Squash

0. Préparation

L'objectif de ce td est de vous faire travailler avec les classes et les objets d'une part et d'autre part de vous introduire à la notion de tests.

Vous trouverez deux fichiers sur moodle pour ce tp :
- squash.py contient le squelette du programme à compléter
- squash_tests.py contient les tests qui vous permettront de vérifier vos classes une à une au fur et à mesure que vous avancerez

Il y a dans ce tp un classe un peu particulière que vous ne devriez pas avoir à retoucher à priori : *MoveCallable*. Si vous vous souvenez pour simuler une boucle d'animation avec tkinter, on utilise la fonction *window_tk.after(millisecondes, animation_function)*. La classe *MoveCallable* implémente une méthode particulière : *__call__(self)*, ce qui en fait une classe dite *Callable* ; cela signifie qu'on peut utiliser les objets de cette classe comme des fonctions. Par exemple *mon_callable()* exécutera la fonction *__call__* de l'objet *mon_callable*. Dans ce tp, la classe *MoveCallable* permet de ne pas avoir à utiliser le mot clef global puisqu'on peut lui donner définir des attributs qu'elle peut réutiliser par la suite. TL/DR : **Vous n'aurez pas besoin de toucher à cette classe dans ce TP.**

Le programme que l'on se propose d'écrire est un genre de pong monojoueur. Vous aurez trois classes à coder :
- WindowInfos – une simple structure sans méthode en dehors du constructeur qui doit contenir trois infos : la fenêtre de TK, la largeur de la fenêtre et la hauteur de la fenêtre
- Ball – qui décrit le comportement de la balle
- Racket – qui décrit le comportement de la rackette

Jetez un œil dans le code partout où vous trouverez un TODO. Vous y trouverez des consignes en commentaires.

1. Classe Ball

Jetez un œil aux commentaires et aux TODO dans squash.py, tout y est.

Pensez à tester votre classe en faisant un python squash_tests

2. Classe Racket

Jetez un œil aux commentaires et aux TODO dans squash.py, tout y est.
Pensez à tester votre classe en faisant un python squash_tests

3. Le programme fonctionne ! Et maintenant ?

Normalement, une fois Ball et Racket complétées, votre programme fonctionnera. Il donc temps d'y apporter quelques modifications, de le peaufiner !

/!\ Note : vous devrez modifier les tests au fur et à mesure **avant** de modifier le programme afin de i. vous forcer à définir en amont aux objectifs que vous vous fixer et donc au problème, ii. De pouvoir vérifier que votre code répond bien aux objectifs que vous vous êtes fixé.

Je vois au moins deux premières améliorations qui pourraient être apportées.

3.1. Permettre à l'utilisateur de « modifier la direction de la balle »

Pour l'instant la balle ne fait que rebondir sur la raquette (axe des x - dx)) et continue sur sa lancée sans modifier sa trajectoire en y. On voudrait donner un effet à la balle avec la raquette. Faites en sorte que l'endroit où rebondit la balle sur la raquette détermine sa direction sur l'axe des y (dy) :

- Si la balle rebondit sur la moitié haute de la raquette, elle doit repartir vers le haut
- Si la balle rebondit sur la moitié basse de la raquette, elle doit repartir vers le bas
- (pour les perfectionnistes) Cette effet sera plus ou moins important en fonction de la distance de l'impact par rapport au centre de la raquette.

3.2. Forcer une vitesse constante pour la balle

Si vous êtes attentifs, vous vous apercevrez que la vitesse de la balle « varie » en fonction de dx et dy. C'est parce que la longueur du vecteur (dx, dy) n'est pas constante.

Nous allons changer le mode de représentation de la vitesse afin de garantir une vitesse constante à la balle. Pour cela, au lieu de représenter le mouvement de la balle avec deux pas de déplacement en x et y (dx et dy), nous allons le représenter par i. un vecteur direction normalisé (*direction*) et ii. Une vitesse (*speed*) constante.

i. Le plus compliqué c'est le vecteur *direction* car sa longueur doit être égale à 1. Ainsi à chaque modification, vous devrez vous assurer cette propriété est vérifiée. Je vous propose de procéder ainsi :
- *direction* est une liste contenant deux valeur comprises entre 0. et 1. [dx, dy]
- rendez les attributs *direction* et *speed* privés afin de s'assurer qu'ils ne puissent pas être modifiés en dehors de la classe *Ball*. (rappel de cours, il suffit de renommer l'attribut avec __ devant ce qui donnera => *__speed* et *__direction*). Vous pourrez toujours les modifier au sein de la classe avec *self.__speed* et *self.__direction*.
- Créez une méthode privée *__adjust_dx2dy(self)* ; le but de cette méthode est d'ajuster __direction[0] (dx) en fonction de __direction[1] (dy) pour que __direction[0] + __direction[1] = 1. Pensez à Pythagore, dessinez votre vecteur sur un bout de papier et vous aurez la solution.
- Enfin, ppelez cette méthode *__adjust_dx2dy()* à la fin de la méthode *update_direction(self, window_infos, racket).*

ii. La vitesse *speed* a juste besoin d'être initialisée dans le constructeur.

Il vous reste à modifier la formule pour mettre à jour la position de la balle, ce ne sera plus :
self.x = self.x + self.dx
self.y = self.y + self.dy
À vous de trouver la solution.

4. (BONUS) Transformez ce Squash en casse-briques !

Voici quelques ajouts que vous pourriez faire pour transformer ce programme en casse-briques :

/!\ Pensez à préparer vos tests !

=> Créez une classe Brique
- Trouvez une façon de stocker les briques (qui permette aussi leur suppression)
- Trouvez une façon de générer toutes les briques du niveau.
- Trouvez un moyen de gérer la collision entre les briques et la balle
- Quand toutes les briques sont détruites, le jeu s'arrête et un message de félicitation est affiché.

=> Créez une classe Bonus
- À la destruction de certaines briques un bonus sera créé.
- La position des bonus évolue au cours du temps, ils « tombent » vers la gauche
- Lorsqu'un bonus entre en collision avec la raquette, il peut y avoir différentes effets (balle ralentie, raquette agrandie, etc)