#867 #DesenvolveDados Seg • Qua • Sex

Técnicas de Programação II

Conteúdo

1 Matplotlib

Matplotlib

Assim como o NumPy e o Pandas, o matplotlib é uma das bibliotecas que fazem parte do núcleo do SciPy 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. 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`, precisamos passar dois argumentos para essa função

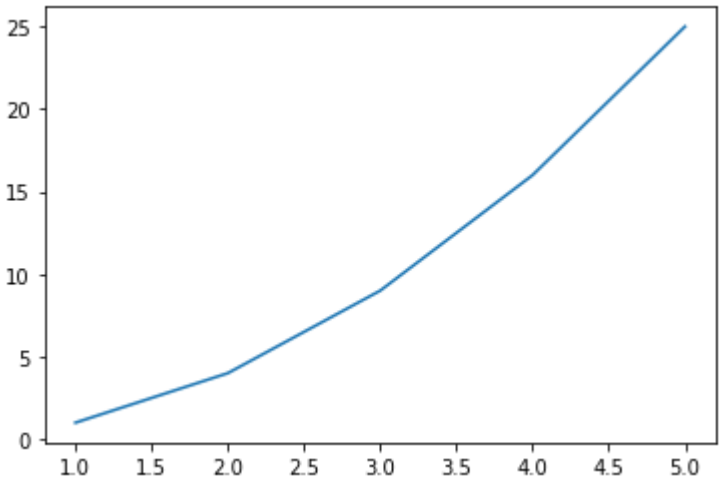
- x
- y

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

$$f(x) = x^2 \quad para \quad x \in [1, 5]$$

Múltiplos gráficos

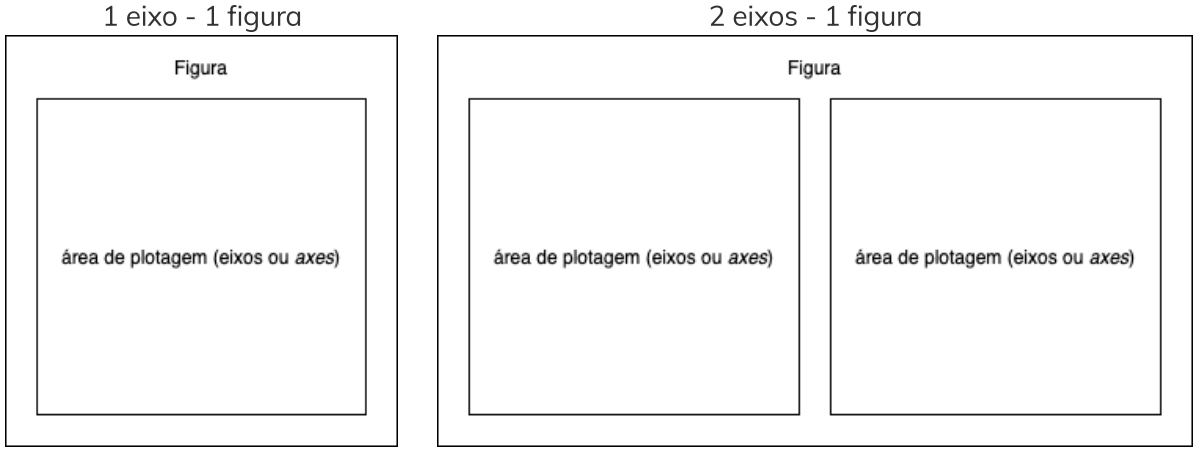
Estrutura

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.



Novos dados

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

$$f(x) = x^3 \quad para \quad x \in [1, 5]$$

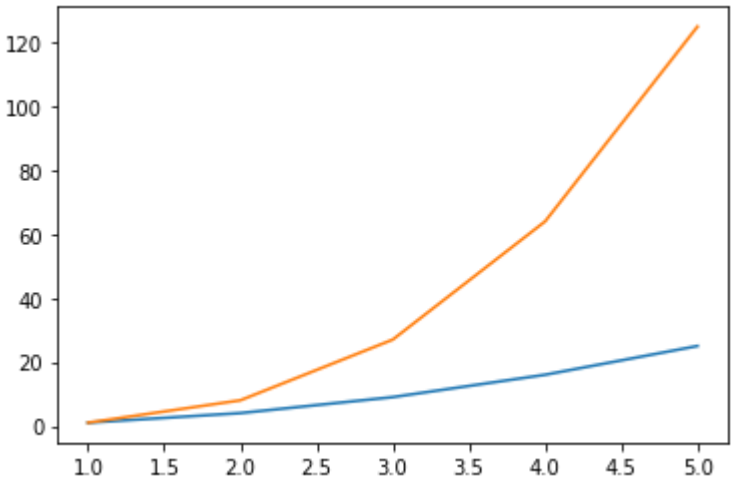
Para os próximos exemplos, iremos usar os dados:

```
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` 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.

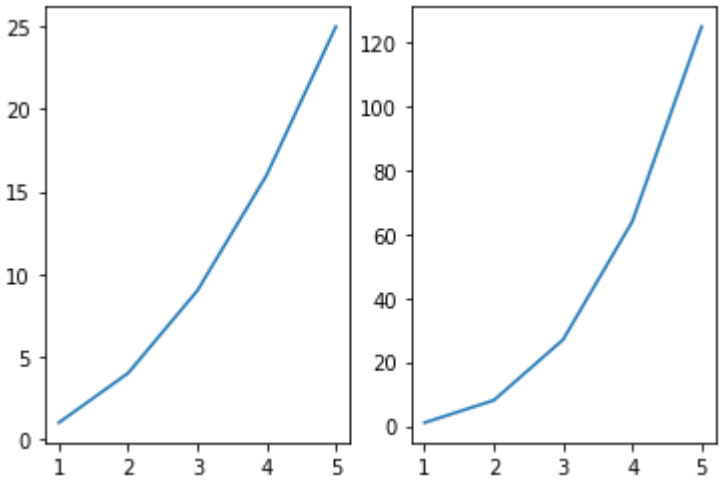
Ele irá retornar uma tupla de dois elementos:

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` 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.

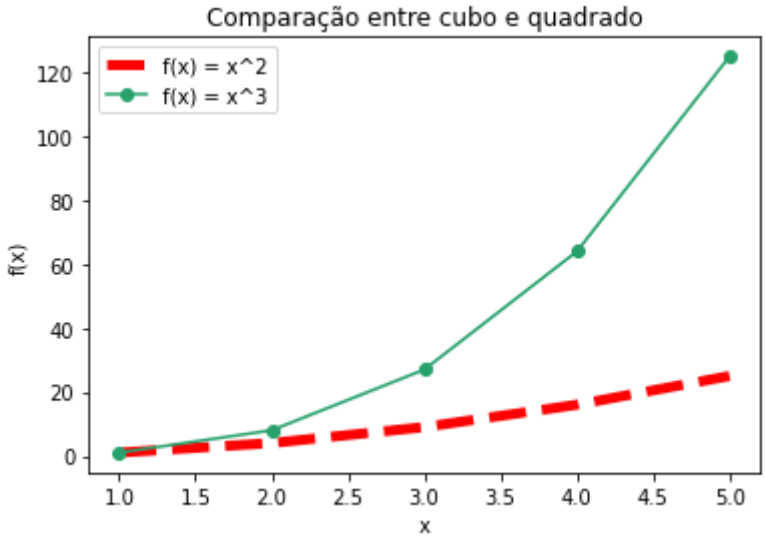
Vamos adicioná-las então!

```
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` e `ylabel` para definir um texto para cada um dos eixos.

Na linha 5 chamamos a função `title` para definir o título da figura.

Por último, chamamos a fução `legend` 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.

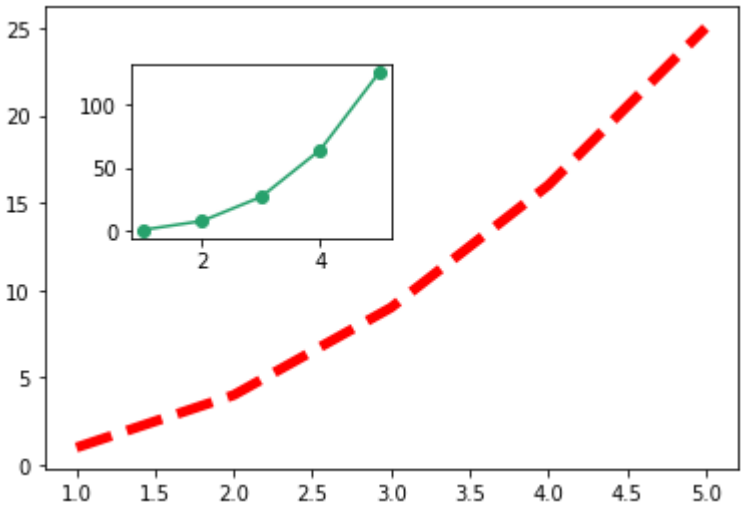
Pode parecer complicado a princípio, e você deve estar se perguntando: por que eu usaria essa API ao invés do funcional?

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()
```

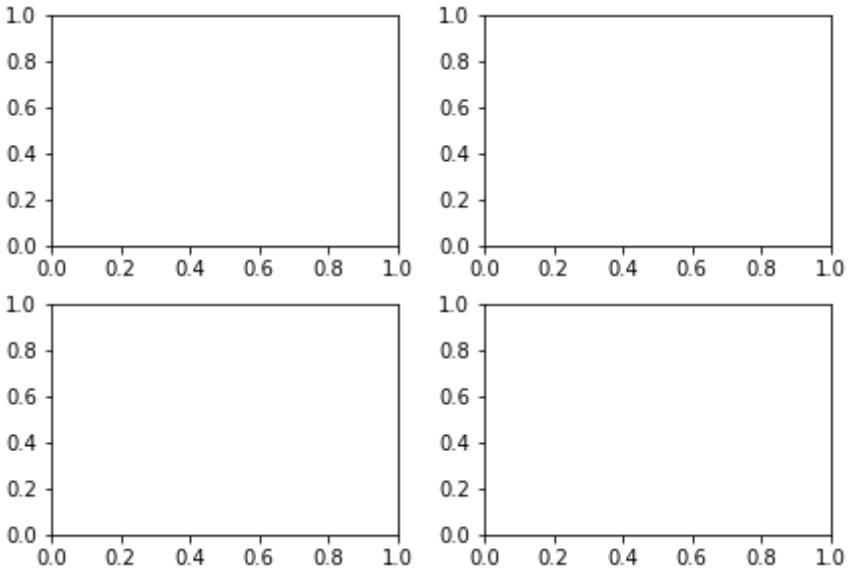


A função que vimos anteriormente, `plt.subplots`, abstrai essa complexidade para nós criando a grade de eixos para nós automaticamente. Caso contrário, teríamos que fazer manualmente algo como:

```
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*

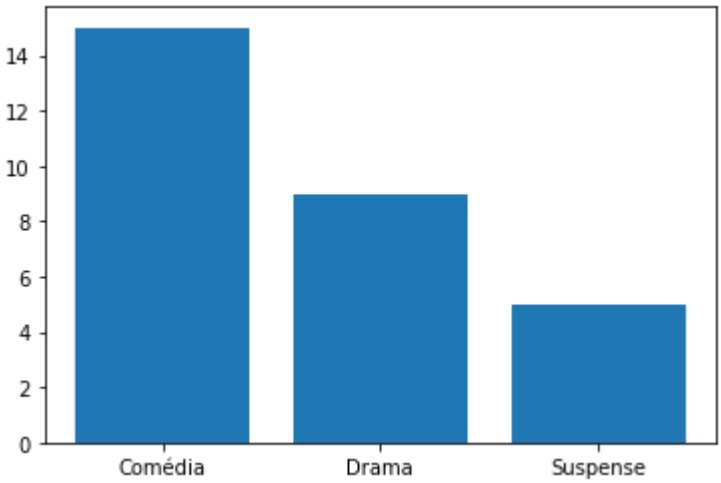
E muitos outros! Mostrarei alguns deles aqui, mas, recomendo fortemente visitar a galeria de exemplos da biblioteca para uma imersão das possibilidades.

Barra

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)
```



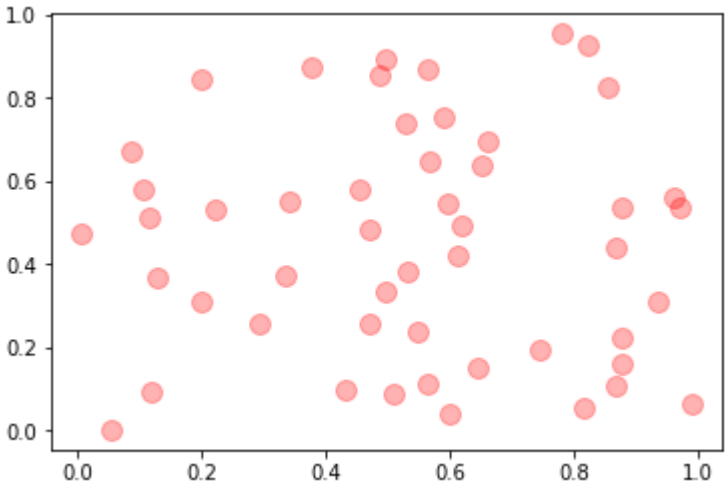
Dispersão

```
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:



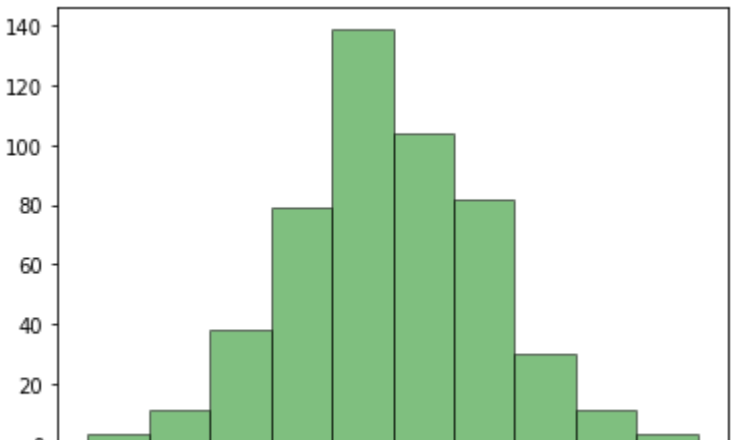
Histograma

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()
```



Próximo Tópico