# Programmation objet

**Exercice 1**
On considère un fichier Python contenant le code suivant :

```
class Compteur:
    def __init__(self,initial,pas):
        self.val = initial
        self.pas = pas

    def avance(self):
        self.val += self.pas

csimple = Compteur(0,1)
```

1. Déterminer tous les éléments dans ce code qui sont des identifiants de :

   (a) classe (b) objet (c) attribut (d) méthode

2. Écrire un code Python permettant de créer une instance `rebours` de `Compteur` ayant pour valeur initiale `10` et un pas de `-1`.
3. Écrire le code Python permettant d'exécuter la méthode `avance` de `rebours`.
   Indiquer (en commentaire ou via un `assert` dans le code) les valeurs des attributs de `rebours` après cette exécution (sans faire de `print`).
4. Donner la définition de la méthode `copie(self)` de la classe `Compteur` qui retourne un compteur ayant les mêmes attributs que l'objet `self`
5. Écrire du code Python réalisant les opérations suivantes :
   - Créer un objet `c1` de la classe `Compteur` avec pour paramètres `init=0` et `pas=2` ;
   - Faire avancer 3 fois (à l'aide d'une boucle) le compteur `c1` ;
   - obtenir une copie `c2` du compteur `c1` ;
   - Faire avancer 2 fois le compteur `c1` ;
   - Afficher la valeur du compteur `c2` et celle de `c1`.
6. Plutôt que d'utiliser `print(c.val)` pour afficher la valeur d'un compteur `c`, on voudrait pouvoir simplement écrire `print(c)`.
   Définir pour cela dans la classe `Compteur` la méthode `__str__(self)` qui retourne la chaîne de caractère associée à l'attribut `val` de l'objet `self`.

   ⚠ la méthode `__str__(self)` n'exécute pas `print` mais retourne une chaîne de caractère !
   Tester alors la méthode en créant un objet `c` et en exécutant l'instruction `print(c)`.

**Exercice 2**
Définir une classe `Point` pour représenter un point du plan. Cette classe possède deux attributs privés, `_x` et `_y` qui sont des réels et désignent respectivement l'abscisse et l'ordonnée du point.

1. Écrire le constructeur de cette classe et une méthode `coordonnees(self)` qui retourne le couple de ses coordonnées.

```
>>> p = Point(5,3)
>>> p.coordonnees()
(5, 3)
```

2. Ajouter une méthode `norme(self)` qui retourne la distance de l'origine du repère au point.

```
>>> p.norme()
5.830951894845301
```

3. Ajouter une méthode `distance(self, q)` qui retourne la distance de `self` au point `q`.

```
>>> q = Point(7,3)
>>> p.distance(q)
2.0
```

4. Ajouter une méthode `__str__(self)` qui renvoie une chaîne de caractère de la forme `(x;y)`.

```
>>> print(p)
(5;3)
>>> str(p)
'(5;3)'
```

**Exercice 3**

Définir une classe `Cercle` qui représente un cercle du plan. Cette classe possède deux attributs privés, `_centre` qui représente le centre du cercle et qui est de la classe `Point` définie dans l'exercice précédent, et son rayon `_r`.

1. Écrire le constructeur `__init__(self,centre,rayon)` de cette classe.
2. Écrire une méthode `perimetre(self)` et une méthode `surface(self)` qui retournent respectivement le périmètre et la surface du cercle.
3. Écrire une méthode `__str__(self)` qui retourne les coordonnées du centre et le rayon sous la forme d'une chaîne de caractère comme par exemple `'(4;3) 2'`.

   On pourra utiliser la méthode de la classe `Point` pour obtenir les coordonnées du centre.
4. Écrire une méthode `possede(self, q)` qui retourne `True` si le point `q` appartient au cercle (le cercle possède le point) et `False` sinon.

   Juger la pertinence de cette fonction.
5. Même question avec la méthode `contient(self, q)` qui retourne `True` si le point `q` est à l'intérieur du cercle et `False` sinon.
6. Tester ces opérations.

**Exercice 4**

On souhaite définir une classe `Date` pour représenter une date avec trois attributs nommés `jour`, `mois` et `annee`, toutes les trois de type `int` (le mois est donc un entier de 1 à 12).

1. Écrire son constructeur (la méthode `__init__(self,j,m,a)`.
2. Ajouter une méthode `__str__(self)` qui renvoie une chaîne de caractères de la forme : `"8 mai 1945"`. On pourra se servir d'un attribut de classe donnant les noms des douze mois de l'année que l'on utilisera pour traduire le numéro du mois en son nom.

   Tester en construisant des objets de la classe `Date` puis en les affichant avec `print`.
3. Ajouter une méthode `__lt__(self,d)` qui permet de déterminer si la date `self` est antérieure à la date `d`.

   Tester alors la méthode. On rappelle qu'étant donnés deux objets `d1` et `d2` de la classe, on pourra alors utiliser la syntaxe `d1 < d2`.

**Exercice 5 (Extrait d'épreuve écrite)**
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 |
| Protéine (grammes) | 3.32 | 11.7 | 0 |
| Glucides (grammes) | 4.85 | 69.3 | 0 |
| Lipides (grammes) | 3.63 | 0.8 | 100 |

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.

```
class Aliment:
    def __init__(self, e, p, g, l):
        self.energie = e
        self.proteines = p
        self.glucides = g
        self.lipides = l
```

1. (a) Écrire, à l'aide du tableau des caractéristiques nutritionnelles, l'instruction en langage Python pour instancier l'objet `lait`.
   (b) Donner l'instruction qui permet d'obtenir la valeur 65.1 de l'objet `lait` instancié dans la question précédente.
   (c) En fait, 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 1a.
2. 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.

   **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`.

   Recopier et compléter les lignes dans la méthode ci-dessous :

   ```
   def energie_reelle(...,masse):
       return ...
   ```

3. On regroupe les caractéristiques nutritionnelles du tableau 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'instruction qui permet d'obtenir la valeur énergétique en kcal du lait à partir des données de ce dictionnaire.
   (b) Donner l'instruction qui permet d'obtenir la valeur énergétique réelle de 220 grammes de lait à partir des données de ce dictionnaire.

4. 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}
```

Écrire, en utilisant la méthode `energie_reelle` de la classe `Aliment`, une définition en Python de la fonction `calcul_energie_tot(recette, nutrition)` (hors de la classe) qui calcule et retourne l'énergie réelle totale d'une `recette` étant données les informations de `nutrition`.
On doit alors par exemple obtenir, dans la console :

```
>>> calcul_energie_tot(recette_gateau, nutrition)
1832.12
```

**Exercice 6 (Extrait des épreuves pratiques)**
On dispose d'un programme permettant de créer un objet de type `PaquetDeCarte`, selon les éléments indiqués dans le code ci-dessous.
Compléter ce code aux endroits indiqués par *#A compléter*, puis ajouter des assertions dans l'initialiseur de `Carte`, ainsi que dans la méthode `getCarteAt()`.

```
class Carte:
  """Initialise Couleur (entre 1 à 4), et Valeur (entre 1 à 13)"""
  def __init__(self, c, v):
    self.Couleur = c
    self.Valeur = v

  """Renvoie le nom de la Carte As, 2, ... 10, Valet, Dame, Roi"""
  def getNom(self):
    if ( self.Valeur > 1 and self.Valeur < 11):
      return str( self.Valeur)
    elif self.Valeur == 11:
      return "Valet"
    elif self.Valeur == 12:
      return "Dame"
    elif self.Valeur == 13:
      return "Roi"
    else:
      return "As"

  """Renvoie la couleur de la Carte (pique, coeur, carreau, trefle)"""
  def getCouleur(self):
    return ['pique', 'coeur', 'carreau', 'trefle' ][self.Couleur-1]

class PaquetDeCarte:
  def __init__(self):
    self.contenu = []

  """Remplit le paquet de cartes"""
  def remplir(self):
    #A compléter

  """Renvoie la Carte qui se trouve à la position donnée"""
  def getCarteAt(self, pos):
    #A compléter
```

Exemple :

```
>>> unPaquet = PaquetDeCarte()
>>> unPaquet.remplir()
>>> uneCarte = unPaquet.getCarteAt(20)
>>> print(uneCarte.getNom() + " de " + uneCarte.getCouleur())
6 de coeur
```

**Exercice 7 (Extrait des épreuves pratiques)**
On définit une classe gérant une adresse IPv4.
On rappelle qu'une adresse IPv4 est une adresse de longueur 4 octets, notée en décimale à point, en séparant chacun des octets par un point. On considère un réseau privé avec une plage d'adresses IP de 192.168.0.0 à 192.168.0.255.
On considère que les adresses IP saisies sont valides.
Les adresses IP 192.168.0.0 et 192.168.0.255 sont des adresses réservées.
Le code ci-dessous implémente la classe AdresseIP.

```
class AdresseIP:
    def __init__(self, adresse):
        self.adresse = ...

    def liste_octet(self):
        """renvoie une liste de nombres entiers,
           la liste des octets de l'adresse IP"""
        return [int(i) for i in self.adresse.split(".")]

    def est_reservee(self):
        """renvoie True si l'adresse IP est une adresse
           réservée, False sinon"""
        return ... or ...

    def adresse_suivante(self):
        """renvoie un objet de AdresseIP avec l'adresse
           IP qui suit l'adresse self
           si elle existe et False sinon"""
        if ... < 254:
            octet_nouveau = ... + ...
            return AdresseIP('192.168.0.' + ...)
        else:
            return False
```

Compléter le code ci-dessus et instancier trois objets : `adresse1`, `adresse2`, `adresse3` avec respectivement les arguments suivants : `'192.168.0.1'`, `'192.168.0.2'`, `'192.168.0.0'`
Vérifier que :

```
>>> adresse1.est_reservee()
False
>>> adresse3.est_reservee()
True
>>> adresse2.adresse_suivante().adresse
'192.168.0.3'
```

**Exercice 8 (Facultatif)**
Dans certains langages de programmation, comme Pascal ou Ada, les tableaux ne sont pas forcément indexés à partir de 0. C'est le programmeur qui choisit sa plage d'indices. Par exemple, on peut déclarer un tableau dont les indices vont de -10 à 9 si on le souhaite. Dans cet exercice, on se propose de construire une classe `Tableau` pour réaliser de tels tableaux.
Un objet de cette classe aura deux attributs, un attribut `premier` qui est la valeur de premier indice et un attribut `contenu` qui est un tableau Python contenant les éléments. Ce dernier est un vrai tableau Python, indexé à partir de 0.

1. Écrire un constructeur `__init__(self, imin, imax, v)` où `imin` est le premier indice, `imax` est le dernier indice et `v` la valeur utilisée pour initialiser toutes les cases du tableau. Ainsi, on peut écrire `t = Tableau(-10,9,42)` pour construire un tableau de vingt cases, indexées de -10 à 9 et toutes initialisées avec la valeur 42.
2. Écrire une méthode `__len__(self)` qui renvoie la taille du tableau.
3. Écrire une méthode `__getitem__(self, i)` qui renvoie l'élément du tableau `self` d'indice `i`. De même écrire une méthode `__setitem__(self, i, v)` qui modifie l'élément d'indice `i` pour lui donner la valeur `v`. Ces deux méthodes doivent vérifier que l'indice `i` est bien valide et , dans le cas contraire, lever l'exception `IndexError` avec la valeur `i` en argument (`raise IndexError(i)`).
4. Enfin, écrire une méthode `__str__(self)` qui renvoie une chaîne de caractères décrivant le contenu du tableau.

**Exercice 9 (Facultatif)**
On veut définir une classe `TaBiDir` pour des tableaux bidirectionnels, dont une partie des éléments ont des indices positifs et une partie des éléments des indices négatifs, et qui sont extensibles aussi bien par la gauche que la droite. Plus précisément, les indices d'un tel tableau bidirectionnel vont aller d'un indice $i_{min}$ à un indice $i_{max}$, tous deux inclus, et tels que $i_{min} \leq 0$ et $-1 \leq i_{max}$. Le tableau bidirectionnel vide correspond au cas où $i_{min}$ vaut 0 et $i_{max}$ vaut -1.
La classe `TaBiDir` a pour attributs deux tableaux Python : un tableau `droite` contenant l'élément d'indice `0` et les autres éléments d'indices positifs, et un tableau `gauche` tel que `gauche[0]` contient l'élément d'indice `-1` du tableau bidirectionnel, et `gauche[1]` et `gauche[2]`, etc. contiennent les éléments d'indices négatifs suivants, en progressant vers la gauche.

- Écrire un constructeur `__init__(self, g, d)` construisant un tableau bidirectionnel contenant, dans l'ordre, les éléments des tableaux `g` et `d`. Le dernier élément de `g` (si `g` n'est pas vide), devra être calé sur l'indice `-1` du tableau bidirectionnel, et le premier élément de `d` (si `d` n'est pas vide) sur l'indice `0`. Écrire également les méthodes `imin(self)` et `imax(self)` renvoyant respectivement l'indice minimum et maximum.
- Ajouter une méthode `append(self, v)`, qui ajoute un élément `v` à droite du tableau bidirectionnel et une méthode `prepend(self, v)` ajoutant l'élément `v` à gauche du tableau bidirectionnel.
- Ajouter une méthode `__getitem__(self, i)` qui renvoie l'élément du tableau bidirectionnel `self` à l'indice `i`, et une méthode `__setitem__(self,i,v)` qui modifie l'élément du tableau `self` d'indice `i` pour lui donner la valeur `v`.
- Ajouter une méthode `__str__(self)` qui renvoie une chaîne de caractères décrivant le contenu du tableau.

## EXERCICE 5 (4 points)

*Cet exercice porte sur la Programmation Orientée Objet.*

Les participants à un jeu de LaserGame sont répartis en équipes et s'affrontent dans ce jeu de tir, revêtus d'une veste à capteurs et munis d'une arme factice émettant des infrarouges.

Les ordinateurs embarqués dans ces vestes utilisent la programmation orientée objet pour modéliser les joueurs. La classe `Joueur` est définie comme suit :

```
class Joueur:
    def __init__(self, pseudo, identifiant, equipe):
        ''' constructeur '''
        self.pseudo = pseudo
        self.equipe = equipe
        self.id = identifiant
        self.nb_de_tirs_emis = 0
        self.liste_id_tirs_recus = []
        self.est_actif = True

    def tire(self):
        '''méthode déclenchée par l'appui sur la gachette'''
        if self.est_actif == True:
            self.nb_de_tirs_emis = self.nb_de_tirs_emis + 1

    def est_determine(self):
        '''methode qui renvoie True si le joueur réalise un
        grand nombre de tirs'''
        return self.nb_de_tirs_emis > 500

    def subit_un_tir(self, id_recu):
        '''méthode déclenchée par les capteurs de la
        veste'''
        if self.est_actif == True:
            self.est_actif = False
            self.liste_id_tirs_recus.append(id_recu)
```

**1.** Parmi les instructions suivantes, recopier celle qui permet de déclarer un objet `joueur1`, instance de la classe `Joueur`, correspondant à un joueur dont le pseudo est "Sniper", dont l'identifiant est 319 et qui est intégré à l'équipe "A":

**Instruction 1 :** `joueur1 = ["Sniper", 319, "A"]`

**Instruction 2 :** `joueur1 = new Joueur["Sniper", 319, "A"]`

**Instruction 3 :** `joueur1 = Joueur("Sniper", 319, "A")`

**Instruction 4 :**
```
joueur1 = Joueur{"pseudo":"Sniper",
                 "id":319, "equipe":"A"}
```

**2.** La méthode `subit_un_tir` réalise les actions suivantes :
Lorsqu'un joueur actif subit un tir capté par sa veste, l'identifiant du tireur est ajouté à l'attribut `liste_id_tirs_recus` et l'attribut `est_actif` prend la valeur `False` (le joueur est désactivé). Il doit alors revenir à son camp de base pour être de nouveau actif.

**a.** Écrire la méthode `redevenir_actif` qui rend à nouveau le joueur actif uniquement s'il était précédemment désactivé.

**b.** Écrire la méthode `nb_de_tirs_recus` qui renvoie le nombre de tirs reçus par un joueur en utilisant son attribut `liste_id_tirs_recus`.

**3.** Lorsque la partie est terminée, les participants rejoignent leur camp de base respectif où un ordinateur, qui utilise la classe `Base`, récupère les données.
La classe `Base` est définie par :
- ses attributs :
  - `equipe` : nom de l'équipe (`str`). Par exemple, "A" ,
  - `liste_des_id_de_l_equipe` qui correspond à la liste (`list`) des identifiants connus des joueurs de l'équipe,
  - `score` : score (`int`) de l'équipe, dont la valeur initiale est 1000 ;
- ses méthodes :
  - `est_un_id_allie` qui renvoie `True` si l'identifiant passé en paramètre est un identifiant d'un joueur de l'équipe, `False` sinon,
  - `incremente_score` qui fait varier l'attribut `score` du nombre passé en paramètre,
  - `collecte_information` qui récupère les statistiques d'un participant passé en paramètre (instance de la classe `Joueur`) pour calculer le score de l'équipe .

```
1 def collecte_information(self,participant):
2    if participant.equipe == self.equipe : # test 1
3       for id in participant.liste_id_tirs_recus:
4          if self.est_un_id_allie(id): # test 2
5             self.incremente_score(-20)
6          else:
7             self.incremente_score(-10)
```

**a.** Indiquer le numéro du test (**test 1** ou **test 2**) qui permet de vérifier qu'en fin de partie un participant égaré n'a pas rejoint par erreur la base adverse.

**b.** Décrire comment varie quantitativement le score de la base lorsqu'un joueur de cette équipe a été touché par le tir d'un coéquipier.

On souhaite accorder à la base un bonus de 40 points pour chaque joueur particulièrement déterminé (qui réalise un grand nombre de tirs).

**4.** Recopier et compléter, en utilisant les méthodes des classes `Joueur` et `Base`, les 2 lignes de codes suivantes qu'il faut ajouter à la fin de la méthode `collecte_information` :

```
........ #si le participant réalise un grand nombre de tirs
    ......... #le score de la Base augmente de 40
```

## EXERCICE 4 (4 points)

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

Simon souhaite créer en Python le jeu de cartes « la bataille » pour deux joueurs. Les questions qui suivent demandent de reprogrammer quelques fonctions du jeu. On rappelle ici les règles du jeu de la bataille :

Préparation

- Distribuer toutes les cartes aux deux joueurs.
- Les joueurs ne prennent pas connaissance de leurs cartes et les laissent en tas face cachée devant eux.

Déroulement

- A chaque tour, chaque joueur dévoile la carte du haut de son tas.
- Le joueur qui présente la carte ayant la plus haute valeur emporte les deux cartes qu'il place sous son tas.
- **Les valeurs des cartes sont :** dans l'ordre de la plus forte à la plus faible : As, Roi, Dame, Valet, 10, 9, 8, 7, 6, 5, 4, 3 et 2 (la plus faible)

Si deux cartes sont de même valeur, il y a "bataille".

- Chaque joueur pose alors une carte face cachée, suivie d'une carte face visible sur la carte dévoilée précédemment.
- On recommence l'opération s'il y a de nouveau une bataille sinon, le joueur ayant la valeur la plus forte emporte tout le tas.

Lorsque l'un des joueurs **possède toutes les cartes du jeu**, la partie s'arrête et ce dernier gagne.

Pour cela Simon crée une classe Python `Carte`. Chaque instance de la classe a deux attributs : un pour sa valeur et un pour sa couleur. Il donne au valet la valeur 11, à la dame la valeur 12, au roi la valeur 13 et à l'as la valeur 14. La couleur est une chaîne de caractères : `"trefle"`, `"carreau"`, `"coeur"` ou `"pique"`.

**1.** Simon a écrit la classe Python `Carte` suivante, ayant deux attributs `valeur` et `couleur`, et dont le constructeur prend deux arguments : `val` et `coul`.

**a.** Recopier et compléter les [.....] des lignes 3 et 4 ci-dessous.

```
1. class Carte:
2.       def __init__(self, val, coul):
3.           [.....].valeur = [.....]
4.           [.....] = coul
```

**b.** Parmi les propositions ci-dessous quelle instruction permet de créer l'objet « 7 de cœur » sous le nom `c7` ?

- `c7.__init__(self, 7, "coeur")`
- `c7 = Carte(self, 7, "coeur")`
- `c7 = Carte(7, "coeur")`
- `from Carte import 7, "coeur"`

**2.** On souhaite créer le jeu de cartes. Pour cela, on écrit une fonction `initialiser()` :

- sans paramètre
- qui renvoie une liste de 52 objets de la classe `Carte` représentant les 52 cartes du jeu.

  Voici une proposition de code. Recopier et compléter les lignes suivantes pour que la fonction réponde à la demande :

```
def initialiser() :
  jeu = []
  for c in ["coeur", "carreau", "trefle", "pique"] : # couleur carte
    for v in range(...) :                            # valeur carte
      carte_cree  = ...
      jeu.append(carte_cree)
  return jeu
```

**3.** On rappelle que dans une partie de bataille, les deux joueurs tirent chacun une carte du dessus de leur tas, et celui qui tire la carte la plus forte remporte les deux cartes et les place en dessous de son tas.

Parmi les structures linéaires de données suivantes : Tableau, File, Pile, quelle est celle qui modélise le mieux un tas de cartes dans ce jeu de la bataille ? Justifier votre choix.

**4.** Écrire une fonction `comparer(carte1, carte2)` qui prend en paramètres deux objets de la classe `Carte`. Cette fonction renvoie :

- `0` si la force des deux cartes est identique,
- `1` si la carte `carte1` est strictement plus forte que `carte2`
- `-1` si la carte `carte2` est strictement plus forte que `carte1`

## Exercice 4

*Thème abordé : programmation objet en langage Python*

*Un fabricant de brioches décide d'informatiser sa gestion des stocks. Il écrit pour cela un programme en langage Python. Une partie de son travail consiste à développer une classe* `Stock` *dont la première version est la suivante :*

```
class Stock:
  def __init__(self):
    self.qt_farine = 0  # quantité de farine initialisée à 0 g
    self.nb_oeufs = 0   # nombre d'œufs (0 à l'initialisation)
    self.qt_beurre = 0  # quantité de beurre initialisée à 0 g
```

1. Écrire une méthode `ajouter_beurre(self, qt)` qui ajoute la quantité `qt` de beurre à un objet de la classe `Stock`.

On admet que l'on a écrit deux autres méthodes `ajouter_farine` et `ajouter_oeufs` qui ont des fonctionnements analogues.

2. Écrire une méthode `afficher(self)` qui affiche la quantité de farine, d'œufs et de beurre d'un objet de type `Stock`. L'exemple ci-dessous illustre l'exécution de cette méthode dans la console :

```
>>> mon_stock = Stock()
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 0
>>> mon_stock.ajouter_beurre(560)
>>> mon_stock.afficher()
farine: 0
oeuf: 0
beurre: 560
```

3. Pour faire une brioche, il faut 350 g de farine, 175 g de beurre et 4 œufs. Écrire une méthode `stock_suffisant_brioche(self)` qui renvoie un booléen : `VRAI` s'il y a assez d'ingrédients dans le stock pour faire une brioche et `FAUX` sinon.
4. On considère la méthode supplémentaire `produire(self)` de la classe `Stock` donnée par le code suivant :

```
def produire(self):
```

```
    res = 0
    while self.stock_suffisant_brioche():
      self.qt_beurre = self.qt_beurre - 175
      self.qt_farine = self.qt_farine - 350
      self.nb_oeufs = self.nb_oeufs - 4
      res = res + 1
    return res
```

On considère un stock défini par les instructions suivantes :

```
>>> mon_stock=Stock()
>>> mon_stock.ajouter_beurre(1000)
>>> mon_stock.ajouter_farine(1000)
>>> mon_stock.ajouter_oeufs(10)
```

**a.** On exécute ensuite l'instruction

```
>>> mon_stock.produire()
```

Quelle valeur s'affiche dans la console ? Que représente cette valeur ?

**b.** On exécute ensuite l'instruction

```
>>> mon_stock.afficher()
```

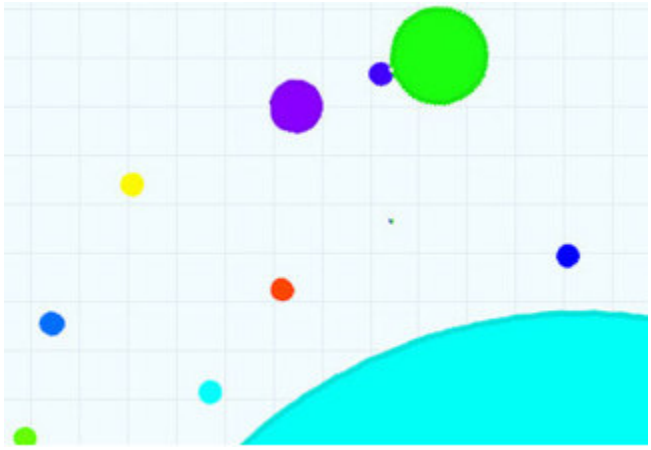
Que s'affiche-t-il dans la console ?

**5.** L'industriel possède `n` lieux de production distincts et donc `n` stocks distincts. On suppose que ces stocks sont dans une liste dont chaque élément est un objet de type `Stock`. Écrire une fonction Python `nb_brioches(liste_stocks)` possédant pour unique paramètre la liste des stocks et renvoie le nombre total de brioches produites.

## 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`.

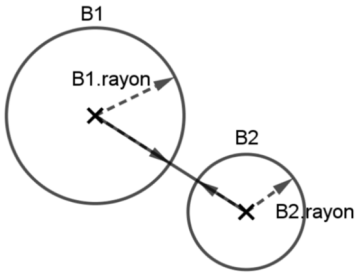
**a.** Recopier les quatre dernières lignes et compléter les ..... du code python ci-dessous.

```
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
    .....
```