

Evaluation : partie 1 (50 minutes – 15 points)

Exercice 1 (3 points) :

- 1) Rappeler brièvement ce qui caractérise le paradigme fonctionnel.
Deux appels d'une même fonction avec le même argument doivent renvoyer le même résultat.
- 2) Parmi les 3 séquences d'instructions suivantes, laquelle ou lesquelles ne respectent pas le paradigme fonctionnel ? **La troisième.**

<pre>tab = [5, 6, 7, 8, 9] somme = 0 for elt in tab: somme = somme + elt</pre>	<pre>def incremente(tab): for i in range(len(tab)): tab[i] = tab[i] + 1 return tab tableau = [5, 6, 7, 8, 9] tab2 = incremente(tableau)</pre>	<pre>def augmentation(nombre): return nombre + 1 x = 123 y = augmentation(x)</pre>
--	--	---

Réaffectation pour la première et mutations pour la seconde enfreignent le paradigme fonctionnel.

- 3) Ecrire le code d'une fonction modulo_dix vérifiant les spécifications suivantes :
 - Argument : fonction f.
 - Valeur renvoyée : fonction g.
 - Préconditions : f prend en argument des nombres entiers et renvoie des nombres entiers.
 - Postconditions : si $g = \text{modulo_dix}(f)$, pour tout nombre x entier on a : $g(x) = f(x) \% 10$
 - Exemples d'utilisation :

<pre>def plus_sept(x): return x+7 g = modulo_dix(plus_sept) g(1) = 8 g(12) = 9 g(123) = 0</pre>	<pre>def fois_trois(x): return 3*x g = modulo_dix(fois_trois) g(4) = 2 g(5) = 5 g(11) = 3</pre>	<pre>def modulo_dix(f): def g(x): resultat = f(x)%10 return resultat return g</pre>
--	--	---

Rappel : l'opérateur modulo % calcule le reste dans la division euclidienne : $67 \% 10 = 7$, $1234 \% 10 = 4$, $125 \% 10 = 5$.

Exercice 2 (6 points) :

Ecrire le code d'une classe Rectangle disposant :

- de deux attributs longueur et largeur qui sont initialisés par le constructeur `__init__`,
- d'une méthode `perimetre(self)` qui renvoie le perimetre du rectangle,
- d'une méthode `aire(self)` qui renvoie l'aire du rectangle,
- d'une méthode `grossir(self, delta)` qui permet d'augmenter simultanément les attributs largeur et longueur du rectangle de delta unités,
- d'une méthode `__lt__(self, rect)` qui renvoie True si l'aire du rectangle qui appelle la méthode est plus petite que l'aire du rectangle rect et qui renvoie False sinon.

```
>>> rect_a = Rectangle(7, 5)
>>> rect_b = Rectangle(10, 20)
>>> rect_a.largeur
5
>>> rect_a.aire()
35
>>> rect_b.perimetre()
60
>>> rect_b.grossir(3)
>>> rect_b.longueur
13
>>> rect_b.largeur
23
>>> rect_a < rect_b #pour tester __lt__
True
```

Rappel : pour un rectangle de largeur 5 et de longueur 7, le périmètre est $2 \times 5 + 2 \times 7 = 24$ et l'aire est $5 \times 7 = 35$.

```
class Rectangle():
    def __init__(self, long, larg):
        self.largeur = larg
        self.longueur = long

    def perimetre(self):
        return 2 * self.largeur + 2 * self.longueur

    def aire(self):
        return self.largeur * self.longueur

    def grossir(self, delta):
        self.largeur = self.largeur + delta
        self.longueur = self.longueur + delta

    def __lt__(self, rect):
        return self.aire() < rect.aire()
```

Exercice 3 (6 points) :

On considère la fonction récursive mystere qui prend en argument un nombre entier positif et dont le code est indiqué ci-contre.

- 1) Recopier les lignes de code correspondant au(x) cas de base.

```
if x == 0:
    return 2
elif x == 1:
    return 3
elif x == 2:
    return 5
```

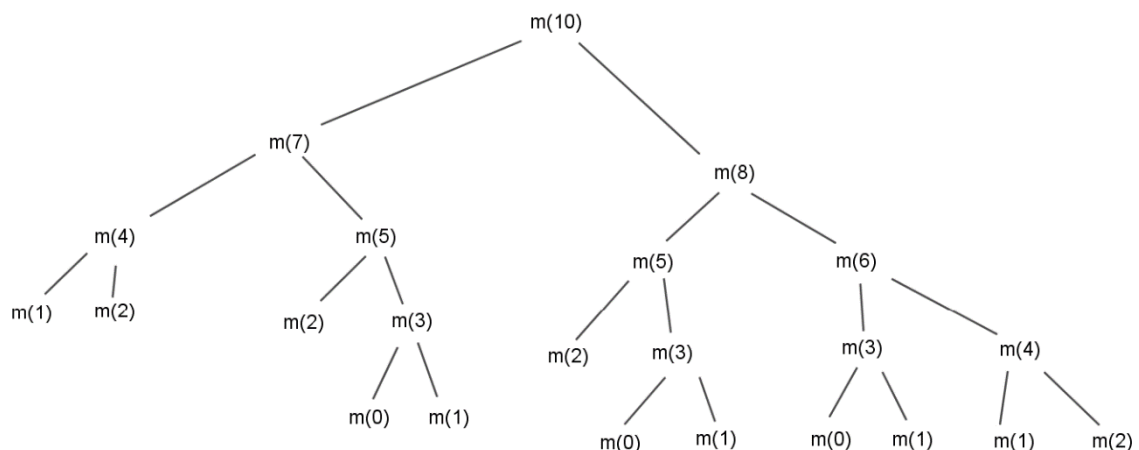
```
def mystere(x):
    if x == 0:
        return 2
    elif x == 1:
        return 3
    elif x == 2:
        return 5
    else:
        return mystere(x-3) * mystere(x-2)
```

- 2) Donner la formule de récursivité correspondant au cas général.

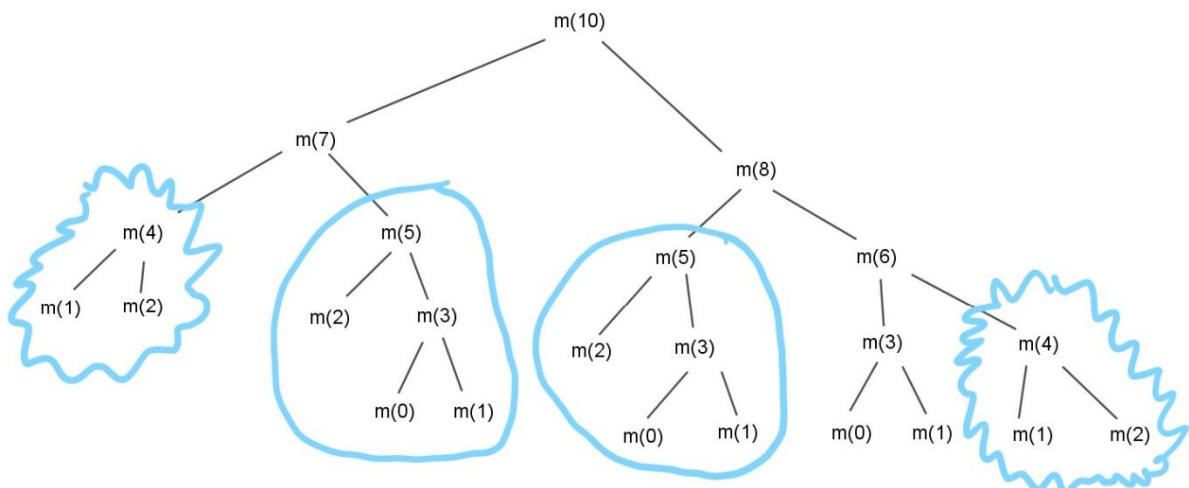
$\text{mystere}(x) = \text{mystere}(x-2) * \text{mystere}(x-3)$

- 3) Donner l'arbre récursif des appels lors de l'appel principal $\text{mystere}(10)$. On dessinera cet arbre sur la feuille A4 blanche fournie par l'enseignant.

On pourra utiliser $m(x)$ comme abréviation de $\text{mystere}(x)$.



- 4) Sur cet arbre récursif, entourer deux sous-arbres identiques qui montrent qu'on répète certains calculs plusieurs fois lors de l'appel à $\text{mystere}(10)$



On avait aussi le sous-arbre correspondant à $\text{mystere}(3)$ qui pouvait être entouré.

5) Donner la valeur renvoyée par $\text{mystere}(10)$.

$\text{mystere}(1) = 3$ et $\text{mystere}(2) = 5$ donc $\text{mystere}(4) = 15$

$\text{mystere}(0) = 2$ et $\text{mystere}(1) = 3$ donc $\text{mystere}(3) = 6$ donc (puisque $\text{mystere}(2) = 5$) $\text{mystere}(5) = 30$.
donc $\text{mystere}(7) = 15 \times 30 = 450$.

En faisant de même on trouve $\text{mystere}(6) = 6 \times 15 = 90$ donc $\text{mystere}(8) = 30 \times 90 = 2700$.

Et finalement $\text{mystere}(10) = 450 \times 2700 = 1\,215\,000$

Evaluation : partie 2 (50 minutes – 10 points)

On dispose d'une classe Piece qui offre l'interface suivante à l'utilisateur :

Classe Piece(valeur, masse)

Attributs :

- valeur : représente la valeur de la pièce de monnaie.
- masse : représente la masse de la pièce de monnaie.

Méthodes : Aucune

Note : l'affichage d'une pièce n'affiche que sa valeur et pas sa masse

```
>>> p1 = Piece(10, 20)
>>> p1.masse
20
>>> p1.valeur
10
>>> p2 = Piece(5, 12)
>>> p2.valeur
5
>>> p2
5
```

On dispose également d'une classe Tas_de_pieces qui offre l'interface suivante à l'utilisateur :

Class Tas_de_pieces()

Lors de sa création, un tas de pièces est vide.

Attributs : Aucun

Méthodes :

- ajouter(*pieces) : Ajoute une ou plusieurs pièces (de type Piece) dans le tas.
- est_vide() : Renvoie True si le tas est vide et False sinon.
- piocher() : Renvoie une seule pièce piochée au hasard dans le tas.
La pièce qui est piochée est enlevée du tas qui perd donc une pièce.
- diviser() : Renvoie deux tas de pièces en divisant au hasard le tas en deux. Si le tas comporte un nombre impair de pieces, le premier des deux tas renvoyés comporte une pièce de plus que le second ; sinon les deux tas renvoyés comportent le même nombre de pièces.
À l'issue de la division, le tas est vide.
- peser() : Renvoie la masse totale du tas de pièces.
- taille() : Renvoie le nombre de pièces présentes dans le tas de pièces.

Note : L'affichage d'un tas n'affiche que les valeurs des pièces qu'il comporte et pas leurs masses.

Un exemple d'utilisation est donné page suivante.

```

>>> p10 = Piece(10, 20)
>>> p5 = Piece(5, 12)
>>> p2 = Piece(2, 10)
>>> p1 = Piece(1, 7)
>>> mon_tas = Tas_de_pieces()
>>> mon_tas.ajouter(p10, p5, p10, p5, p1,
                    p2, p2, p1, p2, p5,
                    p2, p10, p1, p5, p2)

>>> mon_tas
      2
    5  1
  10  2  5
 2  1  2  2
1  5  10  5  10

>>> piece_a = mon_tas.piocher()
>>> piece_b = mon_tas.piocher()
>>> mon_tas
      5
    1  10  2
  5  2  1  2
 2  1  5  10  10

>>> mon_tas.taille()
13

```

```

>>> mon_tas.peser()
157

>>> tas_a, tas_b = mon_tas.diviser()
>>> tas_a
    10  1  1
    2  5  2  5

>>> tas_b
      2
    10  1
    2  10  5

>>> mon_tas.est_vide()
True

>>> tas_a.peser() + tas_b.peser()
157

>>> mon_tas.ajouter(p10)
>>> mon_tas.peser()
20
>>> mon_tas.taille()
1

```

Partie 1 : Compréhension de code

- 1) Quelle est la valeur de la pièce p5 ? et sa masse ?
Sa valeur est 5 et sa masse est 12.
- 2) Dans l'exemple de code ci-dessus, deux pièces ont été piochées dans le tas mon_tas. En comparant le tas avant et après les deux pioches, indiquer quelles sont les valeurs de ces deux pièces.
En déduire les masses de ces deux pièces.
Il s'agit d'une pièce de valeur 2 (de masse 10) et d'une pièce de valeur 1 (et de masse 7).
- 3) Refaire le calcul justifiant que la pesée du tas de pièces mon_tas dans l'exemple ci-dessus doit renvoyer 157.
3 pièces de 10 : $3 \times 20 = 60$
3 pièces de 5 : $3 \times 12 = 36$
4 pièces de 2 : $4 \times 10 = 40$
3 pièces de 1 : $3 \times 7 = 21$
total : $60 + 36 + 40 + 21 = 157$
- 4) Le tas tas_a contient sept pièces. On exécute alors l'instruction :
tas_x, tas_y = tas_a.diviser()
Combien de pièces seront présentes dans tas_x ? et dans tas_y ? et dans tas_a ?
4 pièces dans le tas_x, 3 pièces dans le tas_y et 0 pièce dans le tas_a.
- 5) Les objets de type Tas_de_pieces sont-ils mutables ou immuables ?
Les instructions ci-dessus satisfont-elles le paradigme de la programmation fonctionnelle ?
Ils sont mutables.
Non, par exemple le tas mon_tas mute de nombreuses fois.

Partie 2 : Une première fonction récursive

La fonction ci-contre prend un tas de pièces en argument.

- 1) Pourquoi lorsqu'un appel récursif est effectué à la dernière ligne de la fonction, le tas comporte-t-il une pièce en moins que lors de l'appel initial ?

Parce qu'une pièce y a été piochée.

```
def mystere(tas):  
    if tas.est_vide():  
        return 0  
    else:  
        piece = tas.piocher()  
        return piece.valeur + mystere(tas)
```

- 2) Pourquoi est-il important que la taille du tas diminue de 1 lors de chacun des appels récursifs successifs ?

Pour être certain d'atteindre le cas de base.

```
>>> p1 = Piece(1, 7)  
>>> p3 = Piece(3, 11)  
>>> p5 = Piece(5, 10)  
>>> t = Tas_de_pieces(p1, p3, p3, p1, p5,  
                       p5, p1, p1, p3, p5)  
>>> mystere(t)
```

- 3) On exécute les instructions ci-contre.
Quelle valeur renvoie la fonction mystere ?

Elle va renvoyer la somme des valeurs des différentes pièces, c'est-à-dire $4 \times 1 + 3 \times 3 + 3 \times 5 = 28$.

- 4) Que fait la fonction mystere ?

Elle renvoie la valeur totale du tas de pièces et vide le tas (la vidange du tas est un effet de bord).

Partie 3 : Trouver une fausse pièce

On rappelle que les opérateurs `//` et `%` permettent d'obtenir le quotient et le reste dans la division euclidienne de deux nombres :

```
192 // 60 = 3    # 192 contient 3 paquets de 60 (soit 180)  
192 % 60 = 12   # et il reste alors 12 unités (192 - 180)
```

Sur la planète Azerty, les pièces officielles, ont toutes la même masse : 10.

En revanche les rares fausses pièces en circulation ont toutes pour masse : 9.

On dit qu'un tas de pièces est simplement corrompu de taille N (avec N entier supérieur ou égal à 1) lorsqu'il comporte N pièces dont une seule exactement est une fausse pièce.

- 1) Cinq tas de pièces ont pour masses 600, 12 380, 989 et 4 710, 876 549.
Combien y'a-t-il de tas simplement corrompus parmi ces quatre tas ? Le(s)quel(s) ?

Il y en a deux : les tas de masses 989 et 876 549.

- 2) Soit M la masse d'un tas de pièces. Parmi les tests suivants, lequel permet de savoir si le tas est corrompu ?

$M // 10 == 9$ $M \% 10 == 9$ $M // 9 == 10$ $M \% 9 == 10$

Le chiffre des unités est donné par $M \% 10$: le test qui permet de savoir si le tas est corrompu est $M \% 10 == 9$.

- 3) Proposer une fonction récursive `trouver_fausse_piece` prenant en argument un tas de pièces simplement corrompu et qui renvoie la fausse pièce qu'il contient en opérant par divisions successives du tas en deux.

L'idée est de diviser le tas en deux tas `tas_a` et `tas_b` : en pesant `tas_a` on peut savoir s'il est simplement corrompu ou pas :

- s'il est simplement corrompu on applique la recherche de la fausse pièce à `tas_a`,

- sinon la fausse pièce est située dans `tas_b` : on applique la recherche de la fausse pièce à `tas_b`.

Le cas de base est le cas où le tas est de taille 1 (c'est-à-dire comporte une seule pièce). Dans ce cas on sait qu'il s'agit de la fausse pièce : il suffit de la piocher et de la renvoyer.

Cela donne finalement :

```
def trouver_fausse_piece(tas):
    if tas.taille() == 1:
        piece = tas.piocher()
        return piece
    else:
        tas_a, tas_b = tas.diviser()
        masse_tas_a = tas_a.peser()
        if masse_tas_a % 10 == 9:
            return trouver_fausse_piece(tas_a)
        else:
            return trouver_fausse_piece(tas_b)
```

Partie 4 : Retour sur la fonction mystere

Lorsqu'on appelle successivement deux fois la fonction mystere sur un tas, le second appel renvoie systématiquement 0.

```
>>> mystere(mon_tas)
47
>>> mystere(mon_tas)
0
```

- 1) Quel paradigme de programmation cela ne respecte-t-il pas ? **Celui de programmation fonctionnelle.**
- 2) Expliquer brièvement pourquoi cela se produit.
Parce que la fonction mystere effectue des pioches au fur et à mesure des appels récursifs dans mon_tas. À la fin du premier appel, mon_tas est par conséquent complètement vidé ! On a donc "détruit" mon_tas.
- 3) Proposer une modification du code de la fonction récursive mystere pour que le contenu du tas après l'appel à la fonction soit le même qu'avant l'appel à la fonction.
Il faut piocher une pièce, calculer la somme, remettre la pièce dans le tas puis renvoyer la somme.

```
def mystere_v2(tas):
    if tas.est_vide():
        return 0
    else:
        piece = tas.piocher()
        somme = piece.valeur + mystere_v2(tas)
        tas.ajouter(piece)
        return somme
```

Une variante plus simple dont beaucoup d'entre vous ont eu l'intuition est de copier le tas. Mais dans ce cas il faut faire attention : en python si mon_objet est une instance d'une classe que l'on a définie soi même alors l'instruction :

```
copie_objet = mon_objet
```

est une copie par référence : copie_objet et mon_objet continueront de désigner la même instance comme pour les tableaux (list python). Dans ce cas il faut essayer de recourir au module copy et à la fonction deepcopy() qui va permettre d'effectuer une vraie copie de mon_objet :

```
from copy import deepcopy

def mystere_v3(tas):
    if tas.est_vide():
        return 0
    else:
        copie_tas = deepcopy(tas)
        piece = copie_tas.piocher()
        return piece.valeur + mystere_v3(copie_tas)
```

- 4) Est-ce que votre fonction trouver_fausse_piece présente le même type de problème ?
Oui puisque lorsqu'on divise un tas, celui-ci se retrouve vidé.