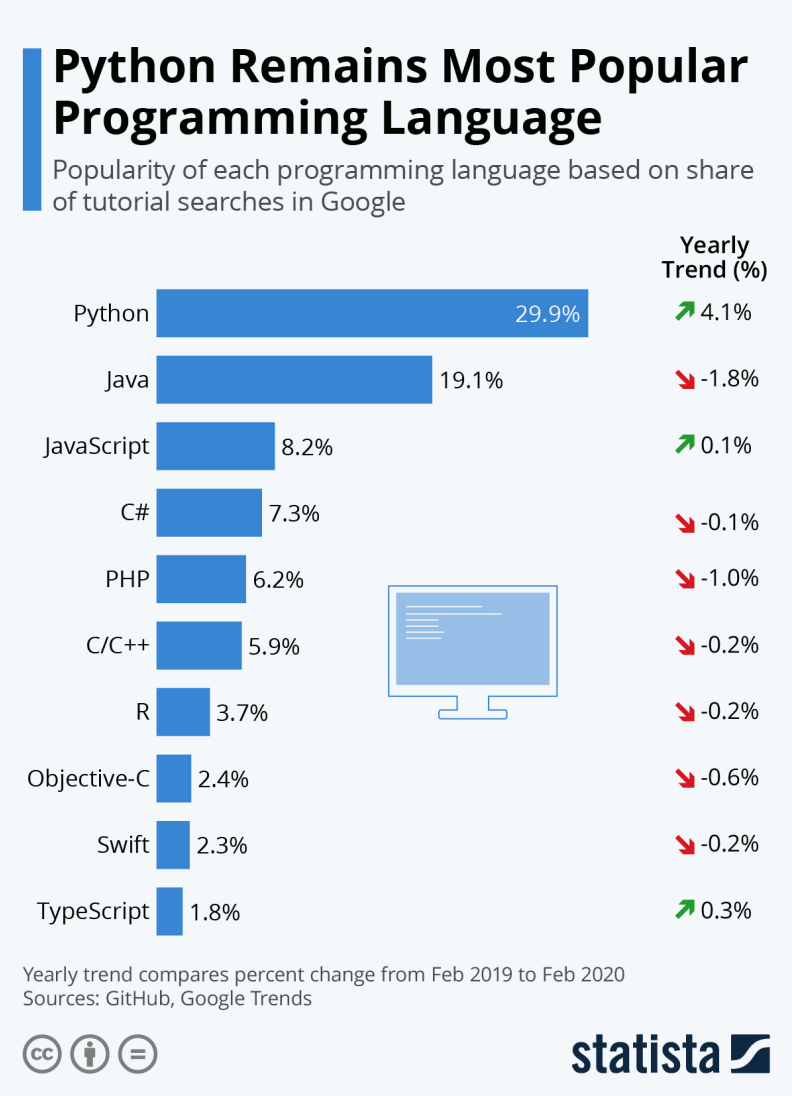
Python para Engenharia

André Nepomuceno

Universidade Federal Fluminense

19 de outubro de 2022

Roubando a Cena



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

Google Colaboratory

- 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.

Tipos de Dados

Inicialmente, vamos investigar quatro tipos de dados: string, inteiro e flutuante e complexo.

Strings (type:`str`)
Strings são textos formados por caracteres. Em Python, strings são sempre escrito entre aspas.

Inteiros (type:`int`)
Podem ser positivos, negativos ou zero, por exemplo 1, 9, - 54, 38523. A aritmética de inteiros é exata.
Em Python, é possível separar dígitos com "_". Por exemplo, o número 10525839 pode ser escrito como 10_525_839.

Tipos de Dados

Flutuante (type:`float`)
Flutuantes são representações de números reais. Por exemplo: 1.3, -0.456 ou 1.65×10^{-6}. **Note que usamos ponto (.) como separador decimal**.
Notação científica é escrita como `1.65e-6`.
Exemplo: 13 é inteiro, 13.0 é flutuante, e "13" é um string.

Complexo (type:`complex`)
Números complexos são formados por uma parte real e uma imaginária. Em Python, um número complexo é formado por dois flutuantes. Por exemplo, o número $5.1 + 2.3\mathbf{i}$ pode ser escrito de duas formas: `5.1+2.3j` ou `complex(5.1,2.3)`.

Conversão

Conversão (Type Casting)

As vezes é necessário transformar um tipo de dado em outro tipo (transformar um string em um inteiro, por exemplo). As funções que fazem essa conversão são chamadas *type casting*, e são as seguintes:

int() - transforma o valor entre parênteses em um inteiro;
float() - transforma o valor entre parênteses em um flutuante;
complex() - transforma o valor entre parênteses em um complexo;
str() - transforma o valor entre parênteses em um string.

Exercício:Digite os comandos abaixo no jupyter e observe o resultado.

```
float(3)
int(7.2)
int(7.9)
complex(3.)
complex(0,3.)
```

Operações

Operações Aritméticas do Python

+ adição
- subtração
* multiplicação
/ divisão
// inteiro da divisão (retorna o **menor** inteiro)
% módulo (resto da divisão)
** potência (m elevado a n, m**n)

Warning

O operador de divisão (/) sempre retorna um **flutuante**, mesmo que a divisão seja entre inteiros divisíveis.

Atributos e Métodos

Em Python, tudo é um **objeto**, incluindo números (um objeto é uma combinação de dados e funções). Objetos possuem *atributos* e *métodos*. O método é uma função que atua no objeto. A sintaxe para acessar atributos e métodos é:

- Atributo: <objeto>.<atributo>
- Métodos: <objeto>.<metodo>()

Example

```
>>> (4 + 5j).real
4.0
>>> (4 + 5j).imag
5.0
>>> (4 + 5j).conjugate()
(4-5j)
```

Funções Matemáticas

Existem duas funções matemáticas disponíveis "by default" no Python. A função **abs**, que retorna o módulo de um número, e a função **round**, que faz a aproximação de um número flutuante.

Example

```
>>> abs(-6.4)
6.4
>>> abs(3 + 4j)
5.0
>>> round(5.56)                    #retorna um inteiro
6
>>> round(3.141592653589,2) #2 casas decimais
3.14
```

Como calcular raiz quadrada e funções trigonométricas ?

Módulos

Em Python, um módulo é um conjunto de funções. Para utilizar uma função de um determinado módulo, você precisará **importar** o módulo para o seu programa, da seguinte forma:
from <nome_do_modulo> **import** <funcão1> <função2 >...
Por exemplo, a função **sqrt()**, que calcula a raiz quadrada de um número, pertente ao módulo *math*. Para usá-la num programa, devemos escrever:

```
from math import sqrt
sqrt(2)
```

Em geral, um módulo contém várias funções. E como podemos saber quais funções estão disponíveis num módulo ? Basta importamos o módulo e usar o comando **help**. Tente os seguintes comandos no jupyter:

```
import math
help(math)
```

Variáveis

Em termos simples, uma variável é um letra ou palavra que guarda um valor de qualquer tipo de dado (string, inteiro, flutuante ou complexo). Mais precisamente, uma variável reserva um espaço na memória do computador para guardar o valor a ela atribuído.
Para criar uma variável, escolhemos um nome e atribuímos um valor a variável usando o operador (=). Execute o exemplo abaixo:

```
x = 5
print(x)
```

O que acontece aqui ?
1. Primeiro, criamos a variável x e atribuímos a ela o valor 5. Automaticamente, Python interpretará que a variável x é um inteiro.
2. O valor atribuído a x é mostrado pela função print. Note que ela mostra o **valor** de x, e não a letra (nome da variável).

Variáveis

Quando um novo valor é atribuído a variável, o valor anterior é "esquecido", e a variável passa a ter o valor atual. Exemplo:

```
x  = 5
print(x)
x = 9*2 + 1  #substitui o valor anterior pelo novo
print(x)
x = "agora sou string"
print(x)
```

É possível criar várias variáveis numa única linha:

```
a,b,c = 2.1,-5.6,9.2
print(a,b,c)
```

Variáveis

Uma vez que uma variável é criada, podemos utilizá-la em qualquer parte do programa para executar cálculos. Exemplo:

```
x = 2
y = 3
print("x = ", x )
print("y = ", y )
print("x + y =", x + y )
print("x*y =", x * y )
```

Também é possível usar a própria variável para atribuir a ela um novo valor. Tente o código abaixo:

```
z = 2
print(z)
z = z + 1
print(z)
```

Variáveis - Exemplo

Vamos implementar a fórmula de Heron para calcular a área de um triângulo de lados a,b e c:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

onde $s = \frac{1}{2}(a+b+c)$

Example

```
import math
a,b,c = 4.5,2.4,3.9
s = (a+b+c)/2.
area = math.sqrt(s*(s - a)*(s - b)*(s - c))
print(area)
```

Comparações

O resultado de uma comparação em Python é um objeto *booleano* que pode ter dois valores: **True** ou **False**. Os principais operadores de comparação são listados abaixo.

Operadores de Comparação

< menor que
<= menor ou igual
== igual a
>= maior ou igual
> maior que
!= diferente

Warning

Não confundir o operador atribuição (=) com o operador de comparação (==)!

Comparações

Example

```
>>> 7 == 8
False
>>> 4 > 3.18
True
>>> var_comp = 5 ==2
>>> print(var_comp)
False
>>> "teste" != "teste1"
True
```

Warning

Nunca se deve comparar flutuantes com (==) ou (!=).

Condições - A palavra-chave *if*

Uma condição é um teste realizado pelo programa para saber se uma determina ação deve ou não ser executada. A maneira mais comum de se escrever condições num programa é usando a palavra **if**. A sintaxe (forma de escrever a expressão) é a seguinte:

if <condição >:
    <o_que_sera_executado >

É importante notar que:

1. Logo após a condição, devemos colocar **dois pontos** (:). Python interpretará que todo o comando abaixo deverá ser executado SE a condição for satisfeita.
2. Os comandos que serão executados devem ser colocados abaixo da condição, com **espaços a esquerda** (em geral, utilizamos quatro espaços). Todos as linhas que estiverem nessa margem são consideradas como dentro do "bloco if". Editores como jupyter criarão esse espaço automaticamente, assim que a tecla <enter >for pressionada após (:).

Condições - A palavra-chave *if*

Example

```
x = 7
if x < 10:
    print("Essa linha sera executada se  x < 10." )
    print("Mesmo que a linha acima.")
print("Essa linha sera sempre executada.")
```

Exercício

Teste o programa acima para x = 8,9,10 e 11.

Iterações - while loops

Iterações são necessárias quando queremos que o programa repita uma operação várias vezes. Basicamente, existem duas maneiras de executar repetições: **while** loop e **for** loop.
No caso do comando **while**, a sintaxe é:

while <condição>:
    <comando>

Note que a sintaxe é semelhante ao **if**. O código acima fará com que <comando> seja executado enquanto <condição> for **True** (verdadeira). Se <condição> nunca for **False**, o programa entrará num loop infinito.

Iterações - while loops

Example

```
num = 1
while num <= 5:
    print( num )
    num += 1        #equivalente a num = num+1
print( "Fim!" )
```

Nesse programa, a condição "num <= 5" é testada, e cada vez que ela for True, o bloco dentro do **while** é executada, ou seja, o valor da variável num é impresso na tela e aumentado em uma unidade. Quando num for igual a 6, a condição será False, o loop para, e a linha print("Fim!") é executada.

Iterações - for loops

Outra maneira bastante comum de implementar loops é através do comando **for**. A sintaxe é:

for <variaveis> in <coleção>:
    <comandos>

Uma coleção pode ser um string, uma variável ou números. Exemplos:

Example

```
for letter in "banana":
    print(letter)
print("Fim!")
#outra forma
fruta = "banana"
for letter in fruta:
    print(letter)
print("Fim!")
```

Iterações - for loops

Para executar iterações sobre uma coleção de números, Python oferece a função **range()**. Essa função pode ser usada de três formas:

- range(x) – Será criada uma sequência de TODOS os números inteiros de 0 até x -1. Por exemplo, range(4) gerará a sequência (0,1,2,3).
- range(x,y) - Será criada uma sequência de números inteiros de x até y-1.
- range(x,y,z) – Nesse caso, x e y são os limites da sequência, e z é o passo. Por exemplo, range(1,11,2) criará (1,3,5,7,9). Note que o último número NÃO é incluído.

Loops dentro de loops

Loops podem ser criados dentro de loops. Nessas situações, é importante atentar para o **correto espaçamento** de cada comando, que definirá a qual loop ele pertente. Estude o exemplo abaixo.

```
for i in range(3):
    print("Entrando no primeiro loop para i =" ,i)
    for j in range(4):
        print("Entrando no segundo loop para j = ", j)
        print("i, j = ", i, j)
    print("Saindo do primeiro loop para i =" ,i)
```

Listas

Em Python, uma **lista** é um conjunto ordenado de objetos que podem ser de vários tipos (inteiro, flutuante, complexo, boleano, 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 = []`.

Listas

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']
```

Listas - Métodos

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 no 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.

Listas - Exemplos

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 Arrays

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 crucias 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(inicio, 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
0.0634665182543926
```

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.

Arrays - Atributos

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]])
```

Arrays - Slicing

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:

[inicio: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[inicio], a[inicio + passo], a[inicio+2*passo], ...a[inicio+N*passo], com a posição "inicio+N*passo" $<$ fim.
O array que retorna dessa operação **não** é um 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.])
```

Álgebra Linear com NumPy - Operações com Matrizes

Multiplicação dos **Elementos** das Matrizes

```
In[x]:  A * B
Out[x]: array([[ 0. , -0.25],
               [-3.,  3.  ]])
```

Matriz Transposta

```
In[x]:  A.T   #ou A.transpose()
Out[x]: array([[ 0. , -1. ],
               [ 0.5,  2. ]])
```

Matriz Identidade

```
In[x]:  np.eye(3,3)
Out[x]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

Álgebra Linear com NumPy - Operações com Matrizes

Potência de Matrizes

```
In[x]:  np.linalg.matrix_power(A,3)
Out[x]: array([[-1.  ,  1.75],
               [-3.5 ,  6.  ]])
```

Potência dos Elementos

```
In[x]:  A**3
Out[x]: array([[ 0. ,  0.125],
               [-1. ,  8.   ]]))
```

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
```

Álgebra Linear com NumPy - Autovalores e Autovetores

Problema de autovalor

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

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

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

Para calcular autovetores e autovetores 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]
```

Álgebra Linear com NumPy - Sistemas Lineares

NumPy dispões 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 $\boldsymbol{Mx} = \boldsymbol{b}$

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

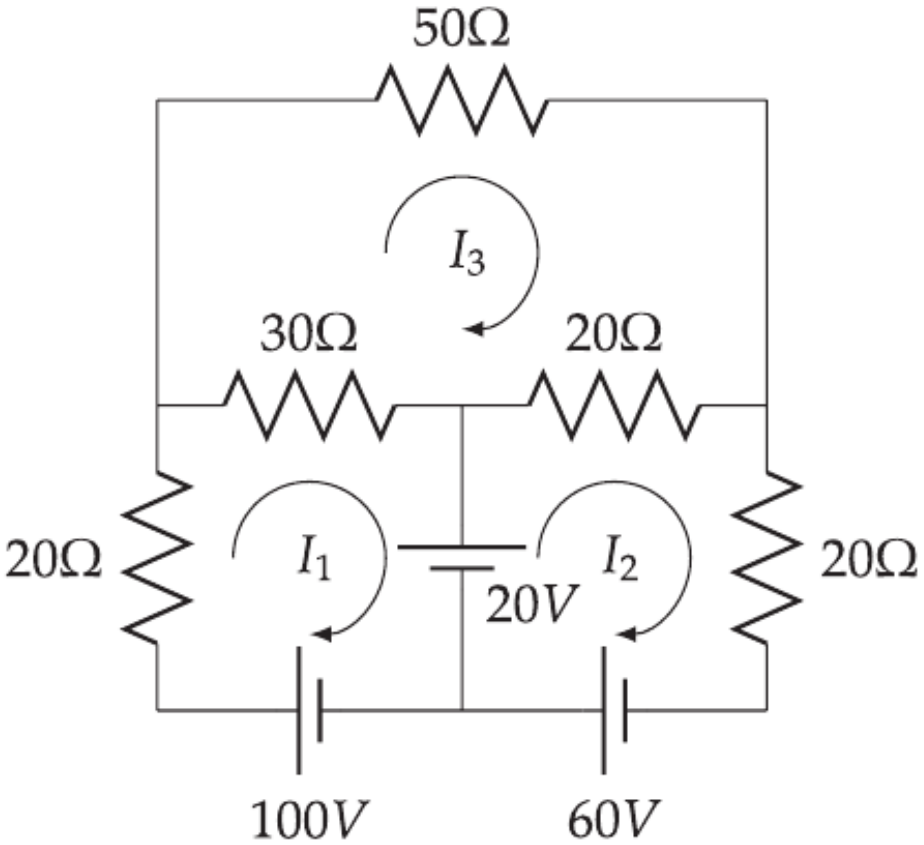
Álgebra Linear com NumPy - Sistemas Lineares

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
```

Álgebra Linear com NumPy - Aplicação

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



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

$$\begin{aligned} 50I_1 - 30I_3 &= 80 \\ 40I_2 - 20I_3 &= 80 \\ -30I_1 - 20I_2 + 100I_3 &= 0 \end{aligned}$$

Importando e Exportando Dados

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.

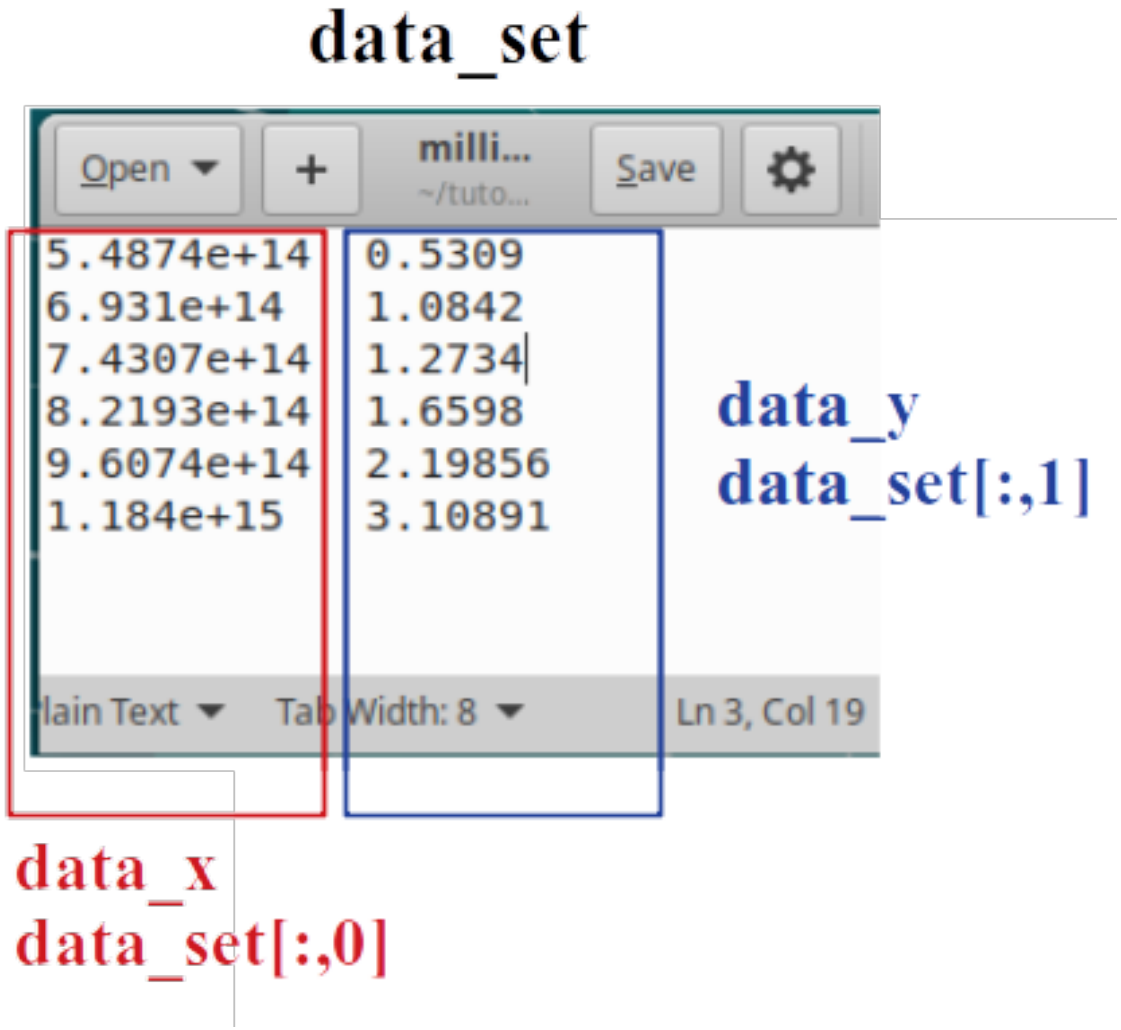
```
import numpy as np
data_set = np.loadtxt("millikan.txt")
data_x = data_set[:,0]
data_y = data_set[:,1]
```

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.



Importando e Exportando Dados

Também é possível importar dados diretamente da web:

```
import urllib
web_file = urllib.request.urlopen("http://www-personal
.edu/~mejn/cp/data/millikan.txt")
data_set = np.loadtxt(web_file)
```

Para ler o arquivo a partir de uma determinada linha, use a opção:
`skiprows=<inteiro>`

Importando e Exportando Dados

Abrindo arquivos com open()

Para abrir arquivos em formatos menos "amigáveis", usamos a função **open()**. O arquivo será lido linha por linha como um **string**. Por exemplo, suponha que um arquivo "HIVseries.csv" tenha os valores separados por vírgula.

```
my_file = open('HIVseries.csv')
data_col1 = []
data_col2 = []
for line in my_file:
    list_data = line.split(',')  #retorma uma lista
    data_col1.append( float(list_data[0]) )
    data_col2.append( float(list_data[1]) )
my_file.close()
```

Importando e Exportando Dados

Salvando dados em um arquivo.

Usando o método savetxt()

```
x = np.linspace(0,1,100)
y = 3*np.sin(x)**3 - np.sin(x)
np.savetxt("x_valores.dat",x)
np.savetxt("xy_values.dat",list(zip(x,y)),fmt="%8.3f")
```

A última linha salva os valores 'x' e 'y' num mesmo arquivo.

Usando *Loops*

```
my_file = open("xy_values.txt","w")  #w=writing
for i in range(x.size):
    my_file.write("{:f}\t\t{:f}\n".format(x[i],y[i]))
my_file.close()
```

Gráficos - Matplotlib

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`.

Gráficos - Gráfico de Funções

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)
```

Gráficos - Legenda

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 tamanha da fonte, use `fontsize=<inteiro>`.

Gráficos - Legenda

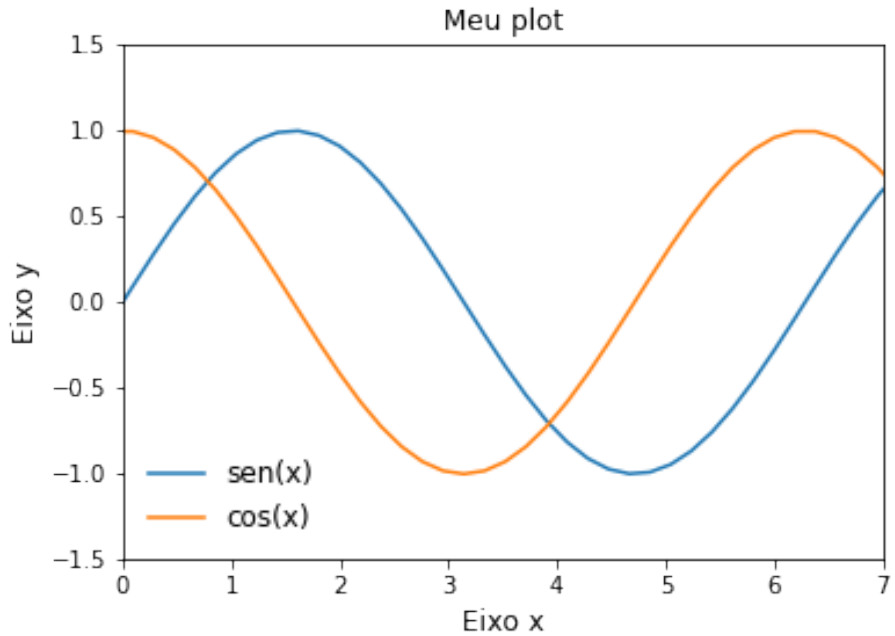
Opções de localização da legenda

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 lilnha)

```
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

Estilos de linha: (-) (–) (:) (-.)

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

Gráfico de Barras

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')
```

SciPy - Raízes de Funções

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 sgn$[f(a)]$ = - 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, `x0` (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 Brent

Vamos encontrar uma das raízes da função abaixo pelo método de Brent

$$f(x) = \frac{1}{5} + x\cos\left(\frac{3}{x}\right).$$

```
In[x]: from scipy.optimize import brentq
In[x]: f = lambda x: 0.2 + x*np.cos(3/x)
In[x]: print(np.sign(f(-0.7)))
       print(np.sign(f(-0.5)))
In[x]: brentq(f,-0.7,-0.5)
Out[x:]: -0.5933306271014237
```

Raízes - Método de Newton

Vamos encontrar a raíz 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
```

SciPy - Ajuste de Curvas

O método `scipy.optimize.curve_fit` é o mais direto e permite passarmos de forma mais 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.

SciPy - Ajuste de Curvas

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 a função Lorentziana a um conjunto de pontos:

$$f(x) = \frac{A\gamma^2}{\gamma^2 + (x - x_0)^2},$$

onde A, γ e x_0 serão os parâmetros ajustados.

Veja solução no código.

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 intergração `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]:
(0.7500000000000002, 1.9132345482589 95e-09)
```

SciPy - Integração Numérica

Para integrar funções com singularidades, devemos passar um 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]:
(3.9999999999999813, 5.6843418860808e-14)
```

SciPy - Integração Numérica - Integrais Múltiplas

Integrais duplas, triplas e múltiplas ($n > 3$) podem ser calculadas, respectivamente, com os métodos `dblquad`, `tplquad` e `nquad`. O método `dblquad` calcula integrais do tipo:

$$I = \int_a^b \int_{g(x)}^{h(x)} f(y,x)dydx.$$

O integrando deve ser definido como uma função de pelo menos duas variáveis, `func(y,x...)`, tomando, **necessariamente**, `y` como primeiro argumento e `x` como segundo. Os limites de integração devem ser passados como flutuantes, `a` e `b`, para a integral na variável `x`, e como **funções** de x para a variável `y`.

SciPy - Equações Diferenciais

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

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{F}(\boldsymbol{y}, t)$$

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

SciPy - Equações Diferenciais

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

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

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

EDOs Acopladas - Implementação

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

Note que agora, `y0` será um sequência de n elementos.

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:

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

$$\frac{dx_1}{dt} = x_2,$$

$$\frac{dx_2}{dt} = -\omega^2 x_1,$$

onde $x_1 = x$ e $x_2 = dx/dt$.

Para Saber Mais

Meu canal no YouTube: Python Para Cientistas
`https://www.youtube.com/user/HeavyState`

Material do minicurso disponível em:
`github.com/aanepomuceno/`