Fundamentos de Programação
@ LEIC/LETI

Semana 9

Abstração de dados

Abstração em programação. Números complexos. Essência da abstração de dados. Complexos como dicionários. Tipos abstratos de dados.

Alberto Abad, Tagus Park, IST, 2018

Abstração em programação

- A abstração é um conceito central em programação (e não só):
  - Descrição simplificada de uma entidade com foco nas propriedades mais relevantes, deixando de parte (escondendo) os pormenores.
- Até agora, vimos **abstração procedimental** para definir funções:
  - Em uma função definimos um *nome*, entradas e saídas, assim escondemos os pormenores de implementação ao utilizador/resto do programa → Separação do **que** e do **como**
  - Permite substituir funções por outras que fazem o mesmo, de uma forma diferente.
- Os programas podem ser considerados como um conjunto de construções abstractas que podem ser executadas por um computador.

Abstração em programação

- Até agora, utilizamos instâncias de tipos já existentes:
  - Nunca considerámos novos tipos de dados não *built-in*.
  - Mas utilizamos abstrações já existentes, por exemplo as listas.
- Desejamos representar e utilizar diferentes tipos de informação nos nossos programas e que não existem na linguagem.
- Esta semana, veremos como definir tipos estruturados de informação *custom* recurrendo ao conceito de **abstração de dados**:
  - Equivalente às abstracções procedimentais mas para estruturas de dados.
  - Permite separar o modo como pode ser utilizada, e o que representa (o **que**), da forma como é construída e representada a partir de outros tipos e estruturas de (o **como**).

Definição de novos tipos e abstração

- Um tipo de informação é em geral caracterizado pelo conjunto de operações que suporta e pelo conjunto de instâncias ou entidades associadas:
  - O conjunto de instâncias denomina-se domínio do tipo.
  - Cada instância no conjunto denomina-se elemento do tipo.
- A abstração de dados consiste em considerar a definição de novos tipos de informação em duas fases sequenciais:
  - **1.** Estudo das propriedades do tipo.
  - **2.** Pormenores da realização do tipo numa linguagem de programação.
- Vejamos com um exemplo a importância de esta sequência: **números complexos**.

Exemplo motivador: Números complexos

- Um número complexo é um número que pode ser expressado da forma $a + bi$, em que tanto a, a parte real, como b, a parte imaginária, são números reais, e o símbolo i satisfaz a equaçãao $i^2 = -1$ (chamada unidade imaginária porque nenhum número satisfaz esta equação).

- A soma, subtração, multiplicação e divisão de números complexos são definidas do seguinte modo:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$$

$$\frac{a + bi}{c + di} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

</br>

Números complexos - Primeira abordagem: Solução dependente da representação

- **Sequência errada:** Desenvolver para uma representação concreta. Por exemplo, tuplos.

In [61]:
```
def sum_compl(c1, c2):
    c3 = (c1[0] + c2[0], c1[1] + c2[1])
    return c3

def sub_compl(c1, c2):
    c3 = (c1[0] - c2[0], c1[1] - c2[1])
    return c3

def mul_compl(c1, c2):
    c3 = (c1[0] * c2[0] - c1[1] * c2[1], c1[0] * c2[1] + c1[1] * c2[0] )
    return c3

def div_compl(c1, c2):
    den = c2[0]*c2[0] + c2[1]*c2[1]
    c3 = ((c1[0] * c2[0] + c1[1] * c2[1])/den , (c1[0] * c2[1] - c1[1] * c2[0])/de
n)
    return c3

mul_compl((1,2),(2,3))
```

Out[61]:
```
(-4, 7)
```

- Qual é o problema com esta solução?

Números complexos - Segunda abordagem: Solução independente da representação

- Imaginemos que existe um módulo/biblioteca com as seguintes funções:
  - **cria_compl(r, i)** - recebe como argumentos dois números reais e retorna um número complexo.
  - **p_real(c)** - recebe como argumento um número complexo e rertona a parte real.
  - **p_imag(c)** - recebe como argumento um número complexo e rertona a parte imaginária.
- Podemos escrever uma solução que utilize estas funções **independentemente** da representação.

Números complexos - Segunda abordagem: Solução independente da representação

In [69]:
```
def sum_compl(c1, c2):
    p_r = p_real(c1) + p_real(c2)
    p_i = p_imag(c1) + p_imag(c2)
    return cria_compl(p_r, p_i)

def sub_compl(c1, c2):
    p_r = p_real(c1) - p_real(c2)
    p_i = p_imag(c1) - p_imag(c2)
    return cria_compl(p_r, p_i)

def mul_compl(c1, c2):
    p_r = p_real(c1) * p_real(c2) - p_img(c1) * p_img(c2)
    p_i = p_real(c1) * p_img(c2) + p_img(c1) * p_real(c2)

    return cria_compl(p_r, p_i)

def div_compl(c1, c2):
    den = p_real(c2) * p_real(c2) + p_img(c2) * p_img(c2)
    p_r = (p_real(c1) * p_real(c2) + p_img(c1) * p_img(c2))/den
    p_i = (p_real(c1) * p_img(c2) - p_real(c1) * p_img(c2))/den
    return cria_compl(p_r, p_i)
```

Números complexos - Segunda abordagem: Solução independente da representação

- Baseada em esta *biblioteca* podemos definir novas funções, por exemplo de *representação externa*:

In [31]:
```
def compl_para_string(c):
    if p_imag(c) >= 0:
        rep_ext = str(p_real(c)) + '+' + str(p_imag(c)) + 'i'
    else:
        rep_ext = str(p_real(c)) + '-' + str(abs(p_imag(c))) + 'i'
    return rep_ext
```

Números complexos - Segunda abordagem: Solução independente da representação

- Podemos repesentar os nossos complexos como **tuplos**: $R\{a + bi\}$ = (a, b)

In [70]:
```
#Representing as a tuple
def cria_compl(r, i):
    return (r, i)

def p_real(c):
    return c[0]

def p_imag(c):
    return c[1]

c1 = cria_compl(10, 5)
c2 = cria_compl(3, 10)
print(compl_para_string(sub_compl(c1, c2)))
type(c1)
```

```
7-5i
```

Out[70]:
```
tuple
```

Números complexos - Segunda abordagem: Solução independente da representação

- Ou podemos repesentar os noss complexos como **dicionários**: $R\{a + bi\}$ = {'r':a, 'i':b}

In [73]:
```
#Representing as a dictionary
def cria_compl(r, i):
    return {'r':r, 'i':i}

def p_real(c):
    return c['r']

def p_imag(c):
    return c['i']


c1 = cria_compl(10, 5)
c2 = cria_compl(3, 10)
print(escreve_compl(sum_compl(c1, c2)))
c2['r']
```

```
13+15i
```

Out[73]:
```
3
```

Outro Exemplo: Vectores

- Consideremos um tipo de dados abstracto para representar vectores num espaço bidimensional.

- Operações a suportar:

  - cria_vector(x, y): dados dois número reais x e y retorna o vector (x, y)
  - vector_abcissa(v): dado um vector v retorna a abcissa
  - vector_ordenada(v): dado um vector v retorna a ordenada
  - e_vector(e): dado um qualquer elemento e reconhece se o mesmo é um vector ou não
  - vector_igual(u,v): dados dois vectores indica se os mesmos são ou não iguais
  - vector_para_string(v), dado um vector v retorna um string que o representa.

Outro Exemplo: Vectores

In [79]:
```
# constructor
def cria_vector(x, y):
    if isinstance(x,(int, float)) and isinstance(y,(int, float)):
        return (x,y)
    raise ValueError("cria_vector: argumentos invalidos")

cria_vector(10,20)
```

Out[79]: (10, 20)

In [80]:
```
#selector
def vector_abscissa(v):
    if e_vector(v):
        return v[0]
    raise ValueError("vector_abscissa: argumento nao vector")
```

In [81]:
```
#selector
def vector_ordenada(v):
    if e_vector(v):
        return v[1]
    raise ValueError("vector_ordenada: argumento nao vector")
```

Outro Exemplo: Vectores

In [78]:
```
#reconhecedor
def e_vector(v):
    return isinstance(v, tuple) and \
           len(v) == 2 and \
           isinstance(v[0], (int, float)) and \
           isinstance(v[1], (int, float))
```

In [77]:
```
#teste
def vector_igual(u,v):
    if e_vector(u) and e_vector(v):
        return vector_abscissa(u) == vector_abscissa(v) \
               and vector_ordenada(u) == vector_ordenada(v)
```

In [76]:
```
#transformador
def vector_para_string(v):
    return "<" + str(vector_abscissa(v)) + "," + str(vector_ordenada(v)) + ">"
```

Outro Exemplo: Vectores

Produto escalar (*dot product*)

$$\boldsymbol{u} \cdot \boldsymbol{v} = (u_1, u_2) \cdot (v_1, v_2) = u_1 \cdot v_1 + u_2 \cdot v_2$$

In [84]:

```
# u . v = (1,2) x (4,5) = 1x4 + 5x2 = 14
def produto_escalar(u, v):
    if e_vector(u) and e_vector(v):
        return vector_abscissa(u)*vector_abscissa(v) + vector_ordenada(u)*vector_o
rdenada(v)
    raise ValueError("produto_escalar: parametros nao vectores")

u = cria_vector(1,2)
v = cria_vector(4,5)
print(vector_para_string(u))
print(vector_para_string(v))
print(produto_escalar(u, v))
```

```
<1,2>
<4,5>
14
```

Tipos abstractos de dados (TAD)

- Um **tipo de dados abstracto (TAD)** ou *abstract data type* (ADT) é caracterizado pelo conjunto de operações que suporta e pelo conjunto de instâncias ou entidades associadas (domínio).

- Um TAD é um mecanismo de encapsulamento composto por:

  - Uma estrutura ou estruturas de dados
  - Um conjunto de operações básicas.
  - Uma descrição precisa dos tipos das operaçoes (chamados assinatura).
  - Um conjunto preciso de regras sobre como ele se comporta (chamado descrição axiomática).
  - Uma implementação oculta do cliente/programador.

- Nesta aula:

  - Metodologia para criar novos TADs
  - Barreiras de abstracção.

Metodologia dos tipos abstractos de dados

- **Objetivo:** Separar o modo como os elementos de um tipo são utilizados do modo como esses elementos são representados e como as operações sobre os mesmos são implementadas.

- Passos a seguir:

  1. Identificação das operações básicas;

  2. Axiomatização das operações básicas;

  3. Escolha de uma representação (interna) para os elementos do tipo;

  4. Concretização das operações básicas.

Metodologia dos TAD: 1. Operações básicas

- Conjunto mínimo de operações que caracterizam o tipo. Também conhecido como **assinatura do tipo**.

- Dividem-se em seis grupos (podem não existir todas para um tipo específico):

  - construtores: permitem construir novos elementos do tipo;
  - selectores: permitem aceder às propriedades e partes dos elementos do tipo;
  - modificadores: permitem modificar os elementos do tipo;
  - transformadores: permitem transformar elementos do tipo em outro tipo;
  - reconhecedores: permitem reconhecer elementos como sendo do tipo oudistinguir elementos do tipo com características particulares;
  - testes: permitem efectuar comparações entre elementos do tipo.

- A definição de um TAD é independente da linguagem de programação e em geral é utilizada notação matemática.

Metodologia dos TAD: 1. Operações básicas

Exemplo de definição do tipo (imutável) complexo

Construtores:

```
cria_complexo : real x real --> complexo
cria_complexo(x, y) tem como valor o número complexo (x + y i).

cria_complexo_zero : {} --> complexo
cria_complexo_zero() tem como valor o número complexo (0 + 0 i)
```

Selectores:

```
complexo_parte_real : complexo --> real
complexo_parte_real(z) tem como valor a parte real de z.

complexo_parte_imaginaria : complexo --> real
complexo_parte_imaginaria(z) tem como valor a parte imaginária de z
```

Metodologia dos TAD: 1. Operações básicas

Exemplo de definição do tipo (imutável) complexo

Reconhecedores:

```
e_complexo : universal --> lógico
e_complexo(u) tem valor verdadeiro se e só se u é um número complexo.

e_complexo_zero : complexo --> lógico
e_complexo_zero(z) tem como valor verdadeiro se a parte real e a parte imaginária
 são ambas 0.

e_imaginario_puro : complexo --> lógico
e_imaginario_puro(z) tem como valor verdadeiro se z tem parte real 0 e parte imag
inária diferente de 0
```

Testes:

```
complexo_igual : complexo x complexo --> lógico
complexo_igual(z, w) tem valor verdaeiro se z e w corresponderem ao mesmo número
 complexo
```

Transformadores:

```
complexo_para_string: complexo --> string
complexo_para_string(z) tem como valor a string com a representação externa de z
 na forma 'x + y i'.
```

(Notar transformadores de saída e de entrada)

Metodologia dos TAD: 1. Operações básicas

Exemplo de assinatura do tipo (imutável) complexo

```
cria_complexo : real x real --> complexo
cria_complexo_zero : {} --> complexo
complexo_parte_real : complexo --> real
complexo_parte_imaginaria : complexo --> real
e_complexo : universal --> lógico
e_complexo_zero : complexo --> lógico
e_imaginario_puro : complexo --> lógico
complexo_igual : complexo x complexo --> lógico
complexo_para_string: complexo --> string
```

Metodologia dos TAD: 2. Axiomatização

- Conjunto de expressões lógicas (axiomas) que têm de ser verdadeiras para qualquer realização/implementação do tipo.
- Axiomatização do tipo complexo:
  - e_complexo(cria_complexo(x, y))
  - e_complexo(cria_complexo_zero())
  - e_complexo_zero(cria_complexo_zero())
  - complexo_igual(cria_complexo_zero(), cria_complexo(0, 0))
  - e_imaginario_puro(cria_complexo(0, y)), para qualquer y != 0.
  - complexo_parte_real(cria_complexo(x, y)) = x, para quaisquer x e y.
  - complexo_parte_imaginaria(cria_complexo(x, y)) = y, para quaisquer x e y.
  - complexo_igual(cria_complexo(x, y), cria_complexo(x, y)), para quaisquer x e y.
  - complexo_igual(z, cria_complexo(complexo_parte_real(z),complexo_parte_imaginaria(z))), se e_complexo(z), indefinido caso contrário.

Metodologia dos TAD: 3. Representação interna

- Escolher uma representação interna para os elementos do tipo, tendo por base outros tipos existentes ou já definidos.
- Ter em conta aspectos de eficiência relativos à realização das operações básicas.
- Como exemplo, no caso dos números complexos e uma implementação em Python, podemos utilizar um dicionário com duas chaves, real e imaginario.

Metodologia dos TAD: 4. Realização/implementação das operações básicas

- Realização/implementação das operações básicas, tendo em conta os passo anteriores: a **assinatura**, a **axiomatização** e a **representação interna**.

Construtores

In [85]:
```
def cria_complexo(x, y):
    if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
        raise ValueError('cria_complexo: argumentos invalidos, x e y tem de ser nu
meros')
    return {'real' : x, 'imaginario' : y}

def cria_complexo_zero():
    return cria_complexo(0, 0)
```

Metodologia dos TAD: 4. Realização/implementação das operações básicas

- Realização/implementação das operações básicas, tendo em conta os passo anteriores: a **assinatura**, a **axiomatização** e a **representação interna**.

Seletores

In [86]:
```
def complexo_parte_real(z):
    if not e_complexo(z):
        raise ValueError('complexo_parte_real: z tem de ser um complexo')
    return z['real']

def complexo_parte_imaginaria(z):
    if not e_complexo(z):
        raise ValueError('complexo_parte_imaginaria: z tem de ser um complexo')
    return z['imaginario']
```

Metodologia dos TAD: 4. Realização/implementação das operações básicas

- Realização/implementação das operações básicas, tendo em conta os passo anteriores: a **assinatura**, a **axiomatização** e a **representação interna**.

Reconhecedores

In [87]:

```
def e_complexo(x):
    if isinstance(x, (dict)):
        if len(x) == 2 and 'real' in x and 'imaginario' in x:
            return isinstance(x['real'] , (int, float)) \
                and isinstance(x['imaginario'] , (int, float))
    return False

def e_complexo_zero(z):
    if not e_complexo(z):
        raise ValueError('complexo_parte_imaginaria: z tem de ser um complexo')
    return zero(complexo_parte_real(z)) and zero(complexo_parte_imaginaria(z))

def e_imaginario_puro(z):
    if not e_complexo(z):
        raise ValueError('complexo_parte_imaginaria: z tem de ser um complexo')
    return zero(complexo_parte_real(z)) and not zero(complexo_parte_imaginaria(z))

def zero(x):
    return abs(x) < 0.0000001
```

Metodologia dos TAD: 4. Realização/implementação das operações básicas

- Realização/implementação das operações básicas, tendo em conta os passo anteriores: a **assinatura**, a **axiomatização** e a **representação interna**.

Testes

In [88]:
```
def complexo_igual(z, w):
    if not(e_complexo(z) and e_complexo(w)):
        raise ValueError('complexo_parte_imaginaria: z e w tem de ser complexos')
    return zero(complexo_parte_real(z) - complexo_parte_real(w)) \
           and zero(complexo_parte_imaginaria(z) - complexo_parte_imaginaria(w))
```

Metodologia dos TAD: 4. Realização/implementação das operações básicas

- Realização/implementação das operações básicas, tendo em conta os passo anteriores: a **assinatura**, a **axiomatização** e a **representação interna**.

Tranformadores

In [89]:
```
def complexo_para_string(z):
    if not e_complexo(z):
        raise ValueError('complexo_parte_imaginaria: z tem de ser um complexo')
    return str(complexo_parte_real(z)) + '+' + str(complexo_parte_imaginaria(z)) + 
 'i'
```

Metodologia dos TAD: Exemplos de utilização

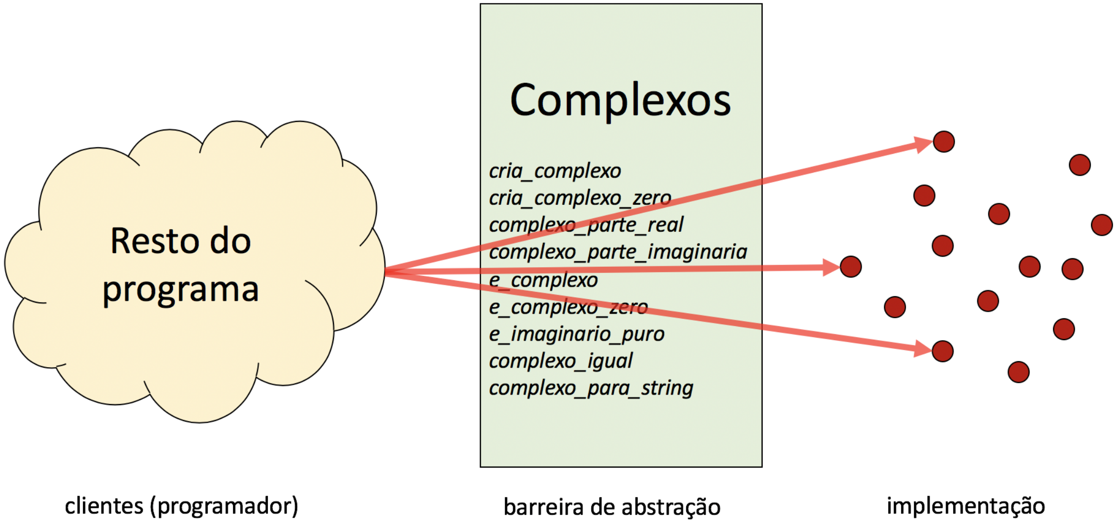
In [102]:
```
print(e_complexo(cria_complexo(1,10)))

a = cria_complexo(1,10)
print(complexo_parte_real(a) == 1)
print(complexo_para_string(a))



# cliente!!!!
def subtracao_complexo(a,b):
    if e_complexo(a) and e_complexo(b):
        p_r = complexo_parte_real(a) - complexo_parte_real(b)
        p_i = complexo_parte_imaginaria(a) - complexo_parte_imaginaria(b)
        return cria_complexo(p_r, p_i)
    raise ValueError()

b = cria_complexo(2,2)
complexo_para_string(subtracao_complexo(a,b))
```

```
True
True
1+10i
```

Out[102]:
```
'-1+8i'
```

Barreiras de abstração

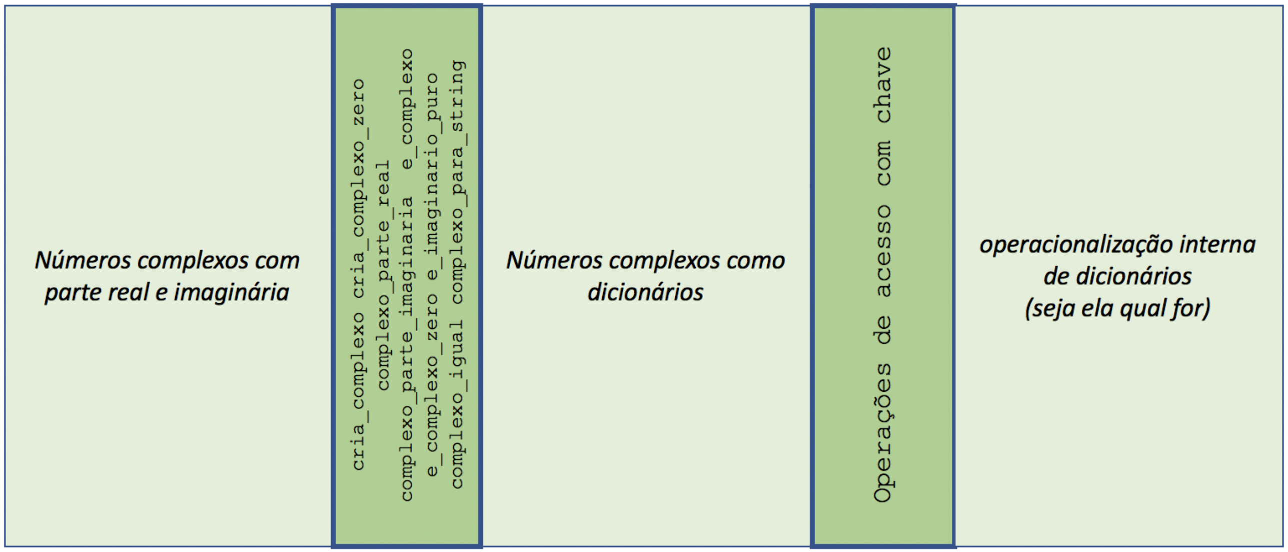
- A definição dum TAD implica a definição de uma barreira entre os programas (ou partes do programa) que utilizam a abstracção de dados e os programas (ou partes do programa) que realizam/implementam a abstracção de dados.
- Esta barreira denomina-se por **barreira de abstracção**.



- Podemos considerar os TADs como tipos built-in:
  - Implementação escondida (ainda que não é bem)
  - Manipulação realizada através das operações básicas.

Barreiras de abstração

- A violação das barreiras de abstracção (utilização das representações internas por partes do programa que não na implementação das operações básicas) corresponde a uma má prática de programação:
  - Programas dependentes de representação
  - Programas menos compreensível e de difícil escrita.



Exercício - Racionais

Um número racional é qualquer número que possa ser expresso como o quociente de dois inteiros: o numerador (um inteiro positivo, negativo ou nulo) e o denominador (um inteiro positivo). Os racionais a/b e c/d são iguais se e só se $a * d = b * c$.

- Especificar operações básicas
- Escolher uma representação
- Escrever operações básicas
- Escrever funções soma e produto (respeitando barreiras de abstração)

In [108]:

```
def cria_racional(a, b):
    if isinstance(a,int) and isinstance(b, int) and b >0:
        return {'num': a, 'den': b}
    raise ValueError()

def numerador(r):
    if e_racional(r):
        return r['num']
    raise ValueError()


def denominador(r):
    if e_racional(r):
        return r['den']
    raise ValueError()

def e_racional(u):
    return isinstance(u, dict) and len(u) == 2 \
            and 'num' in u and 'den' in u \
            and isinstance(u['num'], int) \
            and isinstance(u['den'], int) and u['den'] > 0

def e_racional_zero(u):
    if e_racional(r):
        return numerador(r) == 0
    raise ValueError("")


def racional_iguais(r1, r2):
    if e_racional(r1) and e_racional(r2):
        return numerador(r1) * denominador(r2) == numerador(r2) * denominador(r1)

    raise ValueError("")

def racional_para_string(r):
    if e_racional(r):
```

```python
        return str(numerador(r)) + "/" + str(denominador(r))
    raise ValueError("")
```

In [112]:

```python
a = cria_racional(1,2)
b = cria_racional(2,5)
print(racional_para_string(a))
print(racional_para_string(b))
racional_iguais(a,b)

def simetrico(r):
    if e_racional(r):
        return cria_racional(-numerador(r), denominador(r))
    raise ValueError("")



def soma(r1, r2):
    pass

def produto(r1, r2):
    pass

print(racional_para_string(simetrico(a)))
```

```
1/2
2/5
-1/2
```