

Programação Python

Aula 07: NumPy

Prof. Eduardo Corrêa Gonçalves

26/03/2021

Sumário

Introdução

O que é NumPy?

O que é ndarray?

Criação de Vetores e Matrizes

Propriedades e Operações

Propriedades

Operações Básicas

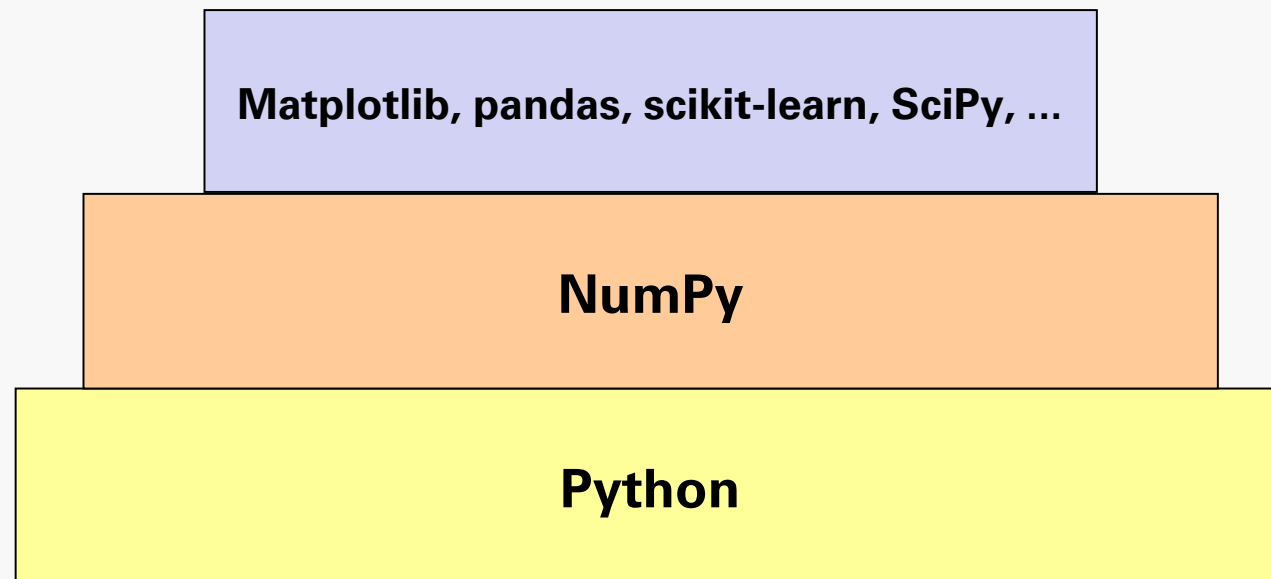
Operações Matemáticas e Estatísticas

Números Aleatórios

Indexação Booleana

Introdução (1/4)

- O que é NumPy (*Numerical Python*)?
 - Biblioteca que estende a linguagem Python com a ED **ndarray** (*n-dimensional array*).
 - Também chamada de “**Array NumPy**”.
 - A ‘NumPy’ é considerada a “pedra fundamental” da computação científica em Python.
 - Ela serviu de **base** para o desenvolvimento de diversas outras bibliotecas importantes para ciência de dados.



Introdução (2/4)

- **ndarray**
 - É um container que armazena uma coleção de valores do **mesmo tipo**.
 - Cada valor é armazenado em uma posição ou **célula** específica.
 - Os ndarrays podem ter diferentes **dimensões**.
- Na figura ao lado:
 - Um array unidimensional (**vetor**) com 5 células.
 - Um array bidimensional (**matriz**) com 12 células – 3 linhas e 4 colunas.
- Também podemos criar arrays com 3 ou mais dimensões.

[0]	[1]	[2]	[3]	[4]
9.1	7.2	5.5	10.0	6.9

array unidimensional

	[0]	[1]	[2]	[3]
[0]	1	0	1	0
[1]	0	1	0	1
[2]	0	0	0	1

array bidimensional

3D array



Introdução (3/4)

- ndarray

[0]	[1]	[2]	[3]	[4]	
9.1	7.2	5.5	10.0	6.9	<i>vet_notas</i>

- Características:
 - Permite elementos **duplicados**, porém todos devem ser do **mesmo tipo**.
 - **Mutável** – pode ser alterado.
 - **Iterável** – capaz de retornar seus elementos um por vez em um laço.
 - **Sequência** – elementos possuem ordem determinada.
 - **Eficiente** – armazenamento em memória otimizado e provê processamento com o uso de computação vetorizada.
 - O primeiro índice de um **eixo** é 0 e o último $n-1$.

Introdução (4/4)

- **Importação da Biblioteca NumPy**

- A ‘NumPy’ **não** faz parte do Python padrão.
 - No entanto, está inclusa no Google Colab e em todas distribuições do Python voltadas para computação científica (ex: WinPython e Anaconda).
- Para utilizá-la, você precisa usar o comando **import** (além de importar a biblioteca, a renomeamos para “np”):

import numpy as np.

Criação de Arrays NumPy (1/5)

• Criando Vetores (1/3)

- Os dois principais métodos para criar vetores são:
 - array()**: cria um array a partir de uma lista
 - arange()**: versão NumPy da função range() do Python.

#importa a biblioteca – vou omitir dos exemplos subsequentes!!!
import numpy **as** np

lst_notas = [9.1, 7.2, 5.5, 10, 6.9]
vet_notas = np.**array**(lst_notas) *#cria o vetor a partir da lista acima*

a1 = np.**arange**(11) *#seq. de 0 a 10*
a2 = np.**arange**(0, 16, 5) *#seq. de 0 a 15, com 5 como incremento*
a3 = np.**arange**(5, 0, -1) *#seq. de 5 a 1*

Criação de Arrays NumPy (2/5)

- Criando Vetores (2/3)

- Mas há outros métodos...

#append(): permite combinar vetores

```
n1 = np.array([1,2,3]); n2 = np.array([4,5])
```

```
n3 = np.append(n1, n2)
```

[1 2 3 4 5]

#repeat(): gera sequência de valores repetidos

```
rep1 = np.repeat(100,5)
```

[100 100 100 100 100]

#ones() e zeros(): geram uma sequência de

#valores 1 e 0, respectivamente

```
o1 = np.ones(5)
```

#[1. 1. 1. 1. 1.]

```
z1 = np.zeros(5)
```

#[0. 0. 0. 0. 0.]

Criação de Arrays NumPy (3/5)

- Criando Vetores (3/3)

*#linspace(): gera elementos "uniformemente espaçados" entre o
início e o fim da sequência (neste caso, o valor final é incluído)*

ls1 = np.linspace(1, 3, 9) *#seq. de 1 a 3, com 9 elementos:
[1. 1.25 1.5 1.75 2. 2.25 2.5 2.75 3.]*

ls2 = np.linspace(0, 2.0/3, 4) *#seq. de 0 a 2/3, com 4 elementos:
[0., 0.22222222, 0.44444444, 0.66666667]*

Criação de Arrays Numpy (4/5)

- Criando Matrizes (1/3)

- Os dois métodos mais populares são:
 - **array()**: cria uma matriz a partir de uma lista 2D
 - **reshape()**: converte um vetor (array 1D) para uma matriz (array 2D).

#importa a biblioteca – vou omitir dos exemplos subsequentes

import numpy as np

```
m1 = np.array([[7,8,9], [10,11,12]])
```

#	7	8	9
#	10	11	12

```
m2 = np.arange(8)
```

#cria vetor [0, 1, 2, 3, 4, 5, 6, 7]

```
m2 = m2.reshape(4,2)
```

#depois transforma em uma matriz 4 x 2:

#	0	1
#	2	3
#	4	5
#	6	7

Criação de Arrays NumPy (5/5)

• Criando Matrizes (2/2)

- Outros métodos:
 - **zeros()** e **ones()**: também servem para criar matrizes.
 - **identity(n)**: cria a matriz identidade de ordem n .

```

mz = np.zeros((4,4))    #matriz 4x4 de zeros
mu = np.ones((3,2))     #matriz 3x2 de 1's
mi = np.identity(3)      #matriz identidade 3x3
print(mz); print(mu); print(mi)

```

```

> [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]

[[1. 1.]
 [1. 1.]
 [1. 1.]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

```

Propriedades (1/3)

- **Propriedades básicas:**
 - **dtype**: tipo de dado dos elementos contidos no array. Ex: int16, float64, ...
 - **ndim**: número de dimensões: 1 p/ vetor, 2 para matriz, 3 se array é 3d, etc.
 - **size**: número de elementos
 - **shape**: formato do ndarray
 - **Ex.:** o valor é retornado em uma tupla. Para vetores, retorna o número de posições; para matrizes o número de linhas e colunas.
 - **axes**: números dos eixos do array. Vetores possuem apenas a axis (eixo) 0 e matrizes as axis 0 (linha) e 1 (coluna).
 - **itemsize**: memória ocupada por um elemento do vetor, em bytes
 - **nbytes**: soma da memória ocupada por todos os elementos do vetor, em bytes
 - **flags**: layout de memória do vetor

Propriedades (2/3)

- Recuperando as propriedades
 - Exemplo com array 2D, definido explicitamente com o dtype int16.

```
m1 = np.zeros((3,2), dtype=np.int16)
```

```
print(m1)          # [[0 0]
                   # [0 0]
                   # [0 0]]
```

```
print(type(m1))    # <class 'numpy.ndarray'>
```

```
print(m1.dtype)    # int16
```

```
print(m1.ndim)     # 2      - número de dimensões
```

```
print(m1.shape)    # (3, 2) - tupla c/ número de linhas e colunas
```

```
print(m1.size)     # 6      - número de células
```

```
print(m1.itemsize) # 2      - total de bytes ocupado por um item
```

```
print(m1.nbytes)   # 12     - total de bytes ocupado por todos os itens
```

```
print(m1.data)     # <memory at 0x00000243CA2D0208>
```

```
print(m1.strides)  # (4, 2) - com 4 bytes avanço para a próxima linha
                  #      com 2 bytes para a próxima coluna
```

Propriedades (3/3)

- **dtypes** - a NumPy trabalha com um rico conjunto de tipos de dados.

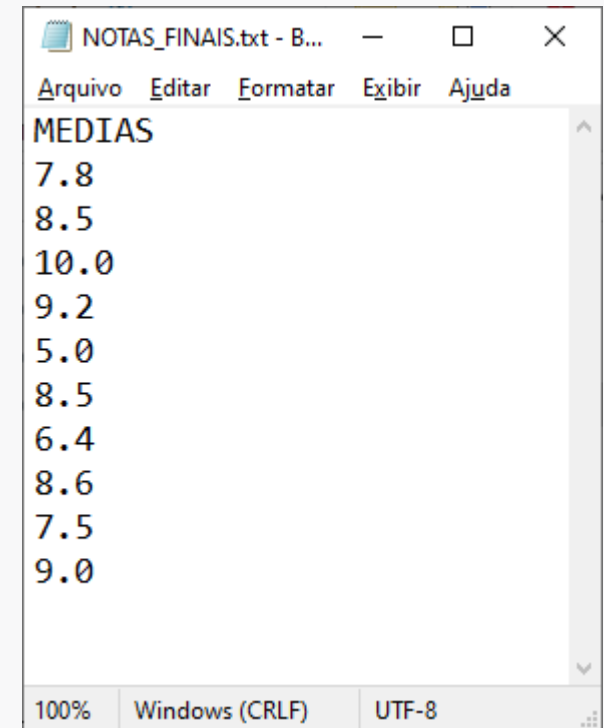
- Quadro 27 - dtypes da NumPy**
- **bool_**: valores booleanos;
 - **str**: valores string (alfanuméricos);
 - **int8**: inteiros com representação em 8 bits. Faixa de valores: -128 a 127;
 - **int16**: inteiros com representação em 16 bits. Faixa de valores: -32.768 a 32.767;
 - **int32**: inteiros com representação em 32 bits. Faixa de valores: -2.147.483.648 a 2.147.483.647;
 - **int64**: inteiros com representação em 64 bits. Faixa de valores: -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807;
 - **int_**: apelido para o tipo inteiro *default*, normalmente "int64";
 - **uint8**: inteiros sem sinal com representação em 8 bits. Faixa de valores: 0 a 255;
 - **uint16**: inteiros sem sinal com representação em 16 bits. Faixa de valores: 0 a 65.535;
 - **uint32**: inteiros sem sinal com representação em 32 bits. Faixa de valores: 0 a 4.294.967.295;
 - **uint64**: inteiros sem sinal com representação em 64 bits. Faixa de valores: 0 a 18.446.744.073.709.551.615;
 - **float16**: valores reais positivos e negativos, representação utilizando 5 bits para o expoente e 10 para a mantissa (*half-precision*). Faixa de valores aproximada: -2^{16} a 2^{16} ;
 - **float32**: valores reais positivos e negativos, representação utilizando 8 bits para o expoente e 23 para a mantissa (*single-precision*). Faixa de valores aproximada: 2^{-128} a 2^{128} ;
 - **float64**: valores reais positivos e negativos, representação utilizando 11 bits para o expoente e 52 para a mantissa (*double-precision*). Faixa de valores aproximada: 2^{-1022} a 2^{1023} ;
 - **float_**: apelido para "float64";
 - **complex64**: Número complexo representado por dois floats de 32 bits;
 - **complex128**: Número complexo representado por dois floats de 64 bits;
 - **complex_**: apelido para "complex128".

Carga a partir de Arquivos (1/2)

- Carregando Vetores

- Com **genfromtxt()**, podemos gerar um vetor a partir de um arquivo texto.
- Neste exemplo, o parâmetro **skip_header=1** é usado para indicar que desejamos pular a primeira linha.
 - Ele é necessário sempre que o arquivo a ser importado possuir um cabeçalho.

notas_finais.txt



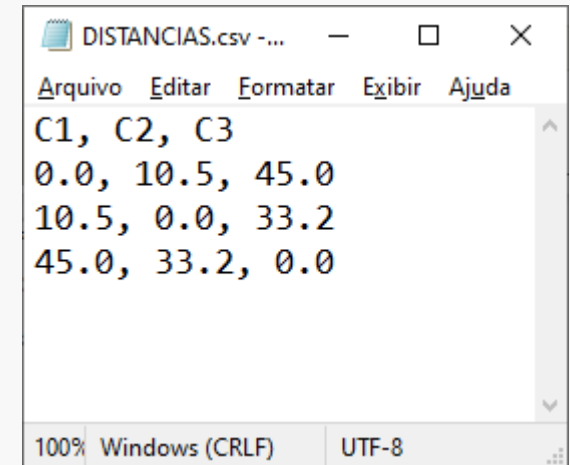
```
vet_notas = np.genfromtxt('NOTAS_FINALS.txt', skip_header=1)
print(vet_notas)
>
> [ 7.8  8.5 10.  9.2  5.  8.5  6.4  8.6  7.5  9.]
```

Carga a Partir de Arquivos (2/2)

- Carregando Matrizes

- Com **genfromtxt()** também podemos gerar uma matriz a partir de um arquivo texto.
- Parâmetros
 - skip_header=1** para pular a primeira linha.
 - delimiter="delim."** para especificar o delimitador.

distancias.csv



C1	C2	C3
0.0	10.5	45.0
10.5	0.0	33.2
45.0	33.2	0.0

```
m_dist = np.genfromtxt('DISTANCIAS.csv', skip_header=1, delimiter=',')  
print(m_dist)
```

```
>
```

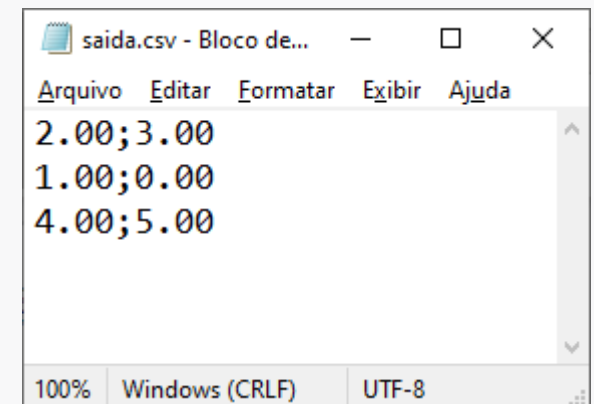
```
[[ 0. 10.5 45. ]  
 [10.5 0. 33.2]  
 [45. 33.2 0. ]]
```


Gravando um Arquivo

- Exportando um ndarray para arquivo texto
 - Com a função **savetxt()** podemos exportar os dados de um ndarray para um arquivo CSV.
 - **delimiter** é o parâmetro para especificar o delimitador.
fmt: para especificar o formato.
 - Se não for usado, os dados serão gravados com notação científica.
 - “%d”: para gravar como inteiro
 - “%.2f”: para gravar com duas casas decimais

```
a = np.array([2,3,1,0,4,5]).reshape((3,2))
```

```
np.savetxt("saida.csv", a, delimiter = ";", fmt="%.2f ")
```



Operações

- **Operações básicas*:**
 - Recuperar elemento pelo índice
 - Modificar elemento pelo índice
 - Verificar se elemento x pertence ao array
 - Iterar (percorrer todos os elementos *em ordem*)
 - Fatiar (obter um subvetor)
 - Adicionar um elemento x (em qualquer posição)
 - Remover um elemento (pela posição)
 - Contar número de ocorrências do elemento x
 - Obter o(s) índice(s) do elemento x
 - Inverter
 - Ordenar
- **Obs.:** *Essa são apenas as operações análogas às das listas. Métodos que implementam operações matemáticas/estatísticas serão mostrados posteriormente.*

Operações - Vetores (1/6)

- Recuperando elementos pelos índices

```
v = np.array([9.1, 7.2, 5.5, 10, 6.9])
```

```
primeiro = v[0]          # 9.1
```

```
ultimo = v[4]            # 6.9
```

```
ultimo_tambem = v[-1]    # também retorna 6.9
```

```
penultimo = v[-2]        # 10.0
```

- Modificando um elemento pelo índice

```
v[2] = 4.0                # [9.1, 7.2, 4.0, 10, 6.9]
```

```
v[4] = 7.2                # [9.1, 7.2, 4.0, 10, 7.2]
```

```
v[-1] = 7.5               # [9.1, 7.2, 4.0, 10, 7.5]
```

- Verificando se elemento pertence ao vetor

```
10.0 in v                 # True
```

```
-5 in v                   # False
```

Operações - Vetores (2/6)

- Iterando

```
v = np.array([9.1, 7.2, 5.5, 10, 6.9])
```

```
for nota in v:
```

```
    print(nota)
```

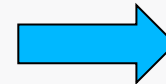


```
9.1  
7.2  
5.5  
10.0  
6.9
```

- Iterando com base nos índices

```
for k in range(v.size):
```

```
    print("elemento {} = {}".format(k, v[k]))
```



```
elemento 0 = 9.1  
elemento 1 = 7.2  
elemento 2 = 5.5  
elemento 3 = 10.0  
elemento 4 = 6.9
```

Operações - Vetores (3/6)

- **Fatiando – *sintaxe igual a das listas...***

```
v = np.array(["A", "B", "C", "D"])
```

```
v[2:4]          # ['C' 'D']
```

```
v[:3]           # ['A' 'B' 'C']
```

```
v[2:]           # ['C' 'D']
```

```
v[::2]          # ['A' 'C']
```

```
v[1::2]         # ['B' 'D']
```

Fatiando da direita para esquerda, com a sintaxe v[n:m:-k], onde n > m

```
v[3:1:-1]       # ['D' 'C']
```

```
v[::-1]         # ['D' 'C' 'B' 'A'] – obtém o array invertido!
```

- **Modificando vários elementos de uma vez (*modif. estilo fatiamento*):**

```
v[1:4] = ["X", "Y", "Z"]    # ['A' 'X' 'Y' 'Z']
```

Operações - Vetores (4/6)

- Inserindo e Removendo Elementos:

```
numeros = np.array([5, 10])
```

```
numeros = np.append(numeros, 20)
```

```
# insere 20 no final: [ 5 10 20]
```

```
numeros = np.append(numeros, [15, 5])
```

```
# insere 15 e 5 no final: [ 5 10 20 15 5]
```

```
numeros = np.insert(numeros, 2, 40)
```

```
# insere 40 na posição 2: [ 5 10 40 20 15 5]
```

```
numeros = np.delete(numeros, 4)
```

```
# remove o item da pos. 4: [ 5 10 40 20 5]
```

```
numeros = np.delete(numeros, np.s_[:2])
```

```
# remove os 2 primeiros itens: [40 20 5]
```

```
numeros = np.insert(numeros, 0, 100)
```

```
# insere 100 no início: [100 40 20 5]
```

```
numeros = np.delete(numeros, [1, 3])
```

```
# remove os itens nas posições 1 e 3:
```

```
# [100 20]
```

Operações - Vetores (5/6)

- **Contando o número de ocorrências de um elemento**

```
notas = np.array([95, 70, 75, 100, 70, 65, 70, 100])
```

```
np.count_nonzero(notas==70)          # 3 (= 3 ocorrências)
```

- **Obtendo os índices de um elemento**

```
np.where(notas==70)                  # (array([1, 4, 6], dtype=int64),)
```

- O valor retornado é uma tupla com 2 elementos
 - O primeiro um ndarray com as posições do elemento
 - O segundo seu dtype dos elementos

```
x = np.where(notas==70)              # (array([1, 4, 6], dtype=int64),)
```

```
print(type(x))                       # <class 'tuple'>
```

```
print(type(x[0]))                    # <class 'numpy.ndarray'>
```

Operações – Vetores (6/6)

- Ordenando e invertendo um vetor

```
notas = np.array([95, 70, 75, 100, 70, 65, 70, 100])
```

```
notas = np.sort(notas)           # [ 65  70  70  70  75  95 100 100]
```

```
notas = notas[::-1]             # [100 100  95  75  70  70  70  65]
```


Operações – Matrizes (1/3)

- Iterando

```
m = np.array ([[1,2,4], [3, 5, 10]])
```

```
for i in range(m.shape[0]):  
    for j in range(m.shape[1]):  
        print(m[i, j], end=" ")  
    print("")
```



1	2	4
3	5	10

- Utilizamos a propriedade **shape** para obter:
 - o número de linhas - shape[0]
 - e o número de colunas - shape[1].

Operações – Matrizes (2/3)

- Indexação e Fatiamento I (estilo lista)

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[1, 2]$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[:, :3]$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[-1, -2]$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[:, -2:]$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[2]$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[:, 1:3]$

- Obs.:** a **indexação** retorna um **novo objeto** do tipo escalar. Já o **fatiamento** retorna uma **visão** da matriz original.

Operações - Matrizes (3/3)

- Fatiamento II – estilo *range*

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[0:3:2, 0:4:2]$

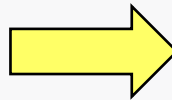
	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[:, 1::2]$

- Fatiamento III – *fancy indexing*

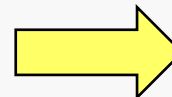
- Seleção de células passando uma lista ou um ndarray.

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[[2, 0]]$


	[0]	[1]	[2]	[3]
[0]	9	10	11	12
[1]	1	2	3	4

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

 $m[[2, 0], [1, 3]]$


	[0]	[1]
[0]	10	4

Operações Matemáticas e Estatísticas

- Operações entre arrays e escalares
- Operações entre arrays do mesmo tamanhoFunções universais (*ufuncs*)
- Funções estatísticas
- Álgebra Linear básica
- Geração de números aleatórios
- Gravação em arquivo texto

Operações Matemáticas e Estatísticas (1/12)

- **Operações Aritméticas**

- **Não** precisamos implementar laços
- **Não** precisamos usar list comprehension
- As operações aritméticas utilizam **computação vetorizada** (*vectorization*).
 - Se x é um vetor e fazemos $x * 2$, obtemos como resultado um vetor que conterá todos os elementos de x multiplicados por 2.
 - Ao efetuarmos a soma de dois vetores $v1$ e $v2$ de shape compatível (mesmo tamanho), teremos como resultado um novo vetor, onde cada célula de índice $[k]$ será igual a $v1[k] + v2[k]$.

Operações Matemáticas e Estatísticas (2/12)

- Operações aritméticas

#operações entre vetor e escalares

v1 = np.array([0,5,10])

print('v1*2 = ', v1 * 2) *# [0 10 20]*

print('v1-1 = ', v1 - 1) *# [-1 4 9]*

print('v13 = ', v1 ** 3)** *# [0 125 1000]*

#soma dois vetores com 3 elementos

x = np.array([0,5,10])

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

z = x + y

print('z = ', z) *# [1 7 13]*

Operações Matemáticas e Estatísticas (3/12)

- Operações aritméticas

Cálculo da série $S = (1/1) + (3/2) + (5/3) + (7/4) + \dots + (99/50)$

sem programar laço (computação vetorizada)

`numerador = np.arange(1,100, 2)` *# [1 3 5 ... 99]*

`denominador = np.arange(1, 51)` *# [1 2 3 ... 50]*

`S = sum(numerador / denominador)` *# [1/1 3/2 5/3 ... 99/50]*

`print('* * * resposta: s = {:.2f}'.format(S))` *# 95.50*

- Obs.:** as funções built-in `len()`, `min()`, `max()` e `sum()` podem ser normalmente aplicadas sobre ndarrays.

Operações Matemáticas e Estatísticas (4/12)

- Operações entre arrays e escalares

```
m = np.array([[1,2,4], [3, 5, 10]])
```

```
print(1 / m)
```

```
[[1.      0.5     0.25   ]
 [0.33333333 0.2     0.1   ]]
```

- Operações entre arrays de mesmo tamanho

```
print(m - m)
```

```
[[0 0 0]
 [0 0 0]]
```

```
print(m * m)
```

```
[[ 1  4 16]
 [ 9 25 100]]
```

- Note que $m * m$ não é a multiplicação matricial da matemática
- Na verdade, faz $m[0,0] * m[0,0]$; $m[0,1] * m[0,1]$; etc.

Operações Matemáticas e Estatísticas (5/12)

- **Funções universais – *ufunc* (1/2)**
 - São funções simples, executadas de forma vetorizada.
 - **Isto é:** são automaticamente aplicadas sobre cada elemento.

```
numeros = np.arange(1,10).reshape((3,3))
```

```
numeros
```

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

```
 [4 5 6]
```

```
 [7 8 9]]
```

```
np.sqrt(numeros)    # sqrt é a ufunc que calcula a raiz quadrada
```

```
[[1.      1.41421356 1.73205081]
```

```
 [2.      2.23606798 2.44948974]
```

```
 [2.64575131 2.82842712 3.      ]]
```

Operações Matemáticas e Estatísticas (6/12)

- **Funções universais – Exemplos:** *(McKinney, 2017)*

Function	Description
<code>abs, fabs</code>	Compute the absolute value element-wise for integer, floating point, or complex values. Use <code>fabs</code> as a faster alternative for non-complex-valued data
<code>sqrt</code>	Compute the square root of each element. Equivalent to <code>arr ** 0.5</code>
<code>square</code>	Compute the square of each element. Equivalent to <code>arr ** 2</code>
<code>exp</code>	Compute the exponent e^x of each element
<code>log, log10, log2, log1p</code>	Natural logarithm (base e), log base 10, log base 2, and $\log(1 + x)$, respectively
<code>sign</code>	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
<code>ceil</code>	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
<code>floor</code>	Compute the floor of each element, i.e. the largest integer less than or equal to each element
<code>rint</code>	Round elements to the nearest integer, preserving the dtype
<code>modf</code>	Return fractional and integral parts of array as separate array
<code>isnan</code>	Return boolean array indicating whether each value is NaN (Not a Number)
<code>isfinite, isinf</code>	Return boolean array indicating whether each element is finite (non- <code>inf</code> , non-NaN) or infinite, respectively
<code>cos, cosh, sin, sinh, tan, tanh</code>	Regular and hyperbolic trigonometric functions
<code>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</code>	Inverse trigonometric functions
<code>logical_not</code>	Compute truth value of not x element-wise. Equivalent to <code>-arr</code> .

Operações Matemáticas e Estatísticas (7/12)

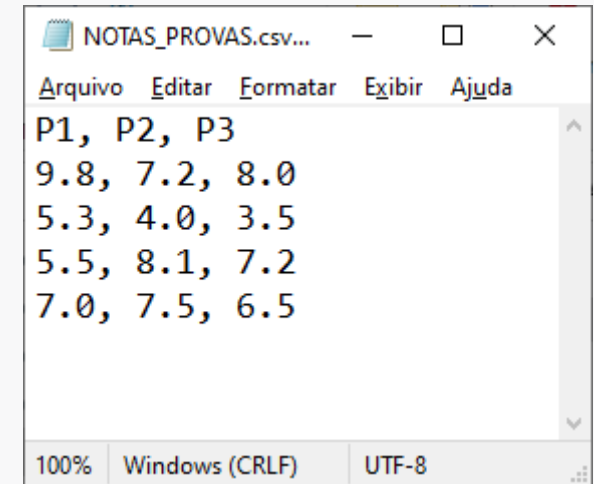
- **Funções Estatísticas (1/3)**

- Para arrays com mais de 1 dimensão, podem ser aplicadas a todos os elementos de um array ou a apenas um de seus eixos.
- **axis=None**
 - Calcula as estatísticas considerando todos os dados do array.
 - É o default, logo não precisa ser especificado.
- **axis=0**
 - Diz para calcular as estatísticas em todas as linhas, ou seja, para cada coluna da matriz.
- **axis=1**
 - Diz para calcular as estatísticas em todas as colunas, ou seja, para cada linha da matriz.

Operações Matemáticas e Estatísticas (8/12)

- Funções Estatísticas (1/2)

- Exemplo.: notas de 4 alunos em 3 provas



P1	P2	P3
9.8	7.2	8.0
5.3	4.0	3.5
5.5	8.1	7.2
7.0	7.5	6.5

```
notas = np.loadtxt('C:/CursoPython/NOTAS_PROVAS.csv', skiprows=1, delimiter=',')
```

```
print('maior nota geral: ', notas.max())
```

```
print('maior nota de cada prova: ', notas.max(axis=0))
```

```
print('maior nota de cada aluno: ', notas.max(axis=1))
```

```
>
```

```
maior nota geral: 9.8
```

```
maior nota de cada prova: [9.8 8.1 8.]
```

```
maior nota de cada aluno: [9.8 5.3 8.1 7.5]
```

Operações Matemáticas e Estatísticas (10/12)

- **Funções estatísticas – Exemplos:** (McKinney, 2017)

Table 4-5. Basic array statistical methods

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

Operações Matemáticas e Estatísticas (11/12)

- Álgebra Linear (1/3): Transposição (propriedade T)
- A propriedade T retorna a **matriz transposta** em uma **visão**
 - Ou seja: não há cópia os dados!!!

```
m = np.arange(15).reshape((3,5))
```

```
print(m)
```

```
[[ 0  1  2  3  4]
```

```
 [ 5  6  7  8  9]
```

```
[10 11 12 13 14]]
```

```
print(m.T)
```

```
[[ 0  5 10]
```

```
 [ 1  6 11]
```

```
 [ 2  7 12]
```

```
 [ 3  8 13]
```

```
 [ 4  9 14]]
```

Operações Matemáticas e Estatísticas (12/12)

- Álgebra Linear (2/3): Operações

- O cálculo de determinantes, obtenção da diagonal principal, obtenção da matriz inversa e outras operações úteis estão disponíveis.
- Ex.1:** `dot()` executa o produto de duas matrizes $m \times p$ e $p \times n$

```
a = np.array([2,3,1,0,4,5]).reshape((3,2))
```

```
[[2 3]
 [1 0]
 [4 5]]
```

```
b = np.array([1,2,3,4]).reshape((2,2))
```

```
[[1 2]
 [3 4]]
```

```
c = np.dot(a,b)
```

```
[[11 16]
 [ 1  2]
 [19 28]]
```

- Ex.2:** para calcular $X^T X$, basta fazer `np.dot(X.T, X)`.

Operações Matemáticas e Estatísticas (13/12)

- **Álgebra Linear (3/3) - Exemplos de Funções:** *(McKinney, 2017)*

Function	Description
<code>diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
<code>dot</code>	Matrix multiplication
<code>trace</code>	Compute the sum of the diagonal elements
<code>det</code>	Compute the matrix determinant
<code>eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>inv</code>	Compute the inverse of a square matrix
<code>pinv</code>	Compute the Moore-Penrose pseudo-inverse inverse of a square matrix
<code>qr</code>	Compute the QR decomposition
<code>svd</code>	Compute the singular value decomposition (SVD)
<code>solve</code>	Solve the linear system $Ax = b$ for x , where A is a square matrix
<code>lstsq</code>	Compute the least-squares solution to $y = Xb$

Número Aleatórios (1/3)

- Números Aleatórios

- O Módulo **numpy.random()** fornece rotinas para a geração de números aleatórios com base em diferentes tipos de distribuições.
- Ex.1:** `seed()`, `rand()`, `randint()`

```
np.random.seed(210720) #estabelece a semente
```

```
#gera vetor com 3 números aleatórios, intervalo (0..1], dist. uniforme
```

```
r1 = np.random.rand(3)
```

```
#gera um inteiro aleatório, intervalo (5, 10]
```

```
r2 = np.random.randint(5,10)
```

```
print('r1 = ', r1)           # [0.22079126 0.83385294 0.87334346]
```

```
print('r2 = ', r2)           # 7
```

Números Aleatórios (2/3)

- Números Aleatórios

- Ex.2: gerando uma base com dados “sintéticos” com **randint()**
- randint**(menor, maior, size=(linhas, colunas), dtype=tipo)
 - menor = limite inferior (menor número)
 - maior = limite superior (números podem ir até, mas sem incluir, esse valor)
 - size = dimensões da matriz

#o programa abaixo gera uma matriz 7x10 onde todos os valores são números aleatórios entre 1 e 5

np.random.seed(210720) *#estabelece a semente*

bd = np.random.randint(1, 6, size=(7,10), dtype = np.int8)

print(bd)

```
[[1 1 5 5 1 3 2 4 4 5]
 [4 3 3 2 4 3 1 3 3 1]
 [3 5 5 3 4 1 5 1 5 5]
 [2 1 1 1 5 4 4 3 2 5]
 [1 2 1 4 1 4 5 2 2 5]
 [4 1 2 1 2 4 2 4 1 4]
 [3 3 3 5 2 5 4 5 1 1]]
```

Números Aleatórios (3/3)

- Números Aleatórios

- Ex.3: distribuições de probabilidade

A biblioteca NumPy também possui geradores para diversas outras

distribuições além da distribuição uniforme. Por exemplo:

binomial, multinomial, Poisson, ... Abaixo, um exemplo que gera vetor

com 5 números, considerando a Distribuição Normal padrão ($\mu=0$, $\sigma=1$)

```
np.random.seed(210720)
```

```
m = np.random.normal(size=(3,3))
```

```
print(m)
```

```
> [[-0.45364239  0.35051664  0.16079046]
```

```
[-0.27040396 -0.69894534 -0.37336035]
```

```
[ 1.52501494  0.40233551  2.63906279]]
```

Números Aleatórios (4/4)

- **Números Aleatórios - Exemplos de Funções:** *(McKinney, 2017)*

Function	Description
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

Indexação Booleana (1/3)

- Indexação booleana (1/3)

- Consiste em selecionar os elementos de um array, com base no conteúdo de um outro array, que deve ser de booleanos.

```
v = np.array([5, 10, 15, 20, 25, 30])
```

```
b = np.array([True, True, False, False, True, True])
```

```
v[b]           # [ 5 10 25 30]
```

- Podemos construir filtros com o uso dos operadores `|` (ou), `&` (and) e `!` (not)

```
filtro1 = (v < 10) | (v > 25)
```

```
filtro2 = (v > 10) & (v < 25)
```

```
v[filtro1]     # [ 5 30]
```

```
v[filtro2]     # [ 15 20]
```

Indexação Booleana (2/3)

- **Indexação booleana (2/3)**

- Observe que o filtro passado deve ser um array de booleanos!

```
vet_notas = np.array([7.4, 8.7, 7.6, 7.3, 8.3, 7.0, 7.3, 7.4, 6.9, 6.7])
```

```
print(vet_notas > 7.5)
```

```
> [False True True False True False False False False False]
```

```
vet_selecionados = vet_notas[vet_notas > 7.5]
```

```
print(vet_selecionados)
```

```
> [8.7 7.6 8.3]
```

Indexação Booleana (3/3)

- Indexação booleana (3/3)

- No exemplo abaixo, como selecionar todas as linhas cuja soma é menor ou igual a 2 (*retirado do manual da numpy*).

```
x = np.array([[0, 1], [1, 1], [2, 2]])
```

```
[[0 1]
 [1 1]
 [2 2]]
```

```
soma_linhas = x.sum(axis=1)
```

```
[1 2 4]
```

```
filtro = soma_linhas <=2
```

```
[ True  True False]
```

```
selecao = x[filtro, :]
```

```
[[0 1]
 [1 1]]
```

```
print(selecao)
```

Atualização Condicional (1/2)

- I. Usando Indexação booleana

- Também podemos atualizar um ndarray usando a indexação booleana
- No exemplo abaixo, trocamos todos os NaN por 0

```
a = [np.NaN, 50, np.NaN, 9], [6, np.NaN, 18, 100]]
```

```
[[ nan 50. nan  9.]  
 [  6. nan 18. 100.]]
```

```
a[np.isnan(a)] = 0
```

```
[[ 0. 50.  0.  9.]  
 [ 6.  0. 18. 100.]]
```


Atualização Condicional (2/2)

- II. Com a função `where()`
 - Com a função `where()`, podemos definir condições no estilo if-else
 - `np.where(condição, valor p/ resultado True, valor p/ resultado False)`
 - **Ex.:** Atualiza todos os valores abaixo de 5 para 0 e os demais para 1

```
numeros = np.arange(1,10).reshape((3,3))
```

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

```
 [4 5 6]
```

```
 [7 8 9]]
```

```
np.where(numeros < 5, 0, 1)
```

```
[[0 0 0]
```

```
 [0 1 1]
```

```
 [1 1 1]]
```

np.NaN

- **np.NaN** (*Not-a-Number*) é um valor especial utilizado para representar dados omissos (*missing*).
- **isnan()**: função que testa se valor é NaN.

```
m = np.genfromtxt('omissos.csv', delimiter=',')  
print(m)
```

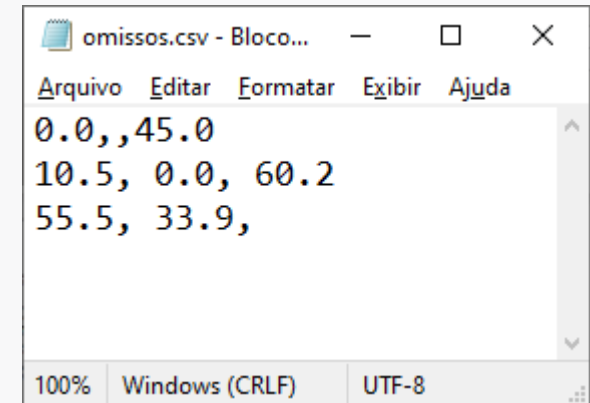
```
>
```

```
[[ 0.  nan 45. ]  
 [10.5  0. 60.2]  
 [55.5 33.9 nan]]
```

```
print(np.isnan(m))
```

```
>
```

```
[[False True False]  
 [False False False]  
 [False False True]]
```



Resumo (1/2)

- **Lista *versus* ndarray**

- **lista**

- ED nativa.
- Implem. por array de referências.
- Aceita elementos de diferentes tipos (lista mista).
- Simples.
- Projetada para aplicações de propósito geral.
- Ineficiente e inadequada para operações matemáticas.
- Pouco prática para trabalhar com arrays com 2 ou mais dimensões.
- Possui menos de 15 métodos, todos para operações básicas (inserir, remover, etc.).

- **ndarray**

- ED do pacote numpy
- Implementado por array compacto.
- Todos os elementos devem ser de um mesmo tipo.
- Sofisticado.
- Projetado para computação científica.
- Rápido e conveniente para operações matemáticas.
- Muito prático para trabalhar com arrays com 2 ou mais dimensões.
- Enorme número de métodos, não apenas para operações básicas, mas também para operações matemáticas, estatísticas, etc.

Resumo (2/2)

• NumPy e outras bibliotecas

- A NumPy é a base que estrutura a maioria dos outros pacotes para ciência de dados do Python.
- É muito comum encontrar na documentação destas bibliotecas diversos exemplos que se baseiam na NumPy.
- **Ex.:** Naive Bayes na scikit-learn

PARTE 1: gera uma base de dados usando a NumPy

import numpy as np

rng = np.random.RandomState(1)

X = rng.randint(5, size=(6, 100))

matriz com 6 linhas e 100 colunas

Y = np.array([1, 2, 3, 4, 4, 5])

vetor com 6 posições

#PARTE 2: classificação com a scikit-learn

from sklearn.naive_bayes **import** BernoulliNB

clf = **BernoulliNB()** *# instancia um objeto da classe BernoulliNB*

clf.fit(X, Y) *# executa o método fit() para treinar o modelo*

print(clf.predict(X[2:3])) *# executa o método predict() para classificar*

[3] *# a classe é retornada em um ndarray*

Tarefa

- (1) – Sem implementar um laço, escreva um programa para testar se um array possui ao menos um elemento com valor negativo.
- (2) – Escreva um programa para criar um array 5x6, contendo todos os inteiros compreendidos entre 21 e 50
- (3) – Escreva um programa para criar um array 3x10, onde a primeira linha tem 10 zeros, a segunda 10 números 1 e a terceira 10 números 5.
- (4) – Escreva um programa para calcular a soma da diagonal principal de uma matriz quadrada de ordem m.
- (5) – Escreva um programa que gere a matriz da página 42 (reproduzida abaixo) e depois obtenha a fatia com os elementos sombreados em amarelo.

```
[[1 1 5 5 1 3 2 4 4 5]  
 [4 3 3 2 4 3 1 3 3 1]  
 [3 5 5 3 4 1 5 1 5 5]  
 [2 1 1 1 5 4 4 3 2 5]  
 [1 2 1 4 1 4 5 2 2 5]  
 [4 1 2 1 2 4 2 4 1 4]  
 [3 3 3 5 2 5 4 5 1 1]]
```

Referências

- Corrêa, E. (2020). “Meu Primeiro Livro de Python”. V 2.0.0, edubd, 2020. (*capítulo 6*).
 - Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python
- NumPy – Routines. <https://numpy.org/doc/stable/reference/routines.html>
- Numerical Python Course – Numerical Programming with Python
https://www.python-course.eu/numerical_programming_with_python.php