

Fundamentos da Programação

Abstração de Dados

Aula 21

ALBERTO ABAD, IST, 2022-23

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 a **abstração procedimental** para definir funções:
 - Para uma função definimos um *nome*, entradas e saídas, assim escondemos os pormenores de implementação ao utilizador/resto do programa → Separação do **quê** 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 abstratas 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.
- Muitas vezes é necessário representar e operar sobre diferentes tipos de informação nos nossos programas e que não existem na linguagem.
- Esta semana, veremos como definir tipos estruturados de informação *custom* recorrendo ao conceito de **abstração de dados**:
 - Equivalente às abstrações procedimentais mas para estruturas de dados.
 - Permite separar o modo como pode ser utilizada, e o que representa (o **quê**), da forma como é construída e representada a partir de outros tipos e estruturas de dados (o **como**).

Abstração de Dados

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**.

Abstração de Dados: Números Complexos

EXEMPLO DE MOTIVAÇÃO

- 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 equação $i^2 = -1$.
- A soma, subtração, multiplicação e divisão de números complexos são definidas do seguinte modo:

$$(a + bi) + (c + di) = (a + c) + (b + d)i \quad (1)$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i \quad (2)$$

$$(a + bi) \times (c + di) = (ac - bd) + (ad + bc)i \quad (3)$$

$$\frac{a + bi}{c + di} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i \quad (4)$$

Números Complexos: Primeira Abordagem

SOLUÇÃO DEPENDENTE DA REPRESENTAÇÃO

- **Solução/sequência errada:** Desenvolver para uma representação concreta, neste caso, tuplos.

```
In [25]: def sum_compl(c1, c2):
          r = c1[0] + c2[0]
          i = c1[1] + c2[1]
          return (r, i)

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

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

def div_compl(c1, c2):
    den = c2[0]**2 + c2[1]**2
    if den != 0:
        r = (c1[0] * c2[0] + c1[1] * c2[1])/den
        i = (c1[1] * c2[0] - c1[0] * c2[1])/den
        return r, i
    raise ValueError("divisao por 0")

sum_compl((1,2),(2,3))
```

Out[25]:

(3, 5)

- 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 retorna a parte real.
 - **p_imag(c)** - recebe como argumento um número complexo e retorna a parte imaginária.
- Podemos escrever uma solução que utilize estas funções **independentemente** da representação.

Números Complexos: Segunda Abordagem

SOLUÇÃO INDEPENDENTE DA REPRESENTAÇÃO

```
In [26]: def sum_compl(c1, c2):  
         r = p_real(c1) + p_real(c2)  
         i = p_imag(c1) + p_imag(c2)  
         return cria_compl(r, i)  
  
def sub_compl(c1, c2):  
    r = p_real(c1) - p_real(c2)  
    i = p_imag(c1) - p_imag(c2)  
    return cria_compl(r, i)  
  
def mul_compl(c1, c2):  
    r = p_real(c1) * p_real(c2) - p_imag(c1) * p_imag(c2)  
    i = p_real(c1) * p_imag(c2) + p_imag(c1) * p_real(c2)  
    return cria_compl(r, i)  
  
def div_compl(c1, c2):  
    pass  
  
# sum_compl(cria_compl(1,2), cria_compl(2,3))
```

Números Complexos: Segunda Abordagem

SOLUÇÃO INDEPENDENTE DA REPRESENTAÇÃO

- Baseada nesta *biblioteca* podemos definir novas funções, por exemplo de *representação externa*, `compl_para_string(c)`:

```
In [30]: def compl_para_string(c):
         return '{}{}{}'.format(p_real(c), '+' if p_img(c) >= 0 else '-', abs(p_img(c))
         # return str(p_real(c)) + '+' + str(p_img(c)) + 'i'

compl_para_string(cria_compl(1,2))
```

Out[30]:

'1+2i'

Números Complexos: Segunda Abordagem

SOLUÇÃO INDEPENDENTE DA REPRESENTAÇÃO

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

```
In [35]: #Representing as a tuple
         def cria_compl(r, i):
             if isinstance(r, (int, float)) and isinstance(i, (int, float)):
                 return r, i
             raise ValueError('argumentos invalidos')

         def p_real(c):
             if type(c) == tuple and len(c) == 2 and isinstance(c[0], (int, float)) and isinstance(
                 return c[0]
             raise ValueError('argumento invalido')

         def p_img(c):
             if type(c) == tuple and len(c) == 2 and isinstance(c[0], (int, float)) and isinstance(
                 return c[1]
             raise ValueError('argumento invalido')

         c1 = cria_compl(10, 5)
         c2 = cria_compl(3, -10)
         compl_para_string(mul_compl(c1, c2))
```

Out[35]:

'80-85i'

Números Complexos: Segunda Abordagem

SOLUÇÃO INDEPENDENTE DA REPRESENTAÇÃO

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

```
In [37]: #Representing as a dictionary
def cria_compl(r, i):

    return {'r':r, 'i':i}

def p_real(c):
    if type(c) == dict and len(c) == 2 and 'r' in c and 'i' in c:
        return c['r']

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

c1 = cria_compl(10, 5)
compl_para_string(c1)
```

Out[37]:

'10+5i'

Abstração de Dados: Vetores

OUTRO EXEMPLO

- Consideremos um tipo de dados abstrato para representar vetores num espaço bidimensional.
- Operações a suportar:
 - **cria_vetor(x, y)**: dados dois números reais x e y retorna o vector (x, y)
 - **vetor_abcissa(v)**: dado um vetor v retorna a abcissa
 - **vetor_ordenada(v)**: dado um vetor v retorna a ordenada
 - **e_vetor(e)**: dado um qualquer elemento e reconhece se o mesmo é um vetor ou não
 - **vetor_igual(u,v)**: dados dois vetores indica se os mesmos são ou não iguais
 - **vetor_para_string(v)**: dado um vetor v retorna uma *string* que o representa.

Abstração de Dados: Vetores

OUTRO EXEMPLO

```
In [13]: # construtor (como tuplo)
def cria_vetor(x, y):
    # verifica validade dos argumentos
    pass

cria_vetor(10,20)
```

Out[13]:

(10, 20)

```
In [14]: # seletor
def vetor_abscissa(v):
    # verifica validade do argumento
    pass
```

```
In [15]: # seletor
def vetor_ordenada(v):
    # verifica validade do argumento
    pass
```

Abstração de Dados: Vetores

OUTRO EXEMPLO

```
In [24]: # reconhecedor
def e_vetor(arg):
    pass
```

```
In [23]: # teste
def vetor_igual(u,v):
    pass
```

```
In [22]: # transformador
def vetor_para_string(v):
    # <x,y>
    pass
```

Abstração de Dados: Vetores

OUTRO EXEMPLO

Produto escalar (*dot product*) → Utilizar funções anteriores

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

```
In [25]: # u . v = (1,2) x (4,5) = 1x4 + 5x2 = 14
         def produto_escalar(u, v):
             pass

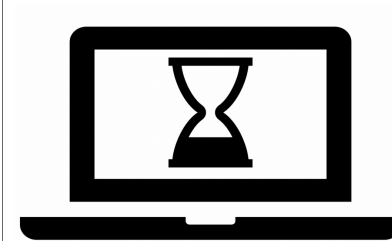
u = cria_vetor(1,2)
v = cria_vetor(4,5)
print(vetor_para_string(u))
print(vetor_para_string(v))
print(produto_escalar(u, v))
```

```
<1,1>
<1,1>
14
```

Abstração de dados

TAREFAS PRÓXIMAS AULAS

- Nas teóricas:
 - Ler secções 9.3 e 9.4 do livro da disciplina
 - Completar exemplos
- Nas práticas:
 - Avaliação: Cadeias de caracteres e listas
 - Primeira aula (L09): Dicionários
 - Segunda aula (L10): TADs



```
In [ ]:
```