Structure de données

Introduction

Jusqu'à présent nous avons utilisé des objets dont le type est prédéfini :
`int` , `float`, `bool` , `str` , `list` , `tuple`, `dict`
Nous avons même appris à utiliser des méthodes associées à ces types, comme `ma_liste.append(valeur)`.

Python, comme d'autres langages (Java, C++, ...), est un langage orienté objet. On peut même dire que tout y est objet.

Une variable de type int est en fait un objet de type int donc construit à partir de la classe int. Pareil pour les float et string. Mais également pour les list, tuple, dict, etc.

On peut créer ses propres objets qui auront leurs méthodes, ce qui aura pour effet:
- De faciliter l'implémentation de données
- D'augmenter la lisibilité du programme

Définition :
Une **classe** définit des **objets** qui sont des **instances** (des représentants) de cette classe. On utilisera le mot objet ou instance pour désigner la même chose. Les objets peuvent posséder des **attributs** (variables associées aux objets) et des **méthodes** (qui sont des fonctions associées aux objets et qui peuvent agir sur ces derniers ou encore les utiliser).

Je comprends par l'exemple...

Imaginons que nous ayons un objet Eleve, qui permettrait pour un élève de stocker dans une variable eleve1 des données comme son nom, son prénom, sa date de naissance et ses notes dans différentes matières, ainsi qu'une méthode qui calculerait sa moyenne.

Création de la classe

Méthode :
Pour créer une classe on utilise le mot-clé : `class` suivi par le nom de la classe et des incontournables " : "
Le nom de la classe commence par une majuscule et ne contient que des caractères alphanumériques.

```
# création de la classe Eleve
class Eleve:
```

Ajoutons l'instruction **pass** et créons une variable eleve1 de type Eleve (*une instance de la classe Eleve*).

```
class Eleve:
    pass # signifie ne fait rien...

eleve1=Eleve()
print(type(eleve1))
```

Le résultat <class '__main__.Eleve'> signifie que eleve1 est une instance de la classe Eleve (une variable de type Eleve).

Les attributs

Les attributs sont des variables associées à la classe. Il y en a de deux types:

- **Les attributs de classe :** Un attribut de classe (ou variable de classe) est un attribut qui sera identique pour chaque instance et n'a pas vocation à être changé.

- **Les attributs d'instance :** Une variable ou attribut d'instance est une variable accrochée à une instance et qui est spécifique à cette instance. Et d'une instance à l'autre il ne prendra pas forcément la même valeur.

Pour notre exemple, nous choisirons les matières comme attributs de classe et les nom, prénom, date de naissance et les notes comme attributs d'instance:

Méthode :

Le constructeur:

L'endroit le plus approprié pour déclarer un attribut est à l'intérieur d'une méthode appelée le constructeur. S'il est défini, il est implicitement exécuté lors de la création de chaque instance. Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe à ceci près que son nom est imposé : **__init__** (deux underscores de chaque côté...). Hormis le premier paramètre, invariablement **self**, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

```
class Eleve:
    # attributs de classe
    matiere1="Programmation"
    matiere2="Algorithmique"
    matiere3="Projet"
    # constructeur (attention deux underscores de chaque côté !!)
    def __init__(self,Nom,Prenom,Date,Note1,Note2,Note3):
        # attributs d'instance
        self.nom=Nom
        self.prenom=Prenom
        self.date=Date
        self.note_mat1=Note1
        self.note_mat2=Note2
        self.note_mat3=Note3
#création d'un élève
eleve1=Eleve("Térieur","Alain","01/01/2000",12,10,15)
```

Comment accéder aux valeurs des attributs d'instance créés?
On utilise la syntaxe : `instance.attribut` *(notez bien le point)*

```
print(eleve1.prenom,eleve1.nom)
print(eleve1.matiere1,":",eleve1.note_mat1)
print(eleve1.matiere2,":",eleve1.note_mat2)
print(eleve1.matiere3,":",eleve1.note_mat3)
```

? Question 1:

Noter ici l'affichage obtenu:

.....
.....
.....
.....

Les Méthodes

Il serait intéressant de pouvoir obtenir la moyenne de l'élève avec l'instruction `eleve1.moyenne()`. c'est à dire en appliquant la méthode `moyenne()` à l'instance `eleve1`

Pour cela on crée une fonction à l'intérieur de la classe `Eleve` qui retourne la moyenne de l'élève :

```
    def moyenne(self):
        return ((self.note_mat1+self.note_mat2+self.note_mat3)/3)
```

À faire 1:

Ajouter cette méthode et faire afficher la moyenne de l'élève.
Ajouter ces deux élèves et faire afficher leurs résultats et leurs moyennes.

- Nom : Onette
- Prénom : Camille
- Date: 01/07/2004
- Programmation: 7
- Algorithmique : 14
- Projet : 11

- Nom : Oma
- Prénom : Modeste
- Date: 01/11/2002
- Programmation: 13
- Algorithmique : 8
- Projet : 17

? Exercice 1 :

Écrire une fonction (hors de la classe bien sûr..) qui prend en paramètres une liste constituée de ces trois élèves et qui retourne les moyennes par matières.
On attend le rendu :
{'Programmation': 10.666666666666666, 'Algorithmique': 10.666666666666666, 'Projet': 14.333333333333334}

```
    #Écrire votre fonction ici
```

La documentation

Notre classe `Eleve` est une structure de données qui peut être utilisée dans différents programmes par différents programmeurs, il est donc important voire primordial de bien la documenter.

Cette documentation doit montrer à minima comment sont construites les instances, quels sont les attributs et les méthodes disponibles.

Cette documentation est accessible via l'instruction : **`help`**`(Eleve)`
Par exemple voici une documentation possible :

```
class Eleve:
    '''
    Création d'une instance eleve:
    eleve=Eleve(nom(str),prenom(str),date(str),note1(programm)(float)
    ,note2(algo)(float),note3(projet(float)))
    attributs d'instance : nom,prenom,date,note_mat1,note_mat2,note_mat3
    attributs de classe: matiere1,matiere2,matiere3
    Méthode : moyenne() retourne la moyenne de l'élève
    '''
    .....
```

C'est d'autant plus important lorsque cette classe se trouve dans un autre fichier:

À FAIRE 2:

Copier coller la classe `Eleve` dans un fichier que vous enregistrerez sous le nom Maclasse.py
Créer un autre fichier mon_programme.py, dans lequel vous écrirez :
(*les deux fichiers doivent être dans le même dossier...*)

```
from Maclasse import Eleve
eleve1=Eleve("Térieur","Alain","01/01/2000",12,10,15)
print(eleve1.prenom,eleve1.nom,":",eleve1.moyenne()
```

Si vous donnez cette classe à un autre programmeur, il faut qu'il puisse savoir comment l'utiliser sans avoir à en décortiquer le code...

Remarque :
Nous avons construit une structure de données répondant à un cahier des charges pas trop ambitieux. On pourrait se poser les questions suivantes:

- Que faire si un élève n'a pas de notes dans une des matières?
- Comment rendre impossible la saisie d'une note supérieure à 20?
- Etc.

Un autre exemple...

Python comme la plupart des langages ne dispose pas d'une structure de données de type rationnel (des fractions..)
Nous allons en créer une ...(même s'il est certain que cela existe déjà...)

Le cahier des charges

On doit créer une classe `Rationnel` dont les instances auront les attributs `numerateur` et `denominateur`(celui-ci ne pourra être nul!!) et des méthodes pour:

- Simplifier la fraction et normaliser son écriture
- Additionner deux fractions
- Soustraire deux fractions
- Multiplier et diviser deux fractions

Méthode :
Un peu de Mathématiques..
la fraction $\frac{12}{-15}$ doit s'écrire $\frac{-4}{5}$
Il y a eu simplification par le PGCD de 12 et 15 et transfert du signe au numérateur
Il nous faudra:

- Une fonction qui calcule le PGCD
- Une méthode pour simplifier et normaliser la fraction.

À faire 3:
Créer un fichier `ratio.py` et écrire le code suivant :

```
# fichier ratio.py
class Rationnel:
    # création des instances
    def __init__(self,num,den=1):# par défaut le dénominateur vaut 1
        if den == 0:
            # on déclenche une exeption spécifique
            raise ZeroDivisionError('denominateur nul')
        else:
            self.num=num
            self.den=den
    # pour voir une fraction sur la console appelée par print
    def __str__(self):
        return str(self.num)+'/'+str(self.den)
```

La méthode `__str__` *permet de gérer l'affichage de la fonction* **print**

Dans un autre fichier `test.py`, écrire le code suivant et expliquer les affichages dans la console

```
from ratio import Rationnel

f=Rationnel(12,-15)
print(f)
print(f.num,f.den)

f1=Rationnel(7)
print(f1)
print(type(f1))
```

.....
.....
.....
.....

Simplification de la fraction

Pour cela il nous faut une fonction qui calcule le PGCD de deux entiers.

À FAIRE 4:

Un peu d'algorithmique...

Écrire une fonction `pgcd(x,y)` qui retourne le pgcd des entiers x et y passés en paramètres et la mettre dans le fichier `ratio.py` (en dehors de la classe):

```
def pgcd(x,y):
    # code
    return ...

class Rationnel:
    # création des instances
    def __init__(self,num,den=1):# par défaut le dénominateur vaut 1
...
```

À FAIRE 5:

Voici la fonction à rajouter dans la classe Rationnel:

```
    #simplification des fractions
    def normalise(self):
        g = pgcd(abs(self.num), abs(self.den))
        self.num = self.num // g
        self.den = self.den //g
        if(self.num*self.den<0):
            if self.den<0:
                self.den=-self.den
                self.num=-self.num
        else:
            if self.den <0:
                self.den=-self.den
                self.num=-self.num
```

Écrire un algorithme en langage naturel qui représente cette fonction:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Pour que la fraction soit automatiquement simplifiée et normalisée, il suffit de l'exécuter dans la fonction **__init__**:

```
    def __init__(self,num,den=1):
        if den == 0:
            # on déclenche une exeption spécifique
            raise ZeroDivisionError('denominateur nul')
        else:
            self.num=num
            self.den=den
            self.normalise()
```

À FAIRE 6:

Exécuter de nouveau le code dans le fichier `test.py` et vérifier que l'on obtient bien :

```
-4/5
-4 5
7/1
<class 'ratio.Rationnel'>
```

Une première méthode : l'addition

On pourrait imaginer une fonction `add(self, autre)`:

```
def add(self,other):
   return Rationnel(self.num*other.den+self.den*other.num,self.den*other.den)
```

Mais l'écriture d'une addition de deux fractions f1+f2 serait `f1.add(f2)`

```
f1=Rationnel(2,7)
f2=Rationnel(5,3)
print(f1.add(f2))
affiche 41/21
```

Pensons alors à l'utilisateur qui s'attend à pouvoir écrire `print(f1+f2)`
Heureusement nous pouvons définir l'addition '+' pour nos rationnels...

Il existe une fonction prédéfinie pour cela : `__add__`:

```
    def __add__(self, other): #addition
        n=self.num*other.den+other.num*self.den
        d=self.den*other.den
        return Rationnel(n,d)
```

À faire 7:

Rajouter cette méthode dans la classe `Rationnel` et testez le code suivant dans `test.py`

```
f1=Rationnel(2,7)
f2=Rationnel(5,3)
print(f1+f2)
```

Les autres opérations

- Pour la soustraction '-' on utilise la fonction : `__sub__(self,other)`
- Pour la multiplication '*' on utilise la fonction : `__mul__(self,other)`
- Pour la division "/" ce sera : `__truediv__(self,other)`, il faudra gérer la division par 0

À faire 8:

Réaliser ces méthodes et testez les.

Exercice 2 :

Le nombre d'Euler : $e = 2{,}718281....$ vérifie l'égalité suivante:

$$e-1 = \frac{1}{1} + \frac{1}{1\times 2} + \frac{1}{1\times 2\times 3} + ... + \frac{1}{1\times 2\times 3\times ...\times n} + ...$$

En utilisant la classe `Rationnel`, écrire une fonction `nombre_euler(n)` qui retourne une fraction qui permet d'obtenir une approximation de e.

Exercice 3 :

Toujours en utilisant la classe `Rationnel`, écrire une fonction qui permet d'obtenir une fraction donnant une approximation du nombre pi par la formule:

$$\pi = 4\times\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... + \frac{(-1)^n}{2n+1} + ...\right)$$

Prolongement possible

Pour un projet, on peut imaginer de développer davantage la structure `Rationnel` en y ajoutant d'autres méthodes qui permettent de comparer des rationnels, de calculer $\left(\frac{a}{b}\right)^n$ etc.
En documentant l'ensemble bien sûr...

Exercices

Exercice 4 :

Écrire une classe Domino pour représenter une pièce de domino.
Les objet sont initialisés avec les valeurs des deux faces, A et B.
Ajouter une méthode `affichePoints(self)` qui affiche les valeurs des deux faces, et une méthode `total(self)` qui retourne la somme des deux valeurs.

Exercice 5 :

Écrire une classe CompteBancaire. Les objets sont initialisés avec le nom du titulaire et le solde. L'argument solde doit être facultatif et avoir une valeur prédéfinie zéro. Ajouter deux méthodes `depot(self,somme)` et `retrait(self,somme)` pour changer le solde. Ajouter une méthode `affiche(self)` qui montre le solde courant.

Exercice 6 :

Écrire une classe Rectangle , permettant de construire un rectangle doté d'attributs longueur et largeur.
Ajouter deux méthodes `perimetre(self)` et `surface(self)`.

Exercice 7 :

Écrire une classe Personnage, avec son nom et ses points de vie comme attribut.
Ajouter une méthode `combat(self,other)` qui diminue de façon aléatoire les points de vie de l'un des personnages.

Exercice 8 :

Écrire une classe Robot, avec ses coordonnées comme attributs et une direction (None par défaut).
Ajouter la méthode `avancer(self)`, qui permet au robot d'avancer d'une case dans la direction choisie.

corrigés...

```
class Eleve:
    '''
    Création d'une instance eleve:
    var=Eleve(nom(str),prenom(str),date(str),note1(programm)(float),
    note2(algo)(float),note3(projet(float)))
    attributs d'instance : nom,prenom,date,note_mat1,note_mat2,note_mat3
    attributs de classe: matiere1,matiere2,matiere3
    Méthode : moyenne() retourne la moyenne de l'élève
    '''
    # attributs de classe
    matiere1="Programmation"
    matiere2="Algorithmique"
    matiere3="Projet"
    # constructeur (attention deux underscore de chaque côté !!)
    def __init__(self,Nom,Prenom,Date,Note1,Note2,Note3):
        # attributs d'instance
        self.nom=Nom
        self.prenom=Prenom
        self.date=Date
        self.note_mat1=Note1
        self.note_mat2=Note2
        self.note_mat3=Note3

    def moyenne(self):
        return ((self.note_mat1+self.note_mat2+self.note_mat3)/3)

def moy_mat(classe):
    '''
    classe est du type liste
    cette liste contient des éléments du type Eleve
    retourne un dictionnaire { clé=matière,valeur=moyenne}
    '''
    s1=0
    s2=0
    s3=0
    for el in classe:
        s1=s1+el.note_mat1
        s2=s2+el.note_mat2
        s3=s3+el.note_mat3
    return {classe[0].matiere1:s1/3,classe[0].matiere2:s2/3,
    classe[0].matiere3:s3/3}
```

```
#création d'un éleve
eleve1=Eleve("Térieur","Alain","01/01/2000",12,10,15)
eleve2=Eleve("Onette","Camille","01/07/2004",7,14,11)
eleve3=Eleve("Oma","Modeste","01/11/2002",13,8,17)
classe=[eleve1,eleve2,eleve3]



# accès aux données  et affichage
for eleve in classe:
    print(eleve.prenom,eleve.nom)
    print(eleve.matiere1,":",eleve.note_mat1)
    print(eleve.matiere2,":",eleve.note_mat2)
    print(eleve.matiere3,":",eleve.note_mat3)
    print("moyenne :",eleve.moyenne())
    print("--------------------")
print(moy_mat(classe))
```

```
# PGCD on pourra le reprendre pour la récursivité
def pgcd(x,y):
    if(x==0 or y==0):
        return 1
    if (y%x==0):
        return x
    else:
        return pgcd(y%x,x)
```

```
class Rationnel:

    #simplification des fractions
    def normalise(self):
        g = pgcd(abs(self.num), abs(self.den))
        self.num = self.num // g
        self.den = self.den //g
        if(self.num*self.den<0):
            if self.den<0:
                self.den=-self.den
                self.num=-self.num
        else:
            if self.den <0:
                self.den=-self.den
                self.num=-self.num
        #return

    def __init__(self,num,den=1):
        if den == 0:
            # on déclenche une exeption spécifique
            raise ZeroDivisionError('denominateur nul')
        else:
            self.num=num
```

```
        self.den=den
        self.normalise()


# simplification de la fraction
def simpl(self):
    return Rationnel(self.num//pgcd(self.num,self.den),
    self.den//pgcd(self.num,self.den))

# pour voir une fraction sur la console appelée par print
def __str__(self):
    return str(self.num)+'/'+str(self.den)
#teste si nulle
def nul (self):
    return self.num==0


#opérations
def __add__(self, other): #addition
    n=self.num*other.den+other.num*self.den
    d=self.den*other.den
    return Rationnel(n,d)
def __mul__(self,other): #multiplication
    n=self.num*other.num
    d=self.den*other.den
    return Rationnel(n,d)
def __sub__(self, other): #soustraction
    n=self.num*other.den-other.num*self.den
    d=self.den*other.den
    return Rationnel(n,d)
def __truediv__(self,other): #division
    if other.nul():
        raise ZeroDivisionError('diviseur nul')
    else:
        n=self.num*other.den
        d=self.den*other.num
        return Rationnel(n,d)



"""autres opérations"""
def __neg__(self): #opposé
    return Rationnel(-self.num,self.den)

def inverse(self):
    if self.nul(): #inverse
        raise ZeroDivisionError('diviseur nul')
    else:
        return Rationnel (self.den,self.num)
```

```
        def __pow__(self,n):
        if n==0:
            return Rationnel(1,1)
        elif n>0:
            return Rationnel (self.num**n,self.den**n)
        else:
            if self.nul():
                raise ZeroDivisionError('puissance negative d\' nul')
            else:
                return Rationnel(self.den**(-n),self.num**(-n))


    """Les comparaisons"""
    def __eq__(self,other):
        return self.num*other.den==self.den*other.num
    def __lt__(self,other):
        return self.num*other.den<self.den*other.num
    def __le__(self,other):
        return self.num*other.den<=self.den*other.num
    def __gt__(self,other):
        return self.num*other.den>self.den*other.num
    def __ge__(self,other):
        return self.num*other.den>=self.den*other.num
    def __ne__(self,other):
        return self.num*other.den!=self.den*other.num
```

```
# euler
def nombre_euler (n):
    h=1
    b=1
    e=Rationnel(0,1)
    for i in range (1,n+1):
        u=Rationnel(h,b)
        e=e+u
        b=b*i
    return e

#pi
def frac_pi(n):
    h=1
    b=1
    p=Rationnel(0,1)
    for i in range(1,n+1):
        u=Rationnel(h,b)
        p=p+u
        h=(-1)**i
        b=2*i+1
    return p
```