

Tipos abstratos de dados

Ao escrever um programa ou sistema, o objetivo é pensar em primeiro lugar no tipo de dado e nas operações sobre esses dados, separando ou isolando o conceito do particular problema que estamos tentando resolver. Dizemos então que estamos criando uma abstração desse tipo de dado. Com isso, podemos conseguir construir um tipo de dado que seja reutilizável para resolver também outros problemas similares. Um mínimo de adaptações pode ser admitido com extensões possíveis do tipo de dado utilizado.

Python, num certo sentido já proporciona isso. O tipo de dado em Python é dinâmico, conhecido e estabelecido em tempo de execução. O mesmo nome, ou referência pode no mesmo programa estar associado a objetos de diferentes tipos. Podemos ter no mesmo programa:

```
x = 1
...
x = 'exemplo'
...
x = True
...
x = [4, 6, "chatice", True, (2,5,4)]
...
```

Uma função pode ser construída e funcionar para diferentes tipos de dados sem que haja a necessidade de adaptações:

```
def Max_Valor(seq):

    ''' Recebe uma sequência (lista, tupla, string, etc.).
        Devolve o máximo desta.'''

    mv = seq[0]
    for k in range(1, len(seq)):
        if mv < seq[k]: mv = seq[k]
    return mv
```

Podemos usar essa função com vários dados diferentes. De fato, com qualquer objeto “iterável” como parâmetro:

```
# testes de Max_Valor

# com lista de int
x = [5, 2, 8, 0]
print("lista - máximo valor de ", x, " = ", Max_Valor(x))

# com tupla de int
x = 5, 2, 8, 0
print("tupla - máximo valor de ", x, " = ", Max_Valor(x))

# com string
x = "abracadabra"
print("string - máximo valor de ", x, " = ", Max_Valor(x))
```

```
# com lista de string
x = ["cinco", "dois", "oito", "zero"]
print("lista de string - máximo valor de ", x, " = ", Max_Valor(x))

# com lista de tuplas
x = [(3, 4, 2), (3, 4, 6), (3, 4, 4)]
print("lista de tuplas - máximo valor de ", x, " = ", Max_Valor(x))

# com tupla de tuplas
x = (3, 4, 2), (3, 4, 6), (3, 4, 4)
print("tupla de tuplas - máximo valor de ", x, " = ", Max_Valor(x))
```

Será impresso:

```
lista - máximo valor de [5, 2, 8, 0] = 8
tupla - máximo valor de (5, 2, 8, 0) = 8
string - máximo valor de abracadabra = r
lista de string - máximo valor de ['cinco', 'dois', 'oito', 'zero'] =
zero
lista de tuplas - máximo valor de [(3, 4, 2), (3, 4, 6), (3, 4, 4)] =
(3, 4, 6)
tupla de tuplas - máximo valor de ((3, 4, 2), (3, 4, 6), (3, 4, 4)) =
(3, 4, 6)
```

Essa independência do tipo de dados proporciona construir uma interface para o usuário de uma função ou um conjunto de funções, de modo que a relação entre os algoritmos e as estruturas de dados não sejam explícitas.

Rigorosamente, na função `Max_Valor` acima teríamos que fazer a consistência do tipo. O tipo dos elementos da sequência `seq` deve ser o mesmo.

```
def Max_Valor(seq):

    ''' Recebe uma sequência (lista, tupla, string, etc.).
        Devolve o máximo desta.'''

    mv, tmv = seq[0], type(seq[0])
    for k in range(1, len(seq)):
        # consistência dos tipos
        if type(seq[k]) != tmv:
            raise ValueError("tipos diferentes")
        if mv < seq[k]: mv = seq[k]
    return mv
```

O conceito de Tipo de Dados Abstrato (Abstract Data Type ou ADT)

É um conjunto de objetos e as operações sobre esses objetos. As operações definem uma interface entre o ADT e o resto do programa. A interface define o comportamento destas operações, o que elas fazem e não como fazem. Existe então uma abstração que isola o resto do programa das estruturas de dados e algoritmos utilizados.

Mais especificamente, um ADT é um modelo de estrutura de dados que especifica:

- a) O tipo de dados armazenado (tipo conceitual, não int, bool, list, etc.)
- b) As operações suportadas sobre esses dados
- c) Os parâmetros dessas operações
- d) O que cada operação faz e não como faz

Em Python, a representação de um ADT são as classes. As classes com seus atributos (as variáveis que representam os dados, os seus tipos e que definem o estado deste objeto) e os métodos (as operações, parâmetros e o que pode ser feito sobre um objeto desta classe), implementam esses conceitos.

Exemplo de ADT – Fração Ordinária (classe Fração)

Uma fração ordinária é definida por dois inteiros a e b. a é o numerador e b o denominador e representam o número a/b , com $b > 0$. Consideramos que o numerador é negativo quando a fração é negativa.

Definimos então a classe Fração que será um ADT no nosso conceito. Primeiramente, além do construtor da classe (método `__init__`), vamos definir um método `Some_Fração(f1, f2)` que devolve a soma $f1 + f2$ e um método `Mostre_Fração(f1)` que mostra o número f1 no formato a/b:

```
class Fração:

    '''Define a classe Fração Ordinária.'''

    # Construtor da classe
    def __init__(self, topo = 0, base = 1):

        '''Cria uma instância de Fração.'''

        self.num = topo
        self.den = base

    # Retorna a soma de dois elementos da classe
    def Some_Fração(self, f1, f2):
        self.num = f1.num * f2.den + f1.den * f2.num
        self.den = f1.den * f2.den

    # Mostre uma fração na forma a/b
    def Mostre_Fração(self):
        print(self.num, "/", self.den)

# Exemplos de uso da classe Fração
```

```
x = Fração(3, 4)
y = Fração(5, 12)
z = Fração()

# z = x + y
z.Some_Fração(x, y)
z.Mostre_Fração()

# z = 4/5 + 3/7
z.Some_Fração(Fração(4, 5), Fração(3, 7))
z.Mostre_Fração()
```

Será impresso:

```
56 / 48
43 / 35
```

Algumas observações sobre a solução acima:

- Seria conveniente guardar o resultado da soma na forma mais simples. $56/48$ é o mesmo que $7/6$. Para tanto, basta dividir o numerador e denominador pelo máximo divisor comum entre eles. Vamos então incorporar a função `mdc(n, m)` dentro da nossa classe.
- Seria também conveniente, em vez de usar a função `Some_Fração` usar o mecanismo de sobrecarga de operadores e escrever `z = x + y` em vez de `z.Some_Fração(x, y)`. O programa fica muito mais claro. Para isso a função de somar passará a ser `__add__()` como visto anteriormente.
- Seria também conveniente, usar o mesmo mecanismo para escrever `print(z)` em vez de `z.Mostre_Fração()`. Para isso, teremos que transformar uma fração em uma string, construindo a função `__str__()`. Importante aqui entender o mecanismo de sobreposição de métodos. A função `print` sempre tenta transformar os seus parâmetros em string. Quando o parâmetro pertence a uma determinada classe, a função `__str__()` desta classe é chamada. Se existe a conversão é feita e o `print` segue normalmente. Se não existe a conversão para esta classe é impresso um erro.
- O denominador de uma Fração não pode ser zero. Assim vamos sinalizar essa exceção.

Abaixo a nova versão da classe Fração com alguns testes.

```
# função mdc
def mdc(n, m):
    resto = n % m
    while resto != 0:
        n = m
        m = resto
```

```
        resto = n % m
    return m

class Fração:

    '''Define a classe Fração Ordinária.'''

    # Construtor da classe
    def __init__(self, topo = 0, base = 1):

        '''Cria uma instância de Fração.'''

        if base == 0:
            raise ValueError("Denominador igual a zero")
        self.num = topo
        self.den = base

    # Retorna a soma de dois elementos da classe
    def __add__(self, f2):

        xnum = self.num * f2.den + self.den * f2.num
        xden = self.den * f2.den
        xmdc = mdc(xnum, xden)
        return Fração(xnum // xmdc, xden // xmdc)

    # Transforma em string para o print
    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    # Compara dois elementos da classe
    def __eq__(self, f2):
        pri_fator = self.num * f2.den
        seg_fator = self.den * f2.num
        return pri_fator == seg_fator

# testes da classe

x = Fração(3, 4)
y = Fração(5, 12)
z = Fração()

print(x + y)

z = x + y
print(z)

print(x == y)
```

Será impresso:

```
7/6  
7/6  
False
```

P3.5.1) Complete a classe acima, definindo as operações de subtração, multiplicação e divisão de frações (operadores -, * e /)

P3.5.2) Idem para os operadores de comparação maior, menor, maior ou igual, menor ou igual e diferente (operadores >, <, >=, <=, != ou as funções __gt__, __lt__, __ge__, __le__, __ne__)

Conjuntos como uma lista

Python já possui a classe set (conjuntos) que provê todas as operações com esse tipo de objeto. Supondo que tal classe não existisse, ou mesmo que para uma aplicação específica queremos representar um conjunto como uma lista. Queremos também que esse conjunto esteja classificado em ordem crescente dos elementos e para isso usaremos a função sort(). Vamos denominá-la por super conjunto (Sset).

O construtor da classe deverá então receber uma lista, eliminar possíveis repetições de elementos e classificar a lista.

```
def Elimina_Repetidos(lista):  
    nova_lista = []  
    for k in range(0, len(lista)):  
        tem_rep = False  
        # procura lista[k] nos elementos à frente  
        for j in range(k + 1, len(lista)):  
            if lista[k] == lista[j]:  
                tem_rep = True  
                break  
        # se não tem repetidos inclua na nova lista  
        if not tem_rep: nova_lista.append(lista[k])  
    return nova_lista  
  
class Sset:  
  
    ''' Define a classe Super Conjunto. '''  
  
    # Construtor da classe  
    def __init__(self, lista = []):  
  
        ''' Cria uma instância de Sset. '''  
  
        self.lset = Elimina_Repetidos(lista)  
        self.lset.sort()  
  
    # união (+)  
    def __add__(s1, s2):  
        # cria Sset com a concatenação de duas listas  
        return Sset(s1.lset + s2.lset)  
  
    # Tamanho do Sset  
    def __len__(self):  
        return len(self.lset)
```

```
# Transforma elemento da classe em str para o print
# Mostrar um '*' na frente para diferenciar do print(lista)
def __str__(self):
    return '*' + str(self.lset)

# testes da classe
x = Sset([1, 2, 25, 5, 72, 5, 1])
print(len(x))
print(x)

y = Sset([])
print(len(y))
print(y)

z = Sset([1, 9, 113, 5, 9, 2, 22, 12, 2, 11])
print(len(z))
print(z)

t = x + z
print(t)

print(x + z)
```

Saída:

```
5
*[1, 2, 5, 25, 72]
0
*[]
8
*[1, 2, 5, 9, 11, 12, 22, 113]
*[1, 2, 5, 9, 11, 12, 22, 25, 72, 113]
*[1, 2, 5, 9, 11, 12, 22, 25, 72, 113]
```

P3.5.3) Existem formas mais simples de escrever a Elimina_Repetidos acima. Escreva-a de outras maneiras.

P3.5.4) Incremente agora a classe Sset acima definindo as operações de intersecção de conjuntos com o operador * (__mul__) e subtração de conjuntos com o operador - (__sub__).

P3.5.5) Idem com as operações booleanas “está contido” com o operador < (__lt__), “contém” com o operador > (__gt__) e “igual” com o operador == (__eq__).

P3.5.6) Idem com a operação “pertinência” com o operador in (__contains__), “inclusão” de elemento com += (__iadd__) e exclusão com -= (__isub__).

P3.5.7) Incremente a classe Vector vista anteriormente com as operações de subtração de vetores, produto escalar e produto vetorial.

Polinômios como uma lista

Um polinômio pode ser representado como uma lista de valores float, onde o elemento j representa o coeficiente de x^j . Portanto uma lista com n floats representa um polinômio de grau n-1.

```
class Polinomio:
```

```
''' Define a classe Polinomio. '''  
  
def __init__(s, coef):  
  
    s.cfs = coef[:] # coeficientes  
  
# Devolve o valor do polinômio no ponto x  
def valor(self, x):  
    soma = 0  
    for k in range(len(self.cfs)):  
        soma += self.cfs[k] * pow(x, k)  
    return soma  
  
# testes  
p1 = Polinomio([1.0, -2.2, 3.5])  
v = p1.valor(1.0)  
print(v)
```

Saída:

2.3

P3.5.8) Construa uma classe Polinômio, baseada no exemplo acima onde cada polinômio é representado por uma lista. Construa também as operações usuais: soma, subtração, multiplicação, divisão, derivação e integração. Algumas das operações alteram o grau do polinômio.

Matrizes como listas de listas

Uma matriz de n linhas por m colunas é uma lista de n elementos onde cada elemento é uma lista de m elementos.

```
class Matriz:  
  
    ''' Define a classe Matriz de n linhas por m colunas. '''  
  
    def __init__(s, n, m):  
  
        # inicialmente todos zerados  
        s.elem = [m * [0] for k in range(n)]  
        # pode ser também [m * [0]] * n  
  
        # define a k-esima linha da Matriz  
    def __setitem__(s, k, v):  
        s.elem[k] = v[:]  
  
        # tamanho das matrizes para consistência  
    def dim(s):  
        # linhas e colunas  
        n, m = len(s.elem), len(s.elem[0])  
        return (n, m)  
  
        # soma duas matrizes de mesma dimensão  
    def __add__(m1, m2):  
        # consistência dos tamanhos  
        if m1.dim() != m2.dim():  
            raise TypeError("tamanhos diferentes")
```



```
# linhas e colunas
n, m = len(m1.elem), len(m1.elem[0])
m3 = Matriz(n, m)
for i in range(n):
    for j in range(m):
        m3.elem[i][j] = m1.elem[i][j] + m2.elem[i][j]
# resultado
return m3

def __str__(s):
    lin = ''
    for i in range(len(s.elem)):
        for j in range(len(s.elem[i])):
            lin += str(s.elem[i][j]) + ' '
        lin += '\n'
    return lin

# testes
ma = Matriz(2, 3)
mb = Matriz(2, 3)
# define as linhas
ma[0] = [1, 2, 3]
ma[1] = [4, 5, 6]
mb[0] = [5, 5, 5]
mb[1] = [9, 9, 9]
# some
mc = ma + mb
# imprima
print(ma)
print(mb)
print(mc)
```

Saída:

```
1 2 3
4 5 6

5 5 5
9 9 9

6 7 8
13 14 15
```

P3.5.9) Construa uma classe Matriz, baseada no exemplo acima. O construtor da classe Matriz(self, n, m) cria uma lista de listas de n por m elementos zerados. Implemente as operações de subtração de matrizes (-) e multiplicação (*). Importante verificar a compatibilidade do tamanho das listas antes de realizar as operações.