Pour avoir accès aux classes et aux objets déclarés dans le fichier troll.py, il vous suffit d'écrire la ligne suivante au début de votre fichier

```
import troll
```

Pour créer un objet partie0 de type Partie avec x cases et y pierres, la syntaxe est la suivante :

```
partie0 = troll.Partie(x, y)
```

Pour afficher l'état d'une partie, on peut utiliser la fonction print (la méthode __repr__ a été implémentée dans la classe Partie).
Ainsi, pour afficher l'état d'une partie partie0, il suffit d'écrire

```
import troll
partie0 = troll.Partie(9, 20)
print(partie0)
```

```
-------------------------------------------------------------------------
-
```

L'état d'une partie est stocké dans les attributs de l'objet correspondant.

En voici la liste complète :

Un entier **nombreCases** qui représente le nombre de cases entre les deux châteaux (en incluant les cases des châteaux)
Un entier **positionTroll** qui indique la position du troll sur ce chemin
Un entier **stockInitial** qui indique le nombre initial de pierres
Un entier **stockGauche** qui indique le nombre de pierres restantes dans la réserve du château de gauche
Un entier **stockDroite** qui indique le nombre de pierres restantes dans la réserve du château de droite
Un entier **gagnant** qui indique l'éventuel gagnant de la partie

-> 0 si la partie n'est pas encore terminée
-> 1 si le joueur de gauche a gagné la partie
-> 2 si le joueur de droite a gagné la partie
-> 3 si la partie s'est terminée sur un match nul

Une liste **coupsPrecedents**, qui décrit les coups joués depuis le début de la partie :

-> Les coups sont listés du plus ancien au plus récent
-> Chaque tour de jeu est représenté par un couple de valeurs, qui contient
- le nombre de pierres joué par le joueur de gauche à ce tour

- le nombre de pierres joué par le joueur de droite à ce tour

Vous pouvez accéder directement à chacun de ces attributs grâce à la syntaxe suivante :
nomObjet.nomAttribut
Exemples

```
p = partie0.positionTroll
print(p)


g = partie0.gagnant
print(g)


l = partie0.coupsPrecedents
print(l)
```

Tour de jeu

La classe **Partie** possède plusieurs méthodes, dont notamment la méthode **tourDeJeu**, qui comme son nom l'indique permet de jouer un tour. Cette fonction prend deux arguments :

le nombre de pierres lancées par le joueur de gauche
le nombre de pierres lancées par le joueur de droite.

Exemple

```
partie0.tourDeJeu(2,4)
print(partie0) # Affiche



partie0.tourDeJeu(4,7)
print(partie0) # Affiche



partie0.tourDeJeu(2,4)
print(partie0)
```

Simuler une partie

Le fichier source contient également une fonction **jouerPartie**, qui attend les arguments suivants :

- Un entier **nombreCases** qui correspond au nombre de cases entre les deux châteaux pour cette simulation
- Un entier **stockInitial** qui correspond au stock initial de pierres dans chaque château pour cette simulation
- Une fonction **strategieGauche**, qui correspond à l'IA utilisée par le joueur du château de gauche
- Une fonction **strategieDroite**, qui correspond à l'IA utilisée par le joueur du château de droite

Vous avez également à votre disposition de deux exemples de stratégies basiques, intitulées **strategieExemple1** et **strategieExemple2**.
Comme ces fonctions font partie du module **troll** (importé au début du TD), on utilisera **troll.nomFonction** pour y faire appel.

Par exemple, pour lancer une partie entre les deux stratégies d'exemple, il vous suffit donc d'écrire les lignes suivante

```
troll.jouerPartie(7, 15, troll.strategieExemple1, troll.strategieExemple2)
```

Simuler plusieurs parties
Dans le même ordre d'idée, le fichier source contient aussi une fonction **jouerPlusieursParties**, qui attend les mêmes arguments que la fonction **jouerPartie**:

- Un entier **nombreCases** qui correspond au nombre de cases entre les deux châteaux pour cette simulation
- Un entier **stockInitial** qui correspond au stock initial de pierres dans chaque château pour cette simulation
- Une fonction **strategieGauche**, qui correspond à l'IA utilisée par le joueur du château de gauche
- Une fonction **strategieDroite**, qui correspond à l'IA utilisée par le joueur du château de droite

Contrairement à la fonction précédente, cette fonction organise 1000 matchs successifs entre les deux stratégies :

```
troll.jouerPlusieursParties(7, 15, troll.strategieExemple1,
troll.strategieExemple2)
```

Une stratégie est simplement une fonction qui, étant donné l'état d'un affrontement entre deux joueurs, indique le prochain coup à jouer pour le joueur de gauche.

Plus précisément, une stratégie est une fonction qui prend deux arguments :

- Un objet de type **Partie**, nommé **partieEnCours**, qui correspond à la partie en cours
- Une liste d'objets de type Partie, nommée **partiesPrecedentes**, qui contient la liste des parties déjà jouées entre les deux adversaires (dans le cas où on organise 1000 matchs successifs).

Cette fonction doit renvoyer un nombre entier, correspondant au nombre que devrait selon vous renvoyer le joueur de gauche dans cette situation (le programme de simulation s'occupera d'adapter ce code s'il doit être utilisé pour le château de droite).

Ecrire votre première stratégie pour "Trolls & Châteaux".

Exemples

```
def strategieTresTimide(partieEnCours, partiesPrecedentes):

   return 1

def strategiePasAssezSubtile(partieEnCours, partiesPrecedentes):

   return partieEnCours.stockGauche

troll.jouerPartie(7, 15, strategieTresTimide, strategiePasAssezSubtile)
```

S'adapter à la stratégie adverse

Les stratégies de la question précédente n'utilisent pas la liste **partiesPrecedentes** : elles n'évoluent donc pas au fil des parties.

En pratique, il peut être très intéressant d'analyser les parties précédentes pour essayer de comprendre la stratégie adverse et essayer de la contrer.

On pourra pour cela utiliser par exemple la syntaxe

```
def strategieEvoluee(partieEnCours, partiesPrecedentes):
   if (partiesPrecedentes == []):
      # Code pour la première partie
      if (partieEnCours.coupsPrecedents == []):
         # Code pour le premier coup de la première partie
         # ...
      else:
```

```
        # Code pour les coups suivants de la première partie
        dernierTour = partieEnCours.coupsPrecedents.pop()
        dernierCoupAdverse = dernierTour[1]
        # ...
   else:
      # Code pour les parties suivantes
      # ...
```

Simulations et classement

Chaque IA va affronter toutes les autres sur 4 catégories :

- 7 cases et 15 pierres
- 7 cases et 30 pierres
- 15 cases et 30 pierres
- 15 cases et 50 pierres

Pour chaque catégorie, chaque IA affronte chacune des autres sur 1000 matchs :

- l'IA qui remporte le plus de matches est déclarée vainqueur et gagne 3 points.
- En cas de match nul, chaque IA gagne 1 point.
- Si une IA propose un coup invalide (c'est-à-dire un nombre de pierres négatif ou nul, ou un nombre supérieur au stock de pierres restantes), cet IA est disqualifiée pour cet affrontement et perd 1 point.
- Si les deux IA proposent simultanément un coup invalide, elles sont toutes deux disqualifiées et perdent toutes deux 1 point.

L'IA qui obtiendra le plus de points, toutes catégories confondues, sera nommée la meilleure IA (et ses créateurs recevront un cadeau surprise) !

Remarque

- Vous avez tout à fait le droit de prévoir des stratégies différentes pour ces 4 configurations. Vous pouvez par exemple utiliser la syntaxe suivante :

```
def maStrategie(partieEnCours, partiesPrecedentes):
   if (partieEnCours.nombreCases == 7):
      if (partieEnCours.stockInitial == 15):
         return maStrategie7Cases15Pierres(partieEnCours,
partiesPrecedentes)
      else:
         return maStrategie7Cases30Pierres(partieEnCours,
partiesPrecedentes)
   else:
```

```
      if (partieEnCours.stockInitial == 30):
         return maStrategie15Cases30Pierres(partieEnCours,
partiesPrecedentes)
      else:
         return maStrategie15Cases50Pierres(partieEnCours,
partiesPrecedentes)

def maStrategie7Cases15Pierres(partieEnCours, partiesPrecedentes):
   # Code pour 7 cases et 15 pierres
   # ...

def maStrategie7Cases30Pierres(partieEnCours, partiesPrecedentes):
   # Code pour 7 cases et 30 pierres
   # ...

def maStrategie15Cases30Pierres(partieEnCours, partiesPrecedentes):
   # Code pour 15 cases et 30 pierres
   # ...

def maStrategie15Cases50Pierres(partieEnCours, partiesPrecedentes):
   # Code pour 15 cases et 50 pierres
   # ...
```