

Python para Engenharia

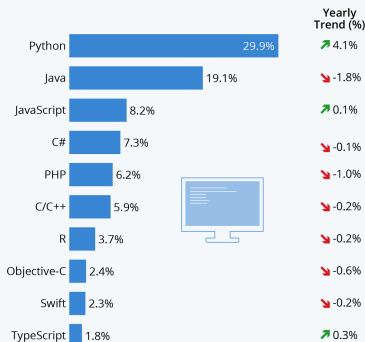
André Nepomuceno

Universidade Federal Fluminense

19 de outubro de 2023

Python Remains Most Popular Programming Language

Popularity of each programming language based on share of tutorial searches in Google



Yearly trend compares percent change from Feb 2019 to Feb 2020

Sources: GitHub, Google Trends



statista

Python é usado em diferentes áreas

- Ciência de Dados
- Inteligência Artificial
- Desenvolvimento Web
- Desenvolvimento de jogos
- Medicina e Farmacologia (AstraZeneca)
- Bioinformática
- Neurociência
- Física e Astronomia
- Business

Meu canal no YouTube: Python Para Cientistas

<https://www.youtube.com/@python4scientists/videos>

Material do minicurso disponível em:

<https://github.com/aanepomuceno/>

Minicurso-Python-Engenharia

- Entre o site do Google Colab
`https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index#recent=true`
- Escolha a opção **New Notebook**
- Renomeie o arquivo de `Untitled0` para um nome apropriado.

Em Python, uma **lista** é um conjunto ordenado de objetos que podem ser de vários tipos (inteiro, flutuante, complexo, booleano, string, etc.). Por exemplo, para criar uma lista, fazemos:

```
L = [1, 2.5, "Olá", True]
```

Cada entrada da lista é chamada de **elemento**, cada elemento tem uma **posição** na lista, e cada posição tem um inteiro associada a ela. Assim, o número (elemento) 1 está na posição zero da lista, o número 2.5 na posição um, e assim por diante. O **índice** que indica uma posição na lista sempre começa em zero.

Uma lista é um objeto **mutável**, e portanto podemos acrescentar ou retirar um elemento da lista.

Podemos também criar uma lista vazia: `L0 = []`.

Um elemento da lista pode ser acessado pelo seu índice. O operador **in** pode ser usado para verificar se um dado elemento pertence a lista.

Example

```
>>> L = [1, 2.5, 5.69, "x"]
>>> L[0]
1
>>> L[3]
'x'
>>> L[-1]
'x'
>>> 2 in L
False
>>> 'x' in L
True
```

Listas - objetos mutáveis

Como lista são mutáveis, é possível modificar itens da lista.

Example

```
>>> L = [1, 'dois', 3.14, 0]
>>> L[2] = 2.6
>>> L
[1, 'dois', 2.6, 0]
```

Atenção ao exemplo abaixo

Example

```
>>> q1 = [1, 2, 3]
>>> q2 = q1
>>> q1[2] = 'x'
>>> q1
[1, 2, 'x']
>>> q2
[1, 2, 'x']
```

Existem vários métodos que podem ser usados com listas. Exemplos de alguns métodos:

- `append()` - adiciona um elemento ao final da lista.
- `insert()` - semelhante ao `append()`, mas podemos escolher a posição onde o novo elemento será alocado. Exemplo: `L.insert(1,4.56)`
- `remove()` - remove um elemento específico que está na lista. Exemplo: `L.remove(4.56)`
- `pop()` - remove um elemento da lista, dado sua posição. Exemplo: `L.pop(1)` vai remover o elemento que está na posição "1" da lista, ou seja, o segundo elemento. `L.pop()` remove o último elemento da lista.
- `index()` - retorna o índice da primeira ocorrência de um elemento da lista (posição do elemento). Exemplo: `L.index(2.5)`
- `sort()` - ordena os elementos de uma lista em ordem crescente.
- `reverse()` - inverte a ordem dos elementos da lista.

Example

```
>>> import math
>>> L = []
>>> for i in range(5):
    L.append( round(math.sqrt(i**2.5),2) )
>>> L
[0.0, 1.0, 2.38, 3.95, 5.66]
>>> L.insert(1,5.3)
>>> L
[0.0, 5.3, 1.0, 2.38, 3.95, 5.66]
>>> L.sort()
>>> L
[0.0, 1.0, 2.38, 3.95, 5.3, 5.66]
```

NumPy é o pacote padrão para programação científica em Python. O módulo NumPy implementa de forma eficiente operações matemáticas. Para usar os métodos do módulo, devemos importá-lo no início do programa:

```
import numpy as np
```

Os objetos do NumPy são **arrays**, que é um conjunto ordenado de valores, mas que possuem diferenças cruciais em relação a listas:

- O número de elementos de um array é **fixo**. Não se pode adicionar ou remover itens de um array.
- Os elementos de um array são todos do mesmo tipo.
- Arrays podem ter n dimensões. Por exemplo, arrays com $n = 2$ são matrizes.
- Operações com arrays são **mais rápidas** do que com listas.

Criando Arrays

Vamos ver diversas formas de criar um array.

Array a partir de listas

```
>>> a = np.array([1.,2,3.1])
>>> a
array([1. , 2. , 3.1])
>>> a[0]
1.0
>>> b = np.array([ [1.,2.],[3.,4.] ]) #2D array
>>> b
array([[1., 2.],
       [3., 4.]])
>>> b[0,0]
1.0
>>> b[1,0]
3.0
```

Criando Arrays

Array com todas as entradas iguais a zero

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
>>> np.zeros(5, dtype=int)
array([0, 0, 0, 0, 0])
```

Array com todas as entradas iguais a um

```
>>> np.ones(6)
array([1., 1., 1., 1., 1., 1.])
>>> np.ones((3,4))    #matrix 3 x 4
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Criando Arrays

Array com todas as entradas iguais a um dado valor

```
>>> np.full(4, 3.14)
array([3.14, 3.14, 3.14, 3.14])
>>> np.full((4,3), 3.14)
array([[3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14]])
```

Array como matrix identidade

```
>>> np.eye(3)          #mesmo que np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Criando Arrays

Criando array com o método `arange()`

```
>>> np.arange(7)
array([0, 1, 2, 3, 4, 5, 6])
>>> np.arange(1.5, 3.0, 0.5)
array([1.5, 2. , 2.5])
>>> np.arange(6.5, 0, -1)
array([6.5, 5.5, 4.5, 3.5, 2.5, 1.5, 0.5])
```

A sintaxe do método **`arange()`** é `np.arange(início, fim, passo)`. Se apenas um número for dado, por exemplo, `np.arange(N)`, será criado um array de zero até o valor `N-1`, com passo de um.

Criando Arrays

A função `np.linspace(x, y, N)` gera N números entre x e y , com y **incluso**.

Criando array com o método `linspace()`

```
>>> np.linspace(0,10,6)
array([ 0.,  2.,  4.,  6.,  8., 10.])
>>> z,dz = np.linspace(0.,2*np.pi,100,retstep=True)
>>> dz
0.06346651825433926
```

A opção `retstep = True` retorna o tamanho do passo.

Warning

Note a diferença entre `arange()` e `linspace()`. Use `linspace()` sempre que desejar um array de tamanho precisamente N .

Atributos de um array

```
>>> a = np.array([ [1,0,1], [1,2,2] ])
>>> a.shape
(2, 3)
>>> a.ndim
2
>>> a.size
6
>>> a.dtype
dtype('int64')
>>> a.nbytes
48
```


Operações com Arrays

O grande poder do NumPy reside na realização de operações em todos os elementos de um array sem a necessidade de *loops* explícitos. Esse tipo de operação é chamada **vetorização**, e é muito mais rápida que *for loops*.

Example

```
>>> a = np.array([1.3, 2.5, 10.1])
>>> b = np.array([9.3, 0.2, 1.2])
>>> a + b
array([10.6,  2.7, 11.3])
>>> a*b
array([12.09,  0.5, 12.12])
>>> a/b
array([0.13978495, 12.5,  8.41666667])
>>> a/b + 1
array([ 1.13978495, 13.5,  9.41666667])
>>> a**2
array([1.69,  6.25, 102.01])
```

Operações com Arrays

Produtos

```
>>> a = np.array( [1.,2.,3.])
>>> b = np.array( [2.,4.,5.])
>>> np.dot(a,b) # produto interno, (mesmo que a @ b)
25.0
>>> np.cross(a,b) #produto vetorial
array([-2.,  1.,  0.] )
```

Operadores de comparação e lógica

```
>>> a = 2*np.linspace(1,6,6)
>>> a
array([ 2.,  4.,  6.,  8., 10., 12.])
>>> t = a > 10
>>> t
array([False, False, False, False, False,  True])
```

Operações com Arrays

Exemplo: Vamos implementar o cálculo abaixo:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} + 2 \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} -3 & 5 \\ 0 & 2 \end{pmatrix}$$

Código

```
>>> a = np.array([ [1,3], [2,4] ])
>>> b = np.array([ [4,-2], [-3,1] ])
>>> c = np.array([ [1,2], [2,1] ])
>>> r = np.dot(a,b) + 2*c
>>> r
array([[ -3,  5],
       [ 0,  2]])
```

Operações com Arrays - Funções

As funções disponíveis no módulo `math` também existem no NumPy. Teste os exemplos abaixo.

Funções

```
theta = np.linspace(0.1,np.pi,4)
print("theta = ", theta)
print("sen(theta) = ",np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("ln(theta) = ",np.log(theta))
print("log(theta) = ",np.log10(theta))
print("exp(theta) = ", np.exp(theta))
print("modulo = ", np.absolute(np.log(theta)))
```

Arrays - Funções de Agregação

Quando trabalhamos com um grande conjunto de dados, é comum calcularmos estatísticas para uma análise inicial. NumPy oferece vários métodos para essa tarefa.

Example

```
#soma, media, max. e min.  
>>> data = np.random.random(100)  
>>> data.sum()  
51.72239489031435  
>>> data.mean()  
0.5172239489031435  
>>> data.max()  
0.9946152525979709  
>>> data.min()  
0.0023509304159052835
```

Arrays - Funções de Agregação

Para arrays em n -dimensões ($n > 1$), podemos escolher o *eixo* sobre o qual os valores serão agregados.

Example

```
#2D array
>>> M = np.random.random((3,4))
>>> M
array([[0.37019599, 0.15892146, 0.23032805, 0.37...],
       [0.17968684, 0.69242006, 0.51502879, 0.06...],
       [0.78280796, 0.63324658, 0.22553994, 0.94...]])
>>> M.sum()
5.168637614536063
>>> M.sum(axis=0)
array([1.33269079, 1.4845881 , 0.97089679, 1.380...])
>>> M.max(axis=1)
array([0.37897901, 0.69242006, 0.94005747])
```

Arrays - Funções de Agregação

Método	Descrição
<code>np.sum</code>	soma dos elementos
<code>np.cumsum</code>	soma cumulativa dos elementos
<code>np.prod</code>	produto dos elementos
<code>np.mean</code>	valor médio
<code>np.std</code>	desvio padrão
<code>np.var</code>	variância
<code>np.min</code>	valor mínimo
<code>np.max</code>	valor máximo
<code>np.argmin</code>	índice do valor mínimo
<code>np.argmax</code>	índice do valor máximo
<code>np.conj</code>	complex. conjugado de todos elementos
<code>np.trace</code>	soma dos elementos da diagonal

Veja mais detalhes neste [LINK](#).

Muitas vezes precisamos obter um “subarray” a partir de um array, ou seja, um array com apenas alguns elementos do array original. Para isso, existe uma técnica chamada **slicing**. A sintaxe é:

[início:fim:passo]

onde “início” é o índice (posição) da primeira entrada desejada, e “fim” o índice do último elemento, que **NÃO** entrará no novo array. Esse comando vai gerar um array com entradas $a[\text{início}]$, $a[\text{início} + \text{passo}]$, $a[\text{início} + 2 * \text{passo}]$, \dots , $a[\text{início} + N * \text{passo}]$, com a posição “ $\text{início} + N * \text{passo}$ ” $< \text{fim}$.

O array que retorna dessa operação **não** é uma cópia, ou seja, não é um novo objeto.

Arrays - Slicing

Example

```
>>> a = np.linspace(1,6,6); a
array([1., 2., 3., 4., 5., 6.])
>>> a[:3]      #mesmo que a[0:3]
array([1., 2., 3.])
>>> a[1:4:2]
array([2., 4.])
>>> a[1:]
array([2., 3., 4., 5., 6.])
>>> a[3::-2]
array([4., 2.])
>>> a[::-1]
array([6., 5., 4., 3., 2., 1.])
```

Exemplo 1: Dados dois arrays de posição x e tempo t de uma partícula, calcule a velocidade média \bar{v} para cada intervalo de tempo, utilizando *slicing*.

```
x = np.array([0., 1.3, 5. , 10.9, 18.9, 28.7, 40.])
t = np.array([0., 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```

Com

$$\bar{v} = \frac{x_i - x_{i-1}}{t_i - t_{i-1}}$$

Abrindo arquivos com NumPy

Para abrir arquivos de dados dos tipos `.txt`, `.dat` ou `.csv`, podemos usar o métodos **`np.loadtxt()`**. Os dados serão transformados num array. Como default, é assumido que os dados estão separados por espaços ou tabulação.

```
import numpy as np
data_set = np.loadtxt("millikan.txt")
data_x = data_set[:,0]
data_y = data_set[:,1]
```

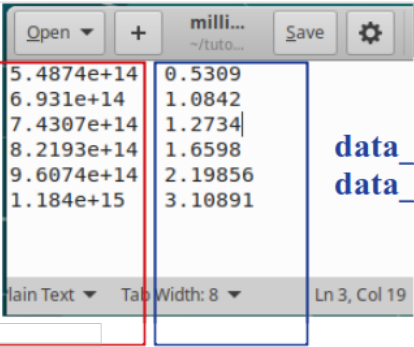
Se os valores estiverem separados por um caractere, ele dever ser especificado usando a palavra chave `delimiter`.

```
data_set = np.loadtxt("millikan.csv", delimiter=',')
```

Importando e Exportando Dados

A figura ilustra o exemplo acima.

data_set



5.4874e+14	0.5309
6.931e+14	1.0842
7.4307e+14	1.2734
8.2193e+14	1.6598
9.6074e+14	2.19856
1.184e+15	3.10891

data_x
data_set[:,0]

data_y
data_set[:,1]

Álgebra Linear com NumPy - Normas e *Rank*

Normas são calculadas com o módulo `np.linalg.norm`. O *rank* (posto) é obtido pelo método `np.linalg.matrix_rank`.

1. Norma de um Vetor

$$\|a\| = \left(\sum_i |z_i|^2 \right)^{1/2}$$

2. Norma de Frobenius

$$\|A\| = \left(\sum_{i,j} |a_{ij}|^2 \right)^{1/2}$$

3. *Rank*: número de colunas linearmente independentes.

Álgebra Linear com NumPy - Normas e *Rank*

Cálculo de Normas

```
In[x]: np.linalg.norm(A)
Out[x]: 2.29128784747792
In[x]: c = np.array([1, 2j, 1-1j])
        np.linalg.norm(c)
Out[x]: 2.6457513110645907
```

Cálculo do Rank

```
In[x]: np.linalg.matrix_rank(A)
Out[x]: 2
In[x]: D = np.array([[1, 1], [2, 2]])
Out[x]: array([[1, 1],
               [2, 2]])
In[x]: np.linalg.matrix_rank(D)
Out[x]: 1
```

Álgebra Linear com NumPy - Determinante e Inversa

Determinante

```
In[x]: np.linalg.det(A)
```

```
Out[x]: 0.5
```

Traço

```
In[x]: np.trace(A)
```

```
Out[x]: 2
```

Matriz Inversa

```
In[x]: np.linalg.inv(A)
```

```
Out[x]: array([[ 4., -1.],  
               [ 2.,  0.]])
```

Se a matriz não tiver inversa, será retornado o erro

LinAlgError: Singular matrix

Problema de autovalor

Para uma matriz quadrada \mathbf{A} , um *autovetor* \mathbf{v} é um vetor que satisfaz

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

onde λ são chamados *autovalores*. Para um matriz $N \times N$, existem N autovalores e N autovetores.

Para calcular autovetores e autovalores existe o módulo

`np.linalg.eig`, que retorna os autovalores como um array de forma $(n,)$ e os autovetores como **colunas** de um array de forma (n, n) .

Use `np.linalg.eigval` para calcular os autovalores apenas.

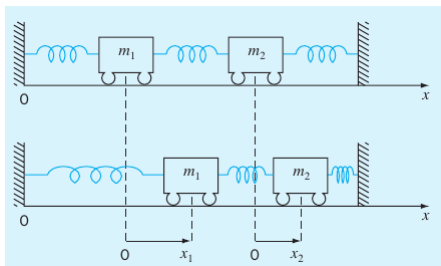
Autovalores e Autovetores

```
In[x]:  vals, vecs = np.linalg.eig(A)
        print(vals)
```

```
Out[x]: [0.29289322  1.70710678]
```


Exemplo de Aplicação

Exemplo 2 No sistema massa-mola abaixo, vamos assumir que as molas tem os mesmos comprimentos naturais e as mesmas constantes k . O deslocamento de cada mola é medido em relação ao seu próprio sistema de coordenada.



Aplicando a segunda lei de Newton:

$$m_1 \frac{dx_1^2}{dt^2} = -kx_1 + k(x_2 - x_1)$$

$$m_2 \frac{dx_2^2}{dt^2} = -k(x_2 - x_1) - kx_2$$

Exemplo de Aplicação

A solução é dada por

$$x_i = X_i \sin(\omega t)$$

Substituindo nas equações anteriores:

$$\left(\frac{2k}{m_1} - \omega^2 \right) X_1 - \frac{k}{m_1} X_2 = 0$$

$$-\frac{k}{m_2} X_1 + \left(\frac{2k}{m_2} - \omega^2 \right) X_2 = 0$$

Vamos considerar o caso $m_1 = m_2 = 40$ kg, e $k = 200$ N/m.

Álgebra Linear com NumPy - Sistemas Lineares

NumPy dispõe de um método eficiente e estável para resolver sistemas de equações lineares: `np.linalg.solve`. Exemplo: o sistema abaixo

$$\begin{aligned}3x - 2y &= 8, \\ -2x + y - 3z &= -20, \\ 4x + 6y + z &= 7\end{aligned}$$

pode ser escrito como uma equação matricial $\mathbf{Mx} = \mathbf{b}$

$$\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}$$

Solução de Sistemas Lineares

```
In[x]: M = np.array([ [3., -2, 0], [-2, 1, -3],  
                      [4, 6, 1]])
```

```
In[x]: print(M)
```

```
Out[x]: [[ 3. -2.  0.]  
         [-2.  1. -3.]  
         [ 4.  6.  1.]]
```

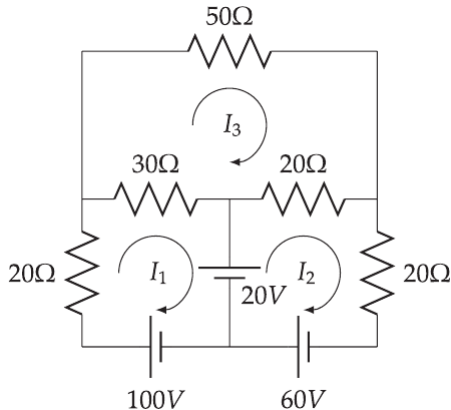
```
In[x]: b = np.array([8, -20, 7])
```

```
In[x]: x, y, z = np.linalg.solve(M,b)  
        print('x = {}, y = {}, z = {}'.format(x,y,z))
```

```
Out[x]: x = 2.0, y = -1.0, z = 5.0
```

Exemplo de Aplicação

Exemplo 3 No circuito abaixo, determine os valores das correntes I_1 , I_2 , I_3 .



Vamos aplicar a 2ª lei de Kirchhoff ($\sum_k V_k = 0$) e a lei de Ohm ($V = RI$) ao circuito:

$$50I_1 - 30I_3 = 80$$

$$40I_2 - 20I_3 = 80$$

$$-30I_1 - 20I_2 + 100I_3 = 0$$

Python tem uma poderosa biblioteca para produção de gráficos de boa qualidade: **Matplotlib**. Para gráficos simples, podemos usar o módulo **pyplot** que deve ser importado da seguinte forma:

```
import matplotlib.pyplot as plt
```

No jupyter, para que o gráfico apareça numa célula do notebook, digite na primeira célula: `%matplotlib inline`. Para que o gráfico seja mostrado numa janela separada, digite na primeira célula: `%matplotlib`.

Se quisermos fazer um gráfico de uma função, as entradas para o pyplot devem ser arrays (ou listas) correspondentes aos valores x e y. Exemplo:

Exemplo 1 - Gráfico simples

```
x = np.linspace(-3*np.pi, 3*np.pi, 100)
y = np.sin(x)      #vetorização
plt.plot(x, y)
```

Para adicionar um segundo plot, basta chamar `plt.plot` novamente:

```
z = np.cos(x)
plt.plot(x, z)      #ou plt.plot(x, y, x, z)
```

Para nomear um gráfico, devemos atribuir um string ao argumento `label` da função `plot`. Para adicionar a legenda no gráfico, faça:

```
plt.legend()
```

Exemplo 2 - Legenda

```
x = np.linspace(-3*np.pi, 3*np.pi, 100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, label='sen(x)')
plt.plot(x, z, label='cos(x)')
plt.legend()
```

Para retirar a legenda da "caixa", use a opção `frameon=False`. Para selecionar o tamanho da fonte, use `fontsize=<inteiro>`.

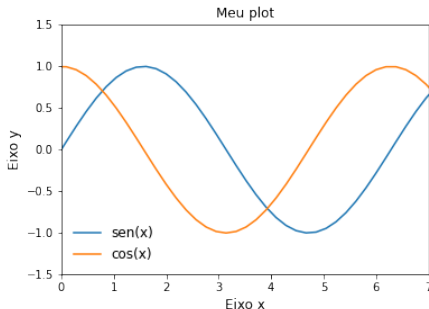
Opções de localização da legenda

String	Inteiro
'best'	0
'upper right'	1
'upper left'	2
'lower left'	4
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'upper center'	10

Gráficos - Descrição dos Eixos e Título

Exemplo 3 - Eixos e Título

```
plt.title('Meu plot')  
plt.xlabel('Eixo x', fontsize=12)  
plt.ylabel('Eixo y', fontsize=12)  
plt.xlim(0, 7.)      #x-range  
plt.ylim(-1.5, 1.5)  #y-range
```



Gráficos - Marcadores, Cores e Linhas

Existem diversas opções de marcadores, linhas e cores, que devem ser especificados por strings. Por exemplo, se quisermos linha vermelha tracejada, basta incluir 'r- -' na função plot.

Exemplo 3 - Cores e Linhas

```
plt.plot(x,y, 'r--', label='sen(x)')  
plt.plot(x,y, 'r--o', label='sen(x)')    #marcador 'o'
```

Também é possível passar os atributos explicitamente com `c` (color), `marker` (marcador) `ls` (estilo da linha) e `lw` (largura da linha)

```
plt.plot(x,y, c='r', marker='o', ls='--', lw=2)
```

Também é possível selecionar o tamanho do marcador (`markersize`), a cor (`markerfacecolor` ou `mfc`), e a cor da borda (`markeredgecolor` ou `mec`).

Gráficos - Marcadores, Cores e Linhas

Marcadores

Código	Marcador
.	Ponto
o	Círculo
+	Cruz
x	Cruzado
D	Diamante
v	Triângulo p/ baixo
^	Triângulo p/ cima
s	Quadrado
*	Estrela

Cores Básicas

Código	Cor
r	Vermelho
g	Verde
b	Azul
c	ciano
m	magenta
y	Amarelo
k	Preto
w	Branco
brown	Marrom
gray	Cinza
purple	Roxo

Estilos de linha: (-) (-) (:) (-.)

Gráficos de barras são feitos com a função `plt.bar`.

Example

```
anos = np.arange(2014,2021)
consumo = np.array([327,392,490,643,806,981,1171])
plt.bar(anos, consumo, width=0.6,color='g',
        edgecolor=None)
plt.xlabel('Anos')
plt.ylabel('Consumo em milhares de sacas')
```

Gráficos com barras de erro podem ser criados com a função `plt.errorbar`. Pode-se escolher várias opções de formatação para a barra de erro.

Example

```
x = np.linspace(0,10,30)
dy = 0.8
y = np.sin(x) + dy*np.random.random(30)
#plt.errorbar(x,y,yerr=dy,fmt='ko')
plt.errorbar(x,y,yerr=dy,ecolor='gray',elinewidth=2,
             capsize=3,fmt='ko')
```

Matplotlib - Histogramas

Histograma é uma representação gráfica de uma distribuição discreta de probabilidade. Para plotar um histograma, basta passar um array para a função `plt.hist()`. O exemplo abaixo ilustra as várias opções disponíveis.

Example

```
data = np.random.randn(1000)
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='bar', color='steelblue')
```

Para obter a contagem em cada bin, podemos usar o método `np.histogram()`:

Example

```
contagem, x_bin = np.histogram(data, bins=10)
```

A função `pyplot.scatter()` permite criar gráficos de dispersão onde as propriedades de cada ponto (cor, tamanho) podem ser controladas. Além dos valores `x` e `y`, podemos passar uma sequência de valores para os argumentos `s` e `c`, que controlam o tamanho e a cor da cada ponto. Esse tipo de gráfico é útil para visualizar dados multidimensionais.

Example

```
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000*np.random.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)
plt.colorbar()
```


Exemplo 4 Vamos escrever um código em Python para ler o arquivo `dados_bola_caindo.dat` e fazer um gráfico dos dados e de duas funções:

$$y1 = y_0 - \frac{g}{2}t^2$$

$$y2 = y_0 - \frac{v_T^2}{g} \log \left(\cosh \frac{gt}{v_T} \right)$$

onde $v_T = \sqrt{g/D}$, g é a aceleração da gravidade e $D = 0.065 \text{ m}^{-1}$. Considere $y_0 = 2 \text{ m}$.

O pacote `scipy.optimize` implementa vários métodos para calcular raízes de funções. Os argumentos passados devem ser uma função contínua, $f(x)$, e um intervalo $[a, b]$ dentro do qual a raiz será encontrada, tal que $\text{sgn}[f(a)] = -\text{sgn}[f(b)]$. Alguns dos métodos disponíveis:

- Método de Brent (`scipy.optimize.brentq`)
- Método da bisseção (`scipy.optimize.bisect`)
- Método de Newton (`scipy.optimize.newton`)

No caso do método Newton–Raphson, deve-se passar um ponto inicial, x_0 (próximo a raiz), e opcionalmente, a primeira derivada da função, `fprime`. Note que nesse método, temos menos controle sobre a raiz encontrada se a função tem várias raízes.

Métodos numéricos devem ser utilizados com cuidado. Verifique se a raiz x encontrada produz $f(x) \approx 0$.

Raízes - Método de Newton

Vamos encontrar a raiz da função abaixo pelo método de Newton

$$f(x) = e^x - 2$$

```
In[x]: from scipy.optimize import newton
In[x]: f = lambda x: np.exp(x) - 2
In[x]: fprime = lambda x: np.exp(x)
In[x]: x0 = 2
In[x]: xsol = newton(f, x0, fprime=fprime)
        print(xsol)
Out[x]: 0.6931471805599453
In[x]: np.isclose(f(xsol), 0, atol=1e-10)
Out[x]: True
```

O método `scipy.optimize.curve_fit` permite passarmos de forma transparente os erros da variável `y` e obter as incertezas nos parâmetros ajustados. O método é chamado da seguinte forma:

```
curve_fit(f, xdata, ydata, p0, sigma, absolute_sigma).
```

- `f`, `xdata`, `ydata` são, respectivamente, a função a ser ajustada aos dados (`xdata`, `ydata`);
- `p0` é um valor inicial para os parâmetros;
- `sigma` é um array com as incertezas de `ydata`, de mesmo tamanho de `ydata`;
- `absolute_sigma` é uma variável booleana. Se **True**, os valores absolutos de `sigma` são usados. Essa deve ser a opção usada para obter os valores absolutos nas incertezas dos parâmetros. Se escolhermos a opção **False**, os valores de `sigma` são tratados como valores relativos.

O método `curve_fit` retorna o array `popt`, com o valor dos parâmetros ajustados, e o array 2D `pcov`, a matriz de covariância dos parâmetros. A incerteza nos parâmetros é dada pela raiz quadrada da diagonal de `pcov`:
`np.sqrt(np.diag(pcov))`.

Para ilustrar o uso deste método, vamos ajustar uma função linear a um conjunto de pontos de um experimento para determinar a aceleração da gravidade local:

$$T^2 = \frac{4\pi^2}{g} L$$

onde $a = 4\pi^2/g$ é o parâmetro a ser ajustado.

SciPy - Integração Numérica

O pacote `scipy.integrate` contém funções para o cálculo numérico de integrais definidas próprias (limites finitos) e impróprias (limites infinitos). A rotina está implementada em `scipy.integrate.quad`, que é baseada na biblioteca QUADPACK (FORTRAN 77). Os argumentos básicos são o integrando (`func`), e os limites de integração `a` e `b`. O resultado será um flutuante com o valor da integral e outro com uma estimativa do erro absoluto.

Example

$$I = \int_1^4 x^{-2} dx$$

```
In[x]: from scipy.integrate import quad
```

```
In[x]: f = lambda x: 1/x**(2)
```

```
In[x]: quad(f, a=1, b=4)
```

```
Out[x]:
```

```
(0.75000000000000002, 1.913234548258995e-09)
```

Se uma função depende de outros parâmetros além da variável independente, estes devem ser passados como **tuplas** para o argumento `args`.

Example

$$I = \int_{-\pi/2}^{\pi/2} \sin^n x \cos^m x \, dx$$

```
In[x]: def f3(x,n,m):  
        return np.sin(x)**n*np.cos(x)**m  
In[x]: quad(f3,-np.pi/2,np.pi/2,args=(2,1))  
Out[x]:  
(0.6666666666666666, 1.6257269518146785e-13)
```

Note que os parâmetros `n` e `m` devem aparecer como argumentos do integrando **depois** da variável de integração `x`.

Para integrar funções com singularidades, devemos passar uma lista de pontos onde ocorrem as divergências usando o argumento `points`.

Example

$$I = \int_{-1}^1 \frac{dx}{\sqrt{|x|}}$$

```
In[x]: f5 = lambda x: 1/np.sqrt(np.abs(x))
```

```
In[x]: quad(f5,-1,1)
```

```
Out[x]:
```

```
RuntimeWarning: divide by zero encountered in  
double_scalars
```

```
(inf, inf)
```

```
In[x]: quad(f5,-1,1,points=[0,])
```

```
Out[x]:
```

```
(3.99999999999999813, 5.684341886080802e-14)
```


Equações diferenciais ordinárias (EDOs) podem ser resolvidas numericamente com `scipy.integrate.odeint` ou `scipy.integrate.solve_ivp`. Esses métodos resolvem equações da forma:

$$\frac{dy}{dt} = \mathbf{F}(\mathbf{y}, t)$$

onde \mathbf{y} é um vetor de componentes $y_i(t)$, e \mathbf{F} um vetor de componentes $F(y_i, t)$.

Para resolver EDOs de ordem $n > 1$, devemos transformá-las em um sistema de EDOs de primeira ordem (exemplos nos próximos slides).

O método `scipy.integrate.solve_ivp` toma pelo menos três argumentos: uma função que retorna dy/dt , os pontos iniciais e finais da variável t , e um conjunto de condições iniciais y_0 .

Exemplo 1:

$$\frac{dy}{dt} = -ky$$

- 1 Primeiro, definimos dy/dt (note a ordem das variáveis!)

```
def dydt(t, y):  
    return -k*y
```
- 2 Os tempos iniciais e finais devem ser passados como tuplas para o argumento `t_span`: `t_span = (t0, tf)`.
- 3 Os valores iniciais `y0` devem ser passados como sequência (lista, array), **mesmo que só tenha um valor**.
- 4 A solução será um objeto `soln` com os arrays `soln.y`, `soln.t` e `soln.success` (booleano).

EDOs Acopladas

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(y_1, y_2, \dots, y_n; t), \\ \frac{dy_2}{dt} &= f_2(y_1, y_2, \dots, y_n; t), \\ &\dots \\ \frac{dy_n}{dt} &= f_n(y_1, y_2, \dots, y_n; t).\end{aligned}$$

Nesse caso, a função a ser passada para o método `solve_ivp()` deve retornar uma sequência com as funções $f_i(y_1, y_2, \dots, y_n; t)$.

EDOs Acopladas - Implementação

```
# Y = [y1, y2, y3, ...]
#(sequencia de variáveis independentes)
def deriv(t, Y):
    y1,y2,y3... = Y
    dy1dt = f1(Y, t)
    dy2dt = f2(Y, t)
    #....
    return dy1dt, dy2dt, ...,dyndt
solve_ivp(deriv, (t0, tf), y0 )
```

Note que agora, y_0 será um sequência de n elementos.

Exemplo 2: Sistema de EDOs

$$\frac{dx}{dt} = xy - x,$$

$$\frac{dy}{dt} = y - xy + \text{sen}^2 \omega t$$

Exemplo 3: EDO de segunda ordem

Para resolver uma EDO de ordem $n > 1$, primeiro devemos reduzi-la a um sistema de EDOs de primeira ordem:

$$\frac{d^2x}{dt^2} = -\omega^2 x$$

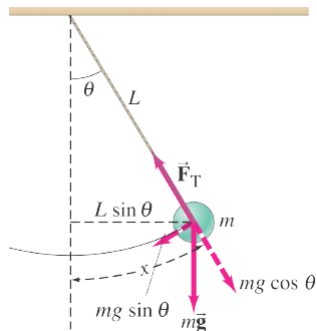
$$\frac{dx}{dt} = v,$$

$$\frac{dv}{dt} = -\omega^2 x,$$

Métodos disponíveis (argumento `method`)

- **RK45**(default): Explicit Runge-Kutta method of order 5(4)
- **RK23** Explicit Runge-Kutta method of order 3(2)
- **DOP853** Explicit Runge-Kutta method of order 8
- **Radau** Implicit Runge-Kutta method of the Radau IIA family of order 5
- **BDF** Implicit multi-step variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation.

Exemplo 4: Pêndulo Não-Linear



Equação do Movimento:

$$\frac{d^2\theta}{dt^2} + \omega^2 \sin \theta = 0$$

onde $\omega = \sqrt{g/L}$.

Energia total

$$E = \frac{1}{2} m L^2 \dot{\theta}^2 + mg l (1 - \cos \theta)$$