EXERCICES PILES, FILES ET POO

EXERCICE 1 (4 points)

Cet exercice porte sur les structures de données (listes, piles et files).

On cherche ici à mettre en place des algorithmes qui permettent de modifier l'ordre des informations contenues dans une file. On considère pour cela les structures de données abstraites de Pile et File définies par leurs fonctions primitives suivantes :

Pile :

- `creer_pile_vide()` renvoie une pile vide ;
- `est_pile_vide(p)` renvoie `True` si la pile `p` est vide, `False` sinon ;
- `empiler(p, element)` ajoute `element` au sommet de la pile `p` ;
- `depiler(p)` renvoie l'élément se situant au sommet de la pile `p` en le retirant de la pile `p` ;
- `sommet(p)` renvoie l'élément se situant au sommet de la pile `p` sans le retirer de la pile `p`.

File :

- `creer_file_vide()` renvoie une file vide ;
- `est_file_vide(f)` renvoie `True` si la file `f` est vide, `False` sinon ;
- `enfiler(f, element)` ajoute `element` dans la file `f` ;
- `defiler(f)` renvoie l'élément à la tête de la file `f` en le retirant de la file `f`.

On considère de plus que l'on dispose d'une fonction permettant de connaître le nombre d'éléments d'une file :

- `taille_file(f)` renvoie le nombre d'éléments de la file `f`.

On représentera les files par des éléments en ligne, l'élément de droite étant la tête de la file et l'élément de gauche étant la queue de la file. On représentera les piles en colonnes, le sommet de la pile étant le haut de la colonne.

La file suivante est appelée `f` :

4	3	8	2	1

La pile suivante est appelée `p` :

5
8
6
2

1. Les quatre questions suivantes sont indépendantes. Pour chaque question, on repartira de la pile `p` et de la file `f` initiales (présentées ci-dessus)

a. Représenter la file `f` après l'exécution du code suivant.

```
enfiler(f, defiler(f))
```

b. Représenter la pile `p` après l'exécution du code suivant.

```
empiler(p, depiler(p))
```

c. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

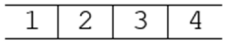
```
for i in range(2):
    enfiler(f, depiler(p))
```

d. Représenter la pile `p` et la file `f` après l'exécution du code suivant.

```
for i in range(2):
    empiler(p, defiler(f))
```

2. On donne ici une fonction `mystere` qui prend une file en argument, qui modifie cette file, mais qui ne renvoie rien.

```
def mystere(f):
    p = creer_pile_vide()
    while not est_file_vide(f):
        empiler(p, defiler(f))
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
    return p
```

Préciser l'état de la variable `f` après chaque boucle de la fonction `mystere` appliquée à la file 

Indiquer le contenu de la pile renvoyée par la fonction.

3. On considère la fonction `knuth(f)` suivante dont le paramètre est une file :

```
def knuth(f):
    p=creer_pile_vide()
    N=taille_file(f)
    for i in range(N):
        if est_pile_vide(p):
            empiler(p, defiler(f))
        else:
            e = defiler(f)
            if e >= sommet(p):
                empiler(p, e)
            else:
                while not est_pile_vide(p) and e < sommet(p):
                    enfiler(f, depiler(p))
                empiler(p, e)
    while not est_pile_vide(p):
        enfiler(f, depiler(p))
```

a. Recopier et compléter le tableau ci-dessous qui détaille le fonctionnement de cet algorithme étape par étape pour la file `2,1,3`. Une étape correspond à une modification de la pile ou de la file. Le nombre de colonnes peut bien sûr être modifié.

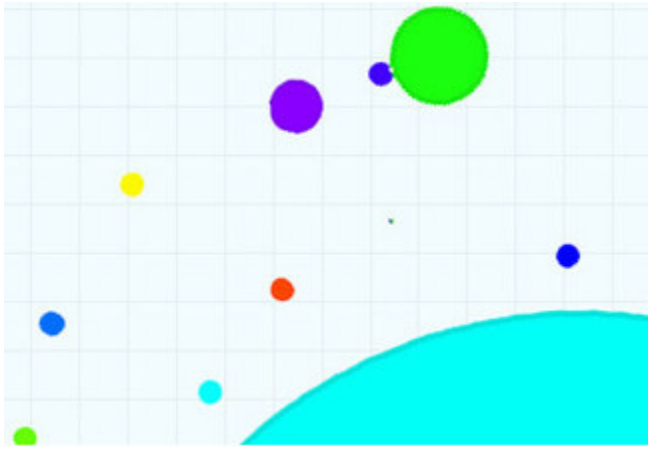
f	2,1,3	2,1									
p	⌊ ⌋	⌊ 3 ⌋									

b. Que fait cet algorithme ?

EXERCICE 2 (4 points)

Cet exercice porte sur les structures de données (programmation objet)

Dans un jeu de plateforme, des bulles de couleurs et de diamètres différents se déplacent de manière aléatoire. A chaque fois qu'une bulle touche une bulle plus grande, la petite cède son contenu à la plus grande, et donc celle-ci augmente de surface. Par exemple, si une bulle de 1 cm² rencontre une bulle de 4 cm², la petite bulle disparait et la plus grande a désormais une surface de 5 cm². A chaque collision, la vitesse de la grande bulle est réduite de moitié.



Le développeur a choisi de coder en Python, chaque bulle est un objet disposant entre autre des attributs suivants :

- `xc`, `yc` sont deux entiers, les coordonnées du pixel placé au centre de la bulle,
- `rayon` est un entier, le rayon de la bulle en pixels,
- `couleur` est un entier, la couleur de la bulle,
- `dirx, diry` sont deux décimaux (float) qui déterminent les déplacements à l'horizontale et à la verticale à chaque fois que la bulle se déplace. Ces deux valeurs déterminent donc la direction et la vitesse de la bulle. Par exemple si `dirx` vaut 0.5 et `diry` vaut 0.0, la bulle se déplace vers la droite uniquement alors que si `dirx` vaut -1.0 et `diry` vaut 0.0, la bulle se déplace vers la gauche et deux fois plus vite que précédemment.

On suppose que toutes les fonctions de la bibliothèque `math` ont déjà été importées par l'instruction `from math import *`.

La fonction `randint` de la bibliothèque `random` prend en paramètre deux entiers et renvoie un entier alétoire dans la plage définie par les deux paramètres.

Exemple : randint (-1, 5) peut renvoyer une des valeurs suivantes : -1, 0, 1, 2, 3, 4, 5.

1. Pour simplifier, on se limitera à un jeu de six bulles. Au départ, on crée une liste appelée `Mousse` de longueur six contenant six emplacements vides :

```
Mousse = [None, None, None, None, None, None]
```

Le code ci-dessous montre le début du programme et notamment la structure définition de la classe nommée `Cbulle` ainsi que le code permettant le déplacement d'une bulle.

```
from random import randint
from math import *

class Cbulle:
  def __init__(self):
    self.xc = randint(0, 100)
    self.yc = randint(0, 100)
    self.rayon = randint(0, 10)
    self.dirx = float(randint(-1, 1)) # dirx et diry valent
    self.diry = float(randint(-1, 1)) # -1.0 ou 0.0. ou 1.0
    self.couleur = randint(1,65535)

  def bouge(self):
     # déplace la bulle
     self.xc = self.xc + self.dirx
     self.yc = self.yc + self.diry
```

On crée les six bulles une à une et ces objets sont stockés dans les emplacements vides de la liste `Mousse`.

```
Mousse = [bulle1, bulle2, bulle3, bulle4, bulle5, bulle6]
```

Lors d'une collision, la bulle la plus petite disparait et est remplacée dans la liste par la valeur `None` tandis que la plus grosse a sa surface qui augmente.

Au cours d'une partie, si une ou plusieurs bulles ont disparue, le programme peut en introduire de nouvelles dans le jeu: dans ce cas, lorsqu'une nouvelle bulle apparaît, elle remplace le premier `None` de la liste `Mousse`.

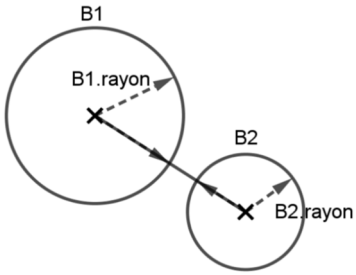
a. Recopier les quatre dernières lignes et compléter les ..... du code python ci-dessous.

```
def donnePremierIndiceLibre(Mousse):
     """
      Mousse est une liste.
      La fonction doit renvoyer l'indice du premier
      emplacement libre (contenant None) dans la liste Mousse
      ou renvoyer 6 en l'absence d'un emplacement libre dans
      Mousse.
     """
     i = 0
     while ..... and Mousse[i] != None :
          .....
     return i
```

b. Lorsque le jeu crée une bulle (instance de la classe `Cbulle`), il doit ensuite la placer dans la liste `Mousse` à la place d'un `None`.
Ecrire la fonction `placeBulle(B)` qui reçoit en paramètre un objet de type `Cbulle` et qui place cet objet dans la liste `Mousse`. Cette fonction ne renvoie rien, mais la liste `Mousse` est modifiée. Si aucun emplacement n'est disponible, la fonction ne modifie rien.

2. Pour le bon déroulement du jeu, on a besoin aussi d'une fonction `bullesEnContact(B1, B2)` qui renvoie `True` si la bulle `B2` touche la bulle `B1` et `False` dans le cas contraire.

On peut remarquer que deux bulles sont en contact si la distance qui sépare leur centre est inférieure ou égale à la somme de leurs rayons.



On dispose de la fonction `distanceEntreBulles(B1, B2)` qui calcule et renvoie la distance entre les centres de bulles B1 et B2.

Ecrire la fonction `bullesEnContact(B1, B2)`.

3. Quand une petite bulle touche une plus grosse bulle, on appelle la fonction `collision`, ci-dessous, où `indPetite` est l'indice de la petite bulle et `indGrosse` l'indice de la grosse bulle dans `Mousse`.
Recopier et compléter les `.....` de la fonction `collision`.

```
def collision(indPetite, indGrosse, mousse) :
    """
    Absorption de la plus petite bulle d'indice indPetite
    par la plus grosse bulle d'indice indGrosse. Aucun test
    n'est réalisé sur les positions.
    """
    # calcul du nouveau rayon de la grosse bulle
    surfPetite = pi*Mousse [indPetite].rayon**2
    surfGrosse = pi*Mousse [indGrosse].rayon**2
    surfGrosseApresCollision = .....
    rayonGrosseApresCollision = sqrt(surfGrosseApresCollision/pi)
    #réduction de 50% de la vitesse de la grosse bulle
    Mousse[indGrosse].dirx = .....
    Mousse [indGrosse].diry = .....
    #suppression de la petite bulle dans Mousse
    .....
```