

O Symbolic Python é um pacote Python que implementa um sistema de computação simbólica (CAS=Computer Algebra System) no ambiente Python. Para entender o que é um sistema deste tipo, acompanhe:

```
comput. numérica: sqrt(8)=2.82842712
comput. simbólica: sqrt(8)=2*sqrt(2)
```

A vantagem da segunda é que é fácil passar da segunda para a primeira, mas não é possível o sentido oposto. Acompanhe:

```
>>> print(sp.sqrt(8))
2*sqrt(2)
>>> print(sp.sqrt(8).n())
2.82842712474619
>>> print(sp.sqrt(8).n(30))
2.82842712474619009760337744842
```

O paradigma de computação simbólica é o Maple. As vantagens de usar o sympy versus o Maple (por exemplo) são pelo menos duas: sympy é de graça (Maple custa 1.000 dólares) e por ser em python admite que você use tudo o que já sabe de python e também permite usar os mais de 135.000 pacotes também de graça.

Instalação e uso O sympy precisa ser instalado. Na versão para windows, você deve localizar o diretório scripts e nele fazer `pip install sympy` em um computador que tenha acesso à Internet. Se quiser usar winpython, nele o sympy já está pré-instalado. Num ou noutro, precisa importar o pacote. Faça `from sympy import *` ou algum comando equivalente. Este pacote exige como dependência o pacote NUMPY. A documentação completa do sympy ocupa hoje 2.127 páginas A4. Sua última versão deve ser buscada na Internet (procure `sympy doc`). Este é um produto relativamente recente na história de Python (2013), e portanto alguma instabilidade pode aparecer. Paciência, nem tudo nasce pronto neste planeta.

Começo

```
>>> import sympy as sp
>>> sp.init_session()
```

Importação Neste material, vai-se executar sempre o comando de importação

```
import sympy as sp
init_session() # estabelece ambiente
```

Iniciando Uma novidade neste pacote é o surgimento de variáveis simbólicas que terão um comportamento diferente das variáveis convencionais do Python. Quem cria variáveis simbólicas é o comando `symbols()`, veja no exemplo

```
>>> x,y,z = sp.symbols('x,y,z')
A partir deste ponto as variáveis x, y e z estarão disponíveis para armazenar expressões simbólicas. Uma definição alternativa, usando ferramentas do Python é
```

```
>>> a0,a1,a2,a3 = sp.symbols('a0:4')
Note que falando estritamente, não é obrigatório que os símbolos tenham o mesmo nome das variáveis, mas para evitar fortes emoções, vamos usar sempre o mesmo nome.
```

Expressões Pode-se definir expressões em Sympy usando símbolos junto com as operações básicas da matemática além de outras funções. Para escrever expressões, deve-se obedecer ao padrão Python (por exemplo $2x$ deve ser escrito como $2*x$, a potência x^2 como $x**2$). Há uma diferença importante em que a igualdade em sympy deve ser escrita como `Eq(a,b)` para

representar $a = b$. Ou melhor ainda, se $a = b$, então $a - b = 0$ e daí pode-se escrever em sympy `a-b`.

Há um método em sympy chamado `equals` que testa 2 expressões em pontos aleatórios e retorna True se todos resultarem iguais.

```
>>> a,b,x=symbols('a,b,x')
>>> a=cos(x)**2-sin(x)**2
>>> b=cos(2*x)
>>> a.equals(b)
True
```

Substituição A substituição é usada para avaliar uma equação num ponto

```
>>> expr
cos(x) + 1
>>> expr.subs(x,pi/2)
1
>>> (x**2+2*x+10).subs(x,3)
25
```

Também é usada para trocar uma subexpressão por outra, como em

```
>>> (x**2+2*x+10).subs(x,y**2)
y**4 + 2*y**2 + 10
```

Às vezes, o uso de métodos primitivos pode levar a resultados indesejados, como em

```
>>> expr=sin(2*x)+cos(2*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(x)**2 - 1
```

Ao querermos a simplificação apenas do seno duplo, faz-se

```
expr.subs(sin(2*x),2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*x)
```

Note-se que o método subs devolve outra instância de expressão já que a expressão original permanece imutável. A substituição pode ser de mais de uma variável/expressão

```
>>> expr=3*x**3+4*y**2-2*z
>>> expr.subs([(x,3),(y,5),(z,1)])
179
```

Simplificação Não confundir com simplificação. Trata-se de transformar uma string em uma expressão sympy

```
>>> a="x**2+3"
>>> a
'x**2+3'
>>> b=sympify(a)
>>> b
x**2 + 3
>>> b.subs(x,1)
4
```

Impressão Há vários dispositivos em sympy. `pprint()` imprime a saída na tela. Há `latex` e `mathml`, entre outros.

```
>>> pprint(Integral(sqrt(1/x),x))
```

$$\int \sqrt{\frac{1}{x}} dx$$

```
>>> latex(Integral(sqrt(1/x),x))
'\\int \\sqrt{\\frac{1}{x}}\\, dx'
```

Simplificação Sympy tem dúzias de funções para fazer diversos tipos de simplificação. Há uma função principal chamada `simplify()` que aplica todas estas simplificações escolhendo depois a mais simples. Isto nem sempre funciona bem, já que a expressão *a mais simples* não está claramente definida.

```
>>> simplify(sin(x)**2+cos(x)**2)
1
>>> simplify((x**3+x**2-x-1)/(x-1))
x**2 + 2*x + 1
>>> simplify(x**2+2*x+1)
x**2 + 2*x + 1
```

expand() e factor() Para este último caso, pode-se escolher `expand()` ou `factor()` que são dois simplificadores importantes em sympy.

```
>>> expand((x+1)**3)
x**3 + 3*x**2 + 3*x + 1
>>> factor(x**3+3*x**2+3*x+1)
(x + 1)**3
```

A expansão de um polinômio coloca-o na forma de uma soma de monômios. Já fatoração transforma a soma de monômios em uma multiplicação de fatores irredutíveis. Note-se que a entrada de ambos não precisa ser um polinômio estrito senso.

```
>>> init_printing()
>>> expand((cos(x)+sin(x))**2)
sin(x)**2 + 2*sin(x)*cos(x) + cos(x)**2
```

collect() Outro simplificador é `collect()` que junta potências comuns dentro da expressão.

```
collect(x*y+x-3+2*x**2-z*x**2+x**3,x)
x**3 + x**2*(2 - z) + x*(y + 1) - 3
```

cancel() `cancel()` coloca qualquer função racional na forma canônica $\frac{p}{q}$ onde p e q são polinômios expandidos sem fator comum e com fatores inteiros.

```
>>> e=symbols('e')
>>> e=1/x+(3*x/2-2)/(x-4)
>>> cancel(e)
```

$$\frac{3.x^2 - 2.x - 8}{2.x^2 - 8.x}$$

apart() `apart()` executa uma decomposição de frações parciais de uma função racional.

```
>>> e=(x**3+x**2+1)/(x**4+x**2)
```

$$\frac{x}{x^2 + 1} + \frac{1}{x^2}$$

trigsimp() Simplifica usando identidades trigonométricas circulares e hiperbólicas.

```
>>> trigsimp(sinh(x)/tanh(x))
cosh(x)
```

expand_trig() Expande funções trigonométricas, ou seja, aplica as identidades de ângulos duplos ou soma de ângulos

```
>>> expand_trig(sin(x+y))
sin(x)*cos(y) + sin(y)*cos(x)
>>> expand_trig(tan(2*x))
```

$$\frac{2.tan(x)}{1 - tan(x)^2}$$

Potência As potências obedecem às seguintes identidades:

- $x^a x^b = x^{a+b}$
- $x^a y^a = (xy)^a$ só é verdade quando $x, y \geq 0$ e $a \in \mathbb{R}$
- $(x^a)^b = x^{ab}$ só é verdade quando $b \in \mathbb{Z}$

Veja-se para as regras 2 e 3: $(-1)^{1/2}(-1)^{1/2} \neq (-1 - 1)^{1/2}$ e $((-1)^{1/2})^2 \neq (-1)^{2.1/2}$ Esta observação é importante pois o sympy só aplica simplificações quando elas são sempre verdadeiras. Assim, as regras 2 e 3 só serão aplicadas se expressamente suas pré-condições forem explicitamente declaradas.

powsimp() Aplica as identidades 1 e 2 da esquerda para a direita

```
>>> powsimp(x**a*x**b)
a + x
b
```

expand_power_exp(), expand_power_base() e powdenest() Aplicam as identidades acima.

Logaritmos Têm tratamento similar às potências. Há 2 identidades:

1. $\log(xy) = \log(x) + \log(y)$
2. $\log(x^n) = n \cdot \log(x)$

Nenhuma dessas 2 é verdadeira para $x, y \in \mathbb{C}$. Para serem verdadeiras, $x, y \geq 0$ e $n \in \mathbb{R}$.

expand_log() Aplica as identidades 1 e 2 da esquerda para a direita.

```
>>> expand_log(log(x*y))
log(x*y)
>>> expand_log(log(x*y), force=True)
log(x) + log(y)
```

logcombine() aplica as identidades 1 e 2 da direita para a esquerda.

```
>>> logcombine(log(x)+log(y))
log(x) + log(y)
>>> logcombine(log(x)+log(y),
force=True)
log(x*y)
```

factorial(n) e **binomial(n,k)**

rewrite() Permite reescrever uma função em termos de outra

```
>>> tan(x).rewrite(cos)
```

$$\frac{\cos(x - \frac{\pi}{2})}{\cos(x)}$$

diff() Acha a derivada de uma função. Seu formato **diff(função, variável)**.

```
>>> diff(exp(x**2), x)
```

$$2x \cdot e^{x^2}$$

```
>>> diff(tan(x), x)
2
tan(x) + 1
```

Para conseguir derivadas múltiplas, passe a variável diversas vezes, ou a variável seguida da ordem da derivada.

```
>>> diff(x**4, x, x, x)
24.x
>>> diff(x**4, x, 3)
24.x
```

Derivative() estabelece literalmente a derivada, tendo o mesmo formato de **diff()**. Para forçar o cálculo posterior, chame **doit()**.

```
>>> e=exp(x)+sin(x)
>>> e
```

$$e^x + \sin(x)$$

```
>>> diff(e, x)
```

$$e^x + \cos(x)$$

```
>>> Derivative(e)
```

$$\frac{d}{dx} e^x + \sin(x)$$

```
>>> Derivative(e).doit()
```

$$e^x + \cos(x)$$

integrate() Há 2 tipos de integrais: a definida e a indefinida. Para a integral indefinida é **integrate(função, variável)**.

```
>>> integrate(cos(x))
sin(x)
```

Note-se que não aparece a constante aditiva. Se ela for necessária deve ser manualmente incluída ou então o problema deve ser redesenhado para ser resolvido como uma equação diferencial.

Para computar uma integral definida, o formato é **integrate(funç, (var, linf, lsup))**

```
>>> integrate(exp(x), (x, 0, oo))
oo
>>> integrate(exp(-x), (x, 0, oo))
1
>>> integrate(exp(-x**2-y**2),
(x, -oo, oo), (y, -oo, oo))
pi
```

Também há **Integral()** e depois para implementá-lo, chame **doit()**.

```
>>> e=log(x)**2
>>> Integral(e, x)
```

$$\int \log(x)^2 dx$$

```
>>> Integral(e, x).doit()
2
x*log(x) - 2*x*log(x) + 2*x
```

limit() Seja o limite $\lim_{x \rightarrow x_0} f(x)$

```
>>> limit(sin(x)/x, x, 0)
1
```

$$\lim_{x \rightarrow 0+} \frac{\cos(x) - 1}{x}$$

```
>>> Limit((cos(x)-1)/x, x, 0).doit()
0
```

Note-se que limites à esquerda e a direita, exigem um quarto parâmetro que pode ser '+' ou '-' respectivamente.

series() Permite a expansão de séries em torno a um ponto. Para computar a expansão de $f(x)$ em torno do ponto $x = x_0$ em termos de x^n , usa-se **f(x).series(x, x0, n)**. x_0 e n podem ser omitidos, no caso em que serão assumidos $x_0 = 0$ e $n = 6$.

```
>>> exp(sin(x)).series(x, 0, 4)
```

$$1 + x + \frac{x^2}{2} + O(x^4)$$

solveset() A função principal para solução de equações. Sua sintaxe é **solveset(equação, variável, domínio)**. A equação $x = y$ deve ser escrita usando **Eq(x, y)** ou mais facilmente como **x-y**.

```
>>> solveset(x**2-x, x)
{0, 1}
>>> solveset(sin(x)-1, x, domain=Reals)
```

$$2n\pi + \frac{\pi}{2} \in \mathbb{Z}$$

linsolve() Resolve sistemas de equações lineares. Seu formato **linsolve([eq1, eq2, ...], (v1, v2, ...))**.

Seja o sistema: $2x + 3y - z = 4$, $x + y + z = 7$ e $x - 2y + 3z = 9$

```
>>> linsolve([2*x+3*y-z-4, x+y+z-7,
x-2*y+3*z-9], (x, y, z))
{(1, 2, 4)}
>>> linsolve(Matrix([[2, 3, -1, 4],
[1, 1, 1, 7], [1, -2, 3, 9]]), (x, y, z))
{(1, 2, 4)}
```

nonlinsolve() Resolve sistemas não lineares.

```
>>> nonlinsolve([x**2+y-z+1, 2*x+y+z-8,
x-2*y+3*z-9], (x, y, z))
{(-1, 4, 6), (1, 2, 4)}
```

dsolve() Resolve equações diferenciais. Para tanto deve-se criar funções

```
>>> f,g=symbols('f,g', cls=Function)
>>> f(x)
f(x)
>>> f(x).diff(x)
d
--(f(x))
dx
Supondo a equação diferencial  $f''(x) - 2f'(x) + f(x) = \sin(x)$  deve-se fazer
>>> dx=(f(x).diff(x,x)-
2*f(x).diff(x)+f(x)-sin(x))
>>> dx
```

$$f(x) - \sin(x) - 2 \cdot \frac{d}{dx}(f(x)) + \frac{d^2}{dx^2}(f(x))$$

```
>>> dsolve(dx, f(x))
```

$$f(x) = (C_1 + C_2 \cdot x) \cdot e^x + \frac{\cos(x)}{2}$$

Matrizes Criação de matrizes podem ser feitas com **Matrix([[l1],[l2],...])**, com **eye(ordem)**, **zeros(lin,col)**, **ones(lin,col)** e **diag(vetor)** em que o vetor é colocado na diagonal.

Matrizes podem ser operadas como em **M+N**, **3*M**, **M**2** e **M**-1** (esta última traz a matriz inversa).

```
>>> M=Matrix([[1,2,3],[1,2,1],[1,0,6]])
>>> M
[1 2 3]
[1 2 1]
[1 0 6]
>>> M**-1
[-3 3 1]
[5/4 -3/4 -1/2]
[1/2 -1/2 0]
```

det() calcula o determinante de uma matriz

```
>>> M.det()
-4
```

rref() Calcula a forma reduzida por linhas, retornando uma tupla de 2 elementos. O primeiro é a matriz reduzida por linhas e o segundo os índices das colunas pivot.

```
>>> N=Matrix([[2,3,-1,4],[1,1,1,7],
[1,-2,3,9]])
>>> N
[2 3 -1 4]
[1 1 1 7]
[1 -2 3 9]
>>> N.rref()
[1 0 0 1]
([0 1 0 2], (0, 1, 2))
[0 0 1 4]
```