Atelier Projet 1ère partie

- Base et propriétés -

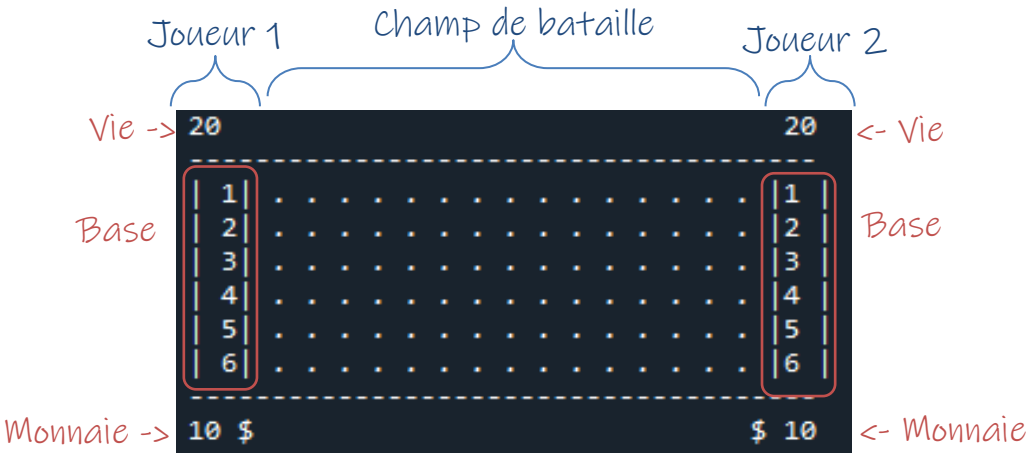
A. Introduction

L'évaluation de l'UE programmation se fera via un **travail individuel** à **réaliser** et à **défendre** oralement.

Le travail consiste en la réalisation d'un petit jeu de A à Z qui vous a été présenté le 18 février (voir slides de présentation).

Il s'agit d'un jeu de plateau qui voit 2 joueurs s'affronter pour défendre leur base située de part et d'autre d'un champ de bataille. Le but du jeu est de détruire la base adverse en premier. Pour ce faire, un joueur pourra envoyer des personnages attaquer l'adversaire. Les personnages devront alors traverser le champ de bataille sans se faire tuer avant de commencer à attaquer la base qui se trouve devant eux.

L'interface graphique du jeu sera en console. Voici un exemple :



Le jeu sera développé en 3 étapes :

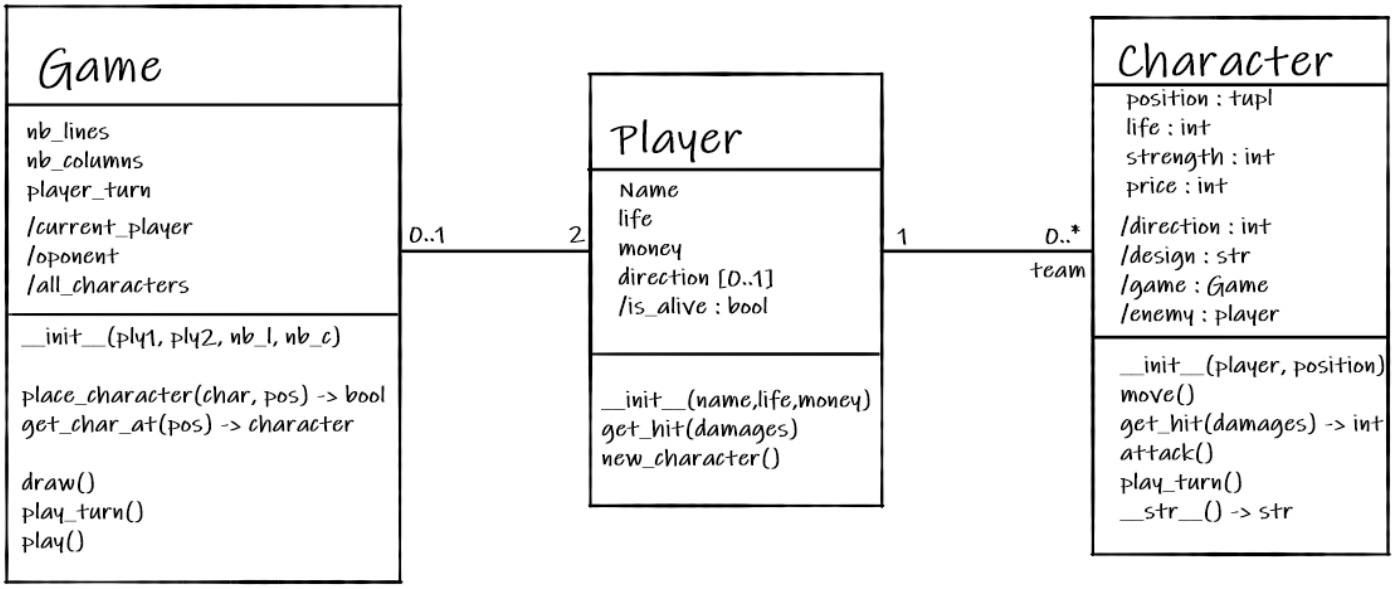
1. Jeu de base
2. Diversification des personnages
3. Amélioration du jeu

Cet atelier (et le prochain labo) concerne(nt) **la première partie, à savoir, faire le jeu de base**. Après cet atelier, vous aurez, si vous suivez bien les étapes, une base solide pour votre projet. Cette base (complète et correcte) vous amènera déjà **4** points sur 20.
Au cours de cet atelier, vous aurez aussi l'occasion de vous entrainer à écrire des propriétés (pour les attributs dérivables).

Bon travail !

B. UML

Voici le diagramme de classe UML correspondant au jeu de base :



Une partie (`Game`) est caractérisée pas le nombre de lignes et de colonnes qui représentent le plateau (6 lignes et 15 colonnes pour l'exemple en page 1) ainsi que le joueur à qui c'est le tour de jouer (0 ou 1). Une partie possède 2 joueurs. On voudra pouvoir connaitre le joueur actuel, l'opposant et la liste de tous les personnages présents sur le plateau.

Un joueur (`Player`) peut jouer à un jeu. Il a un nom, de la vie, de l'argent (pour pouvoir acheter des personnages) et une direction. La direction représente le sens d'attaque : +1 s'il attaque vers la droite et -1 s'il attaque vers la gauche. Dans le cas où le joueur ne joue à aucune partie, la direction est inconnue.

Un personnage (`Character`) possède de la vie, de la force, un prix et une position sur le plateau. La position est un tuple (ligne, colonne). Chaque personnage a une direction (qui est celle du joueur auquel il appartient), un design (le symbole qui le représente sur le plateau) et une partie sur laquelle il est en train de combattre. Il connait aussi son ennemi qui est le joueur adverse.

Avant de pouvoir traduire ce schéma en python, transformez-le en retirant les relations. Prenez le temps de bien le faire pour ne pas avoir de souci par la suite.

C. Python

Nous allons maintenant nous lancer dans l'implémentation. Allons-y progressivement pour ne rien oublier.

Pour commencer, **téléchargez le fichier « squelette.py » qui se trouve sur Moodle** dans la section « Projet ».

Il s'agit du squelette du projet. Tous les en-têtes des méthodes y sont écrits et, pour chaque méthode, il y a les spécifications complètes. Partout où il y a un `#TODO`, il va falloir compléter le code !

Vous pouvez également voir que certaines méthodes ont déjà été écrites (tout ou en partie) pour vous aider.

Player

Commençons par la classe `Player`.

Quels sont les attributs de `Player` ?

On retrouvera les attributs se trouvant dans `Player` sur le diagramme UML ainsi que ceux issus des relations :

- Une partie (`game`) : de type `Game` et qui est facultative
- Une équipe (`team`) : qui est une liste de `Character`

Constructeur

Complétez le constructeur de `Player` en suivant les spécifications écrites dans le squelette : `name`, `life` et `money` sont passés en paramètre, `team` est initialisé comme une liste vide alors que `game` et `direction` sont à `None` pour l'instant.

Attribut dérivable

Occupons-nous de l'attribut dérivable `is_alive`. En effet, il n'est pas nécessaire de stocker directement cet attribut car on peut le calculer à partir d'autres attributs.

Écrivons `is_alive` ensemble :

> **Rappel :**
>
> Une propriété en Python est une **méthode d'instance** décorée avec `@property`.
> Le **nom de la méthode** correspond au nom de la propriété (=attribut dérivable).
> La méthode doit **retourner la valeur** de la propriété (=attribut dérivable).

La méthode à écrire est une méthode d'instance donc elle prend `self` comme premier paramètre… et n'en prendra pas d'autre, donc `def is_alive(self):`. Mettons `@property` juste avant et il ne reste plus qu'à écrire l'intérieur. La méthode doit retourner un booléen indiquant si le joueur est en vie ou non.

Voici ce que ça donne :

```
@property
def is_alive(self):
    return self.life > 0
```

Méthodes d'instance

Écrivez le contenu de la fonction `get_hit` en suivant les spécifications données.

Pensez à tester de temps en temps ce que vous faites. Par exemple en créant un joueur, en affichant ses différents attributs et en testant ses méthodes. N'hésitez pas à réécrire `__str__` pour vous permettre de « printer » un joueur et voir plus rapidement ses attributs.

La méthode `new_character` a été écrite pour vous ! Jetez-y tout de même un œil.

La classe `Player` est finie ! 😊

Game

Ecrivez le **constructeur** de la classe `Game`. Le constructeur prend en paramètre deux joueurs ainsi que le nombre de lignes (6 par défaut) et colonnes (15 par défaut). Initialisez les différents attributs :

- `nb_lines` et `nb_columns` en fonction des paramètres reçus ;
- `players` (attribut issu de la relation) comme une liste de 2 joueurs ;
- `player_turn` à 0.

Dans le constructeur de `Game`, il faut également que vous mettiez à jour deux infos dans chacun des joueurs :

- La partie (qui est celle que vous êtes en train de créer) ;
- La direction (le joueur 0 ira vers la droite (donc +1) et le joueur 1 vers la gauche (donc -1)).

N'oubliez pas que `self` représente l'objet courant, c'est-à-dire la partie que vous êtes en train de créer. C'est donc elle qu'il faut assigner comme `game` du joueur.

Réfléchissez à comment connaitre les différents **attributs dérivables** à partir de ce qu'on connait déjà.

Pour `current_player`, il s'agit simplement du joueur qui se trouve à l'indice correspondant à `player_turn` dans la liste de joueurs (`players`). Pour `oponent`, ce sera l'autre joueur.

Pour `all_characters`, il s'agit de reprendre toute l'équipe (`team`) du joueur 0 et toute l'équipe (`team`) du joueur 1.

Écrivez `current_player`, `oponent` et `all_characters`.

Écrivez la méthode `get_character_at` qui prend une position (sous forme de tuple) en paramètre et retourne le personnage se trouvant à cette position (None si aucun ne s'y trouve). Pour ce faire, il faut parcourir tous les personnages de la partie et voir s'il y en a un dont la position correspond à la position passée en paramètre.

« Tous les personnages »… on a justement un attribut dérivable pour ça ! On pourra donc l'utiliser (comme n'importe quel attribut) : `self.all_characters`.

Vous n'avez pas encore implémenté `Character`… ce n'est pas un souci. Vous savez que les personnages ont un attribut position qui est un tuple.

Écrivez la méthode `place_character` qui prend un personnage et une position en paramètre et s'occupe de placer le personnage à cette position si c'est possible. Pour cela, regardez d'abord si la position se trouve bien dans le plateau et, si oui, vérifiez qu'il n'y a pas déjà quelqu'un à cette position. Si tout va bien, changez la position du personnage et retournez `True` pour indiquer que le changement a bien été fait. N'oubliez pas de retourner `False` si rien n'est fait.

« il n'y a pas déjà quelqu'un à cette position »… On vient justement d'écrire une méthode pour ça… Donc utilisons-la !

Nous allons passer un peu au personnages… Nous reviendrons à la partie après.

Character

Avant d'implémenter `Character`, prenons 2 minutes pour parler des attributs de classe `base_price`, `base_life` et `base_strength`, vous les avez vu dans le code ? Pourquoi en faire des attributs de classe ? Parce que tous les personnages ont, à la base, le même prix, la même force et la même vie !

On garde des attributs d'instance `price`, `life` et `strength` parce que, une fois créé, un personnage peut voir ces stats changer (gagner de la force ou de la valeur ou perdre de la vie).

Un autre avantage de ces attributs de classe, c'est qu'on n'a pas besoin de créer une instance pour connaitre leur valeur puisqu'on peut y accéder en faisant `Character.base_price` par exemple.

 Le constructeur a été implémenté pour vous. Lisez la spécification et le code pour voir si vous comprenez.

 Implémentez les attributs dérivables de la classe `Character` en suivant le squelette donné.

La **direction** d'un personnage correspond à celle du joueur à qui il appartient.

La **partie** est celle à laquelle joue le joueur.

L'ennemi est l'autre joueur de la partie.
Une manière de faire est de prendre le joueur 0 si le perso attaque à gauche et le joueur 1 s'il attaque à droite.

Le **design** est la manière dont sera représenté le personnage sur le plateau. Vous pouvez laisser libre cours à votre imagination. Pour ceux qui bloquent, je vous propose ceci : un > si le perso attaque vers la droite et un < s'il attaque vers la gauche.
Vous pouvez aussi adapter le signe en fonction de la vie par exemple… Mais ne vous emballez pas trop car, plus tard, on fera différent type de perso donc on leur donnera des design différents… et il ne faudrait pas s'emmêler les pinceaux.

 Implémentez `move`. Quand il bouge, le personnage fait un pas en avant. C'est-à-dire qu'il avance d'une case… ce qui dépendra de sa direction. Calculez donc cette nouvelle position puis appelez la méthode `place_character` de `Game` pour que la partie s'occupe de déplacer le perso.
Si la partie ne bouge pas le perso, c'est que ce n'est pas possible. Tant pis, on ne fait rien de plus alors.

 Il est temps d'implémenter `get_hit`. Lorsqu'il **prend des dommages**, un personnage enlève ces dommages de ses points de vie. Dans le cas ou le personnage est tué, il faut le retirer de son équipe (= l'équipe de son joueur). Lorsqu'un perso meurt, il rapporte de l'argent (qui correspond à la moitié de son prix) à celui qui l'a tué. Pour ce faire, la fonction `get_hit` retourne la « récompense ».

Implémentez maintenant `attack`. Si vous avez besoin d'aide, en voici :

Pour **l'attaque**, elle dépendra de si le personnage est en face de la base adverse ou non.

Un perso est en face de la base adverse si :

- *Il attaque vers la gauche et est sur la colonne 0*
- *Il attaque vers la droite et est sur la dernière colonne (qui dépend de la taille du plateau de jeu)*

S'il est **devant la base**, il tape (avec sa force) directement sur l'ennemi.
L'ennemi est un joueur (que vous trouvez grâce à l'attribut dérivable `enemy`). Vous le tapez donc via sa méthode `get_hit`.

S'il est sur le champ de batail *(donc pas devant la base)*, calculez quelle est la case devant lui *(ça dépend de sa direction)* puis prenez le personnage présent sur cette case *(vous vous souvenez de `get_character_at` de `Game` ?)*. Si un tel perso existe, tapez le… et pensez à récupérer la récompense s'il y en a une. *Cette récompense est à ajouter à la monnaie du joueur.*

Implémentez `play_turn` qui permet au personnage de jouer un tour. C'est-à-dire de bouger puis d'attaquer.

On vient de définir des méthodes pour faire ça… donc ne réécrivez pas tout hein, soyez malins ! Appelez simplement la méthode qui permet de bouger puis celle pour attaquer.

Implémentez `__str__` pour qu'il retourne une chaine de caractère représentative de l'objet. Par exemple « Personnage (1$) – vie : 5 – force : 1 ».

Ça vous aidera pour tester…

D'ailleurs… vous avez pensé à tester ???

Super, vous avancez bien ! Le plus difficile est derrière vous !

Finissons Game

Terminons notre jeu de base !

Vous allez voir, on a bien bossé jusqu'ici et on va pouvoir réutiliser tout ça pour nous faciliter la vie maintenant.

Terminez d'écrire la méthode `draw` qui permet d'afficher le plateau. Actuellement, la méthode affiche des points partout. Faites en sorte que les personnages soient également affichés.

À chaque case, s'il y a un personnage à cette position, alors, j'affiche son design (sinon, un point) … On a tout ce qu'il faut pour faire ça facilement.

Implémentez `play_turn` qui permet de jouer un tour de jeu. Il se passe 4 choses lors d'un tour de jeu :

- Le joueur actuel peut ajouter un personnage
- Les personnages du joueur actuel jouent leur tour
- Les personnages du joueur adverse jouent leur tour
- Le plateau est affiché

Il n'y a plus grand-chose à implémenter à ce stade car on va pouvoir appeler la méthode adéquate pour ajouter un nouveau personnage au joueur (allez voir dans la classe joueur… on a ce qu'il faut). Ensuite, il faut appeler la méthode `play_turn` sur chaque (tiens, une boucle !) personnage de l'équipe du joueur actuel puis adverse (et on a des attributs dérivables pour retrouver chaque joueur facilement). Pour finir, on a déjà la méthode pour afficher, il suffit de l'appeler.

Il reste à implémenter `play` qui joue une partie complète. Faites-le. Tant que le joueur actuel est en vie, la partie joue un tour puis changer le `player_turn`.

Bloc main

Il est temps de tester tout ça !

Mais j'espère franchement que vous avez déjà un peu testé…

Dans le bloc main, créez 2 joueurs et une partie… et lancez le jeu !