# Tipos abstratos de dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Você acaba de chegar em uma empresa e a sua primeira atividade é concluir um código que havia sido iniciado pelo Roberto, que foi transferido para outra equipe.

Roberto é um bom desenvolvedor e costuma fazer a especificação antes de fazer a implementação, então o seu trabalho é escrever a implementação e fazer a verificação.

Note que a empresa usa uma convenção particular para nomear as funções.

# Especificação Robô

```python
@dataclass
class Robo:
    '''Um robo com um nome que está em uma posição da linha do jogo, que deve
    ser um valor entre 1 e 10.'''

def robo_cria(nome: str) -> Robo:
    '''Cria um novo robo com o *nome* e que está na posição 1.'''

def robo_posicao(r: Robo) -> int:
    '''Devolve a posição atual do robo *r*.'''

def robo_info(r: Robo) -> str:
    '''Devolve um texto com o nome do robo *r* seguido da sua posição entre parêntes.'''

def robo_move(r: Robo, n: int):
    '''
    Altera a posição de *r* avançando *n* posições (até no máximo a posição 10)
    se *n* for positivo, ou recuando -*n* posições (até no mínimo a posição 1)
    se *n* for negativo. O robo *r* permanece na mesma posição se *n* for 0.
    '''
```

# Especificação Robô

```
def robo_cria(nome: str) -> Robo:
    '''Cria um novo robo com o *nome* e que
    está na posição 1.

    Exemplos
    >>> r = robo_cria('r2d2')
    >>> robo_info(r)
    'r2d2 (1)'
    '''

def robo_posicao(r: Robo) -> int:
    '''
    Devolve a posição atual do robo *r*.

    Exemplos
    >>> r = robo_cria('rob')
    >>> robo_move(r, 2)
    >>> robo_posicao(r)
    3
    '''
```

```
def robo_info(r: Robo) -> str:
    '''Devolve um texto com o nome do robo *r*
    seguido da sua posição entre parêntes.

    Exemplos
    >>> r = robo_cria('rob')
    >>> robo_move(r, 2)
    >>> robo_info(r)
    'rob (3)'
    '''
```

# Especificação Robô

```
def robo_move(r: Robo, n: int):
    '''
    Altera a posição de *r* avançando *n* posições (até no máximo a posição 10)
    se *n* for positivo, ou recuando -*n* posições (até no mínimo a posição 1) se *n* for negativo...
    >>> r = robo_cria('rob')
    >>> # Avança
    >>> robo_move(r, 5)
    >>> robo_posicao(r)
    6
    >>> robo_move(r, 6)
    >>> robo_posicao(r)
    10
    >>> # Recua
    >>> robo_move(r, -3)
    >>> robo_posicao(r)
    7
    >>> robo_move(r, -8)
    >>> robo_posicao(r)
    1
    '''
```

O que a especificação feita pelo Roberto tem de diferente do que estamos acostumados?

- Estamos acostumados a fazer a **especificação de uma função** que resolve um problema específico.
- A especificação feita pelo Roberto envolve um tipo e uma coleção de funções relacionadas com esse tipo.

# Atividade

Forme uma dupla e implemente a definição e operações do tipo `Robo` e confira se a implementação funciona corretamente.

Inicie com o arquivo `robo.py` disponível na página da disciplina.

# Implementação

```python
@dataclass
class Robo:
    nome: str
    posicao: int

def robo_cria(nome: str) -> Robo:
    return Robo(nome, 1)

def robo_posicao(r: Robo) -> int:
    return r.posicao

def robo_info(r: Robo) -> str:
    return r.nome + ' (' + str(r.posicao) + ')'

def robo_move(r: Robo, n: int):
    r.posicao = max(1, min(10, r.posicao + n))
```

# Uso

```
>>> r = robo_cria('rob')
>>> # Avança
>>> robo_move(r, 5)
>>> robo_posicao(r)
6
>>> # Recua
>>> robo_move(r, -3)
>>> robo_posicao(r)
3
>>> # Info
>>> robo_info(r)
'rob (6)'
```

O que podemos observar de “diferente” na forma de uso da classe **Robo** em relação ao que estamos acostumados?

- Estamos acostumados a criar valores de classes diretamente e também a ler e modificar os valores dos campos diretamente.
- Nesse exemplo criamos valores da classe com a função `cria_robo` e usamos funções para ler e modificar os campos.

# Tipos concretos e abstratos de dados

Um tipo de dado (classe) em que a representação interna é conhecida e pode ser manipulada diretamente é chamado de **tipo concreto de dado**.

Um tipo de dado em que a representação interna não é conhecida e que é manipulado apenas através de funções é chamada do **tipo abstrato de dado**.

A ocultação da representação interna é chamada de encapsulamento.

# Tipo abstrator de dados

De maneira mais formal, um **tipo abstrato de de dado** (TAD) é um modelo teórico de tipo de dado, definido pela comportamento do ponto de vista do usuário do tipo, incluindo:

- Possíveis valores
- Possíveis operações
- Comportamento das operações

A especificação criada pelo Roberto define um TAD para um Robô!

A forma de criar novos tipos em Python é usando a construção **`class`**.

Até agora usamos classes como um forma de definir dados compostos, que usávamos como tipos concretos de dados, isto é, manipulando diretamente os campos (representação interna) da classe.

Na especificação feita pelo Roberto usamos uma classe para implementar um TAD. Observe a diferença entre quem implementa o TAD e quem usa o TAD:

- Quem implementa o tipo tem acesso e manipula diretamente os campos da classe
- Quem usa o tipo não tem acesso direto aos campos e utiliza funções para fazer operações com os dados do tipo

# Métodos

Apesar de podemos usar funções “livres” para implementar TADs, o comum em Python é usar métodos.

Um método é uma função que está associada com uma classe particular.

Para criar um método em um classe, basta definir uma função “dentro” da classe!

# Definição de Métodos

```python
@dataclass
class Robo:
    nome: str
    posicao: int

def robo_cria(nome: str) -> Robo: ...

def robo_posicao(r: Robo) -> int: ...

def robo_info(r: Robo) -> str: ...

def robo_move(r: Robo, n: int):
    '''
    >>> r = robo_cria('rob')
    >>> robo_move(r, 5)
    >>> robo_posicao(r)
    6
    '''
```

```python
@dataclass
class Robo:
    nome: str
    posicao: int

    def robo_cria(nome: str) -> Robo: ...

    def robo_posicao(r: Robo) -> int: ...

    def robo_info(r: Robo) -> str: ...

    def robo_move(r: Robo, n: int):
        '''
        >>> r = Robo.robo_cria('rob')
        >>> r.robo_move(5)
        >>> r.robo_posicao()
        6
        '''
```

# Definição de Métodos

```python
@dataclass
class Robo:
    nome: str
    posicao: int

def robo_cria(nome: str) -> Robo: ...

def robo_posicao(r: Robo) -> int: ...

def robo_info(r: Robo) -> str: ...

def robo_move(r: Robo, n: int):
    '''
    >>> r = robo_cria('rob')
    >>> robo_move(r, 5)
    >>> robo_posicao(r)
    6
    '''
```

```python
@dataclass
class Robo:
    nome: str
    posicao: int

    def cria(nome: str) -> Robo: ...

    def posicao(r: Robo) -> int: ...

    def info(r: Robo) -> str: ...

    def move(r: Robo, n: int):
        '''
        >>> r = Robo.cria('rob')
        >>> r.move(5)
        >>> r.posicao()
        6
        '''
```

# Definição de Métodos

```python
@dataclass
class Robo:
    nome: str
    posicao: int

def robo_cria(nome: str) -> Robo: ...

def robo_posicao(r: Robo) -> int: ...

def robo_info(r: Robo) -> str: ...

def robo_move(r: Robo, n: int):
    '''
    >>> r = robo_cria('rob')
    >>> robo_move(r, 5)
    >>> robo_posicao(r)
    6
    '''
```

```python
class Robo:
    nome: str
    posicao: int

    def __init__(r: Robo, nome: str): ...

    def posicao(r: Robo) -> int: ...

    def info(r: Robo) -> str: ...

    def move(r: Robo, n: int):
        '''
        >>> r = Robo('rob')
        >>> r.move(5)
        >>> r.posicao()
        6
        '''
```

# Definição de Métodos

```python
@dataclass
class Robo:
    nome: str
    posicao: int

def robo_cria(nome: str) -> Robo: ...

def robo_posicao(r: Robo) -> int: ...

def robo_info(r: Robo) -> str: ...

def robo_move(r: Robo, n: int):
    '''
    >>> r = robo_cria('rob')
    >>> robo_move(r, 5)
    >>> robo_posicao(r)
    6
    '''
```

```python
class Robo:
    nome: str
    posicao: int

    def __init__(self, nome: str): ...

    def posicao(self) -> int: ...

    def info(self) -> str: ...

    def move(self, n: int):
        '''
        >>> r = Robo('rob')
        >>> r.move(5)
        >>> r.posicao()
        6
        '''
```

Usamos `@dataclass` quando queremos um dado composto simples, sem operações operações associadas (ou com operações simples).

Quando usamos `@dataclass` um construtor que recebe um argumento para cada campo é criado, dessa forma não precisamos criar o método `__init__`.

Além do construtor as funções `__eq___`, `__repr__`, `__str__`, `__hash__` são criadas automaticamente.

```python
@dataclass
class Ponto:
    x: int
    y: int
```

Essa construção é mais ou menos equivalente ao código ao lado!

Não se preocupe com essas funções “estranhas”, a única que vamos utilizar por enquanto é o `__init__`.

```python
class Ponto:
    x: int
    y: int

    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def __str__(self) -> str: ...

    def __repr__(self) -> str: ...

    def __hash__(self) -> int: ...

    def __eq__(self, other: Ponto) -> bool:...
```

Porque não usar `@dataclass` na classe `Robo`?

Porque essas coisas geradas automaticamente não são adequadas para a classe `Robo`!

Pode parecer confuso quando usar ou não o `@dataclass`, mas não se preocupe, isso vai ficar mais claro com a prática!

O importante por enquanto é saber como usar classes para especificar e implementar TADs.

# Referências

Capítulo 2 - Visão geral das coleções - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca da UEM)

Seção 1.2 - Interfaces - Open Data Structures (in pseudocode)