



Introdução à Inteligência Artificial

Módulo 01: Introdução à Inteligência Artificial (IA)

Professor: Dr. João Paulo R. R. Leite



UNIFEI



Softex



**MCTI
FUTURO**

FUTURO DO TRABALHO, TRABALHO DO FUTURO



Sumário

Introdução ao Python

01 Bibliotecas

02 NumPy

03 Pandas

04 Matplotlib



UNIFEI



Softex



**MCTI
FUTURO**

FUTURO DO TRABALHO. TRABALHO DO FUTURO



Objetivo principal

- O **objetivo** desta aula é aumentar as ferramentas disponíveis para desenvolvimento, mostrando algumas **bibliotecas** embutidas na linguagem Python e outras de terceiros. As bibliotecas são fundamentais para o desenvolvimento de sistemas de **Inteligência Artificial**, e esta aula tem como objetivo dar uma visão geral sobre algumas delas e apresentar alguns exemplos básicos.

Bibliotecas

Existem **dois tipos de bibliotecas** em Python:

- **Padrão**, chamada de “**The Python Standard Library**”, que vem inclusa na distribuição da linguagem. Por exemplo, temos módulos como **os**, **sys** e **time**. Mais informações podem ser obtidas em:

<https://docs.python.org/3/library/>

- Bibliotecas de **terceiros** (***third-party***), que precisam ser instaladas separadamente para serem utilizadas em seus programas. Por exemplo, temos: *NumPy*, *pandas* e *scikit-learn*. Algumas plataformas, como a Anaconda, já vem com essas bibliotecas previamente instaladas, facilitando seu uso.

The Python Standard Library

Módulo **sys**:

Este módulo provê acesso a algumas variáveis utilizadas ou mantidas pelo interpretador e a funções que interagem fortemente com ele. Veja:

```
import sys
sys.path
```

```
['C:\\Users\\João Paulo\\OneDrive\\Jupyter',
 'C:\\ProgramData\\Anaconda3\\python37.zip',
 'C:\\ProgramData\\Anaconda3\\DLLs',
 'C:\\ProgramData\\Anaconda3\\lib',
 'C:\\ProgramData\\Anaconda3',
 '',
 'C:\\ProgramData\\Anaconda3\\lib\\site-packages',
 'C:\\ProgramData\\Anaconda3\\lib\\site-packages\\win32',
 'C:\\ProgramData\\Anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\ProgramData\\Anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\ProgramData\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\João Paulo\\.ipython']
```

```
sys.platform
```

```
'win32'
```

sys.path retorna uma lista com todos os caminhos que o interpretador conhece para módulos da linguagem Python.

sys.platform retorna um valor que representa a plataforma ou sistema utilizado. No meu caso, *win32*, mas poderia ser também *linux*, *darwin*, etc.

The Python Standard Library

Módulo **os**:

Este módulo provê acesso a algumas funcionalidades dependentes do sistema operacional.

```
import os
print("Process id:", os.getpid()) # gets current process id
cwd = os.getcwd() # gets current directory
print("Current directory:", cwd)
print("List all files:", os.listdir(cwd)) # lists all files from directory "cwd"

# functions from os.path module
print("Absolute Pathname: ", os.path.abspath("text.txt"))
print("Exists or not?", os.path.exists("text.txt"))
print("File size:", os.path.getsize("text.txt"))
print("Is file?", os.path.isfile("text.txt"))
print("Is directory?", os.path.isdir("text.txt"))
```

Process id: 13268

Current directory: C:\Users\João Paulo\OneDrive\Jupyter

List all files: ['.ipynb_checkpoints', 'matematica.py', 'Python_Course1.ipynb', 'Python_Course2.ipynb', 'text.txt', '__pycache__']

Absolute Pathname: C:\Users\João Paulo\OneDrive\Jupyter\text.txt

Exists or not? True

File size: 57

Is file? True

Is directory? False

The Python Standard Library

Módulo **time**:

Este módulo provê acesso a funções relacionadas ao tempo.

```
import time

time_now = time.time() # gets current timestamp
print("time stamp:", time_now)

localtime = time.localtime(time_now) # gets local time
print(localtime)
print("%d/%d/%d" % (localtime.tm_year, localtime.tm_mon, localtime.tm_mday)) # gets date (yyyy/mm/dd)

localtime = time.asctime(localtime) # converts localtime struct to string
print("Local Time:", localtime)

time stamp: 1673010584.4963098
time.struct_time(tm_year=2023, tm_mon=1, tm_mday=6, tm_hour=10, tm_min=9, tm_sec=44, tm_wday=4, tm_yday=6, tm_isdst=0)
2023/1/6
Local Time: Fri Jan 6 10:09:44 2023
```

Mesmo sendo da biblioteca padrão do Python, é sempre necessário indicar o módulo a ser utilizado, acrescentando antes de seu uso o comando **import <nome_modulo>**.

Third-Party Libraries

Um dos motivos que tem tornado a linguagem Python popular e atraente é a **quantidade e qualidade de suas bibliotecas** fornecidas por terceiros, ou seja, que não fazem parte de sua biblioteca padrão.

Neste curso, veremos algumas bibliotecas que são úteis em projetos relacionados a **ciências de dados e inteligência artificial**, como **NumPy**, **pandas**, **matplotlib**, **SciPy** e **scikit-learn**. A partir delas, seremos capazes de implementar projetos com tratamento e visualização rica de dados e construir modelos de aprendizado de máquina com relativamente pouco esforço e poucas linhas de código.

Mas como acrescentar essas bibliotecas à minha instalação do Python?

Third-Party Libraries

Para isso, você deverá utilizar o ***pip*** (*Package Installer for Python*).

Sistema de gerenciamento de pacotes padrão utilizado para instalar e gerenciar pacotes de software escritos em Python. A maioria das distribuições do Python vem com o *pip* pré-instalado.

Para instalar um pacote, basta digitar no terminal do sistema operacional o comando “pip install”:

```
> pip install numpy
> pip install pandas
> pip install matplotlib
> pip install scipy
> pip install scikit-learn
```

Caso o pacote ainda não esteja instalado em seu sistema, ele fará automaticamente o *download* do pacote com suas dependências e o disponibilizará para uso em suas aplicações.

NumPy Library



A biblioteca numérica *open-source* NumPy (*Numerical Python*) é utilizada em praticamente todos os campos de ciências e engenharia, e serve como base para diversas outras bibliotecas. <https://numpy.org/>

Seu *core* é o **objeto *ndarray***, que é um arranjo de dados (*array*) n-dimensional homogêneo (matriz). O NumPy pode ser utilizado para realizar uma ampla variedade de operações em matrizes de dados, adicionando estruturas poderosas que garantem sua eficiência e uma enorme quantidade de **funções matemáticas de alto nível** para aplicação nos *ndarrays*.

A biblioteca em si é parcialmente escrita em Python, mas a maioria de **suas funcionalidades que requerem computação rápida são escritas em C e C++**, para melhor desempenho. Listas em Python são lentas para processar. O objeto *ndarray* deve ser priorizado, pois chega a ser **50 vezes mais rápido** do que uma lista tradicional.

A biblioteca pandas é uma ferramenta *open-source* rápida (utiliza NumPy), poderosa e fácil de usar. É amplamente utilizada para **análise e manipulação de dados** provenientes de vários tipos de fontes. <https://pandas.pydata.org/>.

Há três estruturas de dados principais disponibilizadas pelo pandas: **Series**, **DataFrame** e **Panel**:

- **Series** é um array unidimensional rotulado, capaz de conter qualquer tipo de dado. Os rótulos dos eixos são chamados de *index*. É uma estrutura semelhante aos ndarrays e *dict (index -> value)*.
- **DataFrame** é a estrutura mais popular do pandas e consiste em uma estrutura de dados bidimensional rotulada que possui colunas de tipos potencialmente diferentes. Funcionam como uma tabela (linhas e colunas), e são muito utilizadas para representar conjuntos de dados em sistemas de aprendizado de máquina.
- **Panel** é um container de dados tridimensional, parecido com um *array* de DataFrames.

matplotlib Library



É uma biblioteca muito popular e completa, utilizada para a criação de **visualizações de dados** em Python, que podem ser estáticas, animadas e mesmo interativas. <https://matplotlib.org/>.

Matplotlib é uma ferramenta para plotar gráficos, representando visualmente as estruturas de dados da linguagem Python de forma simples, através de poucos comandos.

Faz uso extensivo da biblioteca **NumPy** para prover boa performance na manipulação de coleções de dados.

É uma biblioteca que contém uma coleção de algoritmos matemáticos e funções para **computação científica** utilizando também as estruturas do NumPy. Contém um grande número de ferramentas, por exemplo, para **integração numérica, transformada de Fourier, interpolação, otimização, álgebra linear e estatística**. Mais informações em <https://scipy.org/>.

O “ecossistema” do SciPy inclui pacotes relacionados à computação científica e engenharia que podem ser baixados em conjunto, como os próprios NumPy, matplotlib e pandas, além de outros. No entanto, esta seção se refere apenas à biblioteca SciPy.

scikit-learn Library



É uma biblioteca *open-source* voltada para o **aprendizado de máquina**, suportando uma grande quantidade de modelos e algoritmos relacionados a esse campo de estudo. Provê ainda várias ferramentas para ajuste e seleção de modelos, pré-processamento de dados, e muito mais. <https://scikit-learn.org/stable/>.

- **Ajuste de modelos:** scikit-learn provê dezenas de algoritmos e modelos de *machine learning*, chamados de *estimators*. Cada *estimator* pode ser ajustado (*fit*) para um conjunto de dados.
- **Pré-processamento:** O fluxo de um sistema de *machine learning* (***pipeline***) geralmente começa com um passo de pré-processamento que transforma os dados de entrada visando a extrair um desempenho melhor do modelo. O scikit-learn fornece métodos para esta etapa.
- **Avaliação:** O ajuste de um modelo de machine learning aos dados de entrada não garante que o modelo terá o mesmo desempenho em dados nunca vistos. O scikit-learn provê métodos para avaliação do desempenho dos modelos, como a validação cruzada (*cross validation*).
- **Busca de parâmetros:** A biblioteca provê métodos para busca automatizada de melhores valores para parâmetros específicos de cada modelo, como o método *Grid Search*.

NumPy

Introdução

O principal objeto do NumPy é o **array n-dimensional homogêneo**, que funciona como uma tabela indexada de elementos (geralmente numéricos). As dimensões são chamadas de eixos ou **axis**. Para utilizar suas funcionalidades, a primeira coisa é expressar essa intenção através do comando **import**:

```
import numpy as np
```

E como **criar um ndarray**? Há diversas formas, dependendo de seu objetivo:

```
arr = np.array([1,2,3,4,5]) #one dimensional  
print(arr)
```

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

```
arr2 = np.array([[1,2,3], [4,5,6]]) #multidimensional  
print(arr2)
```

```
[[1 2 3]  
 [4 5 6]]
```

Podemos criar ndarrays utilizando **listas padrão do Python** como inicializadores. No exemplo ao lado são criados arranjos com uma ou mais dimensões.

Mas qual a necessidade? Lista não são idênticas? **Não!** **ndarrays** são muito mais eficientes computacionalmente.

Além disso, muitas vezes precisamos criar arrays com determinado tamanho, mas cujos elementos ainda são desconhecidos. NumPy oferece algumas opções para esta tarefa, criando arrays com dados padrão (**zero**, **um** ou **aleatório**). Para isso, basta especificar a dimensão dos arrays:

```
z1 = np.zeros(10) # one-dimensional (10 elements)
z2 = np.zeros([2,2]) # bidimensional (2x2)
print(z)
print(z2)
```

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

```
o1 = np.ones([3,1]) # array filled with "ones"
print(o1)
```

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

```
ran = np.empty([2,2]) # array filled with random values
print(ran)
```

```
[[1.11444155e-311 1.37959604e-306]
 [3.56022683e-307 1.37961913e-306]]
```

Há ainda a possibilidade de se criar uma matriz identidade através da função “**eye**”:

```
iden = np.eye(4) # creates 4x4 identity matrix
print(iden)
```

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

É possível também inicializar ndarrays com **sequencias predefinidas de valores**. Isso pode ser muito interessante ao trabalharmos com séries temporais ou de comportamento previsível. Veja:

```
seq = np.arange(10, 30, 3) # create array from range (10 to 30, step 3)
print(seq)
```

```
[10 13 16 19 22 25 28]
```

arange cria um array a partir de um **range** (valor inicial, valor final e passo).

```
seq2 = np.linspace(0, 99, 10) # 10 elements from 0 to 99, evenly spaced
print(seq2)
```

```
[ 0. 11. 22. 33. 44. 55. 66. 77. 88. 99.]
```

linspace(a, b, n) cria um array com n elementos **igualmente espaçados** entre a e b.

O ndarray possui ainda alguns atributos, caso seja necessário verificar suas **propriedades**:

```
a = np.array([[1,2], [3,4], [5,6]])
print("Dimensions:", a.shape)
print("Number of elements:", a.size)
print("Number of Dimensions:", a.ndim)
print("Data type:", a.dtype)
```



```
Dimensions: (3, 2)
Number of elements: 6
Number of Dimensions: 2
Data type: int32
```

É possível acessar os elementos do ndarray individualmente, através de **indexação**, ou acessar subsequências de dados através das técnica de fatiamento, ou **slicing**. Veja:

```
a = np.array([[1,2], [3,4], [5,6]])
print(a)
print("Third line (2), first column (0) =", a[2][0])
```

```
[[1 2]
 [3 4]
 [5 6]]
Third line (2), first column (0) = 5
```

```
a = np.arange(10)
print("Array: ", a)
print("Third element (2):", a[2])
print("Third to before eighth (2:7):", a[2:7])
print("From the second on (1:):", a[1:])
print("From the beginning to before ninth (:8): ", a[:8])
print("From second to before ninth, step 2 (1:8:2): ", a[1:8:2])
```

```
Array: [0 1 2 3 4 5 6 7 8 9]
Third element (2): 2
Third to before eighth (2:7): [2 3 4 5 6]
From the second on (1:): [1 2 3 4 5 6 7 8 9]
From the beginning to before ninth (:8): [0 1 2 3 4 5 6 7]
From second to before ninth, step 2 (1:8:2): [1 3 5 7]
```

O *fatiamento* pode ser muito importante para selecionar sequências de dados em sistemas de IA.

Algumas funções comuns para processamento de ndarrays:

```
a = np.array([[1,2], [3,4], [5,6]])
print("Flattened=")
print(a.flat[:]) # iterates through array, return an 1-D version
print("Transposed=")
print(a.transpose()) # transposes array
```

```
Flattened=
[1 2 3 4 5 6]
Transposed
[[1 3 5]
 [2 4 6]]
```

Processamento de arrays

```
vec = np.arange(6)
mat = vec.reshape(2,3)
print("Original=")
print(vec)
print("Reshaped=")
print(mat)
```

```
Original=
[0 1 2 3 4 5]
Reshaped=
[[0 1 2]
 [3 4 5]]
```

```
print("a=\n",a)
print("b=\n",b)
c = np.concatenate((a,b), axis=0)
d = np.concatenate((a,b), axis=1)
print("Concat (Axis 0) =\n", c)
print("Concat (Axis 1) =\n", d)
```

```
a=
[[0 1]
 [2 3]]
b=
[[5 6]
 [7 8]]
Concat (Axis 0) =
[[0 1]
 [2 3]
 [5 6]
 [7 8]]
Concat (Axis 1) =
[[0 1 5 6]
 [2 3 7 8]]
```

```
a = np.array([[1,5], [0,1]])
b = np.array([[1,1], [2,2]])
print("dot multiply=")
print(np.dot(a, b)) # dot multiplies matrices (also a@b)
print("multiply=")
print(np.multiply(a,b)) # multiplies element-by-element (also a*b)
print("add=")
print(np.add(a,b)) # adds element-by-element
print("subtract=")
print(np.subtract(a,b)) # subtracts element-by-element
print("divide=")
print(np.divide(a,b)) # divides element-by-element
```

```
dot multiply=
[[11 11]
 [ 2  2]]
multiply=
[[1 5]
 [0 2]]
add=
[[2 6]
 [2 3]]
subtract=
[[ 0  4]
 [-2 -1]]
divide=
[[1.  5. ]
 [0.  0.5]]
```

Operações em matrizes

Algumas funções comuns para **processamento de ndarrays**:

Permutações e Ordenação

```
a = np.arange(10)
print("Original =", a)
np.random.shuffle(a) # shuffles a
print("Shuffled =", a)
b = np.random.permutation(a) # returns permutation of a
print("Permutation =", b)
c = np.sort(b) # sorts b
print("Sorted =", c)
```

```
Original = [0 1 2 3 4 5 6 7 8 9]
Shuffled = [2 7 4 1 0 9 8 3 6 5]
Permutation = [6 2 0 9 7 3 8 1 4 5]
Sorted = [0 1 2 3 4 5 6 7 8 9]
```

```
a = np.array([[5,6,2], [0,1,4]])
print("a=\n",a)
print("min values (axis 0) =", np.amin(a, 0)) # min values (axis 0)
print("max values (axis 1) =", np.amax(a, 1)) # max values (axis 1)
print("Median value =", np.median(a))
print("Mean value =", np.mean(a))
```

```
a=
[[5 6 2]
 [0 1 4]]
min values (axis 0) = [0 1 2]
max values (axis 1) = [6 4]
Median value = 3.0
Mean value = 3.0
```

Grandezas estatísticas

```
r1 = np.random.normal(0, 0.1, 5) # mean, std dev, size
r2 = np.random.uniform(0, 5, 3) # low, high, size
r3 = np.random.poisson(5, 6) # expected number of events at interval, size
```

```
print("r1 =", r1)
print("r2 =", r2)
print("r3 =", r3)
```

```
r1 = [ 0.01322488  0.02668272  0.15369526 -0.0539219  -0.15328944]
r2 = [0.77247727 0.16240803 1.87572014]
r3 = [1 2 3 4 5 6]
```

Distribuições de probabilidade

```
a = np.array([0,30,45,60,90]) # array of degrees
arad = a*np.pi/180 # array of radians
print("degrees = ", a)
print("radians = ", arad)
print("sine= ", np.sin(arad))
print("cosine =", np.cos(arad))
print("tangent = ", np.tan(arad))
```

```
degrees = [ 0 30 45 60 90]
radians = [0.          0.52359878 0.78539816 1.04719755 1.57079633]
sine= [0.          0.5          0.70710678 0.8660254  1.          ]
cosine = [1.00000000e+00 8.66025404e-01 7.07106781e-01 5.00000000e-01
 6.12323400e-17]
tangent = [0.00000000e+00 5.77350269e-01 1.00000000e+00 1.73205081e+00
 1.63312394e+16]
```

Funções trigonométricas

pandas

Introdução

A biblioteca **pandas** é uma ferramenta rápida e poderosa para análise e manipulação de dados. Para utilizá-la, é necessário importar também a biblioteca NumPy.

```
import pandas as pd
import numpy as np
```

A primeira estrutura importante disponibilizada pelo pandas é chamada de **Series**. Trata-se de um array **unidimensional** capaz de armazenar qualquer tipo de dados.

```
s1 = pd.Series([1,3,5,7,9]) # create Series from Python List
s2 = pd.Series(np.linspace(0,9,5)) # creates Series from NumPy ndarray
s3 = pd.Series({'a': 0., 'b': 1., 'c': 2.}) # creates Series from Python dict
print(s1)
print(s2)
print(s3)
```

```
0    1
1    3
2    5
3    7
4    9
dtype: int64
0    0.00
1    2.25
2    4.50
3    6.75
4    9.00
dtype: float64
a    0.0
b    1.0
c    2.0
dtype: float64
```

Podemos criar séries utilizando **listas padrão ou dict do Python** como inicializadores. Além disso, também é possível utilizar um **ndarray** do NumPy.

Repare que a série mantém uma estrutura de dados de dimensão única, em que cada elemento possui um **índice** (que pode ser numérico ou não).

Uma das estruturas mais importantes e populares utilizadas em projetos de ciência de dados e inteligência artificial é o **pandas DataFrame**. Trata-se de uma estrutura bidimensional rotulada cujas colunas podem ter dados de tipos diferentes. Funciona como uma **planilha eletrônica** ou uma **tabela SQL**.

```
In [24]: mat = np.random.randn(7,4) # creates 7x4 ndarray with random numbers (normal dist).
df = pd.DataFrame(mat, index=[2,3,4,5,6,7,8], columns=list("ABCD")) # creates dataframe with labels
df
```

Out[24]:


	A	B	C	D
2	1.421406	0.729310	-0.489288	-1.215472
3	-1.603686	0.152658	0.131814	-0.137363
4	-0.432319	0.804078	-0.541167	-1.132314
5	-0.119594	0.854355	0.173708	0.765991
6	0.725029	-0.738269	-0.923636	0.212585
7	-1.884556	-0.403590	1.077085	-0.200729
8	-0.091594	0.557940	-0.667125	-0.650090

Repare que, para inicializá-lo, foram passados como parâmetros uma **matriz de dados** (ndarray), os índices a serem utilizados para cada linha da tabela (2 a 8) e os **nomes das colunas** da tabela (A, B, C e D). Se não forem passados nomes de linhas e colunas, o DataFrame cria rótulos numéricos **a partir de zero**.

```
print(df['A'][5]) # accessing by index
```

-2.2639683432942337

Para acessar um elemento do DataFrame, selecione **primeiro a coluna e, depois, a linha**.



	0	1	2	3
0	1.868789	-0.934684	-0.185940	0.284661
1	-0.093929	1.733739	-0.899163	0.071886
2	0.997319	-0.136561	-0.190550	-1.322918
3	-1.056015	-0.285627	1.049264	-0.238058
4	0.548422	-1.392888	0.687372	-0.561092
5	0.714918	-0.140793	0.780877	1.095373
6	-1.237656	-1.603354	-0.700277	-1.003014

Caso tenhamos **DataFrames** muito grandes, podemos “dar apenas uma olhada” em seus primeiros ou últimos registros (linhas da tabela), utilizando, respectivamente, as funções **head** e **tail**.

```
dt = np.arange(300).reshape(50,6)
df2 = pd.DataFrame(dt) # dataframe contains 50 rows x 6 cols
df2.head() # shows first 5 records (default)
```

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23
4	24	25	26	27	28	29

```
df2.tail(3) # show last 3 records
```

	0	1	2	3	4	5
47	282	283	284	285	286	287
48	288	289	290	291	292	293
49	294	295	296	297	298	299

O DataFrame possui atributos que retornam os índices relacionados às linhas e colunas, além dos valores puros contidos na estrutura.

```
df = pd.DataFrame(np.random.randn(6,3)
                  , index=['a', 'b', 'c', 'd', 'e', 'f']
                  , columns=list("ABC"))
print("Rows=\n", df.index)
print("Columns=\n", df.columns)
print("Values=\n", df.values)
```

```
Rows=
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
Columns=
Index(['A', 'B', 'C'], dtype='object')
Values=
[[ 1.34797015 -1.16230074  1.4309906 ]
 [-0.73024564 -0.41366743 -1.29370851]
 [ 0.43383433  2.59835691  1.67759534]
 [-0.46337847  0.07030282  0.68182318]
 [-1.45239302 -1.11944693 -0.42079567]
 [-1.69678679  0.68686998  1.01289226]]
```

O DataFrame conta ainda com vários métodos para **extração de informações estatísticas dos dados**, incluindo um contador de valores, média, desvio padrão, soma, mediana, variância, e assim por diante. Este tipo de informação pode ser importante durante a fase de pré-processamento de dados para IA conhecida como extração de características.

```
print(df.mean()) # mean value per column
print(df.std()) # standard deviation per column
print(df.sum()) # total sum per column
```

```
A    -0.426833
B     0.110019
C     0.514800
dtype: float64
A     1.153137
B     1.409311
C     1.150090
dtype: float64
A    -2.560999
B     0.660115
C     3.088797
dtype: float64
```

```
df.describe() # summarizes statistical data
```

	A	B	C
count	6.000000	6.000000	6.000000
mean	-0.426833	0.110019	0.514800
std	1.153137	1.409311	1.150090
min	-1.696787	-1.162301	-1.293709
25%	-1.271856	-0.943002	-0.145141
50%	-0.596812	-0.171682	0.847358
75%	0.209531	0.532728	1.326466
max	1.347970	2.598357	1.677595

Outro método, presente tanto nas Series quanto nos DataFrames, é `pct_change()`. Este método retorna a **mudança percentual** do valor a cada novo registro. Imagine uma série que começa com 1.0 e vai para 2.0. Mudança de 100%. Depois, cai para 0.5, em uma mudança de -75%. Assim por diante. O DataFrame faz o mesmo, mas por coluna da tabela.

```
s = pd.Series([1.0, 2.0, 0.5, 3.5, 4.0, 8.0])
print(s.pct_change())
```

```
0      NaN
1    1.000000
2   -0.750000
3    6.000000
4    0.142857
5    1.000000
dtype: float64
```

```
df.pct_change() # show the percentage change per column
```

	A	B	C
a	NaN	NaN	NaN
b	-1.541737	-0.644096	-1.904065
c	-1.594094	-7.281270	-2.296734
d	-2.068100	-0.972943	-0.593571
e	2.134356	-16.923215	-1.617162
f	0.168270	-1.613580	-3.407088

Outro ponto importante, é que muitas vezes os dados não são coletados de maneira perfeita, e há falhas em nossas bases de dados: os dados faltantes, ou **missing values**. O marcador padrão para este tipo de dado é o valor “NaN” (*not a number*), presente no NumPy como *np.nan*. Veja:

```
s = pd.Series([1, 3, 5, np.nan, 9])  
print(s)
```

```
0    1.0  
1    3.0  
2    5.0  
3    NaN  
4    9.0  
dtype: float64
```

Neste caso, a série possui um valor faltante, entre 5 e 9. É preciso **lidar com este fato** de alguma maneira

Primeiramente, precisamos ser capazes de **detectar a presença de missing values** em nossas bases de dados. Considere o DataFrame abaixo:

	one	two	three
a	-0.600268	1.230474	-0.951396
b	NaN	NaN	NaN
c	0.450227	-0.210962	0.426927
d	NaN	NaN	NaN
e	-1.021449	1.647701	0.797621
f	-0.031304	-0.445783	-1.772174
g	-0.915414	0.859549	-0.870282

Função `isna()`



```
df.isna() # returns if elements are NaN
```

	one	two	three
a	False	False	False
b	True	True	True
c	False	False	False
d	True	True	True
e	False	False	False
f	False	False	False
g	False	False	False

Além disso, podemos utilizar a função **isna()** em uma única coluna do DataFrame (uma Series) e, também, verificar a presença de “algum” *missing value* na massa de dados utilizando a função **any()**. Veja:

```
df["one"].isna()
```

```
a    False
b     True
c    False
d     True
e    False
f    False
g    False
Name: one, dtype: bool
```

Teste se algum elemento da matriz ao longo de um determinado eixo é avaliado como Verdadeiro (True).

```
df.isna().values.any() # are there any NaN?
```

```
True
```

Se houver dados faltantes, podemos optar por **remover** todas as linhas que os contenham no banco de dados, através da função **dropna()** ou **substituí-los** por algum valor de interesse (média, mediana) com a função **fillna()**.

```
df.dropna()
```

	one	two	three
a	0.086704	-0.066938	1.558439
c	-0.524289	-0.983664	1.869039
e	0.676707	1.369433	-0.856776
f	-0.967795	1.069423	-1.871538
g	0.430263	0.325301	0.218659

Removeu linhas **b** e **d** do banco de dados, pois continham **NaN**.

```
df.fillna(df.mean())
```

	one	two	three
a	-1.412682	-0.278751	0.717026
b	-0.209864	0.262924	0.292562
c	-0.108953	0.532590	0.260538
d	-0.209864	0.262924	0.292562
e	0.420629	0.228384	-0.299560
f	0.020465	-0.046401	0.543470
g	0.031222	0.878796	0.241336

Substituiu as linhas do banco de dados que continham **NaN**, pelos valores médios das colunas

```
df.mean()
```

```
one      -0.209864
two       0.262924
three     0.292562
dtype: float64
```

matplotlib

Introdução

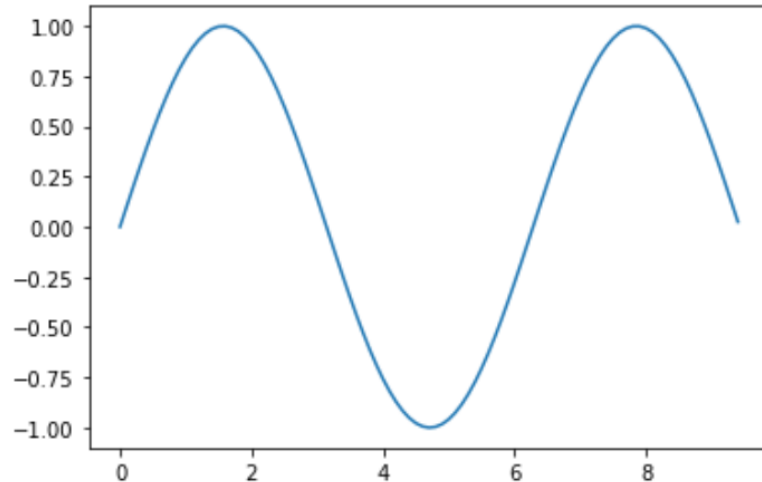
A biblioteca **matplotlib** é utilizada para geração de gráficos e outros tipos de visualizações para os dados. Para utilizá-la, devemos importar as bibliotecas envolvidas:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Para plotar um gráfico simples, basta seguir alguns passos:

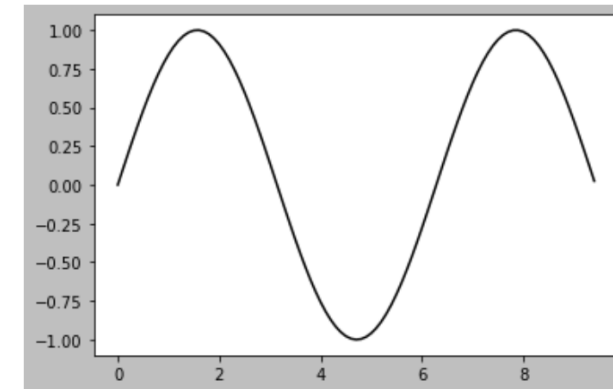
```
# input data
x = np.arange(0, 3*np.pi, 0.1) # values from 0 to 3*pi, step 0.1
y = np.sin(x) # sin values from x

#plot graph
plt.plot(x, y) #create graph
plt.show() # show graph
```



O código ao lado gera um gráfico com todas as **configurações básicas de cor, estilo, formato, etc.** Se desejar mudar o estilo:

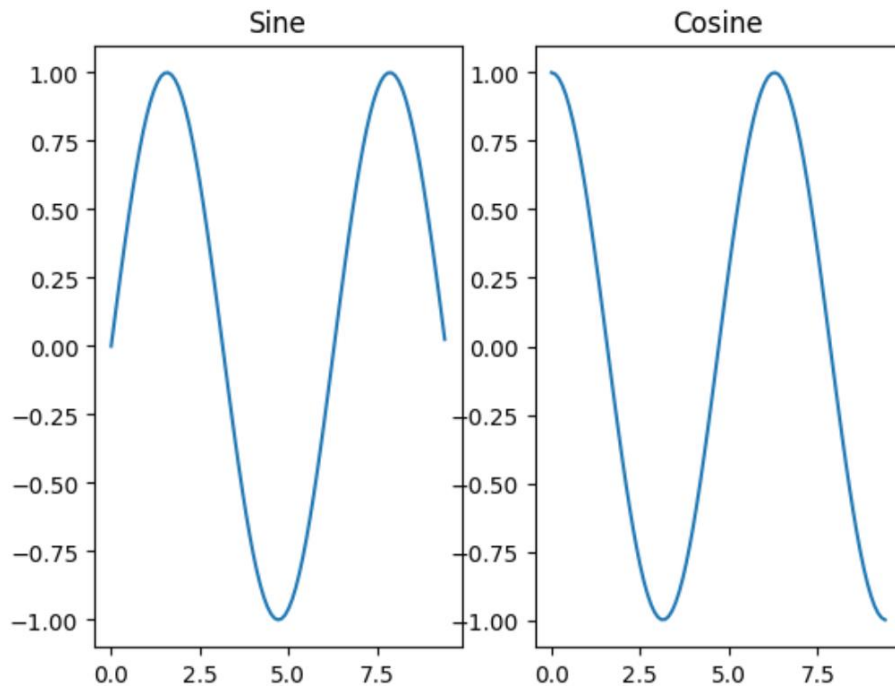
```
plt.style.available # shows list of available styles
plt.style.use("grayscale") # sets new style
plt.plot(x, y) #create graph
plt.show() # show graph
```



É possível **combinar mais de um gráfico** em uma única imagem e personalizar o estilo:

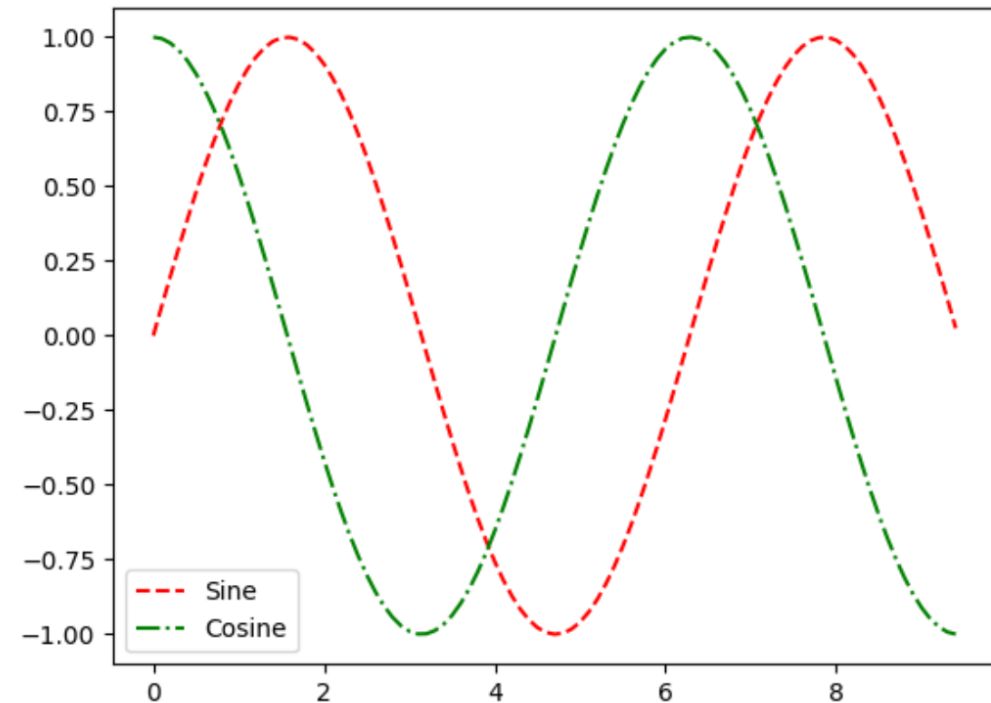
```
x = np.arange(0, 3*np.pi, 0.1)
y_sin = np.sin(x) # sin values
y_cos = np.cos(x) # cos values

#plot graph 1
plt.subplot(1,2,1) # grid with 1 line and 2 columns (slot 1)
plt.plot(x, y_sin)
plt.title("Sine") # give title
#plot graph 2
plt.subplot(1,2,2) # grid with 1 line and 2 columns (slot 2)
plt.plot(x, y_cos)
plt.title("Cosine") # give title
# show graph
plt.show()
```



```
x = np.arange(0, 3*np.pi, 0.1)
y_sin = np.sin(x) # sin values
y_cos = np.cos(x) # cos values

plt.plot(x, y_sin, color="red", linestyle="--", label="Sine")
plt.plot(x, y_cos, color="green", linestyle="-.", label="Cosine")
plt.legend() # show Legend
plt.show() # show graph
```

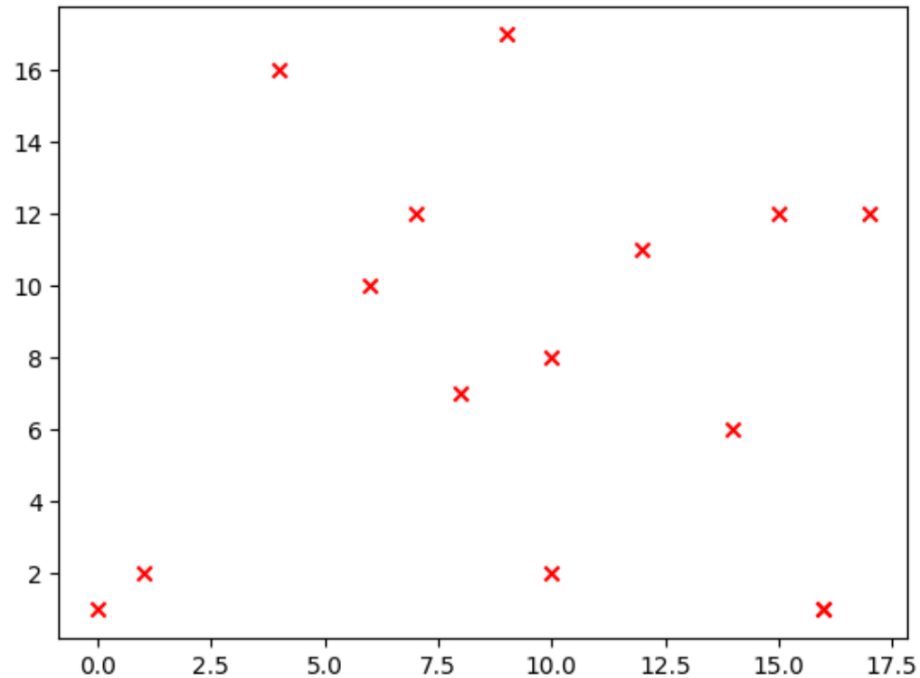


Outros tipos de gráficos estão disponíveis, como *scatterplots* e *histogramas*:

scatterplot

```
a = np.random.randint(0,20,15)
b = np.random.randint(0,20,15)
print(a)
print(b)
plt.scatter(a, b, c='red', marker='x')
plt.show()
```

```
[10 15  7  1  4  0 17 14 16 16 12  9  6  8 10]
[ 8 12 12  2 16  1 12  6  1  1 11 17 10  7  2]
```

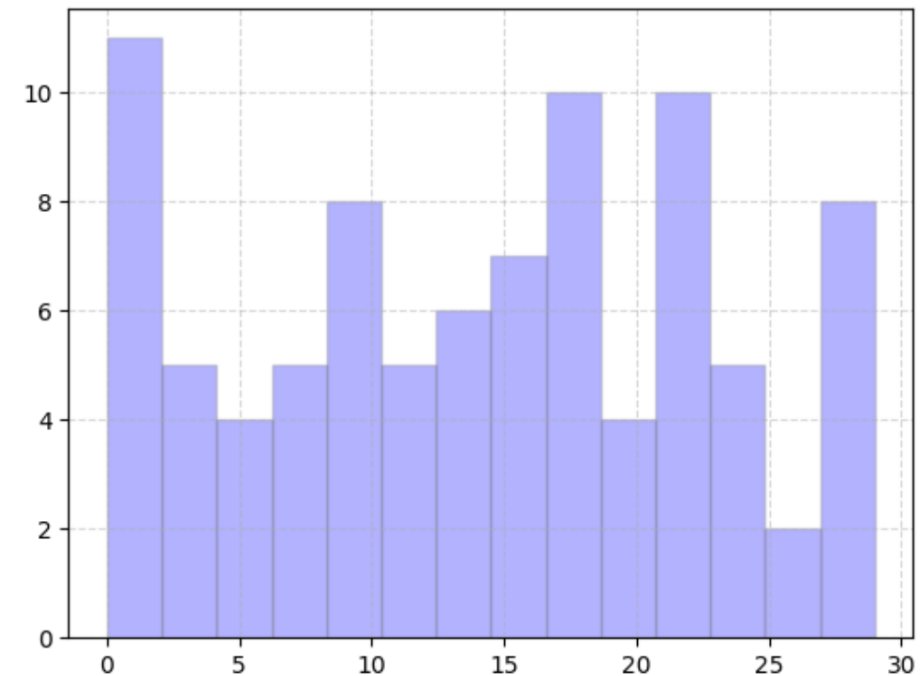


histograma

```
t = np.random.randint(0,30,90)
print(t)
```

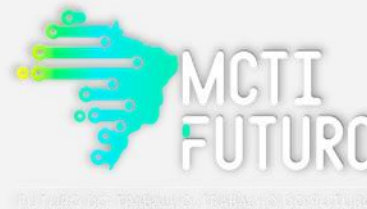
```
plt.hist(t, facecolor="blue", edgecolor="gray", bins=14, alpha=0.3)
plt.grid(linestyle="--", alpha=0.5)
plt.show()
```

```
[12 22 14 12 10 28  5 17  2 15 14 23 18 24 11 24  9 24 18  8  1 19  1 28
15 12 25  4  1  1 20  0 13 27  9  5 10  0 17  4 16 15  0 19 18 22  7 10
 0 28  8 18  7  9 19  3  6 16 13  7 18  9 17 21 21 17 13 22 28 16 28  4
29 18 22 28 21 22  0 12 14 22  2  5  9 22 24 25  3 16]
```



Apoio

Este projeto é apoiado pelo Ministério da Ciência, Tecnologia e Inovações, com recursos da Lei nº 8.248, de 23 de outubro de 1991, no âmbito do [PPI-Softex | PNM-Design], coordenado pela Softex.





João Paulo Reus Rodrigues Leite
Universidade Federal de Itajubá
e-mail: joaopaulo@unifei.edu.br



UNIFEI



Softex



FUTURO DO TRABALHO. TRABALHO DO FUTURO

