TD_POO

January 7, 2021

1 Exercice 1 (Niveau : facile)

1. Écrire une **classe** `Eleve` qui contiendra les attributs `nom`, `classe` et `note`.
2. **Instancier** trois élèves de cette classe.
3. Écrire une **fonction** `compare` :
   - qui prend en **paramètres** 2 élèves
   - qui **renvoie** le nom de l'élève ayant la meilleure note. (en cas de note identique, la fonction devra renvoyer un **tuple** contenant les noms des 2 élèves.
4. Ecrire quelques appels de la fonction `compare` pour la tester.
5. Doter la classe `Eleve` d'une **méthode** `a_la_moyenne` qui renvoie `True` si l'élève a obtenu la moyenne. Tester

2 Exercice 2.1 (Niveau : facile)

1. Écrire une classe `TriangleRectangle` qui contiendra les attributs `cote1`, `cote2` et `hypothenuse`. La méthode constructeur ne prendra en paramètres que `cote1` et `cote2`, l'attribut `hypothenuse` pouvant se déduire des 2 autres côtés en utilisant le théorème de pythagore (conseil : utiliser le module `math`)
2. Instancier un triangle de type `TriangleRectangle` et afficher les 3 attributs du triangle pour vérifier que tout fonctionne

3 Exercice 2.2 (Niveau : intermédiaire)

On considère la classe `TriangleRectangle` de l'exercice précédent (si nécessaire, prendre le corrigé)

1. Soient les lignes de codes suivantes. Qu'est-ce qui est incohérent dans ces lignes de code ? (on ne demande pas de les corriger)

[2]:
```
triangle = TriangleRectangle(3,4)
print(triangle.cote1)
print(triangle.cote2)
print(triangle.hypothenuse)

triangle.cote1 = 4
print(triangle.cote1)
print(triangle.cote2)
print(triangle.hypothenuse)
```

```
3
4
5.0
4
4
5.0
```

2. Comment peut-on “protéger” le code de la classe `TriangleRectangle` contre une telle utilisation ?

   On veut laisser la possibilité aux utilisateurs de la classe `TriangleRectangle` de : * Accéder aux 3 côtés du triangle * Modifier `cote1` et `cote2` de manière cohérente (mais pas `hypothénuse`)

3. Doter la classe des méthodes `get_cote1`, `get_cote2`, `get_hypothénuse`, `set_cote1` et `set_cote2`
   Remarque : On pourra, dans un souci de factorisation du code créer une méthode “privée” `__calcul_hypothenuse`

4. Pour tester :
   - Instancier un triangle
   - Modifier les côtés du triangle
   - Vérifier que le triangle reste “cohérent” (c’est-à-dire que le triangle est resté un triangle rectangle)
5. Répartir les méthodes et attributs de la classe `TriangleRectangle` dans un tableau à 2 colonnes suivant selon qu’ils relèvent de l’implémentation ou de l’interface

IMPLEMENTATION	INTERFACE

4 Exercice 3 : un jeu de rôle (Niveau intermédiaire)

Dans cet exercice on se propose de mettre en place une “boîte à outil” (typiquement des `class`) pouvant être utilisée pour programmer un jeu de rôle. Tout se déroulera en “mode texte” : Il serait bien sûr plus agréable d’avoir une interface graphique mais ceci demanderait beaucoup de travail et serait largement hors programme…

1. Comme souvent dans les jeux de rôles, il y a des personnages et parfois ceux-ci sont amenés à se battre… Analyser le code suivant de la boîte à outil : que fait-il ? comment fonctionne t’il ?

[104]:
```python
import random
class Personnage:

    def __init__(self, points_de_vie):
        self.vie = points_de_vie

    def perd_vie(self):
```

```
        if random.random() < 0.5:
            nbPoint = 1
        else:
            nbPoint = 2

        self.vie = self.vie - nbPoint

def game():
    bilbo = Personnage(20)
    gollum = Personnage(20)

    while bilbo.vie > 0 and gollum.vie > 0:
        bilbo.perd_vie()
        gollum.perd_vie()

    if bilbo.vie <= 0 and gollum.vie > 0:
        msg = "Gollum est vainqueur, il lui reste encore " + str(gollum.vie) +␣
↪" points alors que Bilbo est mort"
    elif gollum.vie <= 0 and bilbo.vie > 0:
        msg = "Bilbo est vainqueur, il lui reste encore " + str(bilbo.vie) + "␣
↪points alors que Gollum est mort"
    else:
        msg = "Les deux combattants sont morts en même temps"

    return msg
```

La suite de cet exercice va consister à améliorer la boîte à outil pour l'instant rudimentaire. Pour cela, il faudra progresivement modifier et compléter le code précédent…

4.1 Amélioration 1 : Nommer les personnages

Pour l'instant on ne peut donner de nom au personnage !

1. Créer un attribut `nom` qu'on doit donner en premier paramètre, ce qui permet de créer une instance de `Personnage` comme ceci :

[ ]:
```
gollum = Personnage("Gollum", 20)
```

2. Modifier la fonction `Game` pour qu'elle tienne compte du nom des personnages. On doit pouvoir créer d'autres personnages et les messages doivent tenir compte des noms de ceux-ci. Remarquez que la fonction `game` a changé : elle doit prendre en paramètre les personnages qui vont combattre :

[230]:
```
frodon = Personnage("Frodon", 20)
araignee = Personnage("Araignée", 10)
game(frodon, araignee)
```

[230]:
```
'Frodon est vainqueur, il lui reste encore 9 points alors que Araignée est mort'
```

4.2 Amélioration 2 : Afficher un journal détaillé du combat

Améliorer encore la fonction `game` pour qu'elle affiche un journal détaillé du combat :

[240]:
```
aragorn = Personnage("Aragorn", 10)
orc = Personnage("Orc", 10)
game(aragorn, orc)
```

```
Aragorn perd 1 point de vie
Orc perd 1 point de vie
Aragorn perd 2 point de vie
Orc perd 1 point de vie
Aragorn perd 2 point de vie
Orc perd 1 point de vie
Aragorn perd 1 point de vie
Orc perd 2 point de vie
Aragorn perd 2 point de vie
Orc perd 1 point de vie
Aragorn perd 2 point de vie
Orc perd 1 point de vie
```

[240]:
```
'Orc est vainqueur, il lui reste encore 3 points alors que Aragorn est mort'
```

4.3 Amélioration 3 : Prendre en compte l'aptitude au combat des personnages

> On souhaite modéliser le fait que les personnages ont des aptitudes différentes au combat (suivant leur entraînement, leur condition physique, leur expérience, etc…). On va donc leur attribuer une nouvelle caractéristique `aptitude_combat`. On va donc créer un nouvel attribut `aptitude_combat` lors de l'instanciation du personnage. C'est un entier entre 0 et 4.

Implémentez l'aptitude au combat des personnages qui doit satisfaire les 3 points suivants :

1. Il faudra pouvoir créer nos personnages ainsi :

[ ]:
```
aragorn = Personnage("Aragorn", 30, 2)
```

2. L'effet de l'aptitude au combat est le suivant : Dans la méthode `perd_vie`, on tire toujours un nombre aléatoire entre 0 et 1 :
   - Si ce nombre multiplié par 10 dépasse l'aptitude au combat du personnage, il perd un point de vie.
   - Sinon il ne perd pas de vie

Par exemple, Aragorn a une aptitude au combat de 2 : * Si dans `perd_vie`, on tire 0.3 $\Longrightarrow$ 10 * 0.3 = 3 et 3 > 2 $\Longrightarrow$ il perd un point de vie. * Si dans `perd_vie`, on tire 0.12345 $\Longrightarrow$ 10 * 0.12345 = 1.2345 et 1.2345 < 2 $\Longrightarrow$ il ne perd pas de point de vie.

Attention, si vous donnez une aptitude au combat trop élevée, le personnage ne perdra jamais de vie et la boucle de la fonction game pourrait être infinie : il suffit de donner une aptitude au combat de 10 pour qu'un personnage soit invicible !

3. Créer une méthode interne `__limiter_aptitude_combat` qui empêche de créer un personnage ayant une aptitude au combat supérieure à 4 : Si le paramètre chance est inférieure ou égale à 4, il est inchangé, S'il dépasse 4, il est ramené à 4. Cette méthode interne ne sera pas appelée par les éléments extérieurs au programme, seulement par le programme lui même : On utilise cette méthode interne dans **init**, il faut penser à l'appeler.

4.4 Amélioration libre

Implémentez une nouvelle amélioration pour ce jeu. Vous êtes libre de choisir ce que vous voulez implémenter. Voici à titre d'exemples quelques idées (sans obligation) :

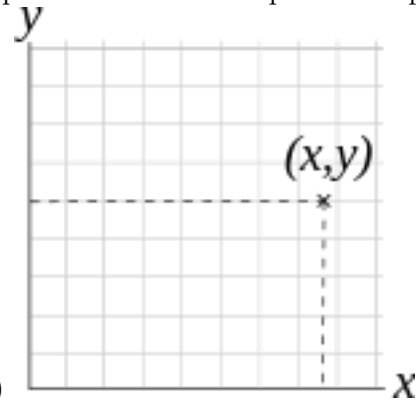
On pourrait par exemple créer une nouvelle classe `Combat` qui prendrait en paramètres deux personnages. Un combat se déroulerait par tours et à chaque tour les personnages pourraient prendre des initiatives comme : * `frapper` qui dépendrait de la chance et d'un attribut `force` à définir : `frapper` pourrait renvoyer un nombre aléatoire entre 1 et `force`, par exemple. Et c'est ce nombre qui définirait le nombre de points perdus par le personnage… * `se_rendre` qui donnerait le choix à un des combattants d'abandonner le combat. On pourrait imaginer un attribut `statut` pour les personnages indiquant si celui-ci est libre ou fait prisonnier

5 Exercice 4 (Niveau : facile)

Attention : Pour toutes les méthodes et fonctions de ce cours, on demande d'écrire leur documentation. On pourra se servir de ce modèle à adapter et à compléter pour la docstring (voir cours de première)

[ ]:
```python
# Exemple de docstring pour une fonction à 2 paramètres
def nom_de_la_fonction(parametre1,parametre2):
    """
    Description de la fonction :
    parametre1 (type) :
    parametre2 (type) :
    return (type) :
    """
    # code de la fonction
```

1. Créer une classe `Point` permettant de modéliser un point de coordonnées (x,y) dans un repère orthonormé du plan. Chaque point aura 2 attributs (son abscisse `x` et son ordonnée y)



2. Doter la classe d'une méthode `distance` permettant de renvoer la distance entre le point et l'origine du repère. On rappelle que $distance = \sqrt{x^2 + y^2}$
3. Doter la classe de la méthode spéciale `__repr__` permettant d'afficher ou de représenter un point comme ceci `(x,y)` où `x` et `y` représente les coordonnées du point
4. Instancier ces 4 points et calculer leur distance à l'origine :
   - A(-2,5)
   - B(5,5)
   - C(-2,-2)
   - D(5,-2)
5. Ecrire une **fonction** `longueur_segment` qui :
   - prend 2 points en paramètres
   - renvoie la distance entre ces 2 points. On rappelle que $distAB = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$
6. Calculer la distance des segments `[A,B]` et `[C,D]`

6 Exercice 5 (Niveau intermédiaire)

L'objectif de cet exercice est de définir une classe `Fraction` pour représenter un nombre rationnel. Cette classe possède 2 attributs, `num` et `denom`, qui sont de type entier et désignent respectivement le numérateur et le dénominateur. On se limitera au cas où le dénominateur est un entier strictement positif.

6.1 Première partie

Dans cette partie, on ne cherche pas à exprimer les fractions sous leur forme irréductible

1. Ecrire le constructeur de cette classe
2. Ajouter la méthode spéciale `__repr__` qui renvoie une chaîne de caractères de la forme `12 / 35`, ou simplement `12` lorsque le dénominateur vaut un.
3. Ajouter les 2 méthodes spéciales `__eq__` et `__lt__` qui reçoivent en paramètre une deuxième fraction. Ces méthodes spéciales permettent de définir respectivement les opérateurs de comparaison `==` et `<` pour des objets de type `Fraction`.
4. Ajouter les 2 méthodes spéciales `__add__` et `__mul__` qui reçoivent en paramètre une deuxième fraction. Ces méthodes spéciales permettent de définir respectivement les opérateurs arithmétiques *somme* et *multiplication* pour des objets de type `Fraction`.
5. Instancier quelques objet de type `Fraction` et tester toutes ces méthodes

6.2 Deuxième partie

Dans cette partie, on souhaite améliorer l'implémentation précédente de `Fraction` afin d'exprimer **toutes** les fractions sous leur forme irréductible

1. Ecrire une fonction `pgcd` qui :
   - prend en paramètres nombres entiers
   - renvoie le PGCD de ces 2 entiers

Indice : On pourra utiliser un algorithme récursif : L'algorithme d'Euclide

Cette fonction sera bien évidemment utile pour atteindre l'objectif de cette deuxième partie...

2. Modifier l'implémentation écrite en première partie afin d'exprimer toutes les fractions sous leur forme irréductible. *Conseil : Avant de vous lancer dans le code, identifier dans l'implémentation précédente ce qui doit-être modifié de ce qui peut être conservé*