

# Exercices de POO pour la 1<sup>re</sup> B

## Exercice 1 : Robots

### PARTIE A

Créer la classe **Robot** avec les attributs **name** (de type **str**), **position** (tuple à deux coordonnées entières, valeur par défaut **(0,0)**) et **direction** (de type **str**, prenant l'une des 4 valeurs '**Nord**', '**Est**', '**Sud**' ou '**Ouest**', la valeur par défaut est '**Est**').

La classe **Robot** a les méthodes suivantes :

- le constructeur **\_\_init\_\_** : initialise les attributs sur les valeurs passés aux paramètres ;
- **\_\_str\_\_** : retourne un string du type : "**<nom> se trouve en <pos> et pointe vers <direction>**".
- **move(self,n)** : déplace le robot de **n** unités dans la direction actuelle ;
- **turn(self,n)** : change la direction du robot en le tournant de **n \* 90°** dans le sens des aiguilles d'une montre.

### PARTIE B

Compléter et modifier la classe **Robot** :

- (1) Modifier la méthode **move(self)** de façon à ce que le robot ne sort jamais de son espace de vie : **[-xmax,xmax]x[-ymax,ymax]**. On pourra ajouter **xmax = 2** et **ymax = 2** comme attributs de classe. (On peut supposer que ce sont des entiers.)
- (2) Ecrire la méthode **set\_random\_position(self)** qui fixe aléatoirement les coordonnées du robot dans son espace de vie.
- (3) Ecrire la méthode **random\_move(self)** qui change la direction de marche du robot de façon aléatoire, puis fait avancer le robot d'un pas dans cette direction.
- (4) Ecrire la méthode **hits(self,other)**, qui retourne **True** lorsque le robot (**self**) se trouve au même endroit qu'un autre robot (**other**), **False** sinon.

### PARTIE C

Dans la programme principal (main.py), on simulera une marche aléatoire de deux robots A et B stockés dans deux variables **r1** et **r2** respectivement. Les deux robots partent d'une position aléatoire (pas nécessairement la même) et font aléatoirement **n** pas, **n** étant fixé par l'utilisateur. Les « instants » et « lieux » de rencontre seront stockés dans un tableau **rencontres** et seront affichés à la fin. Voici un exemple d'exécution du programme, avec **xmax = ymax = 2** :

```

Initialisation :
A se trouve en (-1, 0) et pointe vers Est
B se trouve en (0, 1) et pointe vers Est
Entrez le nombre de pas : 20
Step 1: A(-1, 1)      B(1, 1)
Step 2: A(0, 1)       B(1, 0)
Step 3: A(1, 1)       B(1, -1)
Step 4: A(1, 0)       B(1, 0)
Step 5: A(0, 0)       B(0, 0)
Step 6: A(0, -1)      B(-1, 0)
Step 7: A(-1, -1)     B(-1, -1)
Step 8: A(-1, -2)     B(-1, -2)
Step 9: A(0, -2)      B(-1, -1)
Step 10: A(0, -2)     B(-1, -2)
Step 11: A(-1, -2)    B(-1, -1)
Step 12: A(-1, -2)    B(-1, -1)
Step 13: A(0, -2)     B(-1, -2)
Step 14: A(-1, -2)    B(-1, -1)
Step 15: A(-1, -1)    B(-1, -1)
Step 16: A(-1, -1)    B(0, -1)
Step 17: A(-1, -1)    B(-1, -1)
Step 18: A(-1, -1)    B(-1, 0)
Step 19: A(0, -1)     B(-1, 0)
Step 20: A(-1, -1)    B(-1, 0)
Rencontres :
[(4, (1, 0)), (5, (0, 0)), (7, (-1, -1)), (8, (-1, -2)), (15, (-1, -1)), (17,
(-1, -1))]

```

## Exercice 2 : Fractions – extension de l'exemple du cours

Le but de cet exercice est d'ajouter des méthodes magiques à la classe des fractions.

Un article complet assez cocasse à ce sujet se trouve ici :

<http://sametmax.com/le-guide-ultime-et-definitif-sur-la-programmation-orientee-objet-en-python-a-lusage-des-debutants-qui-sont-rassures-par-les-textes-detailles-qui-prennent-le-temps-de-tout-expliquer-partie-6/>

Appeler la nouvelle classe : **Rational**

- (1) Ecrire la méthode **simplify**, qui permet de simplifier une fraction. (Définir avant la classe la fonction **gcd(a,b)** qui retourne le pgcd de deux entiers donnés.)
- (2) Redéfinir la méthode **\_\_init\_\_**, de façon à ce que a) l'utilisateur ne puisse plus entrer comme dénominateur 0 et b) la version simplifiée de la fraction est stockée dans les attributs **num** et **denom**.
- (3) Ajouter la méthode **\_\_str\_\_** qui retourne
  - a. un string du type '**a**' lorsque la fraction est un entier **a**
  - b. un string du type '**a/b**' sinon
- (4) Ajouter la méthode **\_\_float\_\_** qui permet de convertir une fraction en un nombre à virgule flottante. L'effet magique de cette méthode est qu'on pourra effectivement utiliser la fonction **float** sur les fractions dans le programme principal. On dit qu'on a **surchargé** la fonction **float**.

- (5) Ajouter les méthodes magiques `__add__`, `__sub__`, `__mul__`, et `__truediv__` qui **surchargent** les opérateurs `+`, `-`, `*` et `/` respectivement.
- (6) Ajouter la méthode magique `__pow__` qui surcharge l'opérateur `**` de sorte que l'on pourra calculer `(a/b)**n` où l'exposant `n` est un entier (positif ou négatif).
- (7) Ajouter la méthode `__eq__` qui permet de comparer deux fractions à l'aide de l'opérateur `==`.
- (8) Tester toutes les méthodes de la classe **Rational** dans le programme principal (`main.py`)

### Exercice 3 : Compte bancaire – extension de l'ex 3.2 du cours

Modifier la classe **Bankaccount** :

- (1) Ajouter un attribut de classe `number_of_accounts` qui permettra de compter le nombre de comptes créés.
- (2) Modifier le constructeur :
  - a) Le nombre de comptes est incrémenté à chaque appel.
  - b) Si l'utilisateur ne précise pas de nom lors de l'appel du constructeur, le nom par défaut est `'CUSTOMER' + 'x'` où `x` est le nombre de comptes créés, donc `'CUSTOMER1'`, `'CUSTOMER2'`, ...
  - c) Ajouter un **attribut privé** `__code` (deux `_` avant le nom de l'attribut !) qui permet de stocker le mot de passe de l'utilisateur pour accéder à son compte. Retenez que ceci ne protège absolument pas le mot de passe. Googlez pour trouver l'astuce qui permet d'accéder au mot de passe en dehors de la classe.
- (3) Modifier la méthode `withdraw(self, amount)` : pour retirer de l'argent, l'utilisateur doit entrer le code secret correct, sinon il obtient un message d'erreur. De même lorsque le montant à retirer est trop grand.

**Exemple d'exécution** : ALBERT qui avait 2000 € veut retirer 70 € ou 3000 €:

```
ALBERT, ENTER YOUR CODE : 1234
CORRECT CODE !
NEW BALANCE ALBERT : 1930
```

ou :

```
ALBERT, ENTER YOUR CODE : 1235
WRONG CODE !
```

ou :

```
ALBERT, ENTER YOUR CODE : 1234
CORRECT CODE !
INSUFFICIENT FUNDS !
```

- (4) Ajouter une méthode `transaction(self, other, amount)` qui permet à l'utilisateur de transférer un montant (`amount`) du compte en cours (`self`) vers

un autre compte (**other**), à condition qu'il entre le bon mot de passe pour le compte en cours et que le montant n'est pas trop grand.

Exemple d'exécution :

```
ALBERT, ENTER YOUR CODE : 1234
CORRECT CODE !
-----
FROM          TO          AMOUNT
-----
ALBERT        DUMONT      400

NEW BALANCE ALBERT : 1530

NEW BALANCE DUMONT : 3400
```

### Exercice 4 : Point – Droites - Rationnels

Le but est d'implémenter en Python les droites du plan passant par deux points aux *coordonnées supposées entières*.

- (1) Copier les fiches avec les classes **Point** (exercice 2.1 du cours) et **Rational** (exercice 2 ci-dessus) dans de nouvelles fiches du projet. On retiendra uniquement les méthodes `__init__`, `__str__`, `__eq__` de la classe **Point**. Ajouter sur la fiche avec la classe **Point** une classe **Vector** avec des méthodes identiques à celles de la classe **Point** : `__init__`, `__str__`, `__eq__` (copy – paste ...).
- (2) Créer sur une nouvelle fiche la classe **Line**.
  - a) Une droite est définie par deux points **p1** et **p2** et son nom **name**. On supposera que l'utilisateur entre toujours des points à coordonnées entières et on n'aura pas besoin de le tester dans le programme ! Les attributs **p1** et **p2** et le nom **name** sont les paramètres du constructeur `__init__`. Par ailleurs le constructeur calculera à l'aide des coordonnées de **p1** et **p2** trois attributs supplémentaires **a**, **b** et **c** : ce sont des entiers *premiers entre eux* de sorte que :  $a x + b y == c$  soit une équation cartésienne de la droite en question.
  - b) Programmer la méthode `direction_vector` qui retourne un vecteur directeur (aux coordonnées entières) de la droite en cours.
  - c) Programmer les méthodes `slope` et `y_intercept` qui retournent respectivement la pente et l'ordonnée à l'origine de la droite en cours *sous forme de nombres rationnels*, si elle n'est pas parallèle à (Oy).
  - d) Ajouter la méthode `__str__` qui retourne l'équation cartésienne de la droite en cours sous la forme  $a x + b y == c$ .
  - e) Ajouter la méthode `explicit_equation` qui retourne l'équation cartésienne réduite de la droite en cours. Pour les droites parallèles à (Oy), cette équation est de la forme :  $x == k$  avec  $k$  rationnel, pour les droites parallèles à (Ox),

elle est de la forme :  $y = k$  avec  $k$  rationnel, pour les autres droites l'équation réduite est de la forme  $y = p x + q$  avec  $p$  et  $q$  rationnels.

- f) Programmer la méthode `is_parallel_to(self, other)` qui retourne `True` si les deux droites sont parallèles, `False` sinon
- g) Programmer la méthode `intersection_point(self, other)` qui retourne le point d'intersection de deux droites sécantes *avec des coordonnées rationnelles*. Lorsque les droites sont confondues la méthode retournera l'équation cartésienne  $a x + b y = c$  commune des deux droites, lorsqu'elles sont strictement parallèles, la méthode retournera `'empty set'`.

- (3) Tester les méthodes de la classe `Point` dans une nouvelle fiche `main`. Par exemple le code :

```
from class_Droite import *
from class_Point_Vecteur import *

a=Point(4,-3)
b=Point(-2,1)
c=Point(0,3)
d=Point(5,4)
d1=Line(d1,a,b)
d2=Line(d2,c,d)

print('1re droite :')
print(d1)
print(d1.direction_vector())
print(d1.explicit_equation())
print()

print('2e droite :')
print(d2)
print(d2.direction_vector())
print(d2.explicit_equation())
print()

print("Point d'intersection :")
print(d1.intersection_point(d2))
```

produira la fenêtre d'exécution :

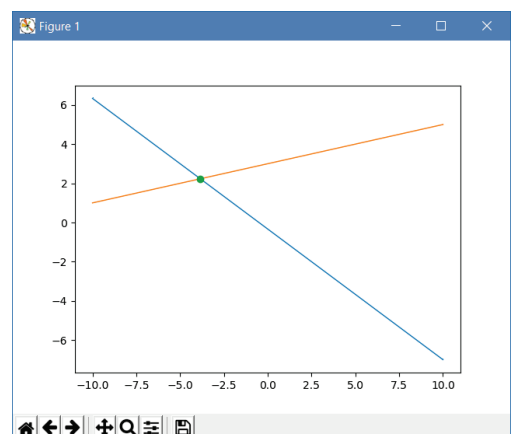
```
1re droite :
d1 : 2 x + 3 y == -1
(-3,2)
y == -2/3 x + -1/3

2e droite :
d2 : 1 x + -5 y == -15
(5,1)
y == 1/5 x + 3

Point d'intersection :
(-50/13,29/13)

Process finished with exit code 0
```

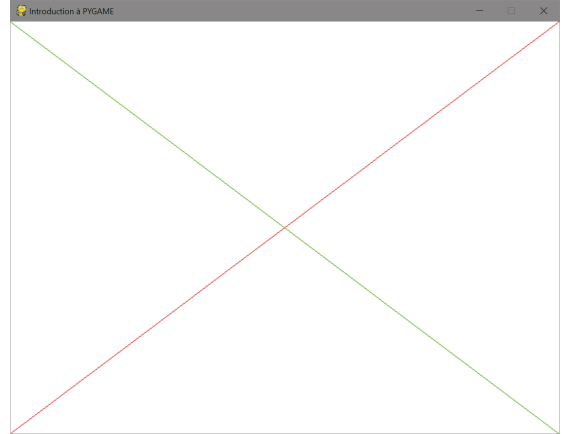
- (4) Représenter graphiquement à l'aide de `matplotlib` les droites `d1` et `d2` de la question précédente sur l'intervalle `[-10,10]` ainsi que leur point d'intersection. Le résultat se trouve ci-contre. Tester le programme avec d'autres droites ... (Ajouter sur la fiche `main` une fonction `representation(d,x1,x2)` qui produit le graphe d'une droite quelconque sur l'intervalle `[x1,x2]`.)



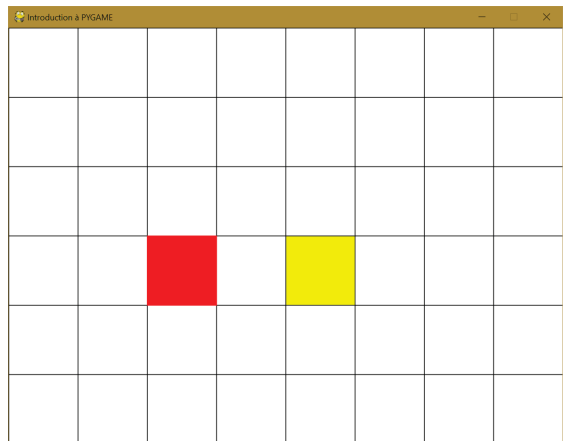
## Exercice 5 : Introduction à Pygame

Créer une application Pygame qui tourne dans une fenêtre avec un arrière-fond blanc de taille 800 x 600 pixels et ayant comme titre : « Introduction à PYGAME ». Au lancement de l'application, on doit voir seulement l'arrière-fond blanc. Dans la boucle principale :

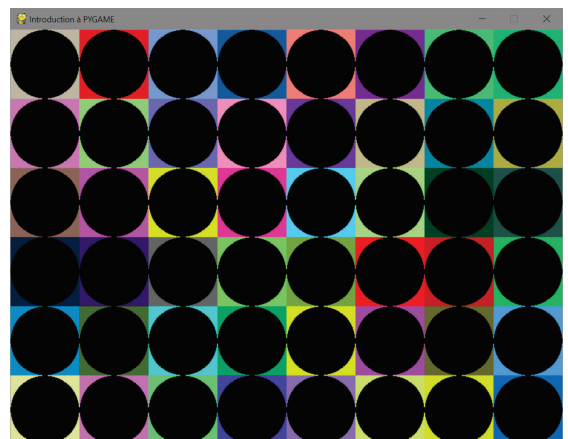
- (1) Si l'on enfonce la touche 0 du clavier, les deux diagonales de l'écran sont dessinées en vert et en rouge respectivement.



- (2) Si l'on enfonce la touche 1 du clavier, le programme dessine une grille sur l'écran, dont les lignes sont placées aux abscisses et aux ordonnées qui sont des multiples de 100. Deux carrés doivent être complètement remplis, l'un en rouge avec le bord, l'autre en jaune sans le bord, comme sur la figure ci-dessous. (Remarquez que les bords à droite et en bas de la fenêtre ne sont pas noirs. Cela est dû au fait que la fenêtre a bien 800 pixels en largeur et 600 pixels en hauteur : l'origine étant le pixel (0,0), l'abscisse varie de 0 à 799 et l'ordonnée varie de 0 à 599.)



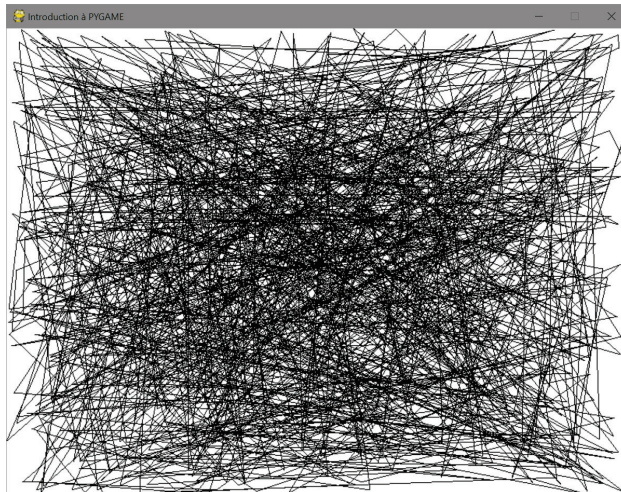
- (3) Si l'on enfonce la touche 2 du clavier, le programme remplit l'écran par des carrés 100 x 100 avec des couleurs aléatoires et place un disque noir sur chaque carré. (Utiliser `pygame.draw.rect` et `pygame.draw.ellipse`)



- (4) Si l'on enfonce la touche 3 du clavier, le programme dessine 1000 disques dont les centres se situent aléatoirement sur la surface de dessin et les rayons sont également choisis au hasard entre 20 et 50 pixels. Afin de voir avec quelle vitesse les disques sont dessinés, on actualisera l'écran après chaque disque. (Utiliser `pygame.draw.circle`)



- (5) Si l'on enfonce la touche 4 du clavier, le programme dessine une ligne polygonale noire reliant 1000 points aléatoires de la surface de dessin. On utilisera la fonction `pygame.draw.lines` (avec `s !`). Consultez la documentation officielle sur internet pour cette fonction utile.



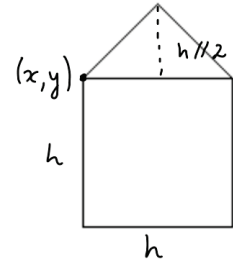
- (6) Si l'on clique sur la croix de fermeture, le programme doit se terminer correctement.

N.B. La surface de dessin doit être réinitialisée (donc effacée) à chaque fois qu'on enfonce une touche (de 1 à 6) du clavier.



## Exercice 6 : Houses

- (1) Programmer la classe **House** à 5 attributs **x**, **y**, **h**, **color** (valeur par défaut **Color('black')**) et **selected** (valeur par défaut **False**) qui représente une maison au sommet **(x,y)** et aux dimensions de la figure ci-contre. Outre le constructeur **\_\_init\_\_**, cette classe possède :



- a) une méthode **draw** qui permet de dessiner la maison à l'aide de **pygame** sur une surface de dessin **screen** (paramètre)
- b) une méthode **move**, qui permet de déplacer la maison horizontalement de **dx** pixels et verticalement de **dy** pixels.
- c) une méthode **toggle\_selection**, qui permet d'inverser la valeur de l'attribut **selected**.
- (2) Ecrire une fonction **draw\_all(h\_list,screen)** qui représente toutes les maisons se trouvant dans la liste **h\_list** sur l'écran **screen**.
- (3) Dans le programme principal, l'utilisateur est invité à entrer le nombre **n** de maisons. Le programme initialise et dessine alors **n** maisons ayant des attributs **x**, **y**, **h** et **color** aléatoires (de sorte que **(x,y)** appartient à la surface de dessin) et telles que seul l'attribut **selected** de la 1<sup>re</sup> maison soit initialisé à **True**, les attributs correspondants des autres maisons gardent leur valeur par défaut **False**. Les maisons seront placées dans une liste **house\_list**. La surface de dessin blanche a les dimensions 800 x 600 pixels.



- (4) Si l'on enfonce l'une des flèches de direction du clavier, toutes les maisons dont l'attribut **selected** est **True** se déplacent de 10 pixels dans la direction correspondante (et peuvent éventuellement sortir de l'image.)
- (5) Pour sélectionner ou désélectionner une maison, on clique avec le **bouton de droite** de la souris dans le carré inférieur de la maison. On écrira une fonction **get\_house(house\_list,x,y)** qui devra retourner l'indice de la 1<sup>re</sup> maison dans la liste **house\_list**, dont le carré inférieur contient le point **(x,y)**. Si aucun carré ne contient le point **(x,y)**, la fonction devra retourner **-1**.



- (6) A l'aide du bouton gauche de la souris on peut aussi déplacer une maison en cliquant à l'intérieur de son carré et en glissant avec la souris sur l'image en maintenant enfoncé le bouton.

### **Exercice 7 : Robots – Représentation graphique**

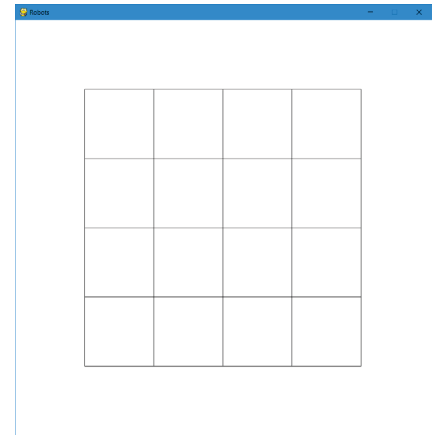
On reprend l'exercice 1. Le but est de représenter graphiquement la marche aléatoire d'un nombre quelconque de robots.

- (1) Ajouter à la classe Robot trois attributs :
  - a) **color** (valeur par défaut **None**),
  - b) **alive** (valeur par défaut **'True'**)
  - c) **size** (valeur par défaut **2**)
- (2) Modifier les méthodes **move**, **set\_random\_position**, **random\_move** et **turn** de sorte qu'un robot peut seulement bouger ou tourner ou changer de position si son attribut alive est **'True'**. (La méthode **hits** ne doit pas être modifiée).
- (3) Dans le fichier **representation**, on importera la bibliothèque **pygame** et on mettra toutes les instructions permettant de représenter graphiquement la marche aléatoire des robots.
  - a) Demander à l'utilisateur le nombre de robots ( $\leq 26$ ) à représenter et changer éventuellement les attributs de classe **Robot.xmax** et **Robot.ymax**. Instancier les robots et les placer dans une liste nommée **robots**. Chaque robot aura une position initiale et une couleur choisies au hasard. Le robot d'indice **i** dans la liste aura comme nom la lettre d'indice **i** dans l'alphabet et sa taille est fixée à **2\*(i+1)**. (N.B. : Le nom **'A'** est réservée pour le robot n° 0, etc.).
  - b) Ecrire la fonction **draw\_grid()** qui efface l'écran, puis trace la grille sur laquelle les robots évoluent. On déterminera les dimensions en pixels **dx** resp. **dy** d'une cellule de la grille de sorte qu'il reste de la place pour les 4 bords blancs de **dx** pixels (à gauche et à droite de la grille) respectivement de **dy** pixels (en haut et en bas de la grille).

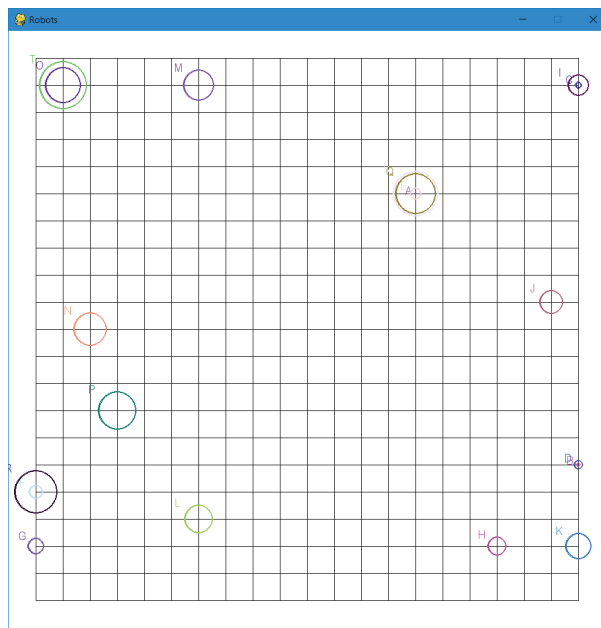
Par exemple sur la figure ci-contre, l'utilisateur a choisi :

**Robot.xmax = Robot.ymax = 2**

Le coin supérieur gauche de la grille a donc les coordonnées  $(-2,2)$  et le coin inférieur droit  $(2,-2)$ .



- c) Ecrire la fonction **draw\_robot(r)** qui représente le robot **r** par un cercle de centre la position du robot (ses coordonnées), de rayon sa taille et de couleur sa couleur (épaisseur = 2 pixels). Le nom du robot sera ajouté à côté du cercle qui le représente.



- d) Dans la boucle principale, on commencera toujours par représenter la grille vierge, puis on testera si les robots créés sont éventuellement déjà entrés en collision. Deux robots « nés » au même endroit ou entrés en collision meurent, c.-à-d. leur attribut **alive** prendra la valeur '**False**', de sorte qu'ils ne bougeront plus après. Les robots survivants font un pas aléatoire dans une direction aléatoire. Tous les robots (même ceux qui sont morts) seront représentés. Le programme tourne à 20 FPS (ou une autre vitesse, éventuellement au choix de l'utilisateur).

## Exercice 8 : Polynômes et représentation graphique

### PARTIE A

Sur une fiche `class_polynomial`, programmer la classe `Polynomial` qui représente un polynôme à coefficients réels, initialisé par la donnée de la liste de ses coefficients suivant les puissances croissantes de la variable. Par exemple, l'appel

```
p = Polynomial(-2.5, 3, 0, 7)
```

permettra de créer le polynôme  $p(x) = -2.5 + 3x + 7x^3$ .

- (1) La classe `Polynomial` a deux attributs : `coefficients` et `degree`. Le constructeur reçoit le paramètre `*coefficients`. C'est la liste « déballée » (`unpacked`) des coefficients. Cette technique permet d'appeler le constructeur sans devoir placer artificiellement les coefficients dans une liste (voir exemple ci-dessus). Le constructeur détermine le degré du polynôme (`degree`). Attention au cas où l'utilisateur mettrait des 0 superflus à la fin lors de l'appel :

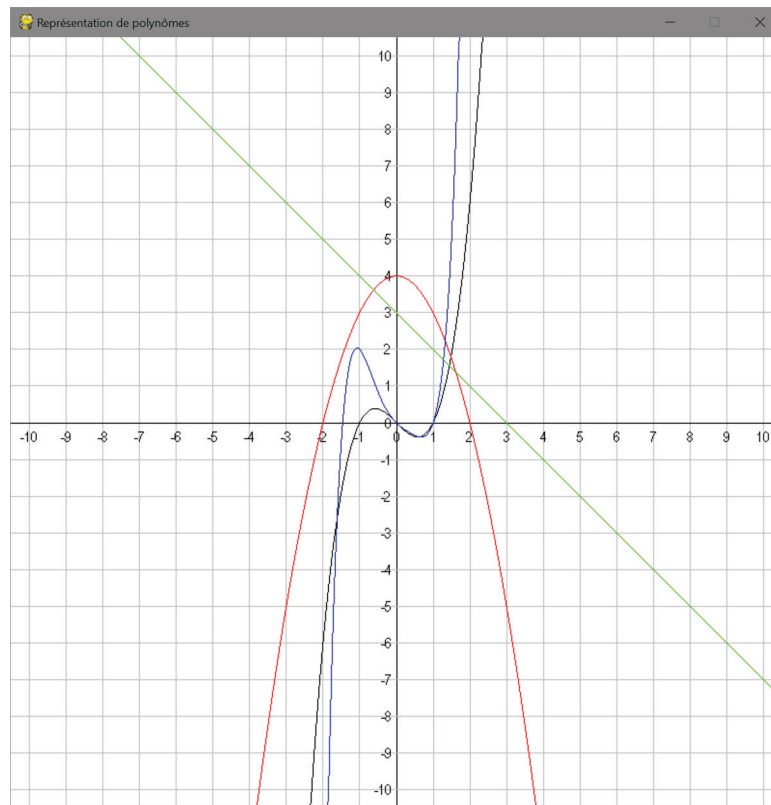
```
p = Polynomial(-2.5, 3, 0, 7, 0, 0)
```

doit toujours retourner le polynôme de degré 3 ci-dessus.

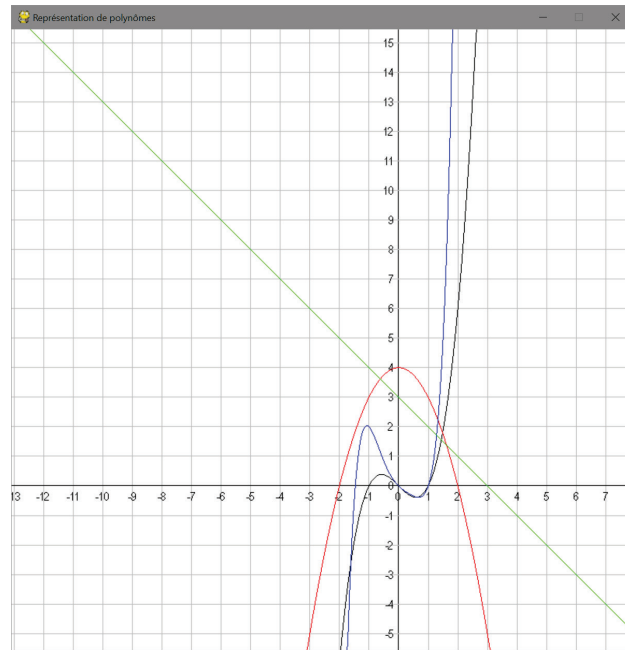
- (2) Ecrire la méthode `__str__` qui convertit le polynôme sous forme d'une chaîne de caractères. Donc `print(p)` écrira sur l'écran : `7x^3 + 0x^2 + 3x^1 + -2.5x^0` dans le cas du polynôme ci-dessus.
- (3) Ecrire la méthode magique `__call__`, qui devra retourner la valeur numérique du polynôme en un réel donné `x` (paramètre). Cette méthode permet d'appeler un polynôme comme une fonction en mathématiques : par exemple `print(p(1))` affichera `7.5` sur l'écran dans le cas du polynôme ci-dessus.
- (4) Ecrire les méthodes `__add__`, `__sub__` et `__mul__` qui permettent respectivement d'additionner, de soustraire et de multiplier deux polynômes. Pour la soustraction, on pourra d'abord écrire la méthode `__neg__`, qui retournera l'opposé d'un polynôme.
- (5) Ecrire la méthode `derivative` qui retourne la dérivée d'un polynôme.
- (6) Tester les méthodes de la classe sur des exemples simples.

### PARTIE B

Sur une nouvelle fiche `main_polynomial`, écrire un programme `pygame` qui permet de représenter graphiquement une liste de polynômes, avec différentes couleurs. Voici un exemple d'exécution :



- (1) Initialiser l'écran carré de taille **SIZE = 800**.
- (2) La fiche doit contenir la classe **Representation** aux attributs **xmin, xmax, ymin, ymax, dx** et **dy**. **[xmin,xmax]** et **[ymin,ymax]** sont les intervalles représentés respectivement sur l'axe des abscisses et l'axe des ordonnées et peuvent être initialisés par le constructeur de la classe. Le constructeur calculera à l'aide des formules du cours les valeurs des attributs **dx** et **dy** qui sont l'accroissement réel de **x** et de **y** lorsqu'on se déplace de 1 pixel vers la droite respectivement de 1 pixel vers le bas sur l'image.
- (3) Ecrire les méthodes suivantes de la classe **Representation** :
  - a. **pt2px(self, x, y)** qui retourne les coordonnées pixel d'un point **(x,y)** du plan réel.
  - b. **px2pt(self, px, py)** qui retourne les coordonnées réelles d'un pixel **(px,py)** de l'image.
  - c. **draw\_grid(self)** qui représente le repère (axes en noir avec abscisses et ordonnées entières, droites verticales  $x = k$  avec **k** entier et **xmin**  $\leq$  **k**  $\leq$  **xmax** et horizontales  $y = l$  avec **l** entier **ymin**  $\leq$  **l**  $\leq$  **ymax** en gris).
  - d. **move(self, dpdx, dpy)** qui permet de déplacer le repère horizontalement de **dpdx** pixels et verticalement de **dpy** pixels. *Indication* : redéfinir les attributs **xmin, xmax, ymin** et **ymax**.



- e. **Difficile** : `stretch_horizontal(self, x, px)` qui permet d'étirer (ou de contracter) l'axe des abscisses, comme dans Geogebra. Le paramètre **x** représente l'abscisse *réelle* du pixel où l'utilisateur enfonce le bouton de la souris. **px** est l'abscisse du *pixel* où il relâche la souris. Il faudra recalculer **xmax** et **xmin** (à l'aide de règles de trois) de sorte que l'abscisse réelle après le changement d'échelle (rescaling) soit encore **x**. N'oubliez pas de recalculer l'attribut **dx**. **Indication** : l'origine du repère ne doit pas bouger lors du rescaling.



- f. `stretch_vertical(self, y, py)` qui permet d'étirer (ou de contracter) l'axe des ordonnées.

g. `draw_polynomial(self, poly, color=Color('black'))` : qui permet de représenter graphiquement un polynôme donné dans le repère.

(4) Ecrire les fonctions suivantes du programme principal :

a. `get_polynomial()` qui permet à l'utilisateur d'entrer le degré et les coefficients d'un polynôme et qui retourne ce polynôme.

b. `display_polynomials(polynomials)` qui affiche, à l'aide de la méthode `__str__` les polynômes de la liste `polynomials`. On obtient l'affichage suivant dans le cas de l'exemple d'exécution :

`p0 : 1x^3 + 0x^2 + -1x^1 + 0x^0`

`p1 : -1x^2 + 0x^1 + 4x^0`

`p2 : 1x^5 + 0x^4 + -1x^3 + 1x^2 + -1x^1 + 0x^0`

`p3 : -1x^1 + 3x^0`

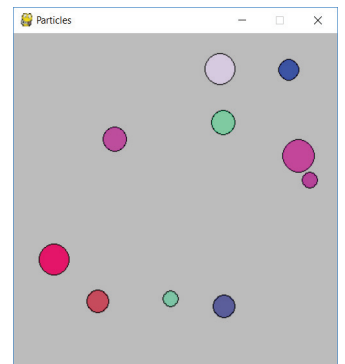
c. `draw_polynomials(polynomials)` qui représente tous les polynômes de la liste `polynomials`.

(5) Ecrire la boucle principale : le programme réagit aux touches **n** (pour entrer un nouveau polynôme, qui sera ajouté à la liste des polynômes à représenter) et **d** pour effacer un polynôme. A l'aide de la souris on peut translater le repère ou de dilater / contracter les échelles des axes, comme dans Geogebra ...

## Exercice 9 : Particules en mouvement

Définissez la classe `Particle`, qui possède comme attributs :

- les coordonnées `x` et `y` du centre de la particule ;
- le rayon (`radius`) de la particule ;
- la couleur (`color`) de la particule ;
- les composantes horizontale et verticale de la vitesse de la particule (`x_speed` et `y_speed`). Un mouvement vers la droite/vers le bas est représenté par une vitesse positive, un mouvement vers la gauche/vers le haut par une vitesse négative ;
- l'épaisseur (`thickness`) du bord de la particule, fixée à 1, c'est un attribut de classe ;
- le coefficient d'élasticité (`elasticity`) de la particule, fixé à 0.95, c'est un attribut de classe.



La classe possède un constructeur qui permet de donner des valeurs aux attributs d'instance.

La classe possède également les méthodes suivantes :

- `draw` dessine la particule aux coordonnées (`x`, `y`) en dessinant d'abord un disque de couleur `color` et de rayon `radius` suivi d'une couronne de cercle de couleur noire, de rayon `radius` et d'épaisseur `thickness` ;
  - `move` vérifie d'abord si un mouvement dans la direction actuelle dépasse un des 4 bords de l'écran. Dans ce cas, la vitesse dans la direction concernée est inversée et multipliée par le coefficient d'élasticité. Ensuite, on ajoute à chacune des coordonnées la vitesse correspondante ;
  - `distance_to` à un paramètre supplémentaire (correspondant à une deuxième particule) calcule la distance entre les centres des 2 particules ;
  - `manage_collisions_with` à un paramètre supplémentaire `other` (correspondant à une deuxième particule) vérifie si la particule actuelle et la particule passée en paramètre sont entrées en collision (il y a collision si la distance entre leurs centres est inférieure à la somme de leurs rayons). Dans ce cas, les deux composantes de la vitesse (horizontale et verticale) des deux particules sont échangées, en les multipliant par le coefficient d'élasticité de la particule actuelle. Afin d'éviter des situations où des particules sont « collées » l'une contre l'autre, les deux particules sont ensuite déplacées jusqu'à ce qu'elles ne soient plus en situation de collision.
1. La fonction `reset` à un paramètre entier `number` retourne une liste contenant `number` particules.
    - Le rayon des particules est un nombre entier aléatoire choisi dans l'intervalle  $[0,20]$ .
    - La position (attributs `x` et `y`) des particules est choisie de manière aléatoire, mais de sorte qu'elles se trouvent entièrement à l'intérieur de l'écran.
    - Les composantes de la vitesse des particules sont des nombres réels aléatoires dans l'intervalle  $[-5,5]$ .
    - La couleur des particules est choisie de manière aléatoire.
  2. Programmez maintenant la boucle principale selon les instructions suivantes :
    - avant d'entrer dans la boucle principale, la liste contenant les particules est remplie avec 10 particules aléatoires en appelant la fonction `reset` ;
    - le programme tourne dans une fenêtre avec un arrière-fond gris de taille 400x400 pixels ;



- le titre de la fenêtre est «Particles » ;
- le programme réagit à la touche « n » du clavier : la liste contenant les particules est remplacée par une liste avec 10 nouvelles particules ;
- une boucle parcourt la liste de particules et :
  - déplace chacune des particules ;
  - affiche chacune des particules à l'écran ;
  - pour chacune des particules du reste de la liste, une éventuelle collision des 2 particules en question est gérée.
- le programme tourne à 25 FPS ;
- le programme se termine proprement lorsqu'on clique sur la croix de fermeture.

### Exercice 10 : Tours de Hanoi

Le problème des tours de Hanoi a été expliqué dans l'exercice 15 sur la récursivité. Le but de cet exercice est d'écrire un programme avec une interface graphique conviviale qui permet à l'utilisateur de résoudre le problème soit manuellement (en déplaçant les disques à l'aide de la souris), soit automatiquement (à l'aide de la fonction récursive de l'exercice 15, légèrement adaptée).

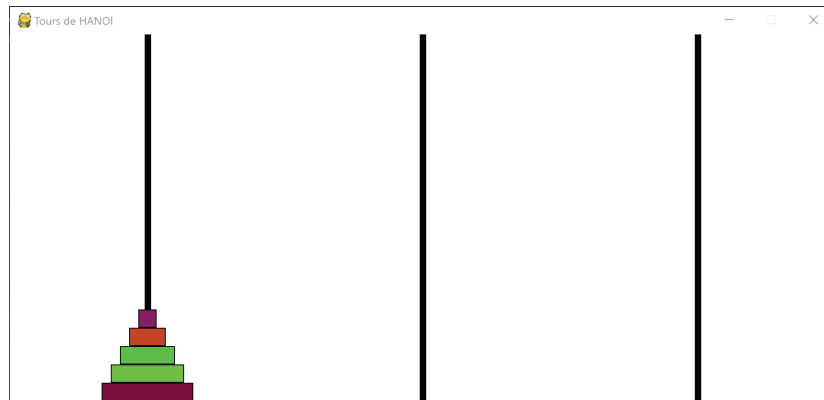


fig. 1 : Configuration initiale avec 5 disques

- (1) On utilisera la classe **Disk** suivante et très simple pour modéliser un disque :

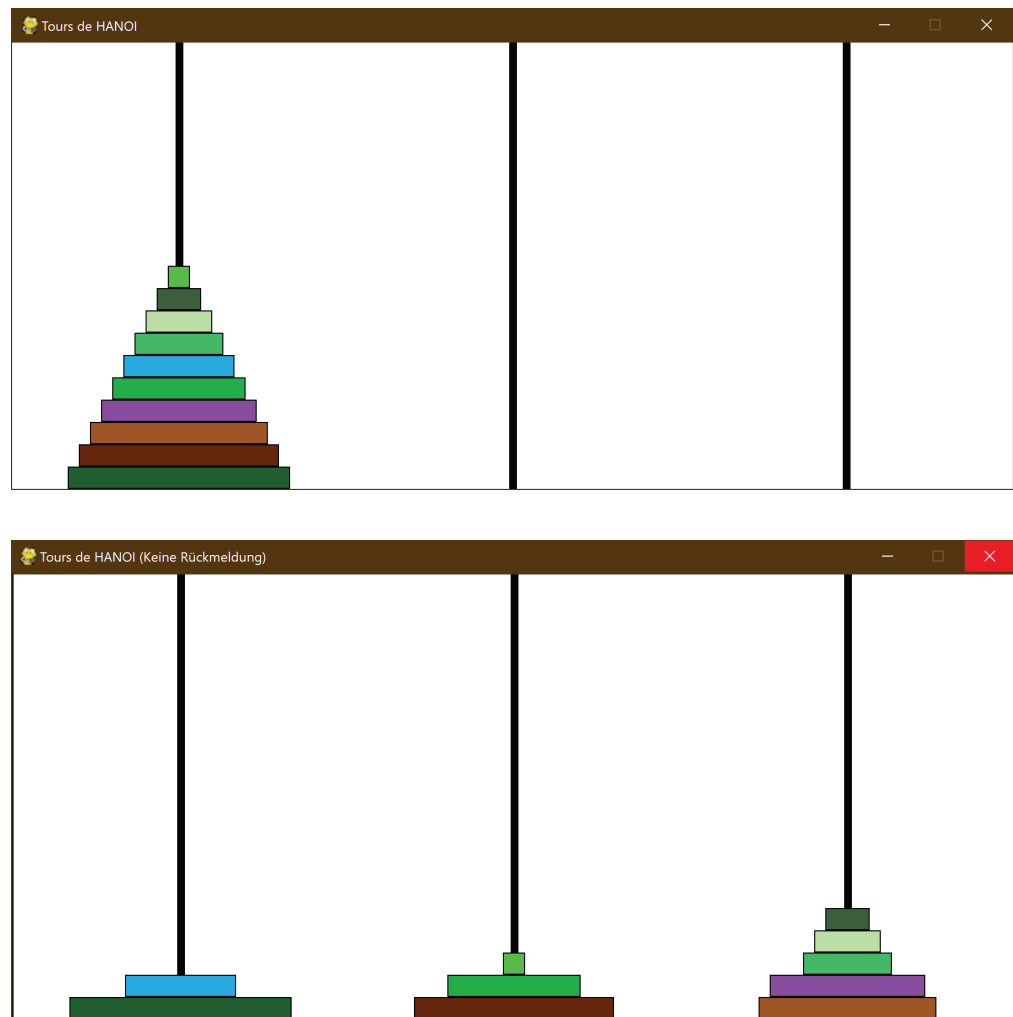
```
class Disk:
    def __init__(self, size, color):
        self.size = size
        self.color = color
```

L'attribut **size** permet de contrôler la largeur du rectangle représentant un disque sur le graphique. Si initialement il y a  $n$  disques ( $n = 5$  sur la fig.1), **size** varie de 1 (pour le plus petit) à  $n$  pour le plus grand. La largeur du rectangle

représentant le disque de taille **size** sera égale à **20\*size**. La couleur de chaque disque sera choisie au hasard.

- (2) Ecrire la classe **Tower** qui représente une tour de disques empilés. Cette classe a deux attributs : **n\_disks** (le nombre de disques de la tour) et **disks** (une liste avec des objets de type **Disk**). Le constructeur a comme seul paramètre **n\_disks**. Il initialise l'attribut **n\_disks** sur la valeur de ce paramètre (valeur par défaut 0). L'attribut **disks** est initialisé de sorte que la liste contient **n\_disks** disques dont les tailles varient de **n\_disks** à 1 (donc le disque le plus large est le premier élément de la liste) et dont les couleurs sont choisies aléatoirement. En outre la classe a trois méthodes : a) **add\_disk** permet d'ajouter un disque en fin de liste, b) **pop\_disk** enlève et retourne le dernier disque de la liste, c) **move\_disk(self, other)** permet de déplacer un disque de la tour courante vers une autre tour.
- (3) Ecrire la fonction **draw\_tower(t, i)** qui dessine sur le graphique la tour **t** (de type **tower**) et d'indice **i**. Comme il y a trois tours sur la figure, l'indice **i** pourra prendre comme valeurs 0 (tour à gauche), 1 (tour au milieu) ou 2 (tour à droite). Chaque tour est formée d'un piquet (segment vertical noir de largeur 7 pixels), placé au milieu du tiers correspondant de la figure et des disques qui la composent. La hauteur de chaque disque est de 20 pixels, sa largeur est déterminé par son attribut **size** comme expliqué dans (1). Chaque disque a un bord noir.
- (4) Initialiser le jeu avec 5 disques. On stockera les 3 tours dans une liste **towers** de sorte que la tour d'indice 0 (à gauche sur la figure) contienne tous les disques dans le bon ordre et les deux autres tours soient vides. On choisira comme dimensions de la fenêtre d'application **WIDTH = 900** et **HEIGHT = 400**. On prendra **FPS = 1** au début, pour bien pouvoir suivre les déplacements des disques lors du mode automatique. On pourra ensuite après quelques exécutions, augmenter **FPS** pour accélérer l'exécution, surtout lorsque le nombre de disques est élevé. (Il faut savoir que le problème se résout en  $2^n - 1$  étapes, dans le cas de  $n$  disques.)
- (5) Ecrire la fonction **draw\_game()** qui redessine les 3 tours, sur fond blanc.
- (6) Ecrire la boucle principale. Il doit être possible de déplacer un disque manuellement avec la souris d'un piquet vers un autre. Lorsque l'utilisateur tape sur la touche **n** du clavier, il peut choisir le nombre de disques dans la console et le jeu est réinitialisé. Lorsqu'il tape sur la touche **s** du clavier, le jeu est résolu automatiquement (en supposant qu'on part de la configuration initiale !). Pour

cela il faut écrire une version légèrement adaptée de la fonction récursive `solve_hanoi(n, start, dest)` vue dans l'exercice 15 des fonctions récursives.



*fig. 2 Configuration initiale avec 10 disques et  
une étape pendant la solution automatique*

**Bonus :** Afficher sur le graphique le numéro de l'étape lors de la solution automatique.