Aula 2B - Computação Simbólica - II

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Junho de 2020

1 Computação Simbólica com *sympy* - Parte 2

Vamos continuar nossa trilha em computação simbólica estendendo o conhecimento sobre o tipo `bool`, expressões e testes lógicos.

1.1 Operadores lógicos

Vimos que `True` e `False` são os dois valores atribuíves a um objeto de tipo `bool`. Eles são úteis para testar condições, realizar verificações e comparar quantidades. Vamos estudar *operadores de comparação*, *operadores de pertencimento* e *operadores de identidade*.

1.1.1 Operadores de comparação

A tabela abaixo resume os operadores de comparação utilizados em Python.

operador	significado	símbolo matemático
<	menor do que	$<$
<=	menor ou igual a	$\leq$
>	maior do que	$>$
>=	maior ou igual a	$\geq$
==	igual a	$=$
!=	diferente de	$\neq$

Podemos usá-los para comparar objetos.

Nota: == está relacionado à igualdade, ao passo que = é uma atribuição. São conceitos operadores com finalidade distinta.

[1]:
```
2 < 3 # o resultado é um 'bool'
```

[1]:
```
True
```

[2]:
```
5 < 2 # isto é falso
```

[2]:
```
False
```

[3]:
```
2 <= 2 # isto é verdadeiro
```

[3]:
```
True
```

[4]:
```
4 >= 3 # isto é verdadeiro
```

[4]:
```
True
```

[5]:
```
6 != -2
```

[5]:
```
True
```

[6]:
```
4 == 4 # isto não é uma atribuição!
```

[6]:
```
True
```

Podemos realizar comparações aninhadas:

[7]:
```
x = 2
1 < x < 3
```

[7]:
```
True
```

[8]:
```
3 > x > 4
```

[8]:
```
False
```

[9]:
```
2 == x > 3
```

[9]:
```
False
```

As comparações aninhadas acima são resolvidas da esquerda para a direita e em partes. Isso nos leva a introduzir os seguintes operadores.

operador	símbolo matemático	significado	uso relacionado a
`or`	$\vee$	"ou" booleano	união, disjunção
`and`	$\wedge$	"e" booleano	interseção, conjunção
`not`	$\neg$	"não" booleano	exclusão, negação

[10]:
```
# parênteses não são necessários aqui
(2 == x) and (x > 3) # 1a. comparação: 'True'; 2a.: 'False'. Portanto, ambas:␣
↪'False'
```

[10]:
```
False
```

[11]:
```
# parênteses não são necessários aqui
(x < 1) or (x < 2) # nenhuma das duas é True. Portanto,
```

[11]: False

[12]:
```
not (x == 2) # nega o "valor-verdade" que é 'True'
```

[12]: False

[13]:
```
not x + 1 > 3 # estude a precedência deste exemplo. Por que é 'True'?
```

[13]: True

[14]:
```
not (x + 1 > 3) # estude a precedência deste exemplo. Por que também é 'True'?
```

[14]: True

1.1.2 Operadores de pertencimento

A tabela abaixo resume os operadores de pertencimento.

operador	significado	símbolo matemático
`in`	pertence a	$\in$
`not in`	não pertence a	$\notin$

Eles terão mais utilidade quando falarmos sobre sequências, listas. Neste momento, vejamos exemplos com objetos `str`.

[15]:
```
'2' in '2 4 6 8 10' # o caracter '2' pertence à string
```

[15]: True

[16]:
```
frase_teste = 'maior do que'
'maior' in frase_teste
```

[16]: True

[17]:
```
'menor' in frase_teste # a palavra 'menor' está na frase
```

[17]: False

[18]:
```
1 in 2 # 'in' e 'not in' não são aplicáveis aqui
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-18-5c8c51a582d3> in <module>
----> 1 1 in 2 # 'in' e 'not in' não são aplicáveis aqui


TypeError: argument of type 'int' is not iterable
```

1.1.3 Operadores de identidade

A tabela abaixo resume os operadores de identidade.

operador	significado
`is`	"aponta para o mesmo objeto"
`is not`	"não aponta para o mesmo objeto"

Esses operadores são úteis para verificar se duas variáveis se referem ao mesmo objeto. Exemplo:

```
a is b
a is not b
```

- `is` é `True` se `a` e `b` se referem ao mesmo objeto; `False`, caso contrário.
- `is not` é `False` se `a` e `b` se referem ao mesmo objeto; `True`, caso contrário.

[19]:
```
a = 2
b = 3
a is b # valores distintos
```

[19]:
```
False
```

[20]:
```
a = 2
b = a
a is b # mesmos valores
```

[20]:
```
True
```

[21]:
```
a = 2
b = 3
a is not b # de fato, valores não são distintos
```

[21]:
```
True
```

[22]:
```
a = 2
b = a
a is not b # de fato, valores são distintos
```

[22]:
```
False
```

1.2 Equações simbólicas

Equações simbólicas podem ser formadas por meio de Eq e não com = ou ==.

[23]:
```
# importação
from sympy.abc import a,b
import sympy as sy
sy.init_printing(pretty_print=True)
```

[24]:
```
sy.Eq(a,b) # equação simbólica
```

[24]: $a = b$

[25]:
```
sy.Eq(sy.cos(a), b**3) # os objetos da equação são simbólicos
```

[25]: $\cos(a) = b^3$

1.2.1 Resolução de equações algébricas simbólicas

Podemos resolver equações algébricas da seguinte forma:

solveset(equação,variável,domínio)

Exemplo: resolva $x^2 = 1$ no conjunto $\mathbb{R}$.

[26]:
```
from sympy.abc import x
sy.solveset( sy.Eq( x**2, 1), x,domain=sy.Reals)
```

[26]: $\{-1, 1\}$

Podemos reescrever a equação como: $x^2 - 1 = 0$.

[27]:
```
sy.solveset( sy.Eq( x**2 - 1, 0), x,domain=sy.Reals)
```

[27]: $\{-1, 1\}$

Com solveset, não precisamos de Eq. Logo, a equação é passada diretamente.

[28]:
```
sy.solveset( x**2 - 1, x,domain=sy.Reals)
```

[28]: $\{-1, 1\}$

Exemplo: resolva $x^2 + 1 = 0$ no conjunto $\mathbb{R}$.

[29]:
```
sy.solveset( x**2 + 1, x,domain=sy.Reals) # não possui solução real
```

[29]: $\emptyset$

Exemplo: resolva $x^2 + 1 = 0$ no conjunto $\mathbb{C}$.

[30]:
```
sy.solveset( x**2 + 1, x,domain=sy.Complexes) # possui soluções complexas
```

[30]: $\{-i, i\}$

Exemplo: resolva $\text{sen}(2x) = 3 + x$ no conjunto $\mathbb{R}$.

[31]:
```
sy.solveset( sy.sin(2*x) - x - 3,x,sy.Reals) # a palavra 'domain' também pode␣
↪ser omitida.
```

[31]: $\{x \mid x \in \mathbb{R} \wedge -x + \sin(2x) - 3 = 0\}$

O conjunto acima indica que nenhuma solução foi encontrada.

Exemplo: resolva $\text{sen}(2x) = 1$ no conjunto $\mathbb{R}$.

[32]:
```
sy.solveset( sy.sin(2*x) - 1,x,sy.Reals)
```

[32]: $\left\{2n\pi + \frac{5\pi}{4} \mid n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{\pi}{4} \mid n \in \mathbb{Z}\right\}$

1.3 Expansão, simplificação e fatoração de polinômios

Vejamos exemplos de polinômios em uma variável.

[33]:
```
a0, a1, a2, a3 = sy.symbols('a0 a1 a2 a3') # coeficientes
P3x = a0 + a1*x + a2*x**2 + a3*x**3 # polinômio de 3o. grau em x
P3x
```

[33]: $a_0 + a_1 x + a_2 x^2 + a_3 x^3$

[34]:
```
b0, b1, b2, b3 = sy.symbols('b0 b1 b2 b3') # coeficientes
Q3x = b0 + b1*x + b2*x**2 + b3*x**3 # polinômio de 3o. grau em x
Q3x
```

[34]: $b_0 + b_1 x + b_2 x^2 + b_3 x^3$

[35]:
```
R3x = P3x*Q3x # produto polinomial
R3x
```

[35]: $\left(a_0 + a_1 x + a_2 x^2 + a_3 x^3\right)\left(b_0 + b_1 x + b_2 x^2 + b_3 x^3\right)$

[36]:
```
R3x_e = sy.expand(R3x) # expande o produto
R3x_e
```

[36]: $a_0 b_0 + a_0 b_1 x + a_0 b_2 x^2 + a_0 b_3 x^3 + a_1 b_0 x + a_1 b_1 x^2 + a_1 b_2 x^3 + a_1 b_3 x^4 + a_2 b_0 x^2 + a_2 b_1 x^3 + a_2 b_2 x^4 + a_2 b_3 x^5 + a_3 b_0 x^3 + a_3 b_1 x^4 + a_3 b_2 x^5 + a_3 b_3 x^6$

[37]:
```
sy.simplify(R3x_e) # simplify às vezes não funciona como esperado
```

[37]: $a_0 b_0 + a_0 b_1 x + a_0 b_2 x^2 + a_0 b_3 x^3 + a_1 b_0 x + a_1 b_1 x^2 + a_1 b_2 x^3 + a_1 b_3 x^4 + a_2 b_0 x^2 + a_2 b_1 x^3 + a_2 b_2 x^4 + a_2 b_3 x^5 + a_3 b_0 x^3 + a_3 b_1 x^4 + a_3 b_2 x^5 + a_3 b_3 x^6$

[38]:
```
sy.factor(R3x_e) # 'factor' pode funcionar melhor
```

[38]: $\left(a_0 + a_1 x + a_2 x^2 + a_3 x^3\right)\left(b_0 + b_1 x + b_2 x^2 + b_3 x^3\right)$

[39]:
```
# simplify funciona para casos mais gerais
ident_trig = sy.sin(x)**2 + sy.cos(x)**2
ident_trig
```

[39]:
$$\sin^2(x) + \cos^2(x)$$

[40]:
```
sy.simplify(ident_trig)
```

[40]:
$$1$$

1.4 Identidades trigonométricas

Podemos usar expand_trig para expandir funções trigonométricas.

[41]:
```
sy.expand_trig( sy.sin(a + b) ) # sin(a+b)
```

[41]:
$$\sin(a)\cos(b) + \sin(b)\cos(a)$$

[42]:
```
sy.expand_trig( sy.cos(a + b) ) # cos(a+b)
```

[42]:
$$-\sin(a)\sin(b) + \cos(a)\cos(b)$$

[43]:
```
sy.expand_trig( sy.sec(a - b) ) # sec(a-b)
```

[43]:
$$\frac{1}{\sin(a)\sin(b) + \cos(a)\cos(b)}$$

1.5 Propriedades de logaritmo

Com expand_log, podemos aplicar propriedades válidas de logaritmo.

[44]:
```
sy.expand_log( sy.log(a*b) )
```

[44]:
$$\log(ab)$$

A identidade não foi validada pois a e b são símbolos irrestritos.

[45]:
```
a,b = sy.symbols('a b',positive=True) # impomos que a,b > 0
```

[46]:
```
sy.expand_log( sy.log(a*b) ) # identidade validada
```

[46]:
$$\log(a) + \log(b)$$

[47]:
```
sy.expand_log( sy.log(a/b) )
```

[47]:
$$\log(a) - \log(b)$$

[48]:
```
m = sy.symbols('m', real = True) # impomos que m seja um no. real
sy.expand_log( sy.log(a**m) )
```

[48]:
$$m\log(a)$$

Com logcombine, compactamos as propriedades.

[49]:
```
sy.logcombine( sy.log(a) + sy.log(b) ) # identidade recombinada
```

[49]: $\log(ab)$

1.6 Fatorial

A função factorial(n) pode ser usada para calcular o fatorial de um número.

[50]:
```
sy.factorial(m)
```

[50]: $m!$

[51]:
```
sy.factorial(m).subs(m,10) # 10!
```

[51]: 3628800

[52]:
```
sy.factorial(10) # diretamente
```

[52]: 3628800

Exemplo: Sejam m,n,x inteiros positivos. Se $f(m) = 2m!$, $g(n) = \frac{(n+1)!}{n^2!}$ e $h(x) = f(x)g(x)$, qual é o valor de $h(2)$?

[53]:
```
from sympy.abc import m,n,x

f = 2*sy.factorial(m)
g = sy.factorial(n + 1)/sy.factorial(n**2)

h = (f.subs(m,x)*g.subs(n,x)).subs(x,4)
h
```

[53]: $\frac{1}{3632428800}$

1.7 Funções anônimas

A terceira classe de funções que iremos aprender é a de *funções anônimas*. Uma função anônima em Python consiste em uma função cujo nome não é explicitamente definido e que pode ser criada em apenas uma linha de código para executar uma tarefa específica.

Funções anônimas são baseadas na palavra-chave lambda. Este nome tem inspiração em uma área da ciência da computação chamada de cálculo-λ.

Uma função anônima tem a seguinte forma:

```
lambda lista_de_parâmetros: expressão
```

Funções anônimas podem são bastante úteis para tornar um código mais conciso.

Por exemplo, na aula anterior, definimos a função

```
def repasse(V):
    return 0.0103*V
```

para calcular o repasse financeiro ao corretor imobiliário.

Com uma função anônima, a mesma função seria escrita como:

[54]:
```
repasse = lambda V: 0.0103*V
```

Não necessariamente temos que atribui-la a uma variável. Neste caso, teríamos:

[55]:
```
lambda V: 0.0103*V
```

[55]:
```
<function __main__.<lambda>(V)>
```

Para usar a função, passamos um valor:

[56]:
```
repasse(100000) # repasse sobre R$ 100.000,00
```

[56]:
1030.0

O modelo completo com "bonificação" seria escrito como:

[57]:
```
r3 = lambda c,V,b: c*V + b # aqui há 3 parâmetros necessários
```

Redefinamos objetos simbólicos:

[58]:
```
from sympy.abc import b,c,V
r3(b,c,V)
```

[58]:
$V + bc$

O resultado anterior continua sendo um objeto simbólico, mas obtido de uma maneira mais direta. Podemos usar funções anônimas para tarefas de menor complexidade.

1.8 "Lambdificação" simbólica

Usando lambdify, podemos converter uma expressão simbólica do *sympy* para uma expressão que pode ser numericamente avaliada em outra biblioteca. Essa função desempenha papel similar a uma função *lambda* (anônima).

[59]:
```
expressao = sy.sin(x) + sy.sqrt(x) # expressão simbólica
f = sy.lambdify(x,expressao,"math") # lambdificação para o módulo math
f(0.2) # avalia
```

[59]:
0.6458829262950192

Para avaliações simples como a anterior, podemos usar evalf e subs. A lambdificação será útil quando quisermos avaliar uma função em vários pontos, por exemplo. Na próxima aula, introduziremos sequencias e listas. Para mostrar um exemplo de lambdificação melhor veja o seguinte exemplo.

[60]:
```
from numpy import arange # importação de função do módulo numpy

X = arange(40) # gera 40 valores de 0 a 39
```

[61]:
```
X
```

[61]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39])
```

[62]:
```
f = sy.lambdify(x,expressao,"numpy")(X) # avalia 'expressao' em X
f
```

[62]:
```
array([0.        , 1.84147098, 2.32351099, 1.87317082, 1.2431975 ,
       1.2771437 , 2.17007424, 3.30273791, 3.81778537, 3.41211849,
       2.61825655, 2.31663458, 2.9275287 , 4.02571831, 4.73226474,
       4.52327119, 3.71209668, 3.16170813, 3.49165344, 4.50877615,
       5.38508121, 5.41923133, 4.68156445, 3.94961112, 3.99340112,
       4.86764825, 5.86157796, 6.15252835, 5.56240841, 4.72153092,
       4.48919395, 5.16372672, 6.20828093, 6.74447451, 6.36003458,
       5.48789711, 5.00822115, 5.4392244 , 6.46078258, 7.20879338])
```