Autotest n°2 – Correction

Exercice 1 : *Cet exercice porte sur la programmation orientée objet et les dictionnaires.*
Dans le tableau ci-dessous, on donne les caractéristiques nutritionnelles, pour une quantité de 100 grammes, de quelques aliments.

	Lait entier UHT	Farine de blé	Huile de tournesol
Énergie (kcal)	65.1	343	900
Protéines (grammes)	3.32	11.7	0
Glucides (grammes)	4.85	69.3	0
Lipides (grammes)	3.63	0.8	100

Figure 1 : Caractéristiques nutritionnelles

Pour chaque aliment, on souhaite stocker les informations dans un objet de la classe `Aliment` définie ci-dessous, où `e`, `p` ,`g` et `l` sont de type `float` et désignent respectivement les quantités d'énergie, de protéines, de glucides et de lipides de l'aliment.

```
1 class Aliment:
2     def __init__(self, e, p, g, l):
3         self.energie = e
4         self.proteines = p
5         self.glucides = g
6         self.lipides = l
```

1) a) Écrire, à l'aide du tableau des caractéristiques nutritionnelles de la figure 1, l'instruction en langage Python pour instancier l'objet `lait`.
Solution : `lait = Aliment(65.1, 3.32, 4.85, 3.63)`

   b) Donner l'expression qui permet d'obtenir la valeur 65.1 de l'objet lait instancié dans la question précédente.
Solution : `lait.energie`

   c) Une erreur s'est introduite dans le tableau de la figure 1 : la masse de protéines dans le lait est 3.4 au lieu de 3.32.
Donner l'instruction qui modifie la masse de protéines de l'objet `lait` instancié dans la question 1)a).
Solution : `lait.proteines = 3.4`

On souhaite ajouter une méthode `energie_reelle` à la classe `Aliment` qui calcule l'énergie en kcal d'un aliment en fonction d'une masse donnée.
Par exemple :
Pour 245 grammes de lait, l'énergie réelle sera $245 \times 65.1 \div 100 = 159.495$ kcal.
L'instruction `lait.energie_reelle(245)` renvoie alors 159.495.

2) Compléter la **méthode** de la classe `Aliment` ci-dessous.

```
    def energie_reelle(self, masse):
        return masse * self.energie / 100
```

3) On regroupe les caractéristiques nutritionnelles du tableau de la figure 1 dans le dictionnaire suivant, les clés étant des chaînes de caractères donnant le nom de l'aliment et les valeurs associées des objets de la classe `Aliment` :

```
nutrition = {'lait': Aliment(65.1, 3.4, 4.85, 3.63),
             'farine': Aliment(343, 11.7, 69.3, 0.8),
             'huile': Aliment(900, 0, 0, 100)
```

```
        }
```

a) Donner l'expression qui permet d'obtenir la valeur énergétique en kcal du lait à partir des données de ce dictionnaire.
Solution : `nutrition['lait'].energie`

b) Donner l'expression qui permet d'obtenir la valeur énergétique réelle de 220 grammes de lait à partir des données de ce dictionnaire.
Solution : `nutrition['lait'].energie_reelle(200)`

Une recette de gâteau (sans œuf) utilise les ingrédients suivants :
- 230 g de farine ;
- 220 g de lait ;
- 100 g d'huile.

Les quantités d'ingrédients, en grammes, sont regroupées dans le dictionnaire suivant :
`recette_gateau = {'lait': 220, 'farine': 230, 'huile': 100}`

4) Ecrire, en utilisant la classe `Aliment` et la méthode `energie_reelle`, les instructions nécessaires pour calculer et afficher l'énergie réelle totale du gâteau.

```
total = 0
for aliment in recette_gateau:
    total += nutrition[aliment].energie_reelle(recette_gateau[aliment])
print(total)
```

Exercice 2 : *Cet exercice porte sur la programmation en Python en général, la programmation orientée objet et la récursivité.*

On se déplace dans une grille rectangulaire. On s'intéresse aux chemins dont le départ est sur la case en haut à gauche et l'arrivée en bas à droite. Les seuls déplacements autorisés sont composés de déplacements élémentaires d'une case vers le bas ou d'une case vers la droite.

Un itinéraire est noté sous la forme d'une suite de lettres :
- `D` pour un déplacement vers la droite d'une case ;
- `B` pour un déplacement vers le bas d'une case.

Le nombre de caractères `D` est la longueur de l'itinéraire. Le nombre de caractères `B` est sa largeur.

Ainsi l'itinéraire `'DDBDBBDDDDB'` a pour longueur 7 et pour largeur 4. Sa représentation graphique est :

```
S++
  ++
   +
   +++++
       E
```

- `S` représente la case de départ (*start*). Ses coordonnées sont (0;0) ;
- `+` représente les cases visitées ;
- `E` représente la case d'arrivée (*end*).

Partie A – Programmation orientée objet

On représente un itinéraire avec la classe `Chemin` dont une partie du code est donné la page suivante.

1) Donner un attribut et une méthode de la classe `Chemin`.

Solution : `itineraire`, `grille`, `longueur` et `largeur` sont des attributs.
`__init__` et `remplir_grille` sont des méthodes.

On exécute le code ci-dessous dans la console Python :

```
chemin_1 = Chemin("DDBDBBDDDDB")
a = chemin_1.largueur
b = chemin_1.longueur
```

2) Préciser les valeurs contenues dans chacune des variables `a` et `b`.
Solution : La largeur est le nombre de `B`. Donc `a` vaut 4. De même, `b` vaut 7.
3) Recopier et compléter la méthode `remplir_grille` qui remplace les `'.'` par des `'+'` pour signifier que le déplacement est passé par cette cellule du tableau.
4) Écrire une méthode `get_dimensions` de la classe `Chemin` qui renvoie la longueur et la largeur de l'itinéraire sous la forme d'un tuple.
5) Écrire une méthode `tracer_chemin` de la classe `Chemin` qui affiche une représentation graphique d'un itinéraire.

```
class Chemin:
    def __init__(self, itineraire):
        self.itineraire = itineraire
        longueur, largeur = 0, 0
        for direction in self.itineraire:
            if direction == "D":
                longueur = longueur + 1
            if direction == "B":
                largeur = largeur + 1
        self.longueur = longueur
        self.largeur = largeur
        self.grille = [['.' for i in range(longueur+1)]
                       for j in range(largeur+1)]

    def remplir_grille(self):
        i, j = 0, 0  # Position initiale
        self.grille[0][0] = 'S'  # Case départ marquée d'un S
        for direction in self.itineraire:
            if direction == 'D':
                self.j = self.j + 1  # Déplacement vers la droite
            elif direction == 'B':
                self.i = self.i + 1  # Déplacement vers le bas
            self.grille[i][j] = '+'  # Marquer le chemin avec '+'
        self.grille[self.largeur][self.longueur] = 'E' # Case d'arrivée

    def get_dimension(self):
        return self.longueur, self.largeur

    def tracer_chemin(self):
        for i in range(self.largeur+1):
            ligne = ""
            for j in range(self.longueur+1):
                if self.grille[i][j] == '.':
                    ligne = ligne + " "
                else:
                    ligne = ligne + self.grille[i][j]
            print(ligne)
```

Partie B – Génération aléatoire d'itinéraires

On souhaite créer des chemins de façon aléatoire. Pour cela, on utilise la méthode `choice` de la bibliothèque `random` dont on fournit ci-dessous la documentation.

```
`random.choice(sequence : list)`
Renvoie un élément choisi dans une liste non vide.
Si la population est vide, lève `IndexError`.
```

On rappelle que l'opérateur * permet de répéter une chaîne de caractères.
Par exemple, on a :

```
>>> "Hello world ! " * 3
'Hello world ! Hello world ! Hello world ! '
```

L'algorithme proposé est le suivant :

- on initialise :
  - une variable `itineraire` comme une chaîne de caractères vide,
  - les variables `i` et `j` à 0 ;
- tant que l'on n'est pas sur la dernière ligne ou la dernière colonne du tableau :
  - on tire au sort entre un déplacement à droite ou en bas,
  - le déplacement est concaténé à la chaîne de caractères `itineraire`,
  - si le déplacement est vers la droite, alors `j` est incrémenté de 1,
  - si le déplacement est vers le bas, alors `i` est incrémenté de 1 ;
- il reste à terminer le chemin en complétant par des déplacements afin d'atteindre la cellule en bas à droite.

6) Écrire les lignes manquantes dans le code ci-dessous. Le nombre de lignes effacées dans le code n'est pas indicatif.

```
from random import choice

def itineraire_aleatoire(m, n):
    itineraire = ''
    i, j = 0, 0
    while i != m and j != n:
        direction = choice(['D', 'B'])
        itineraire = itineraire + direction
        if direction == 'D':
            j = j + 1
        else:
            i = i + 1
    if i == m:
        itineraire = itineraire + 'D'*(n-j)
    if j == n:
        itineraire = itineraire + 'B'*(m-i)
    return itineraire
```

Partie C – Calcul du nombre de chemins possibles

Soit m et n deux entiers naturels non nuls. On se place dans dans le contexte d'un itinéraire de longueur m et de largeur n de dimension $m \times n$.
On note `N(m, n)` le nombre de chemins distincts respectant les contraintes de l'exercice.

7) Pour un itinéraire de dimension $1 \times n$ justifier, éventuellement à l'aide d'un exemple, qu'il y a un seul chemin, c'est-à-dire que, quel que soit n entier naturel, on a $N(1,n) = 1$.
Solution : Si la longueur est de 1, cela veut dire qu'on ne peut pas aller à droite. On ne peut faire que $n-1$ fois bas.
p.s. : selon les définitions du début de l'exercice, un tel itinéraire aurait des dimensions de $0 \times (n-1)$...

De même $N(m,1) = 1$.

8) Justifier que $N(m,n) = N(m-1,n) + N(m,n-1)$ si $n > 1$ et $m > 1$.
Solution : Si on prend un itinéraire de dimensions $m \times n$, il peut soit se terminer par `B`, soit par `D`. En enlevant le dernier symbole, on avait à l'étape précédente soit un itinéraire de dimension $(m-1) \times n$, soit un itinéraire de dimension $m \times (n-1)$.
On a donc $N(m,n) = N(m-1,n) + N(m,n-1)$ si $n > 1$ et $m > 1$.

9) En utilisant les questions précédentes, écrire une fonction récursive `nombre_chemins(m, n)` qui renvoie le nombre de chemins possibles dans une grille rectangulaire de dimension $m \times n$.

Solution :

```
def nombre_chemins(m, n):
    if m == 1 or n == 1:
        return 1
    else:
        return nombre_chemins(m-1, n) + nombre_chemins(m, n-1)
```

Exercice 3 : Revoir la feuille d'exercices sur la POO.

Exercice 4 : Revoir la feuille d'exercices de bac sur la POO.