piz.cle
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC
```