

Fundamentos da Programação

Metodologia dos Tipos Abstratos de Dados

Aula 16

Alberto Abad, Tagus Park, IST, 2021-22

Tipos Abstratos de Dados (TAD)

- Um **tipo de dados abstrato (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 operações (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 abstração

Metodologia dos Tipos Abstratos 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: 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;
 - seletores: 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 ou distinguir 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: 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)
```

Seletores:

```
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: 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 imaginária diferente de 0

Testes:

`complexo_igual : complexo x complexo --> lógico`

`complexo_igual(z, w)` tem valor verdadeiro 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: 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: 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 $y \neq 0$.
 - $complexo_parte_real(cria_complexo(x, y)) = x$, 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: Representação Interna

- Escolher uma representação interna para os elementos do tipo, tendo por base outros tipos existentes ou já definidos.
- Ter em conta aspetos 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 imaginário.

Metodologia dos TAD: 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 [48]: def cria_complexo(x, y):
          if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
              raise ValueError('cria_complexo: argumentos inválidos, x e y tem de ser números')
          return {'real' : x, 'imaginário' : y}

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

Metodologia dos TAD: 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 [49]: 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: Implementação das Operações Básicas

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

Reconhecedores

```
In [50]: 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: 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 [51]: 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: 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 [52]: 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)) + ('+' if complexo_parte_imaginaria(z) > 0 else '-') + str(abs(complexo_parte_imaginaria(z))) + 'i'
```

Metodologia dos TAD: Exemplos de Utilização


```
In [53]: 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()

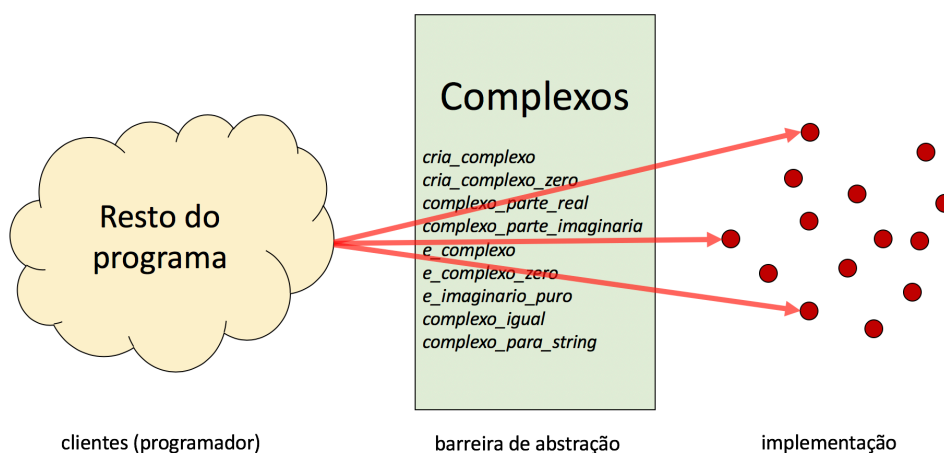
b = cria_complexo(2,2)
complexo_para_string(subtracao_complexo(a,b))
```

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

```
Out[53]: '-1+8i'
```

Barreiras de Abstração

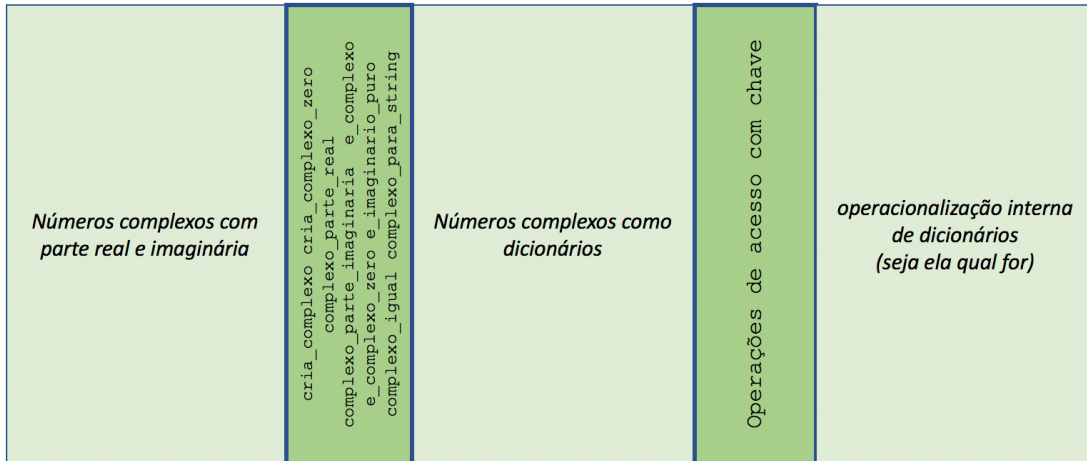
- A definição dum TAD implica a definição de uma barreira entre os programas (ou partes do programa) que utilizam a abstração de dados e os programas (ou partes do programa) que realizam/implementam a abstração de dados.
- Esta barreira denomina-se por **barreira de abstração**.



- 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 da representação
 - Programas menos compreensível e de difícil escrita.



Abstração de dados

Exercício - Racionais (I)

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 $a \times d = b \times c$.

- Especificar operações básicas
- Escolher uma representação
- **Escrever operações básicas**
- Escrever funções simétrico, soma e produto (respeitando barreiras de abstracção)

```

In [57]: # construtor
def cria_racional(a, b):
    if isinstance(a, int) and isinstance(b, int) and b != 0:
        return (a, b)
    raise ValueError('')

# seletores
def numerador(r):
    if e_racional(r):
        return r[0]
    raise ValueError('')

def denominador(r):
    if e_racional(r):
        return r[1]
    raise ValueError('')

# reconhecedores
def e_racional(u):
    return isinstance(u, tuple) and len(u) == 2 and isinstance(u[0]
, int) and \
        isinstance(u[1], int) and u[1] != 0

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

def e_inteiro(u):
    if e_racional(r):
        return numerador(r) % denominador(r) == 0
    raise ValueError('')

# testes
def racional_iguais(r1, r2):
    return numerador(r1) * denominador(r2) == numerador(r2) * denom
inador(r1)

# transformador de saída
def racional_para_string(r):
    return str(numerador(r)) + '/' + str(denominador(r))

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

```

```

1/2
2/5

```

Out[57]: False

Abstração de dados

Exercício - Racionais (II)

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 $a \times d = b \times c$.

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

```
In [60]: def simetrico(r):
          if e_racional(r):
              return cria_racional(-numerador(r), denominador(r))
          raise ValueError("")

          def soma(r1, r2):
              if e_racional(r1) and e_racional(r2):
                  n = numerador(r1)* denominador(r2) + numerador(r2)* denominador(r1)
                  d = denominador(r1) * denominador(r2)
                  return cria_racional(n,d)
              raise ValueError("")

          def produto(r1, r2):
              if e_racional(r1) and e_racional(r2):
                  n = numerador(r1)* numerador(r2)
                  d = denominador(r1) * denominador(r2)
                  return cria_racional(n,d)
              raise ValueError("")

          print(racional_para_string(simetrico(a)))
          print(racional_para_string(soma(a, b)))
          print(racional_para_string(produto(a, b)))
```

-1/2

9/10

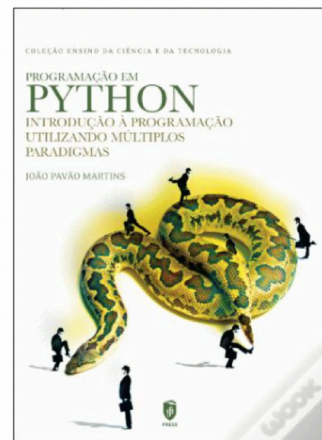
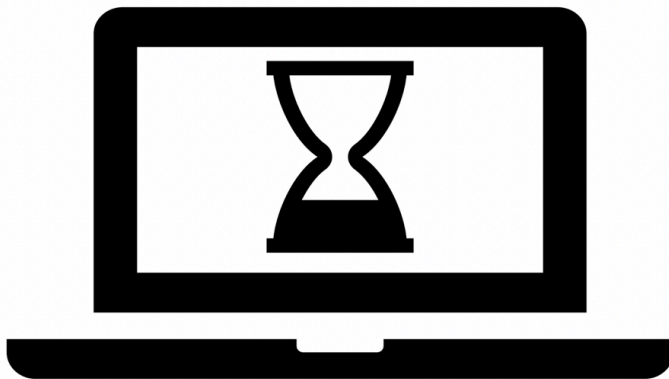
2/10

ATENÇÃO: Esta parte da matéria é central no Projeto 2 de FP!!!

Abstração de dados

Tarefas próximas aulas

- Estudar matéria Abstração e Tipos Abstratos de Dados:
 - Completar exemplos
- Nas aulas teóricas --> Ficheiros (capítulo 10 do livro)
- Na aula práticas --> TADs



In []: