Aula 2A - Computação Simbólica

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Junho de 2020

1 Introdução à Computação Simbólica com *sympy*

1.1 Motivação

Neste ponto do curso, você já aprendeu a realizar operações matemáticas elementares com Python. Por exemplo, se lhe for dado o valor do raio R, você consegue facilmente computar a área πR^2 de um círculo. Todavia, o valor de π que você obtém é finito. As instruções abaixo verificam isto.

```
from math import pi
print(pi)
```
```
3.141592653589793
```

Pense, no entanto, se você pudesse realizar o cálculo desta área de maneira "exata". Matematicamente falando, é impossível fazer isto pois π é um número irracional – o imbróglio desta constante é longo na história da Matemática. Porém, a computação simbólica permite que operemos com π como se fosse simplesmente um símbolo com precisão infinita. Embora 15 casas decimais, como o valor exemplificado acima, sejam suficientes para a maioria dos cálculos do mundo real, a computação simbólica permite que trabalhemos com modelos abstratos que servem a uma diversidade de propósitos.

Aliás, quando se diz que 3.141592653589793 é um valor razoavelmente aceitável, isto é verdade até mesmo para cálculos em escala astronômica. A equipe de engenharia da NASA explica que, usando este valor para calcular o perímetro de uma circunferência com diâmetro igual a 25 bilhões de milhas, o erro de cálculo é próximo de 1,5 polegada [NASA]. Até aí, nada mal para uma aproximação!

1.2 O que é Computação Simbólica e para que serve?

Computação Simbólica (CS) é uma subárea de estudo da matemática e da ciência da computação que se preocupa em resolver problemas usando objetos simbólicos representáveis em um computador. Esses problemas surgem em muitas aplicações em ciências naturais, pesquisa básica, na indústria e principalmente no desenvolvimento de softwares para computação avançada denominados *sistemas de computação algébrica* (SCAs).

A CS existente em um SCA é aplicada em álgebra computacional, projetos assistidos por computação (CAD), raciocínio automatizado, gestão do conhecimento, lógica computacional e sistemas formais de verificação. O desenvolvimento da CS depende da integração de basicamente três campos: *softwares matemáticos*, *álgebra computacional* e *lógica computacional* [RISC/JKU]. Em casos

mais avançados, a CS é útil para solucionar equações da macroeconomia, manipular números para a finalidade de criptografia e criar modelos probabilísticos [Kotzé], [Cohen].

1.3 Principais SCAs

Alguns SCAs são populares de longa data, tais como Maple, Mathematica e MuPad. Entretanto, são comerciais e costumam ter licenças custosas, embora ofereçam versões com desconto para estudantes. Algumas alternativas robustas de uso livre são Scilab, Sagemath, Octave e o próprio módulo *sympy*. Uma lista completa de SCAs está disponível na [Wikipedia].

1.4 Por que *sympy*?

O objetivo principal do *sympy* é ser uma biblioteca de manipulação simbólica para Python. Ele começou a ser desenvolvido em 2006 e atualmente está na versão 1.5.1, lançada em dezembro de 2019 na página oficial [sympy.org]. As principais características do módulo são as seguintes:

- é gratuito;
- é baseado inteiramente em Python;
- é leve e independente.

1.5 Objetos numéricos x objetos simbólicos

Importaremos os módulos `math` e `sympy` para ver algumas diferenças entre objetos numéricos e simbólicos.

[1]:
```
import math as mt
import sympy as sy
sy.init_printing(pretty_print=True) # melhor impressão de símbolos
```

[2]:
```
mt.pi # numérico
```

[2]:
3.141592653589793

[3]:
```
sy.pi # simbólico
```

[3]:
π

Verifiquemos com type.

[4]:
```
type(mt.pi)
```

[4]:
```
float
```

[5]:
```
type(sy.pi) # é um objeto simbólico
```

[5]:
```
sympy.core.numbers.Pi
```

Vejamos mais um exemplo:

[6]:
```
mt.sqrt(2)
```

[6]: 1.4142135623730951

[7]:
```
sy.sqrt(2)
```

[7]: $\sqrt{2}$

[8]:
```
type(mt.sqrt(2))
```

[8]: float

[9]:
```
type(sy.sqrt(2)) # é um objeto simbólico
```

[9]: sympy.core.power.Pow

1.5.1 Função x método

Na aula anterior destacamos que print e type são ‘‘funções’’ similares àquelas do tipo $y = f(x)$ em Matemática. Em Python, essas ‘‘funções’’ recebem o nome de *função* mesmo.

Porém, há módulos que possuem *métodos*, que para seu entendimento, podem ser vistos como ‘‘funções’’ também. Porém, *função* e *método* são conceitos levemenete distintos.

Para aplicarmos funções usamos parênteses envolvendo um ou mais *parâmetros*.

No caso acima, mt.sqrt(2) mostra que sqrt age como uma função e o número 2 é seu único parâmetro. Note, além disso, que o sympy também possui a sua própria função sqrt, que é de uma natureza distinta. Ela é um objeto sympy.core.power.Pow. Não precisamos entender isso agora, mas basta saber que ela pertence a um submódulo do *sympy*.

Por outro lado, também aprendemos que o conjugado de um número complexo z (tipo complex) pode ser obtido como z.conjugate(). Esta forma ‘‘sem parâmetros’’ indica que conjugate é um método do objeto z.

A partir deste ponto, poderemos ver situações como as seguintes:

- f(x): a função f é aplicada ao parâmetro x
- a.f(): f é um método sem parâmetro do objeto a
- a.f(x): f é um método com parâmetro x do objeto a

A partir do último exemplo, podemos dizer que um método é, na verdade, uma função que pertence a um objeto.

1.5.2 Atribuições com símbolos

Podemos atribuir símbolos a variáveis usando a função symbols.

[10]:
```
x = sy.symbols('x')
y = sy.symbols('y')
```

x e y são símbolos sem valor definido.

[11]:
```
x
```

[11]: x

[12]:
```
y
```

[12]: y

Podemos operar aritmeticamente com símbolos e obter uma expressão simbólica como resultado.

[13]:
```
z = sy.symbols('z')
x*y + z**2/3 + sy.sqrt(x*y - z)
```

[13]: $xy + \frac{z^2}{3} + \sqrt{xy - z}$

Exemplo: escreva o produto notável $(x-y)^2$ como uma expressão simbólica.

[14]:
```
x**2 - 2*x*y + y**2
```

[14]: $x^2 - 2xy + y^2$

Note que o nome da variável não tem a ver com o nome do símbolo. Poderíamos fazer o seguinte:

[15]:
```
y = sy.symbols('x') # y é variável; x é símbolo
y
```

[15]: x

1.5.3 Atribuição por desempacotamento

Também poderíamos realizar as atribuições anteriores da seguinte forma:

[16]:
```
x, y, z = sy.symbols('x y z')
```

1.5.4 Alfabeto de símbolos

O *sympy* dispõe de um submódulo chamado abc do qual podemos importar símbolos para letras latinas (maiúsculas e minúsculas) e gregas (minúsculas).

[17]:
```
from sympy.abc import a,b,c,alpha,beta,gamma
(a + 2*b - 3*c)*(alpha/3 + beta/2 - gamma) # símbolico
```

[17]: $(a + 2b - 3c)\left(\frac{\alpha}{3} + \frac{\beta}{2} - \gamma\right)$

[18]:
```
from sympy.abc import D,G,psi,theta
D**a * G**b * psi**c * theta**2 # símbolico
```

[18]: $D^a G^b \psi^c \theta^2$

Nota: algumas letras já são usadas como símbolos especiais, tais como O, que indica "ordem" e I, que é o complexo i. Neste caso, cuidado deve ser tomado com nomes de variáveis

[19]:
```
sy.I # imaginário simbólico
```

[19]: i

[20]:
```
type(sy.I)
```

[20]:
```
sympy.core.numbers.ImaginaryUnit
```

1.5.5 Símbolos com nomes genéricos

Para criar símbolos genéricos, temos de usar symbols ou Symbol.

[21]:
```
sem_nocao = sy.symbols('nada')
sem_nocao
```

[21]: $nada$

[22]:
```
muito_louco = sy.Symbol('massa')
muito_louco
```

[22]: $massa$

1.5.6 Variáveis e símbolos

[23]:
```
sem_medo = sem_nocao + 2
sem_medo
```

[23]: $nada + 2$

[24]:
```
soma = muito_louco + 2
muito_louco = 3 # 'muito_louco' aqui não é o simbólico
soma
```

[24]: $massa + 2$

1.6 Substituição

A operação de *substituição* permite que:

1. substituamos variáveis por valores numéricos para avaliar uma expressão ou calcular valores de uma função em um dado ponto.
2. substituamos uma subexpressão por outra.

Para tanto, procedemos da seguinte forma:

expressao.subs(variavel,valor)

Exemplo: considere o polinômio $P(x) = 2x^3 - 4x - 6$. Calcule o valor de $P(-1)$, $P(e/3)$, $P(\sqrt{3.2})$.

[25]:
```
from sympy.abc import x
P = 2*x**3 - 4*x - 6
P1 = P.subs(x,-1)
Pe3 = P.subs(x,mt.e/3)
P32 = P.subs(x,mt.sqrt(3.2))
print(P1, Pe3, P32)
```

```
-4 -8.13655822141297 -1.70674948320040
```

Exemplo: sejam $f(x) = 4^x$ e $g(x) = 2x - 1$. Compute o valor da função composta $f(g(x))$ em $x = 3$.

[26]:
```
f = 4**x
fg = f.subs(x,2*x - 1)
```

[27]:
```
fg.subs(x,3)
```

[27]:
1024

Poderíamos também fazer isso com um estilo "Pythônico":

[28]:
```
fg = 4**x.subs(x,2*x - 1).subs(x,3)
fg
```

[28]:
1024

Exemplo: se $a(x) = 2^x$, $b(x) = 6^x$ e $c(x) = \cos(x)$, compute o valor de $a(x)b(c(x))$ em $x = 4$

[29]:
```
a = 2**x
b = 6**x
c = sy.cos(x)
(a * b.subs(x,c)).subs(x,4)
```

[29]:
$$\frac{16}{6^{-\cos(4)}}$$

Ou, de modo direto:

[30]:
```
valor = ( 2**x * ( 6**x.subs(x,sy.cos(x))) ).subs(x,4)
valor
```

[30]:
$$\frac{16}{6^{-\cos(4)}}$$

1.6.1 Avaliação de expressão em ponto flutuante

Note que a expressão anterior não foi computada em valor numérico. Para obter seu valor numérico, podemos usar o método evalf.

[31]:
```
valor.evalf()
```

[31]: 4.96005373851859

Precisão arbitrária evalf permite que escolhamos a precisão do cálculo impondo o número de dígitos de precisão. Por exemplo, a última expressão com 20 dígitos de precisão seria:

[32]:
```
valor.evalf(20)
```

[32]: 4.9600537385185864965

Com 55, seria:

[33]:
```
valor.evalf(55)
```

[33]: 4.960053738518586496500224737177413703698642291256987299

E com 90 seria:

[34]:
```
valor.evalf(90)
```

[34]: 4.960053738518586496500224737177413703698642291256987298543441771600766029871565358403786040

Exemplo: calcule o valor de e com 200 dígitos de precisão.

[35]:
```
sy.exp(1).evalf(200)
```

[35]: 2.71828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642743

1.7 Funções predefinidas x funções regulares

Vamos apresentar aqui três grupos de funções que podem ser criadas em Python para nos auxiliar ao longo do curso sem, no entanto, nos aprofundaremos nos detalhes de cada um.

Como dissemos em um momento anterior, a linguagem Python possui um *core* que contém um conjunto de funções já prontas que podemos usar, como é o caso de print(), type() e até mesmo int() e float() para operações de *casting*. Essas funções podem ser chamadas de predefinidas (*built-in functions*). Ou seja, são aquelas funções "já existentes". Exemplos adicionais seriam as funções do módulo math.

Suponhamos, porém, que você vasculhe módulos e mais módulos atrás de uma função que faça exatamente o que você quer, mas não a encontra. O que você faz? Você a cria! Podemos fazer isto de uma maneira usando uma "palavra-chave" (*keyword*) chamada def da seguinte forma:

```
def f(x):
    (...)
    return y
```

A instrução acima permite que você crie uma *função* chamada f da qual x é um *argumento* e y é um *valor de retorno*, indicado por uma segunda "palavra-chave", *return*. Funções definidas por você dessa maneira são chamadas de regulares, *normais* - pelo fato de serem programadas de um modo regular, seguindo a "normalidade" da linguagem -, ou ainda *definidas pelo usuário* (do inglês *user-defined functions*, ou simplesmente *UDF*). Por conveniência, vamos nos referir a elas por este acrônimo elegante: UDF.

Uma UDF permite que você abstraia seu pensamento para criar basicamente o que quiser dentro dos limites da linguagem Python. Cabe, apesar disso, fazermos as seguintes ressalvas:

- uma UDF pode ter zero ou mais argumentos, tantos quantos se queira;
- uma UDF pode ou não ter valor de retorno;

Vamos entender as UDFs com exemplos.

Exemplo: Suponha que você é um(a) analista de dados do mercado imobiliário e está estudando o impacto do repasse de comissões pagas a corretores mediante vendas de imóveis. Você, então, começa a raciocinar e cria um modelo matemático bastante simples que, antes de tudo, precisa calcular o valor do repasse a partir do preço de venda.

Se c for o percentual de comissão, V o valor da venda do imóvel e r o valor a ser repassado para o corretor, então, a função a ser definida é

$$r(V) = c\,V,$$

assumindo que c seja um valor fixo.

Digamos que c corresponda a 1.03% do valor da venda do imóvel. Neste caso podemos criar uma UDF para calcular r para nós da seguinte forma:

[36]:
```
def repasse(V):
    r = 0.0103*V
    return r
```

Para $V = R\$\,332.130,00$:

[37]:
```
repasse(332130)
```

[37]:
3420.939

O que é necessário observar:

- def vem seguido pelo *nome* da função (repasse) após um espaço;
- o nome precede os argumentos, enclausurados por parênteses (V). Neste caso, temos apenas um *argumento*, que é V;

- após o nome, os dois-pontos (:) são obrigatórios e significam mais ou menos "o que esta função faz será definido da seguinte maneira"
- a instrução r = 0.0103*V é o *escopo* da função, que deve ser escrito em uma ou mais linhas indentadas (pressione TAB para isso, ou use 4 espaços)
- o valor de retorno, se houver, é posto na última linha do escopo.

Podemos atribuir os valores do argumento e resultado a variáveis:

[38]:
```
V = 332130
rep = repasse(V)
rep
```

[38]:
3420.939

Nomes iguais de variável e função são permissíveis.

[39]:
```
repasse = repasse(V) # 'repasse' à esquerda é uma variável; à direita, função
print(repasse)
```

3420.939

Todavia, isto pode ser confuso e é bom evitar.

O estilo "Pythônico" de escrever permite que o valor de retorno não seja explicitamente declarado. No escopo

```
...
    r = 0.0103*V
    return r
```

a variável r não é necessária.

Python é inteligente para permitir o seguinte:

[40]:
```
def repasse(V):
    return 0.0103*V

# note que aqui não indentamos a linha.
# Logo esta instrução NÃO pertence ao escopo da função.
repasse(V)
```

[40]:
3420.939

Podemos criar uma função para diferentes valores de c e V usando *dois* argumentos:

[41]:
```
def repasse_c(c,V): # esta função tem outro nome
    return c*V
```

[42]:
```
c = 0.0234 # equivaleria a uma taxa de repasse de 2.34%
V = 197432 # o valor do imóvel agora é R$ 197.432,00
repasse_c(c,V)
```

[42]:
4619.9088

A ordem dos argumentos importa:

[43]:
```
V = 0.0234 # este deveria ser o valor de c
c = 197432 # este deveria ser o valor de V
repasse_c(c,V)
```

[43]:
4619.9088

Por que o valor resultante é o mesmo? Porque a operação no escopo da função é uma multiplicação, c*V, que é comutativa independentemente do valor das variáveis. Porém, digamos que um segundo modelo tenha uma forma de cálculo distinta para a comissão dada por

$$r_2(V) = c^{3/5}\, V$$

Neste caso:

[44]:
```
def repasse_2(c,V):
    return c**(3/5)*V

V = 197432
c = 0.0234

repasse_2(c,V)
```

[44]:
20746.606005253696

Porém, se trocarmos o valor das variáveis, a função repasse_2 calculará um valor distinto. Embora exista um produto também comutativo, o expoente 3/4 modifica apenas o valor de c.

[45]:
```
# variáveis com valores trocados
c = 197432
V = 0.0234

repasse_2(c,V)
```

[45]:
35.19381804734109

A ordem com que escrevemos os argumentos tem importância relativa aos valores que passamos e ao que definimos:

[46]:
```
# variáveis com valores corretos
V = 197432
c = 0.0234

def repasse_2_trocada(V,c): # V vem antes de c
    return c**(3/5)*V

repasse_2_trocada(V,c)
```

[46]:
20746.606005253696

Mas,

[47]:
```
# os valores das variáveis estão corretos,
# mas foram passados para a função na ordem errada
repasse_2_trocada(c,V)
```

[47]:
35.19381804734109

e

[48]:
```
# a ordem dos argumentos está de acordo com a que foi definida
# mas os valores das variáveis foram trocados
V = 197432
c = 0.0234
repasse_2_trocada(c,V)
```

[48]:
35.19381804734109

1.8 Modelos matemáticos simbólicos

A partir do que aprendemos, podemos definir modelos matemáticos completamente simbólicos.

[49]:
```
from sympy.abc import c,V

def repasse_2_simbolica(c,V):
    return c**(3/5)*V
```

Se chamarmos esta função, ela será um objeto simbólico.

[50]:
```
repasse_2_simbolica(c,V)
```

[50]:
$Vc^{0.6}$

Atribuindo em variável:

[51]:
```
rep_simb = repasse_2_simbolica(c,V)
```

[52]:
```
type(rep_simb) # é um objeto simbólico
```

[52]:
```
sympy.core.mul.Mul
```

Exemplo: Suponha, agora, que seu modelo matemático de repasse deva considerar não apenas um percentual c pré-estabelecido, mas também um valor de "bônus" adicional concedido como prêmio pela venda do imóvel. Considere, então, que o valor deste bônus seja b. Diante disso, nosso novo modelo teria uma fórmula como a seguinte:

$$r_3(V) = c\,V + b$$

Simbolicamente:

[53]:
```
# importaremos apenas o símbolo b,
# uma vez que c e V já foram importados
# como símbolos anteriormente
from sympy.abc import b

def r3(V):
    return c*V + b

rep_3 = r3(V)
rep_3
```

[53]: $Vc + b$

1.8.1 Substituindo valores

Podemos usar a função subs para atribuir quaisquer valores para o modelo.

Exemplo: $c = 0.119$

[54]:
```
rep_3.subs(c,0.119) # substituindo para c
```

[54]: $0.119V + b$

Exemplo: $c = 0.222$

[55]:
```
rep_3.subs(c,0.222) # substituindo para c
```

[55]: $0.222V + b$

Exemplo: $c = 0.222$ e $b = 12.0$

[56]:
```
rep_3.subs(c,0.222).subs(b,12.0) # substituindo para c, depois para b
```

[56]: $0.222V + 12.0$

1.8.2 Substituição múltipla

O modo anterior de substituição não é "Pythônico". Para substituirmos mais de uma variável de uma vez, devemos usar *pares ordenados* separados por vírgula sequenciados entre colchetes como uma *lista*. Mais tarde, aprenderemos sobre pares ordenados e listas.

Exemplo: Modifique o modelo r_3 para que $c = 0.043$ e $b = 54.0$

[57]:
```
# espaços foram adicionados para dar legibilidade
rep_3.subs( [ (c,0.043), (b,54.0) ] )
```

[57]: $0.043V + 54.0$

Pares ordenados Em matemática, o conceito de par ordenado pode ser definido pelo conjunto:

$$X \times Y = \{(x,y); x \in X \text{ e } y \in Y\},$$

onde X e Y são conjuntos quaisquer e x e y são as *coordenadas*. Por exemplo, se $X = Y = \mathbb{R}$, o conjunto acima contém elementos do tipo $(3,2)$, $(-1,3)$, $(\pi, 2.18)$ etc. Na verdade, eles formam o conjunto $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$, que é exatamente o *plano cartesiano*.

Logo, a substituição múltipla com subs ocorre da seguinte forma;

- a primeira coordenada é o *símbolo*;
- a segunda coordenada é o *valor* que você quer dar para o símbolo.

Exemplo: Calcule $r_3(V)$ considerando $c = 0.021$, $b = 34.0$ e $V = 432.000$.

[58]:
```
# armazenaremos o valor na variável 'valor'
valor = r3(V)

# subsituição
valor.subs( [ (c,0.021), (b,54.0) ] )
```

[58]: $0.021V + 54.0$

Com o estilo "Pythônico":

[59]:
```
valor = r3(V).subs( [ (c,0.021), (b,54.0) ] ) #
valor
```

[59]: $0.021V + 54.0$

Podemos seguir esta regra de pares para substituir todos os valores de um modelo simbólico genérico não necessariamente definido através de uma função. Veja o exemplo aplicado a seguir.

1.9 Exemplo de aplicação: o índice de caminhabilidade

Estudos empíricos nos EUA mostraram que a *caminhabilidade* de uma vizinhança impacta substancialmente os preços das casas. A caminhabilidade está relacionada à distância da moradia a locais de amenidades, tais como restaurantes, bares, bibliotecas, mercearias etc.

O *índice de caminhabilidade* W para uma vizinhança de casas é uma medida matemática que assume valores no intervalo $[0,1]$. A fórmula é definida por:

$$W(d) = e^{-5\left(\frac{d}{M}\right)^5},$$

onde d é a distância medida entre a vizinhança (0 metro) e um dado ponto de referência, e M é a distância máxima de avaliação considerada a partir da qual a caminhabilidade é assumida como nula. Ou seja,

- quando estamos na vizinhança, $d = 0$, $W = 1$ e a caminhabilidade é considerada ótima.
- à medida que nos afastamos da vizinhança em direção ao local da amenidade, d aumenta e o valor W decai vertiginosamente até atingir o valor limite M a partir do qual $W = 0$ e a caminhabilidade é considerada "péssima".

O índice de caminhabilidade é, portanto, calculado com relação a um ponto de destino definido e a distância deve levar em consideração as vias de circulação (ruas, rodovias etc) e não a distância mais curta (raio do perímetro). Por exemplo, se a distância máxima a ser considerada para a caminhabilidade for $M = 500\,m$, um bar localizado a 100 metros da vizinhança teria um índice de caminhabilidade maior do que o de uma farmácia localizada a 300 m e muito maior do que o de um shopping localizado a 800 m, ainda que muito famoso. Aliás, neste caso, o valor de W para o shopping seria zero, já que 800 m está além do limite M estabelecido.

Fonte: *De Nadai, M. and Lepri, B. [The economic value of neighborhoods: Predicting real estate prices from the urban environment]*.

1.9.1 Modelo simbólico

Podemos modelar W simbolicamente e calcular seu valor para diferentes valores de d e M usando a substituição múltipla.

[60]:
```
from sympy.abc import d,M,W

W = sy.exp(-5*(d/M)**5) # função exponencial simbólica
W
```

[60]:
$$e^{-\frac{5d^5}{M^5}}$$

Exemplo: A nossa corretora de imóveis gostaria de entender a relação de preços de imóveis para o Condomínio Pedras de Marfim. Considerando $M = 1km$, calcule:

- o índice de caminhabilidade W_1 em relação à farmácia Dose Certa, localizada a 222 m do condomínio.
- o índice de caminhabilidade W_2 em relação ao restaurante Sabor da Arte, localizada a 628 m do condomínio.
- o índice de caminhabilidade W_3 em relação ao Centro Esportivo Physicalidade, localizada a 998 m do condomínio.
- o índice de caminhabilidade W_4 em relação à Padaria Dolce Panini, localizada a 1,5 km do condomínio.

[61]:
```
# note que 1 km = 1000 m
W1 = W.subs([ (d,222), (M,1000) ])
W2 = W.subs([ (d,628), (M,1000) ])
W3 = W.subs([ (d,998), (M,1000) ])
W4 = W.subs([ (d,1500), (M,1000) ])
```

Perceba, entretanto, que os valores calculados ainda não são numéricos, como esperado.

[62]:
```
W1
```

[62]: $e^{-\frac{16850581551}{6250000000000}}$

[63]:
```
W2
```

[63]: $e^{-\frac{95388992557}{195312500000}}$

[64]:
```
W3
```

[64]: $e^{-\frac{30938747502499}{6250000000000}}$

[65]:
```
W4
```

[65]: $e^{-\frac{1215}{32}}$

Lembre-se que podemos usar evalf para calcular esses valores. Faremos isso considerando 3 casas decimais.

[66]:
```
# reatribuindo todos os valores
W1n = W1.evalf(3)
W2n = W2.evalf(3)
W3n = W3.evalf(3)
W4n = W4.evalf(3)

print('W1 =', W1n, '; ' \
      'W2 =', W2n, '; ' \
      'W3 =', W3n, '; ' \
      'W4 =', W4n)
```

W1 = 0.997 ; W2 = 0.614 ; W3 = 0.00708 ; W4 = 3.24e-17

Como era de se esperar, os valores decaem de 0.997 a 3.24e-17, que é um valor considerado nulo em termos de aproximação numérica.

**Quebrando instruções com ** A contra-barra \ pode ser usada para quebrar instruções e continuá-las nas próximas linhas, porém não poderá haver nenhum caracter após ela, nem mesmo espaços. Caso contrário, um erro será lançado.

[67]:
```
print('Continuando' \
      'na linha abaixo')
```

```
Continuandona linha abaixo
```

[68]:
```
# neste exemplo, há um caracter de espaço após \
print('Continuando' \
      'na linha abaixo')
```

```
  File "<ipython-input-68-322bd86bfa93>", line 2
    print('Continuando' \
                        ^
SyntaxError: unexpected character after line continuation character
```

1.9.2 O tipo `bool`

Em Python, temos mais um tipo de dado bastante útil, o bool, que é uma redução de "booleano". Objetos bool, que têm sua raiz na chamada Álgebra de Boole, são baseados nos conceitos *true* (verdadeiro) e *false*, ou *0* e *1* e são estudados em algumas disciplinas, tais como Circuitos Lógicos, Matemática Discreta, Lógica Aplicada, entre outras.

Aprenderemos sobre operadores lógicos mais à frente. Por enquanto, cabe mencionar as entidades fundamentais True e False.

[69]:
```
True
```

[69]:
```
True
```

[70]:
```
False
```

[70]:
```
False
```

[71]:
```
type(True)
```

[71]:
```
bool
```

[72]:
```
type(False)
```

[72]:
```
bool
```

Podemos realizar testes lógicos para concluir verdades ou falsidades quando temos dúvidas sobre objetos e relações entre eles. Por exemplo, retomemos os seguintes valores:

[73]:
```
W1
```

[73]: $e^{-\frac{16850581551}{6250000000000}}$

[74]:
```
W2
```

[74]: $e^{-\frac{95388992557}{195312500000}}$

A princípio, é difícil determinar qual dos dois é o maior. Porém, podemos realizar ‘‘perguntas’’ lógicas para o interpretador Python com operadores lógicos. Mostraremos apenas dois exemplos com > e <.

[75]:
```
W1 > W2 # isto quer dizer: "W1 é maior do que W2?"
```

[75]:
```
True
```

O valor True confirma que o valor de W1 é maior do que W2.

[76]:
```
W4 < 0
```

[76]:
```
False
```

Note que, de acordo com nosso modelo de caminhabilidade, este valor deveria ser zero. Porém, numericamente, ele é uma aproximação para zero. Embora muito pequeno, não é exatamente zero! Por que isso ocorre? Porque o computador lida com uma matemática inexata e aproximada, mas com precisão satisfatória.