

# **Programação II**

**+**

# **Estruturas de Dados para**

# **Bioinformática**

Hugo Pacheco

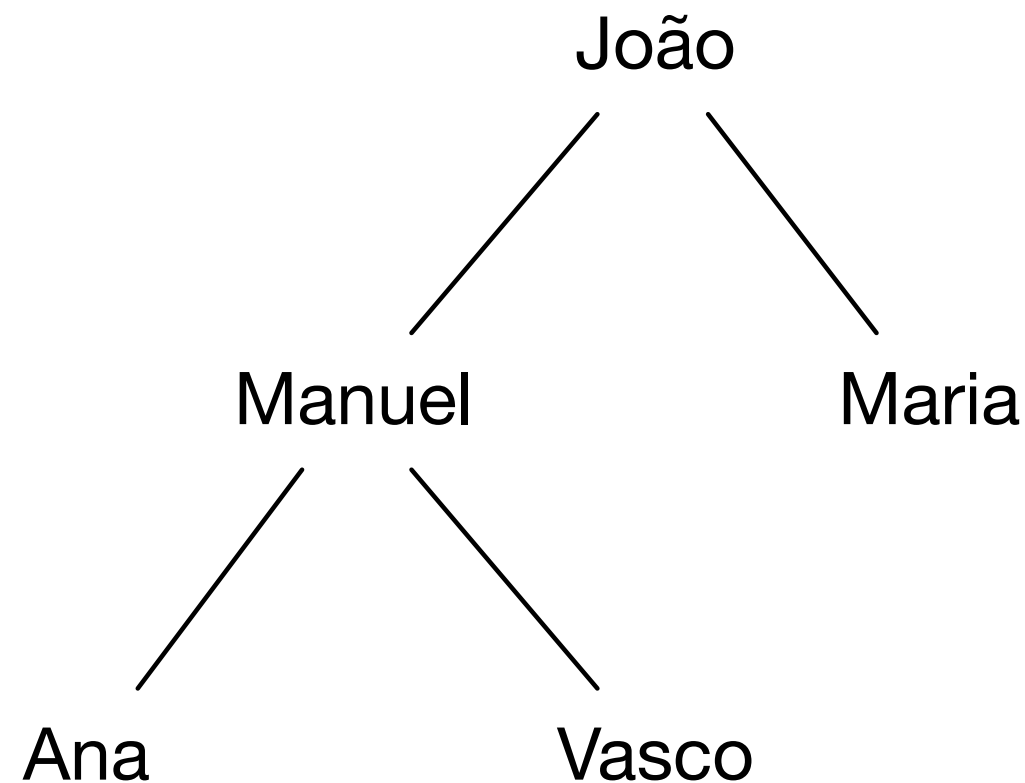
DCC/FCUP

23/24

# Análise de grafos (NetworkX)

# Árvores

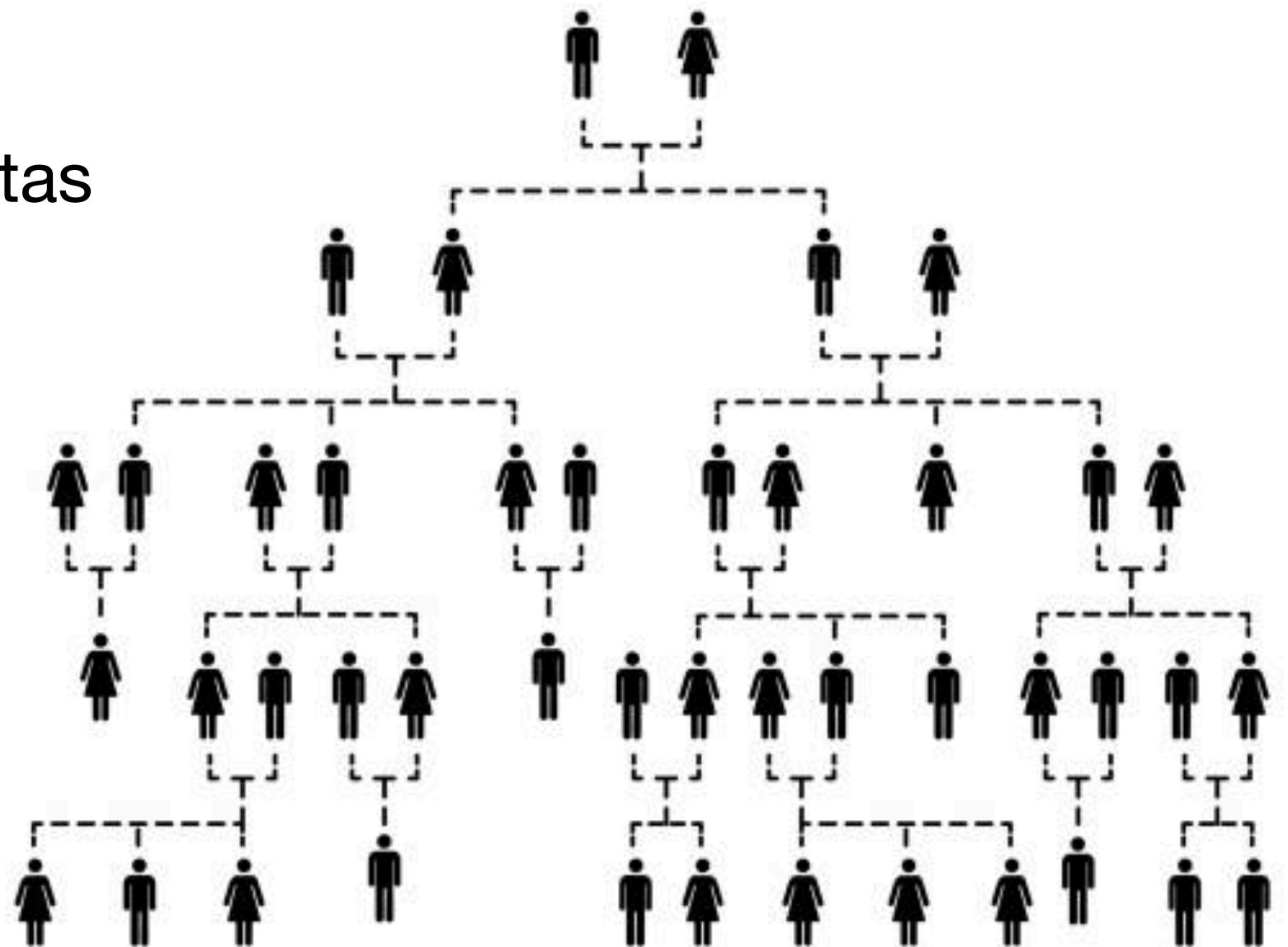
- Já vimos vários exemplos de dados hierárquicos / em árvore (e.g., JSON)
  - Pai guarda uma lista descendente de filhos, e assim consecutivamente



```
{  
  name: "João",  
  children: [{  
    nome: "Manuel",  
    children: [{  
      name: "Ana",  
      children: []  
    }, {  
      name: "Vasco",  
      children: []  
    }]  
  }, {  
    nome: "Maria",  
    children: []  
  }]  
}
```

# Árvores

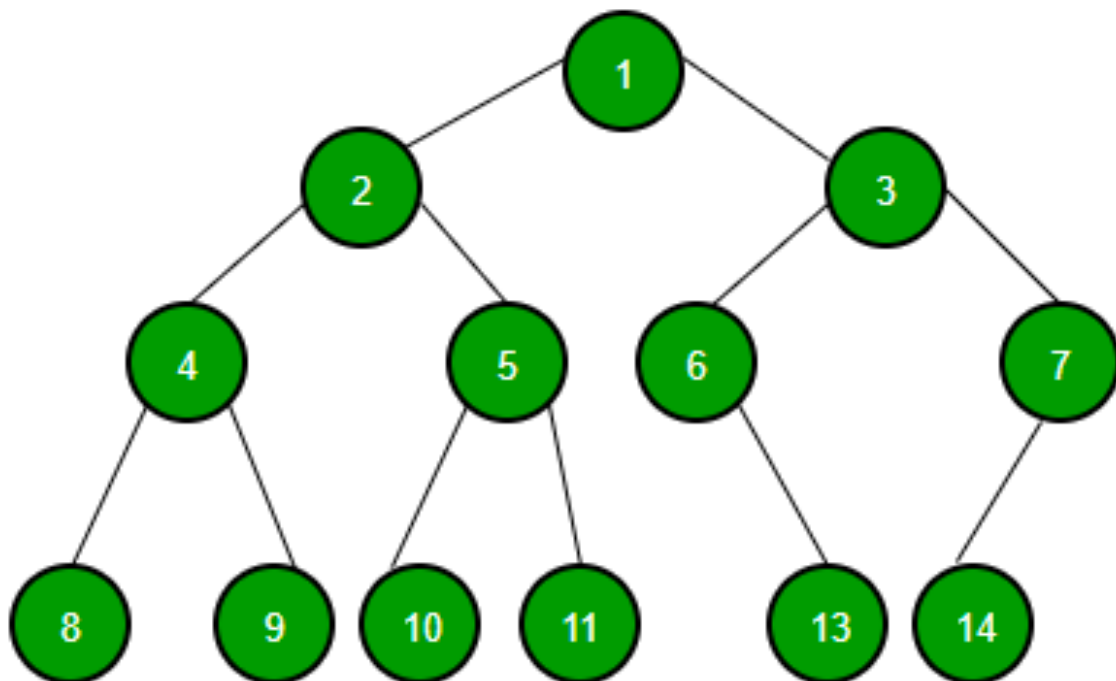
- E para árvores mais complicadas?
- Relações pai/mãe
- Padrastos/madrastas
- ...



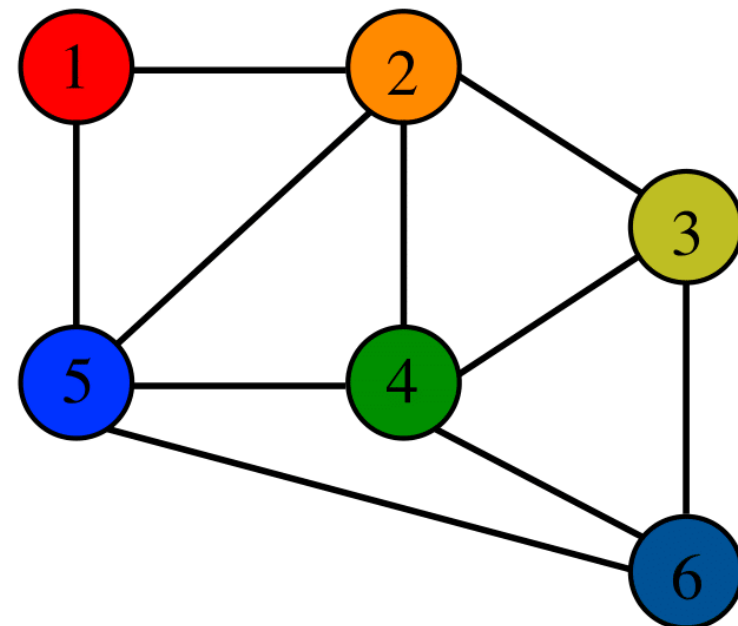
# Grafos

- Um *grafo* é uma coleção de *nodos* interligados
- Tipicamente usado para representar redes
- Uma generalização de árvores

**Árvore: hierarquia**

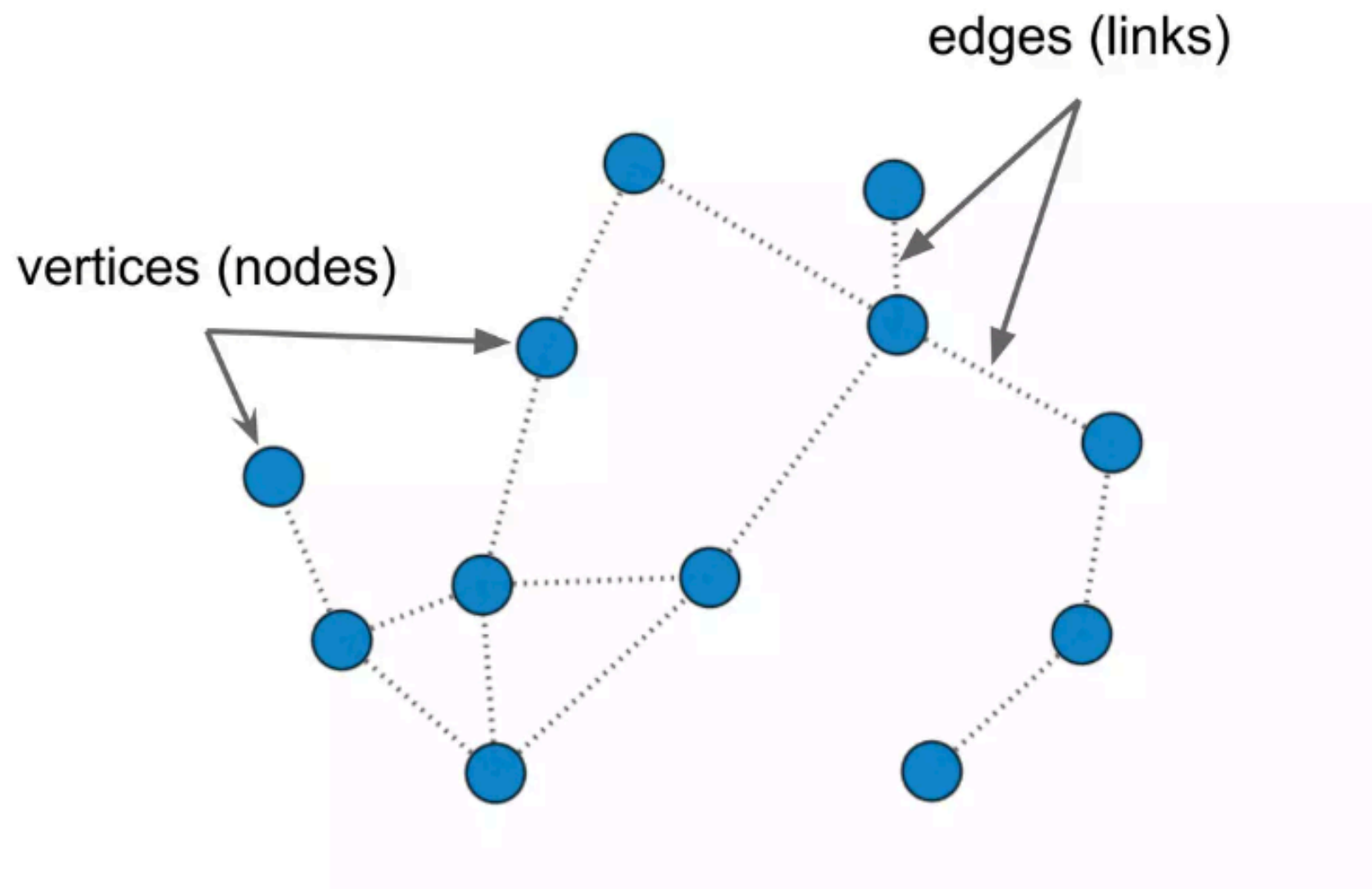


**Grafo: qualquer ligação**



# Grafos

- Um conjunto de vértices e um conjunto de arestas
  - Vértices têm identificadores únicos no grafo
  - Arestas ligam vértices



# Grafos (*NetworkX*)

- Podemos construir um grafo a partir de uma sequência de arestas: nodos implicitamente criados
- Por defeito grafo não-direcionado: arestas em ambos os sentidos

```
import networkx as nx

# lista de arestas
g1 = nx.Graph([(1,2), (2,3)])
# dicionário de adjacências
g2 = nx.Graph({1:[2,3], 2:[1,4]})

print(g1.nodes()) # [1, 2, 3]
print(g2.edges()) # [(1, 2), (1, 3), (2, 4)]
print(g2.has_edge(2,1)) # True
```



Nodos podem ser qualquer valor *hashable*.

**Intuição:** qualquer valor de um tipo imutável!

# Grafos (*NetworkX*)

- Grafos direcionados: arestas num sentido específico
- Uma ligação nos dois sentidos pode ser modelada explicitamente por duas arestas

```
import networkx as nx

# lista de arestas
g1 = nx.DiGraph([(1,2), (2,3)])
# dicionário de adjacências
g2 = nx.DiGraph({1:[2,3], 2:[1,4]})

print(g1.nodes()) # [1, 2, 3]
print(g2.edges()) # [(1, 2), (1, 3), (2, 4)]
print(g2.has_edge(2,1)) # True
print(g2.has_edge(3,1)) # False
```



# *NetworkX* (nodos)

- Podemos adicionar ou remover nodos de um grafo

```
import networkx as nx  
g = nx.Graph()
```

```
g.add_node('a')  
g.add_nodes_from([True, (4.5, 'c')])  
print(g.nodes()) # ['a', True, (4.5, 'c')]
```

```
g.remove_node('c') # fails, node does not exist  
g.remove_nodes_from(['a', (4.5, 'c')])  
print(g.nodes()) # [True]
```

# NetworkX (nodos)

- Nodos podem ter atributos associados (uma espécie de dicionário)

```
import networkx as nx
g = nx.Graph()
```

```
g.add_node(1, color="blue")
print(g.nodes()) # [1]
print(g.nodes(data=True)) # [(1, {'color': 'blue'})]
print(g.nodes()[1]) # {'color': 'blue'}
```

```
g.nodes()[1] = {'color': 'red'} # fails, not assignable
g.nodes()[1]['color'] = 'red'
print(g.nodes()[1]) # {'color': 'red'}
```

```
g.add_nodes_from([(4, {"color": "red"}), \
                  (5, {"color": "green"})])
print(g.nodes()) # [1, 4, 5]
print(g.nodes()[4]) # {'color': 'red'}
```

# NetworkX (arestas)

- Podemos adicionar ou remover arestas de um grafo

```
import networkx as nx
g = nx.Graph()
```

```
g.add_edge(1, 2)
print(g.nodes(), g.edges()) # [1, 2] [(1, 2)]
```

```
g.add_edges_from([(2, 3), (3, 4)])
print(g.nodes(), g.edges()) # [1, 2, 3, 4] [(1, 2), (2, 3), (3, 4)]
```

```
g.remove_node(2)
print(g.nodes(), g.edges()) # [1, 3, 4] [(3, 4)]
```

```
g.remove_edge(3, 4)
print(g.nodes(), g.edges()) # [1, 3, 4] []
```

# NetworkX (arestas)

- Arestas podem ter atributos associados (uma espécie de dicionário)

```
import networkx as nx
g = nx.Graph()
```

```
g.add_edge(1, 2, weight=4.7)
print(g.edges()) # [(1, 2)]
print(g.edges(data=True)) # [(1, 2, {'weight': 4.7})]
```

```
g.add_edges_from([(3, 4), (4, 5)], color='red')
print(g.edges()) # [(1, 2), (3, 4), (4, 5)]
print(g.edges()[3, 4]) # {'color': 'red'}
```

```
g.add_edges_from([(1, 2, {'color': 'blue'})], (1, 3, {'weight': 8}))
print(g.edges[1, 2]) # {'weight': 4.7, 'color': 'blue'}
```

```
g.edges[3, 4]['weight'] = 4.2
print(g.edges[3, 4]) # {'color': 'red', 'weight': 4.2}
```

# NetworkX (grau)

- O grau de cada nodo é o número de arestas ligadas a esse nodo
- Para grafos direcionados, temos grau de entrada e de saída

```
import networkx as nx
```

```
g1 = nx.Graph([('a', 'b'), ('a', 'c'), ('b', 'd'), ('d', 'a')])
```

```
print(g1.degree)
```

```
# [('a', 3), ('b', 2), ('c', 1), ('d', 2)]
```

```
print(g1.degree['b'])
```

```
# 2
```

```
g2 = nx.DiGraph([('a', 'b'), ('a', 'c'), ('b', 'd'), ('d', 'a')])
```

```
print(g2.in_degree)
```

```
# [('a', 1), ('b', 1), ('c', 1), ('d', 1)]
```

```
print(g2.out_degree)
```

```
# [('a', 2), ('b', 1), ('c', 0), ('d', 1)]
```

# NetworkX (adjacências)

- Um grafo é essencialmente uma lista de adjacências
- Podemos aceder às adjacências de um vértice



Grafos não direcionados: arestas duplicadas  
Grafos direcionados: apenas arestas de saída

```
g1 = nx.Graph([('a', 'b'), ('a', 'c'), ('b', 'd'), ('d', 'a')])
print(g1.adj) # {'a': {'b': {}, 'c': {}}, 'b': {'a': {}, 'd': {}}, 'c': {'a': {}}, 'd': {'b': {}, 'a': {}}}
print(g1.adj.items())
print(g1['a']) # {'b': {}, 'c': {}}
g1['b']['a']['color'] = 'red'
```

```
g2 = nx.DiGraph([('a', 'b'), ('a', 'c'), ('b', 'd'), ('d', 'a')])
print(g2.adj) # {'a': {'b': {}, 'c': {}}, 'b': {'d': {}}, 'd': {'a': {}}}
print(g2['a']) # {'b': {}, 'c': {}}
print(g2.succ) # igual a g2.adj
print(g2.pred) # arestas invertidas
```

# NetworkX (operações)

- Formalmente, grafos capturam relações binárias (um conjunto de arestas) entre o conjunto de nodos
- Podemos definir operações matemáticas sobre grafos

- Subgrafo

```
def is_subgraph(g1, g2):  
    return set(g1.nodes()).issubset(set(g2.nodes()))  
           and set(g1.edges()).issubset(set(g2.edges()))
```
- União

```
g12 = nx.compose(g1, g2) # união (sets de nodos arbitrários)  
g12 = nx.union(g1, g2)  # união disjunta (sets de nodos distintos)
```
- Interseção

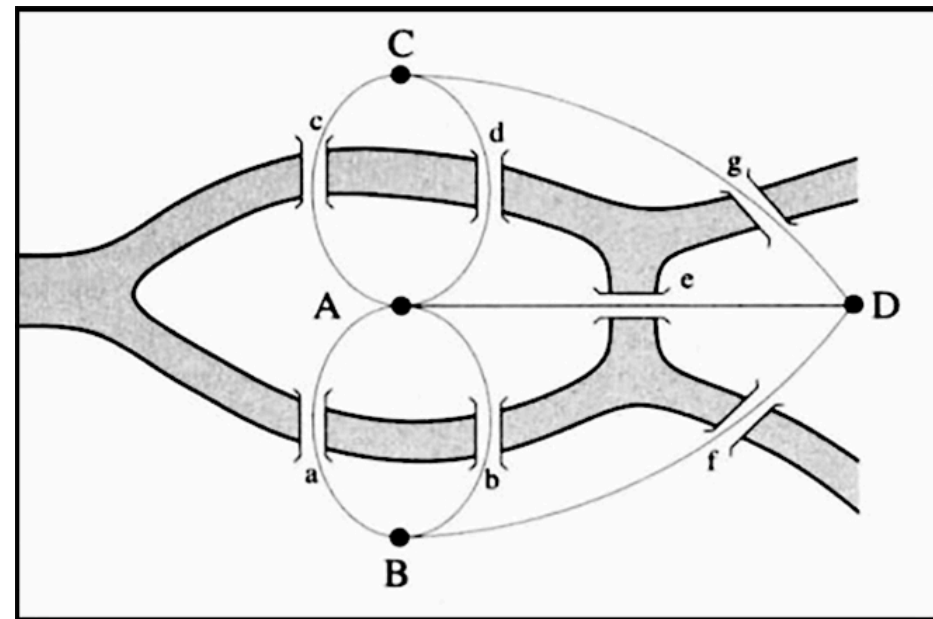
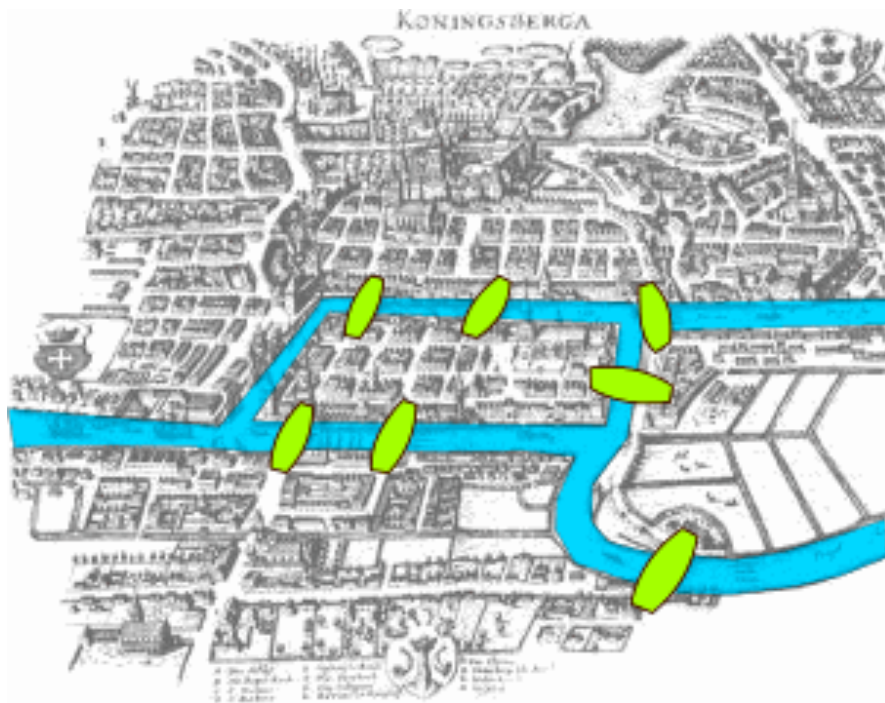
```
g12 = nx.intersection(g1, g2) # ignora atributos
```
- Diferença

```
g12 = nx.difference(g1, g2) # nodos têm que ser os mesmos; ignora atributos
```

# Teoria de Grafos

- Muitos problemas computacionais são descritos e resolvidos com grafos
- Exemplo clássico: 7 pontes de Königsberg

*“Existe uma travessia que passe por todas as pontes exatamente uma vez?”*



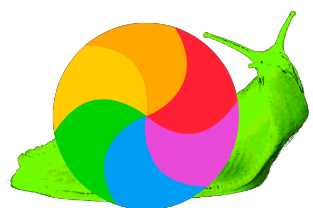
Solução matemática: se e só se existem no máximo dois vértices com grau ímpar



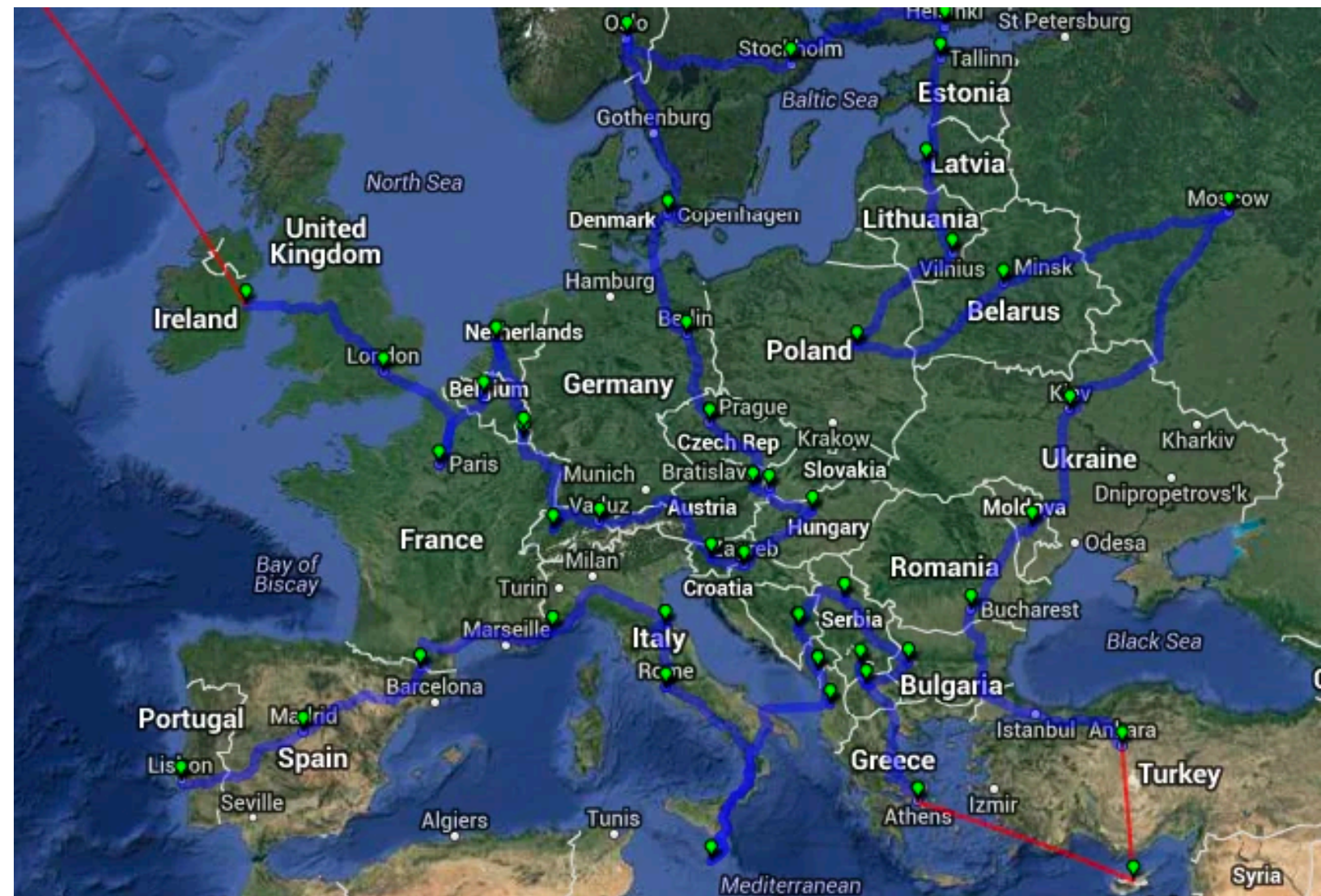
# Teoria de Grafos

- Muitos problemas computacionais são descritos e resolvidos com grafos
- Exemplo clássico: caixeiro viajante

*“Qual o itinerário mais curto que passa por todas as cidades exatamente uma vez?”*



Problema complexo, soluções aproximadas, e.g., Capitais Europeias

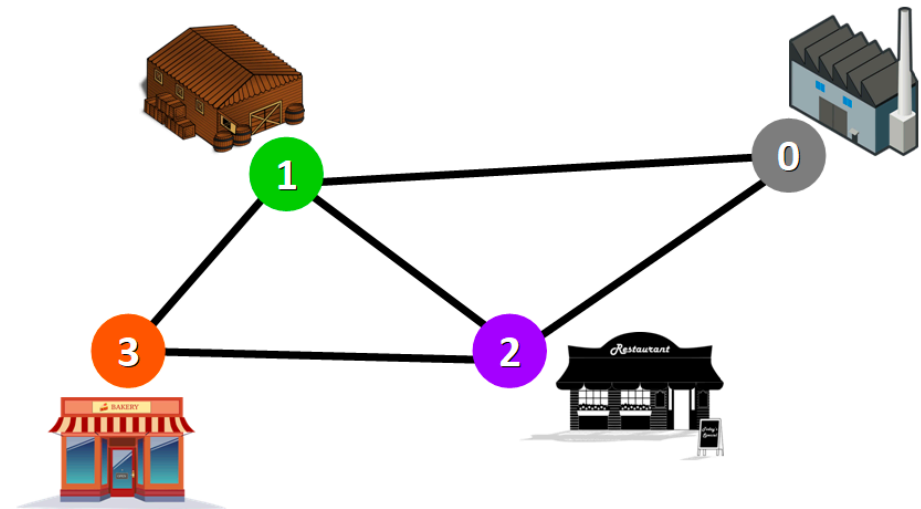


# NetworkX (algoritmos)

- Implementar algoritmos de grafos é complexo
  - Solução básica é ineficiente, e mesmo assim não trivial



*Sabem codificar este grafo em Python e implementar uma função que decide se existe caminho entre dois pontos?*



- Muitas otimizações heurísticas na prática
- A biblioteca NetworkX oferece implementações de vários algoritmos de grafos: [documentação](#)

# NetworkX (caminhos)

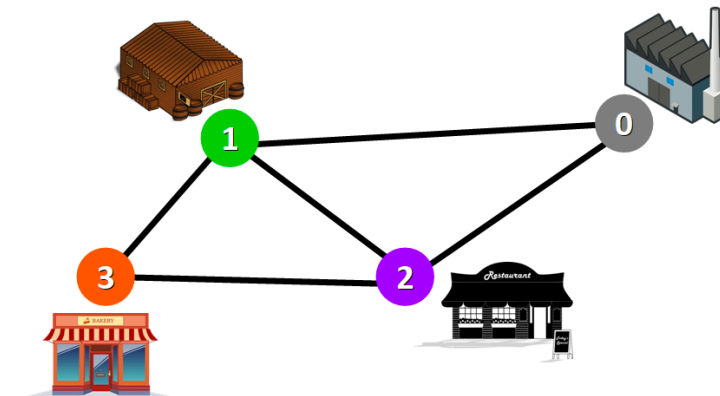
- Pergunta mais frequente em grafos: existe caminho entre dois nodos?
- Um caminho é uma ligação direta ou indireta entre nodos

```
g = nx.Graph()
g.add_nodes_from([(0, {'name': 'factory'})], (1, {'name': 'farm'}), (2,
{'name': 'restaurant'}), (3, {'name': 'bakery'})])
g.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)])
gns = g.nodes(data=True) # [(0, {'name': 'factory'}), ...]
```

```
def findName(name):
    return [id for id, att in gns if att['name'] == name][0]
```

```
bakery = findName('bakery') # 3
factory = findName('factory') # 0
```

```
print(g.has_edge(bakery, factory)) # False
print(nx.has_path(g, bakery, factory)) # True
```



# Exemplo (Game of Thrones)

- Criar um grafo de relações entre personagens de Game of Thrones (dados no formato JSON encontrados [aqui](#))

```
with open('got.json', 'r') as f:
    characters = json.load(f)["characters"]

family_relations = {'parentOf', 'parents', 'siblings', 'sibling', 'marriedEngaged'}
other_relations = {'killedBy', 'killed', 'abductedBy', 'abducted', 'allies',
                  'guardianOf', 'serves', 'servedBy'}
relations = family_relations | other_relations
g = nx.DiGraph()

for character in characters:
    me = character["characterName"]
    g.add_nodes_from([(me, character)])
    for relation in relations:
        if relation in character:
            for other in character[relation]:
                if g.has_edge(me, other): g.edges[me, other]
["kinds"].add(relation)
                else: g.add_edge(me, other, kinds={relation})
```



# Exemplo (Game of Thrones)

- Caminho mais curto entre duas personagens
- Subentende uma noção de distância entre nodos (atributo *weight* em cada aresta)
  - Por defeito todas as arestas têm distância 1
  - conceito generalizável (e.g., arestas são estradas e *weight* = distância em kms)

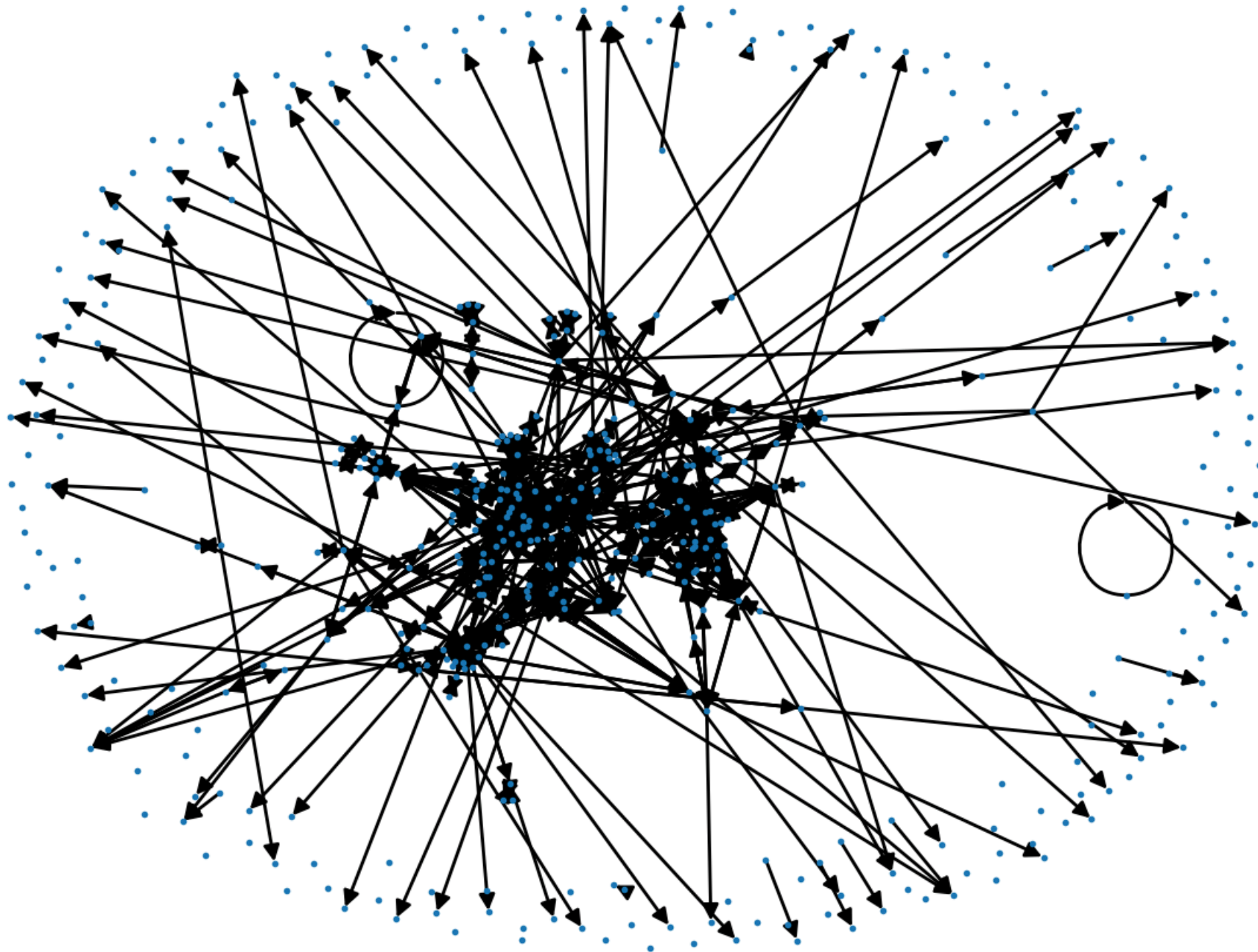
```
def mkWeight(d):  
    return 1 if d["kinds"] & family_relations else 2
```

```
for x,y,d in g.edges(data=True):  
    g.edges[x,y]["weight"] = mkWeight(d)
```

```
p = nx.shortest_path(g, "Margaery Tyrell", "Jon Snow", weight="weight")  
print(p)  
gp = nx.subgraph(g,p)  
print(gp.edges(data=True))
```

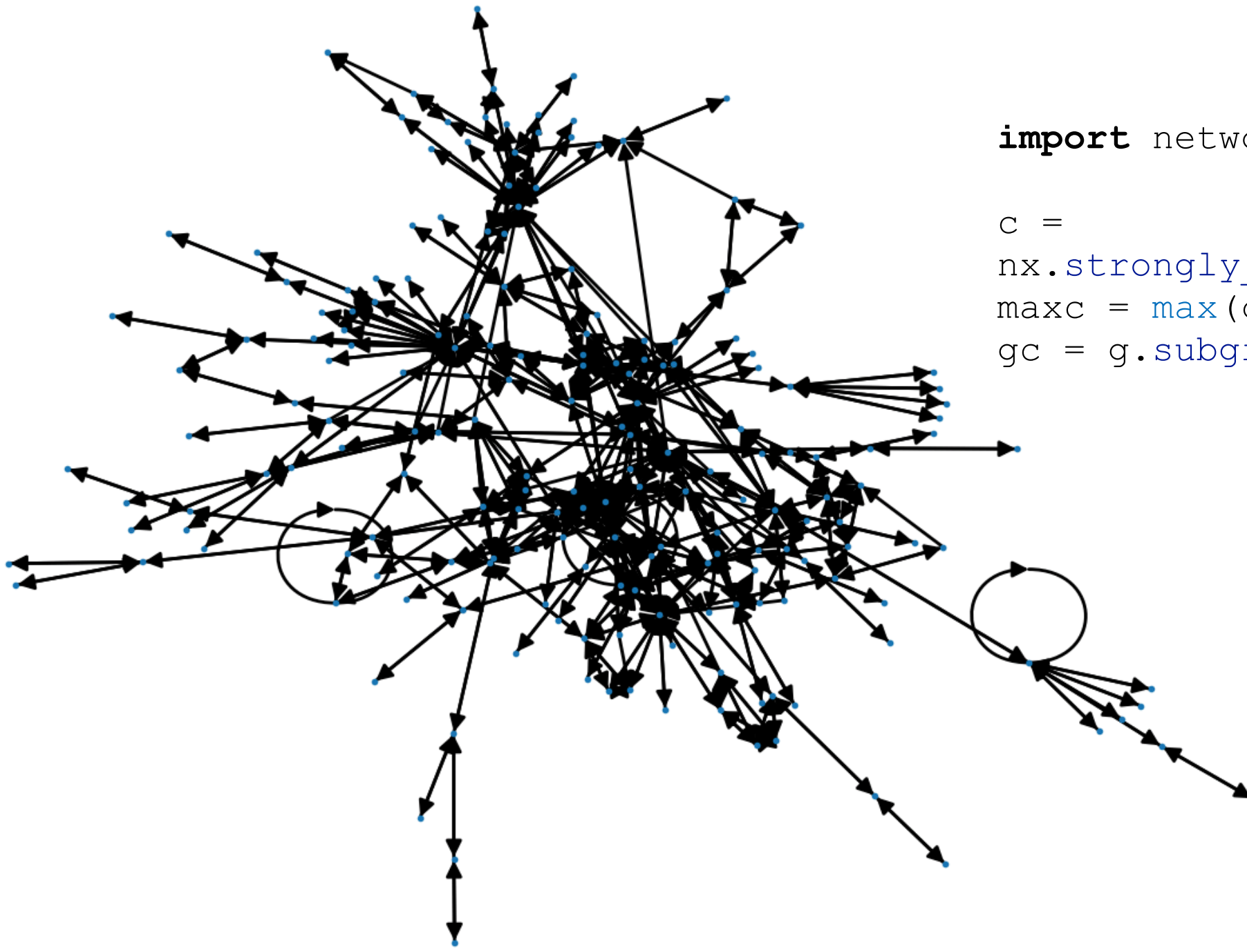
# Exemplo (Game of Thrones)

- Desenhar o grafo (vamos ver mais em detalhe para a frente)



# Exemplo (Game of Thrones)

- Selecionar o maior componente (fortemente) conexo



```
import networkx as nx
```

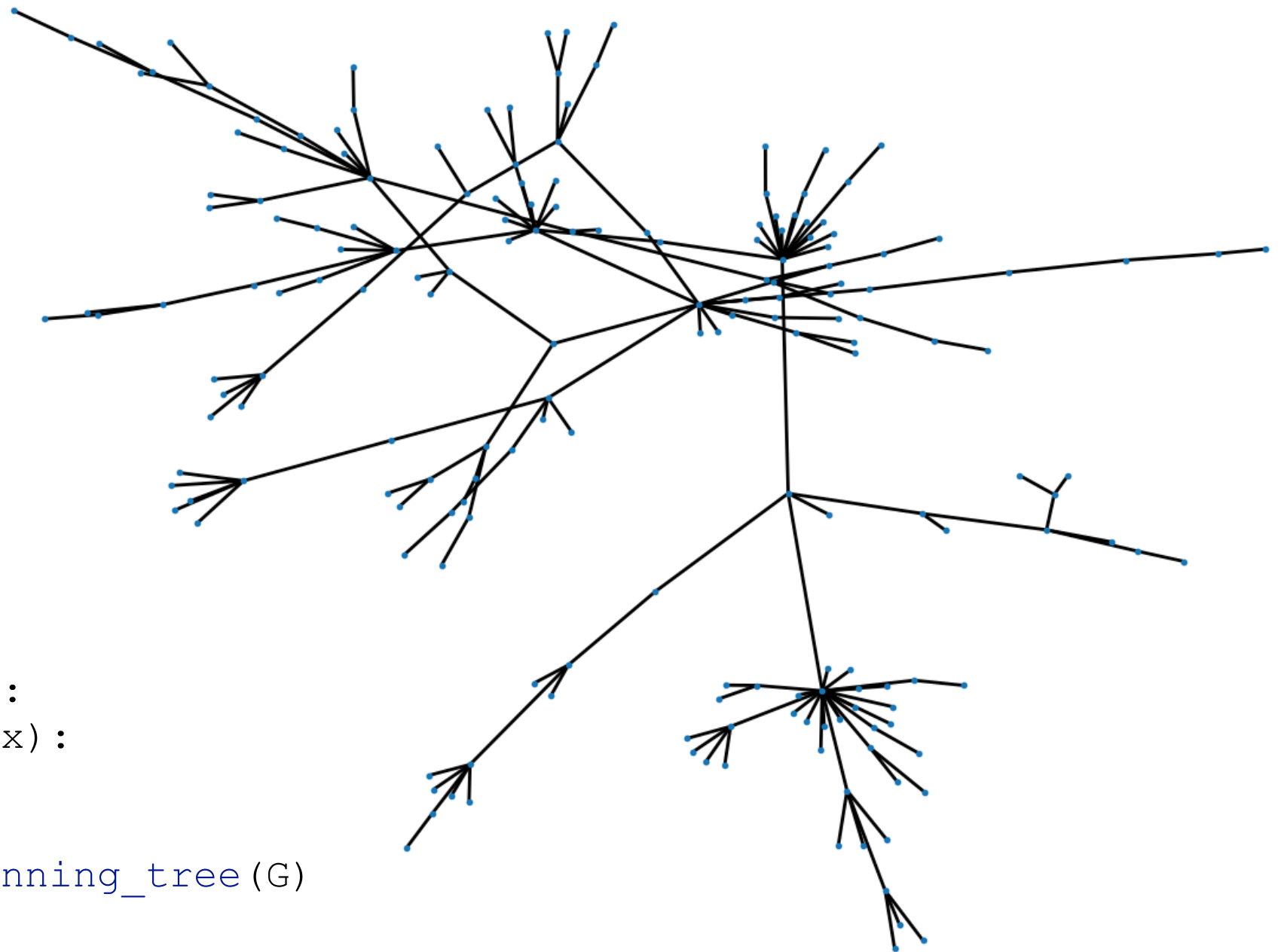
```
c =  
nx.strongly_connected_components(g)  
maxc = max(c, key=len)  
gc = g.subgraph(maxc)
```

# Exemplo (Game of Thrones)

- Simplificar calculando a *minimum spanning tree*

```
G = nx.Graph()
for x,y in gc.edges():
    if gc.has_edge(y,x):
        G.add_edge(x,y)

minG = nx.minimum_spanning_tree(G)
```





# Exemplo (Game of Thrones)

- Encontrar uma das possíveis árvores de família, calculando:

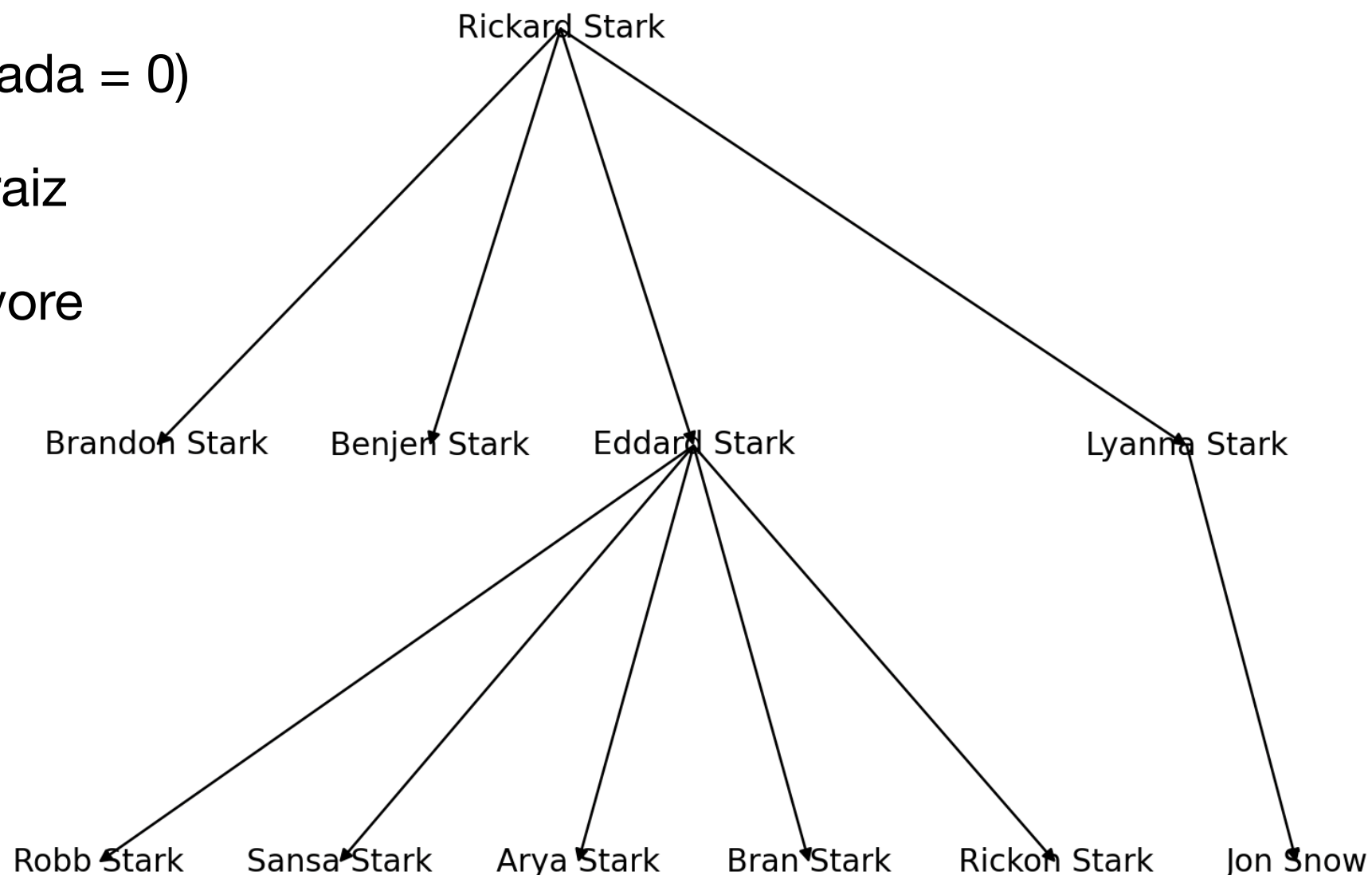
1. Sub-grafo de relações de parentalidade

2. Maior componente (fracamente) conexo

3. Raízes (grau de entrada = 0)

4. Árvore desde cada raiz

5. Retornar a maior árvore



# Exemplo (Game of Thrones)

- Determinar influência de cada nodo?
- Há vários algoritmos, por exemplo *PageRank* do Google

```
ranks = nx.pagerank(g)
sranks = sorted(ranks.items(), key=lambda x: x[1])
top5 = sranks[-5:]
for r in top5:
    print(r)
# ('Cersei Lannister', 0.012929139680036124)
# ('Eddard Stark', 0.017231486555328624)
# ('Arya Stark', 0.018049491729394615)
# ('Daenerys Targaryen', 0.025910182564731432)
# ('Jon Snow', 0.026336320077632543)
```

# Gráficos (matplotlib)

# “Uma imagem vale mil palavras”

- Enormes quantidades de informação em programação e, em particular, na análise computacional de dados
- Os humanos foram feitos para processar dados visuais, não textuais
- Gráficos, e outras formas de visualização, são essenciais para nos ajudar a compreender os dados
  - Ajudam a identificar padrões, tendências e correlações
  - São um veículo de eleição para partilhar informação
  - Permitem fornecer informação de variadas formas e com diferentes níveis de detalhe

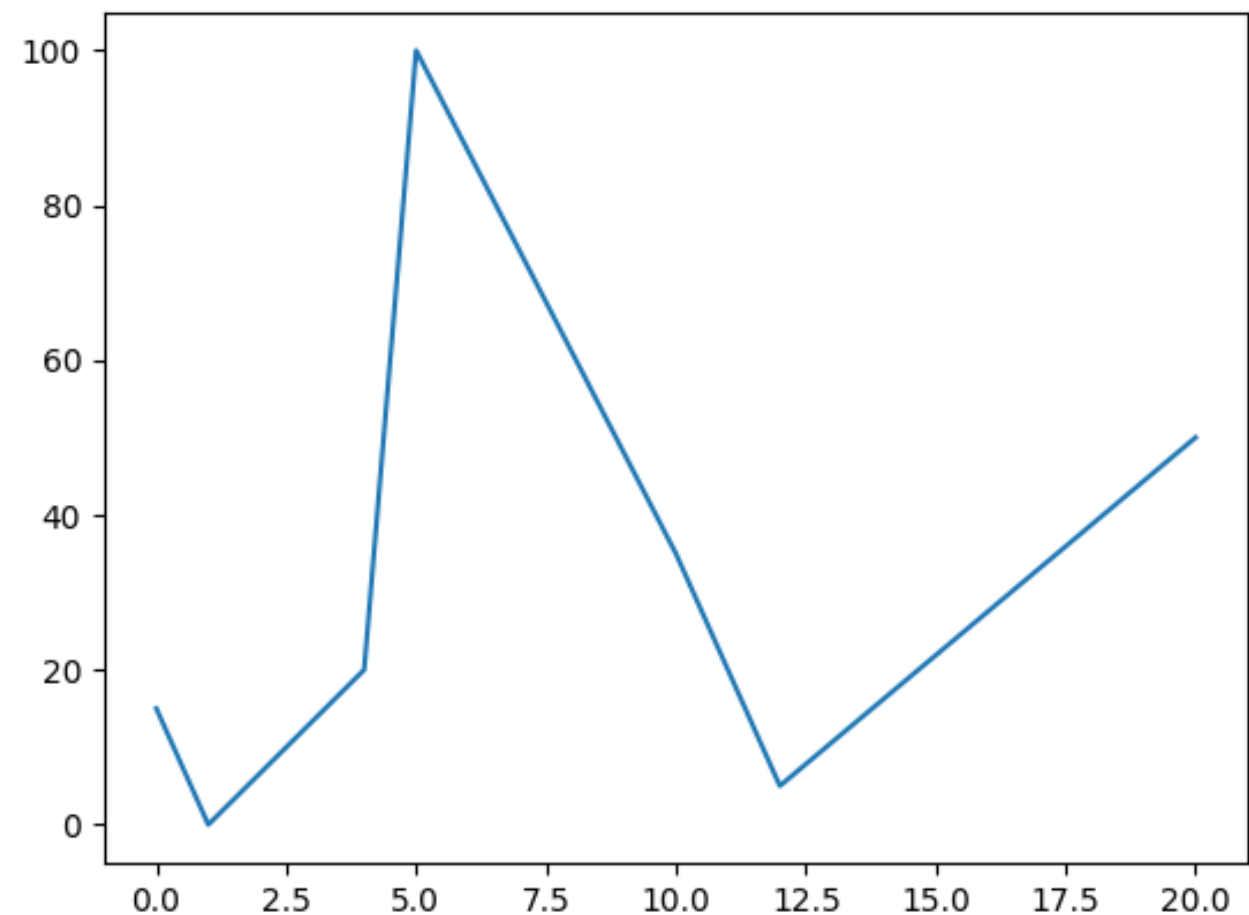
# *Gráficos*

- Vamos aprender a desenhar com a biblioteca *matplotlib*, que é muito utilizada para gerar gráficos científicos para grandes conjuntos de dados
  - Gráficos 2D e 3D (vamos olhar apenas para 2D)
  - Diferentes formatos de output (PNG, PDF, GUI, etc)
  - Suporte para gráficos interativos
  - Suporte para mapas geográficos
  - Suporte para grafos

# *Matplotlib*

- Gráficos 2D são definidos por 2 sequências:
  - Sequência de valores no eixo dos X
  - Sequência de valores no eixo dos Y para cada X

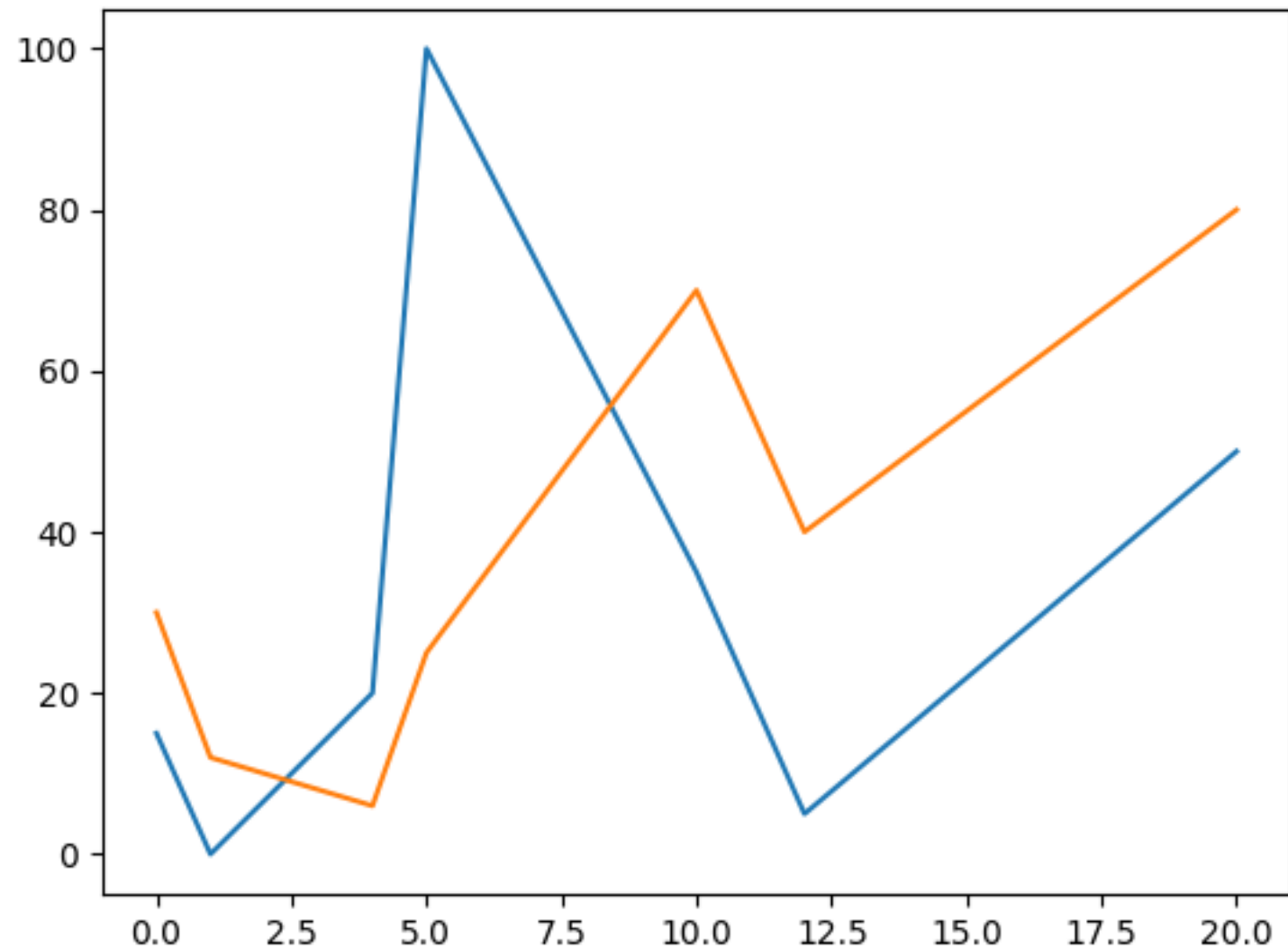
```
import matplotlib.pyplot as plt
x = [0, 1, 4, 5, 10, 12, 20]
y = [15, 0, 20, 100, 35, 5, 50]
plt.plot(x, y)
plt.show()
```



# Matplotlib

- Podemos desenhar mais do que uma curva nos Y, para os mesmos X

```
import matplotlib.pyplot as plt
x = [0, 1, 4, 5, 10, 12, 20]
y1 = [15, 0, 20, 100, 35, 5, 50]
y2 = [30, 12, 6, 25, 70, 40, 80]
plt.plot(x, y1)
plt.plot(x, y2)
plt.show()
```




# *Matplotlib* (guardar)

- Podemos guardar um gráfico para um ficheiro
- Podemos escolher extensão (PNG, JPG, PDF, etc)

```
x = [0, 1, 4, 5, 10, 12, 20]
y = [15, 0, 20, 100, 35, 5, 50]
plt.plot(x, y)
# abre uma GUI
# plt.show()
# guarda para ficheiro
plt.savefig('grafico.png')
```



# *Matplotlib* = magia?

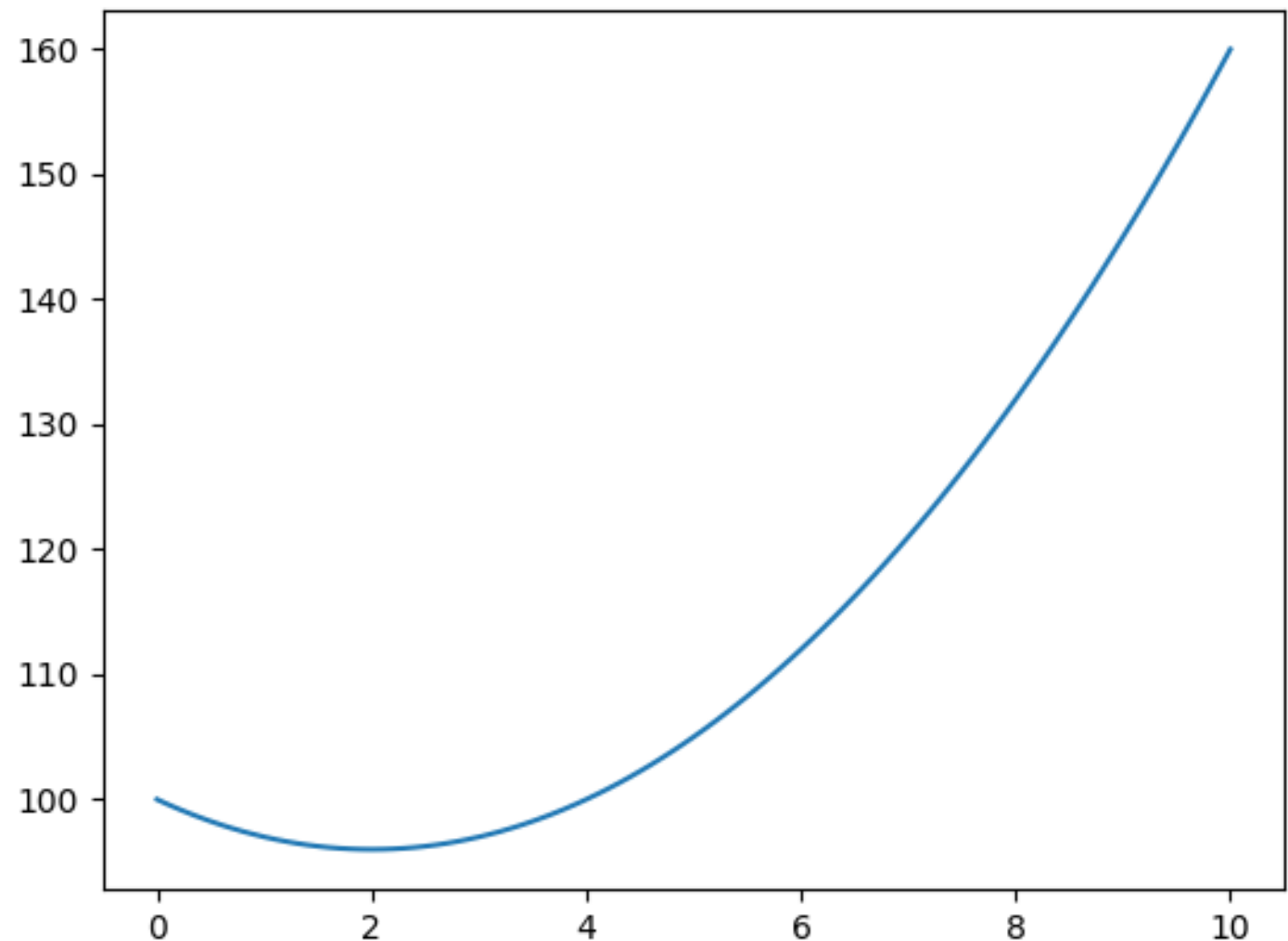
- Como é que a função *show()* sabe o que desenhar?
- **Cuidado:** o módulo tem estado global interno.... 
- Regra: estado “limpa” depois de cada *show()*
- Pode haver algum comportamento imprevisível com múltiplos *show()*

```
import matplotlib.pyplot as plt
x = [0, 1, 4, 5, 10, 12, 20]
y1 = [15, 0, 20, 100, 35, 5, 50]
y2 = [30, 12, 6, 25, 70, 40, 80]
plt.plot(x, y1)
plt.show()
plt.plot(x, y2)
plt.show()
```

# Matplotlib + NumPy

- Podemos utilizar vetores *numpy* como sequências

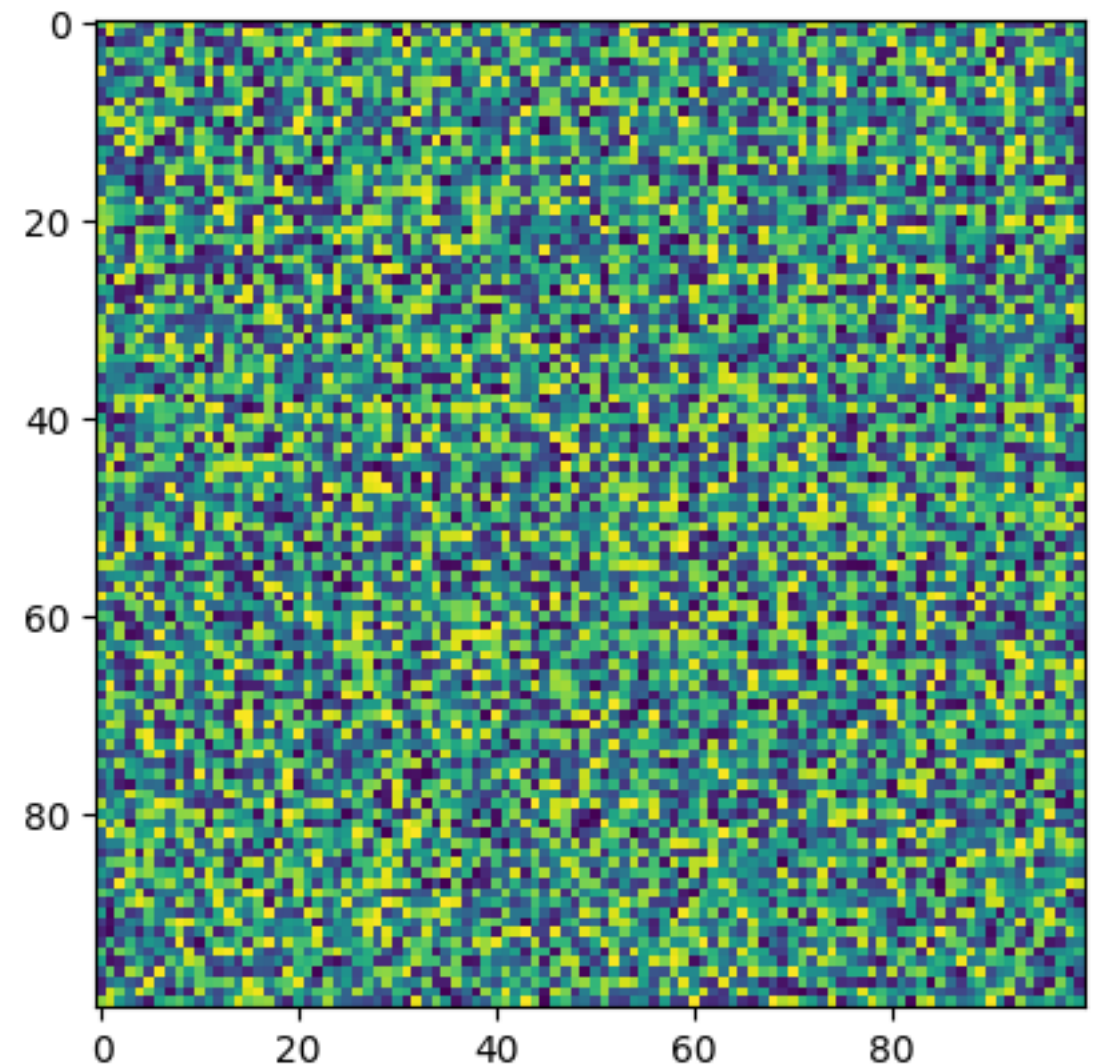
```
import numpy as np
import matplotlib.pyplot
as plt
# 101 values, equally
# spaced in [0,10]
x = np.linspace(0,10,101)
# y as a function of x
y = 100 - 4*x + x**2
plt.plot(x,y)
plt.show()
```



# *Matplotlib + NumPy*

- Podemos visualizar matrizes *numpy* como imagens
  - Mapa de cores escolhido automaticamente

```
import numpy as np
import matplotlib.pyplot as plt
x = np.random.rand(100,100)
plt.imshow(x)
plt.show()
```



# Matplotlib + NumPy

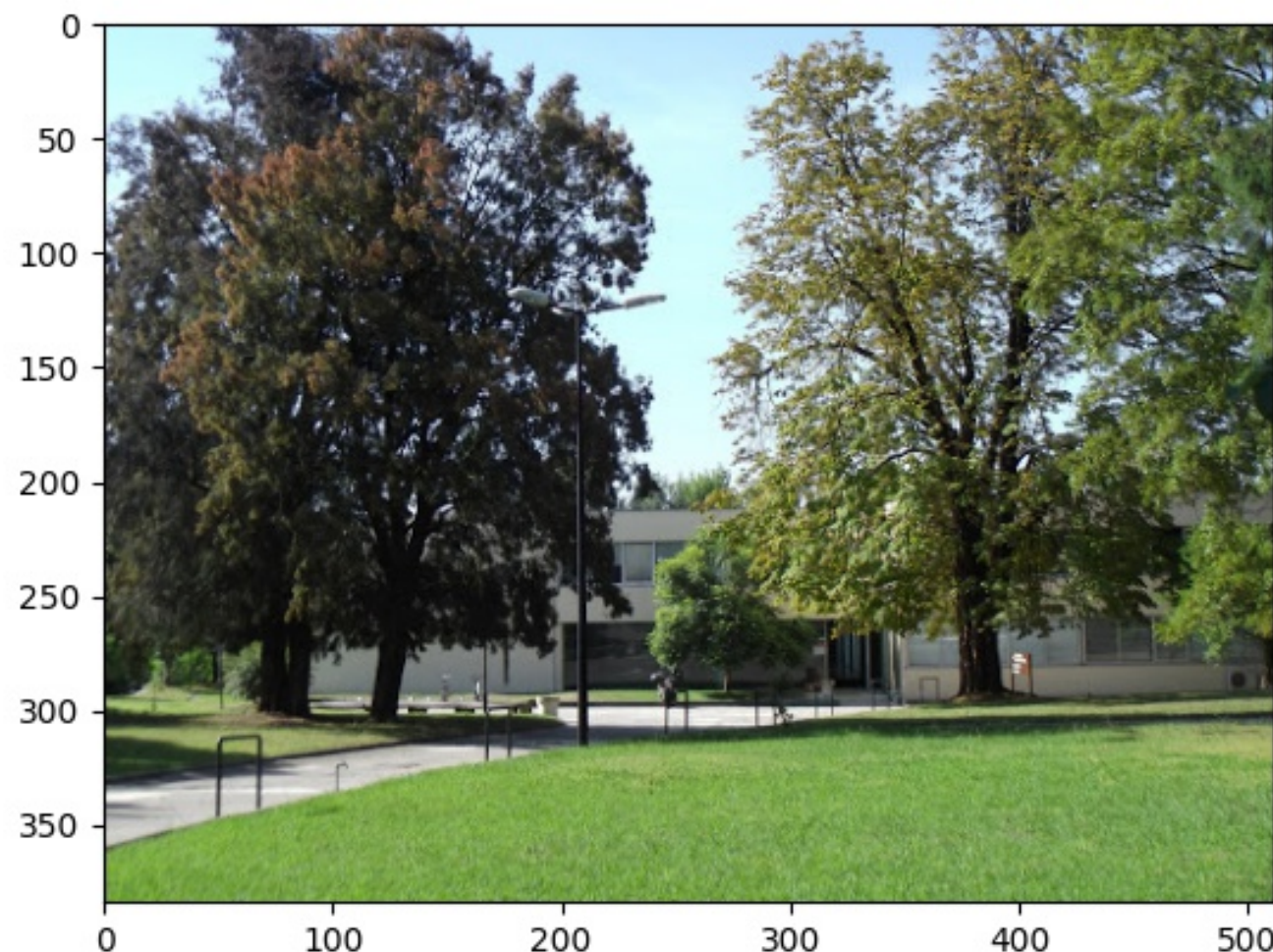
- Podemos ler/escrever imagens para/de matrizes *numpy*, analogamente ao *Pillow*
- Podemos transformar a imagem manipulando a matriz, como vimos antes



**Nota:** a imagem resultante é o gráfico completo: tem margens, eixos, etc

```
dcc=plt.imread("dcc.jpg")
print(dcc.shape)  #(384, 512, 3)
x,y,z = dcc.shape

plt.imshow(dcc)
plt.show()
```





# *Matplotlib + NumPy*

- Podemos alterar a visualização da imagem mudando o mapa de cores

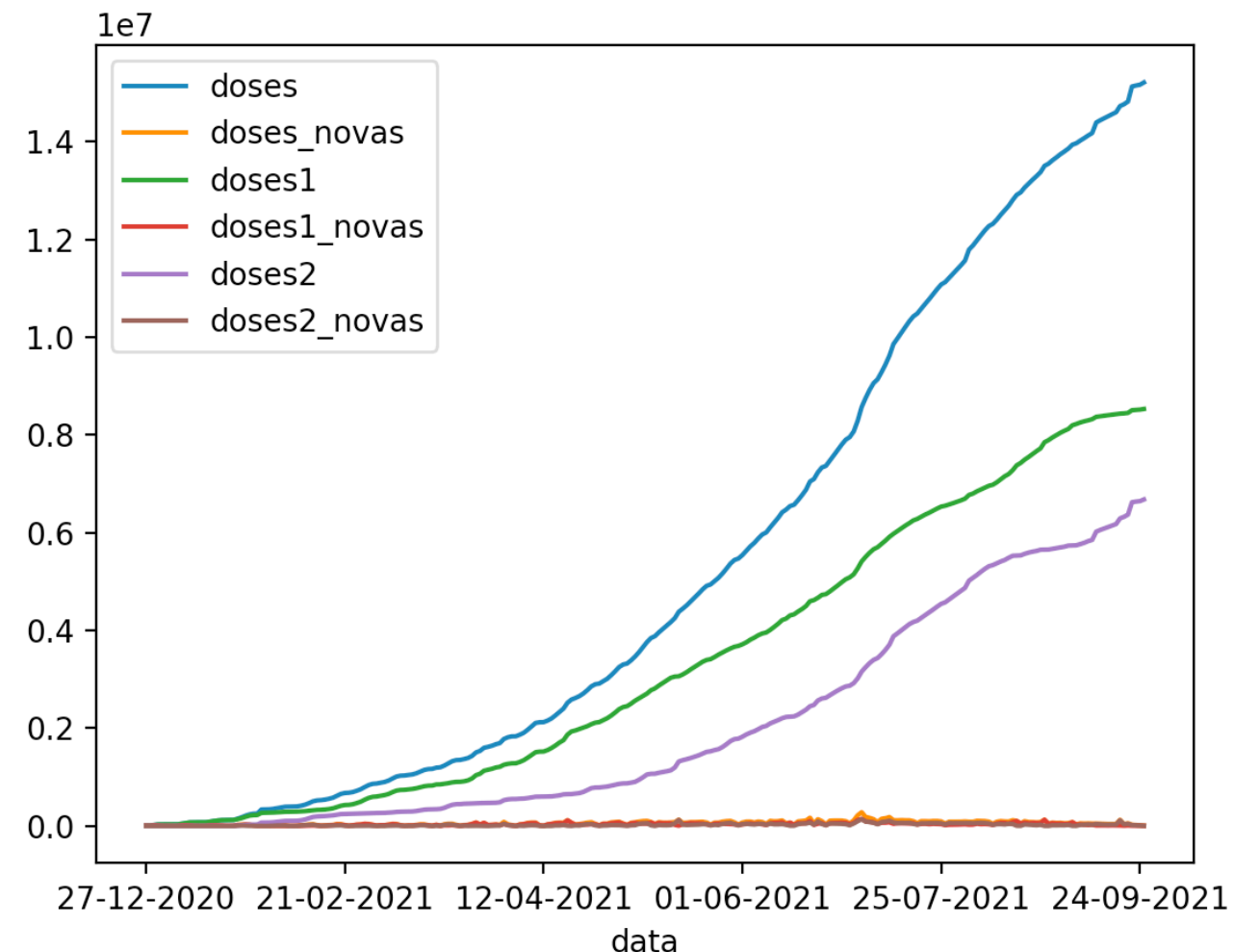
```
dcc=plt.imread("dcc.jpg")  
dcc_gray = dcc.mean(axis=2)  
plt.imshow(dcc_gray, cmap='gray')  
plt.axis('off')  
plt.tightlayout()  
plt.savefig('gray.png')
```



# Matplotlib + Pandas

- Podemos visualizar facilmente um *DataFrame*
  - Índices definem o eixo dos X
  - Cada coluna é uma curva no eixo dos Y
- E.g., dados de vacinação contra a COVID-19 publicados pela DGS e disponíveis [aqui](#)

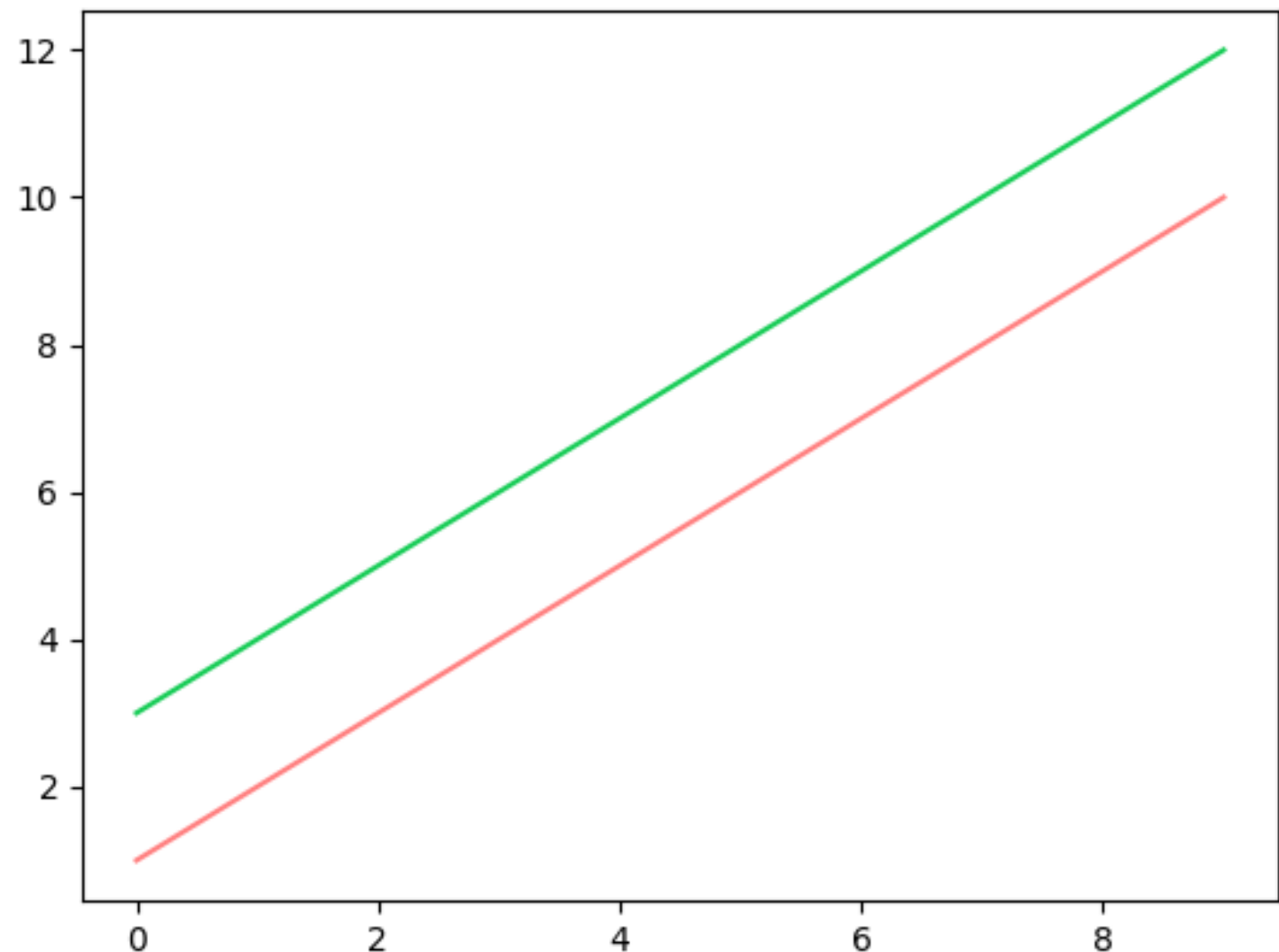
```
vacinas =  
pd.read_csv('vacinas.csv', in  
dex_col='data')  
vacinas = vacinas[[col for  
col in vacinas.columns if  
col.startswith('doses')]]  
vacinas = vacinas.dropna()  
  
vacinas.plot()  
plt.show()
```



# Matplotlib (cores)

- Podemos controlar as cores de cada curva
- Lista de nomes de cores ou cores em hexadecimal

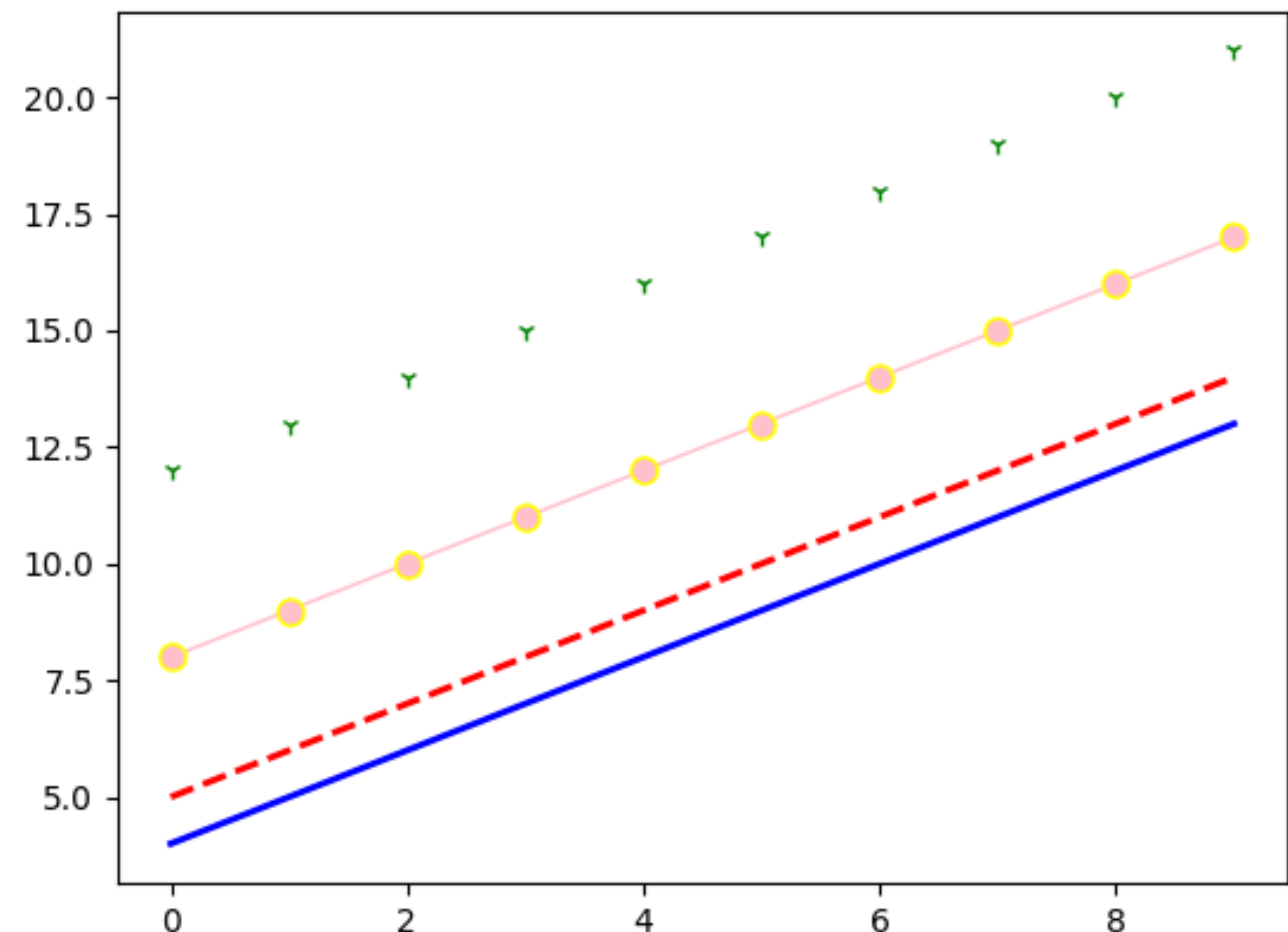
```
x = np.arange(10)
# half-transparent red
plt.plot(x, x+1, color="red",
alpha=0.5)
# RGB hex code for green
plt.plot(x, x+3, color="#15cc55")
plt.show()
```



# Matplotlib (estilo)

- Podemos controlar a grossura, estilo e marcadores de cada linha
  - Estilos de linha: '-', '—', '-.', ':', 'steps', 'none'
  - Marcadores: '+', 'o', '\*', 's', ',', '.', '1', '2', '3', '4', ...

```
x = np.arange(10)
plt.plot(x, x+4, c="blue",
linewidth=2.00)
plt.plot(x, x+5, c="red", lw=2,
linestyle='--')
plt.plot(x, x+8, c="pink", lw=1,
ls='-', marker='o',
markersize=8, markeredgecolor="yellow")
plt.plot(x, x+12, c="green",
ls='none', marker='1')
plt.show()
```

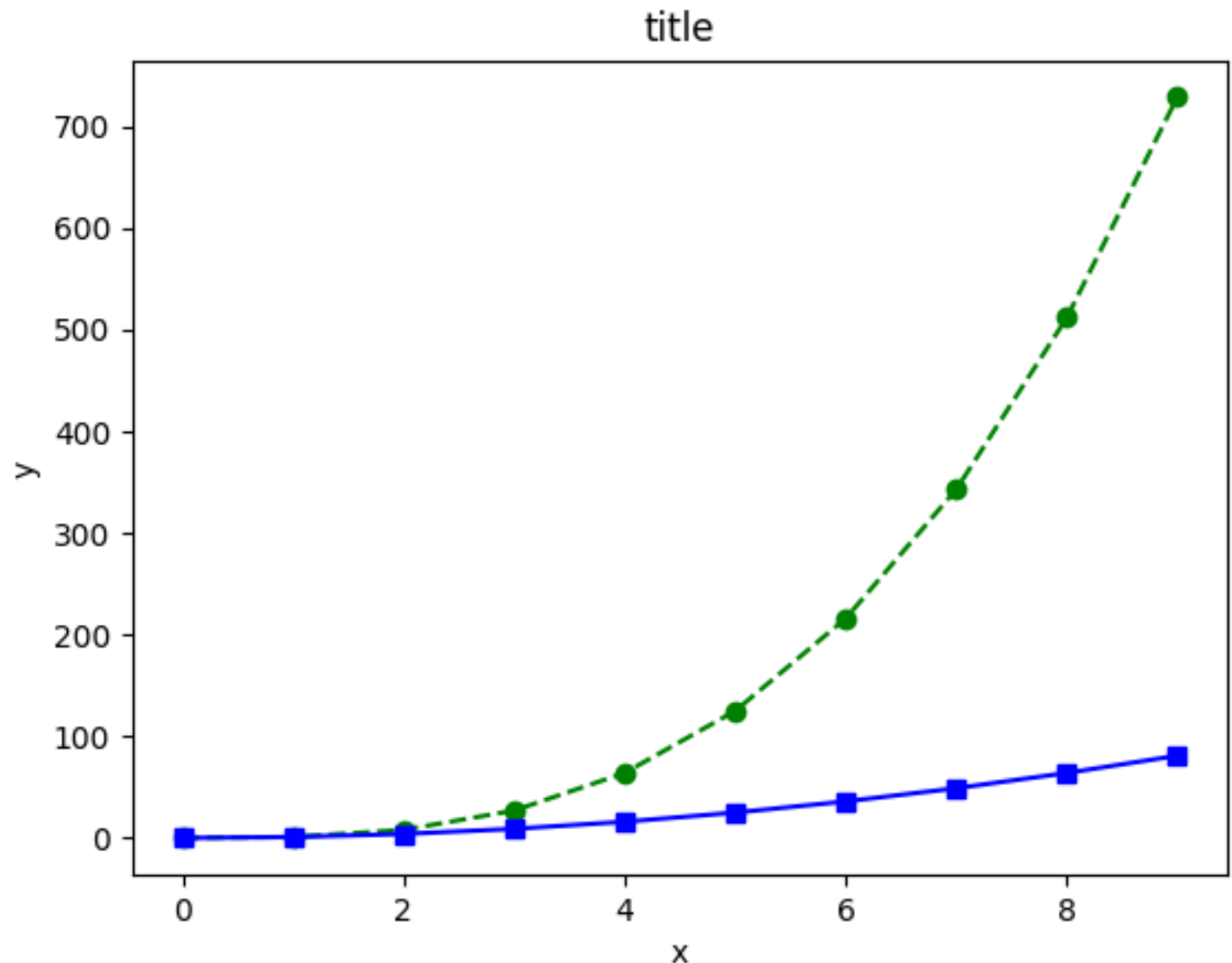




# *Matplotlib* (nomes)

- Podemos atribuir um título ao gráfico, e dar nomes aos eixos dos X e dos Y

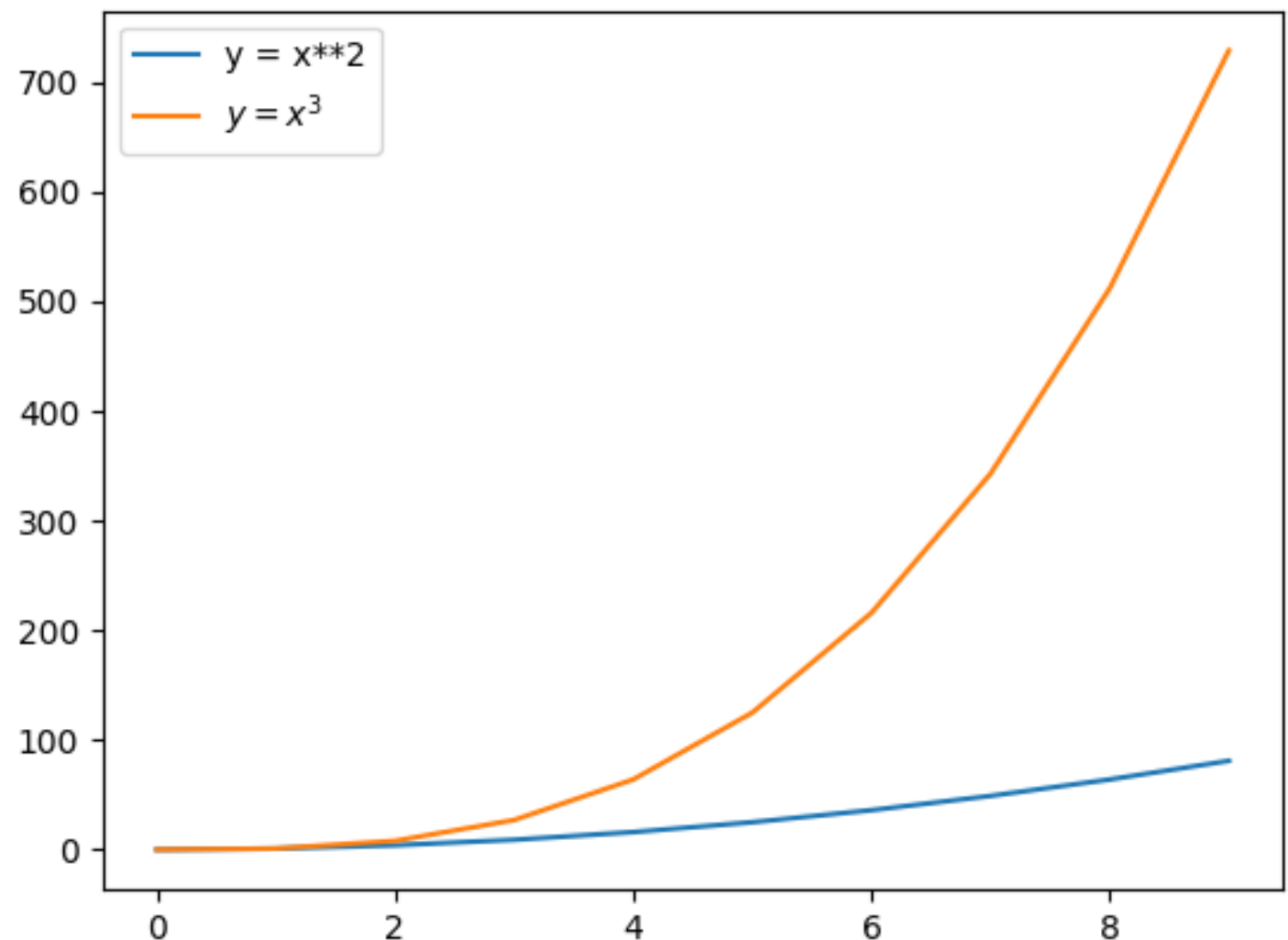
```
x = np.arange(10)
plt.plot(x, x**3,
         'g--', marker='o')
plt.plot(x, x**2,
         'b-', marker='s')
plt.title('title')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



# Matplotlib (legendas)

- Podemos definir manualmente uma legenda, definindo uma label por linha e escolhendo o seu posicionamento
- Podemos utilizar notação LaTeX

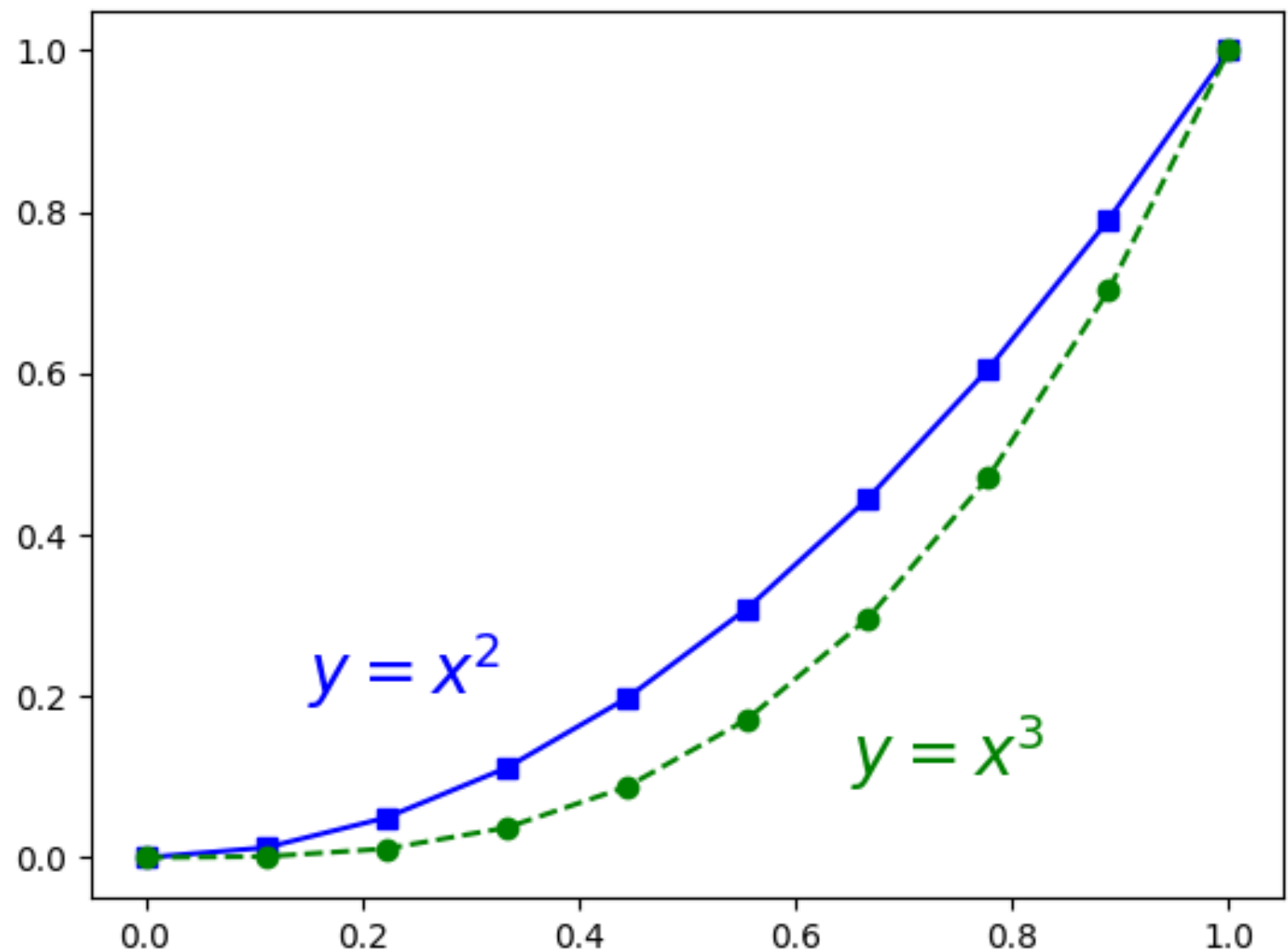
```
x = np.arange(10)
plt.plot(x, x**2,
label="y = x**2")
plt.plot(x, x**3,
label="$y = x^3$")
plt.legend(loc='upper
left')
plt.show()
```



# Matplotlib (anotações)

```
x = np.linspace(0., 1., 10)
plt.plot(x, x**2, 'b-', marker='s')
plt.plot(x, x**3, 'g--', marker='o')
plt.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
plt.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
plt.show()
```

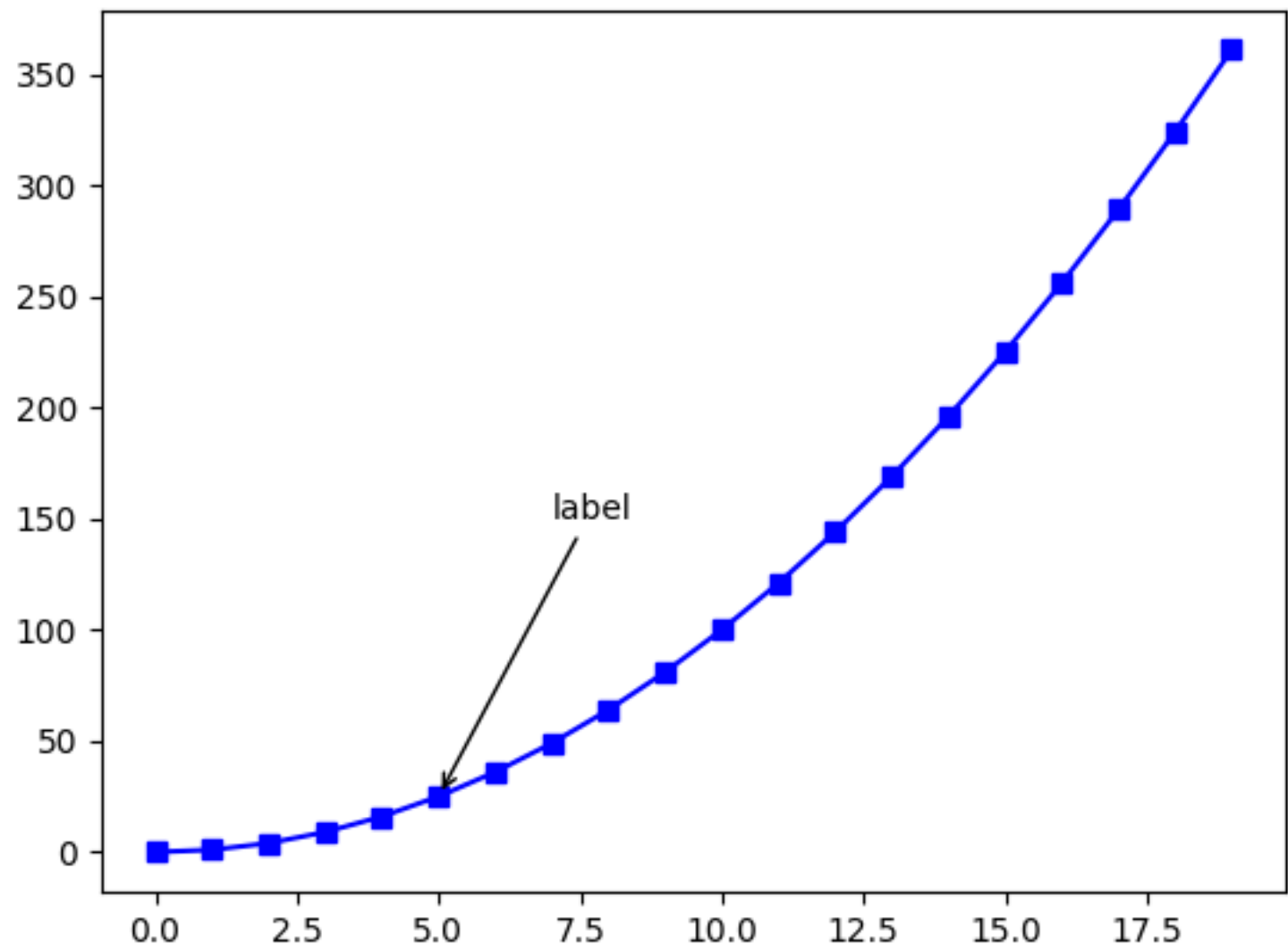
- Podemos definir anotações de texto numa posição (x,y) do gráfico



# Matplotlib (anotações)

```
x = np.arange(20)
plt.plot(x, x**2, 'b-', marker='s')
plt.annotate("label", xy=(5, 25), xycoords='data', xytext=(7, 150),
textcoords='data', arrowprops=dict(arrowstyle="->", connectionstyle="arc3"))
plt.show()
```

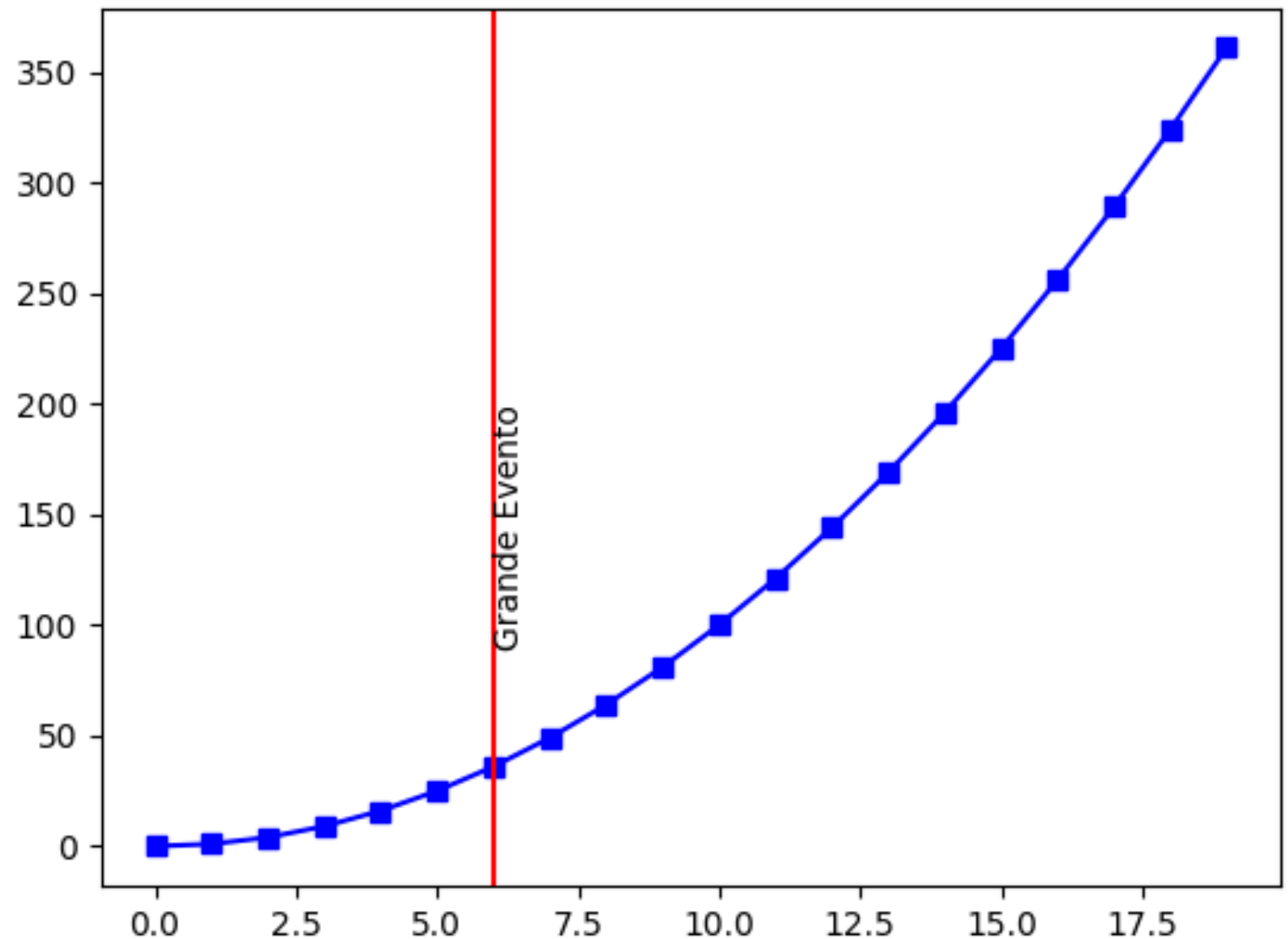
- Podemos definir setas, definindo a posição (x,y) de origem e a posição (x,y) de destino



# Matplotlib (anotações)

```
x = np.arange(20)
plt.plot(x, x**2, 'b-', marker='s')
plt.axvline(6, color='red')
plt.text(6, 200, 'Grande Evento', rotation=90, va='top')
plt.show()
```

- Podemos definir linhas verticais, definindo a posição nos X

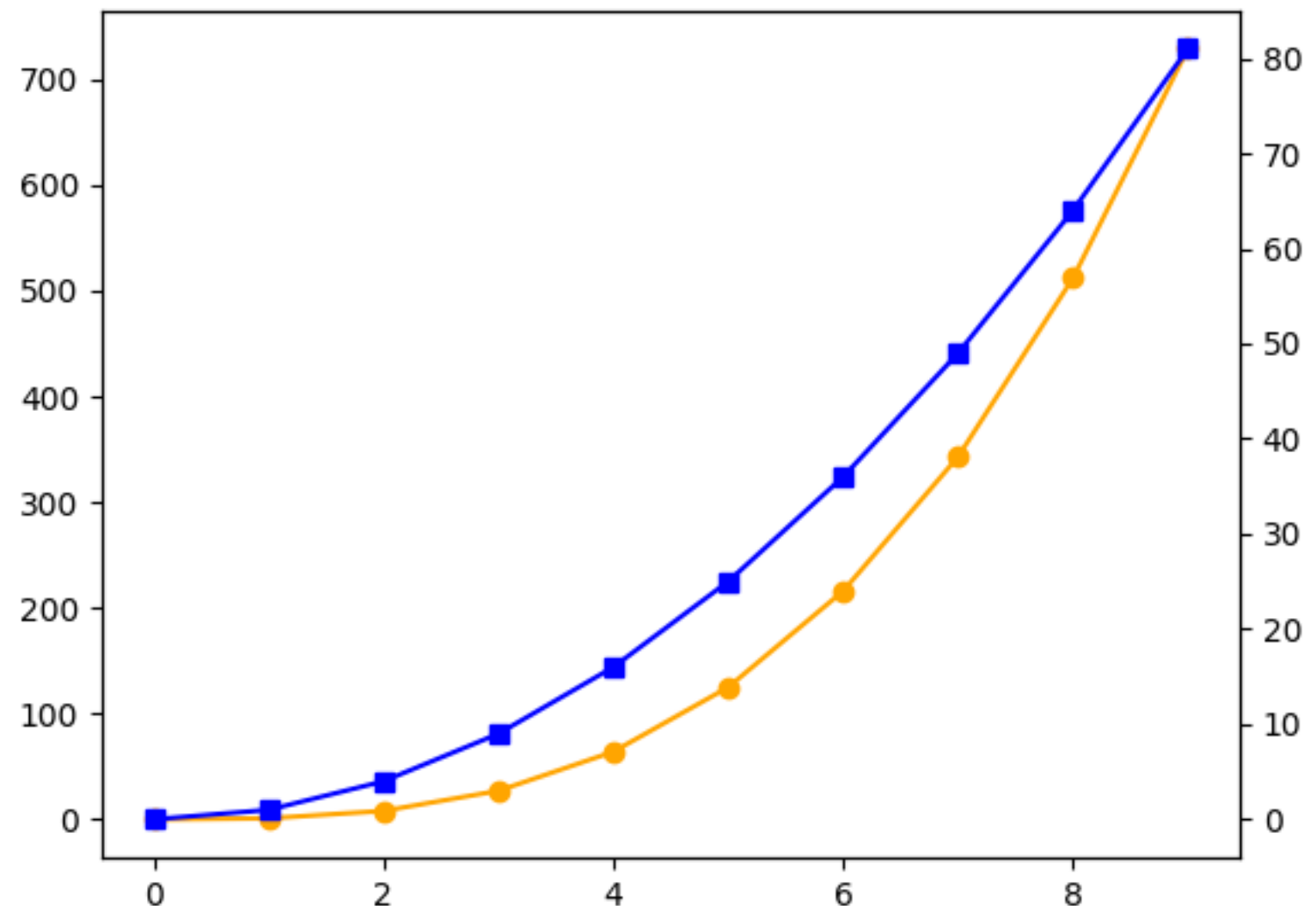


# Matplotlib (eixos)

```
_ ,plt1 = plt.subplots()
plt2 = plt1.twinx()

x = np.arange(10)
plt1.plot(x,x**3,marker='o',c="orange")
plt2.plot(x,x**2,marker='s',c="blue")
plt.show()
```

- Podemos definir eixos dos Y duais com o método *twinx()*
- Note a utilização de *subplots*, cada linha é desenhada num plot (axes) diferente

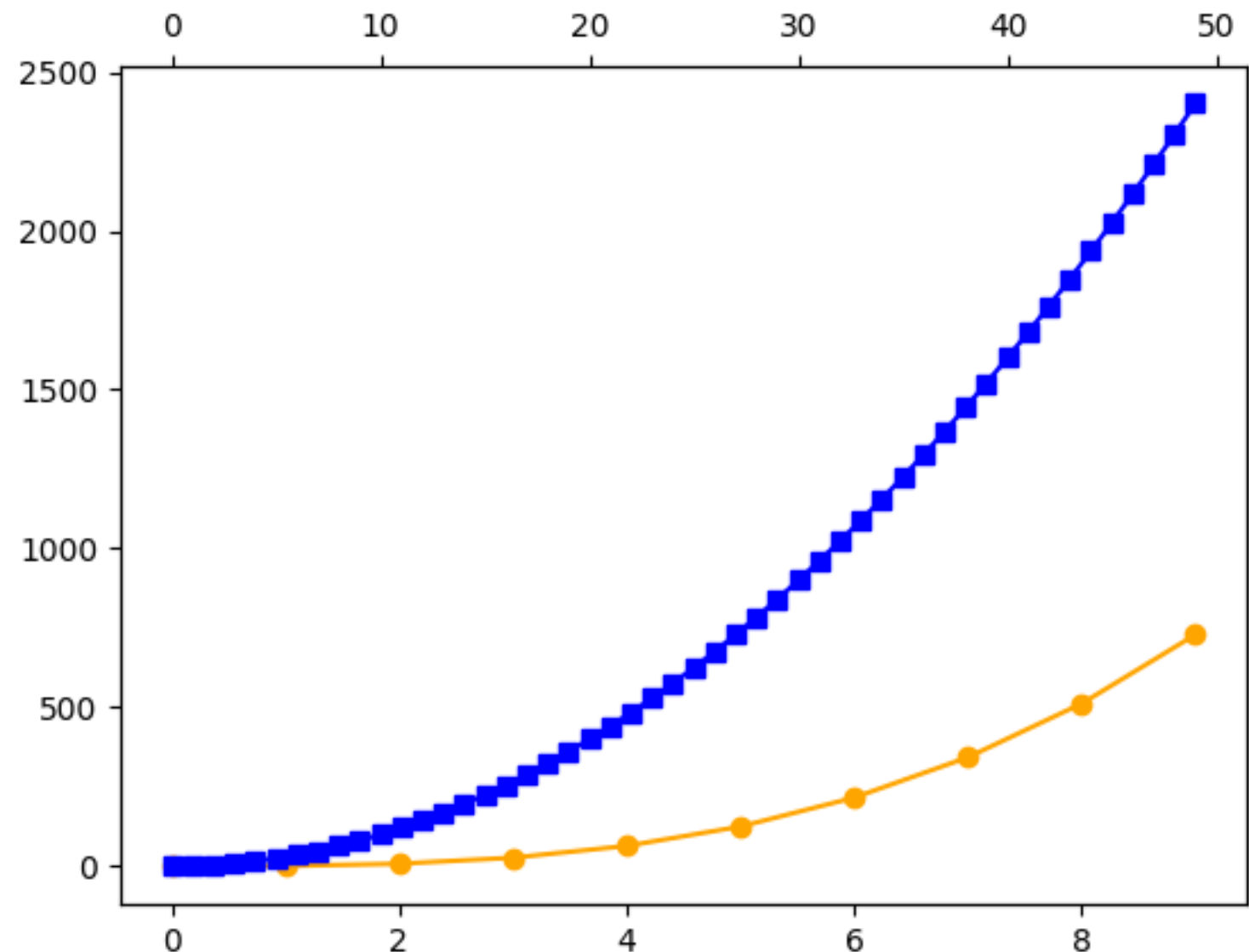


# Matplotlib (eixos)

```
_ ,plt1 = plt.subplots()
plt2 = plt1.twinx()

x1 = np.arange(10)
x2 = np.arange(50)
plt1.plot(x1,x1**3,marker='o',c="orange")
plt2.plot(x2,x2**2,marker='s',c="blue")
plt.show()
```

- Podemos definir eixos dos X duais com o método *twinx()*
- Note a utilização de *subplots*, cada linha é desenhada num plot (*axes*) diferente

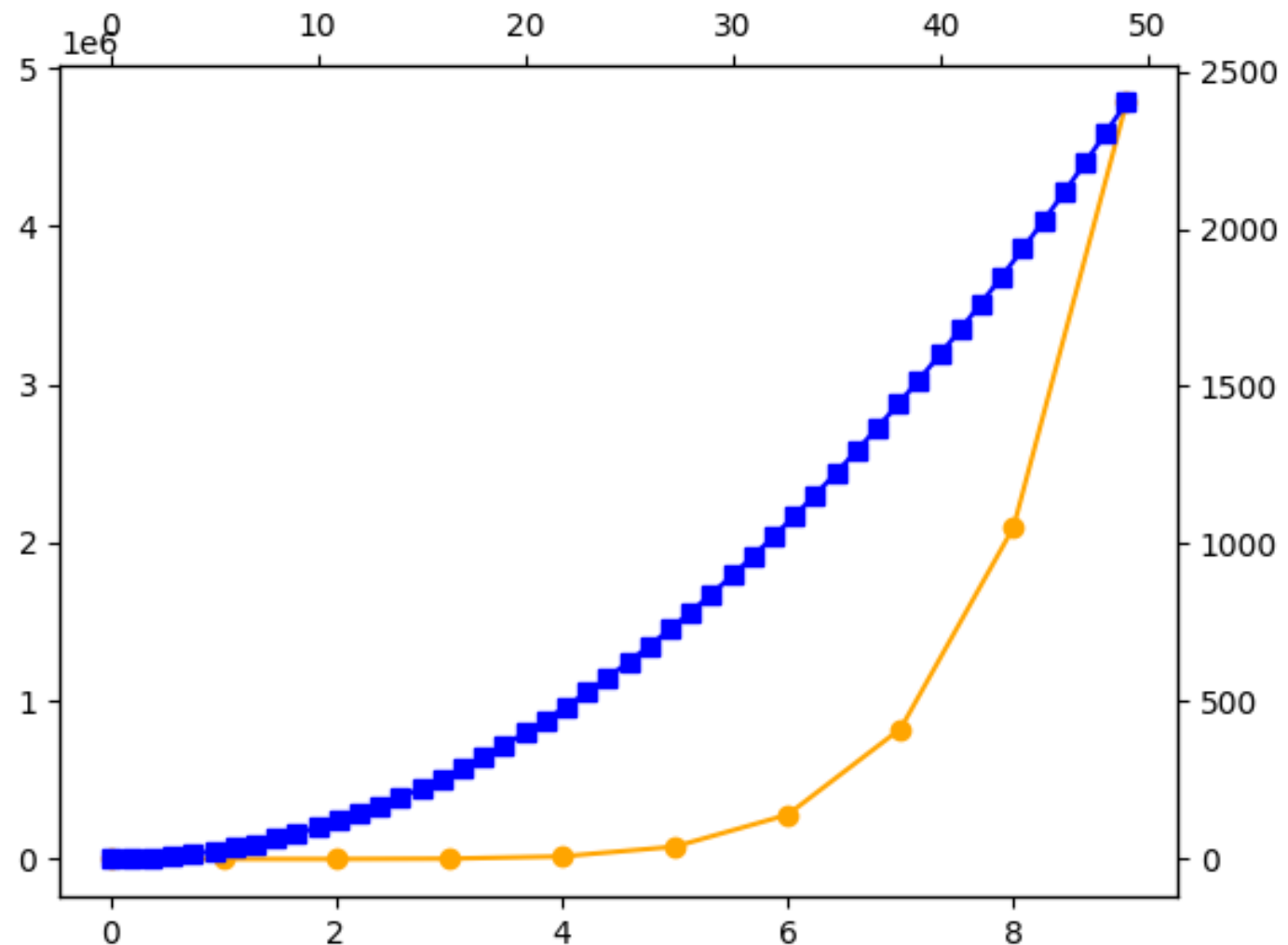


# Matplotlib (eixos)

```
_ ,plt1 = plt.subplots()
plt2 = plt1.twinx()
plt3 = plt2.twinx()

x1 = np.arange(10)
x2 = np.arange(50)
plt1.plot(x1,x1**7,marker='o',c="orange")
plt3.plot(x2,x2**2,marker='s',c="blue")
plt.show()
```

- Podemos definir eixos dos X e dos Y duais
- Note a utilização de *subplots*, cada linha é desenhada num plot (axes) diferente



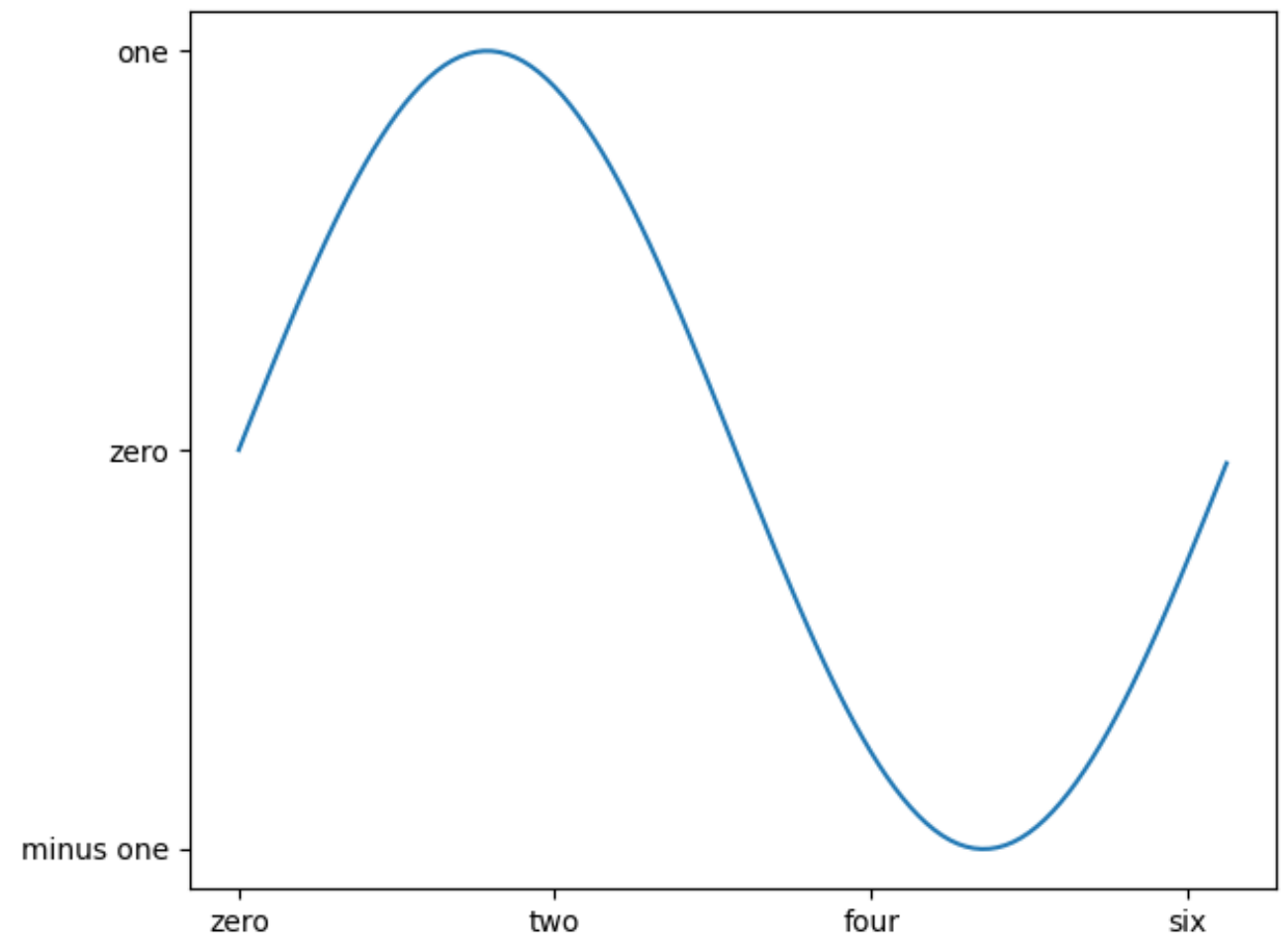


# Matplotlib (eixos)

```
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)

plt.plot(x, y)
plt.xticks([0, 2, 4, 6], ['zero', 'two', 'four', 'six'])
plt.yticks([-1, 0, 1], ['minus one', 'zero', 'one'])
plt.tight_layout()
plt.show()
```

- Podemos controlar os marcadores em cada eixo
- Podemos controlar os nomes de cada marcador

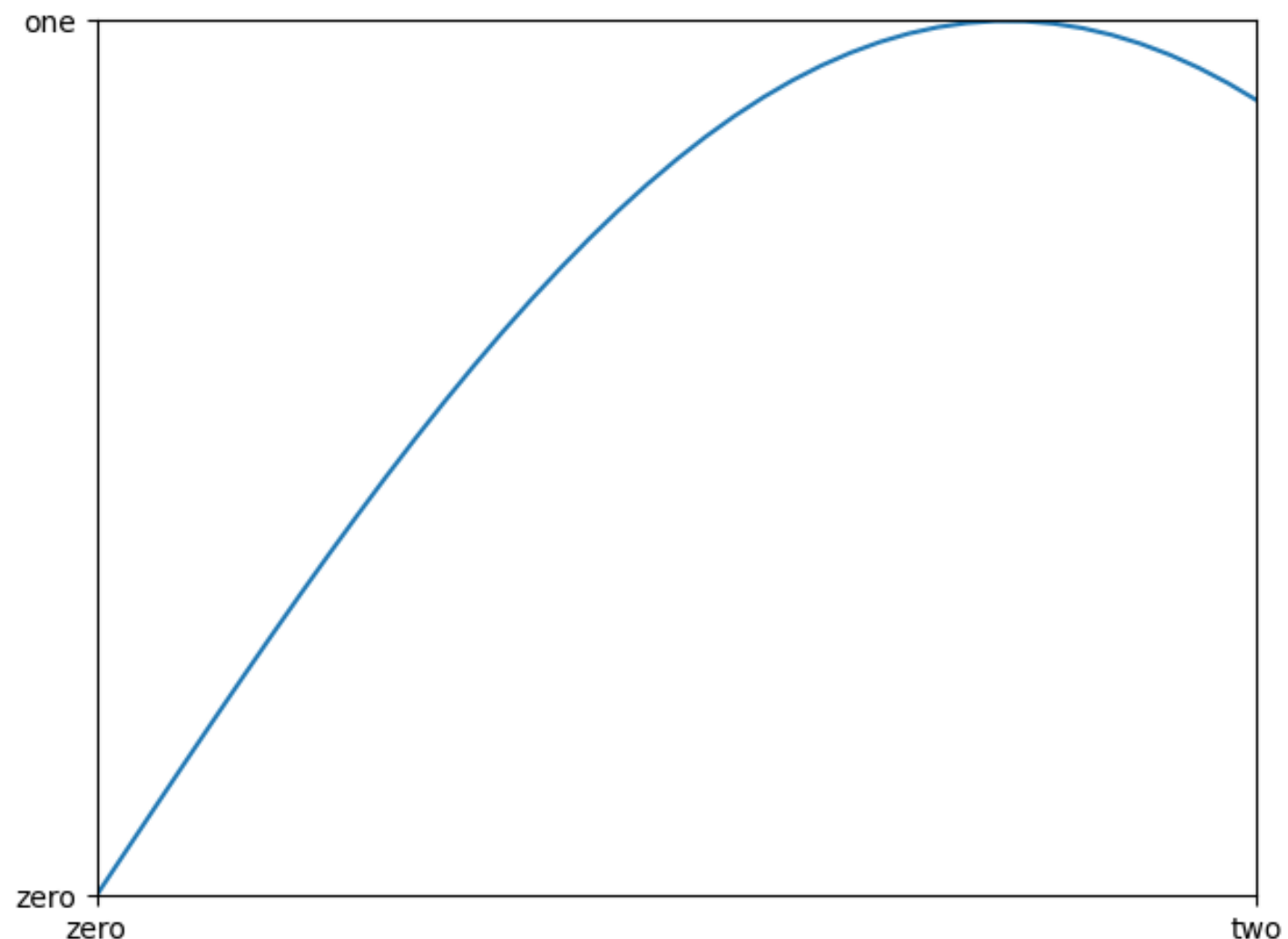


# Matplotlib (eixos)

```
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
```

```
plt.plot(x, y)
plt.xticks([0, 2, 4, 6], ['zero', 'two', 'four', 'six'])
plt.yticks([-1, 0, 1], ['minus one', 'zero', 'one'])
plt.xlim(0, 2)
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```

- Podemos controlar os limites do gráfico em cada eixo
- E.g., zoom in ou zoom out



# Matplotlib (eixos)

```
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
```

```
plt.plot(x,y)
plt.xticks([0,2,4,6],['zero','two','four','six'])
plt.yticks([-1,0,1],['minus one','zero','one'])
plt.tick_params(axis='x',bottom=False,labelbottom=False,
top=True,labeltop=True)
plt.tick_params(axis='y',right=True,labelright=True)
plt.tight_layout()
plt.show()
```

- Podemos controlar eixos manualmente

