# NumPy

## O que é o NumPy?

O NumPy é uma poderosa biblioteca Python que é usada principalmente para realizar cálculos em Arrays Multidimensionais. O NumPy fornece um grande conjunto de funções e operações de biblioteca que ajudam os programadores a executar facilmente cálculos numéricos. Esses tipos de cálculos numéricos são amplamente utilizados em tarefas como:

Tarefas matemáticas: NumPy é bastante útil para executar várias tarefas matemáticas como integração numérica, diferenciação, interpolação, extrapolação e muitas outras. O NumPy possui também funções incorporadas para álgebra linear e geração de números aleatórios. É uma biblioteca que pode ser usada em conjuto do SciPy e Matplotlib, substituindo o MATLAB quando se trata de tarefas matemáticas.

Processamento de Imagem e Computação Gráfica: Imagens no computador são representadas como Arrays Multidimensionais de números. NumPy torna-se a escolha mais natural para o mesmo. O NumPy, na verdade, fornece algumas excelentes funções de biblioteca para rápida manipulação de imagens. Alguns exemplos são o espelhamento de uma imagem, a rotação de uma imagem por um determinado ângulo etc.

Modelos de Machine Learning: Ao escrever algoritmos de Machine Learning, supõe-se que se realize vários cálculos numéricos em Array. Por exemplo, multiplicação de Arrays, transposição, adição, etc. O NumPy fornece uma excelente biblioteca para cálculos fáceis (em termos de escrita de código) e rápidos (em termos de velocidade). Os Arrays NumPy são usados para armazenar os dados de treinamento, bem como os parâmetros dos modelos de Machine Learning.

### Instalando

Existem diversas formas de instalar o numpy. A mais simples é instalar o pacote Anaconda (<https://www.anaconda.com/distribution/>) que já vem com o Python e diversas bibliotecas científicas e ciência de dados instaladas.

Outra forma, caso você já tenha o python instalado mas não o numpy, é o utilizar o gerenciador e pacotes pip, através do comando no seu terminal:

$ pip install numpy

## Explorando a API do NumPy

### Importando numppy com um alias np

np é uma abreviação amplamente utilizada na comunidade python para o numpy.

import numpy as np

### 1D arrays

Array unidimensional, também chamado de vetor ou até mesmo matriz de 1 dimensão:

>>> a = np.array([1, 2, 3])

array([1, 2, 3])

Checando o tipo da variável a:

>>> type(a)

numpy.ndarray

o nd significa n-dimensional

### Checando o tipo de dados do array

Diversos tipos de dados são possíveis em um array numpy, os mais comuns são os numéricos:

* int32
* int64
* float32
* float64

>>> a.dtype

dtype('int64')

#### A consistência do dado é forte...

Se trocarmos um elemento na posição 0 para o valor 10, dará certo:

>>> a[0] = 10

>>> a

array([10, 2, 3])

Se trocarmos para ponto flutuante, o numpy irá truncar a parte decimal, dado que o array que criamos é inteiro.

>>> a[0] = 1.2

>>> a

array([1, 2, 3])

### 2D arrays

Matrizes podem ser consideradas um array de 2 dimensões.

Observação

O NumPy possui também uma estrutura, matrix, mas não é recomendado utilizá-la pela própria [documentação oficial](https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html) e poderá ser removida no futuro.

Para criar uma matriz, basta aninhar múltiplas listas dentro de uma lista, como o exemplo a seguir:

>>> b = np.array([[9.0, 8.0, 7.0],

[6.0, 5.0, 4.0]])

>>> b

array([[9., 8., 7.],

[6., 5., 4.]])

### Propriedades

#### Dimensão e formato

Dois conceitos importantes já mencionados acima é o de dimensão e formato.

Para descobrir essas informações, basta acessar os atributos ndim e shape

>>> a.ndim

1

>>> a.shape

(3,)

>>> b.ndim

2

b.shape

(2, 3)

#### Tipo de dado e tamanho

Na sessão Checando o tipo de dados do array já foi dito dos tipos de dados, mas agora falaremos da diferença de tamanhos que isso ocupa na memória.

Então, temos as variáveis a e b criadas anteriormente com os seguintes tipos:

>>> a.dtype

dtype('int64')

>>> b.dtype

dtype('float64')

Por padrão, se o python instalado é 64 bits, ele irá criar tipos int ou float de 64 bits. Caso seu python fosse 32 bits, seria int32 e float32.

Vamos criar uma outra array, a16, com o tipo inteiro de 16 bits.

>>> a16 = np.array([1, 2, 3], dtype=np.int16)

>>> a16

array([1, 2, 3], dtype=int16)

Note que por ser um tipo diferente do padrão, ele ressalta ao imprimir.

Para descobrir quanto cada elemento individualmente ocupa na memória, podemos acessar o atributo itemsize:

>>> a.itemsize #

8

Ele retorna 8 e não 64! Isso é porque ele já converteu os bits para bytes. Bytes é o conjunto de 8 bits.

Logo:

648=8

Já nosso array int16, temos:

>>> a16.itemsize

2

Dado que:

168=2

# quantidade de elementos total

>>> a.size

3

Quantidade de elementos vezes o tamanho de cada elemento nos dará o tamanho total de bytes que o array inteiro ocupa:

>>> a.size \* a.itemsize

24

Mas ao invés de calcular isso, podemos simplesmente acessar o atributo nbytes, que já é o tamanho total de bytes ocupado pelo array:

>>> a.nbytes

24

Observação

Geralmente não é necessário em reduzir o número de bits a não ser que você tenha certeza que um tamanho reduzido vai atender sua necessidade e você quer ser extremamente eficiente.

### Acessando e modificando elementos (Indexing & Slicing)

Dada a matriz a abaixo:

>>> a = np.array([[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14]])

>>> print(a)

[[ 1 2 3 4 5 6 7]

[ 8 9 10 11 12 13 14]]

Podemos acessar um elemento específico de forma similar a lista, utilzando a sintaxe de colchetes, com a diferença de que podemos separar cada posição com vírgulas.

Para uma array 2D, temos então a sintaxe:



Exemplo:

>>> a[1, 5]

13

Podemos (menos comum) fazer dessa forma também:

>>> a[1][5]

13

Usar números negativsos funciona de trás pra frente a indexação:

>>> a[1, -2]

13

Pegar uma linha específica, podemos utilizar a sintaxe de : que pode ser lida como "todos" daquela dimensão (colunas).

Podemos ler então como: linha zero, todas as colunas

>>> a[0, :]

array([1, 2, 3, 4, 5, 6, 7])

Também podemos fazer assim simplesmente:

>>> a[0]

array([1, 2, 3, 4, 5, 6, 7])

Porém, para coluna específica não tem jeito, precisamos usar :.

Leia-se: todas as linhas, coluna 2

>>> a[:, 2]

array([ 3, 10])

O operador : também conhecido como slicing, aceita o parâmetro:

* start
* end
* step

No formato

[startindex:endindex:stepsize]

O stepsize basicamente é quantos elementos deve ser pular. Podemos pegar do elemento 1 ao 6 pulando de 2 em 2 por exemplo da linha 0.

>>> a[0, 1:6:2]

array([2, 4, 6])

Funciona com negativo também:

a[0, 1:-1:2]

>>> array([2, 4, 6])

Para mudar um elemento específico, basta usar o operador =:

>>> a[1,5] = 20

>>> print(a)

[[ 1 2 3 4 5 6 7]

[ 8 9 10 11 12 20 14]]

Mudando uma coluna inteira para ser 5:

>>> a[:, 2] = 5

>>> print(a)

[[ 1 2 5 4 5 6 7]

[ 8 9 5 11 12 20 14]]

Isso mostra uma característica fundamental do array do NumPy:

**Ao alterar o pedaço da matriz recortada, você altera a matriz original**

Slicing em listas geram cópias!

>>> a = [1, 2, 3]

>>> b = a[1:]

>>> b

[2, 3]

>>> b = [10, 11]

>>> a

[1, 2, 3]

>>> b

[10, 11]

Acessando o formato de um slicing:

>>> a[:, 2].shape

(2,)

>>> a[:, 2] = [5, 10]

>>> print(a)

[[ 1 2 5 4 5 6 7]

[ 8 9 10 11 12 20 14]]

#### Exemplo 3D

>>> b = np.array([[[1, 2], [3,4]], [[5, 6], [7, 8]]])

>>> b

array([[[1, 2],

[3, 4]],

[[5, 6],

[7, 8]]])

Checando a dimensão:

>>> b.ndim

3

Retirando o elemento 4:

>>> b[0, 1, 1]

4

Pegando todos todos os elementos da posição 1 da dimensão 2:

>>> b[:, 1, :]

array([[3, 4],

[7, 8]])

Substituindo:

b[:, 1, :] = [[9, 9], [8, 8]]

### Inicializando arrays usando métodos internos

O NumPy já possui diversos métodos built-in para gerar arrays dos mais diversos tipos

#### array apenas com zeros

>>> np.zeros(5)

array([0., 0., 0., 0., 0.])

É possível gerar um array de qualquer formato, basta apenasr passar o formato como uma sequência (lista, tupla geralmente) como argumento

>>> np.zeros((2, 3))

array([[0., 0., 0.],

[0., 0., 0.]])

>>> np.zeros((2, 2, 3))

array([[[0., 0., 0.],

[0., 0., 0.]],

[[0., 0., 0.],

[0., 0., 0.]]])

#### array apenas com uns

>>> np.ones((4, 2, 2), dtype='int32')

array([[[1, 1],

[1, 1]],

[[1, 1],

[1, 1]],

[[1, 1],

[1, 1]],

[[1, 1],

[1, 1]]], dtype=int32)

Um coisa muito comum é usar o np.ones para criar uma matriz de qualquer número fazendo a operação, exemplo:

>>> np.ones((2, 2)) \* 10

array([[10., 10.],

[10., 10.]])

Mas o numpy já tem uma opção mais elegante, o full:

>>> np.full((2, 2), 99)

array([[99, 99],

[99, 99]])

Qualquer outro número copiando o formato de uma matriz existente

>>> np.full\_like(a, 4)

array([[4, 4, 4, 4, 4, 4, 4],

[4, 4, 4, 4, 4, 4, 4]])

#### Números decimais aleatórios

O numpy tem um sub-módulo chamado random, que pode ser acessando via np.random.

Nele, tem um conjunto de funções para números aleatórios.

>>> np.random.random\_sample((4, 2))

array([[0.59701743, 0.40627135],

[0.23940108, 0.58528558],

[0.04228252, 0.68467361],

[0.68907347, 0.85636431]])

Função igual a de cima, mas padrão do MATLAB (observe que não é uma tupla/lista o formato):

>>> np.random.rand(4, 2)

array([[0.13322589, 0.76493887],

[0.06716679, 0.61834428],

[0.54098045, 0.09752974],

[0.6656696 , 0.79337782]])

Números inteiros:

>>> np.random.randint(7, size=(3, 3))

array([[6, 5, 6],

[0, 6, 2],

[4, 6, 5]])

Os argumentos principais são low, high e size, exemplo: criando uma matriz de 0 a 99 de 100 elementos:

>>> np.random.randint(0, high=100, size=100)

array([66, 91, 70, 95, 73, 73, 70, 98, 10, 70, 47, 11, 67, 69, 60, 30, 23,

51, 58, 18, 79, 46, 8, 12, 84, 45, 44, 90, 13, 86, 92, 20, 9, 93,

41, 7, 56, 39, 11, 58, 36, 94, 80, 65, 94, 9, 20, 47, 74, 5, 90,

28, 3, 86, 29, 51, 83, 44, 58, 33, 55, 69, 9, 15, 54, 87, 87, 68,

35, 6, 91, 98, 99, 81, 73, 37, 86, 79, 91, 65, 44, 22, 68, 5, 85,

28, 49, 70, 64, 23, 23, 73, 94, 56, 76, 11, 12, 21, 14, 48])

Para incluir o 100, basta trocar o high por 101

#### Matriz identidade

Diagonal inteira com 1. É sempre uma matriz quadrada.

>>> np.identity(3)

array([[1., 0., 0.],

[0., 1., 0.],

[0., 0., 1.]])

#### repeat

Método para repetir uma determinada array na direção do eixo escolhido.

Esse é a primeira função, de várias, que possui o parâmetro axis.

Diversas vezes o numpy permite fazer uma operação, nesse caso, repeat, no qual é opcional ou necessário dizer qual o eixo da operação.

Para um vetor de 1D, temos apenas 1 eixo, mas para matrizes, tempos dois:

O eixo 0 é linha, o eixo 1 é coluna

>>> arr = np.array([1, 2, 3])

>>> r1 = np.repeat(arr, 3)

>>> print(r1)

[1 1 1 2 2 2 3 3 3]

Com axis = 0

>>> arr = np.array([[1, 2, 3]])

>>> r1 = np.repeat(arr, 3, axis=0)

>>> print(r1)

[[1 2 3]

[1 2 3]

[1 2 3]]

Com axis = 1

>>> arr = np.array([[1, 2, 3]])

>>> r1 = np.repeat(arr, 3, axis=1)

>>> print(r1)

[[1 1 1 2 2 2 3 3 3]]

#### arange

Função que retorna elementos igualmente espaçados num step (por padrão, 1) dentro de um certo intervalo.

>>> np.arange(0, 10)

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Step diferente de 1:

>>> np.arange(0, 11, 0.1)

array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ,

1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1,

2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2,

3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4. , 4.1, 4.2, 4.3,

4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. , 5.1, 5.2, 5.3, 5.4,

5.5, 5.6, 5.7, 5.8, 5.9, 6. , 6.1, 6.2, 6.3, 6.4, 6.5,

6.6, 6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4, 7.5, 7.6,

7.7, 7.8, 7.9, 8. , 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7,

8.8, 8.9, 9. , 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8,

9.9, 10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9])

#### linspace

Parecido com o arange, mas você diz quantos pontos você quer e o intervalo e ele define o espaçamento linear

>>> np.linspace(0, 100, num=10)

array([ 0. , 11.11111111, 22.22222222, 33.33333333,

44.44444444, 55.55555556, 66.66666667, 77.77777778,

88.88888889, 100. ])

### Tenha cuidado ao copiar arrays!

#### Jeito errado

>>> a = np.array([1, 2, 3])

>>> b = a

>>> b

array([1, 2, 3])

>>> b[0] = 100

>>> b

array([100, 2, 3])

>>> a

array([100, 2, 3])

a também foi modificado!

#### Jeito certo (seguro)

>>> a = np.array([1, 2, 3])

>>> b = a.copy()

>>> b

array([1, 2, 3])

b[0] = 100

>>> b

array([100, 2, 3])

>>> a

array([1, 2, 3])

### Matemática

O numpy te fornece um conjunto de funções matemáticas:

>>> a = np.array([1, 2, 3, 4])

>>> a

array([1, 2, 3, 4])

#### Operação com escalares

Soma:

>>> a + 2

array([3, 4, 5, 6])

Subtração:

>>> a - 2

array([-1, 0, 1, 2])

Multiplicação:

>>> a \* 2

array([2, 4, 6, 8])

Divisão:

>>> a / 2

array([0.5, 1. , 1.5, 2. ])

Incrementar:

>>> a += 2

>>> a

array([3, 4, 5, 6])

Potência:

>>> a \*\* 2

array([ 1, 4, 9, 16])

#### Operação entre arrays

Tudo que você consegue fazer com escalar, você consegue fazer com arrays elemento-a-elemnto, por exemplo, para soma:

>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([1, 0, 1, 0])

>>> a + b

array([2, 2, 4, 4])

#### Funções matemáticas

Função seno:

>>> np.sin(a)

array([ 0.84147098, 0.90929743, 0.14112001, -0.7568025 ])

Funçõa cosseno:

>>> np.cos(a)

array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])

### Álgebra Linear

Da definição do Wikipédia:

Álgebra linear é um ramo da matemática que surgiu do estudo detalhado de sistemas de equações lineares, sejam elas algébricas ou diferenciais. A álgebra linear utiliza alguns conceitos e estruturas fundamentais da matemática como vetores, espaços vetoriais, transformações lineares, sistemas de equações lineares e matrizes.

O numpy nos permite executar diversas diversas operações de álgebra linear, mostradas a seguir:

>>> a = np.ones((2, 3))

>>> print(a)

[[1. 1. 1.]

[1. 1. 1.]]

>>> b = np.full((3, 2), 2)

>>> print(b)

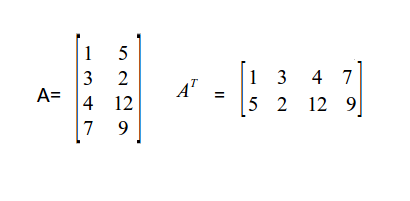
[[2 2]

[2 2]

[2 2]]

#### Transposição

A operação de transposição



Pode ser feita da seguinte forma:

>>> np.transpose(a)

array([[1., 1.],

[1., 1.],

[1., 1.]])

Ou acessando o atributo T:

>>> a.T

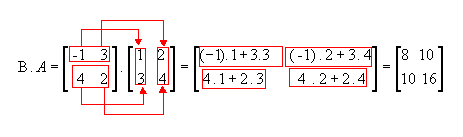
array([[1., 1.],

[1., 1.],

[1., 1.]])

#### Multiplicação de matrizes

A tradicional multiplicação de matrizes, como mostra a imagem abaixo:



Pode ser feita no numpy simplesmente chamando matmul

>>> np.matmul(a, b)

array([[6., 6.],

[6., 6.]])

Operador @ executa a função anterior:

>>> a @ b

array([[6., 6.],

[6., 6.]])

#### Encontrar o determinante

>>> c = np.identity(3)

>>> print(c)

[[1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.]]

>>> np.linalg.det(c)

1.0

Outras funcções de Álgebra Linear: <https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

* Trace
* Decomposição de vetores
* Autovalor/autovetor
* Norma da Matriz
* Inversa
* Etc...

### Estatística

O numpy vem com várias funções básicas de estatística, como mínimo, máximo, média, mediana, etc.

>>> stats = np.array([[1, 2, 3], [4, 5, 6]])

>>> stats

array([[1, 2, 3],

[4, 5, 6]])

>>> np.min(stats)

1

>>> np.max(stats)

6

Mínimo por linha:

>>> np.min(stats, axis=1)

array([1, 4])

Máximo por coluna:

>>> np.max(stats, axis=0)

array([4, 5, 6])

Soma por coluna:

>>> np.sum(stats, axis=0)

array([5, 7, 9])

Média:

>>> np.average(stats)

3.5

### Reorganizar Array

Muitas vezes você quer mudar o formato de array, por exemplo, de 4 elementos pra uma matriz 2x2, ou situações similares.

Para isso, você pode utilizar a função reshape.

#### Reshape

>>> before = np.array([[1,2,3,4],[5,6,7,8]])

>>> print(before.shape)

(2, 4)

>>> after = before.reshape((8, 1)) # tem que possuir a mesma quantidade!

>>> after

array([[1],

[2],

[3],

[4],

[5],

[6],

[7],

[8]])

>>> before.reshape(2, 2, 2)

array([[[1, 2],

[3, 4]],

[[5, 6],

[7, 8]]])

#### Anexar verticalmente os vetores

v1 = np.array([1,2,3,4])

v2 = np.array([5,6,7,8])

>>> np.vstack([v1, v2])

array([[1, 2, 3, 4],

[5, 6, 7, 8]])

>>> np.vstack([v1, v2, v2, v2])

array([[1, 2, 3, 4],

[5, 6, 7, 8],

[5, 6, 7, 8],

[5, 6, 7, 8]])

#### Apendar horizontalmente os vetores

De forma similar ao anterior:

>>> h1 = np.ones((2, 4))

>>> h2 = np.zeros((2, 2))

>>> np.hstack((h1, h2))

array([[1., 1., 1., 1., 0., 0.],

[1., 1., 1., 1., 0., 0.]])

## Funcionalidades extras

### Carregar dados de um arquivo

Vamos supor que temos um arquivo data.txt com o seguinte conteúdo:

1,13,21,11,196,75,4,3,34,6,7,8,0,1,2,3,4,5

3,42,12,33,766,75,4,55,6,4,3,4,5,6,7,0,11,12

1,22,33,11,999,11,2,1,78,0,1,2,9,8,7,1,76,88

Podemos gerar uma matriz a partir desse arquivo da seguinte forma:

>>> filedata = np.genfromtxt('data.txt', delimiter=',')

>>> filedata

array([[ 1., 13., 21., 11., 196., 75., 4., 3., 34., 6., 7.,

8., 0., 1., 2., 3., 4., 5.],

[ 3., 42., 12., 33., 766., 75., 4., 55., 6., 4., 3.,

4., 5., 6., 7., 0., 11., 12.],

[ 1., 22., 33., 11., 999., 11., 2., 1., 78., 0., 1.,

2., 9., 8., 7., 1., 76., 88.]])

O primeiro argumento é o nome do arquivo.

Importante ressaltar o segundo key argumento, delimiter, no qual você especifica o que separa cada número individualmente no arquivo. Nesse caso, vírgula, mas podería ser ; por exemplo, espaços, ou tabs.

Podemos notar também que o numpy converteu para float nossos números, apesar de todos serem inteiros. Ele faz isso como uma medida preventiva dado que ele não sabe ao ler o arquivo qual tipo de dado que é.

Podemos converter manualmente para inteiro usando a função astype:

>>> filedata.astype('int32')

>>> array([[ 1, 13, 21, 11, 196, 75, 4, 3, 34, 6, 7, 8, 0,

1, 2, 3, 4, 5],

[ 3, 42, 12, 33, 766, 75, 4, 55, 6, 4, 3, 4, 5,

6, 7, 0, 11, 12],

[ 1, 22, 33, 11, 999, 11, 2, 1, 78, 0, 1, 2, 9,

8, 7, 1, 76, 88]], dtype=int32)

Podemos também salvar uma matriz de uma forma mais otimizada não textual (binária) para uso futuro.

Isso gera um arquivo binário que inclusive salva o tipo de dado, nesse caso, int32.

Quando lido, vai converter corretamnete o tipo daquele dado.

np.save('data', filedata.astype('int32'))

Igual ao método de cima, porém comprime os dados (economiza espaço em disco, porém é um pouco mais lento pra ler).

np.savez\_compressed('dataz', filedata)

Para ler os dados que acabamos de salvar, basta usar o np.load:

np.load('data.npy')

array([[ 1, 13, 21, 11, 196, 75, 4, 3, 34, 6, 7, 8, 0,

1, 2, 3, 4, 5],

[ 3, 42, 12, 33, 766, 75, 4, 55, 6, 4, 3, 4, 5,

6, 7, 0, 11, 12],

[ 1, 22, 33, 11, 999, 11, 2, 1, 78, 0, 1, 2, 9,

8, 7, 1, 76, 88]], dtype=int32)

### Máscara Boleana e Seleção Avançada

Conceito super importante no numpy e no pandas é o de máscara booleana.

Ao aplicar qualquer operador booleano

* >
* <
* <=
* >=
* ==
* in

o numpy retorna um array de True e False no qual ele aplicou elemento a elemento aquele operador.

Imagine para a matriz abaixo:

>>> mat = np.array([1, 10, 20, 30]).reshape(2, 2)

>>> arr

array([[ 1, 10],

[20, 30]])

Eu quero saber todos os elementos maiores que 10, eu posso aplicar:

>>> mat > 10

array([[False, False],

[ True, True]])

Me é retornado uma matriz de formato (2, 2) assim como com True na posição dos elementos que são maiores que 10.

Podemos então fazer essa operação dentro dos colchetes de seleção:

>>> mat[mat > 10]

array([20, 30])

E será retornado um array com os elementos 20 e 30 como esperado.

Podemos fazer operações linha a linha ou coluna a coluna através de métodos auxiliares como any ou all:

* any: se qualquer elemento da linha for True, retorna True
* all: todos os elementos tem que ser True para retornar True

Exemplos:

Por coluna:

>>> np.any(mat > 10, axis=0)

array([ True, True])

Por linha:

>>> np.any(mat > 10, axis=1)

array([False, True])

#### Operador AND

Similar ao and do python, podemos usar múltiplas condições para filtrar dados da nossa matriz com o operador &.

>>> filt = (mat > 10) & (mat <= 20)

>>> mat[filt]

array([20])

Observação: note que os colchetes além de melhorarem a legibilidade, são necessárias devido a ordem de precedência dos operadores python. Se não colocarmos os colchetes, dará um erro.

#### Operador OR

Similar ao or, só que devemos utilizar |:

>>> filt = (mat == 1) | (mat >= 20)

>>> mat[filt]

array([1, 20, 30])

#### Operador NOT

Similar ao not, mas devemos utilizar um til ~

>>> filt = (mat == 1) | (mat >= 20)

>>> mat[~filt]

array([10])

array([[ True, True, True, True, True, False, True, True, True,

True, True, True, True, True, True, True, True, True],

[ True, True, True, True, True, False, True, False, True,

True, True, True, True, True, True, True, True, True],

[ True, True, True, True, True, True, True, True, False,

True, True, True, True, True, True, True, False, False]])

### Seleção passando listas

Podemos selecionar elementos específicos de um array passando uma lista de posições:

>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a[[1, 2, 8]]

array([2, 3, 9])

## 1. Pandas

O pandas é uma biblioteca muito versátil e simples de se utilizar quando vamos trabalhar com tabelas.  
Para entender o funcionamento do pandas precisamos entender suas estruturas base, a Series e o DataFrame.

### 1.1 Series

As Series são basicamente as colunas das tabelas e armazenam suas informações com o numpy array. Neste caso, porém, esse array possuirá um índice associado, permitindo o acesso aos conteúdos dessa estrutura por ele.

#### 1.1.1 Criação de uma Series

A partir de uma lista  
Vamos ver abaixo, como criar uma Series à partir de uma lista:

import pandas as pd

minha\_lista = [10, 20, 30]

serie = pd.Series(minha\_lista)

O conteúdo de serie será uma pd.Series com os elementos da lista e como não foram definidos índices, os valores serão default, a numeração sequencial iniciada por zero.

A partir de um Array  
O processo é igual ao anterior, com a diferença que ao invés de utilizarmos uma lista, utilizaremos um np.array.

Criação de uma Series com índice  
Nesse caso, iremos passar duas listas, uma contendo os valores e outra contendo os índices:

import pandas as pd

labels = ['a', 'b', 'c', 'd']

valores = [10, 20, 30, 40]

serie = pd.Series(data=valores, index=labels)

#### 1.1.2 Acessando elementos da Series

Pelo índice  
Uma das maneiras de se acessar valores das nossas Series, é sabendo em qual índice eles se encontram:

import pandas as pd

labels = ['a', 'b', 'c', 'd']

valores = [10, 20, 30, 40]

serie = pd.Series(data=valores, index=labels)

serie['c']

Nesse exemplo acessamos o elemento 30, que está associado ao índice c.

Utilizando filtros  
Podemos aplicar filtros para selecionar apenas os elementos que satisfaçam determinada condição.  
No exemplo abaixo, iremos selecionar apenas os elementos que sejam maiores que 15:

import pandas as pd

labels = ['a', 'b', 'c', 'd']

valores = [10, 20, 30, 40]

serie = pd.Series(data=valores, index=labels)

serie[serie > 15]

Note que serie > 15 nos retorna uma series com elementos True e False, caso os elementos da serie satisfaçam a condição. Ao utilizar esse comando dentro dos colchetes, serie[serie > 15], estamos selecionado apenas os elementos que satisfazem a condição.

#### 1.1.3 Métodos

O Pandas possui diversos métodos que podem ser utilizados nessa estrutura.  
Abaixo estão alguns métodos que essa estrutura de dados possui e facilitam alguns cálculos:

| Método | Descrição |
| --- | --- |
| sum | soma |
| mean | média |
| std | desvio padrão |
| mode | moda |
| max | valor máximo |
| min | valor mínimo |
| value\_counts | contagem de valores |
| describe | estatísticas básicas |

Exemplos de utilização

1) Neste exemplo iremos utilizar o método sum para somar os valores da série.

import pandas as pd

valores = [1, 1, 2, 3, 5, 8, 13]

fibonacci = pd.Series(valores)

fibonacci.sum()

2) Podemos utilizar também os filtros, de maneira a soma apenas os valores maiores que 4.

import pandas as pd

valores = [1, 1, 2, 3, 5, 8, 13]

fibonacci = pd.Series(valores)

fibonacci[fibonacci > 4].sum()

### 1.2 DataFrame

O DataFrame é a estrutura que se assemelha à tabela. Ela é representada por um dicionário em que a chave é o nome da coluna e os valores são as Series (todas com mesmo índice).

#### 1.1.1 Criação de um DataFrame

Existem diversas maneiras de se criar um dataframe, pode ser à partir de listas, dicionários etc.  
Um dos modos mais comuns é a criação à partir da leitura de um arquivo do formato .csv, como veremos à seguir para o caso do dataset titanic, muito conhecido por quem trabalha com data science.

Dataframe à partir de um csv

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

Note que titanic\_df é um DataFrame com os dados do arquivo titanic.csv localizados em ../datasets/. Ou seja, o parâmetro do método pd.read\_csv é o arquivo (com a localização) que se deseja ler. Existem outros parâmetros, mas não entraremos neles neste momento.

Observação:  
Note que podemos também utilizar um arquivo do formato .xlsx, natural do excel.  
Para tanto, devemos utilizar o método pd.read\_excel.

DataFrame à partir de um dicionário  
Este é um método muito útil, pois a estrutura do dicionário é bem semelhante à de um DataFrame. Neste caso, cada chave do nosso dicionário se tornará uma coluna e os valores (que podem ser na forma de listas, arrays, series...) serão os elementos do DF.

import pandas as pd

dicionario = {

'coluna\_A': [1, 2, 3, 4, 5],

'coluna\_B': ['a', 'b', 'c', 'd', 'e'],

'coluna\_C': [0.5, 1.5, 4.5, 6.5, 8.5]

}

df = pd.DataFrame(dicionario)

#### 1.1.2 Acessando elementos do DataFrame

Existem diversas maneiras de se acessar valores de um DataFrame, veremos a seguir algumas maneiras principais de o fazer.

Selecionando apenas algumas colunas  
Esse é o método mais simples, entretanto muito útil, para se selecionar apenas algumas colunas da nossa estrutura.  
No exemplo que segue, iremos selecionar apenas as colunas PassengerId, Name, Sex e Survived.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic[['PassengerId', 'Name', 'Sex', 'Survived']]

Acessando pela posição  
O método a seguir se chama iloc, com ele podemos acessar os elementos do df em questão através das posições das linhas e colunas.  
Neste exemplo, iremos selecionar o elemento da linha 2 e coluna 5.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.iloc[1, 4]

Acessando pelos índices De maneira bem semelhante à anterior, podemos acessar os elementos pelos índices e pelos nomes das colunas utilizando o método loc. Neste exemplo, iremos selecionar o elemento do índice 4 (linha 5) e coluna 'Name'.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.loc[4, 'Name']

Utilizando filtros  
De maneira análoga às Series, podemos utilizar os filtros também nos DataFrames. No exemplo abaixo, iremos selecionar apenas os passageiros que tenham mais que 18 anos:

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic[df\_titanic['Age'] > 18]

De maneira alternativa, podemos utilizar o método query do Pandas, como segue:

df\_titanic.query('Age > 19')

#### 1.1.3 Tratando os dados

É muito comum num conjunto de dados, seja ele proveniente de um banco dados ou de um arquivo de texto, existirem valores nulos. Para fins de análises/modelos é muito importante identificar a incidência desses valores e tomar alguma atitude, seja a de remover os valores nulos, ou a de substituí-los. Veremos abaixo como os fazer:

Identificando Elementos Nulos por Coluna  
Identificar a quantidade de nulos por coluna é muito importante, pois assim podemos identificar qual ação é mais adequada.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.isnull().sum()

Removendo os valore Nulos  
Para remover os nulos, iremos utilizar o comando dropna, como segue:

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.dropna()

Substituindo Valores Nulos  
Como muitas vezes não queremos diminuir o tamanho do nosso conjunto de dados e mesmo assim utilizá-los (muitos modelos não aceitam valores nulos), uma abordagem é substituir esses valores (seja pela média dos valores, pela moda etc.).  
Para fazer isso, utilizaremos o método fillna, em que o parâmetro passado será o valor de substituição.  
Neste exemplo irei substituir os valores por -1, mas poderia ser qualquer outro valor, até mesmo uma string.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.fillna(-1)

#### 1.1.4 Agrupando valores

Quando estamos fazendo análises no nosso conjunto de dados, é muito útil saber alguns comportamentos dados pela combinação de duas ou mais variáveis.  
Para tanto, vamos utilizar o método groupby do Pandas.

Utilizando o Groupby  
Neste exemplo, iremos analisar a média de idade por sexo.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.groupby('Sex')['Age'].mean()

Note que nesse exemplo utilizamos a média, mas poderíamos utilizar outras funções de agregação que vimos no item 1.1.3, os métodos das Series.

O comando pivot\_table  
Note que podemos querer cruzar mais informações, como por exemplo a média de idade por sexo e classe.  
Para tanto utilizamos o método pivot\_table, que é semelhante à tabela dinâmica do Excel.

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

pd.pivot\_table(df\_titanic, values='Age', index='Pclass', columns='Sex', aggfunc=np.mean)

#### 1.1.5 Salvando os DFs em arquivos

É muito útil, depois de se tratar um conjunto de dados, salvar esse DataFrame num arquivo de texto.  
Veremos à seguir duas maneiras de se salvar, num .csv e num .xlsx.

Salvando em csv

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.to\_csv('arquivo\_com\_dataframe.csv')

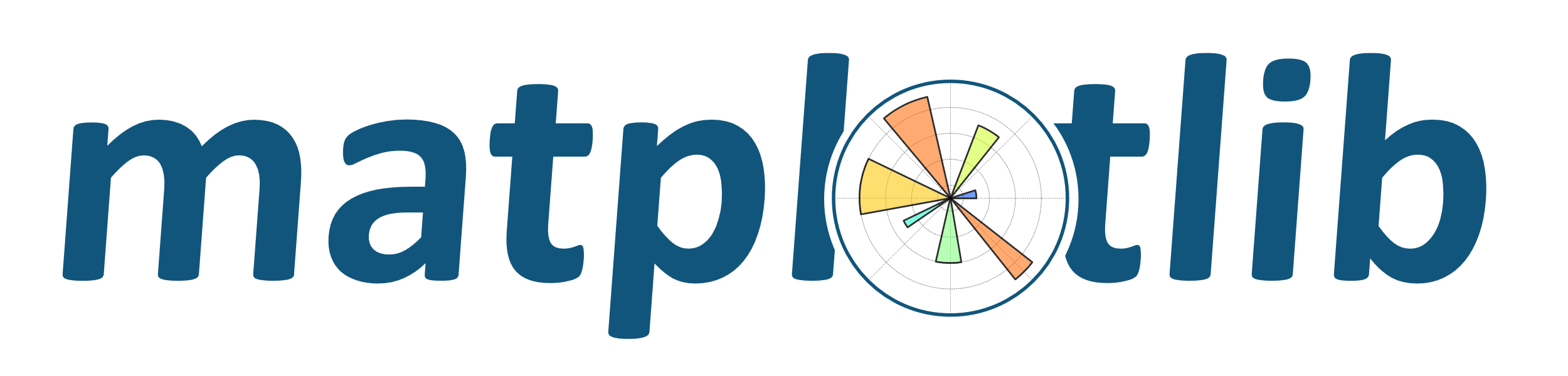
Salvando em xlsx

import pandas as pd

df\_titanic = pd.read\_csv('../datasets/titanic.csv')

df\_titanic.to\_excel('arquivo\_com\_dataframe.csv')

# Matplotlib



Assim como o NumPy e o Pandas, o [matplotlib](http://matplotlib.org/) é uma das bibliotecas que fazem parte do núcleo do [SciPy](https://www.scipy.org/) para visualização de dados 2-D.

## Antes de tudo

### Instalação

Se você tem o Anaconda instalado, você provavelmente já possui a biblioteca instalada.

Para checar se você já tem, no Jupyter Notebook ou interpretador do python, rode:

import matplotlib

Se não der erro, é porque está tudo ok!

Caso você tenha um erro, será necessário instalar. Para isso, rode o comando no prompt/terminal que possua acesso ao comando:

pip install matplotlib

Em caso de problemas nessa parte, entre em contato no chat da sua turma ou no Q&A.

### Importando

No Jupyter Notebook, para de fato utilizar o matplotlib, precisamos importar desse jeito:

import matplotlib.pyplot as plt

matplotlib.pyplot é uma coleção de funções que fazem o matplotlib funcionar como [MATLAB](https://www.mathworks.com/products/matlab.html). Cada função, ao chamada, faz algumas alterações em uma figura, exemplos:

* criar uma figura
* criar uma área de plotagem em uma figura
* gerar linhas em uma área de plotagem
* decorar o gráfico com rótulos, título, etc.

Caso você esteja no Jupyter Notebook, também adicione:

%matplotlib inline

Esse é um comando específico para o ambiente Jupyter Notebook, para mostrar os gráficos nas células do notebook.

Graças a ele, não precisamos (mas se chamar, não tem problema) ao final de cada gráfico o comando:

plt.show()

Para mostrar o gráfico.

## Básico

### Gráfico de linha

Para um gráfico de linha simples, podemos chamar o comando plt.plot passando os dados que queremos visualizar.

Pela [documentação da função plot](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html), precisamos passar dois argumentos para essa função

* x
* y

Que comumente são 1D arrays.

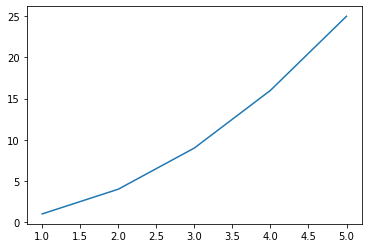
x = [1, 2, 3, 4, 5]

y = [n \*\* 2 for n in x]

plt.plot(x, y)

plt.show()

Esse código irá resultar no seguinte gráfico:



No exemplo acima criamos duas listas. Na lista x passamos valores de um até cinco e na lista quadrados usamos list comprehension para gerar uma lista com os quadrados da primeira. Depois disso utilizamos plt.plot(x, y) que irá gerar a figura. No nosso caso os valores do eixo-x passados são os numeros e no eixo-y os os mesmos valores do eixo-x elevados ao quadrado.

Em outras palavras, essa é a visualização da função

f(x)=x2parax∈[1,5]

### Múltiplos gráficos

#### Estrutura

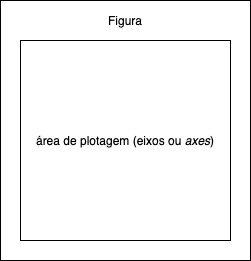
No matplotlib, temos dois conceitos importantes:

* Área de plotagem (eixos ou Axes)
  + onde de fato os gráficos (linhas, barras, pontos, etc) aparecem. Podemos pensar como o eixo-x e eixo-y dos gráficos.
* Figura ou Figure
  + onde os eixos "vivem". Podemos pensar como o PNG/JPG final dos múltiplos gráficos que geramos.

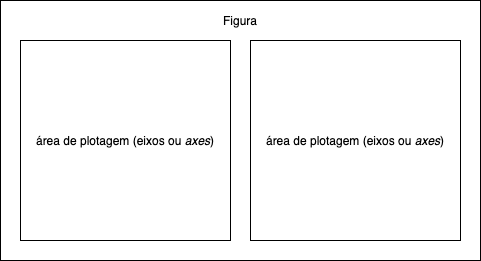
Antes de gerarmos múltiplos gráficos, precisamos decidir:

* podemos imprimir dois gráficos, ex: duas linhas, na mesma área de plotagem
* ou em áreas de plotagens diferentes.

1 eixo - 1 figura



2 eixos - 1 figura



#### Novos dados

Para visualizar múltiplas linhas no mesmo eixo, primeiro precisamos de mais dados. Vamos gerar então dados para a função:

f(x)=x3parax∈[1,5]

Para os próximos exemplos, iremos usar os dados:

x = [1, 2, 3, 4, 5]

y1 = [n \*\* 2 for n in x]

y2 = [n \*\* 3 for n in x]

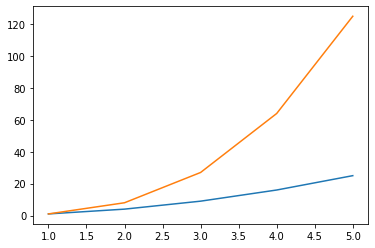
#### No mesmo eixo

plt.plot(x, y1)

plt.plot(x, y2)

plt.show()

Isso irá gerar a figura:



O matplotlib automaticamente detecta os melhor valor para o eixo-y e coloca cores diferentes.

#### Múltiplos eixos

As coisas começam a ficar mais interessantes aqui.

Iremos utilizar uma função nova [plt.subplots](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html) aqui.

Ela é uma função que irá gerar uma figura e um conjunto de eixos pré-posicionados num formato de grade.

| linha/coluna | 0 | 1 | ... | N |
| --- | --- | --- | --- | --- |
| 0 | grafico[0][0] | grafico[0][1] | ... | grafico[0][N] |
| 1 | grafico[1][0] | grafico[1][1] | ... | grafico[1][N] |
| .. | ... | ... | ... | ... |
| N | grafico[N][N] | grafico[N][1] | ... | grafico[N][N] |

Para isso, ao invorcarmos essa função, iremos passar dois argumentos:

* nrows: indica quantas linhas.
* cols: indinca quantas colunas.

Como queremos 2 gráficos, podemos passar 1 linha e 2 colunas ou 2 colunas e 1 linha.

Ele irá retornar uma tupla de dois elementos:

1. um objeto do tipo Figure que representa a figura
2. lista ou matriz de eixos

Ao passar nrows=1, espertamente, o matplotlib não devolve uma matriz e sim uma lista.

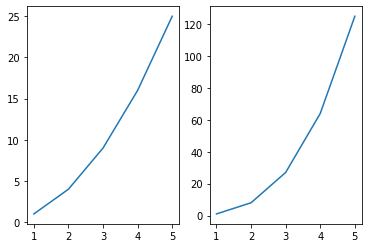
Podemos acessar cada eixo usando a notação de colchetes, e invocar a função que já vimos anteriormente: plot.

fig, axes = plt.subplots(nrows=1, ncols=2)

axes[0].plot(x, y1)

axes[1].plot(x, y2)

plt.show()



## Estilizando

O gráficos que fizemos até então, não definimos nenhuma característica de estilo como cor, grossura da linha, se deve ter um marcador para cada ponto, entre diversas outras características possíveis de serem definidas.

Três propriedades super importantes são:

* os rótulos, ou labels dos eixos
* título do gráfico
* legendas

Elas ajudam as pessoas que analisam o gráfico a entender melhor do que se trata.

Temos uma grande lista de características que podemos passar para a função [plot](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html) que muda as características do gráfico, algumas delas são

* color: para mudar a cor
  + interessante mencionar que podemos passar a representação hexadecimais de uma cor aqui
* linewidth: a grossura da linha
* linestyle: o tipo da linha (pontilhada, tracejada, sólida, etc)
* marker: se queremos um marcador para cada ponto.
* e muito mais!

Recomendo visitar a documentação oficial para maiores detalhes das diversas propriedades.

Vamos adicioná-las então!

plt.plot(x, y1, color='red', linestyle='--', linewidth=5, label='f(x) = x^2')

plt.plot(x, y2, color='#27a16c', marker='o', label='f(x) = x^3')

plt.xlabel('x')

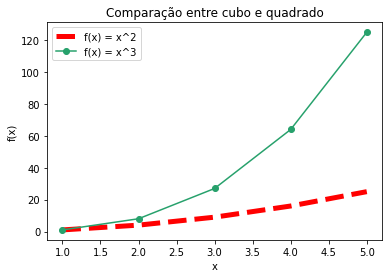
plt.ylabel('f(x)')

plt.title('Comparação entre cubo e quadrado')

plt.legend()

plt.show()

Resultado:



Vamos fazer um passo a passo por linha:

Nas duas primeiras linhas, passamos um novo argumento, label, definindo um nome para aquela linha que será impressa no gráfico. Ela será útil para a legenda posteriormente.

Também passamos alguns argumentos mencionados, como color, linestyle, linewidth e marker.

Na linha 3 e 4 chamamos as funções [xlabel](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html) e [ylabel](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html) para definir um texto para cada um dos eixos.

Na linha 5 chamamos a função [title](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.title.html) para definir o título da figura.

Por último, chamamos a fução [legend](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.legend.html) para dizermos ao matplotlib que queremos que apareça a legenda. Por padrão, ele tenta achar uma boa posição no gráfico automaticamente, mas podemos controlar aonde queremos que apareça a legenda.

## As duas APIs do matplotlib

Até então, não foi mencionamos aqui que temos duas formas de utilizar a biblioteca matplotlib, também chamadas de APIs.

Vamos apresentá-las então:

* API funcional
* API orientado a objetos

Mostramos as duas anteriormente, mas não definimos qual era qual.

A diferença é muito sutil, e se dá basicamente no fato de você estar ou não lidando diretamente com os eixos do gráfico.

No nosso exemplo de Múltiplos eixos, fizemos uso da API orientada a objetos, enquanto nos outros exemplos, utilizamos a API funcional. Nota-se pelo fato estarmos lidando com a variável axes e não chamando plt diretamente para o plot.

O mesmo gráfico do exemplo anterior, utilizando a API orientada a objetos, se ficaria dessa forma:

figure = plt.figure()

ax = figure.add\_axes([0.1, 0.1, 0.8, 0.8])

ax.plot(x, y1, c='red', ls='--', lw=5, label='f(x) = x^2')

ax.plot(x, y2, c='#27a16c', marker='o', label='f(x) = x^3')

ax.set\_xlabel('x')

ax.set\_ylabel('f(x)')

ax.set\_title('Comparação entre cubo e quadrado')

ax.legend()

plt.show()

Na linha 1 criamos uma Figure e a partir dela, adicionamos um eixo, chamando o add\_axes e passando uma lista com 4 floats definindo um retângulo ocupado por esse eixo.

Os 4 floats são definidos por [left, bottom, width, height], definidas como porcentagem da área da figura.

Obs: também utilizamos nesse exemplo as abreviações de linewidth = lw, linestyle = ls e color = c.

Pode parecer complicado a princípio, e você deve estar se perguntando: por que eu usaria essa API ao invés do funcional?

Acontece que ela é mais poderosa em termos de flexibilidade, nos permitindo fazer algo como:

figure = plt.figure()

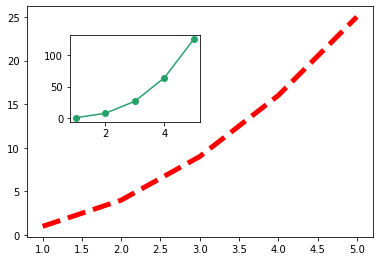
ax1 = figure.add\_axes([0.1, 0.1, 0.8, 0.8])

ax2 = figure.add\_axes([0.2, 0.5, 0.3, 0.3])

ax1.plot(x, y1, c='red', ls='--', lw=5)

ax2.plot(x, y2, c='#27a16c', marker='o')

plt.show()



A função que vimos anteriormente, plt.subplots, abstrai essa complexidade para nós criando a grade de eixos para nós automaticamente. Caso contrário, teríamos que fazer manualmente algo como:

figure = plt.figure()

ax1 = figure.add\_axes([0.1, 0.1, 0.4, 0.4])

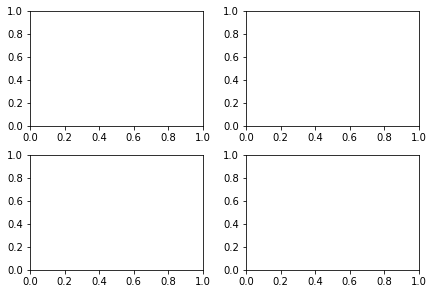
ax2 = figure.add\_axes([0.6, 0.1, 0.4, 0.4])

ax3 = figure.add\_axes([0.1, 0.6, 0.4, 0.4])

ax3 = figure.add\_axes([0.6, 0.6, 0.4, 0.4])

plt.show()

Para conseguir a grade:



### Qual eu devo usar?

Não existe a certa ou errada. O importante aqui é saber da existência das duas, e identificar qual você está usando. Use a que for necessária para alcançar seu objetivo.

No geral, muitos artigos recomendam o uso da API orientada a objeto por ser mais declarativa, fácil de ler e mais poderosa para algumas personalizações avançadas.

### Tamanho da figura

Caso você ache pequeno o tamanho da figura para a quantidade de eixos, você sempre pode passar o parâmetro figsize.

width, heigth = (10, 8) # valores em inches

figure = plt.figure(figsize=(width, height))

# ou

fig, ax = plt.subplots(figsize=(width, height))

## Outros tipos de gráficos

Até então, só vimos gráficos de linha, mas temos uma vasta gama de outos tipos de gráficos:

* dispersão ou scatter
* barra
* pizza
* histogramas
* diagrama de caixa ou boxplot

E muitos outros! Mostrarei alguns deles aqui, mas, recomendo fortemente visitar a [galeria de exemplos da biblioteca](https://matplotlib.org/gallery/index.html) para uma imersão das possibilidades.

### [Barra](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html)

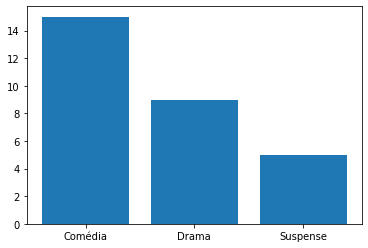
Os gráficos de barra normalmente são utilizados quando temos dados numéricos relacionados a dados categóricos.

Exemplo:

tipos\_filmes = ['Comédia', 'Drama', 'Suspense']

quantidade = [15, 9, 5]

plt.bar(tipos\_filmes, quantidade)



### [Dispersão](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html)

import numpy as np

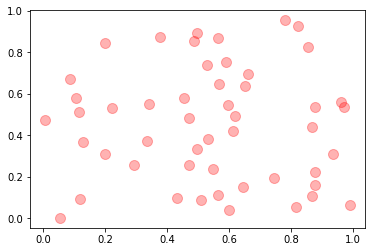
x = np.random.rand(1, 50)

y = np.random.rand(1, 50)

plt.scatter(x, y, color='red', s=100, alpha=0.3)

plt.show()

Utilizamos o parâmetro s, uma abreviação de size para aumentar o tamanho da "bolinha" do gráfico, e alpha para dar um efeito de transparência, resultando no seguinte gráfico:



### [Histograma](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html)

Os histogramas são utilizados para visualização da distribuição numérica de dados.

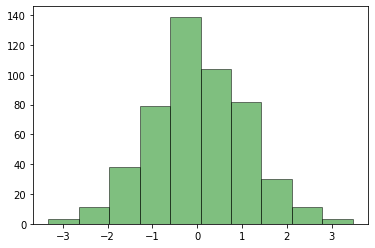
Novamente, vamos utilizar o NumPy para gerar valores aleatórios da distribuição normal.

Para exibir esses respectivos valores como histograma utilizamos plt.hist()

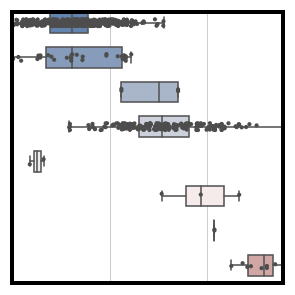
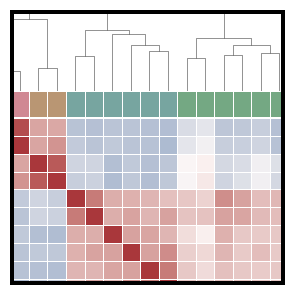
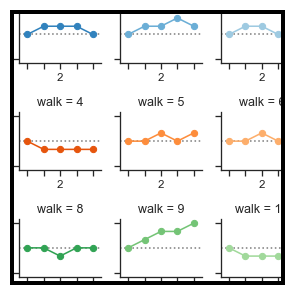
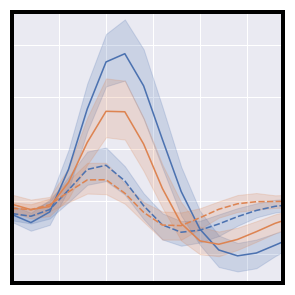
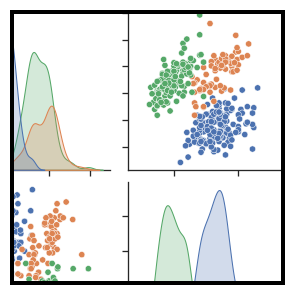
data = np.random.randn(500)

plt.hist(data, alpha=0.5, color='green', edgecolor='black')

plt.show()



# Seaborn



[Seaborn](https://seaborn.pydata.org/) é uma biblioteca de visualização de dados Python baseada no matplotlib. Ela fornece uma interface de alto nível para desenhar gráficos estatísticos atraentes e informativos.

## Antes de tudo

### Instalação

Se você tem o Anaconda instalado, você provavelmente já possui o Seaborn instalado.

Para checar, rode o seguinte comando no Jupyter Notebook ou interpretador do python:

import seaborn

Se não der erro, é porque está tudo ok!

Caso você tenha um erro, será necessário instalar a biblioteca. Para isso, rode o seguinte comando no prompt/terminal de comando:

pip install seaborn

Em caso de problemas nessa parte, entre em contato no chat da sua turma ou no Q&A.

### Importando a biblioteca

Assim como importamos pandas como pd e numpy como np, existe uma convenção de se importar seaborn como sns, como mostrado abaixo:

import seaborn as sns

%matplotlib inline

## API

Sendo uma biblioteca construída como uma casa envolta do matplotlib, tudo que aprendemos sobre matplotlib se aplica aqui em termos de conceitos (Figure, Axes, labels dos eixos, etc). Porém, o seaborn abstrai para o usuário da biblioteca algumas complexidades, tendo uma excelente integração com DataFrames do pandas.

Como mostra a [documentação da sua API](https://seaborn.pydata.org/api.html), ela é separada em 9 grupos, como mostra a lista a seguir:

* Gráficos relacionais
  + gráficos básicos, exemplos: dispersão e linha
* Categóricos
  + exemplo: gráfico de barras, diagrama de caixa
* Distribuição
  + exemplo: gráfico de distribuição
* Regressão:
  + exemplo regressão linear
* Matriz
  + exemplo: mapa de calor
* Multi-gráfico
  + exemplo: pairplot, jointplot, para múltiplos gráficos em 1 passo
* Controle de estilo
  + exemplos: set, set\_style para configurar estilos
* Paleta de cores
  + exemplos: dark\_pallete, light\_pallete para configurar as cores utilizadas
* Utilitários
  + load\_dataset: carregar dados para exemplos

Veremos agora as funções mais comumente utilizadas do Seaborn.

## Análise dos dados

Antes de começar, vamos dar uma olhada nos dados que iremos utilizar em boa parte dos exemplos: um dataset de gorjetas ou tips.

Iremos utilizar a função utilitária que vem com o seaborn, a load\_dataset, para carregar os dados, que já é um DataFrame:

tips = sns.load\_dataset('tips')

tips.head()

|  | total\_bill | tip | sex | smoker | day | time | size |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| 1 | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| 2 | 21.01 | 3.5 | Male | No | Sun | Dinner | 3 |
| 3 | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| 4 | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |

* total\_bill: total da conta do restaurante
* tip: gorjeta
* sex: gênero da pessoa que pagou a conta
* smoker: tinha fumantes no grupo?
* day: dia da semana
* time: almoço ou janta?
* size: tamanho do grupo

Iremos aproveitar o momento e chamar a seguinte função para configurar estilos padrões do seaborn:

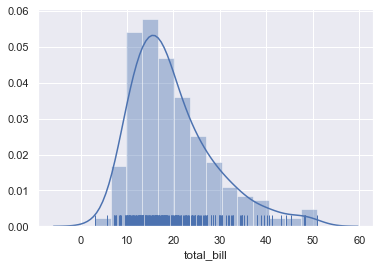
sns.set()

Esse passo é opcional. Caso não efetuado, os gráficos possuirão os estilos padrões do matplotlib. Mais sobre outros estilos na seção Estilos e Cores.

## Distribuição

Essa função combina a função plt.hist, que vimos no artigo de matplotlib (com cálculo automático de uma boa quantidade de bins) com as funções do seaborn kdeplot() e rugplot().

sns.distplot(tips['total\_bill'], rug=True, kde=True)



### Entendendo o gráfico

Para entedermos melhor esse gráfico, precisamos entender um pouquinho o que é KDE, abreviação de Kernel Density Estimation, que traduzido seria Estimativa de densidade kernel e o rug plot (gráfico de tapete).

Do Wikipedia:

Rug Plot

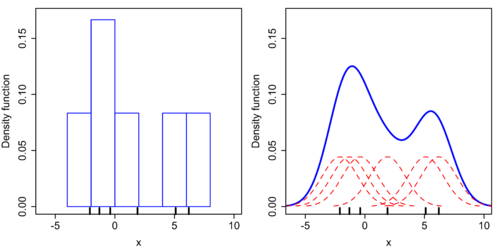
Um gráfico de tapete é um gráfico de dados para uma única variável quantitativa, exibida como marcas ao longo de um eixo. É usado para visualizar a distribuição dos dados. Como tal, é análogo a um histograma com compartimentos de largura zero ou um gráfico de dispersão unidimensional.

KDE:

Em Estatística, estimativa de densidade por Kernel (EDK) é uma forma não-paramétrica para estimar a Função densidade de probabilidade (FDP) de uma variável aleatória. Estimativa da densidade por Kernel é um problema fundamental de suavização de dados onde inferências sobre a população são feitas com base em uma amostra de dados finita.

De forma bem simplifcada então, juntando os dois conceitos, podemos pensar que o gráfico de linha KDE é basicamente a soma das várias gaussianas envolta de cada ponto individual (representado no rug plot pelo traço vertical). Quando mais gaussianas se sobrepõe, maior a soma e portanto maior o pico da função KDE.

A imagem abaixo ilustra bem:



### Customizando

Podemos customizar de várias formas nosso gráfico, por exemplo:

sns.distplot(

tips['total\_bill'],

hist=False, # não mostrar o histograma

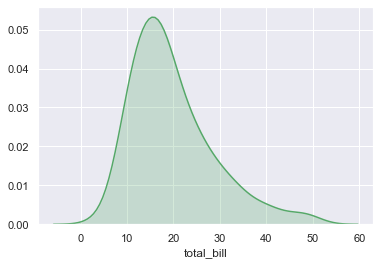
color="g", # cor verde

# para customizar mais profundamente,

# podemos passar parâmetros do matplotlib dessa forma:

kde\_kws={"shade": True} # preencher de verde

)



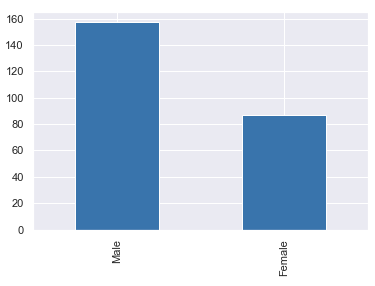
## Categóricos

Quanto possuímos uma variáveis categórias, como por exemplo: genêro (masculino, feminino), fumante (sim ou não), queremos ver como uma variável numérica se comporta para seus diferentes valores, ou mesmo uma mera contagem.

Apenas com a variável categória, podemos ver sua contagem como um gráfico de barras:

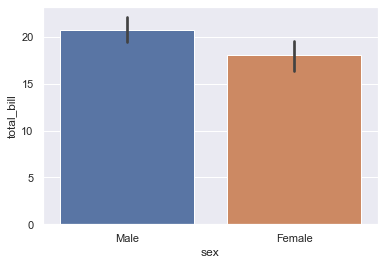
# contagem para a columa "sex"

sns.countplot(x='sex', data=tips)



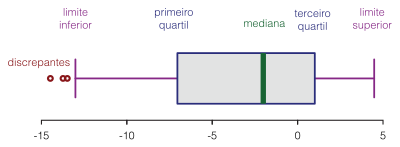
Já em relação a variável contínua total\_bill, podemos visualizar seu comportamento para os diferentes gêneros também na forma de um gráfico de barras:

sns.barplot(x='sex', y='total\_bill', data=tips)



O risco preto no centro da barra é chamado de [intervalo de confiança](https://pt.wikipedia.org/wiki/Intervalo_de_confiança), por padrão no valor de 95%.

Outro tipo de gráfico amplamente utilizado é o boxplot ou diagrama de caixa. Ele é um pouco mais complicado de entender a primeira vista, requerindo um conhecimento prévio para 100% de entendimento:



Sendo o limite superior definido por:

min(max(f(x)),Q3+1,5⋅IQR)

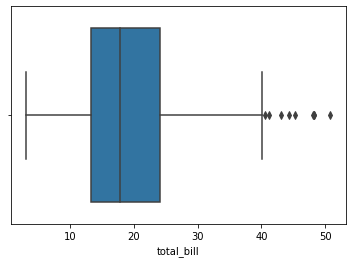
E o inferior:

max(min(f(x)),Q1−1,5⋅IQR)

Sendo $f(x)$ os valores da variável contínua observada e $IQR$ a diferença entre o terceiro quantil e o primeiro quantil ($IQR = Q3 - Q1$).

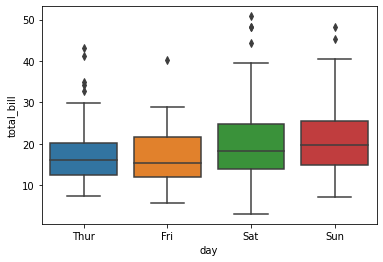
Podemos observar apenas uma variável contínua diretamente, como no exemplo acima:

sns.boxplot(x='total\_bill', data=tips)



Ou, mais comumente, com relação a uma variável categória, nesse caso, o dia da semana day:

sns.boxplot(x='day', y='total\_bill', data=tips)



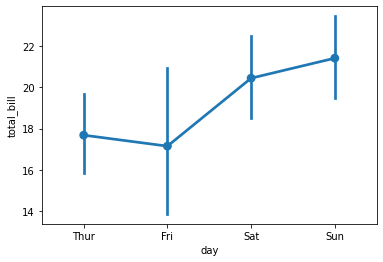
Existem diversos outros gráficos mais avançados disponíveis na biblioteca seaborn, que requerem cuidado ao ser utilizados para serem bem entendidos pelo público, como por exemplo:

* violin plots
* strip plots
* swarm plots
* point plot

Todos eles acabam sendo variações, com suas vantagens e desvantagens, do que é possível fazer com o boxplot.

Exemplo de point plot

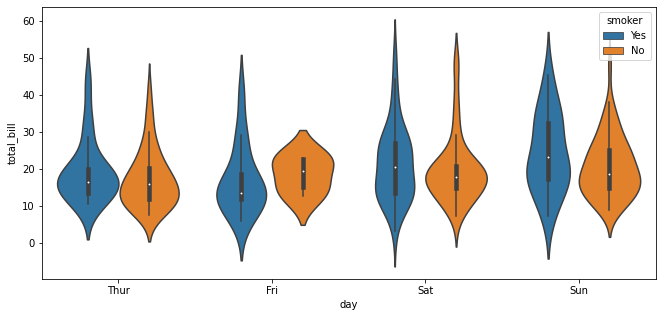
sns.pointplot(x='day', y='total\_bill', data=tips)



Exemplo de violin plot com uso do parâmetro hue para destacar uma segunda variável categórica no gráfico além de day: smoker.

plt.figure(figsize=(11, 5))

sns.violinplot(x='day', y='total\_bill', hue='smoker', data=tips)



Aqui na verdade tem um aspecto muito interessante da biblioteca seaborn: a sua API facilita passar poucos parâmetros como x, y, hue e data para fazer um gráfico bem elaborado que demandaria possivelmente várias linhas de código usando matplotlib diretamente.

Além de ser uma API bem consistente: ela aceita os mesmos parâmetros para diferentes tipos de gráficos.

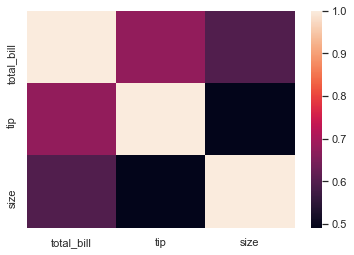
Já matplotlib, por exemplo, temos o comando plot para gráfico de linha que aceita os parâmetros x e y, já o comando bar aceita x e height e width.

## Matriz

O principal gráfico em termos de matriz é o heatmap ou mapa de calor.

Podemos utilizar a função corr() que calcula a correlação linear (por padrão, de [Pearson](https://pt.wikipedia.org/wiki/Coeficiente_de_correlação_de_Pearson)) entre as variáveis contínuas do nosso DataFrame e retorna uma matriz numérica com os valores dessa correlação.

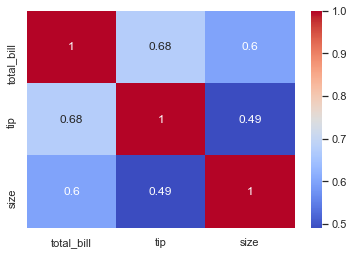
sns.heatmap(tips.corr())



Onde 1 quer dizer correlação perfeita, e zero baixa nenhuma correlação linear, e -1 correlação negativa perfeita (quando uma aumenta, outra diminui).

Para facilitar a visualização, podemos mudar o esquema de cores para o coolwarm ou quente/frio e mostrar os valores:

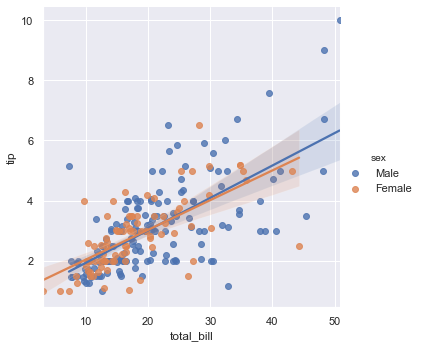
sns.heatmap(tips.corr(), cmap='coolwarm', annot=True)



## Regressão

Primariamente para visualizar relacionamentos lineares via [regressão linear](https://pt.wikipedia.org/wiki/Regressão_linear) ou mesmo logística.

sns.lmplot(x='total\_bill', y='tip', hue='sex', data=tips)



Note de novo aqui a consistência da API do seaborn, na qual com os mesmos parâmetros x, y, hue e data fizemos outro gráfico.

A documentação tem um tutorial bem completo sobre o tema:

* [Visualizing linear relationships](https://seaborn.pydata.org/tutorial/regression.html#regression-tutorial)

## Estilos e Cores

As duas funções mais valiosas para estilo com seaborn:

* [set\_style](https://seaborn.pydata.org/generated/seaborn.set_style.html#seaborn.set_style): darkgrid, whitegrid, dark, white, ticks
* [set\_context](https://seaborn.pydata.org/generated/seaborn.set_context.html): paper, notebook, talk ou poster

A primeira é majoritariamente para mudar as cores, enquanto a segunda tem uma característica importante: facilitar a visualização para diferentes contextos.

Exemplo, para um poster, geralmente deseja-se letras maiores:

sns.set\_context('poster')

sns.countplot(x='sex', data=tips)

