Tipos abstratos de dados

Estruturas 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 fazer a implementação.

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 parênteses.'''

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
    '''
```

Implementação

Vamos fazer a implementação.

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 = r.posicao + n
    if r.posicao < 1:
        r.posicao = 1
    if r.posicao > 10:
        r.posicao = 10
```

Essa é a única implementação possível?

Não!

É possível fazer uma implementação com um representação diferente para Robo?

Sim! Veja o exercício na lista.

Vamos deixar a implementação de lado e focar na especificação.

O que a especificação feita pelo Roberto têm 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**;
- Nos exemplos, estamos acostumados a criar valores de classes e acessar os campos diretamente, nos exemplos do Roberto **os valores são criados e os campos acessados indiretamente através de funções**.

O que os nomes das funções têm em comum?

- O prefixo `robo_`.

Por que?

- Porque estão relacionas com o tipo `Robo`.

Um tipo de dado em que a representação interna é conhecida e pode ser manipulada diretamente pelo usuário do tipo 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 chamado de **tipo abstrato de dado**.

A ocultação da representação interna é chamada de **encapsulamento**.

Os tipos de dados abstratos de forma específica, e a construção de abstrações de forma geral, são formas essências de controlar a complexidade no desenvolvimento de software.

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
- As operações
- O comportamento das operações

Muitos tipos pré-definidos em Python são TADs, como `list`, `dict` e até mesmo `int`.

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

Existem três papéis no desenvolvimento e uso de TADs:

- Quem especifica;
- Quem implementa;
- Quem usa.

Qual é a diferença entre quem implementa e quem usa o TAD?

- Quem implementa pode manipular diretamente a representação interna;
- Quem usa o tipo tem que usar a interface (funções) do tipo e não pode manipular diretamente a representação interna.

Quais são as vantagens dos TADs?

- Facilita o reuso criando abstrações;
- Facilita a verificação permitindo que cada TAD seja testado de forma isolada;
- Aumenta a confiabilidade mantendo os valores em estado consistente;
- Facilita a manutenção permitindo que o software seja decomposto em partes e que a implementação possa ser alterada sem que o código do usuário do TAD tenha que ser modificado;
- Agiliza o desenvolvimento permitindo que diferentes partes de um software sejam desenvolvidas simultaneamente.

E as desvantagens?

- A especificação requer um investimento inicial maior;
- Pode aumentar a complexidade adicionando abstrações desnecessárias;
- Pode gerar perda de desempenho devido as construções de abstrações.

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.

Por último, usamos uma classe para implementar o TAD especificado pelo Roberto.

Métodos

Apesar de podermos 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!

Vamos transformar nossa implementação de TAD em um classe com métodos.

Nos slides a seguir, observe com **ATENÇÃO** as mudanças entre os códigos a esquerda e direita.

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')
        >>> Robo.robo_move(r, 5)
        >>> Robo.robo_posicao(r)
        6
        '''
```

O prefixo `robo_` fui usado inicialmente para indicar que as funções estavam relacionadas com o tipo `Robo`, mas usando métodos essa relação já está clara, então o prefixo não é necessário.

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.robo_cria('rob')
        >>> Robo.robo_move(r, 5)
        >>> Robo.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')
        >>> Robo.move(r, 5)
        >>> Robo.posicao(r)
        6
        '''
```

Apesar de podermos chamar um método usando o nome da classe, como em `list.append(x, 10)` ou `Robo.move(r, 6)`, a forma apropriada para a maioria dos casos é chamar o método diretamente com uma variável, sem usar o nome da classe, como em `x.append(10)` ou `r.move(6)`.

Observe a diferença na ênfase:

`Robo.move(r, 6)` – `Robo.move` com os argumentos `r` e `6`.

`r.move(6)` – considerando `r`, `move` `6` (`r` está em evidência).

Note que para chamar uma método de uma classe sem usar o nome da classe precisamos ter um valor da classe.

Note que as duas formas não são exatamente equivalentes em todos os casos (não vamos discutir sobre isso nessa disciplina).

Definição de métodos

```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')
        >>> 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
        '''
```

Ainda precisamos usar o nome da classe na chamada do método `cria` para criar um `Robo`.

De forma geral, sempre será necessário uma função para criar (construtor) um valor de uma classe, por isso, o Python fornece uma maneira conveniente de fazer isso: o método `__init__`.

De fato, o método `__init__` não serve para criar um valor de uma classe, e sim, para inicializar um valor que já foi criado (implicitamente pelo Python).

Para chamar o método `__init__` usamos o nome do tipo como se fosse uma função: `Robo('r2d2', 1)`. Note que nós não definimos o método `__init__` para a classe `Robo`, mas ele foi criado automaticamente porque usamos o `@dataclass` (mais sobre isso daqui a pouco).

Para definirmos explicitamente o método `__init__`, vamos remover o `@dataclass`.

Definição de métodos

```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
        '''
```

```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
        '''
```

Vamos fazer uma última mudança.

O primeiro argumento de todos os métodos da classe `Robo` é `r: Robo`. Isso não é por acaso, os métodos estão na classe `Robo` porque eles manipulam valores do tipo `Robo`, então precisamos de uma variável do tipo `Robo`.

Por convenção do Python, podemos usar `self` (sem especificar o tipo) como nome para a variável do tipo da classe.

Essa mudança não altera a forma de usar o tipo, apenas o nome da variável que será usado na implementação dos métodos.

Definição de métodos

```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
        '''
```

```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
        '''
```

Daqui para frente, vamos usar essa última forma para implementar TADs.

`@dataclass`

Em geral, usamos `@dataclass` quando queremos um tipo concreto de dado.

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.

@dataclass

```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 a `__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:...
```

@dataclass

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

Porque queremos implementar um TAD e nem tudo que é gerado automaticamente pelo `@dataclass` é necessário para isso.

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)