# Nuevas clases: Los números racionales

Usando las ideas que hemos visto hasta ahora, es sencillo crear una clase de números racionales que tenga como atributos el numerador (llamado num en la clase) y el denominador (llamado den en la clase) y que muestre los objetos por pantalla con la notación "numerador/denominador":

In [1]:
```
class Rational(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
    def __str__(self):
        return str(self.num)+'/'+str(self.den)
```

Podemos definir ahora las fracciones r1 y r2 y probar a imprimirlas por pantalla:

In [2]:
```
r1 = Rational(1,2)
r2 = Rational(9,3)
```

In [3]:
```
print r1
```

```
1/2
```

In [4]:
```
r1 + r2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-430dfacd9545> in <module>()
----> 1 r1 + r2

TypeError: unsupported operand type(s) for +: 'Rational' and 'Rational'
```

Una función interesante para los números racionales es la suma. Sin embargo, la notación que hemos visto hasta ahora resulta poco natural para definirla; sería mucho más interesante permitir sumar dos números racionales con el símbolo +, como ya hacemos para los enteros o los reales. Para ello, podemos usar la función **add**:

In [5]:
```
class Rational(object):
    def __init__(self,num,den):
        self.num=num
        self.den=den
    def __str__(self):
        return str(self.num)+'/'+str(self.den)
    def __add__(self,other):
        return Rational(self.num*other.den+other.num*self.den,other.den*self.den)
```

In [6]:
```
r = Rational(3,4)
q = Rational(1,2)
```

In [7]:
```
p = r + q
print p
```

```
10/8
```

Al definir esta función podemos escribir la suma como hemos hecho hasta ahora con el resto de tipos numéricos:

In [8]:
```
r1=Rational(1,2)
r2=Rational(9,3)
print r1+r2
```

```
21/6
```

El método __init__ puede realizar tareas de "normalización" y de verificación. Para verificar, podemos lanzar una excepción cuando el denominador sea 0. Para normalizar, podemos definir previamente la función que calcula el máximo común entre dos números (gcd) y usarla para simplificar la fracción. Además, si el denominador es negativo podemos dejarlo positivo cambiando el signo del numerador.

In [9]:
```
def gcd(a,b):
    """
    Function that calculates the greatest common divisor of two integers.
    @type a: integer
    @type b:integer
    @rtype: integer
    """
    if b==0:
        return a
    else:
        return gcd(b,a%b)

class Rational(object):
    """
    Class to represent rationals: num/den.
    * den > 0
    * num and den do not have common divisors.
    * if num is 0, then den is 1
    """
    def __init__(self,num,den):
        """
        Builds a rational
        @type num: integer
        @type den: integer
        @raise Exception: if den==0
        """
        if den == 0:
            raise Exception('Division by zero when creating a Rational')
        d = gcd(num,den)
        self.num = num/d
        self.den = den/d
        if self.den < 0:
            self.num,self.den = -self.num,-self.den

    def __str__(self):
        return str(self.num)+'/'+str(self.den)
    def __add__(self,other):
        """
        Sum this rational to other
        @type other: Rational
        @rtype: Rational
        """
        num = self.num*other.den + other.num*self.den
        den = other.den * self.den
        return Rational(num,den)
```

De esta manera, al crear el número 9/3 nuestra clase realmente guardará 3/1 y la suma entre 1/2 y 9/3 se evaluará a 7/2 y no a 21/6, como ocurría en el caso anterior.

In [10]:
```
r1=Rational(1,2)
r2=Rational(9,3)
print r1, r2
print r1+r2
```

```
1/2 3/1
7/2
```

In [11]:
```
print r1 + Rational(3,1)
```

```
7/2
```

También podemos ver que se lanza una excepción cuando intentamos crear el número 8/0:

In [12]:
```
print Rational(8,0)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-12-67d8ec7be4b3> in <module>()
----> 1 print Rational(8,0)

<ipython-input-9-2bd915daadd5> in __init__(self, num, den)
     26         """
     27         if den == 0:
---> 28             raise Exception('Division by zero when creating a Rational')
     29         d = gcd(num,den)
     30         self.num = num/d

Exception: Division by zero when creating a Rational
```

Además, la suma puede ser más *lista*: aunque ahora mismo solo podemos sumar un racional con otro racional, pero es interesante sumar nuestros racionales con, por ejemplo, los enteros predefinidos en Python. Para ello, necesitamos distinguir, usando la función type, el tipo del parámetro other: si es un racional (tipo Rational, es decir, el nombre de la clase) seguiremos usando las ideas anteriores, pero si es un entero (tipo int) primero tendremos que transformarlo en un racional con denominador 1. Cuando la operación no está definida se suele devolver el valor especial NotImplemented:

In [13]:
```
type(r1), type(3), type(3.0), type("hola")
```

Out[13]:
```
(__main__.Rational, int, float, str)
```

In [14]:
```
isinstance(3,Rational)
```

Out[14]:
```
False
```

In [15]:
```
class Rational(object):
    def __init__(self,num,den):
        """
        Class to represent rationals: num/den.
        * den > 0
        * num and den do not have common divisors.
        * if num is 0, then den is 1
        """
        if den==0:
            raise Exception('Division by zero when creating a Rational')
        d=gcd(num,den)
        self.num=num/d
        self.den=den/d
        if self.den<0:
            self.num,self.den=-self.num,-self.den

    def __str__(self):
        return str(self.num)+'/'+str(self.den)
    def __add__(self, other):
        """
        Sum this rational to other
        @type other: Rational
        @rtype: Rational or integer
        """
        if type(other) == Rational:
            num = self.num * other.den + other.num * self.den
            den = other.den * self.den
            return Rational(num,den)
        elif type(other) == int:
            other = Rational(other,1)
            return self + other
        else:
            return NotImplemented
```

De esta manera, la misma función nos sirve en ambos casos:

In [16]:
```
r1=Rational(1,2)
r2=Rational(9,3)
print r1 + r2
print r1 + 2
```

```
7/2
5/2
```

Sin embargo, recuerda que es el primer argumento el que "dirige" la función. Si el primer argumento no es un elemento de la clase Rational Python no sabrá qué hacer con la suma, ya que intentará buscarlo en las funciones para números enteros.

In [17]:
```
print 1 + r1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-3a0dd4469299> in <module>()
----> 1 print 1 + r1

TypeError: unsupported operand type(s) for +: 'int' and 'Rational'
```

De igual forma, fallará si intentamos sumar un real a un racional:

In [18]:
```
r1 + 3.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-521eac3e0b61> in <module>()
----> 1 r1 + 3.

TypeError: unsupported operand type(s) for +: 'Rational' and 'float'
```

Muchas otras funciones pueden definirse de esta manera: la resta (**sub**), la multiplicacion (**mul**) devolverán nuevos racionales, la comparación (**cmp**) devuelve -1 si *self* es menor que *other*, 0 si son iguales y 1 si *other* es mayor que *self*. También podemos definir los operadores booleanos, por ejemplo < (**lt**), para lo que es conveniente tener definida la función de comparación:

In [19]:

```
class Rational(object):
    def __init__(self,num,den):
        """
        Class to represent rationals: num/den.
        * den > 0
        * num and den do not have common divisors.
        * if num is 0, then den is 1
        """
        if den==0:
            raise Exception('Division by zero when creating a Rational')
        d=gcd(num,den)
        self.num=num/d
        self.den=den/d
        if self.den<0:
            self.num,self.den=-self.num,-self.den

    def __str__(self):
        return str(self.num)+'/'+str(self.den)


    def __add__(self, r):
        """
        Sum this rational to r
        @type other: Rational
        @rtype: Rational or integer
        """
        if type(r) == int:
            r = Rational(r,1)
            return self+r
        elif type(r) == Rational:
            num = self.num*r.den + r.num*self.den
            den = r.den * self.den
            return Rational(num,den)
        else:
            raise Exception('these types cannot be added')

    def __sub__(self, r):
        """
        Substracts r to this rational
        @type other: Rational
        @rtype: Rational or integer
        """
        if type(r) == int:
            r = Rational(r,1)
            return self-r
        elif type(r) == Rational:
            num = self.num*r.den - r.num*self.den
            den = r.den * self.den
            return Rational(num,den)
        else:
            raise Exception('these types cannot be subtracted')
```

In [20]:
```
    def __cmp__(self, other):
        """
        Method to compare two numbers. other can be an integer
        or a rational.
        @type other: Rational or integer
        @rtype: int
        @return: 0 if equals
                 <0 if other is bigger than self
                 >0 if self is bigger that other
        """
        if isinstance(other, Rational):
            return (self-other).num
        elif isinstance(other, int):
            diff = self - Rational(other,1)
            return diff.num
        else:
            return NotImplemented


    def __lt__(self, other):
        """
        Method to implement the less than comparation operator.
        @type other: Rational or integer
        @rtype: boolean
        @retrurn: True if self is lower than other
                  False otherwise
        """
        return self.__cmp__(other) < 0
```

```
  File "<ipython-input-20-9f1e4272d697>", line 2
    def __cmp__(self, other):
    ^
IndentationError: unexpected indent
```

De esta forma podemos realizar estas operaciones de una manera más clara:

In [21]:
```
r=Rational(3,4)
s=Rational(7,8)
```

In [22]:
```
print r-s,r<s,cmp(r,s)
```

```
-1/8 False 1
```

Otros operadores que podemos definir:

- *:con el método __mult__
- /:con el método __div__
- %: con el método __mod__
- ==: con el método __eq__
- <=: con el método __le__
- !=: con el método __ne__
- >: con el método __gt__
- >=: con el método __ge__

Siempre es interesante un método para construir objetos a partir de un string. Este es el trabajo de la función parse, que construye

objetos de la clase en la que se encuentra. Por ello, no necesitamos un Rational para aplicarla y no recibe el parámetro self. Este tipo de funciones recibe el nombre de *estáticas*:

In [23]:
```
class Rational(object):
    def __init__(self,num,den):
        """
        Class to represent rationals: num/den.
        * den > 0
        * num and den do not have common divisors.
        * if num is 0, then den is 1
        """
        if den==0:
            raise Exception('Division by zero when creating a Rational')
        d=gcd(num,den)
        self.num=num/d
        self.den=den/d
        if self.den<0:
            self.num,self.den=-self.num,-self.den

    def __str__(self):
        return str(self.num)+'/'+str(self.den)


    def __add__(self, r):
        """
        Sum this rational to r
        @type other: Rational
        @rtype: Rational or integer
        """
        if type(r) == int:
            r = Rational(r,1)
            return self+r
        elif type(r) == Rational:
            num = self.num*r.den + r.num*self.den
            den = r.den * self.den
            return Rational(num,den)
        else:
            return NotImplemented

    def __sub__(self, r):
        """
        Substracts r to this rational
        @type other: Rational
        @rtype: Rational or integer
        """
        if type(r) == int:
            r = Rational(r,1)
            return self-r
        elif type(r) == Rational:
            num = self.num*r.den - r.num*self.den
            den = r.den * self.den
            return Rational(num,den)
        else:
            return NotImplemented
```

In [24]:

```
    def __cmp__(self, other):
        """
        Method to compare two numbers. other can be an integer
        or a rational.
        @type other: Rational or integer
        @rtype: int
        @return: 0 if equals
                  <0 if other is bigger than self
                  >0 if self is bigger that other
        """
        if isinstance(other, Rational):
            return (self-other).num
        elif isinstance(other, int):
            diff = self - Rational(other,1)
            return diff.num
        else:
            return NotImplemented


    def __lt__(self, other):
        """
        Method to implement the less than comparation operator.
        @type other: Rational or integer
        @rtype: boolean
        @retrurn: True if self is lower than other
                  False otherwise
        """
        return self.__cmp__(other) < 0

    @staticmethod
    def parse(s):
        """
        This static function builds a rational from a string. The
        rational can have the folowing forms:
          - <integer>
          - <integer>/<integer>
        @raise Exception: This function can raise an exception
           - There is no number at all
           - if the second integer is zero
        @type s: string
        @rtype rational
        """
        l = s.strip().split("/")
        num = int(l[0].strip())
        if len(l)>1:
            den = int(l[1].strip())
        else:
            den = 1
        return Rational(num, den)
```

```
  File "<ipython-input-24-eb01567142f1>", line 3
    def __cmp__(self, other):
    ^
IndentationError: unexpected indent
```

Las funciones estáticas se ejecutan poniendo el nombre de la clase seguida de un punto y el nombre de la función con los correspondientes argumentos:

In [25]:
```
r=Rational.parse('3/4')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-d08d7333cb0a> in <module>()
----> 1 r=Rational.parse('3/4')

AttributeError: type object 'Rational' has no attribute 'parse'
```

In [26]:
```
print r
```

```
3/4
```

In [26]:
```

```

In [26]:
```

```