```python
# Une version récursive du calcul du PGCD de deux entiers.
def PGCD(a,b):
    if b == 0:
        return a
    else:
        return(PGCD(b,a%b))

class ratio:

    """
    Une classe permettant d'effectuer des manipulations élémentaires sur les
    rationnels.
    Version du 25 septembre 2016.
    """

    # ========================================================================= #
    # CONSTRUCTION                                                              #
    # Les noms des attributs doivent permettre de lire aisément les caluls ! #
    # self.num   : le NUMERATEUR de la fraction.                                #
    # self.denum : le DENOMINATEUR de la fraction                               #
    # ========================================================================= #

    def __init__(self,x,y):
        # Avant toute autre chose, on s'assure que x et y sont bien entiers !
        if type(x) != int or type(y) != int:
            raise TypeError("Au moins un des deux arguments n'est pas entier ! Objet
non créé.")
        # Ensuite, on vérifie que le dénominateur n'est pas nul...
        elif y == 0:
            raise ValueError("Tentative de création d'un rationnel avec un
dénominateur nul ! Objet non créé.")
        else:
            # Simplification par le PGCD
            d = PGCD(abs(x),abs(y))
            x //= d
            y //= d
            # gestion du(des) signe(s) "-"
            if y < 0:
                x, y = -x, -y
            # Définition des attributs
            self.num   = x
            self.denum = y

    # ========================================================================= #
    # REPRESENTATION                                                            #
    # On tient compte des situations particulières...                           #
    # ATTENTION ! La méthode spéciale __repr__ DOIT renvoyer une chaîne de #
    # caractères.                                                               #
    # ========================================================================= #

    def __repr__(self):
        # La fraction est nulle
        if self.num == 0:
            s = str(0)
        # La fraction est non nulle et son dénominateur est égal à 1.
        elif self.denum == 1:
            s = str(self.num)
        # La fraction est non nulle et son dénominateur n'est pas égal à 1.
        else:
            # construction de la chaîne de caractères à afficher
            s = str(self.num) + "/" + str(self.denum)
        return s

    # ========= #
    # INVERSION #
    # ========= #
```

```
    def inversion(self):
        if self.num == 0:
            raise ValueError("0 n'admet pas d'inverse !")
        else:
            return(ratio(self.denum,self.num))

    # ================================================================ #
    # SURCHARGE DE __neg__ POUR LE SUPPORT DE ratio par "-" comme "-r" #
    # ================================================================ #

    def __neg__(self):
        return(ratio(-self.num,self.denum))

    # ================================================================ #
    # SURCHARGE DE __add__ et __radd__ POUR LE SUPPORT DE ratio par "+" #
    # ================================================================ #

    # Méthode spéciale __add__ : surcharge de l'opérateur + afin de pouvoir
    # simplement sommer un rationnel ... et un autre nombre grâce au symbole +.
    # ATTENTION ! Ici, on a traite le cas où le PREMIER TERME de l'addition est
    # un rationnel, le second pouvant être un rationnel ou un entier !

    def __add__(self,r):
        if type(r) != ratio and type(r) != int:
            raise TypeError('L\'addition est impossible !')
        else:
            if type(r) == int:
                r = ratio(r,1)
            return(ratio(self.num * r.denum + self.denum * r.num,self.denum *
r.denum))

    # Méthode spéciale __radd__ : surcharge de l'opérateur + afin de pouvoir
    # simplement sommer un rationnel ... et un autre nombre grâce au symbole +.
    # ATTENTION ! Ici, on a traite le cas où le DEUXIEME TERME de l'addition est
    # un rationnel !

    def __radd__(self,r):
        if type(r) != ratio and type(r) != int:
            return ('L\'addition est impossible !')
        else:
            return(self+r)

    # ================================================================ #
    # SURCHARGE DE __sub__ et __rsub__ POUR LE SUPPORT DE ratio par "-" #
    # ================================================================ #

    # Méthode spéciale __sub__ : surcharge de l'opérateur - afin de pouvoir
    # simplement soustraire un rationnel ... et un autre nombre grâce au symbole
    # -.
    # ATTENTION ! Ici, on a traite le cas où le PREMIER TERME de la soustraction
    # est un rationnel, le second pouvant être un rationnel ou un entier !

    def __sub__(self,r):
        if type(r) != ratio and type(r) != int:
            return ('La soustraction est impossible !')
        else:
            if type(r) == int:
                r = ratio(r,1)
            return(ratio(self.num * r.denum - self.denum * r.num,self.denum *
r.denum))

    # Méthode spéciale __rsub__ : surcharge de l'opérateur - afin de pouvoir
    # simplement soustraire un rationnel ... et un autre nombre grâce au symbole
    # -.
    # ATTENTION ! Ici, on a traite le cas où le DEUXIEME TERME de la soustraction
    # est un rationnel !
```

```
def __rsub__(self,r):
    if type(r) != ratio and type(r) != int:
        return ('La soustraction est impossible !')
    else:
        return((-1)*(self-r))

# ================================================================ #
# SURCHARGE DE __mul__ et __rmul__ POUR LE SUPPORT DE ratio par "*" #
# ================================================================ #

# Méthode spéciale __mul__ : surcharge de l'opérateur * afin de pouvoir
# simplement multiplier un rationnel ... et un autre nombre grâce au symbole
# *.
# ATTENTION ! Ici, on a traite le cas où le PREMIER FACTEUR de la
# multiplication est un rationnel, le second pouvant être un rationnel ou un
# entier !

def __mul__(self,r):
    if type(r) != ratio and type(r) != int:
        return ('La multiplication est impossible !')
    else:
        if type(r) == int:
            r = ratio(r,1)
        return(ratio(self.num * r.num,self.denum * r.denum))

# Méthode spéciale __rmul__ : surcharge de l'opérateur * afin de pouvoir
# simplement soustraire un rationnel ... et un autre nombre grâce au symbole
# *.
# ATTENTION ! Ici, on a traite le cas où le DEUXIEME FACTEUR de la
# multiplication est un rationnel !

def __rmul__(self,r):
    if type(r) != ratio and type(r) != int:
        return ('La multiplication est impossible !')
    else:
        return(self*r)

# ================================================================== #
# SURCHARGE DE __truediv__ et __rtruediv__ POUR LE SUPPORT DE ratio par "/" #
# ================================================================== #

# Méthode spéciale __truediv__ : surcharge de l'opérateur / afin de pouvoir
# simplement diviser un rationnel ... par un autre nombre grâce au symbole
# -.
# ATTENTION ! Ici, on a traite le cas où le DIVIDENDE de la division
# est un rationnel, le second pouvant être un rationnel ou un entier !

def __truediv__(self,r):
    if type(r) != ratio and type(r) != int:
        return ('La soustraction est impossible !')
    else:
        if type(r) == int:
            r = ratio(r,1)
        return self * r.inversion()

# Méthode spéciale __rtruediv__ : surcharge de l'opérateur * afin de pouvoir
# simplement diviser un rationnel ... et un autre nombre grâce au symbole
# -.
# ATTENTION ! Ici, on a traite le cas où le DIVISEUR de la division
# est un rationnel !

def __rtruediv__(self,r):
    if type(r) != ratio and type(r) != int:
        return ('L\'addition est impossible !')
    else:
        return((self/r).inversion())
```