

Módulo de Programação Python

Trilha Python - Aula 31/32: Visualização de dados: Matplotlib recursos básico



Python - Visualização de dados: Matplotlib recursos básico

Residência em Software

Professor:
Esbel T. Valero Orellana

INSTITUIÇÃO EXECUTORA: CEPEDI, UESC
COORDENADORA: MCTI FUTURO, Softex
APOIO: GOVERNO FEDERAL, BRASIL 2023-2025

Uma breve discussão sobre a importância de visualização de dados.

O quarteto de *Anscombe* é formado por quatro conjuntos de dados que, se analisados utilizando descritores estatísticos simples, são quase idênticos. Entretanto estes conjuntos possuem distribuições muito diferentes, o que pode ser constatado quando representadas graficamente.

Cada conjunto de dados consiste em onze pares (x, y) e foram construídos em 1973, pelo estatístico *Francis Anscombe*, para demonstrar a importância de representar graficamente os dados ao analisá-los e o efeito de outliers e outras observações, nas propriedades estatísticas.

O autor descreveu seu artigo como tendo o objetivo de contrariar a impressão entre os estatísticos de que "os cálculos numéricos são exatos, mas os gráficos são aproximados".

```

In [1]: 1 # Quarteto de Anscombe
        2 QA_x1 = [10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0,
        3 QA_y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84,
        4
        5 QA_x2 = [10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0,
        6 QA_y2 = [9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13,
        7
        8 QA_x3 = [10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0,
        9 QA_y3 = [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15,
        10
        11 QA_x4 = [8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 19.0, 8.0, 8.0, 8.0
        12 QA_y4 = [6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56,

```

```

In [2]: 1 # Vamos converter cada quarteto em um ndarray
        2 import numpy as np
        3 QA_x1 = np.array(QA_x1)
        4 QA_y1 = np.array(QA_y1)
        5 QA_x2 = np.array(QA_x2)
        6 QA_y2 = np.array(QA_y2)
        7 QA_x3 = np.array(QA_x3)
        8 QA_y3 = np.array(QA_y3)
        9 QA_x4 = np.array(QA_x4)
        10 QA_y4 = np.array(QA_y4)

```

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Inicialmente podemos tentar visualizar estes dados na forma de uma tabela. Vamos construir um data frame com eles.

```

In [3]: 1 import pandas as pd
        2 #from IPython.display import HTML
        3
        4 colunas = pd.MultiIndex.from_product(['I', 'II', 'III', 'IV'],
        5 index = [str(i) for i in range(1,12)]
        6 index = pd.Index(index, name="Obs. ")
        7 QA = pd.DataFrame(np.array([QA_x1, QA_y1, QA_x2, QA_y2, QA_x3,
        8                             columns=colunas, index=index)
        9 #HTML(QA.to_html(index=False))
       10 #print(QA.to_string(index=False))
       11 QA

```

```

Out [3]:

```

	Quarteto		I		II		III		IV
Coordenada	x	y	x	y	x	y	x	y	
Obs.									
1	10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58	
2	8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76	
3	13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71	
4	9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84	
5	11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47	
6	14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04	
7	6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25	
8	4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50	
9	12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56	
10	7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91	
11	5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89	

Podemos fazer algumas observações básicas deste conjunto de dados:

- Podemos constatar que os valores de x das três primeiras séries são os mesmos;
- Os valores de x da última série são todos 8 menos um que é 19;
- Os valores de x são inteiros enquanto que os valores de y não;
- Na terceira coluna de valores de y se destaca o valor 12.7, que é maior que os restantes valores da coluna;

Entretanto uma análise mais detalhada surge quando valíamos as séries desde o ponto de vista estatístico.

```
In [4]: 1 #QA.mean()
2 QA.loc["Mean", :] = QA.mean()
3 QA.loc["Std", :] = QA.std()
4 QA.loc["Var", :] = QA.var()
5 QA
```

```
Out [4]:
```

	Quarteto		I		II		III		
	Coordenada	x	y	x	y	x	y	x	
	Obs.								
	1	10.000000	8.040000	10.000000	9.140000	10.000000	7.460000	8.000000	
	2	8.000000	6.950000	8.000000	8.140000	8.000000	6.770000	8.000000	
	3	13.000000	7.580000	13.000000	8.740000	13.000000	12.740000	8.000000	
	4	9.000000	8.810000	9.000000	8.770000	9.000000	7.110000	8.000000	
	5	11.000000	8.330000	11.000000	9.260000	11.000000	7.810000	8.000000	
	6	14.000000	9.960000	14.000000	8.100000	14.000000	8.840000	8.000000	
	7	6.000000	7.240000	6.000000	6.130000	6.000000	6.080000	8.000000	
	8	4.000000	4.260000	4.000000	3.100000	4.000000	5.390000	19.000000	1
	9	12.000000	10.840000	12.000000	9.130000	12.000000	8.150000	8.000000	
	10	7.000000	4.820000	7.000000	7.260000	7.000000	6.420000	8.000000	
	11	5.000000	5.680000	5.000000	4.740000	5.000000	5.730000	8.000000	
	Mean	9.000000	7.500909	9.000000	7.500909	9.000000	7.500000	9.000000	
	Std	3.162278	1.937024	3.162278	1.937109	3.162278	1.935933	3.162278	
	Var	11.788128	5.820684	11.788128	5.820912	11.788128	5.816966	11.788128	

Veja que temos agora um conjunto de descritores básicos que parecem indicar um porte semelhança entre as quatro séries.

Outra análise pertinente, neste ponto, é se existe algum tipo de correlação entre x e y .

```
In [5]: 1 QA.loc["Corr", ('I', 'x')] = np.corrcoef(QA.loc['1':'11', ('I',
2 QA.loc["Corr", ('II', 'x')] = np.corrcoef(QA.loc['1':'11', ('II
3 QA.loc["Corr", ('III', 'x')] = np.corrcoef(QA.loc['1':'11', ('I
4 QA.loc["Corr", ('IV', 'x')] = np.corrcoef(QA.loc['1':'11', ('IV
5 QA.loc["Corr", ('I', 'y')] = np.corrcoef(QA.loc['1':'11', ('I',
6 QA.loc["Corr", ('II', 'y')] = np.corrcoef(QA.loc['1':'11', ('II
7 QA.loc["Corr", ('III', 'y')] = np.corrcoef(QA.loc['1':'11', ('I
8 QA.loc["Corr", ('IV', 'y')] = np.corrcoef(QA.loc['1':'11', ('IV
9 QA
```

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

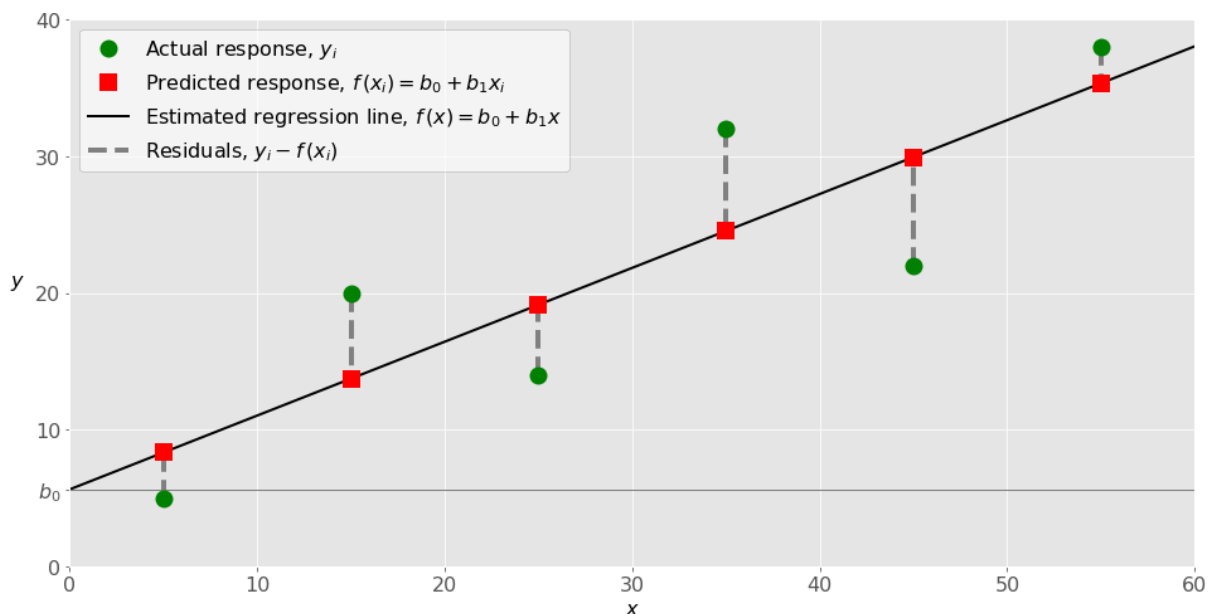
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.

Out [5]:

Quarteto		I		II		III	
Coordenada	x	y	x	y	x	y	x
Obs.							
1	10.000000	8.040000	10.000000	9.140000	10.000000	7.460000	8.000000
2	8.000000	6.950000	8.000000	8.140000	8.000000	6.770000	8.000000
3	13.000000	7.580000	13.000000	8.740000	13.000000	12.740000	8.000000
4	9.000000	8.810000	9.000000	8.770000	9.000000	7.110000	8.000000
5	11.000000	8.330000	11.000000	9.260000	11.000000	7.810000	8.000000
6	14.000000	9.960000	14.000000	8.100000	14.000000	8.840000	8.000000
7	6.000000	7.240000	6.000000	6.130000	6.000000	6.080000	8.000000
8	4.000000	4.260000	4.000000	3.100000	4.000000	5.390000	19.000000
9	12.000000	10.840000	12.000000	9.130000	12.000000	8.150000	8.000000
10	7.000000	4.820000	7.000000	7.260000	7.000000	6.420000	8.000000

11	5.000000	5.680000	5.000000	4.740000	5.000000	5.730000	8.000000
Mean	9.000000	7.500909	9.000000	7.500909	9.000000	7.500000	9.000000
Std	3.162278	1.937024	3.162278	1.937109	3.162278	1.935933	3.162278
Var	11.788128	5.820684	11.788128	5.820912	11.788128	5.816966	11.788128
Corr	0.816421	0.816421	0.816237	0.816237	0.816287	0.816287	0.816521

Um coeficiente de correlação próximo de um indica uma forte correlação entre as variáveis. Desta forma podemos supor que existe alguma relação funcional do tipo $y = f(x)$. Podemos, inicialmente, propor uma função linear e a mais simples é $f(x) = ax + b$. Como estimar qual a função, desta família de funções, que melhor se ajusta aos dados de cada série? A regressão linear fornece um mecanismo, não apenas para estimar os coeficientes desta função mas também uma estimativa de quão bom este ajuste é.



Regressão Linear não está dentro do escopo deste curso mas vamos tentar entender, de forma rápida, como implementar uma.

```
In [6]: 1 #Importe os pacotes e classes que você precisa.
        2 #pip install scikit-learn
        3 from sklearn.linear_model import LinearRegression
```

```
In [7]: 1 # Forneça dados para trabalhar e, eventualmente, faça as transf
        2 x = QA.values[0:11, 0].reshape(-1, 1)
        3 y = QA.values[0:11, 1]
```

Repare no `reshape` que utilizamos em `x`. Agora, você tem duas matrizes: a entrada, `x`, e a saída, `y`. O array `x` deve ser bidimensional, ou mais precisamente, deve ter uma coluna e quantas linhas forem necessárias. Isto para podermos utilizar estas entradas na classe `LinearRegression`. Agora temos

```
In [8]: 1 x
```

```
Out[8]: array([[10.],
               [ 8.],
               [13.],
               [ 9.],
               [11.],
               [14.],
               [ 6.],
               [ 4.],
               [12.],
               [ 7.],
               [ 5.]])
```

```
In [9]: 1 y
```

```
Out[9]: array([ 8.04,  6.95,  7.58,  8.81,  8.33,  9.96,  7.24,  4.26, 10.84,
                4.82,  5.68])
```

```
In [10]: 1 # Crie um modelo de regressão e ajuste-o aos dados existentes.
          2 QA1_model = LinearRegression()
```

```
In [11]: 1 QA1_model.fit(x, y)
```

```
Out[11]: LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.

On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
In [12]: 1 # Verifique os resultados do ajuste do modelo para saber se o m
          2 r_sq = QA1_model.score(x, y)
          3 print(f"coefficient of determination: {r_sq}")
```

```
coefficient of determination: 0.6665424595087748
```

O método `model.score(features, target)` retorna o R^2 do nosso modelo, que é a **porcentagem da variância explicada das previsões do modelo**. Este parâmetro é calculado comparando o modelo ajustado com uma linha de base constante que é escolhida tomando a média dos dados e traçando uma linha horizontal. O R^2 é sempre menor ou iguais a 1 e um valor mais alto é preferido.

```
In [13]: 1 print(f"intercept: {QA1_model.intercept_}")
```

```
intercept: 3.0000909090909094
```

```
In [ ]: 1 print(f"slope: {QA1_model.coef_}")
```

Ou seja, a função $f(x) = 0.5x + 3$ descreve a relação entre nossos dados x e y no que parece ser um bom ajuste de dados.

Podemos agora aplicar o mesmo para avaliar os resultados.

```
In [ ]: 1 # Vamos escolher o menor e o maior valor de x de todas as serie
2 x_min = QA.values[0:11, 0::2].min()
3 x_max = QA.values[0:11, 0::2].max()
4 (x_min, x_max)
5 # gerar um conjunto de valores de x para prever os valores de
6 QA_x1_pred = np.linspace(x_min - 1, x_max + 1, 20).reshape(-1,
7 # Aplique o modelo para previsões.
8 QA_y1_pred = QA1_model.predict(QA_x1_pred)
9 print(f"predicted response: {QA_y1_pred}")
```

Podemos aplicar este mesmo método para as restantes séries.

```
In [ ]: 1 #Para a QA2
2 x = QA.values[0:11, 2].reshape(-1, 1)
3 y = QA.values[0:11, 3]
4 QA2_model = LinearRegression()
5 QA2_model.fit(x, y)
6 r_sq = QA2_model.score(x, y)
7 print(f"coefficient of determination: {r_sq}")
8 print(f"intercept: {QA2_model.intercept_}")
9 print(f"slope: {QA2_model.coef_}")
10 QA_x2_pred = np.linspace(x_min - 1, x_max + 1, 20).reshape(-1,
11 QA_y2_pred = QA2_model.predict(QA_x2_pred)
```



```
In [ ]: 1 #Para a QA3
2 x = QA.values[0:11, 4].reshape(-1, 1)
3 y = QA.values[0:11, 5]
4 QA3_model = LinearRegression()
5 QA3_model.fit(x, y)
6 r_sq = QA3_model.score(x, y)
7 print(f"coefficient of determination: {r_sq}")
8 print(f"intercept: {QA3_model.intercept_}")
9 print(f"slope: {QA3_model.coef_}")
10 QA_x3_pred = np.linspace(x_min - 1, x_max + 1, 20).reshape(-1, 1)
11 QA_y3_pred = QA3_model.predict(QA_x3_pred)
```

```
In [ ]: 1 #Para a QA4
2 x = QA.values[0:11, 6].reshape(-1, 1)
3 y = QA.values[0:11, 7]
4 QA4_model = LinearRegression()
5 QA4_model.fit(x, y)
6 r_sq = QA4_model.score(x, y)
7 print(f"coefficient of determination: {r_sq}")
8 print(f"intercept: {QA4_model.intercept_}")
9 print(f"slope: {QA4_model.coef_}")
10 QA_x4_pred = np.linspace(x_min - 1, x_max + 1, 20).reshape(-1, 1)
11 QA_y4_pred = QA4_model.predict(QA_x4_pred)
```

Podemos então dizer que a função $f(x) = 0.5x + 3$ descreve a relação entre nossos dados x e y no que parece ser um bom ajuste de dados, para todas as quatro séries.

Como ficam estes dados num gráfico?

Visualização de dados com Matplotlib

Matplotlib é uma biblioteca de plotagem 2D/3D para **Python** que foi projetado para usar o tipo de dados **NumPy** e pode ser usada para gerar gráficos dentro de um programa Python.

```
In [ ]: 1 #pip install ipynb
2 import matplotlib.pyplot as plt
3 plt.style.use('classic')
4 %matplotlib widget
```

Usaremos a diretiva `plt.style` para escolher estilos estéticos apropriados para nossas figuras. Vamos definir inicialmente o estilo `classic`, o que garante que os gráficos que criamos usem o estilo clássico do **Matplotlib**:

```
In [ ]: 1 plt.style.use('classic')
```

Podemos agora ver, por exemplo, como fica o gráfico dos nossos modelos ajustados.

```
In [ ]: 1 fig = plt.figure()
2 plt.plot(QA_x1_pred, QA_y1_pred, '-', label='QA1_model')
3 plt.plot(QA_x2_pred, QA_y2_pred, '-', label='QA2_model')
4 plt.plot(QA_x3_pred, QA_y3_pred, '-', label='QA3_model')
5 plt.plot(QA_x4_pred, QA_y4_pred, '-', label='QA4_model')
```

O modo *widget* incorpora o recurso de podermos salvar o gráfico diretamente como uma imagem num arquivo. Entretanto, também é possível salvar o mesmo utilizando os recursos da biblioteca.

```
In [ ]: 1 fig.savefig("my_figure.png")
2 from IPython.display import Image
3 Image('my_figure.png')
```

Veja quais formatos de arquivos são suportados.

```
In [ ]: 1 fig.canvas.get_supported_filetypes()
```

Interface MATLAB-style

Matplotlib foi originalmente criado como uma alternativa **Python** para usuários do **MATLAB**, e grande parte de sua sintaxe reflete esse fato. As ferramentas estilo **MATLAB** estão contidas na interface `pypplot (plt)`. Por exemplo, o código a seguir provavelmente parecerá bastante familiar aos usuários do MATLAB.

```
In [ ]: 1 plt.figure()
2
3 # crie o primeiro dos dois painéis e defina o eixo atual
4 plt.subplot(2,2,1) # (rows, columns, panel number)
5 plt.plot(QA_x1_pred, QA_y1_pred, '-', label='QA1_model');
6
7 # crie o segundo painel e defina o eixo atual
8 plt.subplot(2,2,2)
9 plt.plot(QA_x2_pred, QA_y2_pred, '-', label='QA2_model');
10
11 # crie o terceiro painel e defina o eixo atual
12 plt.subplot(2,2,3)
13 plt.plot(QA_x3_pred, QA_y3_pred, '-', label='QA3_model');
14
15 # crie o quarto painel e defina o eixo atual
16 plt.subplot(2,2,4)
17 plt.plot(QA_x4_pred, QA_y4_pred, '-', label='QA4_model');
```

É importante observar que essa interface mantém o controle da figura e dos eixos “atuais”, onde todos os comandos `plt` são aplicados. Você pode obter uma referência a eles usando as rotinas

- `plt.gcf()` (obter valor atual) e
- `plt.gca()` (obter eixos atuais).

Embora essa interface seja rápida e conveniente para gráficos simples, é fácil encontrar problemas. Por exemplo, uma vez criado o segundo painel, como podemos voltar e adicionar algo ao primeiro? Isso é possível na interface estilo **MATLAB**, mas é um pouco desajeitado. Felizmente, existe uma maneira melhor.

Interface orientada a objetos

A interface orientada a objetos está disponível para situações mais complicadas e para quando você deseja ter mais controle sobre sua figura.

Em vez de depender de alguma referência de uma figura ou eixos "ativos", na interface orientada a objetos as funções de plotagem são **métodos** de objetos `Figura` e `Eixos` explícitos.

```
In [ ]: 1 # Primeiro crie uma grade de gráficos
        2 # ax será um array de dois por dois objetos Axes
        3 fig, ax = plt.subplots(2, 2)
        4 # Chama o método plot() no objeto apropriado
        5 ax[0,0].plot(QA_x1_pred, QA_y1_pred, '-', label='QA1_model')
        6 ax[0,1].plot(QA_x2_pred, QA_y2_pred, '-', label='QA2_model')
        7 ax[1,0].plot(QA_x3_pred, QA_y3_pred, '-', label='QA3_model')
        8 ax[1,1].plot(QA_x4_pred, QA_y4_pred, '-', label='QA4_model')
        9 #plt.show()
```

Para gráficos mais simples, a escolha de qual estilo usar é em grande parte uma questão de preferência, mas a abordagem orientada a objetos pode se tornar uma necessidade à medida que os gráficos se tornam mais complicados.

Ao longo desta aula, alternaremos entre as interfaces estilo MATLAB e orientadas a objetos, dependendo do que for mais conveniente.

Na maioria dos casos, a diferença é tão pequena quanto mudar `plt.plot()` para `ax.plot()`, mas existem algumas dicas que iremos destacar à medida que surgirem nas seções seguintes.

Gráficos de linhas simples

Talvez o mais simples de todos os gráficos seja a visualização de uma única função $y = f(x)$.

Para todos os gráficos do **Matplotlib**, começamos criando uma figura e um eixo. Na sua forma mais simples, uma figura e eixos podem ser criados da seguinte forma.

```
In [ ]: 1 fig = plt.figure()
        2 ax = plt.axes()
```

No **Matplotlib**, a `figure` (uma instância da classe `plt.Figure`) pode ser pensada como um único contêiner que contém todos os objetos que representam eixos, gráficos, texto e rótulos.

Os `axes` (uma instância da classe `plt.Axes`) é o que vemos acima: uma caixa delimitadora com marcas de seleção e rótulos, que eventualmente conterá os elementos do gráfico que compõem nossa visualização.

Frequentemente utilizaremos o nome da variável `fig` para nos referirmos a uma instância de figura, e `ax` para nos referirmos a uma instância de eixos ou grupo de instâncias de eixos.

Depois de criarmos os eixos, podemos usar a função `ax.plot` para plotar alguns dados.

```
In [ ]: 1 fig = plt.figure()
        2 ax = plt.axes()
        3
        4 x = np.linspace(0, 2*np.pi, 100)
        5 ax.plot(x, np.sin(x));
```

Alternativamente, podemos usar a interface **MATLAB** e deixar a figura e os eixos serem criados para nós em segundo plano.

```
In [ ]: 1 plt.figure()
        2 plt.plot(x, np.sin(x));
```

```
In [ ]: 1 plt.figure()
        2 plt.plot(x, np.sin(x))
        3 plt.plot(x, np.cos(x));
```

Ajustando o gráfico: cores e estilos de linha

O primeiro ajuste que você pode querer fazer em um gráfico é controlar as cores e estilos das linhas.

A função `plt.plot()` recebe argumentos adicionais que podem ser usados para especificá-los.

Para ajustar a cor, você pode usar a palavra-chave `color`, que aceita um argumento de string representando praticamente qualquer cor imaginável.

A cor pode ser especificada de várias maneiras

```
In [ ]: 1 plt.figure()
2 plt.plot(x, np.sin(x - 0), color='blue')           # specify color
3 plt.plot(x, np.sin(x - 1), color='g')             # short color code
4 plt.plot(x, np.sin(x - 2), color='0.75')          # Grayscale betw
5 plt.plot(x, np.sin(x - 3), color='#FFDD44')       # Hex code (RRGG
6 plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3))    # RGB tuple, va
7 plt.plot(x, np.sin(x - 5), color='chartreuse');    # all HTML colo
```

Se nenhuma cor for especificada, o **Matplotlib** percorrerá automaticamente um conjunto de cores padrão para várias linhas.

Da mesma forma, o estilo da linha pode ser ajustado usando a palavra-chave `linestyle`.

```
In [ ]: 1 plt.figure()
2 plt.plot(x, x + 0, linestyle='solid')
3 plt.plot(x, x + 1, linestyle='dashed')
4 plt.plot(x, x + 2, linestyle='dashdot')
5 plt.plot(x, x + 3, linestyle='dotted');
6
7 # For short, you can use the following codes:
8 plt.plot(x, x + 4, linestyle='-') # solid
9 plt.plot(x, x + 5, linestyle='--') # dashed
10 plt.plot(x, x + 6, linestyle='-.') # dashdot
11 plt.plot(x, x + 7, linestyle=':'); # dotted
```

Se você quiser ser extremamente conciso, esses códigos `linestyle` e `color` podem ser combinados em um único argumento sem palavra-chave para a função `plt.plot()`

```
In [ ]: 1 plt.figure()
        2 plt.plot(x, x + 0, '-g') # solid green
        3 plt.plot(x, x + 1, '--c') # dashed cyan
        4 plt.plot(x, x + 2, '-.k') # dashdot black
        5 plt.plot(x, x + 3, ':r'); # dotted red
```