**Evaluation : partie 1 (55 minutes – 20 points)**

**Exercice 1 (4 points) :**

1. Rappeler brièvement ce qui caractérise le paradigme fonctionnel.
2. Parmi les 3 séquences d'instructions suivantes, laquelle ou lesquelles ne respectent pas le paradigme fonctionnel ?

|  |  |  |
| --- | --- | --- |
| tab = [5, 6, 7, 8, 9] somme = 0 for elt in tab:  somme = somme + elt | 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) | def augmentation(nombre):  return nombre + 1  x = 123 y = augmentation(x) |

1. 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 = modulo\_dix(f), pour tout nombre x entier on a : g(x) = f(x) % 10
* Exemples d'utilisation :

|  |  |
| --- | --- |
| def plus\_sept(x):  return x+7  g = modulo\_dix(plus\_sept) g(1) = 8  g(12) = 9 g(123) = 0 | def fois\_trois(x):  return 3\*x  g = modulo\_dix(fois\_trois) g(4) = 2  g(5) = 5  g(11) = 3 |

*Rappel : l'opérateur modulo % calcule le reste dans la division euclidienne : 67 % 10 = 7, 1234 % 10 = 4, 125 % 10 = 5.***Exercice 2 (8 points) :**

Ecrire le code d'une classe Rectangle disposant :  
  
- de deux attributs longueur et largeur,  
  
- 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.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 longueur 5 et de largeur 7, le périmètre est 2x5 + 2x7 = 24 et l'aire est 5x7 = 35.

**Exercice 3 (8 points) :**

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)

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.
2. Donner la formule de récursivité correspondant au cas général.
3. Donner l'arbre récursif des appels lors de l'appel principal mystere(10). On dessinera cet arbre sur la feuille A4 blanche fournie par l'enseignant.  
   *On pourra utiliser m(x) comme abréviation de 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 mystere(10)
5. Donner la valeur renvoyée par mystere(10).

**Evaluation : partie 2 ( En devoir maison – 10 points)**

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

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

**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*

On dispose également d'une classe Tas\_de\_pieces qui offre l'interface suivante à l'utilisateur :

**Classe 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.

**Partie 1 : Compréhension de code**

>>> 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

>>> 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

1. Quelle est la valeur de la pièce p5 ? et sa masse ?
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.
3. Refaire le calcul justifiant que la pesée du tas de pièces mon\_tas dans l'exemple ci-dessus doit renvoyer 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 ?
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 ?

**Partie 2 : Une première fonction récursive**

def mystere(tas):  
 if tas.est\_vide():  
 return 0  
 else:  
 piece = tas.piocher()  
 return piece.valeur + mystere(tas)

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

>>> 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)

1. On exécute les instructions ci-contre.   
   Quelle valeur renvoie la fonction mystere ?
2. Que fait la fonction mystere ?

**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) ?
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

1. 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.

**Partie 4 : Retour sur la fonction mystere**

>>> mystere(mon\_tas)  
47  
>>> mystere(mon\_tas)  
0

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

1. Quel paradigme de programmation cela ne respecte-t-il pas ?
2. Expliquer brièvement pourquoi cela se produit.
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.
4. Est-ce que votre fonction trouver\_fausse\_piece présente le même type de problème ?