Les classes

Les classes

Classe ~ Des données et des fonctionnalitées qui modélisent une abstraction

- un "objet" réel — canard, voiture, lampe, vecteur, ...
- un concept — acheteur, afficheur, compositeur, solveur équation, ...

Classe d'étudiant

Etudiant

Nom:

Prenom:

Note:

moyenne

Données

Fonction

ex.

Nom: Dupre

Prenom: Paul

Note: 12, 13, 15

Nom: Francis

Prenom: Patrick

Note: 16, 8.5, 6.3

Classe d'étudiant

Nom: Dupre

Prenom: Paul

Note: 12, 13, 15

Nom: Francis

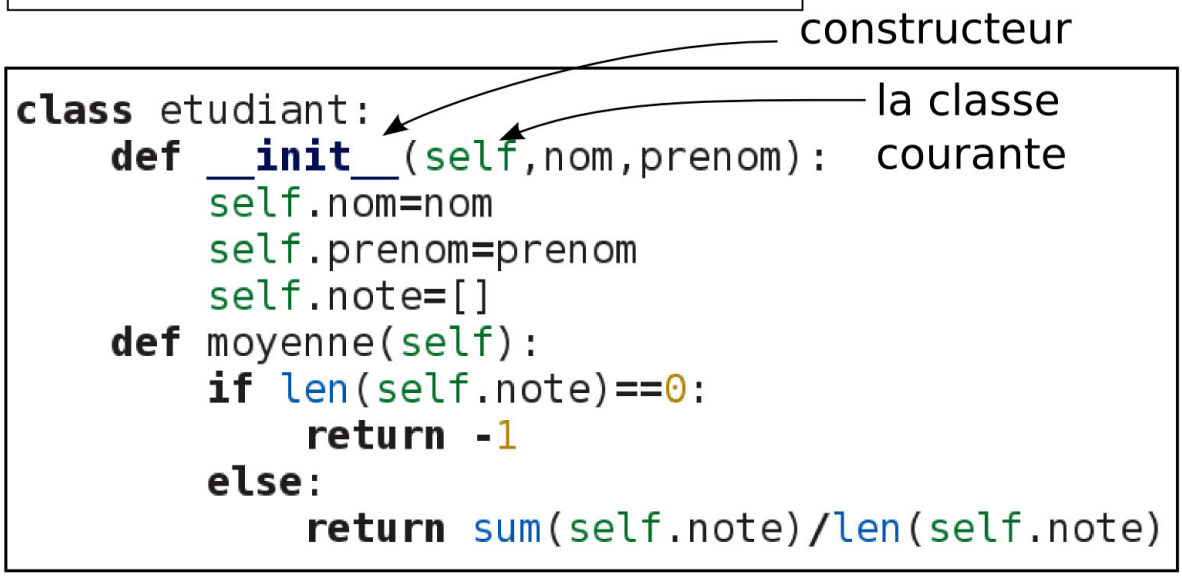
Prenom: Patrick

Note: 16, 8.5, 6.3

```
e0=etudiant("Dupre","Paul")
e1=etudiant("Francis","Patrick")

e0.note=[12,13,15]
e1.note=[16,8.5,6.3]

liste_etudiants=[e0,e1]

for e in liste_etudiants:
    print(e.nom,e.prenom,round(e.moyenne(),2))
```

Classe d'étudiant

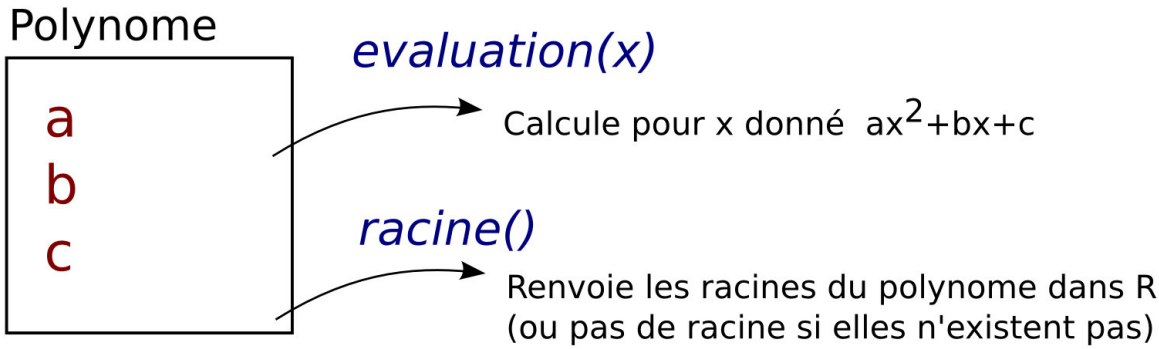
```
e0=etudiant("Dupre","Paul")
e1=etudiant("Francis","Patrick")

e0.note=[12,13,15]
e1.note=[16,8.5,6.3]

liste_etudiants=[e0,e1]

for e in liste_etudiants:
    print(e.nom,e.prenom,round(e.moyenne(),2))
```



Classe de polynome

Construire une classe gérant un polynome d'ordre 2

$p(x)=ax^2+bx+c$



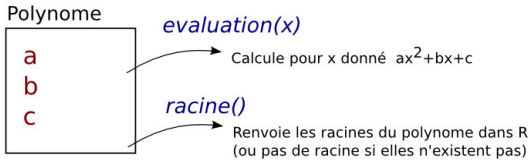
ex.

```
p0=polynome(1,4,3)
p1=polynome(1,0,1)

print(p0.evaluation(1.0))
print(p0.evaluation(1.5))

print(p0.racine())
print(p1.racine())
```

Classe de polynome

Construire une classe gérant un polynome d'ordre 2

$p(x)=ax^2+bx+c$



ex.

```
p0=polynome(1,4,3)
p1=polynome(1,0,1)

print(p0.evaluation(1.0))
print(p0.evaluation(1.5))

print(p0.racine())
print(p1.racine())
```

```
import math

class polynome:
    def __init__(self,a,b,c):
        self.a=a
        self.b=b
        self.c=c
    def evaluation(self,x):
        return self.a*x**2+self.b*x+self.c
    def racine(self):
        delta=self.b**2-4*self.a*self.c
        if delta<0:
            return "pas de solution"
        else:
            return [(-self.b+math.sqrt(delta))/(2*self.a),
                    (-self.b-math.sqrt(delta))/(2*self.a)]
```

Classe de vecteur

Surchage d'opérateur

On peut écrire c=a+b

```
class vecteur:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def __add__(self,vec):
        return vecteur(self.x+vec.x,self.y+vec.y)


a=vecteur(x=4,y=6)
b=vecteur(y=7,x=2)

c=a+b

print(c.x,c.y)
```

mot clé Python
__add__: opérateur +

Classe de vecteur

Surchage d'opérateur

On peut afficher un vecteur avec print()

```
class vecteur:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def __add__(self,vec):
        return vecteur(self.x+vec.x,self.y+vec.y)
    def __str__(self):
        return "["+str(self.x)+","+str(self.y)+"]"

a=vecteur(x=4,y=6)
b=vecteur(y=7,x=2)
c=a+b
print(c)
```

mot clé Python
__str__: conversion en string

Application polynome

Faire en sorte que la classe polynome de degré 2 puisse

- additionner 2 polynomes (somme des coefficients)
- soustraire 2 polynomes (différence des coefficients)
- multiplier un polynome avec un scalaire
- afficher un polynome avec print()
  On affichera alors: ax^2+bx+c avec les valeurs de (a,b,c)

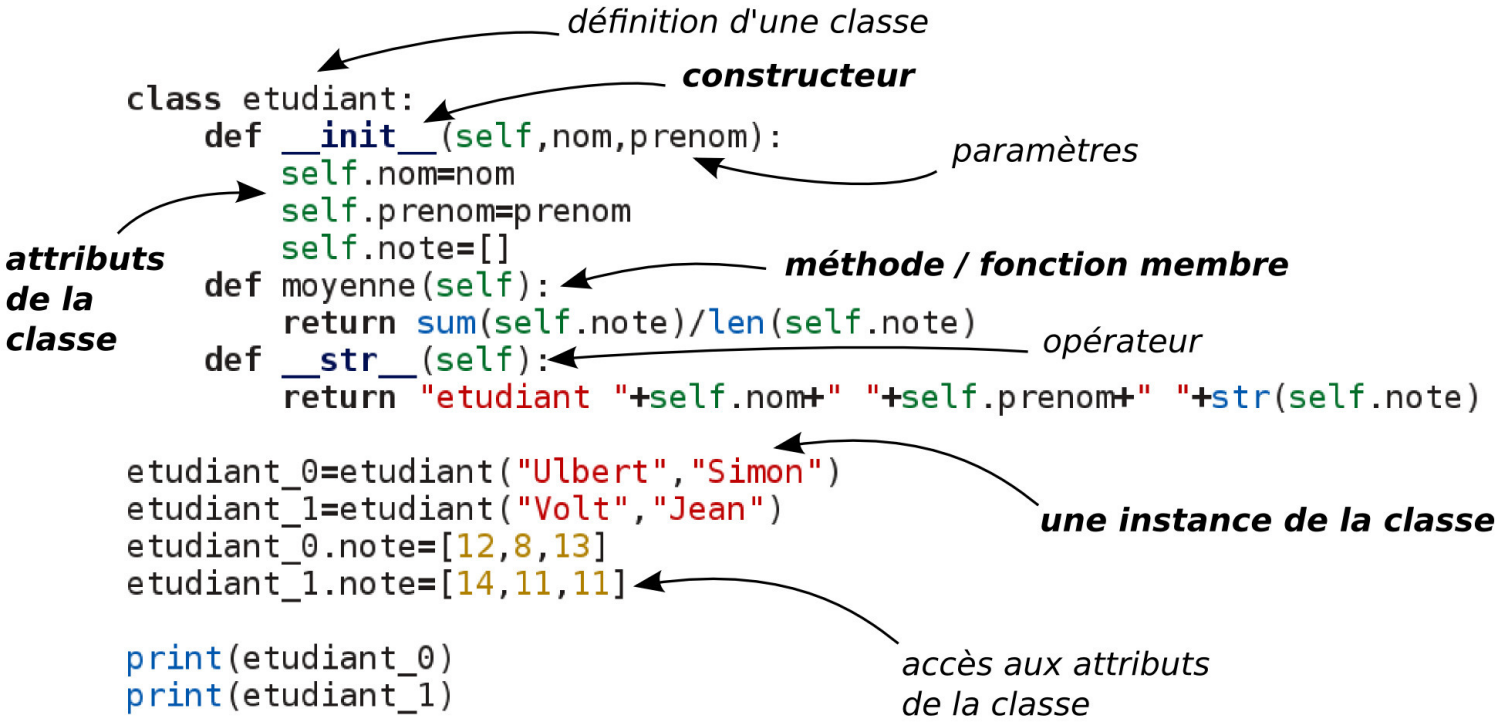
Application polynome

Faire en sorte que la classe polynome de degré 2 puisse

- additionner 2 polynomes (somme des coefficients)
- soustraire 2 polynomes (différence des coefficients)
- multiplier un polynome avec un scalaire
- afficher un polynome avec print()
  On affichera alors: ax^2+bx+c avec les valeurs de (a,b,c)

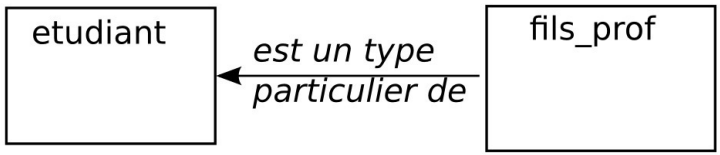
```
class polynome:
    def __init__(self,a,b,c):
        self.a=a
        self.b=b
        self.c=c
    def __add__(self,poly):
        return polynome(self.a+poly.a,self.b+poly.b,self.c+poly.c)
    def __sub__(self,poly):
        return polynome(self.a-poly.a,self.b-poly.b,self.c-poly.c)
    def __mul__(self,s):
        return polynome(s*self.a,s*self.b,s*self.c)
    def __rmul__(self,s):
        return polynome(s*self.a,s*self.b,s*self.c)
    def __str__(self):
        return str(self.a)+"x^2+"+str(self.b)+"x+"+str(self.c)

a=polynome(1,2,3)
b=polynome(1,0,1)

print(a+b)
print(a-b)
print(a*5.2)
```

Vocabulaire



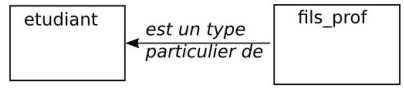
Héritage

```
class etudiant:
    def __init__(self,nom,prenom):
        self.nom=nom
        self.prenom=prenom
        self.note=[]
    def moyenne(self):
        return sum(self.note)/len(self.note)
    def grade(self):
        m=self.moyenne()
        if m>15:
            return "A"
        elif m>12:
            return "B"
        elif m>10:
            return "C"
        else:
            return "E"
    def __str__(self):
        return "etudiant "+self.nom+" "+self.prenom+" "+str(self.moyenne())+"/"+self.grade()

class fils_prof(etudiant):
    def moyenne(self):
        return 18
```



Héritage

```python
class etudiant:
    def __init__(self,nom,prenom):
        self.nom=nom
        self.prenom=prenom
        self.note=[]
    def moyenne(self):
        return sum(self.note)/len(self.note)
    def grade(self):
        m=self.moyenne()
        if m>15:
            return "A"
        elif m>12:
            return "B"
        elif m>10:
            return "C"
        else:
            return "E"
    def __str__(self):
        return "etudiant "+self.nom+" "+self.prenom+" "+str(self.moyenne())+"/"+self.grade()

class fils_prof(etudiant):
    def moyenne(self):
        return 18
```




```python
etudiant_0=etudiant("Ulbert","Simon")
etudiant_1=etudiant("Alu","Joris")
etudiant_2=fils_prof("Volt","Jean")
etudiant_0.note=[12,8,13]
etudiant_1.note=[14,11,11]
etudiant_2.note=[3,6,7]

list_etudiant=[etudiant_0,etudiant_1,etudiant_2]

for e in list_etudiant:
    print(e)
```

Duck Typing

```
class oiseau:
    def __init__(self):
        self.pattes=2
    def parle(self):
        return "cuicui"

class canard(oiseau):
    def parle(self):
        return "coincoin"

class cochon:
    def __init__(self):
        self.pattes=4
    def parle(self):
        return "ronron"

class laurent_gerard:
    def __init__(self):
        self.pattes=2
    def parle(self):
        return "coincoin"
```

```
def communique(animal):
    print("J'ai",animal.pattes,"pattes et je fais",animal.parle())

ferme=[oiseau(),canard(),cochon(),oiseau(),laurent_gerard()]

for animal in ferme:
    communique(animal)
```

Duck Typing="Philosophie Python"

Si j'agit comme un canard,
alors je "suis" un canard

=> Puissance du polymorphisme
Sans nécessiter relation héritage

Application: fraction

Construire une classe pouvant gérer des nombres fractionnaires de manière exacte.
Les attributs de la classes seront le numérateur et le dénominateur.
Ceci seront stockés de manière à ce que la fraction soit sous forme simplifiée.

On pourra par exemple écrire

```
a=fraction(4,8)
b=fraction(4,5)
print(a,"*",b,"=",a*b) #doit afficher  2/5
print(a,"+",b,"=",a+b) #doit afficher 13/10
```

On définira les opérateurs +, -, *

Aide: fonction de pgcd:

```
pgcd(a,b)
Tant que b!=0:
    a=b
    b=a%b
return a
```

Application: fraction

```
def pgcd(a,b):
    while b!=0:
        a,b=b,a%b
    return a

class fraction:
    def __init__(self,num,denum):
        self.num=num
        self.denum=denum
        self.simplify()
    def simplify(self):
        p=pgcd(self.num,self.denum)
        self.num=self.num//p
        self.denum=self.denum//p
    def eval(self):
        return self.num/self.denum
    def __str__(self):
        return str(self.num)+"/"+str(self.denum)
    def __mul__(self,frac):
        return fraction(self.num*frac.num,self.denum*frac.denum)
    def __add__(self,frac):
        return fraction(self.num*frac.denum+self.denum*frac.num,self.denum*frac.denum)
    def __sub__(self,frac):
        return fraction(self.num*frac.denum-self.denum*frac.num,self.denum*frac.denum)
```

Construire une classe pouvant gérer des nombres fractionnaires de manière exacte.
Les attributs de la classes seront le numérateur et le dénominateur.
Ceci seront stockés de manière à ce que la fraction soit sous forme simplifiée.

On pourra par exemple écrire

```
a=fraction(4,8)
b=fraction(4,5)
print(a,"*",b,"=",a*b) #doit afficher  2/5
print(a,"+",b,"=",a+b) #doit afficher 13/10
```

On définira les opérateurs +, -, *

Aide: fonction de pgcd:

```
pgcd(a,b)
Tant que b!=0:
    a=b
    b=a%b
return a
```