

Estudo de Python e dados I

Introdução: Básico de DataFrame Pandas, Estrutura de dados Pandas e Visualização com Pandas, Matplotlib e Seaborn

Sergio Pedro Rodrigues Oliveira

01 janeiro 2026

SUMÁRIO

| | | |
|----------|--|-----------|
| 1 | Objetivo | 1 |
| 2 | Básico sobre o DataFrame do Pandas | 1 |
| 2.1 | Introdução | 1 |
| 2.2 | Carregando seu primeiro conjunto de dados | 2 |
| 2.3 | Observando colunas, linhas e células | 7 |
| 2.3.1 | Obtendo subconjuntos de colunas | 7 |
| | Obtendo subconjuntos de colunas pelo nome | 7 |
| | Obter subconjuntos de colunas pela posição dos índices não funciona mais no Pandas v0.20 | 9 |
| 2.3.2 | Obtendo subconjuntos de linhas | 10 |
| | Obtendo subconjuntos de linhas pelo rótulo dos índices: <code>loc</code> | 10 |
| | Obtendo subconjuntos de linhas pelo número das linhas: <code>iloc</code> | 15 |
| | Obtenção de subconjuntos de linhas com <code>ix</code> não funciona mais no Pandas v0.20 | 17 |
| 2.3.3 | Combinando tudo | 18 |
| | Obtendo subconjuntos de colunas | 18 |
| | Obtendo subconjuntos de colunas por intervalo | 20 |
| | Fatiando colunas | 23 |
| | Obtendo subconjuntos de linhas e de colunas | 26 |
| | Obtendo subconjuntos de várias linhas e de colunas | 27 |
| 2.4 | Cálculos agrupados e agregados | 29 |
| 2.4.1 | Médias agrupadas | 30 |
| 2.4.2 | Condutodes de frequência agrupados | 34 |
| 2.5 | Plotagem básica | 35 |
| 2.6 | Conclusão | 37 |
| 3 | Estrutura de dados do Pandas | 38 |
| 3.1 | Introdução | 38 |
| | Mapa Conceitual | 38 |
| 3.2 | Criando seus próprios dados | 39 |
| 3.2.1 | Criando uma Series | 39 |
| 3.2.2 | Criando um Dataframe | 46 |
| 3.3 | Series | 49 |
| 3.3.1 | Series é semelhante a <code>ndarray</code> | 53 |
| 3.3.1.1 | Métodos de Series | 53 |
| 3.3.2 | Subconjuntos com booleanos: Series | 55 |
| 3.3.3 | Operações são alinhadas e vetorizadas automaticamente (Broadcasting) | 59 |
| 3.3.3.1 | Vetores de mesmo tamanho | 59 |
| 3.3.3.2 | Vetores com inteiros (escalares) | 60 |

| | | |
|--|---|-----------|
| 3.3.3.3 | Vetores com tamanhos diferentes | 61 |
| 3.3.3.4 | Vetores com rótulos de índice comuns (alinhamento automático) | 62 |
| 3.4 | Dataframe | 64 |
| 3.4.1 | Subconjuntos com booleanos: <code>DataFrames</code> | 64 |
| 3.4.2 | Operações são alinhadas e vetorizadas automaticamente (<i>Broadcasting</i>) | 66 |
| Escalar | 67 | |
| Somar - método <code>.add()</code> | 68 | |
| 3.5 | Fazendo alterações em Series e em Dataframe | 70 |
| 3.5.1 | Adicionando mais colunas | 70 |
| 3.5.2 | Alterando diretamente uma coluna | 73 |
| 3.5.3 | Descartando valores | 78 |
| 3.6 | Exportando e importando dados | 79 |
| 3.6.1 | <code>pickle</code> | 79 |
| 3.6.1.1 | <code>Series</code> | 79 |
| 3.6.1.2 | <code>DataFrame</code> | 80 |
| 3.6.1.3 | Lendo dados de <code>pickle</code> | 81 |
| 3.6.2 | <code>CSV</code> | 82 |
| 3.6.2.1 | Exportando dados <code>CSV</code> | 82 |
| 3.6.2.2 | Removendo os números das linhas de saída | 83 |
| 3.6.2.3 | Importando dados <code>CSV</code> | 83 |
| 3.6.3 | <code>Excel</code> | 84 |
| 3.6.3.1 | <code>Series</code> | 84 |
| 3.6.3.2 | <code>DataFrame</code> | 85 |
| 3.6.3.3 | Bibliotecas que converte <code>DataFrame</code> em <code>Excel</code> | 85 |
| 3.6.3.4 | Para leitura de arquivo <code>Excel</code> no Python | 86 |
| 3.6.4 | Formato <code>feather</code> para interface com R | 87 |
| 3.6.5 | Outros tipos de saída de dados | 88 |
| 4 | Introdução à plotagem | 90 |
| 4.1 | Introdução | 90 |
| Mapa Conceitual | 90 | |
| Objetivos | 90 | |
| 4.2 | <code>matplotlib</code> | 93 |
| <code>matplotlib.pyplot</code> | 93 | |
| Subplots | 95 | |
| Partes de uma figura | 99 | |
| 4.3 | Gráficos estatísticos usando a <code>matplotlib</code> | 101 |
| 4.3.1 | Univariado | 102 |
| 4.3.1.1 | Histogramas | 102 |
| 4.3.2 | Bivariado | 103 |
| 4.3.2.1 | Gráfico de dispersão | 103 |
| 4.3.2.2 | Gráfico de caixa | 105 |
| 4.3.3 | Dados multivariados | 107 |

| | | |
|---------|--|-----|
| 4.4 | <code>seaborn</code> | 111 |
| 4.4.1 | Univariado | 111 |
| 4.4.1.1 | Histogramas | 111 |
| 4.4.1.2 | Plotagem de densidade (estimativa de densidade por Kernel) | 116 |
| 4.4.1.3 | Rug plot | 118 |
| 4.4.1.4 | Plotagem de contadores (plotagem de barras) | 119 |
| 4.4.2 | Dados bivariados | 120 |
| 4.4.2.1 | Gráficos de dispersão | 120 |
| 4.4.2.2 | Plotagem <code>hexbin</code> | 125 |
| 4.4.2.3 | Plotagem de densidade 2D | 128 |
| 4.4.2.4 | Plotagem de barras | 131 |
| 4.4.2.5 | Gráfico de caixa | 132 |
| 4.4.2.6 | Plotagem de violino | 133 |
| 4.4.2.7 | Relacionamentos aos pares | 134 |
| 4.4.3 | Dados multivariados | 139 |
| 4.4.3.1 | Cores | 139 |
| 4.4.3.2 | Tamanho e formato | 143 |
| 4.4.3.3 | Facetas | 146 |
| 4.5 | Objetos do Pandas | 157 |
| 4.5.1 | Histogramas | 157 |
| 4.5.2 | Plotagem de densidade | 159 |
| 4.5.3 | Gráfico de dispersão | 160 |
| 4.5.4 | Plotagem <code>hexbin</code> | 161 |
| 4.5.5 | Gráfico de caixa | 163 |
| 4.6 | Temas e estilos do <code>seaborn</code> | 164 |
| 4.7 | Conclusão | 167 |

LISTA DE FIGURAS

| | | |
|----|--|-----|
| 1 | Plotagem básica no Pandas mostrando a expectativa de vida média no tempo. | 36 |
| 2 | Uma das partes mais confusas da plotagem em Python está no uso dos termos “eixo” e “eixos”, pois referem-se a diferentes partes de uma figura. Essa era a versão antiga da figura “Parts of a Figure” da documentação da <code>matplotlib</code> . | 99 |
| 3 | Uma versão mais nova da representação de “Parts of a Figure”, com mais detalhes sobre outros aspectos de uma figura. De modo diferente da figura antiga, a figura mais recente foi totalmente criada com a <code>matplotlib</code> . | 100 |
| 4 | Plotagem do seaborn com facetas criadas manualmente contendo diversas variáveis. | 150 |
| 5 | Plotagem do seaborn com facetas criadas manualmente com duas variáveis. | 152 |
| 6 | Plotagem do seaborn com facetas criadas manualmente com duas variáveis violin. | 155 |
| 7 | Plotagem do seaborn com facetas criadas manualmente com duas variáveis boxplot. | 156 |
| 8 | Histograma de uma Series do Pandas. | 157 |
| 9 | Histograma de um DataFrame do Pandas. | 158 |
| 10 | Plotagem de KDE do Pandas. | 159 |
| 11 | Gráfico de dispersão do Pandas. | 160 |
| 12 | Plotagem hexbin Pandas. | 161 |
| 13 | Plotagem hexbin do Pandas com tamanho de grade modificado. | 162 |
| 14 | Plotagem de caixas do Pandas. | 163 |
| 15 | Estilo básico do seaborn. | 164 |
| 16 | Estilo whitegrid do seaborn. | 165 |
| 17 | Todos os estilos seaborn. | 166 |

LISTA DE TABELAS

| | | |
|---|---|----|
| 1 | Informações do método <code>info()</code> do Pandas | 5 |
| 2 | Tipos do Pandas versus tipos de Python | 6 |
| 3 | Diferentes métodos para indexação de linhas (ou de colunas). | 10 |
| 4 | Alguns dos atributos de uma <code>Series</code> | 52 |
| 5 | Alguns métodos que podem ser executados em uma <code>Series</code> | 55 |
| 6 | Tabela de métodos para obtenção de subconjuntos de <code>DataFrame</code> | 65 |
| 7 | Principais bibliotecas de conversão de dados para Excel | 85 |
| 8 | Métodos de <code>DataFrame</code> para exportação | 88 |

1 Objetivo

O objetivo deste estudo é explorar e documentar as funcionalidades essenciais das principais bibliotecas científicas do Python, como **NumPy**, **Pandas** e outras, através de exemplos práticos e casos de uso selecionados. Pretende-se consolidar o conhecimento sobre a manipulação, análise e visualização de dados, servindo como um guia de referência pessoal para futuros projetos de programação científica.

2 Básico sobre o DataFrame do Pandas

2.1 Introdução

O **Pandas** é uma biblioteca Python de código aberto para análise de dados. Ele dá a Python a capacidade de trabalhar com dados do tipo planilha, permitindo **carregar**, **manipular**, **alinhar** e **combinar dados** rapidamente, entre outras funções.

Para proporcionar esses recursos mais sofisticados ao Python, o **Pandas** introduz dois novos tipos de dados: **Series** e **DataFrame**.

- **DataFrame**

Representa os dados de planilhas ou retangulares completos.

- **Series**

Corresponde a única coluna do **DataFrame**.

- Também podemos pensar em um **DataFrame** do **Pandas** como um **dicionário** ou uma coleção de objetos **Series**.

Por que você deveria usar uma linguagem de programação como **Python** e uma ferramenta como o **Pandas** para trabalhar com dados? Tudo se reduz à automação e à reproduzibilidade.

Objetivos do capítulo:

1. Carga de um arquivo de dados simples e delimitado.
2. Como contar quantas linhas e colunas foram carregadas.
3. Como delimitar quais tipos de dados foram carregados.
4. Observação de diferentes porções de dados criando subconjuntos de linhas e colunas.

2.2 Carregando seu primeiro conjunto de dados

Dado um conjunto de dados inicialmente o carregamos e começamos a observar sua estrutura e conteúdo.

O modo mais simples de observar um conjunto de dados é analisar e criar subconjuntos de linhas e colunas específicas. Podemos ver quais tipos de informação estão armazenadas em cada coluna, e começar a procurar padrões por meio de estatísticas descritivas agregadas.

Como o **Pandas** não faz parte da biblioteca-padrão de Python, devemos dizer antes ao Python que carregue a biblioteca (`import`):

```
import pandas as pd
```

Quando trabalhamos com funções **Pandas**, usar o alias `pd` para `pandas` é uma prática comum.

Com a biblioteca carregada, podemos usar a função `read_csv` para carregar um arquivo de dados **CSV**. Para acessar a função `read_csv` do Pandas, usamos a notação de ponto.

```
# Por padrão, a função read_csv lerá um arquivo separado por vírgula;
# Nossos dados Gapminder estão separados por tabulações;
# Podemos usar o parâmetro sep a representar uma tabulação com \t
import pandas as pd # Importa a biblioteca pandas como 'pd'.

# --- Carregamento e Inspeção Inicial ---
df = pd.read_csv('./Data/Cap_01/gapminder.tsv', sep='\t')
# Carrega o arquivo TSV em um DataFrame, usando tabulação como separador.

# Usamos o método head para que Python nos mostre as 5 primeiras linhas
print(df.head())
```

| | country | continent | year | lifeExp | pop | gdpPercap |
|---|-------------|-----------|------|---------|----------|------------|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 1 | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| 2 | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| 3 | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| 4 | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |

- Função `type()`:

Podemos verificar se estamos trabalhando com um `DataFrame` do Pandas usando a função embutida `type` (isto é, se ele vem diretamente de Python, e não de algum pacote, como o Pandas).

A função `type()` é conveniente quando começamos a trabalhar com vários tipos diferentes de objetos Python e precisamos saber em qual objeto estamos trabalhando no momento.

```
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

- Atributo `shape`:

No momento, o conjunto de dados que carregamos está salvo como um objeto `DataFrame` do **Pandas**, e é relativamente pequeno.

Todo objeto `DataFrame` tem um atributo `shape` que nos dará o número de linhas e de colunas desse objeto.

O atributo `shape` devolve uma tupla¹ na qual o primeiro valor é o número de linhas e o segundo é a quantidade de colunas.

Com base nesse resultado anterior, podemos ver que nosso conjunto de dados Gapminder tem 1704 linhas e 6 colunas.

Como `shape` é um atributo de `DataFrame`, e não uma função ou um método, não há parênteses após o ponto. Se você cometer o erro de colocar parênteses depois do atributo `shape`, um erro será devolvido.

```
# Obtém o número de linhas e colunas
print(df.shape)
```

```
(1704, 6)
```

¹Uma tupla é semelhante a uma `list`, pois ambas podem armazenar informações heterogêneas. A principal diferença é que o conteúdo de uma tupla é “imutável”, o que significa que ela não pode ser alterada. As tuplas também são criadas com parênteses, () .

- Atributo `columns`:

Em geral, quando observamos um conjunto de dados pela primeira vez, queremos saber quantas linhas e colunas há (acabamos de fazer isso).

Para ter uma noção de quais informações ele contém, devemos observar as colunas.

Os nomes das colunas, assim como `shape`, são especificados usando o atributo `columns` do objeto `dataframe`.

```
# Obtém os nomes das colunas
print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtype='object')
```

- Atributo `dtypes`:

O objeto `DataFrame` do **Pandas** é semelhante a objetos do tipo `DataFrame` que se encontra em outras linguagens (por exemplo, Julia e R).

Toda coluna (`Series`) deve ser do mesmo tipo, enquanto cada linha pode conter tipos variados.

Em nosso exemplo atual, podemos esperar que a coluna `country` só contenha strings e que `year` contenha inteiros. No entanto, é melhor garantir que isso seja verdade usando o atributo `dtypes` ou o método `info()`.

O atributo `dtypes` de um `DataFrame` **Pandas** retorna uma `Series` que descreve o tipo de dado de cada coluna do `DataFrame`. Ele é útil para inspecionar os tipos de dados inferidos ou atribuídos às suas colunas, o que é crucial para operações corretas e eficientes.

```
# Obtém o dtype de cada coluna
print(df.dtypes)
```

```
country      object
continent    object
year         int64
lifeExp      float64
pop          int64
gdpPercap    float64
dtype: object
```

- Método `info()`:

O método `info()` de um **DataFrame Pandas** é uma ferramenta essencial para obter um resumo conciso e detalhado do seu **DataFrame**. Ele imprime um resumo conciso do **DataFrame**, incluindo:

Tabela 1: Informações do método `info()` do Pandas

| Informação | Descrição |
|--|--|
| Tipo de índice | Informações sobre o índice (por exemplo, <code>RangeIndex</code>). |
| Número de entradas (linhas) | Quantas linhas seu DataFrame possui. |
| Número de colunas | Quantas colunas seu DataFrame tem. |
| Contagem de valores não nulos por coluna | Para cada coluna, informa quantos valores não são nulos. |
| Dtype (tipo de dado) de cada coluna | Isso é crucial para identificar dados faltantes. Semelhante ao atributo <code>dtype</code> , mas apresentado de forma mais organizada. |
| Uso de memória | A quantidade de memória que o DataFrame está utilizando. |

```
# Obtém mais informações sobre nossos dados
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype  
---  --          --          --      
 0   country     1704 non-null   object 
 1   continent   1704 non-null   object 
 2   year        1704 non-null   int64  
 3   lifeExp     1704 non-null   float64
 4   pop         1704 non-null   int64  
 5   gdpPercap   1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

Tabela 2: Tipos do Pandas versus tipos de Python

| Tipo do Pandas | Tipo de Python | Descrição |
|----------------|----------------|---|
| object | string | Cadeia de caracteres, usado para representar texto. |
| int64 | int | Números inteiros. |
| float64 | float | Números com decimais. |
| datetime64 | datetime | <code>datetime</code> trata-se de uma biblioteca-padrão de Python (ou seja, não é carregado por padrão e deve ser importado). Representa pontos específicos no tempo. |

2.3 Observando colunas, linhas e células

Agora que somos capazes de carregar um arquivo de dados simples, queremos inspecionar o seu conteúdo. Podemos exibir o conteúdo do dataframe com `print`, mas com os dados de hoje em dia, com frequência, haverá células demais para ser possível compreender todas as informações exibidas. Em vez disso, a melhor maneira de observar nossos dados é inspecioná-los por partes, observando vários subconjuntos dos dados.

Já vimos que podemos usar o método `head()` de um dataframe para observar as cinco primeiras linhas de nossos dados. Isso é conveniente para ver se os dados foram carregados de modo apropriado e para ter uma noção de cada uma das colunas, seus nomes e o conteúdo. Às vezes, porém, talvez queiramos ver somente linhas, colunas e valores específicos de nossos dados.

2.3.1 Obtendo subconjuntos de colunas

Se quiser analisar várias colunas, especifique-as com base nos nomes, nas posições ou em intervalos.

Obtendo subconjuntos de colunas pelo nome

Se quiser observar apenas uma coluna específica de nossos dados, podemos acessá-la usando colchetes.

```
# Obtém somente a coluna country e a salva em sua própria variável
country_df = df['country']

# Mostra as 5 primeiras observações
print(country_df.head())
```

```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

```
# Mostra as 5 últimas observações
print(country_df.tail())
```

```
1699    Zimbabwe
1700    Zimbabwe
1701    Zimbabwe
1702    Zimbabwe
1703    Zimbabwe
Name: country, dtype: object
```

Para especificar várias colunas pelo nome, devemos passar uma `list` Python entre os colchetes. Isso pode parecer um pouco estranho, pois haverá dois conjuntos de colchetes.

```
# Observando country, continent e year
subset = df[['country', 'continent', 'year']]
print(subset.head())
```

```
      country continent  year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972
```

```
# Mostra as 5 últimas observações
print(subset.tail())
```

```
      country continent  year
1699  Zimbabwe     Africa  1987
1700  Zimbabwe     Africa  1992
1701  Zimbabwe     Africa  1997
1702  Zimbabwe     Africa  2002
1703  Zimbabwe     Africa  2007
```

Mais uma vez, é possível optar por exibir todo o dataframe subset usando `print`.

Obter subconjuntos de colunas pela posição dos índices não funciona mais no Pandas v0.20

Ocasionalmente, talvez você queira obter uma coluna em particular com base em sua posição, e não em seu nome. Por exemplo, pode querer a primeira (“country”) e a terceira (“year”) colunas, ou somente a última (“gdpPercap”).

No `pandas` v0.20 não é mais possível passar uma lista de inteiros entre colchetes para obter subconjuntos de colunas. Por exemplo, `df[[1]]`, `df[[0,-1]]` e `df[list(range(5))]` não funcionam mais. Há outras formas de obter subconjuntos de colunas, mas não baseadas na técnica usada para obter subconjuntos de linhas.

2.3.2 Obtendo subconjuntos de linhas

Podemos obter subconjuntos de linhas de várias maneiras, pelos nomes ou pelos índices das linhas. A Tabela 3 apresenta uma visão geral rápida dos diversos métodos.

Tabela 3: Diferentes métodos para indexação de linhas (ou de colunas).

| Método para obtenção de subconjuntos | Descrição |
|---|---|
| <code>loc</code> | Subconjunto baseado no rótulo do índice (nome da linha). |
| <code>iloc</code> | Subconjunto baseada no índice da linha (número da linha). |
| <code>ix</code> (não funciona mais no Pandas v0.20) | Subconjunto baseado no rótulo do índice ou no índice da linha. |

Obtendo subconjuntos de linhas pelo rótulo dos índices: `loc`

Vamos observar uma parte de nossos dados Gapminder.

```
print(df.head())
```

```
      country continent  year  lifeExp      pop  gdpPercap
0  Afghanistan      Asia  1952  28.801  8425333  779.445314
1  Afghanistan      Asia  1957  30.332  9240934  820.853030
2  Afghanistan      Asia  1962  31.997 10267083  853.100710
3  Afghanistan      Asia  1967  34.020 11537966  836.197138
4  Afghanistan      Asia  1972  36.088 13079460  739.981106
```

A esquerda do DataFrame exibido, vemos o que parece ser os números das linhas. Essa lista de valores sem coluna é o rótulo dos índices do dataframe.

Pense no rótulo dos índices como um nome de coluna, mas para linhas em vez de colunas. Por padrão, o Pandas preencherá os rótulos dos índices com os números das linhas (observe que a contagem começa em 0).

Um exemplo comum em que os rótulos dos índices das linhas não são iguais ao número das linhas ocorre quando trabalhamos com dados de séries temporais. Nesse caso, o rótulo dos índices será algum tipo de timestamp. Por exemplo, manteremos os valores default, que são os números das linhas.

Podemos usar o atributo `loc` do dataframe para obter subconjuntos de linhas com base no rótulo dos índices.

```
# Obtém a primeira linha  
# Python começa a contar de 0  
print(df.loc[0])
```

```
country      Afghanistan  
continent        Asia  
year            1952  
lifeExp         28.801  
pop             8425333  
gdpPercap     779.445314  
Name: 0, dtype: object
```

```
# Obtém a centesima linha  
# Python começa a contar de 0  
print(df.loc[99])
```

```
country      Bangladesh  
continent        Asia  
year            1967  
lifeExp         43.453  
pop             62821884  
gdpPercap     721.186086  
Name: 99, dtype: object
```

Para obtér a última linha, uma alternativa seria passar `-1` para `loc`, porém acarretaria num erro. Ao passar `-1` para `loc` causará um erro, pois o código procurará a linha cujo rótulo de índice (nesse caso, número da linha) seja “`-1`”, e esse valor não existe no nosso exemplo.

Em vez disso, podemos usar um pouco de Python para calcular o número de linhas e passar esse valor para `loc`.

```
# Obtém a última linha (corretamente)
# Usar o primeiro valor dado por shape para obter o número de linhas
number_of_rows = df.shape[0]

# Subtrai 1 do valor, pois queremos obter o número do último índice
last_row_index = number_of_rows - 1

# Obtem agora o subconjunto usando o índice da última linha
print(df.loc[last_row_index])
```

```
country      Zimbabwe
continent     Africa
year          2007
lifeExp       43.487
pop           12311143
gdpPercap    469.709298
Name: 1703, dtype: object
```

Como alternativa, podemos usar o método `tail` para devolver a última linha, em vez de usar o default de 5.

```
# Método tail, devolvendo a última linha
print(df.tail(n=1))
```

```
country continent year  lifeExp      pop   gdpPercap
1703  Zimbabwe    Africa  2007  43.487  12311143  469.709298
```

Observe que, quando usamos `tail()` e `loc`, os resultados foram exibidos de modo diferentes. Vamos observar o tipo devolvido quando usamos esses métodos.

```
subset_loc = df.loc[0]
subset_head = df.head(n=1)

# type usando loc para uma linha
print("type usando loc para uma linha:")
print(type(subset_loc))

# type usando head para uma linha
print("\ntype usando head para uma linha:")
print(type(subset_head))
```

```
type usando loc para uma linha:
<class 'pandas.core.series.Series'>

type usando head para uma linha:
<class 'pandas.core.frame.DataFrame'>
```

No ínicio desse capítulo, mencionamos que o Pandas introduziu dois novos tipos de dados em Python. Conforme o método que usamos e a quantidade de linhas retornada, o Pandas devolverá um objeto diferente. O modo como o objeto é exibido pela tela pode ser um indicador do tipo, mas sempre é melhor usar a função `type()` por garantia.

Obtenção de subconjuntos com várias linhas. Assim como para as colunas, podemos selecionar várias linhas.

```
# Selecione a primeira, a centesima e a milésima linha
# Observe os colchetes duplos, semelhante a sintaxe usada para
#obter subconjuntos com várias colunas
print(df.loc[[0,99,999]])
```

| | country | continent | year | lifeExp | pop | gdpPercap |
|-----|-------------|-----------|------|---------|----------|-------------|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 99 | Bangladesh | Asia | 1967 | 43.453 | 62821884 | 721.186086 |
| 999 | Mongolia | Asia | 1967 | 51.253 | 1149500 | 1226.041130 |

Obtendo subconjuntos de linhas pelo número das linhas: iloc

`iloc` faz o mesmo que `loc`, mas é usado para obter subconjuntos com base no número de índice das linhas.

Em nosso exemplo atual, `iloc` e `loc` se comportarão exatamente do mesmo modo, pois os rótulos dos índices são os números das linhas. Tenha em mente, porém, que os rótulos dos índices não necessariamente têm de ser os números das linhas.

```
# Obtém a segunda linha
print(df.iloc[1])
```

```
country      Afghanistan
continent        Asia
year            1957
lifeExp         30.332
pop             9240934
gdpPercap       820.85303
Name: 1, dtype: object
```

```
# Obtém a centésima linha
print(df.iloc[99])
```

```
country      Bangladesh
continent        Asia
year            1967
lifeExp         43.453
pop             62821884
gdpPercap       721.186086
Name: 99, dtype: object
```

Observe que quando colocamos 1 na lista, na verdade, obtemos a segunda linha, e não a primeira. Isso está de acordo com o comportamento de indexação a partir do 0 de Python, o que significa que o primeiro item de um contêiner é o índice 0 (ou seja, o item 0 do contêiner).

Com `iloc` podemos passar `-1` para **obter a última linha** - algo que não era possível com `loc`.

```
# Usando -1 para obter a última linha
print(df.iloc[-1])
```

```
country      Zimbabwe
continent     Africa
year          2007
lifeExp       43.487
pop           12311143
gdpPercap    469.709298
Name: 1703, dtype: object
```

Como antes, é possível passar **uma lista de inteiros para obter várias linhas**.

```
# Obtém a primeira, a centésima e a milésima linha
print(df.iloc[[0, 99, 999]])
```

| | country | continent | year | lifeExp | pop | gdpPercap |
|-----|-------------|-----------|------|---------|----------|-------------|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 99 | Bangladesh | Asia | 1967 | 43.453 | 62821884 | 721.186086 |
| 999 | Mongolia | Asia | 1967 | 51.253 | 1149500 | 1226.041130 |

Obtenção de subconjuntos de linhas com ix não funciona mais no Pandas v0.20

O atributo `ix` não funciona em versões de Pandas posteriores a v0.20, pois pode ser confuso. Apesar disso, faremos uma revisão rápida de `ix` nesta seção para que a explicação fique completa.

Podemos pensar em `ix` como uma combinação de `loc` e `iloc`, pois permite que tenhamos subconjuntos por rótulo ou por inteiro. Por padrão, ele procura rótulos. Se não puder encontrar o rótulo correspondente, ele recorrerá ao uso de indexação por inteiros. Isso pode ser causa de muitas confusões, e, assim, esse recurso foi removido.

O código que usa `ix` se parecerá exatamente com o código escrito quando `loc` e `iloc` são usados.

```
# Primeira linha  
df.ix[0]  
  
# Centésima linha  
df.ix[99]  
  
# Primeira, centésima e milésima linhas  
df.ix[[0,99,999]]
```

2.3.3 Combinando tudo

Os atributos `loc` e `iloc` podem ser usados para obter subconjuntos de colunas, linhas ou de ambos.

A sintaxe geral de `loc` e `iloc` faz uso de colchetes com uma vírgula. A parte à esquerda da vírgula são os valores das linhas para o subconjunto; a parte à direita são os valores das colunas. Ou seja, `df.loc[[rows], [columns]]` ou `df.iloc[[rows], [columns]]`.

Obtendo subconjuntos de colunas

Se quiser usar técnicas para obter subconjuntos somente de colunas, use a sintaxe de fatiamento (slicing) de Python. Temos de fazer isso porque, se estivermos gerando subconjuntos de colunas, teremos todas as linhas da coluna especificadas. Portanto precisamos de um método para capturar todas as linhas.

A sintaxe de fatiamento de Python usa dois-pontos, isto é, `[:, :]`. Se tivermos apenas dois-pontos sozinhos, o atributo se referirá a tudo. Assim, se quisermos obter somente a primeira coluna usando a sintaxe `loc` ou de `iloc`, podemos escrever algo como `df.loc[:, [columns]]` para obter o subconjunto da(s) coluna(s).

```
# Obtendo um subconjunto de colunas com loc
# Observe a posição dos dois-pontos
# Ele é usado para selecionar todas as linhas
subset = df.loc[:, ['year', 'pop']]
print(subset.head())
```

| | year | pop |
|---|------|----------|
| 0 | 1952 | 8425333 |
| 1 | 1957 | 9240934 |
| 2 | 1962 | 10267083 |
| 3 | 1967 | 11537966 |
| 4 | 1972 | 13079460 |

```
# Obtendo um subconjunto de colunas com iloc
# iloc nos permitirá usar inteiros
# -1 selecionará a última coluna
subset = df.iloc[:,[2,4,-1]]
print(subset.head())
```

| | year | pop | gdpPercap |
|---|------|----------|------------|
| 0 | 1952 | 8425333 | 779.445314 |
| 1 | 1957 | 9240934 | 820.853030 |
| 2 | 1962 | 10267083 | 853.100710 |
| 3 | 1967 | 11537966 | 836.197138 |
| 4 | 1972 | 13079460 | 739.981106 |

Obtendo subconjuntos de colunas por intervalo

Podemos usar a função embutida `range` para criar um intervalo de valores em Python. Desse modo, é possível especificar os valores de início e fim, e Python criará automaticamente um intervalo com os valores entre eles.

Por padrão, todo valor entre o início e o fim (**inclusive à esquerda, não inclusive a direita**) será criado, a menos que você especifique um passo.

Em Python 3, a função `range` devolve um gerador.

Se estiver usando Python 2, a função `range` devolverá uma lista e a função `xrange` devolve um gerador.

Se observarmos o código apresentado antes, veremos que subconjuntos de colunas foram obtidos usando uma lista de inteiros. Como `range` devolve um gerador, é preciso convertê-lo em uma lista antes.

```
list(range(5))
```

Observe que, quando `range(5)` é chamado, cinco inteiros são devolvidos: 0-4.

```
# Cria um intervalo de inteiros de 0 a 4 inclusive, [0,5)
small_range = list(range(5))
print(small_range)
```

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

```
# Obtém um subconjunto de dataframe usando o intervalo
subset = df.iloc[:,small_range]
print(subset.head())
```

| | country | continent | year | lifeExp | pop |
|---|-------------|-----------|------|---------|----------|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 |
| 1 | Afghanistan | Asia | 1957 | 30.332 | 9240934 |
| 2 | Afghanistan | Asia | 1962 | 31.997 | 10267083 |
| 3 | Afghanistan | Asia | 1967 | 34.020 | 11537966 |
| 4 | Afghanistan | Asia | 1972 | 36.088 | 13079460 |

```
# Cria um intervalo de 3 a 5 incluvie, [3,5] ou [3,6)
small_range = list(range(3,6))
print(small_range)
subset = df.iloc[:,small_range]
print(subset.head())
```

```
[3, 4, 5]
  lifeExp      pop   gdpPercap
0    28.801  8425333  779.445314
1    30.332  9240934  820.853030
2    31.997 10267083  853.100710
3    34.020 11537966  836.197138
4    36.088 13079460  739.981106
```

Pergunta (Desafio 1):

O que acontecerá se você especificar um intervalo que estiver além do número de colunas existente?

Resposta:

Erro do tipo IndexError.

Mais uma vez, observe que os valores são especificados de modo que **o intervalo é inclusivo à esquerda, mas não a direita**.

```
# Cria um intervalo de 0 a 5 inclusive, com inteiros alternados [0,6)
small_range = list(range(0,6,2))
subset = df.iloc[:,small_range]
print(subset.head())
```

| | country | year | pop |
|---|-------------|------|----------|
| 0 | Afghanistan | 1952 | 8425333 |
| 1 | Afghanistan | 1957 | 9240934 |
| 2 | Afghanistan | 1962 | 10267083 |
| 3 | Afghanistan | 1967 | 11537966 |
| 4 | Afghanistan | 1972 | 13079460 |

Converter um gerador em uma lista é um pouco complicado; podemos usar a sintaxe de fatiamento de Python para dar um jeito nisso.

Fatiando colunas

A sintaxe de **fatiamento de Python**, `:`, é semelhante à sintaxe de `range`. Em vez de usar uma função que especifique os valores de início, fim e o passo, delimitados por vírgula, separamos os valores com dois-pontos.

Se você entendeu o que estava acontecendo com a função `range` que usamos antes, o fatiamento então poderá ser visto como um atalho para fazer o mesmo.

Exemplo `range`:

```
small_range = list(range(3))
subset = df.iloc[:,small_range]
print(subset.head())
```

| | country | continent | year |
|---|-------------|-----------|------|
| 0 | Afghanistan | Asia | 1952 |
| 1 | Afghanistan | Asia | 1957 |
| 2 | Afghanistan | Asia | 1962 |
| 3 | Afghanistan | Asia | 1967 |
| 4 | Afghanistan | Asia | 1972 |

Exemplo fatiamento de Python:

```
# Fatia as três primeiras colunas
subset = df.iloc[:, :3]
print(subset.head())
```

| | country | continent | year |
|---|-------------|-----------|------|
| 0 | Afghanistan | Asia | 1952 |
| 1 | Afghanistan | Asia | 1957 |
| 2 | Afghanistan | Asia | 1962 |
| 3 | Afghanistan | Asia | 1967 |
| 4 | Afghanistan | Asia | 1972 |

Exemplo range:

```
small_range = list(range(3,6))
subset = df.iloc[:,small_range]
print(subset.head())
```

| | lifeExp | pop | gdpPercap |
|---|---------|----------|------------|
| 0 | 28.801 | 8425333 | 779.445314 |
| 1 | 30.332 | 9240934 | 820.853030 |
| 2 | 31.997 | 10267083 | 853.100710 |
| 3 | 34.020 | 11537966 | 836.197138 |
| 4 | 36.088 | 13079460 | 739.981106 |

Exemplo fatiamento de Python:

```
# Fazia as colunas de 3 a 5 inclusive, [3,6)
subset = df.iloc[:,3:6]
print(subset.head())
```

| | lifeExp | pop | gdpPercap |
|---|---------|----------|------------|
| 0 | 28.801 | 8425333 | 779.445314 |
| 1 | 30.332 | 9240934 | 820.853030 |
| 2 | 31.997 | 10267083 | 853.100710 |
| 3 | 34.020 | 11537966 | 836.197138 |
| 4 | 36.088 | 13079460 | 739.981106 |

Exemplo range:

```
small_range = list(range(0,6,2))
subset = df.iloc[:,small_range]
print(subset.head())
```

| | country | year | pop |
|---|-------------|------|----------|
| 0 | Afghanistan | 1952 | 8425333 |
| 1 | Afghanistan | 1957 | 9240934 |
| 2 | Afghanistan | 1962 | 10267083 |
| 3 | Afghanistan | 1967 | 11537966 |
| 4 | Afghanistan | 1972 | 13079460 |

Exemplo fatiamento de Python:

```
# Fazia as cinco primeiras colunas alternadamente
subset = df.iloc[:,::6:2]
print(subset.head())
```

| | country | year | pop |
|---|-------------|------|----------|
| 0 | Afghanistan | 1952 | 8425333 |
| 1 | Afghanistan | 1957 | 9240934 |
| 2 | Afghanistan | 1962 | 10267083 |
| 3 | Afghanistan | 1967 | 11537966 |
| 4 | Afghanistan | 1972 | 13079460 |

Obtendo subconjuntos de linhas e de colunas

Temos usado dois-pontos, `:`, em `loc` e em `iloc` à esquerda da vírgula. Quando fazemos isso, selecionamos todas as linhas de nosso dataframe. No entanto, podemos optar por colocar valores à esquerda da vírgula se quisermos selecionar linhas específicas, além de colunas específicas.

```
# Usando loc  
print(df.loc[42, 'country'])
```

Angola

```
# Usando iloc  
print(df.iloc[42,0])
```

Angola

Certifique-se de que não se esquecerá das diferenças entre `loc` (rótulo do índice, nome) e `iloc` (índice, número).

Observe agora como `ix` pode ser confuso. É bom que ele não esteja mais funcionando.

Obtendo subconjuntos de várias linhas e de colunas

Podemos combinar a sintaxe de obtenção de subconjuntos de linhas e de colunas com a sintaxe de subconjuntos de várias linhas e várias colunas a fim de obter fatias de nossos dados.

```
# Obtém a primeira, a centésima e a milésima linha  
# da primeira, quarta e sexta coluna;  
# As colunas que esperamos obter são  
# country, lifeExp e gdpPercap  
print(df.iloc[[0,99,999],[0,3,5]])
```

| | country | lifeExp | gdpPercap |
|-----|-------------|---------|-------------|
| 0 | Afghanistan | 28.801 | 779.445314 |
| 99 | Bangladesh | 43.453 | 721.186086 |
| 999 | Mongolia | 51.253 | 1226.041130 |

Atenção!!!

Em meu trabalho, tento passar os nomes das colunas para obter subconjuntos de dados, sempre que possível. Essa abordagem deixa o código mais legível, pois não será necessário observar o vetor de nomes das colunas para saber qual índice está sendo especificado.

Além disso, usar índices absolutos (número da coluna) pode resultar em problemas caso a ordem das colunas seja alterada por algum motivo.

Essa é somente uma regra geral, uma vez que haverá exceções em que usar a posição do índice será uma opção melhor (por exemplo, para concatenar dados).

```
# Se usarmos os nomes das colunas diretamente,  
# o código será um pouco mais fácil de ler  
# Observe agora que temos que usar loc ao invés de iloc  
print(df.loc[[0,99,999],['country','lifeExp','gdpPercap']])
```

| | country | lifeExp | gdpPercap |
|-----|-------------|---------|-------------|
| 0 | Afghanistan | 28.801 | 779.445314 |
| 99 | Bangladesh | 43.453 | 721.186086 |
| 999 | Mongolia | 51.253 | 1226.041130 |

Usar loc sempre que possível!!!

Lembre-se de que podemos usar a sintaxe de fatiamento na parte referente às linhas dos atributos `loc` e `iloc`.

```
print(df.loc[10:13,['country','lifeExp','gdpPercap']])
```

| | country | lifeExp | gdpPercap |
|----|-------------|---------|-------------|
| 10 | Afghanistan | 42.129 | 726.734055 |
| 11 | Afghanistan | 43.828 | 974.580338 |
| 12 | Albania | 55.230 | 1601.056136 |
| 13 | Albania | 59.280 | 1942.284244 |

2.4 Cálculos agrupados e agregados

Se você já trabalhou com outras bibliotecas numéricas ou linguagens, saberá que muitos cálculos estatísticos básicos estarão disponíveis na biblioteca ou embutidos na linguagem. Vamos observar novamente nossos dados Gapminder.

```
print(df.head(n=10))
```

| | country | continent | year | lifeExp | pop | gdpPerCap |
|---|-------------|-----------|------|---------|----------|------------|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 1 | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| 2 | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| 3 | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| 4 | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |
| 5 | Afghanistan | Asia | 1977 | 38.438 | 14880372 | 786.113360 |
| 6 | Afghanistan | Asia | 1982 | 39.854 | 12881816 | 978.011439 |
| 7 | Afghanistan | Asia | 1987 | 40.822 | 13867957 | 852.395945 |
| 8 | Afghanistan | Asia | 1992 | 41.674 | 16317921 | 649.341395 |
| 9 | Afghanistan | Asia | 1997 | 41.763 | 22227415 | 635.341351 |

Perguntas estatísticas

Há várias perguntas iniciais que podemos nos fazer:

1. Para cada ano em nossos dados, qual era a expectativa de vida média? Qual é a expectativa de vida média, a população e o GDP?
2. E se estratificarmos os dados por continente e fizermos os mesmos cálculos?
3. Quantos países estão listados para cada continente?

2.4.1 Médias agrupadas

Para responder as perguntas que acabaram de ser propostas, precisamos fazer um cálculo **agrupado** (isto é, **agregado**). Em outras palavras, temos de fazer um cálculo, seja uma média ou uma contagem de frequência, mas aplicá-lo em cada subconjunto de uma variável.

Outro modo de pensar em cálculos agrupados é vê-los como um processo do tipo **separar-aplicar-combinar**.

- Inicialmente, separamos nossos dados em várias partes;
- Em seguida, aplicamos uma função (ou cálculo) de nossa escolha em cada parte separada;
- E, por fim, combinamos todos os cálculos individuais em um único dataframe.

Fazemos processamentos agrupados/agregados usando o método `groupby` nos dataframe.

```
# Para cada ano em nossos dados, qual era a expectativa de vida média?  
# Para responder a essa pergunta,  
# temos que separar nossos dados em partes, de acordo com o ano;  
# em seguida, obtemos a coluna 'lifeExp' e calculamos a média  
  
#Agrupamento (year)  
#Separação/subconjunto (lifeExp)  
#Aplicar (média)  
  
print(df.groupby('year')['lifeExp'].mean())
```

```
year  
1952    49.057620  
1957    51.507401  
1962    53.609249  
1967    55.678290  
1972    57.647386  
1977    59.570157  
1982    61.533197  
1987    63.212613  
1992    64.160338  
1997    65.014676  
2002    65.694923  
2007    67.007423  
Name: lifeExp, dtype: float64
```

Vamos detalhar a instrução que usamos nesse exemplo.

- Em primeiro lugar, criamos um objeto **agrupado**. Observe que, se exibíssemos o data-frame agrupado, o Pandas devolveria somente a posição na memória.

```
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))
print(grouped_year_df)
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7cf0b2308550>
```

- A partir dos dados agrupados, podemos obter um **subconjunto** das colunas de nosso interesse, nas quais queremos fazer os cálculos. Para responder à nossa pergunta, precisamos da coluna `lifeExp`. Podemos usar os métodos de obtenção de subconjuntos.

```
grouped_year_df_lifeExp = grouped_year_df['lifeExp']
print(type(grouped_year_df_lifeExp))
print(grouped_year_df_lifeExp)
```

```
<class 'pandas.core.groupby.generic.SeriesGroupBy'>
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7cf0b230b850>
```

Observe que agora temos uma série (pois pedimos apenas uma coluna) cujo conteúdo é agrupado (em nosso exemplo por ano).

- Por fim, sabemos que a coluna `lifeExp` é do tipo `float64`. Uma operação que podemos **executar** em um vetor de números e **calcular** a média para obter o resultado que desejamos.

```
mean_lifeExp_by_year = grouped_year_df_lifeExp.mean()
print(mean_lifeExp_by_year.head(n=10))
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
Name: lifeExp, dtype: float64
```

Podemos executar um conjunto semelhante de cálculos para a população e o GDP, pois eles são dos tipos `int64` e `float64`, respectivamente. Mas e se quiséssemos agrupar e estratificar os dados com base em mais de uma variável? E se quiséssemos fazer o mesmo cálculo em várias colunas? Podemos partir do código anterior apresentado e usar uma lista.

```
# A barra invertida nos permite quebrar uma linha longa de código Python
# em várias linhas.
# df.groupby(['year','continent'])[['lifeExp','gdpPercap']].mean()
# é o mesmo que o código a seguir
multi_group_var = df.\
    groupby(['year','continent'])\.
    [['lifeExp','gdpPercap']]\.
    .mean()
print(multi_group_var.head(n=20))
```

| | | lifeExp | gdpPercap |
|------|-----------|-----------|--------------|
| year | continent | | |
| 1952 | Africa | 39.135500 | 1252.572466 |
| | Americas | 53.279840 | 4079.062552 |
| | Asia | 46.314394 | 5195.484004 |
| | Europe | 64.408500 | 5661.057435 |
| | Oceania | 69.255000 | 10298.085650 |
| 1957 | Africa | 41.266346 | 1385.236062 |
| | Americas | 55.960280 | 4616.043733 |
| | Asia | 49.318544 | 5787.732940 |
| | Europe | 66.703067 | 6963.012816 |
| | Oceania | 70.295000 | 11598.522455 |
| 1962 | Africa | 43.319442 | 1598.078825 |
| | Americas | 58.398760 | 4901.541870 |
| | Asia | 51.563223 | 5729.369625 |
| | Europe | 68.539233 | 8365.486814 |
| | Oceania | 71.085000 | 12696.452430 |
| 1967 | Africa | 45.334538 | 2050.363801 |
| | Americas | 60.410920 | 5668.253496 |
| | Asia | 54.663640 | 5971.173374 |
| | Europe | 69.737600 | 10143.823757 |
| | Oceania | 71.310000 | 14495.021790 |

Os dados de saída estão agrupados por ano e por continente. Para cada par ano-continentes, calculamos a expectativa de vida média e o GDP médio.

Os dados também são exibidos de modo um pouco diferente. Observe que os “nomes das colunas” de ano e continente não estão na mesma linha que os “nomes das colunas” de expectativa de vida e GDP. Há uma certa estrutura hierárquica entre os índices das linhas de ano e continente.

Caso precise “achatar” o dataframe, use o método `reset_index`.

```
flat = multi_group_var.reset_index()
print(flat.head(n=20))
```

| | year | continent | lifeExp | gdpPercap |
|----|------|-----------|-----------|--------------|
| 0 | 1952 | Africa | 39.135500 | 1252.572466 |
| 1 | 1952 | Americas | 53.279840 | 4079.062552 |
| 2 | 1952 | Asia | 46.314394 | 5195.484004 |
| 3 | 1952 | Europe | 64.408500 | 5661.057435 |
| 4 | 1952 | Oceania | 69.255000 | 10298.085650 |
| 5 | 1957 | Africa | 41.266346 | 1385.236062 |
| 6 | 1957 | Americas | 55.960280 | 4616.043733 |
| 7 | 1957 | Asia | 49.318544 | 5787.732940 |
| 8 | 1957 | Europe | 66.703067 | 6963.012816 |
| 9 | 1957 | Oceania | 70.295000 | 11598.522455 |
| 10 | 1962 | Africa | 43.319442 | 1598.078825 |
| 11 | 1962 | Americas | 58.398760 | 4901.541870 |
| 12 | 1962 | Asia | 51.563223 | 5729.369625 |
| 13 | 1962 | Europe | 68.539233 | 8365.486814 |
| 14 | 1962 | Oceania | 71.085000 | 12696.452430 |
| 15 | 1967 | Africa | 45.334538 | 2050.363801 |
| 16 | 1967 | Americas | 60.410920 | 5668.253496 |
| 17 | 1967 | Asia | 54.663640 | 5971.173374 |
| 18 | 1967 | Europe | 69.737600 | 10143.823757 |
| 19 | 1967 | Oceania | 71.310000 | 14495.021790 |

2.4.2 Contadores de frequência agrupados

Outra tarefa comum relacionada aos dados é calcular frequências.

Podemos usar os métodos `nunique` e `value_counts`, respectivamente, para obter contadores de valores únicos e contadores de frequência em uma `Series` do Pandas.

- Método `nunique`:

```
# Uso de nunique (number unique, ou número de únicos)
# para calcular o número de valores únicos em uma série
print(df.groupby('continent')['country'].nunique())
```

```
continent
Africa      52
Americas    25
Asia        33
Europe       30
Oceania      2
Name: country, dtype: int64
```

- Método `value_counts`:

```
# Nova série que mostra cada valor único
# e sua respectiva frequência (quantidade de ocorrências).
print(df['continent'].value_counts())
```

```
continent
Africa      624
Asia        396
Europe      360
Americas    300
Oceania      24
Name: count, dtype: int64
```

Resumo da diferença:

- `nunique()` diz quantos valores únicos existem. O resultado é um número.
- `value_counts()` diz quais são os valores únicos e quantas vezes cada um aparece. O resultado é uma série.

2.5 Plotagem básica

As visualizações são extremamente importantes em quase todos os passos do processamento de dados. Elas nos ajudam a identificar tendências nos dados quando estamos tentando entendê-los e limpá-los, além de contribuir para a apresentação de nossas descobertas finais.

Vamos observar as expectativas de vida anuais da população mundial novamente.

```
global_yearly_life_expectancy = df.groupby('year')['lifeExp'].mean()  
print(global_yearly_life_expectancy)
```

```
year  
1952    49.057620  
1957    51.507401  
1962    53.609249  
1967    55.678290  
1972    57.647386  
1977    59.570157  
1982    61.533197  
1987    63.212613  
1992    64.160338  
1997    65.014676  
2002    65.694923  
2007    67.007423  
Name: lifeExp, dtype: float64
```

Podemos usar o Pandas para criar algumas plotagens básicas.

```
global_yearly_life_expectancy.plot()
```

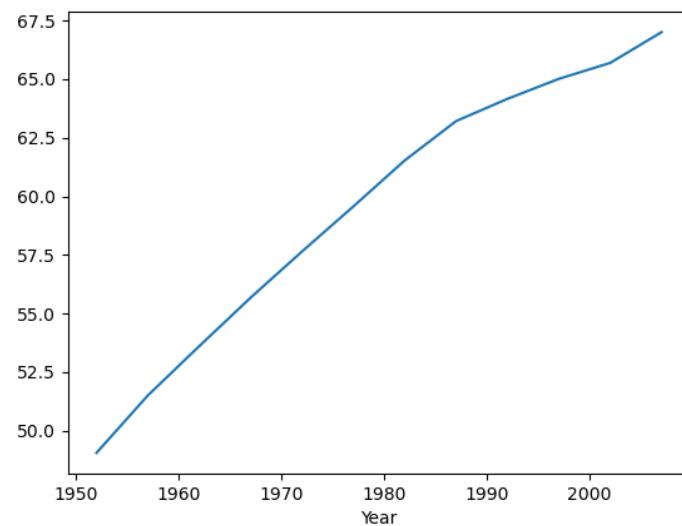


Figura 1: Plotagem básica no Pandas mostrando a expectativa de vida média no tempo.

2.6 Conclusão

Explicamos como carregar um conjunto de dados simples e começar a analisar observações específicas.

Tenha em mente que, quando fazemos análise de dados, o objetivo é gerar resultados reproduzíveis, sem fazer tarefas repetitivas. As linguagens de scripting lhe oferecem esses recursos e essa flexibilidade.

Nesse processo, conhecemos alguns dos recursos fundamentais de programação e as estruturas de dados que Python tem a nos oferecer.

Também vimos um modo rápido de obter estatísticas agregadas e fazer plotagens.

3 Estrutura de dados do Pandas

3.1 Introdução

O capítulo anterior apresentou os objetos `Dataframe` e `Series` do Pandas. Essas estruturas de dados assemelham-se aos contêineres de dados primitivos de Python (lista e dicionários, estruturas de dados básicas do Python que podem armazenar coleções de objetos) para indexação e rótulos, mas têm recursos adicionais que facilitam trabalhar com os dados.

Mapa Conceitual

1. Conhecimento prévio
 1. Contêineres (`list` e `dict`)
 2. Uso de funções
 3. Obtenção de subconjuntos e indexação
2. Carga de dados manual
3. `Series`
 1. Criando uma série
 - `dict`
 - `ndarray`
 - escalar
 - listas
 2. Fatiamento
4. `Dataframe`

3.2 Criando seus próprios dados

3.2.1 Criando uma Series

A `Series` do Pandas é um **contêiner unidimensional**, semelhante à `list` embutida de Python.

É o tipo de dado que representa cada coluna do `Dataframe`. A Tabela 2 lista os possíveis `dtypes` das colunas do `Dataframe` de Pandas. **Cada coluna em um dataframe deve ter o mesmo dtypes.**

Por ser possível pensar em `dataframe` como um dicionário de objetos `Series`, em que cada `key` é o nome da coluna e `value` é a `Series`, podemos concluir que uma `Series` é muito semelhante a uma `list` Python, exceto que todos os elementos devem ser do mesmo `dtype`. As pessoas que já usaram a biblioteca `numpy` perceberão que esse é o mesmo comportamento exibido por `ndarray`.

O modo mais fácil de criar uma `Series` é passando uma `list` Python.

Se passamos uma lista com tipos misturados, a representação mais comum será usada.

Em geral, `dtype` (tipo) da `Series` será um `object`.

```
import pandas as pd

s = pd.Series(['Banana', 42])
print(s)
```



```
0      Banana
1          42
dtype: object
```

Observe que o “número da linha” é exibido à esquerda. Na verdade, esse é o `index` da série. É semelhante ao nome e ao índice da linha que vimos sobre `dataframes`.

Isso implica que podemos atribuir realmente um “nome” aos valores de nossa série.

```
# Atribui valores de índice manualmente em uma série
# passando uma list Python
s = pd.Series(['Wes McKinney', 'Creator of Python'],
index=['Person', 'Who'])

print(s)
```

```
Person      Wes McKinney
Who        Creator of Python
dtype: object
```

Perguntas

- O que acontecerá se você usar outros contêineres Python como `list`, `tuple`, `dict` ou até mesmo o `ndarray` da biblioteca `numpy`?
- Quando você usa uma `list` (lista) ou `tuple` (tupla) para criar uma `Series`, o pandas simplesmente pega os elementos na ordem em que eles aparecem e os usa para popular a `Series`. O índice padrão é gerado automaticamente, começando do 0 e indo até `n-1`, onde `n` é o número de elementos.

```
print("#-----list-----#")
s = pd.Series(['Wes McKinney', 'Creator of Pandas'])
print(s)
```

```
#-----list-----#
0      Wes McKinney
1    Creator of Pandas
dtype: object
```

```
print("#-----tuple-----#")
s = pd.Series(('Wes McKinney', 'Creator of Pandas'))
print(s)
```

```
#-----tuple-----#
0      Wes McKinney
1    Creator of Pandas
dtype: object
```

- O `dict` é um caso especial e muito útil. Quando você cria uma `Series` a partir de um dicionário, o pandas usa as chaves do dicionário como o índice da `Series` e os valores como os dados. Isso permite que você crie uma `Series` já com rótulos significativos, o que é ótimo para dados categorizados.

```
print("#-----dict-----#")
dict_dados = {'a':100, 'b': 200, 'c':300}
s = pd.Series(dict_dados)
print(s)
```

```
#-----dict-----#
a    100
b    200
c    300
dtype: int64
```

- O `ndarray`² é o contêiner mais eficiente para o pandas. O pandas foi construído sobre o NumPy, então o `ndarray` é o formato subjacente de dados para a maioria das operações. Quando você usa um `ndarray` para criar uma Series, o processo é extremamente rápido, pois não há necessidade de converter o tipo de dado. O índice padrão também é gerado automaticamente.

```
print("#-----ndarray-----#")
numpy_dados = np.array([5,6,7])
s = pd.Series(numpy_dados)
print(s)
```

```
#-----ndarray-----#
0    5
1    6
2    7
dtype: int64
```

²`ndarray` é o nome oficial do tipo de dado (a classe) que o NumPy usa para representar arrays multidimensionais. `np.array()` é a função que você chama para criar uma instância (um objeto) dessa classe `ndarray`.

2. O que acontecerá se você passar um `index` com os contêineres?

- Quando os dados vêm de uma `list`, `tuple` ou `ndarray`, o pandas simplesmente combina os dados com o `index` fornecido. O pandas espera que o `index` tenha o mesmo número de elementos que o contêiner de dados.

```
index_dados = ['a','b','c']
dados_lista = [100,200,300]

print("#-----list-----#")
s = pd.Series(dados_lista,index=index_dados)
print(s)

#-----list-----
a    100
b    200
c    300
dtype: int64

index_dados = ['a','b','c']
dados_tupla = (100,200,300)

print("#-----tuple-----#")
s = pd.Series(dados_tupla,index=index_dados)
print(s)

#-----tuple-----
a    100
b    200
c    300
dtype: int64

index_dados = ['a','b','c']
dados_ndarray = np.array([100,200,300])

print("#-----ndarray-----#")
s = pd.Series(dados_ndarray,index=index_dados)
print(s)

#-----ndarray-----
a    100
b    200
c    300
dtype: int64
```

- Quando você passa um dict junto com um index, o pandas não usa as chaves do dicionário para criar o índice da Series. Em vez disso, ele usa o index fornecido para selecionar e reordenar os valores do dicionário.

```
index_dict = ['c','a','d','b']
dict_dados = {'a':100,'b': 200, 'c':300}

print("#-----dict-----#")
s = pd.Series(dict_dados,index=index_dict)
print(s)

#-----dict-----#
c    300.0
a    100.0
d      NaN
b    200.0
dtype: float64
```

3. Passar um `index` quando usamos um dict sobrescreverá o índice? Ou ele ordenará os valores?

O `index` que você passa ao criar a `Series` não sobrescreve os rótulos do dicionário. Em vez disso, ele ordena os valores e determina quais deles serão incluídos na `Series` final.

3.2.2 Criando um Dataframe

Conforme mencionamos podemos pensar em um `Dataframe` como um dicionário de objetos `Series`. É por isso que os dicionários são o modo mais comum de criar um `Dataframe`.

`key` representa o nome da coluna, enquanto os `values` são o conteúdo.

- `key` representa o nome da coluna;
- Os `values` são o conteúdo.

```
scientists = pd.DataFrame({  
    'Nome': ['Rosaline Franklin', 'William Gosset'],  
    'Occupation': ['Chemist', 'Statistician'],  
    'Born': ['1920-07-25', '1876-06-13'],  
    'Died': ['1958-04-16', '1937-10-16'],  
    'Age': [37, 61]  
})  
  
print(scientists)
```

| | Nome | Occupation | Born | Died | Age |
|---|-------------------|--------------|------------|------------|-----|
| 0 | Rosaline Franklin | Chemist | 1920-07-25 | 1958-04-16 | 37 |
| 1 | William Gosset | Statistician | 1876-06-13 | 1937-10-16 | 61 |

A ordem das colunas ao criar um DataFrame a partir de um dicionário não é garantida nas versões mais antigas do Python (anteriores ao 3.7). A partir do Python 3.7, a ordem de inserção dos elementos em dicionários é preservada.

Ordem das colunas e nome dos índices:

- Se consultarmos a documentação do Dataframe, veremos que é possível usar o parâmetro `columns` ou especificar a ordem das colunas. **Ordena as colunas**.
- Se quisermos usar colunas `name` para o índice da linha, podemos usar o parâmetro `index`. **Nomeia o índice**.

```
scientists = pd.DataFrame({  
    'Occupation': ['Chemist', 'Statistician'],  
    'Born': ['1920-07-25', '1876-06-13'],  
    'Died': ['1958-04-16', '1937-10-16'],  
    'Age': [37, 61]  
},  
    index=['Rosaline Franklin', 'William Gosset'],  
    columns=['Occupation', 'Born', 'Died', 'Age'])  
  
print(scientists)
```

| | Occupation | Born | Died | Age |
|-------------------|--------------|------------|------------|-----|
| Rosaline Franklin | Chemist | 1920-07-25 | 1958-04-16 | 37 |
| William Gosset | Statistician | 1876-06-13 | 1937-10-16 | 61 |

Antes do Python 3.7, os dicionários padrão (`dict`) não mantinham a ordem de inserção dos itens. Se você quisesse um dicionário que lembrasse a ordem em que os itens foram adicionados, precisava usar o `OrderedDict()`, classe do módulo `collections`.

```
from collections import OrderedDict
```

A partir do Python 3.7, os dicionários padrão passaram a manter a ordem de inserção por padrão. Isso significa que, na maioria dos casos, você não precisa mais usar o `OrderedDict` para essa finalidade.

Contudo, para efeito de estudo, segue o exemplo de uso do `OrderedDict`:

```
from collections import OrderedDict

# Observe os parênteses após OrderedDict
# Então passamos uma lista com duas tuplas

scientists = pd.DataFrame(OrderedDict([
    ('Nome', ['Rosaline Franklin','William Gosset']),
    ('Occupation',[ 'Chemist','Statistician']),
    ('Born',[ '1920-07-25','1876-06-13']),
    ('Died',[ '1958-04-16','1937-10-16']),
    ('Age',[37,61])
])
)

print(scientists)
```

| | Nome | Occupation | Born | Died | Age |
|---|-------------------|--------------|------------|------------|-----|
| 0 | Rosaline Franklin | Chemist | 1920-07-25 | 1958-04-16 | 37 |
| 1 | William Gosset | Statistician | 1876-06-13 | 1937-10-16 | 61 |

3.3 Series

Vimos como o método de fatiamento afeta o `type` do resultado. Se usarmos o atributo `loc` para gerar o subconjunto com a primeira linha de nosso `dataframe scientists`, obteremos um objeto `Series`.

Vamos recriar inicialmente o nosso `dataframe` de exemplo:

```
scientists = pd.DataFrame({  
    'Occupation': ['Chemist', 'Statistician'],  
    'Born': ['1920-07-25', '1876-06-13'],  
    'Died': ['1958-04-16', '1937-10-16'],  
    'Age': [37, 61]  
},  
    index=['Rosaline Franklin', 'William Gosset'],  
    columns=['Occupation', 'Born', 'Died', 'Age'])  
  
print(scientists)
```

| | Occupation | Born | Died | Age |
|-------------------|--------------|------------|------------|-----|
| Rosaline Franklin | Chemist | 1920-07-25 | 1958-04-16 | 37 |
| William Gosset | Statistician | 1876-06-13 | 1937-10-16 | 61 |

Agora selecionaremos um cientista pelo rótulo do índice da linha:

```
first_row = scientists.loc['William Gosset']  
  
print("\nTipo do objeto: ")  
print(type(first_row))  
  
print("\nObjeto: ")  
print(first_row)
```

```
Tipo do objeto:  
<class 'pandas.core.series.Series'>  
  
Objeto:  
Occupation      Statistician  
Born            1876-06-13  
Died            1937-10-16  
Age              61  
Name: William Gosset, dtype: object
```

Quando uma série é exibida (isto é, a sua representação em string), o índice é representado como a primeira “coluna”, e os valores são mostrados a segunda “coluna”. Há muitos atributos e métodos associados a um objeto **Series**.

Apresentação Objeto **Series**:

- Primeira coluna = índices (**index**)
- Segunda coluna = valores (**values**)

Dois exemplos de atributos são **index** e **values**:

- Atributo **index**:

```
# index é um atributo, não precisa de parênteses
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

- Atributo **values**:

```
# values é um atributo, não precisa de parênteses
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' np.int64(61)]
```

Um exemplo de um **método** de **Series** é **keys**, que é um alias (apelido) para o atributo **index**:

```
# keys é um método, precisa de parênteses
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

A essa altura, talvez você tenha perguntas sobre a sintaxe de `index`, `values` e `keys`.

- Podemos pensar nos **atributos** como propriedades de um objeto (nesse exemplo, nosso objeto é uma `Series`).
`index` e `values`.
- Podemos pensar nos **métodos** como um cálculo ou uma operação executada.
`keys`.

A sintaxe de subconjuntos para `loc`, `iloc` e `ix` é composta de todos os atributos. É por isso que essa sintaxe não depende de um conjunto de parênteses, `()`, mas de colchetes, `[]`.

A sintaxe de subconjuntos para os indexadores `loc`, `iloc` e `ix` no pandas é feita usando colchetes, `[]`. Isso ocorre porque eles são usados para a operação de indexação (seleção de dados), que é a sintaxe padrão de Python para esse fim, e não para a chamada de métodos, que usaria parênteses, `()`.

Como `keys` é um método, se quiséssemos obter a primeira chave (que é também o primeiro índice), usaríamos colchetes após a chamada do método.

```
# Obter o primeiro índice usando atributo index
print(first_row.index[0])
```

Occupation

```
# Obter a primeira key usando método keys
print(first_row.keys()[0])
```

Occupation

Alguns atributos de uma `Series` estão listados na Tabela 4.

Tabela 4: Alguns dos atributos de uma Series

| Atributo de Series | Descrição |
|---|--|
| <code>loc</code> | Subconjunto usando o valor de índice. |
| <code>iloc</code> | Subconjunto usando a posição de índice. |
| <code>ix</code> | Subconjunto usando valor e/ou posição de índice. |
| <code>dtype</code> ou <code>dtypes</code> | Tipos de conteúdo de <code>Series</code> . |
| <code>T</code> | Transposta da série. |
| <code>shape</code> | Dimensões dos dados. |
| <code>size</code> | Número de elementos em <code>Series</code> . |
| <code>values</code> | <code>ndarray</code> ou dado semelhante de <code>Series</code> . |

3.3.1 Series é semelhante a ndarray

- A estrutura de dados do Pandas conhecida como `Series` é muito semelhante ao `numpy.ndarray`.
- Por sua vez, muitos métodos e funções que atuam em um `numpy` funcionarão também em uma `Series`.
- As vezes, uma `Series` poderá ser referenciada como um “vetor”.

3.3.1.1 Métodos de Series

Vamos inicialmente obter uma série da coluna “Age” de nosso dataframe `scientists`.

```
# Obtém a coluna "Age"
ages = scientists['Age']
print(ages)
```

```
Rosaline Franklin    37
William Gosset       61
Name: Age, dtype: int64
```

O `numpy` é uma biblioteca de processamento científico que, em geral, lida com vetores numéricos. Como podemos pensar em uma `Series` como uma extensão de `numpy.ndarray`, há uma sobreposição de atributos e de métodos. Quando temos uma vetor de números, há cálculos comuns que podem ser executados.

Exemplos de métodos no Pandas:

- `mean()` - Média:

```
print(ages.mean())
```

49.0

- `min()` - Mínimo:

```
print(ages.min())
```

37

- `max()` - Máximo:

```
print(ages.max())
```

61

- `std()` - Desvio-padrão:

```
print(ages.std())
```

16.97056274847714

`mean`, `min`, `max` e `std` também são métodos em `numpy.ndarray`. Alguns métodos se Series estão listados na Tabela 5.

3.3.2 Subconjuntos com booleanos: Series

Podemos usar índices específicos para obter subconjuntos de nossos dados (como visto anteriormente). Apenas raramente, porém, saberemos o índice exato das linhas e colunas para obter um subconjunto dos dados. Em geral, você estará procurando valores que satisfaçam (ou não) a um cálculo ou uma observação em particular.

Tabela 5: Alguns métodos que podem ser executados em uma `Series`

| Métodos de <code>Series</code> | Descrição |
|--------------------------------|---|
| <code>append</code> | Concatena duas ou mais <code>Series</code> . |
| <code>corr</code> | Calcula uma correlação com outra <code>Series</code> .* |
| <code>cov</code> | Calcula uma covariância com outra <code>Series</code> .* |
| <code>describe</code> | Calcula estatísticas resumidas.* |
| <code>drop_duplicates</code> | Devolve uma <code>Series</code> sem duplicações. |
| <code>equals</code> | Determina se uma <code>Series</code> tem os mesmos elementos. |
| <code>get_values</code> | Obtém valores da <code>Series</code> ; o mesmo que o atributo <code>values</code> . |
| <code>hist</code> | Desenha um histograma. |
| <code>isin</code> | Verifica se valores estão contidos em uma <code>Series</code> . |
| <code>min</code> | Devolve o valor mínimo. |
| <code>max</code> | Devolve o valor máximo. |
| <code>mean</code> | Devolve a média aritmética. |
| <code>median</code> | Devolve a mediana. |
| <code>mode</code> | Devolve a(s) moda(s). |
| <code>quantile</code> | Devolve o valor em um dado quantil. |
| <code>replace</code> | Substitui valores da <code>Series</code> por um valor especificado. |
| <code>sample</code> | Devolve uma amostra aleatória de valores da <code>Series</code> . |
| <code>sort_values</code> | Ordena valores. |
| <code>to_frame</code> | Converte uma <code>Series</code> em um <code>DataFrame</code> . |
| <code>transpose</code> | Devolve a transposta. |
| <code>unique</code> | Devolve um <code>numpy.ndarray</code> de valores únicos. |

*Indica se valores ausentes serão automaticamente descartados.

Para explorar esse processo, vamos usar um conjunto de dados maior.

```
scientists = pd.read_csv('./Data/Cap_02/scientists.csv')
```

Acabamos de ver como podemos calcular métricas descritivas básicas de vetores.

O método `describe` calculará várias estatísticas descritivas com uma única chamada de método.

```
ages = scientists['Age']
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

- `describe` - Estatísticas básicas:

```
# Obtém estatísticas básicas
print(ages.describe())
```

```
count      8.000000
mean      59.125000
std       18.325918
min      37.000000
25%     44.000000
50%     58.500000
75%     68.750000
max      90.000000
Name: Age, dtype: float64
```

- `mean` - Média aritmética:

```
# Média de todas as idades
print(ages.mean())
```

```
59.125
```

E se quisermos obter o subconjunto de nossas idades identificando aquelas que estejam acima da média?

```
print(ages[ages > ages.mean()])
```

```
1    61
2    90
3    66
7    77
Name: Age, dtype: int64
```

Vamos analisar essa instrução e observar o que `ages > ages.mean()` devolve.

```
print(ages > ages.mean())
```

```
0    False
1    True
2    True
3    True
4    False
5    False
6    False
7    True
Name: Age, dtype: bool
```

```
print(type(ages > ages.mean()))
```

```
<class 'pandas.core.series.Series'>
```

Essa instrução devolve uma `Series` com `dtype` igual a `bool` (booleano, verdadeiro ou falso). Em outras palavras, podemos não só obter subconjunto de valores usando rótulos e índices, mas também especificar um vetor com valores booleanos.

Python tem muitas funções e métodos. Conforme o modo como estão implementados, eles poderão devolver rótulos, índices ou booleanos. Tenha esse ponto em mente quando conhecer novos métodos e tentar combinar várias partes em seu trabalho.

Se quisermos podemos fornecer manualmente um vetor de `bools` para obter um subconjunto de nossos dados.

```
# Obtém os índices 0, 1, 4, 5 e 7
manual_bool_values = [True, True, False, False, True, True, False, True]
print(ages[manual_bool_values])
```

```
0    37
1    61
4    56
5    45
7    77
Name: Age, dtype: int64
```

3.3.3 Operações são alinhadas e vetorizadas automaticamente (Broadcasting)

Se você não tem familiaridade com programação, acharia estranho que `ages > ages.mean()` devolva um vetor sem nenhum laço `for`.

Muitos dos métodos que funcionam em `Series` (e em `DataFrames` também) são vetorizados, o que significa que atuam em todo vetor simultaneamente.

Essa abordagem deixa o código mais legível e, em geral, há otimizações disponíveis para deixar os cálculos mais rápidos.

3.3.3.1 Vetores de mesmo tamanho

Se você executar uma operação entre dois vetores de mesmo tamanho, o vetor resultante será um cálculo feito com os vetores, elemento a elemento.

```
# Soma vetores do mesmo tamanho
print(ages + ages)
```

```
0      74
1     122
2     180
3     132
4     112
5      90
6      82
7     154
Name: Age, dtype: int64
```

```
# Multiplica vetores do mesmo tamanho
print(ages * ages)
```

```
0    1369
1    3721
2    8100
3    4356
4    3136
5    2025
6    1681
7    5929
Name: Age, dtype: int64
```

3.3.3.2 Vetores com inteiros (escalares)

Ao executar uma operação em um vetor usando um escalar, esse será usado em todos os elementos do vetor.

```
# Soma vetor e um escalar
print(ages + 100)
```

```
0    137
1    161
2    190
3    166
4    156
5    145
6    141
7    177
Name: Age, dtype: int64
```

```
# Multiplicação de um vetor por um escalar
print(ages * 2)
```

```
0    74
1   122
2   180
3   132
4   112
5    90
6    82
7   154
Name: Age, dtype: int64
```

3.3.3.3 Vetores com tamanhos diferentes

Quando estiver trabalhando com vetores de tamanhos diferentes, o comportamento dependerá do `type` dos vetores.

Em uma `Series`, os vetores executarão uma operação de acordo com o índice correspondente. O resto do vetor resultante será preenchido com um valor “ausente”, representado por `NaN`, que quer dizer “*Not a Number*” (não é um número).

Esse tipo de comportamento, é chamado de **Broadcasting**.

```
print(ages + pd.Series([1,100]))
```

```
0      38.0
1     161.0
2      NaN
3      NaN
4      NaN
5      NaN
6      NaN
7      NaN
dtype: float64
```

Com outros `types`, os formatos devem coincidir, ou retornará um erro.

```
import numpy as np

# Isto causará um erro
print(ages + np.array([1,100]))
```

3.3.3.4 Vetores com rótulos de índice comuns (alinhamento automático)

Um aspecto interessante no Pandas é o modo como o alinhamento de dados é quase sempre automático.

Se for possível, os dados sempre se alinharão de acordo com o rótulo do índice na execução de ações.

```
# ages conforme aparecem nos dados
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```
# ages invertendo a ordem dos índices
rev_ages = ages.sort_index(ascending=False)
print(rev_ages)
```

```
7    77
6    41
5    45
4    56
3    66
2    90
1    61
0    37
Name: Age, dtype: int64
```

Se executarmos uma operação usando `ages` e `rev_ages`, ela ainda será conduzida elemento a elemento, mas os vetores serão alinhados antes de a operação ser realizada.

```
# Saída de referência para mostrar o alinhamento dos rótulos de índice
print(ages * 2)
```

```
0      74
1     122
2     180
3     132
4     112
5      90
6      82
7     154
Name: Age, dtype: int64
```

```
# Observe que obtemos os mesmos valores
# apesar de o vetor estar invertido
print(ages + rev_ages)
```

```
0      74
1     122
2     180
3     132
4     112
5      90
6      82
7     154
Name: Age, dtype: int64
```

O índice é a referência para as operações e há um realinhamento (ordenamento).

3.4 Dataframe

O `DataFrame` é o objeto mais comum do Pandas. Podemos pensar nele como o modo Python de armazenar dados do tipo planilha.

Muitos dos recursos da estrutura de dados `Series` se aplicam ao `DataFrame`.

3.4.1 Subconjuntos com booleanos: DataFrames

Assim como pudemos obter um subconjunto de uma `Series` usando um vetor *booleano*, podemos obter um subconjunto de um `DataFrame` com um `bool`.

```
# Vetores booleanos servem para obter subconjuntos de linhas
print(scientists[scientists['Age'] > scientists['Age'].mean()])
```

| | Name | Born | Died | Age | Occupation |
|---|----------------------|------------|------------|-----|---------------|
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 61 | Statistician |
| 2 | Florence Nightingale | 1820-05-12 | 1910-08-13 | 90 | Nurse |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 66 | Chemist |
| 7 | Johann Gauss | 1777-04-30 | 1855-02-23 | 77 | Mathematician |

Quando você usa uma lista de booleanos para selecionar linhas em um `DataFrame` do Pandas, o número de valores `True` e `False` deve ser exatamente igual ao número total de linhas do `DataFrame`.

O conceito de *broadcasting* não se aplica aqui. A lista booleana atua como uma máscara de seleção, onde cada `True` ou `False` corresponde a uma linha específica. Se a máscara tiver um tamanho diferente do `DataFrame`, o Pandas não saberá quais linhas incluir ou ignorar, e isso resultará em um erro (`IndexError`).

```
# 8 valores passados como um vetor booleanos
# 3 linhas devolvidas
print(scientists.loc[[True, True, False, True, False, False, False]])
```

| | Name | Born | Died | Age | Occupation |
|---|-------------------|------------|------------|-----|--------------|
| 0 | Rosaline Franklin | 1920-07-25 | 1958-04-16 | 37 | Chemist |
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 61 | Statistician |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 66 | Chemist |

A Tabela 6 resumo os diversos tipos de métodos para obtenção de subconjuntos.

Tabela 6: Tabela de métodos para obtenção de subconjuntos de DataFrame

| Sintaxe | Resultado da seleção |
|---|---|
| <code>df[column_name]</code> | Única coluna. |
| <code>df[[column1,column2,...]]</code> | Várias colunas. |
| <code>df.loc[row_label]</code> | Linha pelo rótulo do índice da linha (nome da linha). |
| <code>df.loc[[label1,label2,...]]</code> | Várias linhas pelos rótulos do índice. |
| <code>df.iloc[row_number]</code> | Linha pelo número da linha. |
| <code>df.iloc[[row1,row2,...]]</code> | Várias linhas pelos números das linhas. |
| <code>df.ix[label_or_number]</code> | Linha pelo rótulo do índice ou pelo número. |
| <code>df.ix[lab_num1,lab_num2,...]</code> | Várias linhas pelos rótulos de índice ou pelos números. |
| <code>df[bool]</code> | Linha baseada em <code>bool</code> . |
| <code>df[bool1,bool2,...]</code> | Várias linhas baseadas em <code>bool</code> . |
| <code>df[start:stop:step]</code> | Linhas baseadas em notação de fatiamento. |

*Observe que `ix` não funciona mais depois do Pandas v0.20.

3.4.2 Operações são alinhadas e vetorizadas automaticamente (*Broadcasting*)

O Pandas aceita *broadcasting*, disponibilizado pela biblioteca `numpy`. Essencialmente, ele descreve o que acontece quando realizamos operações entre objetos do tipo array, que é o caso de `Series` e `DataFrame`. Esses comportamentos dependem do tipo do objeto, de seu tamanho e de qualquer rótulo associado a ele.

Inicialmente, vamos criar subconjuntos de nosso dataframe.

```
first_half = scientists[:4]
second_half = scientists[4:]
```

```
print(first_half)
```

| | Name | Born | Died | Age | Occupation |
|---|----------------------|------------|------------|-----|--------------|
| 0 | Rosaline Franklin | 1920-07-25 | 1958-04-16 | 37 | Chemist |
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 61 | Statistician |
| 2 | Florence Nightingale | 1820-05-12 | 1910-08-13 | 90 | Nurse |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 66 | Chemist |

```
print(second_half)
```

| | Name | Born | Died | Age | Occupation |
|---|---------------|------------|------------|-----|--------------------|
| 4 | Rachel Carson | 1907-05-27 | 1964-04-14 | 56 | Biologist |
| 5 | John Snow | 1813-03-15 | 1858-06-16 | 45 | Physician |
| 6 | Alan Turing | 1912-06-23 | 1954-06-07 | 41 | Computer Scientist |
| 7 | Johann Gauss | 1777-04-30 | 1855-02-23 | 77 | Mathematician |

Quando executamos uma ação em um dataframe com um escalar, há uma tentativa de aplicar a operação em cada célula do dataframe.

Escalar

Nesse exemplo, os números serão multiplicados por 2 e as *strings* serão **duplicadas** (esse é o comportamento usual do Python com *strings*).

```
# Multiplicar por um escalar
print(scientists * 2)
```

| | Name | Born | \ |
|---|----------------------|------------|------------|
| 0 | Rosaline Franklin | 1920-07-25 | 1920-07-25 |
| 1 | William Gosset | 1876-06-13 | 1876-06-13 |
| 2 | Florence Nightingale | 1820-05-12 | 1820-05-12 |
| 3 | Marie Curie | 1867-11-07 | 1867-11-07 |
| 4 | Rachel Carson | 1907-05-27 | 1907-05-27 |
| 5 | John Snow | 1813-03-15 | 1813-03-15 |
| 6 | Alan Turing | 1912-06-23 | 1912-06-23 |
| 7 | Johann Gauss | 1777-04-30 | 1777-04-30 |

| | Died | Age | Occupation |
|---|------------|-----|--------------------|
| 0 | 1958-04-16 | 74 | Chemist |
| 1 | 1937-10-16 | 122 | Statistician |
| 2 | 1910-08-13 | 180 | Nurse |
| 3 | 1934-07-04 | 132 | Chemist |
| 4 | 1964-04-14 | 112 | Biologist |
| 5 | 1858-06-16 | 90 | Physician |
| 6 | 1954-06-07 | 82 | Computer Scientist |
| 7 | 1855-02-23 | 154 | Mathematician |

Somar - método .add()

Se seus dataframes tiverem somente valores numéricos e você quiser “somar” os valores célula a célula, o método `add` poderá ser usado.

O método `.add()` em pandas é uma maneira muito útil de somar dataframes elemento a elemento. Ele é especialmente flexível porque, ao contrário do operador `+`, ele tem um parâmetro chamado `fill_value`.

Esse parâmetro é extremamente útil quando os dataframes têm índices (linhas) ou colunas diferentes. Se uma célula em um dataframe não tiver uma “correspondente” no outro dataframe, o `fill_value` será usado para preencher o valor ausente antes de fazer a soma. Isso evita que o resultado seja `NaN` (*Not a Number*) para essas células, que é o comportamento padrão do operador `+`.

```
# DataFrame 1
df1 = pd.DataFrame({
    'A': [10, 20, 30],
    'B': [40, 50, 60]
}, index=['X', 'Y', 'Z'])

print("DataFrame 1:")
print(df1)
```

DataFrame 1:

| | A | B |
|---|----|----|
| X | 10 | 40 |
| Y | 20 | 50 |
| Z | 30 | 60 |

```
# DataFrame 2
df2 = pd.DataFrame({
    'A': [5, 10, 15],
    'C': [2, 4, 6] # Note a coluna 'C', que não existe em df1
}, index=['X', 'Y', 'W']) # Note a linha 'W', que não existe em df1
```

```
print("DataFrame 2:")
print(df2)
```

DataFrame 2:

| | A | C |
|---|----|---|
| X | 5 | 2 |
| Y | 10 | 4 |
| W | 15 | 6 |

```
# Usando .add() com fill_value
# Preenche os valores ausentes com 0 para que a soma ocorra
df_soma = df1.add(df2, fill_value=0)

print("Resultado da soma com .add(fill_value=0):")
print(df_soma)
```

Resultado da soma com .add(fill_value=0):

| | A | B | C |
|---|------|------|-----|
| W | 15.0 | NaN | 6.0 |
| X | 15.0 | 40.0 | 2.0 |
| Y | 30.0 | 50.0 | 4.0 |
| Z | 30.0 | 60.0 | NaN |

3.5 Fazendo alterações em Series e em Dataframe

Agora que já conhecemos várias maneiras de obter subconjuntos e fatiar nossos dados (veja Tabela 6), podemos alterar nossos objetos de dados.

3.5.1 Adicionando mais colunas

O type das colunas Born e Died é object, ou seja são strings.

```
print(scientists['Born'].dtype)
```

object

```
print(scientists['Died'].dtype)
```

object

É possível converter as strings em um tipo `datetime` apropriado para que possamos executar operações comuns de data e hora (por exemplo, obter as diferenças entre datas ou calcular a idade de uma pessoa).

Você pode fornecer o seu próprio `format` caso tenha uma data com um formato específico. Uma lista de variáveis `format` pode ser encontrada na documentação do módulo `datetime` de Python.

O formato da data que vamos trabalhar tem o aspecto “AAAA-MM-DD”, portanto podemos usar o formato ‘%Y-%m-%d’.

```
# Formata a coluna 'Born' como datetime
born_datetime = pd.to_datetime(scientists['Born'], format='%Y-%m-%d')
print(born_datetime)
```

```
0    1920-07-25
1    1876-06-13
2    1820-05-12
3    1867-11-07
4    1907-05-27
5    1813-03-15
6    1912-06-23
7    1777-04-30
Name: Born, dtype: datetime64[ns]
```

```
died_datetime = pd.to_datetime(scientists['Died'], format='%Y-%m-%d')
print(died_datetime)
```

```
0    1958-04-16
1    1937-10-16
2    1910-08-13
3    1934-07-04
4    1964-04-14
5    1858-06-16
6    1954-06-07
7    1855-02-23
Name: Died, dtype: datetime64[ns]
```

Se quiséssemos, poderíamos criar um novo conjunto de colunas contendo as representações como `datetime` das datas que são `object` (string).

O exemplo a seguir usa a sintaxe de atribuição múltipla do Python.

```
scientists['born_dt'], scientists['died_dt'] = (born_datetime, died_datetime)
print(scientists.head())
```

| | Name | Born | Died | Age | Occupation | born_dt | \ |
|---|----------------------|------------|------------|-----|--------------|------------|---|
| 0 | Rosaline Franklin | 1920-07-25 | 1958-04-16 | 37 | Chemist | 1920-07-25 | |
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 61 | Statistician | 1876-06-13 | |
| 2 | Florence Nightingale | 1820-05-12 | 1910-08-13 | 90 | Nurse | 1820-05-12 | |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 66 | Chemist | 1867-11-07 | |
| 4 | Rachel Carson | 1907-05-27 | 1964-04-14 | 56 | Biologist | 1907-05-27 | |

```
      died_dt
0 1958-04-16
1 1937-10-16
2 1910-08-13
3 1934-07-04
4 1964-04-14
```

```
print(scientists.shape)
```

(8, 7)

3.5.2 Alterando diretamente uma coluna

Podemos também atribuir um novo valor diretamente a uma coluna existente.

O exemplo nesta seção mostra como deixar aleatório o conteúdo de uma coluna.

Inicalmente, vamos observar os valores originais de Age.

```
print(scientists['Age'])
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

Vamos agora embaralhar os valores.

```
import random

# Define uma semente (seed) para que a aleatoriedade seja sempre igual
random.seed(42)
random.shuffle(scientists['Age'])

print(scientists['Age'])
```

A mensagem `SettingWithCopyWarning` no código anterior nos informa que o modo apropriado de lidar com a instrução seria escrevê-la usando `loc`, ou podemos usar o método embutido `sample` para mostrar aleatoriamente o tamanho da coluna.

Neste exemplo, é necessário executar `reset_index`, pois `sample` usa somente índice da linha. Desse modo, se você tentar atribuir-lhe um novo valor ou usá-lo novamente, os valores “embaralhados” serão automaticamente alinhados ao índice e serão ordenados de novo como eram antes de `sample`.

O parâmetro `drop=True` em `reset_index` informa ao Pandas que não insira o índice nas colunas do dataframe, de modo que somente os valores sejam mantidos.

```
import random

# random_state é usado para deixar a 'aleatoriedade' menos aleatória
scientists['Age'] = scientists['Age'].\
    sample(len(scientists['Age']),random_state=24).\
    reset_index(drop=True) #Valores permanecem aleatórios

print(scientists['Age'])
```

```
0    77
1    45
2    41
3    56
4    61
5    37
6    66
7    90
Name: Age, dtype: int64
```

Observe que o método `random.shuffle` parece atuar diretamente na coluna. A documentação de `random.shuffle` menciona que a sequência será embaralhada “*in place*”, o que significa que ela atuará diretamente na sequência. Compare isso com o método anterior, em que atribuímos os valores novos calculados a uma variável diferente antes que pudéssemos atribuí-los à coluna.

- `len(scientists['Age'])`: Este argumento diz à função `sample()` para pegar uma amostra do mesmo tamanho que a coluna inteira. Em outras palavras, ele está pedindo para selecionar todos os valores da coluna, mas em uma ordem aleatória.
- `random_state=24`: Este é o equivalente a `random.seed()`. Usar `random_state` garante que a “aleatoriedade” seja reproduzível. Se você executar o código várias vezes com `random_state=24`, o embaralhamento será sempre o mesmo. Isso é extremamente útil para depuração e para garantir que seus resultados sejam consistentes.
- `reset_index()`: Reseta o índice da Series (ou DataFrame) para a ordem padrão (0, 1, 2, ...).
- `drop=True`: Este argumento é muito importante. Ele diz ao `reset_index()` para desconsiderar o índice antigo. Se você não usar `drop=True`, o pandas adicionará o índice antigo como uma nova coluna no seu DataFrame, o que não é o que queremos aqui.

- “*in-place*” (no lugar): significa que uma função modifica o objeto original diretamente, sem criar uma nova cópia.

Podemos recalcular a idade “real” usando aritmética com `datetime`.

```
# Subtrair datas nos dá o número de dias
scientists['age_days_dt'] = (scientists['died_dt'] - \
    scientists['born_dt'])

print(scientists)
```

| | Name | Born | Died | Age | Occupation |
|---|----------------------|------------|------------|-----|--------------------|
| 0 | Rosaline Franklin | 1920-07-25 | 1958-04-16 | 77 | Chemist |
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 45 | Statistician |
| 2 | Florence Nightingale | 1820-05-12 | 1910-08-13 | 41 | Nurse |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 56 | Chemist |
| 4 | Rachel Carson | 1907-05-27 | 1964-04-14 | 61 | Biologist |
| 5 | John Snow | 1813-03-15 | 1858-06-16 | 37 | Physician |
| 6 | Alan Turing | 1912-06-23 | 1954-06-07 | 66 | Computer Scientist |
| 7 | Johann Gauss | 1777-04-30 | 1855-02-23 | 90 | Mathematician |

| | born_dt | died_dt | age_days_dt |
|---|------------|------------|-------------|
| 0 | 1920-07-25 | 1958-04-16 | 13779 days |
| 1 | 1876-06-13 | 1937-10-16 | 22404 days |
| 2 | 1820-05-12 | 1910-08-13 | 32964 days |
| 3 | 1867-11-07 | 1934-07-04 | 24345 days |
| 4 | 1907-05-27 | 1964-04-14 | 20777 days |
| 5 | 1813-03-15 | 1858-06-16 | 16529 days |
| 6 | 1912-06-23 | 1954-06-07 | 15324 days |
| 7 | 1777-04-30 | 1855-02-23 | 28422 days |

```

# Podemos converter o valor somente para o ano
# usando o método astype
scientists['age_years_dt'] = scientists['age_days_dt'].\
    astype('timedelta64[Y]')
print(scientists)

```

Correção:

o método `astype('timedelta64[Y]')` nunca funcionou para converter dias em anos dessa forma. O problema principal é a variabilidade do ano. Diferente de um dia (24 horas) ou uma hora (60 minutos), um ano não tem uma duração fixa e precisa em dias. Ele pode ter 365 ou 366 dias (nos anos bissextos).

O tipo de dado `timedelta64` foi projetado para lidar com diferenças de tempo exatas e fixas, como dias, horas, segundos, milissegundos, etc. Tentar usar uma unidade como “ano” (Y) ou “mês” (M) não seria preciso, porque o pandas não saberia se deve usar 365 ou 366 dias na conversão.

```

scientists['age_years_dt'] = scientists['age_days_dt'].\
    dt.total_seconds() / (365.25 * 24 * 60 * 60)

print(scientists)

```

| | Name | Born | Died | Age | Occupation |
|---|----------------------|------------|------------|-----|--------------------|
| 0 | Rosaline Franklin | 1920-07-25 | 1958-04-16 | 77 | Chemist |
| 1 | William Gosset | 1876-06-13 | 1937-10-16 | 45 | Statistician |
| 2 | Florence Nightingale | 1820-05-12 | 1910-08-13 | 41 | Nurse |
| 3 | Marie Curie | 1867-11-07 | 1934-07-04 | 56 | Chemist |
| 4 | Rachel Carson | 1907-05-27 | 1964-04-14 | 61 | Biologist |
| 5 | John Snow | 1813-03-15 | 1858-06-16 | 37 | Physician |
| 6 | Alan Turing | 1912-06-23 | 1954-06-07 | 66 | Computer Scientist |
| 7 | Johann Gauss | 1777-04-30 | 1855-02-23 | 90 | Mathematician |

| | born_dt | died_dt | age_days_dt | age_years_dt |
|---|------------|------------|-------------|--------------|
| 0 | 1920-07-25 | 1958-04-16 | 13779 days | 37.724846 |
| 1 | 1876-06-13 | 1937-10-16 | 22404 days | 61.338809 |
| 2 | 1820-05-12 | 1910-08-13 | 32964 days | 90.250513 |
| 3 | 1867-11-07 | 1934-07-04 | 24345 days | 66.652977 |
| 4 | 1907-05-27 | 1964-04-14 | 20777 days | 56.884326 |
| 5 | 1813-03-15 | 1858-06-16 | 16529 days | 45.253936 |
| 6 | 1912-06-23 | 1954-06-07 | 15324 days | 41.954825 |
| 7 | 1777-04-30 | 1855-02-23 | 28422 days | 77.815195 |

Muitas funções e métodos do Pandas terão um parâmetro `inplace` que poderá ser definida com `True` caso você queira executar a ação “*in place*”. Isso fará com que a dada coluna seja alterada diretamente, sem devolver nada.

3.5.3 Descartando valores

Para descartar uma coluna, podemos selecionar todas as colunas que queremos usando as técnicas de obtenção de subconjuntos de colunas ou selecionar as colunas a serem descartadas com o método `drop` de nosso dataframe.

```
# Todas as colunas de nossos dados no momento
print(scientists.columns)

Index(['Name', 'Born', 'Died', 'Age', 'Occupation', 'born_dt', 'died_dt',
       'age_days_dt', 'age_years_dt'],
      dtype='object')

# Descarta a coluna de idade embaralhada
# Especifique o argumento axis=1 para descartar toda a coluna
scientists_dropped = scientists.drop(['Age'], axis=1)

# Colunas após descartar a nossa coluna
print(scientists_dropped.columns)

Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt', 'died_dt',
       'age_days_dt', 'age_years_dt'],
      dtype='object')
```

O argumento `axis=1` (ou `axis='columns'`) indica ao Pandas para procurar e remover nos eixos verticais (colunas). Se você quisesse remover uma linha, usaria `axis=0` (ou `axis='index'`).

3.6 Exportando e importando dados

Em nossos exemplos até agora, importamos dados. Exportar e salvar conjuntos de dados enquanto os processamos também é uma prática comum. Conjuntos de dados são salvos como versões finais limpas ou como passos intermediários. As duas saídas podem ser usadas para análise ou como entrada para outra parte do fluxo de processamento de dados.

3.6.1 pickle

O Python tem uma maneira de executar `pickle` de dados. É o modo Python de serializar e salvar dados em formato binário.

A leitura de dados serializados também é uma operação compatível com versões anteriores.

3.6.1.1 Series

Muitos dos métodos de exportação para `Series` também estão disponíveis para `DataFrame`.

Os leitores que tiverem experiência com o `numpy` saberão que um método `save` está disponível para `ndarrays`.

Esse método (`save`) foi considerado obsoleto, e o método `to_pickle` deve ser usado como substituto.

```
names = scientists['Name']
print(names)

0      Rosaline Franklin
1      William Gosset
2    Florence Nightingale
3        Marie Curie
4      Rachel Carson
5        John Snow
6       Alan Turing
7      Johann Gauss
Name: Name, dtype: object

# Passar uma string como o path para salvar
names.to_pickle('../output/scientists_names_series.pickle')
```

A saída de `pickle` tem **formato binário**. Assim, se você tentar abri-la em um editor de texto, verá um conjunto de caracteres confusos.

Se o objeto que você estiver salvando for um passo intermediário em um conjunto de processamentos que você queira guardar, ou se souber que seus dados permanecerão no mundo Python, a operação de salvar um objeto em um `pickle` será **otimizada** para o Python, assim como no que concerne ao **espaço de armazenamento em disco**.

No entanto, essa abordagem implica que as pessoas que não usam Python não poderão ler os dados.

3.6.1.2 DataFrame

O mesmo método pode ser usado em objetos `DataFrame`.

```
scientists.to_pickle('../output/scientists_df.pickle')
```

3.6.1.3 Lendo dados de pickle

Para ler dados de pickle, podemos usar a função `pd.read_pickle`.

- Series:

```
# Para Series  
scientist_names_from_pickle = pd.read_pickle('../output/scientists_names_series.pickle')
```

- DataFrame:

```
# Para DataFrame  
scientists_from_pickle = pd.read_pickle('../output/scientists_df.pickle')
```

Os arquivos pickle são salvos em extensão `.p`, `.pkl` ou `.pickle`.

3.6.2 CSV

O formato CSV (*Comma-Separated Values*, Valores separados por vírgula) é o tipo mais flexível para armazenagem de dados.

Para cada linha, as informações das colunas são separadas com uma vírgula. Porém a vírgula não é o único tipo de delimitador. Alguns arquivos são delimitados com uma tabulação (TSV) ou até mesmo por ponto e vírgula.

O principal motivo para o CSV ser um formato de dados preferível para colaboração e com-partilhamento de dados deve-se ao fato de qualquer programa ser capaz de abrir esse tipo de estrutura de dados. Ele pode ser aberto até mesmo em um editor de texto.

- Parâmetro `sep`
 - Vírgula (default)
Por default é a opção do sistema, não precisa ser declarado (explicitado).
 - Ponto e vírgula (`sep=';'`)
Normalmente usado em alternativa ao separador com vírgula, principalmente em dados brasileiros, por conta dos números que usam vírgula.
 - Tabulação (`sep='\t'`)
Muito usado para evitar mal-entendidos caso algum tipo de dado use vírgula como separador.

3.6.2.1 Exportando dados CSV

`Series` e `DataFrame` têm um método `.to_csv()` para escrever um arquivo CSV. A documentação de `Series` e de `DataFrame` identifica muitos modos diferentes pelos quais podemos modificar o arquivo CSV resultante. Por exemplo, se quiser salvar um arquivo TSV porque há vírgulas em seus dados, você poderá alterar o parâmetro `sep`.

- `Series`:

```
# Salva (exporta) uma série em um CSV
names.to_csv('../output/scientist_name_series.csv')
```

- `DataFrame`:

```
# Salva (exporta) um DataFrame em um TSV
# que significa Tab-Separated Value (Valor Separado por Tabulação)
scientists.to_csv('../output/scientist_name_series.tsv', sep='\t')
```

3.6.2.2 Removendo os números das linhas de saída

Se você abrir o arquivo CSV ou TSV criado, perceberá que a primeira “coluna” parece ser o número de linhas do `DataFrame`. Muitas vezes ela não será necessária especialmente se você estiver trabalhando em colaboração com outras pessoas.

Tenha em mente que essa “coluna” está realmente salvando o “rótulo da linha”, que poderá ser importante.

A documentação mostrará que há um parâmetro `index` com o qual podemos escrever os nomes das linhas.

- Parâmetro `index`

- True (`index=True`)

Adiciona o rótulo das linhas do arquivo CSV.

```
# Escreve os nomes das linhas na saída CSV
scientists.to_csv('../output/scientist_name_series.tsv', index=True)
```

- False (`index=False`)

Exclui o rótulo das linhas do arquivo CSV.

```
# Não escreve os nomes das linhas na saída CSV
scientists.to_csv('../output/scientist_name_series.tsv', index=False)
```

3.6.2.3 Importando dados CSV

Importação de arquivos CSV. Essa operação usa a função do Pandas `pd.read_csv()`.

```
import pandas as pd

nome_do_dataframe = pd.read_csv('caminho/para/seu/arquivo.csv')
```

3.6.3 Excel

O Excel, que provavelmente é o tipo de dado mais comum usado (ou o segundo mais comum, ficando depois dos CSVs), tem uma reputação ruim na comunidade de ciência de dados, principalmente porque informações sobre cores e outros supérfluos podem facilmente ser incluídos no conjunto de dados, sem mencionar os cálculos feitos uma só vez, que podem arruinar a estrutura retangular de um conjunto de dados.

O objetivo não é atacar o Excel, mas mostar uma ferramenta alternativa razoável para análise de dados. Em suma, quanto mais de seu trabalho você puder fazer com uma linguagem de scripting, mais fácil será escalar em direção a projetos maiores, identificar e corrigir erros e promover colaboração. Entretanto não há rivais para a popularidade e a parcela do mercado do Excel. O Excel tem a sua própria linguagem scripting caso você tenha realmente que trabalhar com ele. Isso permitirá que você trabalhe com os dados de uma maneira mais previsível e reproduzível.

3.6.3.1 Series

A estrutura de dados `Series` não tem um método `to_excel` explícito.

Se você tiver uma `Series` que precise ser exportada para um arquivo Excel, uma opção é converter a `Series` em um `DataFrame` de uma só coluna.

```
# Converte a Series em um DataFrame
# antes de salvá-la em um arquivo Excel
names_df = names.to_frame()

import xlwt
# arquivo .xls
names_df.to_excel('../output/scientists_names_series_df.xls')

import openpyxl
# arquivo .xlsx
names_df.to_excel('../output/scientists_names_series_df.xlsx')
```

3.6.3.2 DataFrame

A documentação mostra várias maneiras de ajustar melhor a saída. Por exemplo, os dados podem estar em uma “planilha” específica usando o parâmetro `sheet_name`.

```
import openpyxl
# arquivo .xlsx

# Salvando um DataFrame em formato Excel
scientists.to_excel('../output/scientists_names_series_df.xlsx',
    sheet_name='scientists',
    index=False)
```

3.6.3.3 Bibliotecas que converte DataFrame em Excel

As bibliotecas que fazem a conversão de DataFrame para Excel são:

- `xlwt`
- `openpyxl`

Tabela 7: Principais bibliotecas de conversão de dados para Excel

| Biblioteca | Formato de Arquivo (Extensão) | Uso Principal |
|-----------------------|---|--|
| <code>xlwt</code> | .xls (Formato antigo do Excel 97-2003) | Apenas escrever dados e formatação em arquivos antigos. |
| <code>openpyxl</code> | .xlsx (Formato moderno do Excel 2007+) | Ler e escrever dados, formatação, fórmulas e gráficos nos formatos modernos. |

3.6.3.4 Para leitura de arquivo Excel no Python

Se ainda não tiver, você precisará instalar o pandas e uma biblioteca que o pandas usa para ler arquivos Excel mais antigos (como .xls), que é o `xlrd`.

```
pip install pandas xlrd
```

```
import pandas as pd

# Substitua 'seu_arquivo.xls' pelo nome e caminho real do seu arquivo
caminho_do_arquivo = 'seu_arquivo.xls'

# Abrindo o arquivo XLS como um DataFrame
# Por padrão, ele lê a primeira aba (sheet)
df = pd.read_excel(caminho_do_arquivo)

# Para verificar as primeiras linhas do DataFrame
print(df.head())
```

Para arquivos .xlsx (mais novos): O `pd.read_excel()` funciona perfeitamente também. Você pode precisar do motor `openpyxl` em vez do `xlrd`.

Especificando uma aba (`sheet`): Se o seu arquivo tiver várias abas e você quiser uma específica, você pode usar o argumento `sheet_name`.

```
pip install pandas openpyxl
```

```
# Lê a aba chamada "Dados_Fevereiro"
df = pd.read_excel(caminho_do_arquivo, sheet_name='Dados_Fevereiro')

# Ou lê a segunda aba (índice 1, pois começa em 0)
df = pd.read_excel(caminho_do_arquivo, sheet_name=1)
```

3.6.4 Formato feather para interface com R

O formato chamado “**feather**” é usado para salvar um objeto binário que possa ser carregado também na linguagem R. A principal vantagem dessa abordagem é que ela é mais rápida que escrever e ler um arquivo CSV entre as linguagens.

A regra geral para usar esse formato de dados é usá-lo somente como intermediário, sem utilizar o formato **feather** para armazenagem de longo prazo. Ou seja, use-o em seu código somente para passar dados para R, e não para salvar uma versão final de seus dados.

Instalando o formato de **feather**:

- No Anaconda:

```
conda install -c conda-forge feather-format
```

- No pip:

```
pip install feather-format
```

O método `to_feather` pode ser usado em um `DataFrame` para salvar o objeto **feather**.

Nem todo `DataFrame` pode ser convertido em um objeto **feather**. Por exemplo, o nosso conjunto atual de dados contém uma coluna de valores `date`; ate o presente momento, esse tipo de valor não era aceito por **feather**.

A recomendação mais moderna e preferível é instalar o `PyArrow` ao invés de `feather-format`.

Instalação:

- No Anaconda:

```
conda install -c conda-forge pyarrow
```

- No pip:

```
pip install pyarrow
```

Você não precisa chamar a biblioteca `pyarrow` diretamente na maioria das vezes para usar o formato **Feather**. Basta chamar apenas o Pandas.

Importar e exportar dados em **feather**:

- Escrever (exportar) dados em **feather**:

```
df.to_feather('arquivo.feather')
```

- Ler (importar) dados em **feather**:

```
pd.read_feather('arquivo.feather')
```

3.6.5 Outros tipos de saída de dados

Há muitas maneiras pelas quais o Pandas pode exportar e importar dados. Na verdade, `to_pickle`, `to_csv`, `to_excel` e `to_feather` são apenas alguns dos formatos de dados que podem ser usados com `DataFrames` no Pandas.

A Tabela 8 lista alguns desses outros formatos de saída.

Tabela 8: Métodos de DataFrame para exportação

| Métodos para exportação | Descrição |
|---------------------------|---|
| <code>to_clipboard</code> | Salva dados na área de transferência (clipboard) do sistema para colar. |
| <code>to_dense</code> | Converte dados em um <code>DataFrame</code> regular “denso”*. |
| <code>to_dict</code> | Converte dados em um <code>dict</code> (dicionário) Python. |
| <code>to_gbq</code> | Converte dados em uma tabela Google BigQuery. |
| <code>to_hdf</code> | Salva dados em HDF (Hierarchical Data Format). |
| <code>to_msgpack</code> | Salva dados em um binário portável do tipo JSON. |
| <code>to_html</code> | Converte dados em uma tabela HTML. |
| <code>to_json</code> | Converte dados em uma string JSON. |
| <code>to_latex</code> | Converte dados em um ambiente tabular LATEX. |
| <code>to_records</code> | Converte dados em um array de registros. |
| <code>to_string</code> | Exibe o <code>DataFrame</code> como um string para stdout. |
| <code>to_sparse</code> | Converte dados em um <code>SparseDataFrame</code> **. |
| <code>to_sql</code> | Salva dados em um banco de dados SQL. |
| <code>to_stata</code> | Converte dados em um arquivo Stata <code>dta</code> . |

*Estruturas Densas (Regulares): São a forma padrão de um `DataFrame`, onde memória é alocada para todas as células, independentemente de conterem um valor ou o valor padrão (zero/vazio).

**Estruturas Esparsas: São aquelas que otimizam o uso de memória armazenando apenas os valores que são diferentes do valor padrão (o valor esparsa). Se 99% das células são zero, a estrutura esparsa só armazena as 1% de células que não são zero, economizando muito espaço.

No que diz respeito as versões de dados mais complicadas e genéricas (não necessariamente apenas para exportar dados), a biblioteca `odo` tem um modo consistente de fazer conversões entre formatos de dados.

O `odo` foi projetado para migrar dados de forma eficiente de uma fonte para um destino, geralmente com apenas uma linha de código.

- Instalando `odo`:

```
pip install odo
```

- Importando biblioteca:

```
from odo import odo
```

- Sintaxe simplificada:

```
odo(fonte, destino) # Carrega a fonte no destino
```

- Exemplo:

```
dataframe_resultado = odo(file_path, pd.DataFrame)
```

4 Introdução à plotagem

4.1 Introdução

A visualização de dados faz parte do passo tanto do processamento quanto da apresentação dos dados. É muito mais fácil comparar valores quando estes são plotados do que comparar valores numéricos. Ao visualizar dados, podemos ter uma noção mais intuitiva deles em comparação a observar somente tabelas de valores. Além do mais, as visualizações podem trazer à tona padrões ocultos nos dados, que você, o analista, poderá explorar para selecionar um modelo.

Mapa Conceitual

1. Conhecimento prévio
 1. Contêineres
 2. Uso de funções
 3. Obtenção de subconjuntos e indexação
 4. Classes
2. `matplotlib`
3. `seaborn`
4. Pandas

Objetivos

Este capítulo abordará:

1. `matplotlib`
2. `seaborn`
3. Plotagem no Pandas

O exemplo quintessencial para criar visualizações de dados é o quarteto de Anscombe. Esse conjunto de dados foi criado pelo estatístico inglês Frank Anscombe para mostrar a importância dos gráficos estatísticos.

Os dados de Anscombe contêm quatro conjuntos de dados, cada um com duas variáveis contínuas. Todos os conjuntos têm a mesma média, variância, correlação e linha de regressão. No entanto, somente quando os dados são visualizados é que se torna óbvia que os conjuntos não seguem o mesmo padrão. Isso serve para mostrar as vantagens das visualizações e as armadilhas de se observarem somente estatísticas resumidas.

```
# O conjunto de dados Anscombe pode ser encontrado na biblioteca seaborn
import seaborn as sns

anscombe = sns.load_dataset("anscombe")
print(anscombe)
```

| | dataset | x | y |
|----|---------|------|-------|
| 0 | I | 10.0 | 8.04 |
| 1 | I | 8.0 | 6.95 |
| 2 | I | 13.0 | 7.58 |
| 3 | I | 9.0 | 8.81 |
| 4 | I | 11.0 | 8.33 |
| 5 | I | 14.0 | 9.96 |
| 6 | I | 6.0 | 7.24 |
| 7 | I | 4.0 | 4.26 |
| 8 | I | 12.0 | 10.84 |
| 9 | I | 7.0 | 4.82 |
| 10 | I | 5.0 | 5.68 |
| 11 | II | 10.0 | 9.14 |
| 12 | II | 8.0 | 8.14 |
| 13 | II | 13.0 | 8.74 |
| 14 | II | 9.0 | 8.77 |
| 15 | II | 11.0 | 9.26 |
| 16 | II | 14.0 | 8.10 |
| 17 | II | 6.0 | 6.13 |
| 18 | II | 4.0 | 3.10 |
| 19 | II | 12.0 | 9.13 |
| 20 | II | 7.0 | 7.26 |
| 21 | II | 5.0 | 4.74 |
| 22 | III | 10.0 | 7.46 |
| 23 | III | 8.0 | 6.77 |
| 24 | III | 13.0 | 12.74 |
| 25 | III | 9.0 | 7.11 |
| 26 | III | 11.0 | 7.81 |
| 27 | III | 14.0 | 8.84 |
| 28 | III | 6.0 | 6.08 |
| 29 | III | 4.0 | 5.39 |
| 30 | III | 12.0 | 8.15 |
| 31 | III | 7.0 | 6.42 |
| 32 | III | 5.0 | 5.73 |
| 33 | IV | 8.0 | 6.58 |
| 34 | IV | 8.0 | 5.76 |

```

35      IV 8.0 7.71
36      IV 8.0 8.84
37      IV 8.0 8.47
38      IV 8.0 7.04
39      IV 8.0 5.25
40      IV 19.0 12.50
41      IV 8.0 5.56
42      IV 8.0 7.91
43      IV 8.0 6.89

```

```

# Para ver o famoso efeito Anscombe:
print("--- Estatísticas Descritivas Agrupadas por Dataset (Revelação do Anscombe) ---")
print(anscombe.groupby('dataset').describe())

```

--- Estatísticas Descritivas Agrupadas por Dataset (Revelação do Anscombe) ---

| dataset | x | | | | | | y | | | \ |
|---------|-------|------|----------|-----|-----|-----|------|------|-------|----------|
| | count | mean | std | min | 25% | 50% | 75% | max | count | |
| I | 11.0 | 9.0 | 3.316625 | 4.0 | 6.5 | 9.0 | 11.5 | 14.0 | 11.0 | 7.500909 |
| II | 11.0 | 9.0 | 3.316625 | 4.0 | 6.5 | 9.0 | 11.5 | 14.0 | 11.0 | 7.500909 |
| III | 11.0 | 9.0 | 3.316625 | 4.0 | 6.5 | 9.0 | 11.5 | 14.0 | 11.0 | 7.500000 |
| IV | 11.0 | 9.0 | 3.316625 | 8.0 | 8.0 | 8.0 | 8.0 | 19.0 | 11.0 | 7.500909 |

| dataset | | std | min | 25% | | | 50% | | 75% | | max |
|---------|----------|------|-------|------|------|-------|-----|-----|-----|--|-----|
| | | | | 50% | 75% | max | 50% | 75% | | | |
| I | 2.031568 | 4.26 | 6.315 | 7.58 | 8.57 | 10.84 | | | | | |
| II | 2.031657 | 3.10 | 6.695 | 8.14 | 8.95 | 9.26 | | | | | |
| III | 2.030424 | 5.39 | 6.250 | 7.11 | 7.98 | 12.74 | | | | | |
| IV | 2.030579 | 5.25 | 6.170 | 7.04 | 8.19 | 12.50 | | | | | |

4.2 matplotlib

matplotlib.pyplot

A `matplotlib` é a biblioteca fundamental de plotagem de Python. É extremamente flexível e dá ao usuário um controle total sobre os elementos da plotagem.

Importar os recursos de plotagem da `matplotlib` é um pouco diferente de nossas importações de pacote anteriores. Podemos pensar nesse operação como importar o pacote `matplotlib`, com todos os utilitários de plotagem encontrados em uma subpasta (ou subpacote) chamado `pyplot`. Assim como importamos um pacote e lhe demos um nome abreviado, podemos fazer o mesmo com `matplotlib.pyplot`.

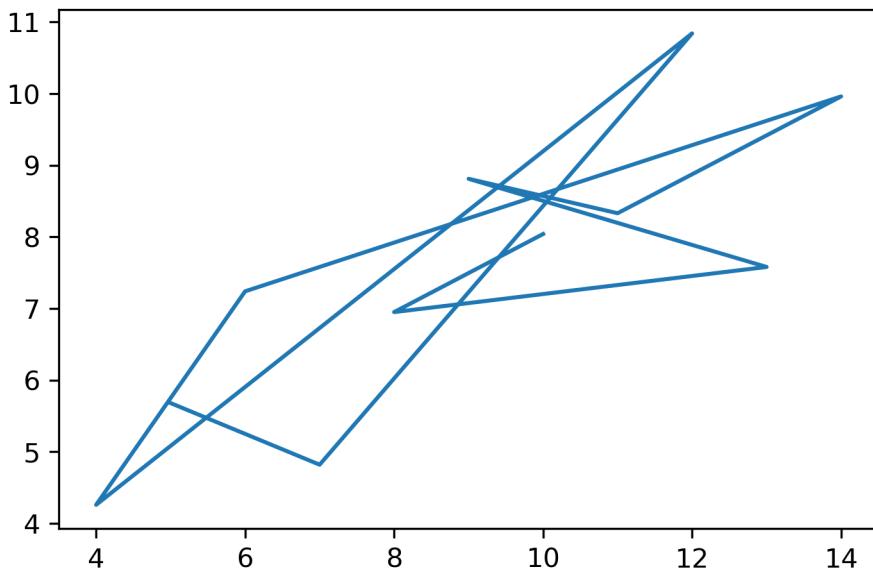
```
import matplotlib.pyplot as plt
```

Os nomes da maioria das plotagens básicas começará com `plt.plot`.

Em nosso exemplo, o recurso de plotagem recebe um vetor para os valores de `x` e um vetor correspondente para os valores de `y`.

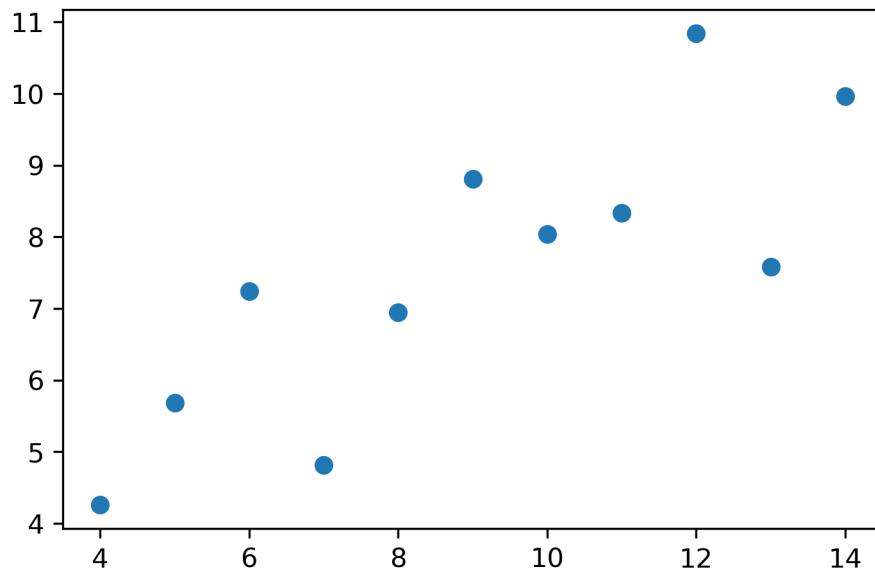
```
# Cria um subconjunto dos dados
# contém somente o primeiro conjunto de dados de Anscombe
dataset_1 = anscombe[anscombe['dataset'] == 'I']

plt.plot(dataset_1['x'], dataset_1['y'])
```



Por padrão, `plt.plot` desenhará linhas. Se quiser que ele desenhe círculos (pontos), podemos passar um parâmetro ‘o’ para dizer ao `plt.plot` que use pontos.

```
plt.plot(dataset_1['x'],dataset_1['y'],'o')
```



Subplots

Podemos repetir esse processo para o restante dos `datasets` em nossos dados de Anscombe.

```
dataset_2 = anscombe[anscombe['dataset'] == 'II']
dataset_3 = anscombe[anscombe['dataset'] == 'III']
dataset_4 = anscombe[anscombe['dataset'] == 'IV']
```

Nesse ponto, poderíamos fazer essas plotagens individualmente, uma de cada vez, mas o `matplotlib` oferece uma forma muito mais conveniente de criar subplotagens. Você pode especificar as dimensões de sua figura final e inserir plotagens menores que se enquadrem nas dimensões especificadas. Desse modo, é possível apresentar os resultados em uma única figura, em vez de exibi-los em figuras totalmente separadas.

A sintaxe de `subplot` aceita três parâmetros:

1. Número de linhas na figura para as subplotagens.
2. Número de colunas na figura para as subplotagens.
3. Local da subplotagem.

O local da subplotagem é numerado sequencialmente, e as plotagens são colocadas inicialmente da esquerda para a direita, e então de cima para baixo (linha, coluna). Se tentarmos plotar isso agora (apenas executando o código a seguir), teremos uma figura vazia.

Tudo que fizemos até agora foi criar uma figura e separá-la em uma grade de 2×2 na qual as plotagens podem ser colocadas. Como nenhuma plotagem foi criada nem inserida, nada é apresentado.

```

# Cria toda a figura na qual nossas subplotagens serão inseridas

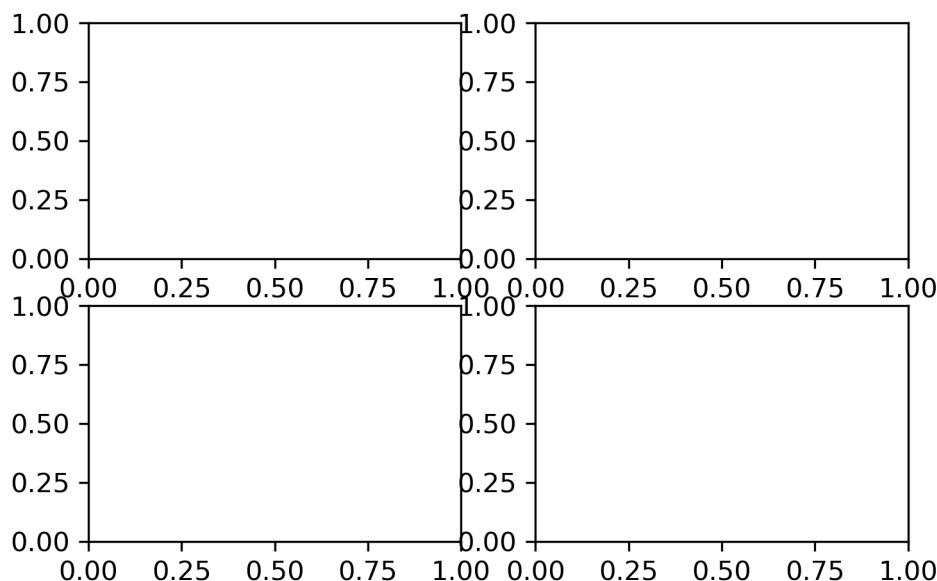
fig = plt.figure()
# diz a figura como as subplotagens deverão ser dispostas no exemplo,
# teremos 2 linhas de plotagens, e cada linha terá 2 plotagens
# a subplotagem tem 2 linhas e 2 colunas, local 1 de plotagem
axes1 = fig.add_subplot(2,2,1)

# a subplotagem tem 2 linhas e 2 colunas, local 2 de plotagem
axes2 = fig.add_subplot(2,2,2)

# a subplotagem tem 2 linhas e 2 colunas, local 3 de plotagem
axes3 = fig.add_subplot(2,2,3)

# a subplotagem tem 2 linhas e 2 colunas, local 4 de plotagem
axes4 = fig.add_subplot(2,2,4)

```



Podemos usar o método `plot` em cada eixo para criar nossa plotagem.

```

# adicionar ma plotagem em cada um dos eixos criados anteriormente
axes1.plot(dataset_1['x'], dataset_1['y'], "o")
axes2.plot(dataset_2['x'], dataset_2['y'], "o")
axes3.plot(dataset_3['x'], dataset_3['y'], "o")
axes4.plot(dataset_4['x'], dataset_4['y'], "o")

```

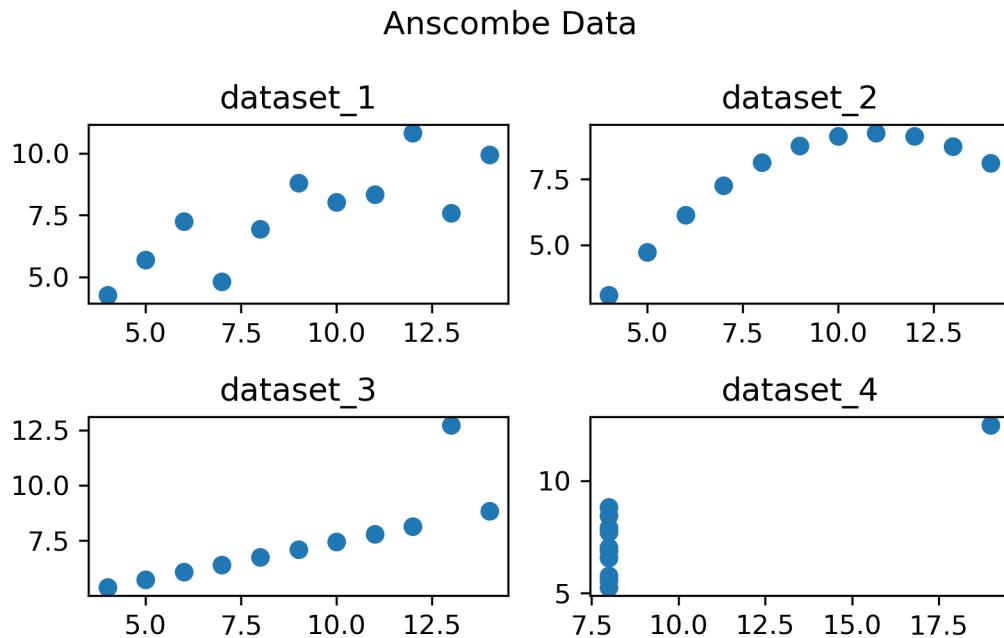
Por fim, podemos acrescentar um rótulo às nossas subplotagens e usar `tight_layout` para garantir que os eixos estejam separados uns dos outros.

```
# acrescenta um pequeno título para cada subplotagem
axes1.set_title("dataset_1")
axes2.set_title("dataset_2")
axes3.set_title("dataset_3")
axes4.set_title("dataset_4")

# adiciona um título para toda figura
fig.suptitle("Anscombe Data")

# usa um layout organizado
fig.tight_layout()

fig
```



As visualizações dos dados de Anscombe mostram por que simplesmente olhar valores estatísticos resumidos pode levar a enganos. No momento em que os pontos são visualizados, torna-se claro que, apesar de cada conjunto de dados ter os mesmos valores estatísticos resumidos, os relacionamentos entre os pontos variam bastante entre os conjuntos de dados.

Para concluir o exemplo com os dados de Anscombe, podemos adicionar `set_xlabel()` e `set_ylabel()` em cada uma das subplotagens a fim de acrescentar rótulos aos eixos x e y,

assim como adicionamos um título à figura.

Partes de uma figura

Antes de prosseguir e aprender a criar mais plotagens estatísticas, você deverá ter familiaridade com a documentação da `matplotlib` sobre “*Parts of a Figure*” (Partes de uma figura). Reproduzi a figura mais antiga na Figura 2, e a nova na Figura 3.

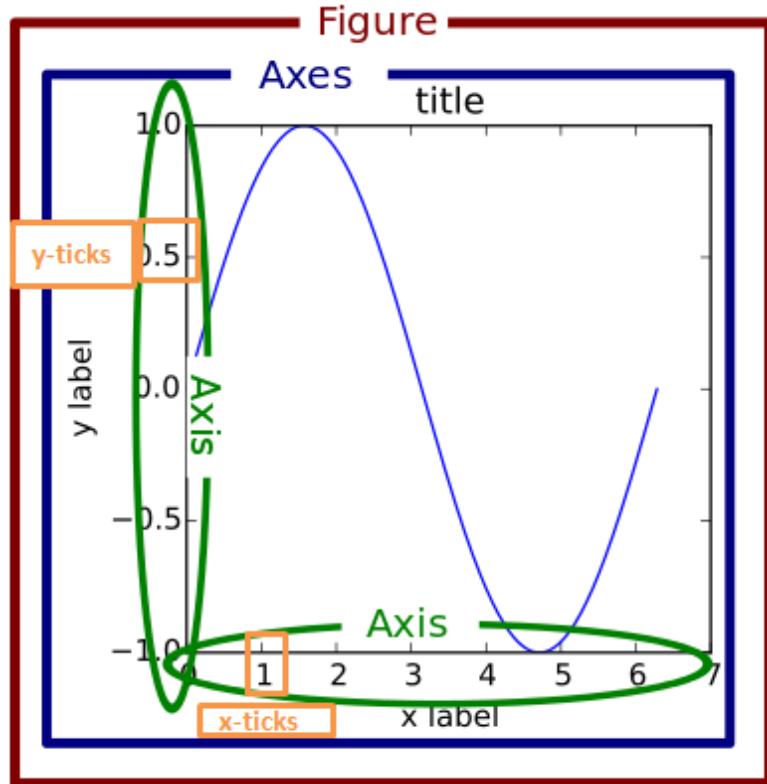


Figura 2: Uma das partes mais confusas da plotagem em Python está no uso dos termos “eixo” e “eixos”, pois referem-se a diferentes partes de uma figura. Essa era a versão antiga da figura “*Parts of a Figure*” da documentação da `matplotlib`.

Uma das partes mais confusas da plotagem em Python é o uso do termo “eixo” e “eixos”, especialmente quando tentamos descrever as diferentes partes. No exemplo com os dados Anscombe, cada subplotagem individual tem eixos. Os eixos contêm tanto um eixo x quanto um eixo y. Todas as quatro subplotagens juntas formam a figura.

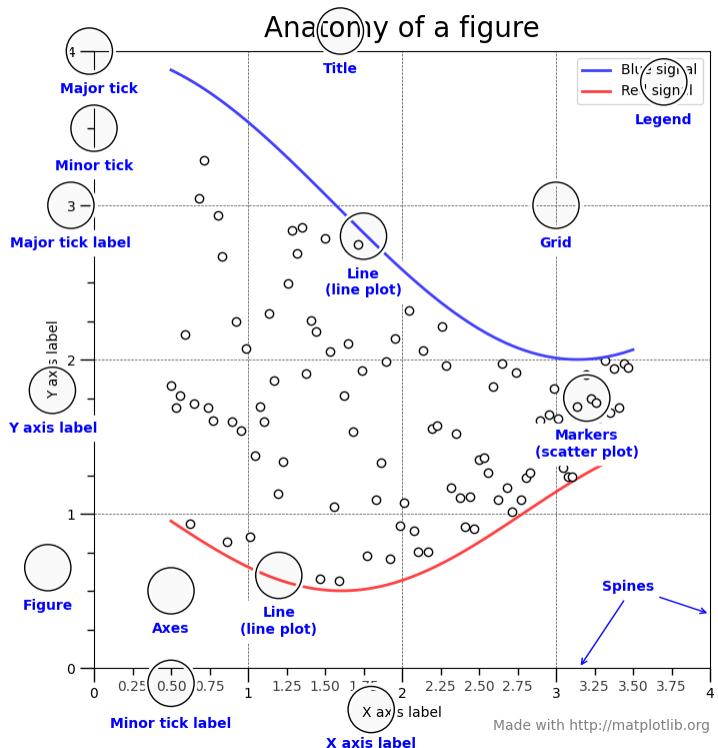


Figura 3: Uma versão mais nova da representação de “Parts of a Figure”, com mais detalhes sobre outros aspectos de uma figura. De modo diferente da figura antiga, a figura mais recente foi totalmente criada com a `matplotlib`.

4.3 Gráficos estatísticos usando a `matplotlib`

Os dados de gorjeta que usaremos na próxima série de visualizações são provenientes da biblioteca `seaborn`. Esse conjunto de dados contém a quantidade de gorjetas que as pessoas deixam para diversas variáveis. Por exemplo, o custo total da conta, o tamanho do grupo, o dia da semana e o horário.

Podemos carregar esse conjunto de dados como fizemos com o conjunto de dados de Anscombe.

```
tips = sns.load_dataset("tips")
print(tips.head())
```

| | total_bill | tip | sex | smoker | day | time | size |
|---|------------|------|--------|--------|-----|--------|------|
| 0 | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| 1 | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| 2 | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 |
| 3 | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| 4 | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |

4.3.1 Univariado

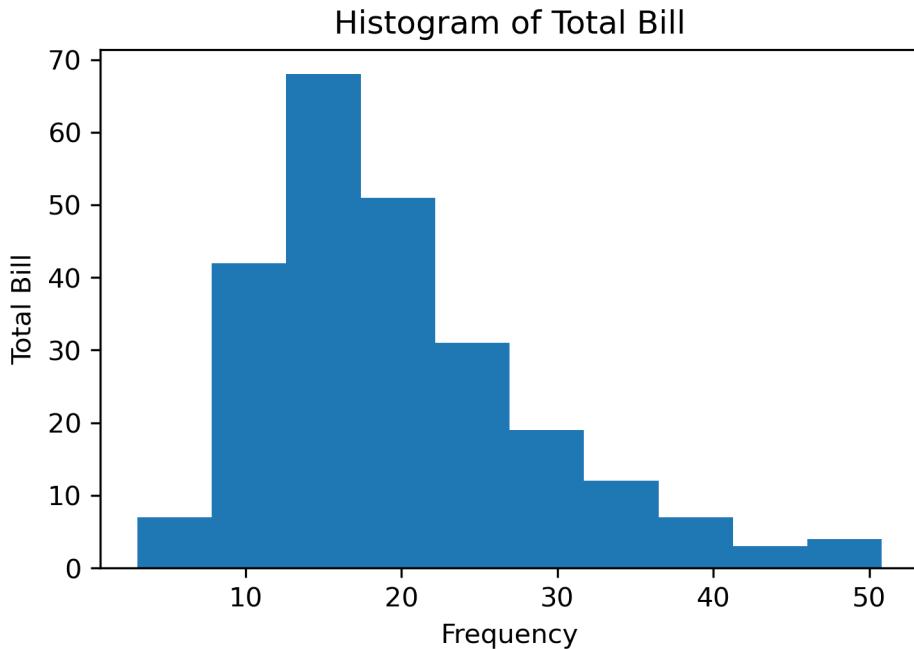
No jargão de estatística, o termo “univariado” refere-se a uma única variável.

4.3.1.1 Histogramas

Os histogramas são o meio mais comum de observar uma única variável. Os valores são colocados em “recipientes”, isto é, são agrupados e plotados para mostrar a distribuição da variável.

```
fig = plt.figure()
axes1 = fig.add_subplot(1,1,1)
axes1.hist(tips['total_bill'],bins=10)
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
axes1.set_ylabel('Total Bill')
```

```
Text(0, 0.5, 'Total Bill')
```



`bins=10`: Define que o intervalo de dados ('total_bill') deve ser dividido em 10 “caixas” ou barras (bins) de igual largura para o histograma.

4.3.2 Bivariado

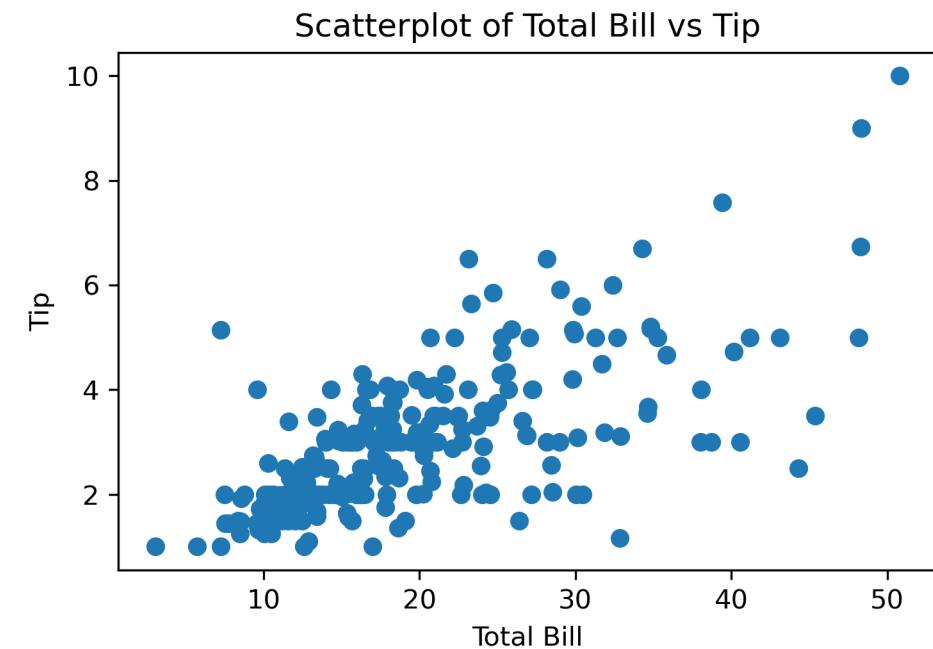
No jargão de estatística, o termo “bivariado” refere-se a duas variáveis.

4.3.2.1 Gráfico de dispersão

O gráfico de dispersão (scatterplots) são usados quando uma variável contínua é plotada em relação a outra variável contínua.

```
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1,1,1)
axes1.scatter(tips['total_bill'],tips['tip'])
axes1.set_title("Scatterplot of Total Bill vs Tip")
axes1.set_xlabel("Total Bill")
axes1.set_ylabel("Tip")
```

Text(0, 0.5, 'Tip')



Sobre o código:

- `plt.figure()`:

Cria uma Figura (Figure). Pense na Figura como a tela ou o pôster onde você vai desenhar. É o contêiner de alto nível que pode conter um ou mais gráficos (eixos).

- `scatter_plot.add_subplot()`:

Adiciona um Eixo (Axes) à sua Figura. O Eixo é o gráfico real onde os dados serão plotados.

(1, 1, 1): Esses três números definem a grade da Figura e onde seu gráfico será posicionado:

- O primeiro 1: A figura terá 1 linha de gráficos.
- O segundo 1: A figura terá 1 coluna de gráficos.
- O terceiro 1: O gráfico será posicionado na primeira posição (linha 1, coluna 1).

Neste caso, você está criando um único gráfico grande na Figura.

O objeto Eixo é armazenado na variável `axes1`.

- `axes1.scatter(...)`:

Este é o método que desenha o gráfico. Você está chamando o método scatter no seu objeto Eixo (`axes1`).

`tips['total_bill']`: Define os valores que serão colocados no eixo X.

`tips['tip']`: Define os valores que serão colocados no eixo Y.

Ele desenha um gráfico de dispersão (`scatterplot`) mostrando a relação entre a conta total (`total_bill`) e a gorjeta (`tip`) do seu DataFrame `tips`.

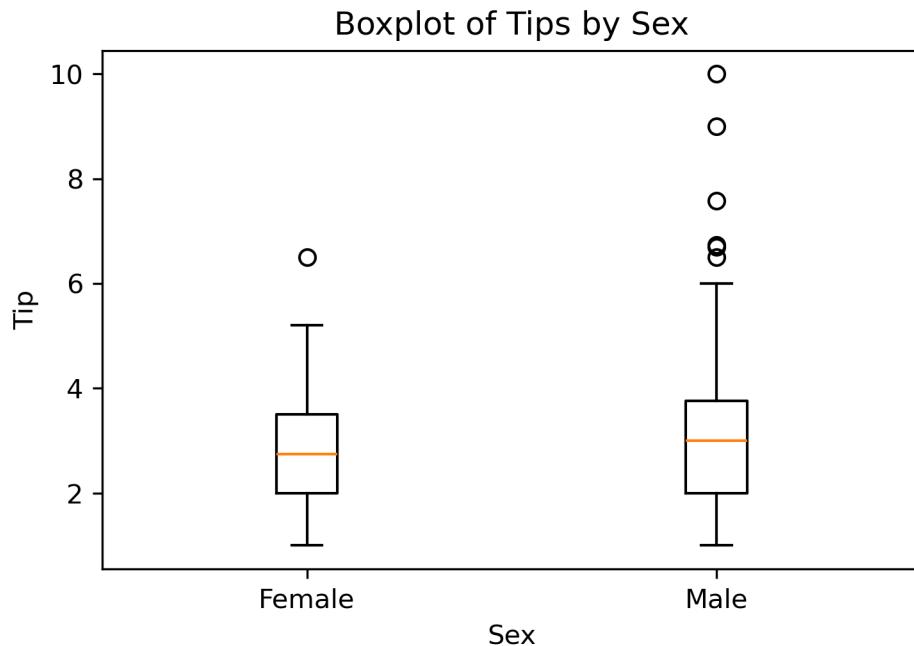
- `set_title()`: Define o título principal do gráfico.
- `set_xlabel()`: Define o rótulo do eixo X.
- `set_ylabel()`: Define o rótulo do eixo Y.

4.3.2.2 Gráfico de caixa

Os gráficos ou diagramas de caixa (boxplots) são usados quando uma variável discreta é plotada em relação a outra variável contínua.

```
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1,1,1)
axes1.boxplot(
    # O primeiro argumento de um gráfico de caixa é o dado,
    # pois estamos plotando várias porções de dados;
    # temos que colocar cada porção de dados em uma lista
    [tips[tips['sex'] == 'Female']['tip'],
     tips[tips['sex'] == 'Male']['tip']],
    # Podemos então passar um parâmetro labels opcional
    # para especificar um rótulo aos dados que passamos
    labels = ['Female','Male']
)
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Sex')
```

Text(0.5, 1.0, 'Boxplot of Tips by Sex')



Sobre o código:

- `boxplot = plt.figure():`

Cria a Figura (a “tela” ou contêiner).

- `axes1 = boxplot.add_subplot(1,1,1):`

Adiciona o Eixo (o gráfico real) que ocupará toda a Figura.

- `axes1.boxplot(...):`

Chama o método para desenhar o gráfico de caixa no seu Eixo.

O Boxplot no Matplotlib espera uma lista de arrays/Series, onde cada item da lista representa uma caixa separada no gráfico.

Você está usando o Pandas para filtrar os dados:

- `tips[tips['sex'] == 'Female']['tip']:`

Seleciona todas as gorjetas (tip) onde o sexo (sex) é ‘Female’.

- `tips[tips['sex'] == 'Male']['tip']:`

Seleciona todas as gorjetas (tip) onde o sexo (sex) é ‘Male’.

Ao passar essas duas Séries em uma lista, o Matplotlib cria dois boxplots lado a lado.

`labels = ['Female', 'Male']:` Atribui os rótulos correspondentes a cada boxplot na ordem em que os dados foram passados, garantindo que você saiba qual caixa representa qual grupo.

4.3.3 Dados multivariados

Plotar dados multivariados é complicado porque não há uma panaceia nem um template que possam ser usados em todos os casos.

Para mostrar o processo de plotar dados multivariados, vamos nos basear no nosso gráfico de dispersão anterior. Se quiséssemos adicionar outra variável, por exemplo, `sex`, uma opção seria colorir os pontos com base no valor da terceira variável.

Se quiséssemos adicionar uma quarta variável, poderíamos acrescentar tamanho aos pontos. A única ressalva em usar o tamanho como uma variável é que os seres humanos não são muito bons para diferenciar áreas. É claro que, se houver um ponto enorme ao lado de um ponto minúsculo, sua informação será evidente, mas diferenças menores são difíceis de distinguir e poderão deixar a sua visualização confusa. Uma maneira de reduzir a confusão é adicionar certa dose de transparência aos pontos individuais, de modo que muitos pontos sobrepostos serão exibidos como uma região mais escura de uma plotagem, em comparação com áreas menos cheias.

Ou seja, tamanho e transparência são recursos interessantes, porém cor é mais recomendados para destacar as variáveis. A cor é o recurso mais direto para adicionar uma variável extra (terceira) em um gráfico de dispersão, enquanto o tamanho, embora útil para uma quarta variável, deve ser usado com cautela devido à dificuldade humana em diferenciar áreas. A transparência serve como um ajuste para aumentar a clareza, revelando regiões de maior concentração de pontos sobrepostos.

Como regra geral, cores diferentes são muito mais fáceis de distinguir do que variações de tamanho. Se você tiver de usar áreas para representar diferenças entre valores, certifique-se de que estará realmente plotando áreas relativas. Uma armadilha comum é mapear um valor para o raio de um círculo nas plotagens, mas, como a fórmula de um círculo é πr^2 , suas áreas na verdade estarão baseadas em uma escala quadrática. Isso não só é enganoso como também é incorreto.

Ao utilizar o tamanho (área) dos pontos para representar valores de dados, uma ‘armadilha’ comum e incorreta é mapear o valor dos dados (V) diretamente para o raio (r) de um círculo. Como a área do círculo é determinada pela fórmula $A = \pi r^2$, se $r = V$, a área plotada crescerá em escala quadrática ($A = \pi V^2$) em relação ao valor original, e não linearmente. Isso é enganoso. A solução, de forma análoga, é mapear o valor (V) para a área desejada ($A_{desejada} = V$) e, então, calcular o raio necessário ($r = \sqrt{\frac{V}{\pi}}$).

Cores também são difíceis de escolher. Os seres humanos não percebem tons em uma escala linear, portanto será necessário pensar cuidadosamente ao selecionar as paletas de cores. Felizmente o `matplotlib` e o `seaborn` têm o próprio conjunto de paletas de cores, e ferramentas como o `colorbrewer` podem ajudar você a escolher boas paletas.

A figura x usa cores para adicionar uma terceira variável, `sex`, em nosso gráfico de dispersão.

```
# Cria uma variável de cor baseada em sexo
def recode_sex(sex):
    if sex == 'Female':
        return 0
    else:
        return 1

tips['sex_color'] = tips['sex'].apply(recode_sex)

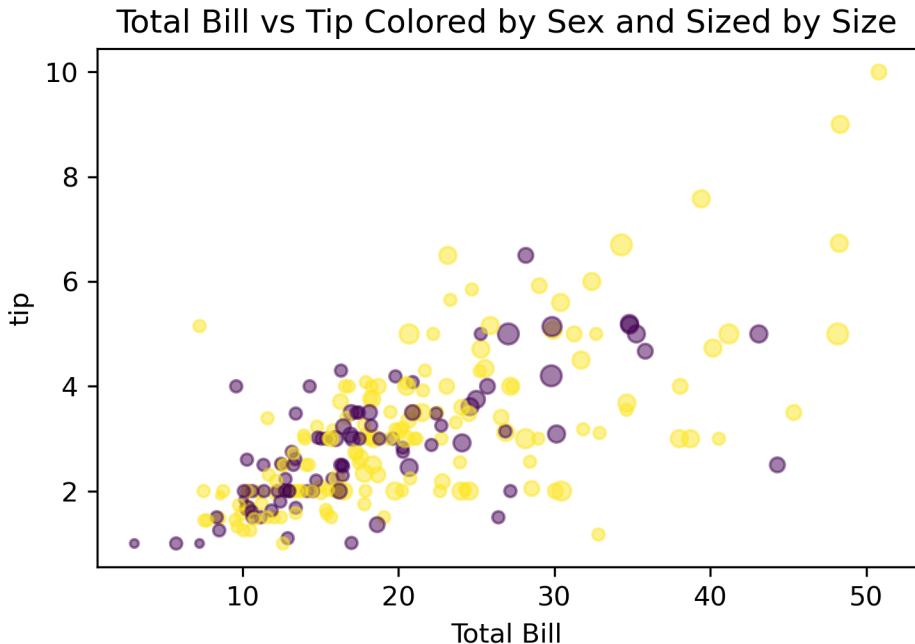
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1,1,1)
axes1.scatter(
    x=tips['total_bill'],
    y=tips['tip'],

    # Define o tamanho dos pontos com base no tamanho dos grupos;
    # Multiplicamos os valores por 10 para deixar os pontos maiores
    # e enfatizar as diferenças.
    s=tips['size']*10,

    # Define a cor para o sexo.
    c=tips['sex_color'],

    # Define o valor de alpha para que os pontos sejam mais transparentes.
    # Isso ajuda no caso de pontos que se sobreponham.
    alpha=0.5
)
axes1.set_title('Total Bill vs Tip Colored by Sex and Sized by Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('tip')

Text(0, 0.5, 'tip')
```



Sobre o programa:

- `recode_sex(sex):`

Esta é uma função simples que converte os valores categóricos da coluna 'sex' ('Female' e 'Male') em valores numéricos.

- Se o sexo for 'Female', retorna 0.
- Se for 'Male' (ou qualquer outro valor), retorna 1.

- `tips['sex_color'] = tips['sex'].apply(recode_sex):`

- Cria uma nova coluna no DataFrame tips chamada `sex_color`.
- Aplica a função `recode_sex` a cada valor da coluna original `tips['sex']`.
- Propósito: O `Matplotlib` (e muitas bibliotecas de plotagem) geralmente exige valores numéricos para mapear cores em um gráfico de dispersão. Esta coluna (`sex_color`) será usada para definir a cor dos pontos.

- `s = tips['size']*10:`

Define o tamanho dos pontos. O valor de `tips['size']` (tamanho do grupo/mesa) é multiplicado por 10 para torná-los mais visíveis e enfatizar as diferenças. `size` (Tamanho do Grupo).

- `c = tips['sex_color']:`

Define a cor dos pontos, usando os valores numéricos 0 (Female) e 1 (Male). `sex` (Sexo).

- `alpha` (Constante 0.5):

Define a transparência dos pontos (0.0 é totalmente transparente, 1.0 é opaco). Isso é muito útil para visualizações onde pontos se sobrepõem (*overplotting*). (Transparência).

4.4 seaborn

Podemos pensar a biblioteca `matplotlib` como a principal ferramenta básica de plotagem em Python. O `seaborn` é baseada na `matplotlib` e oferece uma interface de nível mais alto para **gráficos estatísticos**. Ele provê uma interface para criar visualizações mais bonitas e complexas com menos linhas de código.

A biblioteca `seaborn` está altamente integrada ao `Pandas` e ao restante da pilha PyData (`numpy`, `scipy`, `statsmodels`), fazendo com que as visualizações de qualquer parte do processo de análise de dados seja muito simples.

Como o `seaborn` foi desenvolvido com base no `matplotlib`, o usuário continua tendo a capacidade de fazer ajustes finos nas visualizações.

4.4.1 Univariado

Assim como fizemos com os exemplos da `matplotlib`, criaremos uma série de plotagens univariadas.

4.4.1.1 Histogramas

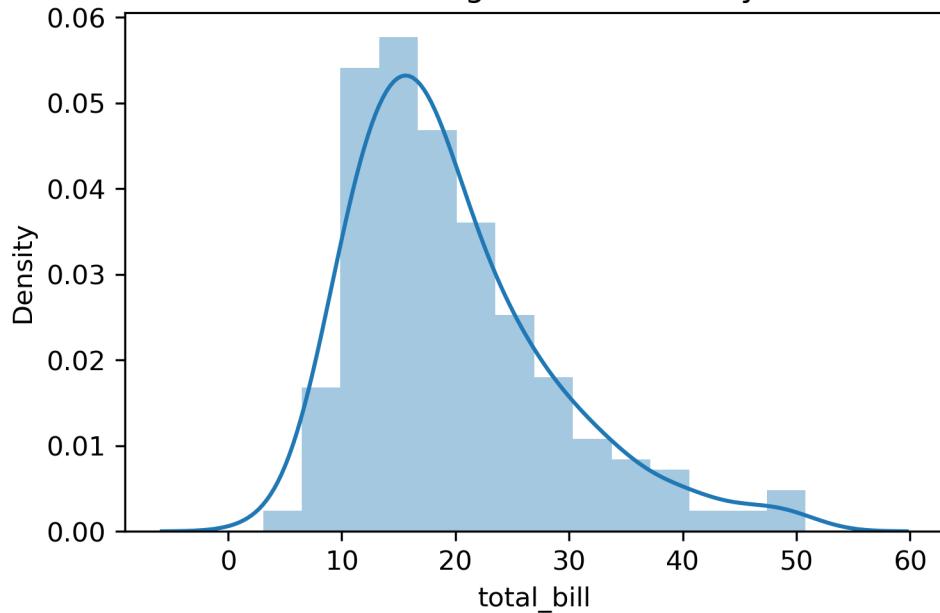
Os histogramas são criados usando `sns.distplot`.

```
# Essa função subplots é um atalho para criar objetos
# separados de figuras e adicionar subplotagens
# individuais (eixos) a figura
hist, ax = plt.subplots()

# Usa a função distplot do seaborn para criar nossa plotagem
ax = sns.distplot(tips['total_bill'])
ax.set_title("Total Bill Histogram with Density Plot")
```

Text(0.5, 1.0, 'Total Bill Histogram with Density Plot')

Total Bill Histogram with Density Plot



O `distplot` default plotará tanto um histograma quanto uma plotagem de densidade (usando uma estimativa de densidade por kernel).

o **KDE** (*Kernel Density Estimate*), que é a linha de densidade suave que aparece sobre o histograma por padrão.

`hist, ax = plt.subplots()` Cria simultaneamente dois objetos principais para a plotagem: a Figure (hist) e os Axes (ax).

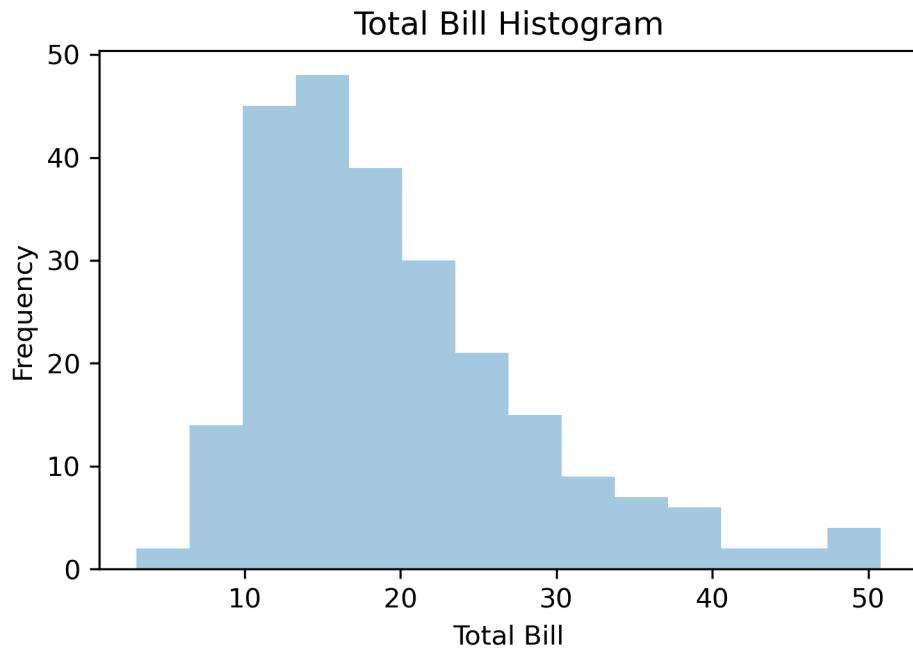
- **hist** (Figure): É o contêiner geral de todos os elementos. Pense na tela em branco ou na janela onde o gráfico será desenhado.
- **ax** (Axes): É o sistema de coordenadas real, ou seja, a área de plotagem com eixos X e Y. É nele que você chama as funções de plotagem (`sns.distplot`, `ax.set_title`, etc.).
- É possível reescrever de uma forma menos comum, mas com o mesmo resultado para um único gráfico:

```
figura = plt.figure() # Cria apenas o contêiner (Figure)
eixos = figura.add_subplot(1, 1, 1) # Adiciona a área de plotagem (Axes)
# Em seguida, você usaria 'eixos' no lugar de 'ax'
```

Se quisermos somente o histograma, podemos definir o parâmetro `kde` com `False`.

```
hist, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], kde=False)
ax.set_title('Total Bill Histogram')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Frequency')
```

```
Text(0, 0.5, 'Frequency')
```



`distplot` caiu em desuso, as alternativas recomendadas são:

- `seaborn.histplot()`:

É a função principal e de “baixo nível” (axes-level) para plotar apenas o histograma (barras) de uma única distribuição, ou um histograma com a curva de Densidade de Kernel (KDE) sobreposta, se você usar o argumento `kde=True`.

- `seaborn.displot()`:

É a função de “alto nível” (figure-level) que substitui o `distplot`. Por padrão, ela plota um histograma usando o `histplot()`, mas tem a capacidade adicional de:

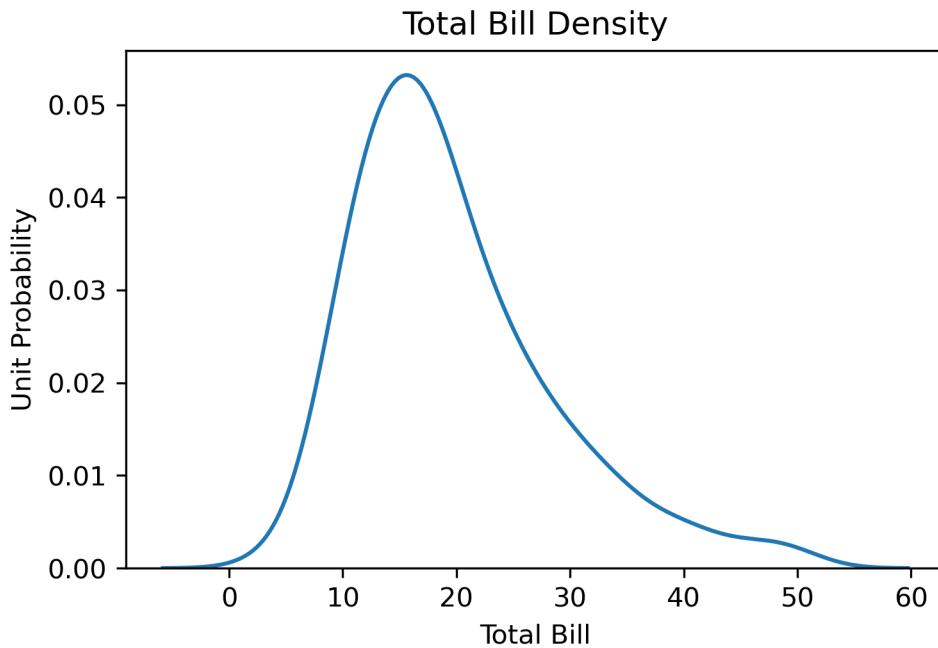
- Criar múltiplos gráficos usando `col` ou `row` (FacetGrid).
- Mudar o tipo de gráfico (por exemplo, `kind='kde'` para um gráfico de densidade puro).

4.4.1.2 Plotagem de densidade (estimativa de densidade por Kernel)

As plotagens de densidade (*density plots*) são outra maneira de visualizar uma distribuição univariada.

Essencialmente, são criadas usando uma distribuição normal centralizada em cada ponto de dado, e então suavizado as plotagens que se sobreponem de modo que a área sob a curva seja 1.

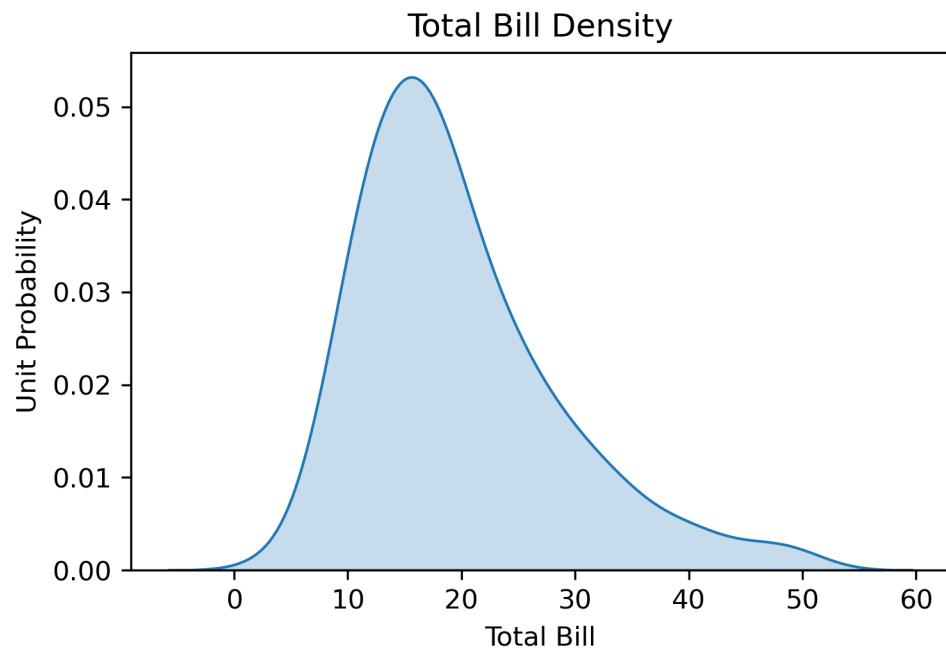
```
den, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], hist=False)
ax.set_title('Total Bill Density')
ax.set_xlabel("Total Bill")
ax.set_ylabel("Unit Probability")  
  
Text(0, 0.5, 'Unit Probability')
```



Você também pode executar `sns.kdeplot` se quiser somente uma plotagem de densidade.

```
den, ax = plt.subplots()
ax = sns.kdeplot(tips['total_bill'], fill=True)
ax.set_title('Total Bill Density')
ax.set_xlabel("Total Bill")
ax.set_ylabel("Unit Probability")
```

```
Text(0, 0.5, 'Unit Probability')
```



`fill=True`: Preenche a área sob a curva de densidade para facilitar a visualização.

4.4.1.3 Rug plot

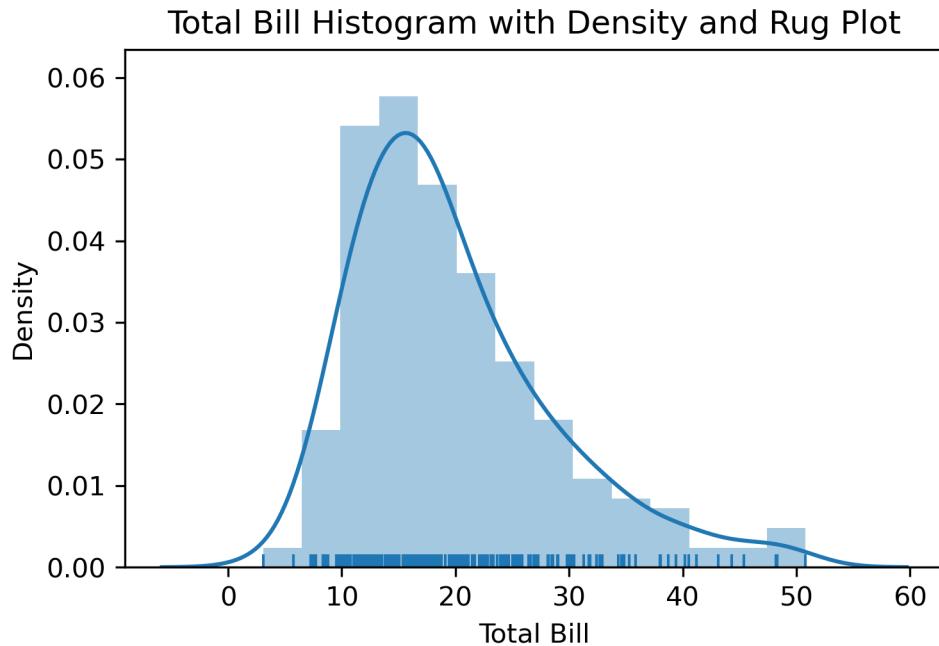
As **rug plots** (literalmente, plotagem de tapete) são uma representação unidimensional da distribuição de uma variável.

Em geral, são usadas com outras plotagens para melhorar uma visualização.

É um histograma sobreposto a uma plotagem de densidade, com um **rug plot** na parte inferior.

```
hist_den_rug, ax = plt.subplots()
ax = sns.distplot(tips["total_bill"], rug=True)
ax.set_title("Total Bill Histogram with Density and Rug Plot")
ax.set_xlabel("Total Bill")
```

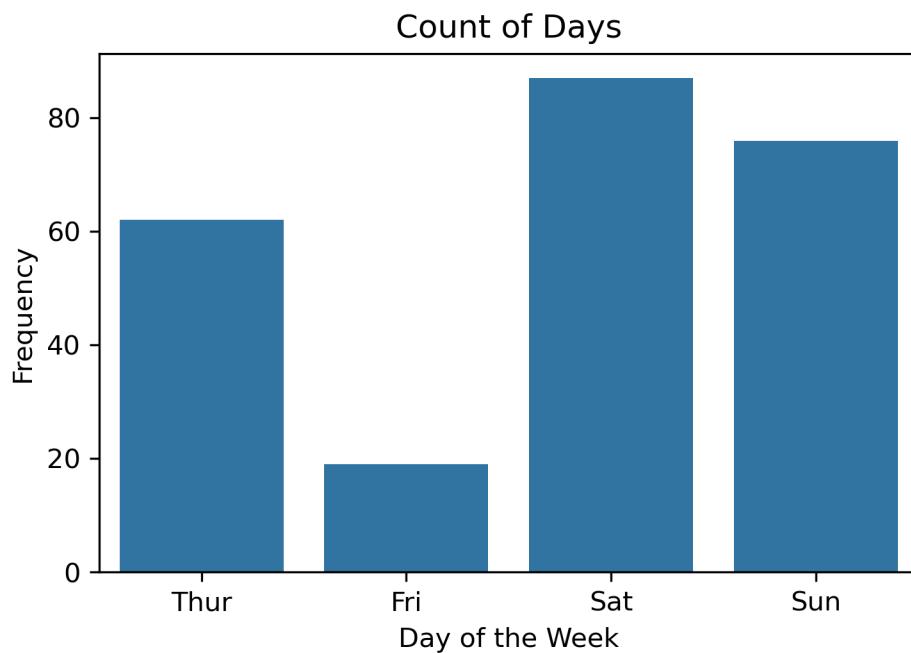
```
Text(0.5, 0, 'Total Bill')
```



4.4.1.4 Plotagem de contadores (plotagem de barras)

As plotagens de barra (bar plots) são muito parecidas com os histogramas, mas, em vez de colocar os valores em “recipientes” para gerar uma distribuição, as plotagens de barras podem ser usadas para contar variáveis discretas. Uma plotagem de contadores (`count plot`) é usada com essa finalidade.

```
count, ax = plt.subplots()  
ax = sns.countplot(x='day', data=tips)  
ax.set_title("Count of Days")  
ax.set_xlabel("Day of the Week")  
ax.set_ylabel("Frequency")  
  
Text(0, 0.5, 'Frequency')
```



4.4.2 Dados bivariados

Usaremos agora a biblioteca `seaborn` para plotar varias variáveis.

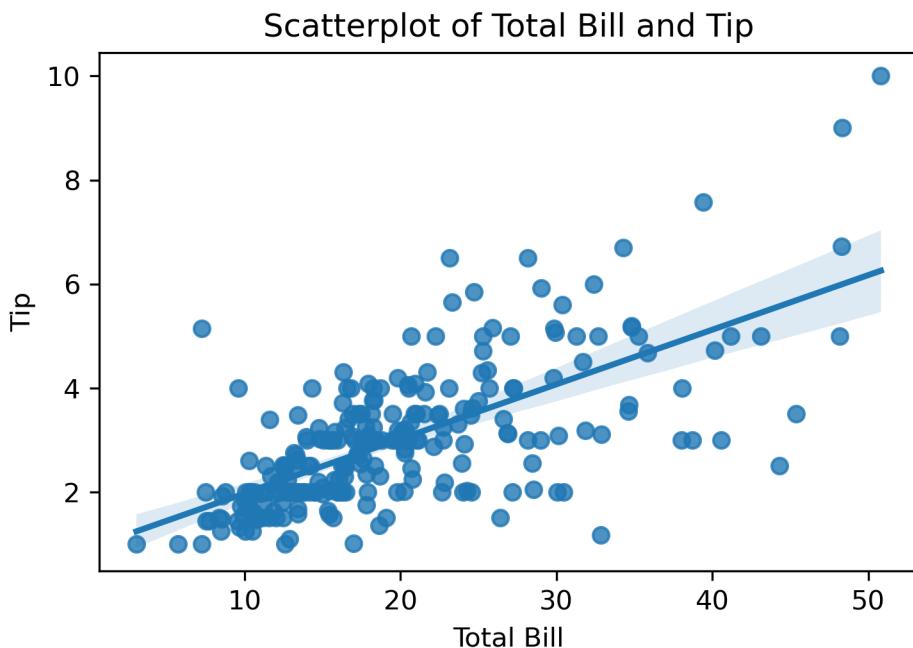
4.4.2.1 Gráficos de dispersão

4.4.2.1.1 `regplot`

Há algumas maneiras de criar um gráfico de dispersão no `seaborn`. Não há nenhuma função explícita chamada `scatter`. Em vez disso, usamos `regplot`. Ela plotará um gráfico de dispersão, além de criar uma **linha de regressão**.

Se definirmos `fit_reg=False`, a visualização mostrará somente o gráfico de dispersão.

```
scatter, ax = plt.subplots()
ax = sns.regplot(
    x='total_bill',
    y='tip',
    data=tips)
ax.set_title("Scatterplot of Total Bill and Tip")
ax.set_xlabel("Total Bill")
ax.set_ylabel("Tip")  
  
Text(0, 0.5, 'Tip')
```



Sobre o programa:

- `sns.regplot()`:

É uma das opções de gráfico de dispersão para `seaborn`, com **linha de regressão** inclusa (para desativar a linha de regressão `fit_reg=False`).

- Argumentos do `regplot` (`x`=, `y`= e `data`=):

A função `regplot`, precisa que sejam explicitadas quais as colunas dos valores de `x` e `y`, e o dataframe da qual esta pegando os valores.

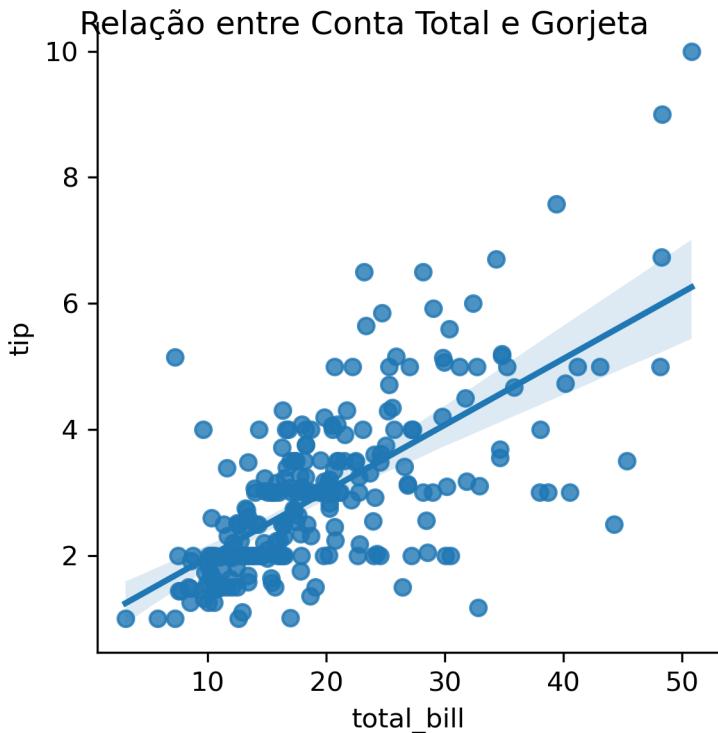
4.4.2.1.2 lmplot

Uma função semelhante, `lmplot`, também é capaz de criar um gráfico de dispersão. Internamente, `lmplot` chama `regplot`, portanto esse é uma função de plotagem mais genérica.

A principal diferença é que `regplot` cria eixos, enquanto `lmplot` cria uma figura.

```
g = sns.lmplot(  
    x='total_bill',  
    y='tip',  
    data=tips,  
    height=4, # Controla a altura de cada painel (em polegadas)  
    aspect=1 # Controla a proporção LARGURA/ALTURA.  
        # Um valor maior que 1 aumenta a largura em relação à altura.  
)  
  
g.fig.suptitle("Relação entre Conta Total e Gorjeta")
```

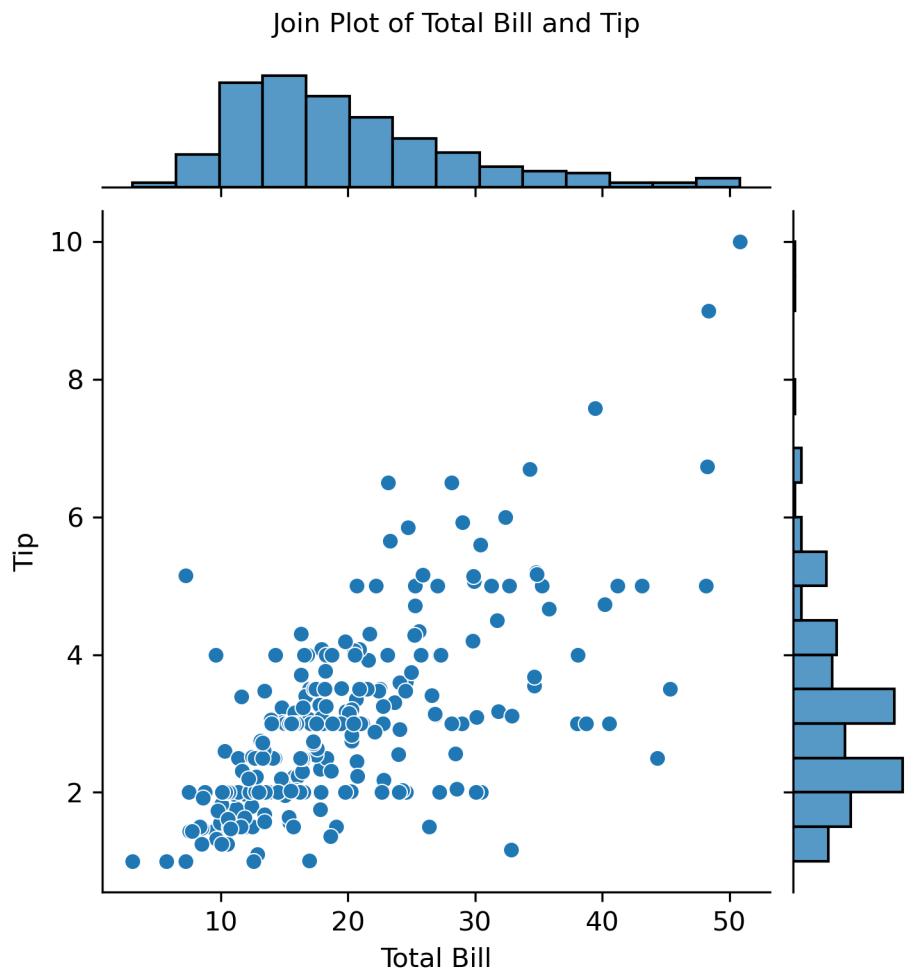
```
Text(0.5, 0.98, 'Relação entre Conta Total e Gorjeta')
```



4.4.2.1.3 jointplot

Também podemos criar um gráfico de dispersão que inclua uma plotagem univariada em cada eixo usando `jointplot`. Uma diferença importante é que `jointplot` não devolve eixos, portanto não precisamos criar uma figura com eixos na qual colocaremos nossa plotagem. Em vez disso, essa função cria um objeto `JointGrid`.

```
joint = sns.jointplot(  
    x='total_bill',  
    y='tip',  
    data=tips,  
    height=5 # O valor padrão é 6. Reduzir para 5, 4 ou 3 diminui o gráfico.  
)  
  
joint.set_axis_labels(  
    xlabel='Total Bill',  
    ylabel='Tip')  
  
#Adiciona um título, define o tamanho da fonte e  
# move o texto acima dos eixos de total das contas  
joint.fig.suptitle('Join Plot of Total Bill and Tip',  
    fontsize=10,  
    y=1.03)  
  
Text(0.5, 1.03, 'Join Plot of Total Bill and Tip')
```



4.4.2.2 Plotagem hexbin

Os gráficos de dispersão são ótimos para comparar duas variáveis. As vezes, porém, há pontos demais para que um gráfico de dispersão seja significativo. Uma forma de contornar esse problema é reunir pontos no gráfico. Assim como os histogramas podem reunir dados de uma variável para criar uma barra, o `hexbin` pode fazer o mesmo com duas variáveis. Um hexágono é usado com essa finalidade, pois é o formato mais eficiente para cobrir uma superfície 2D arbitrária.

Esse é um exemplo do `seaborn` funcionando com base na `matplotlib`, pois `hexbin` é uma função da `matplotlib`.

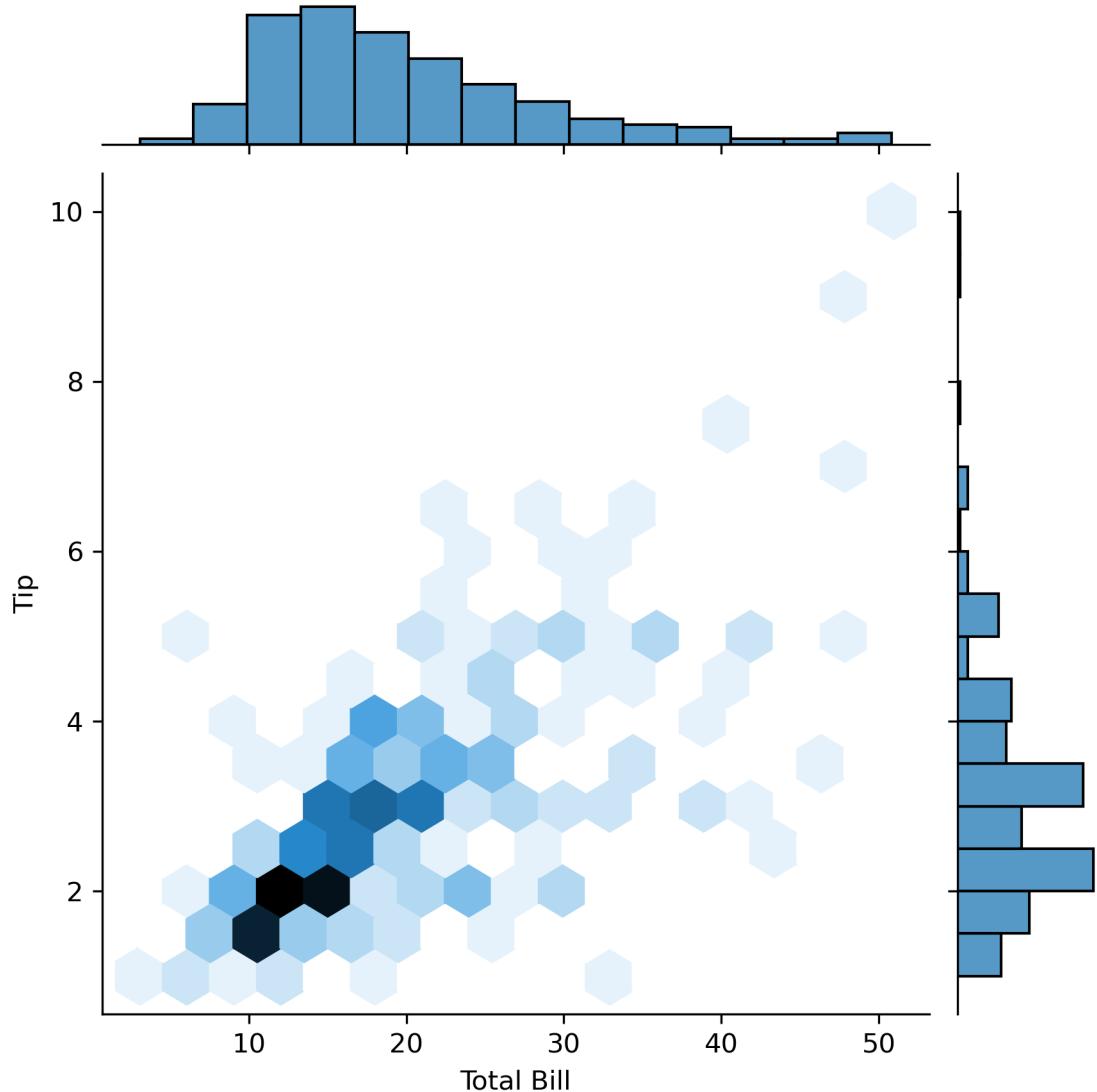
```
hexbin = sns.jointplot(          # Cria o Gráfico de Junção (Joint Plot)
    x="total_bill",             # Define a variável para o Eixo X
    y="tip",                   # Define a variável para o Eixo Y
    data=tips,                 # Especifica o DataFrame a ser usado
    kind="hex"                  # Define o tipo de gráfico como Hexbin
)

hexbin.set_axis_labels(          # Define os Rótulos dos Eixos
    xlabel='Total Bill',       # Rótulo para o Eixo X
    ylabel='Tip'                # Rótulo para o Eixo Y
)

hexbin.fig.suptitle(            # Adiciona um Título Geral à Figura
    'Hexbin Joint Plot of Total Bill and Tip', # Texto do título
    fontsize=10,               # Define o tamanho da fonte
    y=1.03                     # Ajusta a posição vertical do título
)
```

Text(0.5, 1.03, 'Hexbin Joint Plot of Total Bill and Tip')

Hexbin Joint Plot of Total Bill and Tip



Sobre o código:

- O método `sns.jointplot()` cria sua própria figura e seus próprios eixos (`subplots`) internamente. Por essa razão, você não precisa usar o padrão `fig, ax = plt.subplots()` para criar um `jointplot`.
- `hexbin = sns.jointplot(...)`

Esta linha é a principal, responsável por criar e desenhar toda a figura. O resultado é armazenado na variável `hexbin`, que é um objeto `JointGrid` do Seaborn.

- `x="total_bill", y="tip", data=tips`

Definem as variáveis do DataFrame `tips` a serem plotadas: `total_bill` no eixo horizontal (X), `tip` no eixo vertical (Y) e o Dataframe como `tips`.

- `kind="hex"`

Especifica o tipo de plotagem para a área central (bivariada). O valor “hex” indica o uso do gráfico de densidade `Hexbin`, onde a cor representa a frequência de pontos.

- `hexbin.set_axis_labels(...)`

Método usado para definir os rótulos de forma clara para os eixos X e Y do gráfico principal, melhorando a interpretação.

- `hexbin.fig.suptitle(...)`

Adiciona um título grande (`suptitle`) à figura inteira (que contém todos os `subplots`), sendo essencial para contextualizar o gráfico.

4.4.2.3 Plotagem de densidade 2D

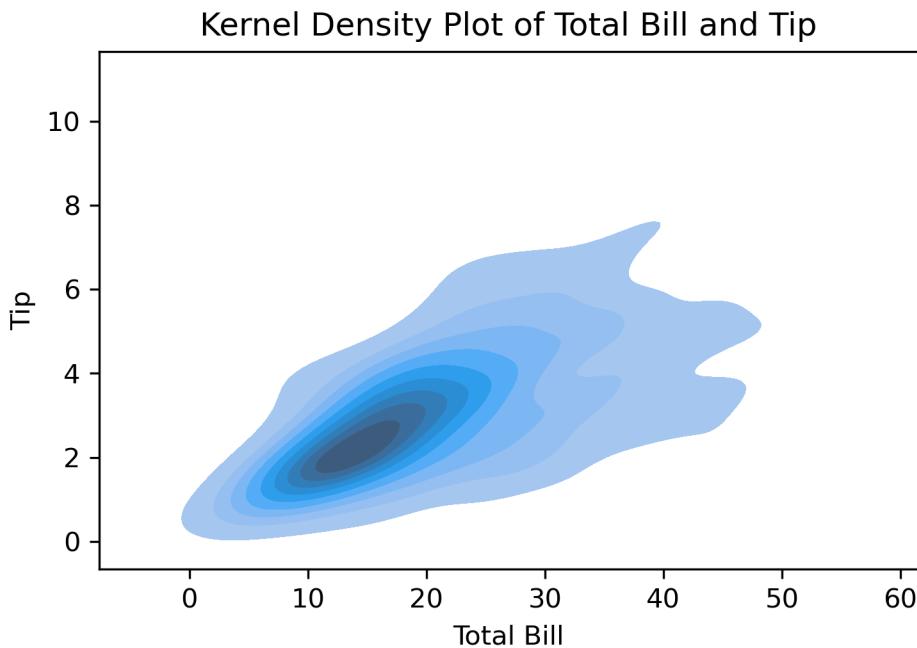
Também é possível criar uma plotagem de densidade por kernel 2D. Esse tipo de processo é semelhante ao modo como `sns.kdeplot` funciona, exceto porcriar uma plotagem de densidade entre duas variáveis.

A plotagem bivariada pode ser mostrada sozinha, ou você pode colocar as duas plotagens univariadas uma ao lado da outra usando `jointplot`.

```
kde, ax = plt.subplots() # Cria uma figura e um conjunto de eixos
ax = sns.kdeplot( # Gera um gráfico de estimativa de densidade kernel 2D
    x=tips['total_bill'], # Usa a coluna 'total_bill' para o eixo X
    y=tips['tip'], # Usa a coluna 'tip' para o eixo Y
    fill=True # Preenche as áreas sob as curvas de densidade
)

# Define o título do gráfico
ax.set_title('Kernel Density Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill') # Define o rótulo do eixo X
ax.set_ylabel('Tip') # Define o rótulo do eixo Y

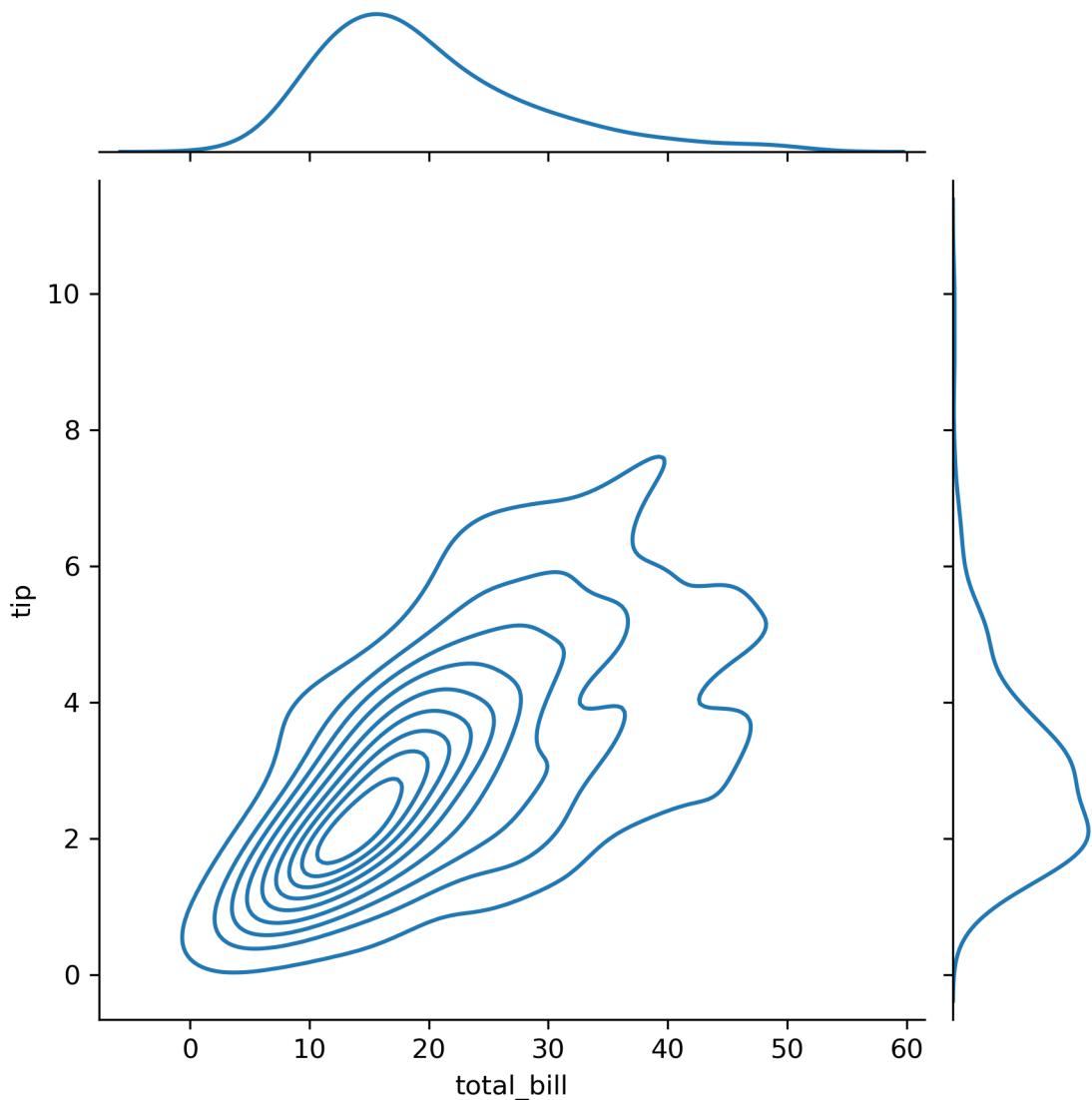
Text(0, 0.5, 'Tip')
```



Sobre o código:

- Na versão mais antiga do `kdeplot` (no livro) ao invés de usar `x`, `y` e `fill`, era respectivamente `data`, `data2` e `shade`, porém o significado é o mesmo.

```
kde_joint = sns.jointplot( # Cria um gráfico de distribuição conjunta
    x='total_bill',           # Define a coluna para o eixo X
    y='tip',                  # Define a coluna para o eixo Y
    data=tips,                # Especifica o DataFrame a ser usado
    kind='kde'                 # Usa a Estimativa de Densidade Kernel (KDE)
    # para o plot central e margens
)
```



4.4.2.4 Plotagem de barras

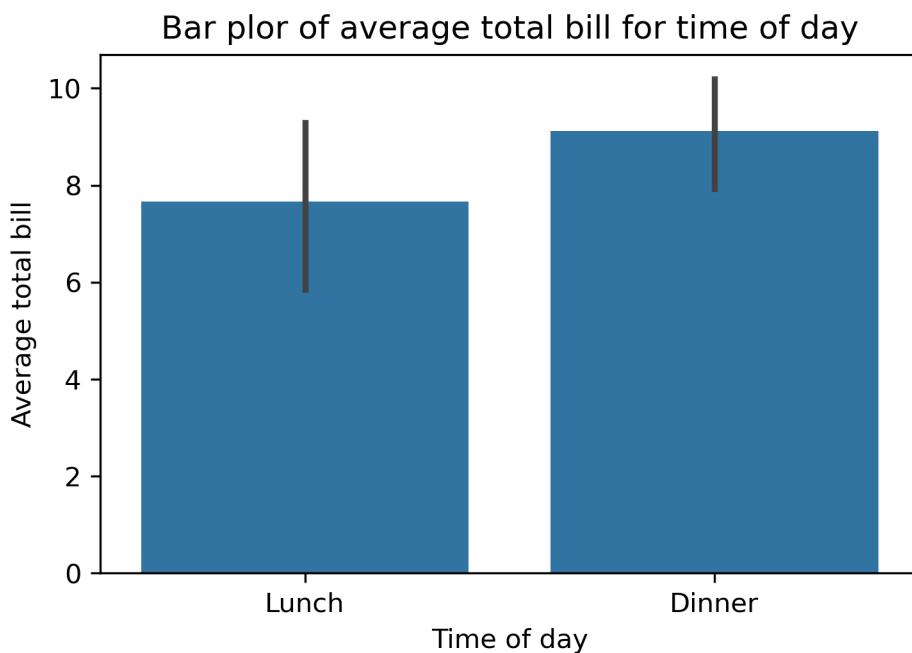
As plotagens de barras (**bar plots**) também podem ser usadas para mostrar diversas variáveis.

Por padrão, `barplot` calculará uma média, mas você pode passar qualquer função para o parâmetro `estimador`. Por exemplo, poderíamos passar a função `numpy.std` para calcular o desvio-padrão.

```
bar, ax = plt.subplots()
ax = sns.barplot(
    x='time',
    y='total_bill',
    data=tips,
    estimator=np.std
)

ax.set_title("Bar plor of average total bill for time of day")
ax.set_xlabel("Time of day")
ax.set_ylabel("Average total bill")

Text(0, 0.5, 'Average total bill')
```



4.4.2.5 Gráfico de caixa

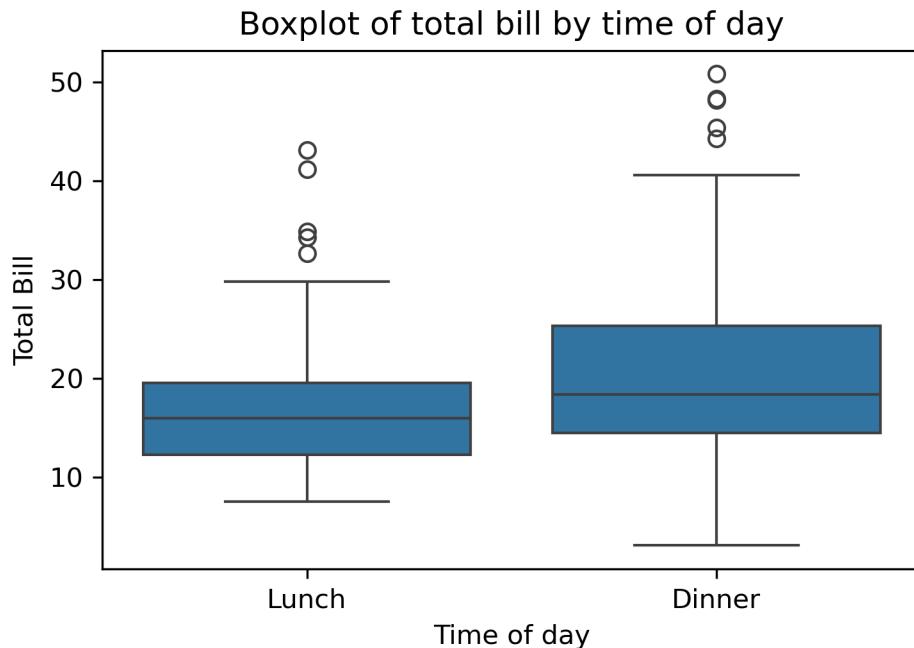
De modo diferente das plotagens mencionadas antes, uma plotagem de caixa (`boxplot`) mostra várias estatísticas: o mínimo, o primeiro quartil, a mediana, o terceiro quartil, o máximo e, se for aplicável, valores discrepantes baseados no intervalo interquartil.

O parâmetro `y` é opcional. Se for omitido, a função de plotagem criará uma única caixa no gráfico.

```
box, ax = plt.subplots()
ax = sns.boxplot(
    x='time',
    y='total_bill',
    data=tips
)

ax.set_title('Boxplot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')

Text(0, 0.5, 'Total Bill')
```



4.4.2.6 Plotagem de violino

As plotagens de caixa são uma visualização estatística clássica, mas elas podem obscurecer a distribuição subjacente dos dados.

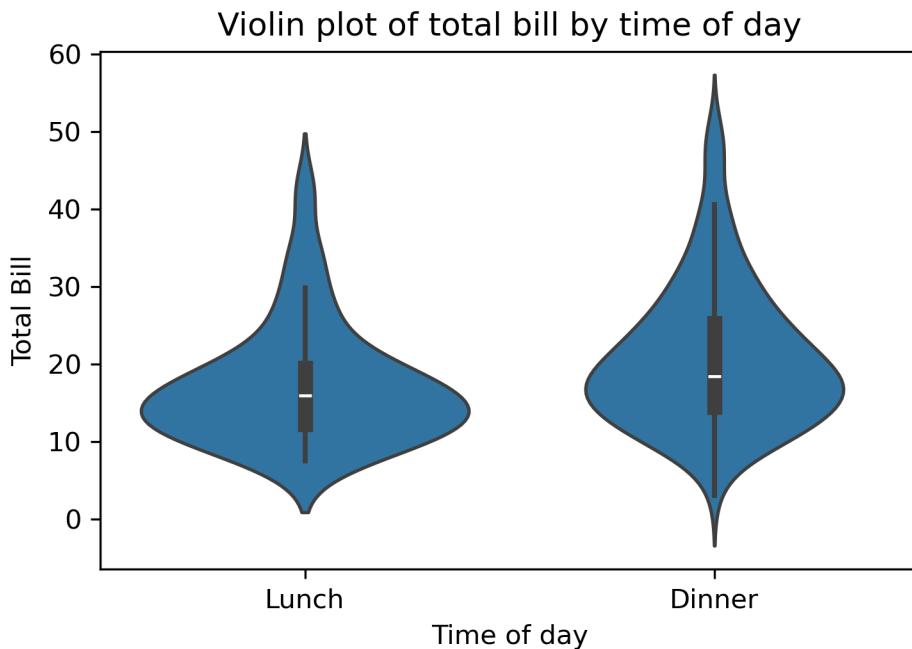
As plotagens de violino (*violin plots*) podem mostrar os mesmos valores que uma plotagem de caixa, porém plotam as “caixas” com uma estimativa de **densidade por kernel**.

Isso pode ajudar a preservar informações mais visuais sobre seus dados, pois é possível que plotar somente estatísticas resumidas leve a enganos, conforme vimos pelo quarteto de Anscombe.

```
violin, ax = plt.subplots()
ax = sns.violinplot(
    x='time',
    y='total_bill',
    data=tips
)

ax.set_title("Violin plot of total bill by time of day")
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')
```

Text(0, 0.5, 'Total Bill')

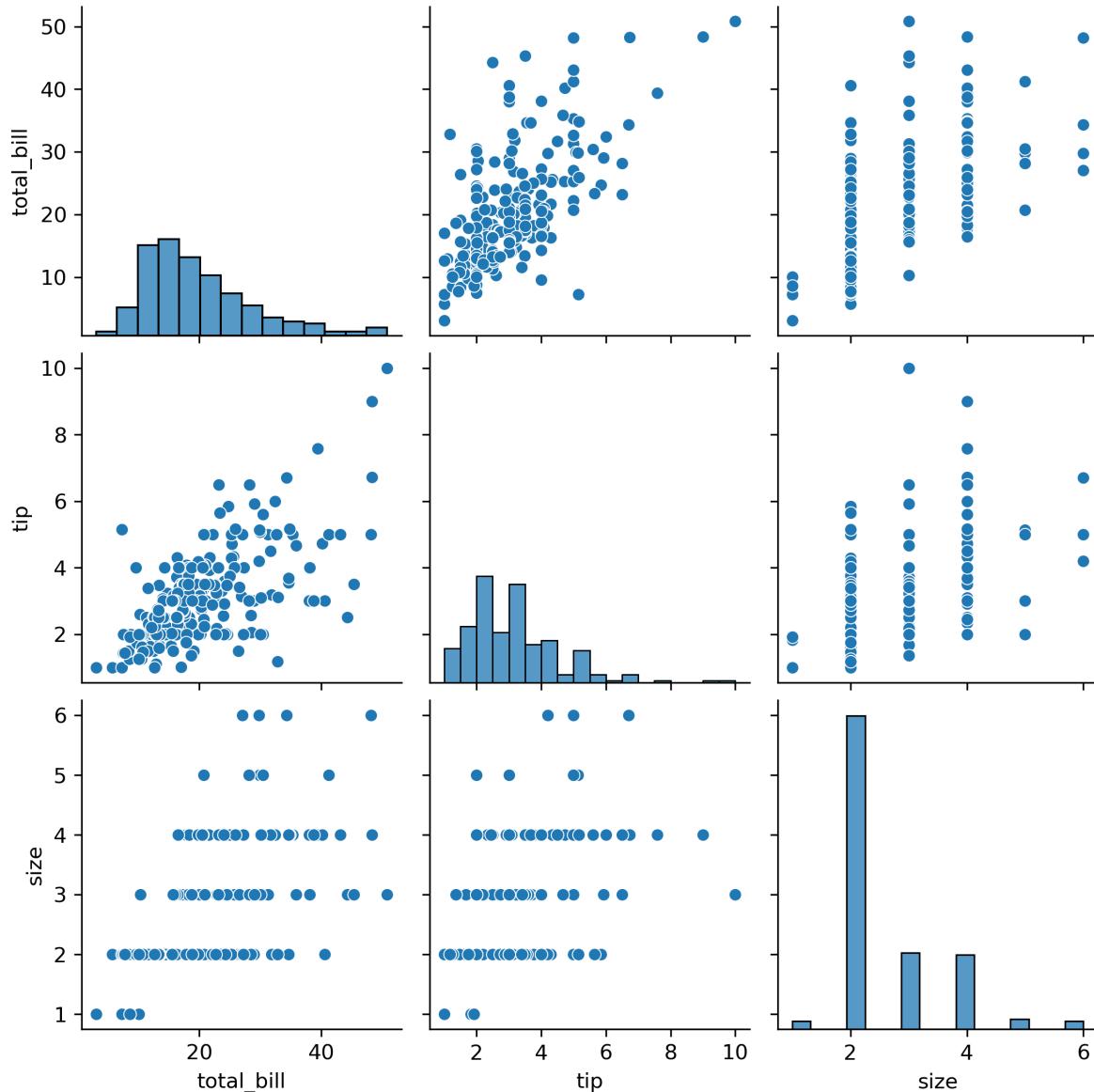


4.4.2.7 Relacionamentos aos pares

Quando temos dados numéricos em sua maioria, a visualização de todos os relacionamentos aos pares pode ser feita facilmente usando `pairplot`.

Essa função plotará um gráfico de dispersão entre cada par de variáveis, e um histograma para os dados univariados.

```
fig = sns.pairplot(tips)
```



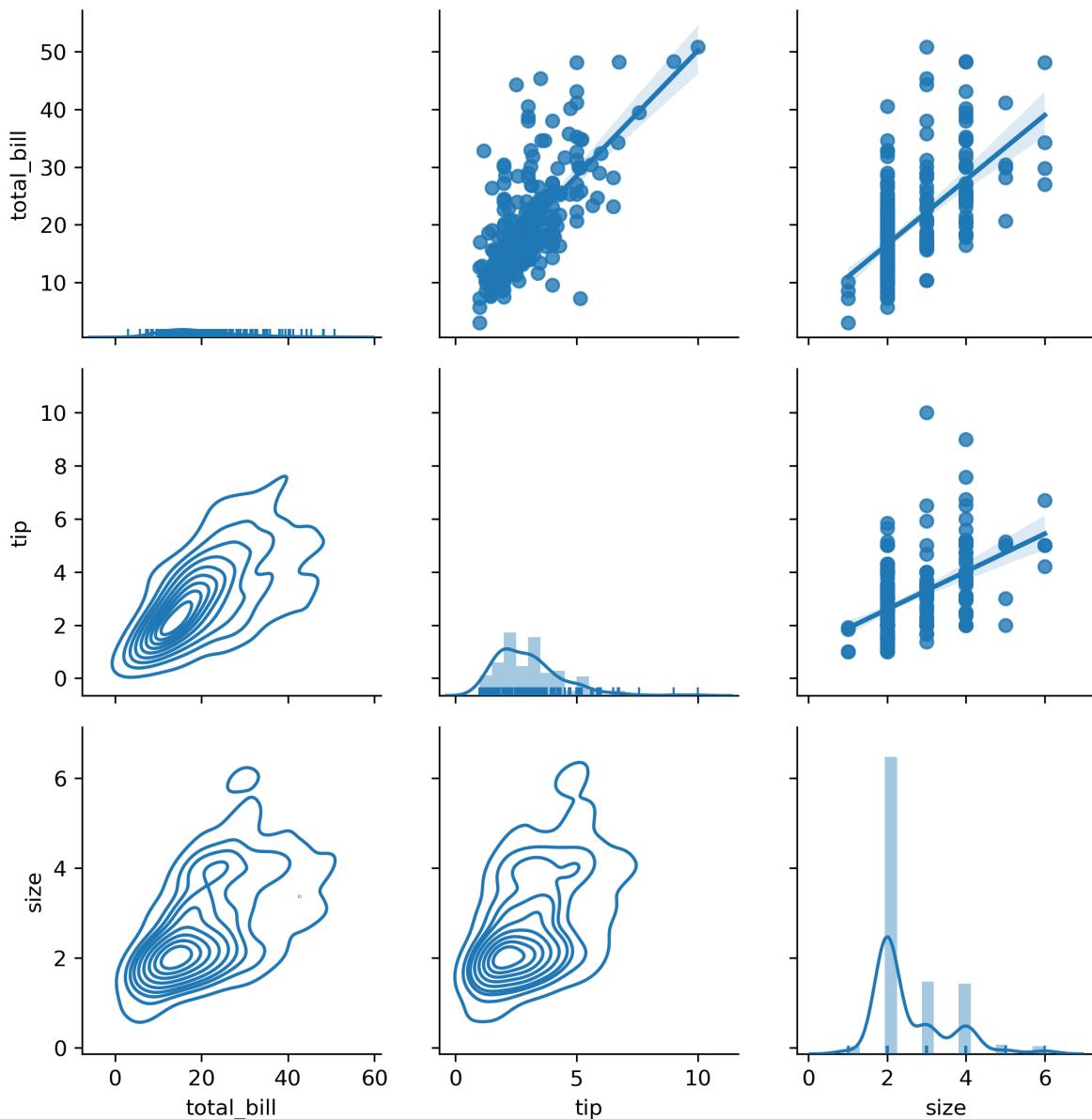
Sobre o código:

- Finalidade: Cria um `pair plot` (gráfico de pares), que é uma matriz de gráficos que visualiza as relações pareadas em um dataset Pandas.
- Visualização Principal: Mostra `scatter plots` (gráficos de dispersão) para as relações entre cada par de variáveis numéricas.
- Diagonal: Por padrão, exibe um histograma para a distribuição de cada variável única na diagonal da matriz.
- Retorno: A função retorna um objeto `PairGrid` (que foi atribuído à variável `fig`), permitindo ajustes posteriores no gráfico.

Uma desvantagem ao usar `pairplot` esta no fato de haver informações redundantes, isto é, a parte superior da visualização é igual a metade inferior.

Podemos usar `pairgrid` para atribuir manualmente as plotagens para a metade superior e a metade inferior.

```
pair_grid = sns.PairGrid(tips)
# podemos usar plt.scatter em vez de sns.regplot
pair_grid = pair_grid.map_upper(sns.regplot)
pair_grid = pair_grid.map_lower(sns.kdeplot)
pair_grid = pair_grid.map_diag(sns.distplot, rug=True)
```



Sobre o código:

- Controle Fino: O uso de `sns.PairGrid()` permite que o usuário especifique funções de plotagem diferentes para as áreas superior, inferior e diagonal da matriz.
- Inicialização: A primeira linha (`pair_grid = sns.PairGrid(tips)`) apenas inicializa a estrutura da matriz; ela não plota nada. O dataframe tips é carregado no objeto `PairGrid`.

- Diagonal (`map_diag`): Plota a distribuição de cada variável única (e.g., histograma com linha de densidade de kernel) usando `sns.distplot` na diagonal. O argumento `rug=True` adiciona `rug plots` (pequenas marcações) no eixo para indicar observações individuais.
- Triângulo Superior (`map_upper`): Plota gráficos de regressão (`regression plots`) usando `sns.regplot` nas células acima da diagonal, visualizando a relação e a linha de regressão linear entre os pares de variáveis.
- Triângulo Inferior (`map_lower`): Plota gráficos de densidade de kernel (Kernel Density Estimate - `KDE plots`) bivariados usando `sns.kdeplot` nas células abaixo da diagonal. Isso mostra a concentração conjunta dos dados entre os pares de variáveis.

4.4.3 Dados multivariados

Conforme mencionamos na seção anterior, não há nenhum template que seja um padrão de fato para plotagem de dados multivariados.

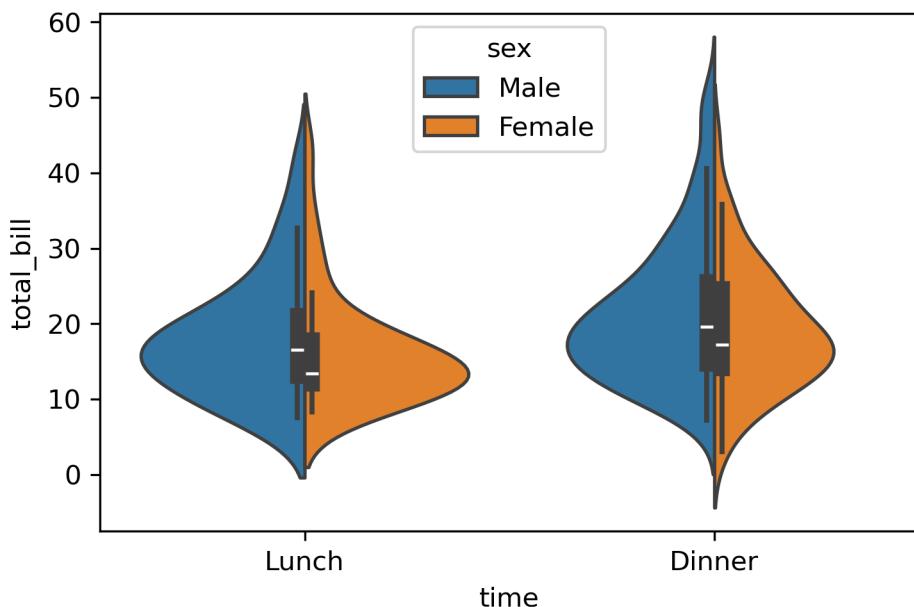
Possíveis maneiras de incluir mais informações são usar **cor**, **tamanho** e **formato** para distinguir os dados na plotagem.

4.4.3.1 Cores

Quando usamos `violinplot`, podemos passar o parâmetro `hue` para colorir a plotagem de acordo com `sex`. É possível reduzir as informações redundantes fazendo com que cada metade dos violinos represente um `sex` diferente.

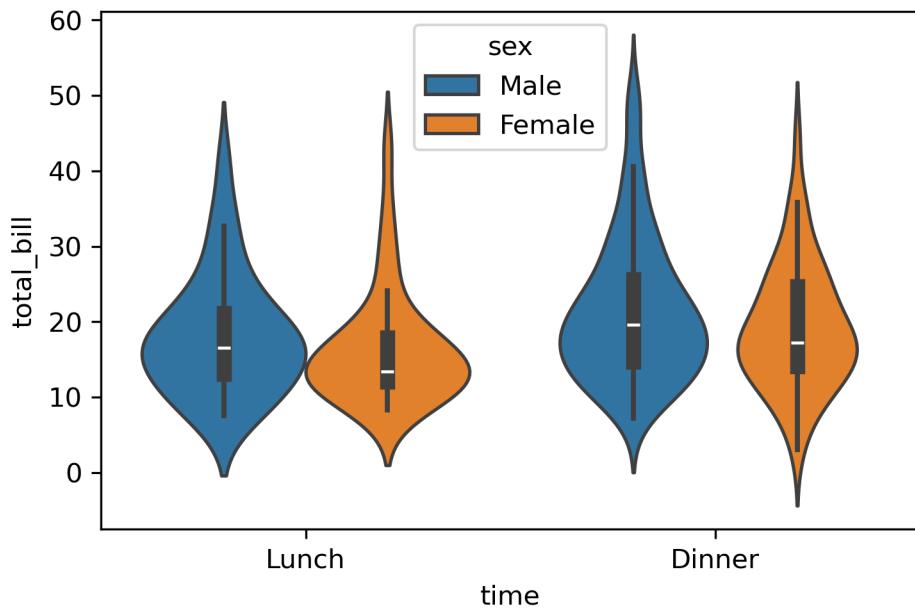
- Com o parâmetro `split`:

```
violin, ax = plt.subplots()
ax = sns.violinplot(
    x='time',
    y='total_bill',
    hue='sex',
    data=tips,
    split=True
)
```



- Sem o parâmetro `split`:

```
violin, ax = plt.subplots()
ax = sns.violinplot(
    x='time',
    y='total_bill',
    hue='sex',
    data=tips,
    split=False
)
```

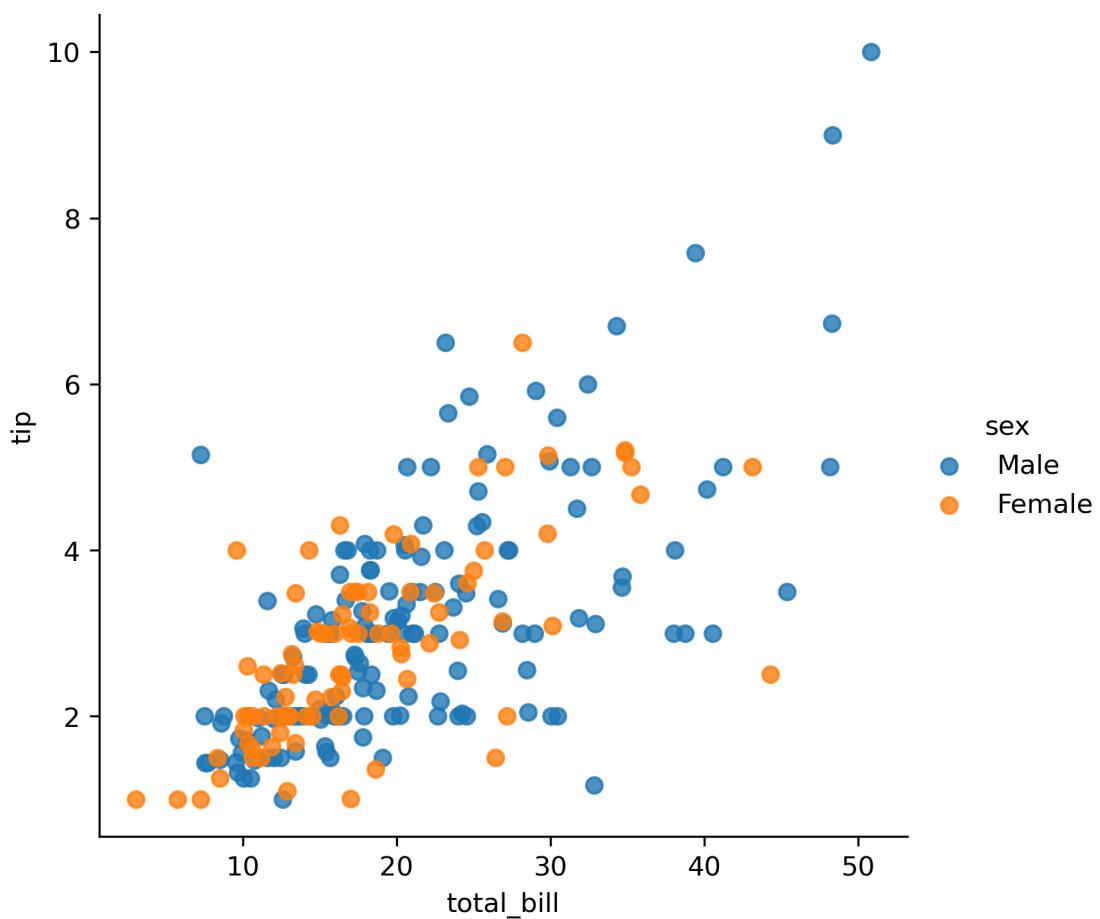


O parâmetro `split` (dividir) é crucial para reduzir a redundância e economizar espaço quando você usa o `hue` no `violinplot`.

O parâmetro `hue` pode ser passado para várias outras funções de plotagem também.

- O seu uso em um `lmplot`:

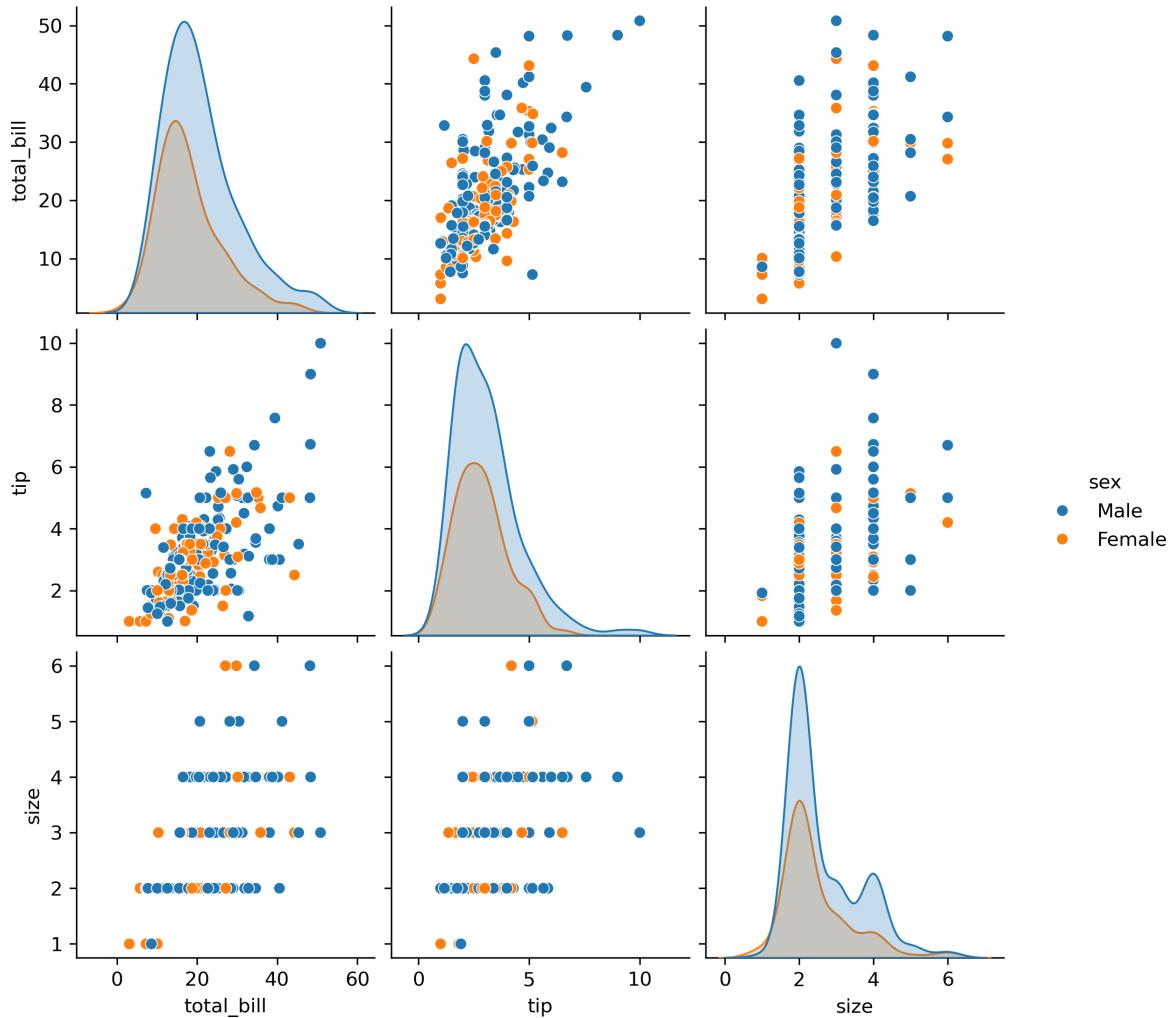
```
# Observe o uso de lmplot no lugar de regplot
scatter = sns.lmplot(
    x='total_bill',
    y='tip',
    data=tips,
    hue='sex',
    fit_reg=False
)
```



Podemos deixar nossas plotagens de pares um pouco mais significativas passando uma das variáveis de categoria como o parâmetro `hue`.

- Essa abordagem em nosso `pairplot`:

```
fig = sns.pairplot(tips, hue='sex')
```



4.4.3.2 Tamanho e formato

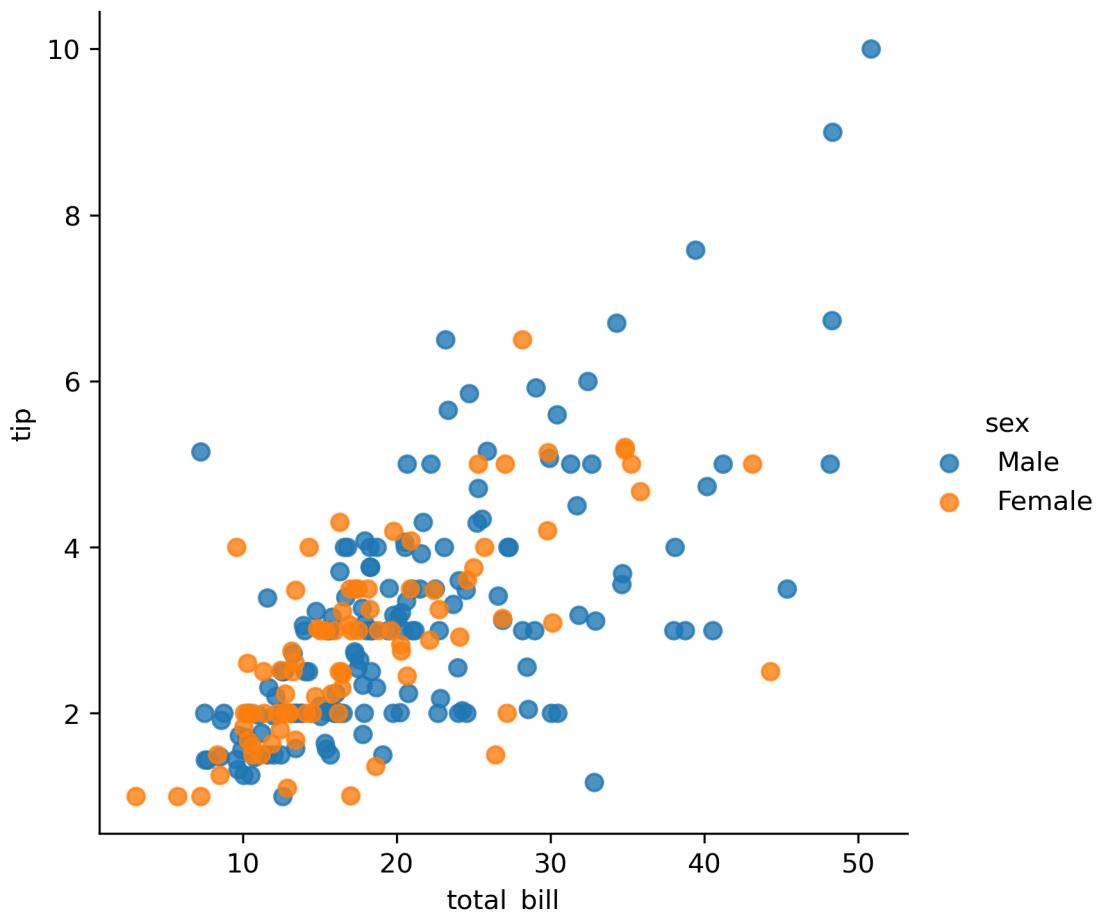
Trabalhar com tamanho de pontos pode ser outra forma de acrescentar mais informações em uma plotagem. No entanto, essa opção deve ser usada com moderação, pois o olho humano não é muito bom para comparar áreas.

Considere um exemplo de como o `seaborn` trabalha com chamadas de função da `matplotlib`. Se consultar a documentação de `lmpplot`, você verá que ela aceita parâmetros de nomes `catter`, `line scatter`, `line_kws`.

O parâmetro `scatter_kws` aceita um par chave-valor - um `dict` (dicionário) Python para ser mais exato. Pares chave-valor passados para `scatter_kws` são então passados para a função `plt.scatter` da `matplotlib`. É assim que acessaríamos o parâmetro `s` para alterar o tamanho dos pontos.

- Alterando tamanho dos pontos no lmplot:

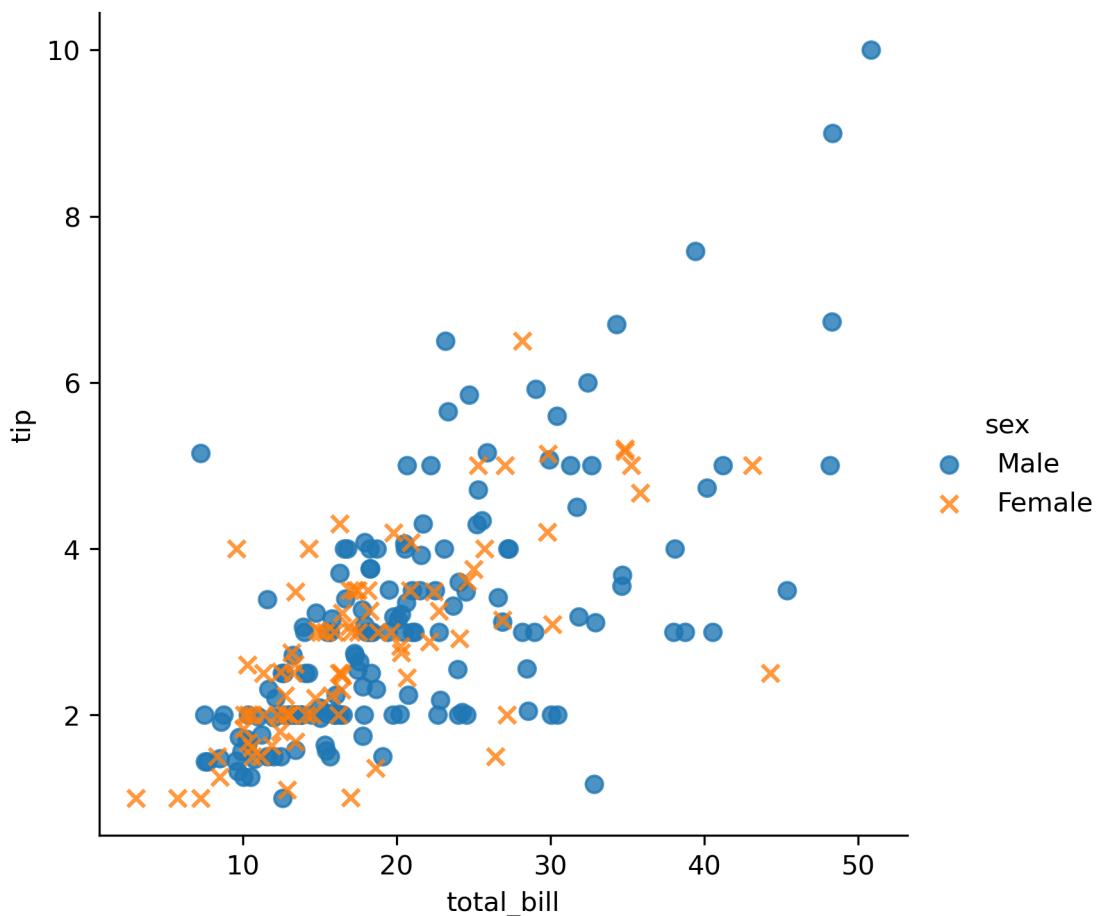
```
scatter = sns.lmplot(  
    x='total_bill',  
    y='tip',  
    data=tips,  
    fit_reg=False,  
    hue='sex',  
    scatter_kws={'s': 40}  
)
```



Além disso, ao trabalhar com diversas variáveis, às vezes ter dois elementos de plotagem que mostrem as mesmas informações será conveniente. O uso de cor e tamanho para distinguir valores diferentes da variável `sex`.

- Alterando Formato dos pontos no `lmplot`:

```
scatter = sns.lmplot(  
    x='total_bill',  
    y='tip',  
    data=tips,  
    fit_reg=False,  
    hue='sex',  
    markers=['o', 'x'],  
    scatter_kws={'s': 40}  
)
```



4.4.3.3 Facetas

“E se quiséssemos mostrar mais variáveis?”

“Ou se soubermos qual plotagem queremos para a nossa visualização, mas desejamos fazer várias plotagens sobre uma categoria?”

As facetas foram criadas para atender a essas necessidades. Em vez de ter de gerar subconjuntos de dados individuais e desenhar os eixos na figura (como visto no `subplots` do `matplotlib`, usando fatiamento de dados), as facetas `seaborn` podem cuidar dessa tarefa para você.

Para usar facetas, seus dados devem ser o que Hadley Wickham chama de “Tidy data” (Dados organizados), em que cada linha representa uma observação dos dados e cada coluna é uma variável (também conhecida como “dado longo” [long data]).

- *Tidy Data (Dados Organizados):*

É uma organização onde os dados são estruturados para análise. Segue três regras:

- cada **variável** em uma **coluna**;
- cada **observação** em uma **linha**;
- cada **valor** em uma **célula**.

Facilita o processamento vetorial no Pandas.

- *Wide Data (Largo):*

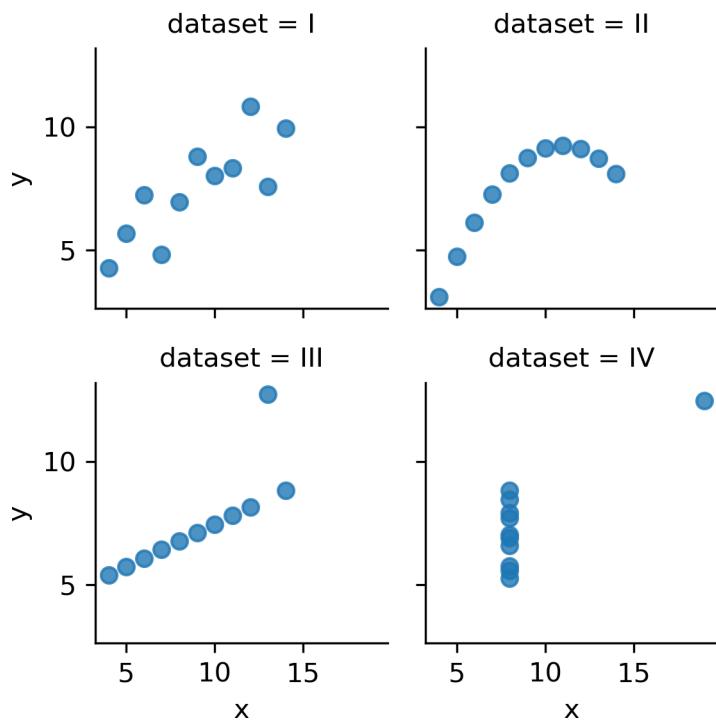
Neste formato, as informações de uma mesma categoria se espalham por várias colunas (ex: uma coluna para cada ano: 2021, 2022, 2023). É comum em planilhas financeiras por ser fácil de ler, mas dificulta a automação.

- *Long Data (Longo):*

Transforma as colunas do formato Wide em linhas. Usamos uma coluna para identificar a variável (ex: Ano) e outra para o conteúdo (ex: Valor). É o formato exigido pelo Seaborn para criar gráficos comparativos de forma automática.

Recriação dos dados do quarteto de Anscombe no **seaborn**:

```
anscombe_plot = sns.lmplot(  
    x='x',  
    y='y',  
    data=anscombe,  
    fit_reg=False,  
    col='dataset',  
    col_wrap=2,  
    height=2,  
    aspect=1  
)
```



Tudo que fizemos para criar essa visualização foi passar dois parâmetros para a função de gráfico de dispersão no **seaborn**.

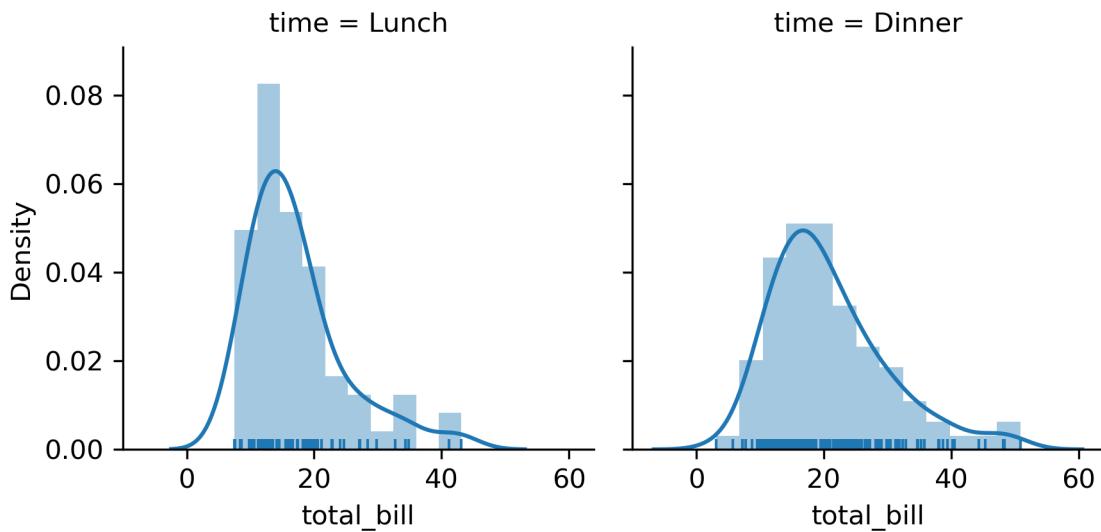
Parâmetros:

- **col:** É a variável que define a faceta da plotagem. Cria subplots por categoria;
- **col_wrap:** Cria uma figura com duas colunas. Se o parâmetro **col_wrap** não for usado, todas as quatro plotagens serão feitas na mesma linha.

Já discutimos as diferenças entre `lmplot` e `regplot`. `lmplot` é uma função no âmbito de figura. Em oposição, muitas das plotagens que criamos no `seaborn` são funções no âmbito de eixos. Isso significa que nem toda função de plotagem terá parâmetros `col` e `col_wrap` para facetas. Em vez disso, devemos criar um `FacetGrid` que saberá sobre qual variável deverá criar a faceta, e então fornecer o código da plotagem individual para cada faceta.

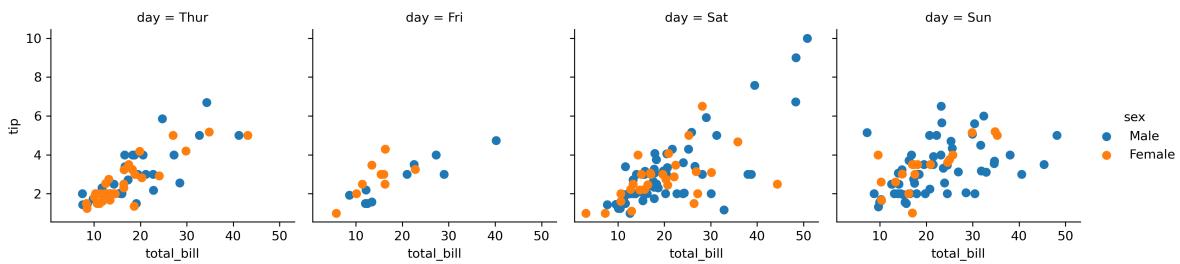
Uma faceta criada manualmente:

```
# Cria a FaceGrid
facet = sns.FacetGrid(          # Inicializa a grade baseada na coluna 'time'
    tips,
    col='time'
)
# Para cada valor de tempo, plota um histograma do total da conta
facet.map(
    sns.distplot,                 # Aplica a função de plotagem em cada subgráfico
    'total_bill',                # Define o tipo de gráfico (distribuição)
    rug=True                     # Adiciona marcas de densidade no eixo
)
```



As facetas individuais não precisam ser plotagens univariadas.

```
# Faceta multivariada, hue='sex'  
facet = sns.FacetGrid(      # Define a grade por 'day' e cores por 'sex'  
    tips,  
    col='day',  
    hue='sex'  
)  
facet = facet.map(          # Mapeia o gráfico de dispersão para a grade  
    plt.scatter,             # Define a função de plotagem (matplotlib)  
    'total_bill',            # Variável do eixo X  
    'tip'                   # Variável do eixo Y  
)  
facet = facet.add_legend()  # Adiciona a legenda baseada no parâmetro 'hue'
```



Podemos construir `seaborn` figure aos moldes de faceta.

Se quiser continuar trabalhando com `seaborn`, pode criar a mesma plotagem usando `lmplot`.

```
fig = sns.lmplot(  
    x='total_bill',      # Eixo X: valor da conta  
    y='tip',             # Eixo Y: valor da gorjeta  
    data=tips,           # Fonte dos dados  
    fit_reg=False,       # Remove a linha de regressão  
    hue='sex',            # Cor por categoria (gênero)  
    col='day'             # Cria subplots por dia  
)
```

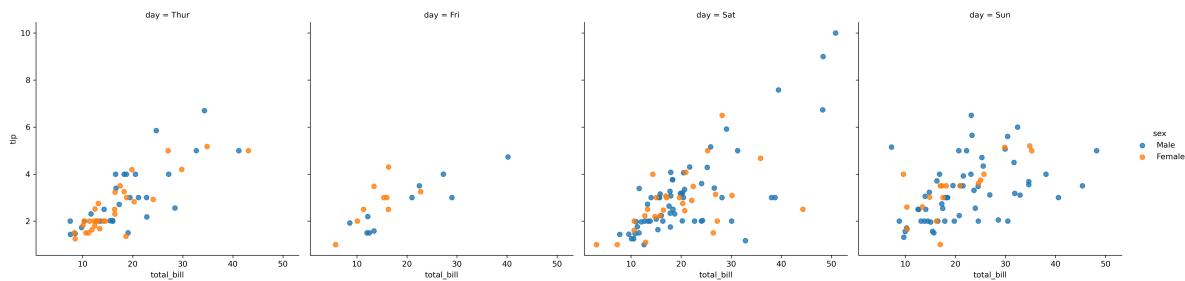


Figura 4: Plotagem do seaborn com facetas criadas manualmente contendo diversas variáveis.

Outra tarefa que pode ser feita com facetas é ter uma variável com faceta no eixo x e outra no eixo y. Fazemos isso passando um parâmetro `row`.

```
facet = sns.FacetGrid(  
    tips,                 # Fonte dos dados  
    col='time',            # Colunas por horário (almoço/jantar)  
    row='smoker',          # Linhas por fumante/não fumante  
    hue='sex',              # Cores por gênero  
    height=2.5,             # Altura de cada subplot (polegadas)  
    aspect=1                # Define a largura de cada gráfico  
    #como um múltiplo da altura  
    #relação altura x largura  
)  
  
facet.map(  
    plt.scatter,           # Tipo de gráfico (dispersão)  
    'total_bill',          # Variável do eixo X  
    'tip'                  # Variável do eixo Y  
)
```

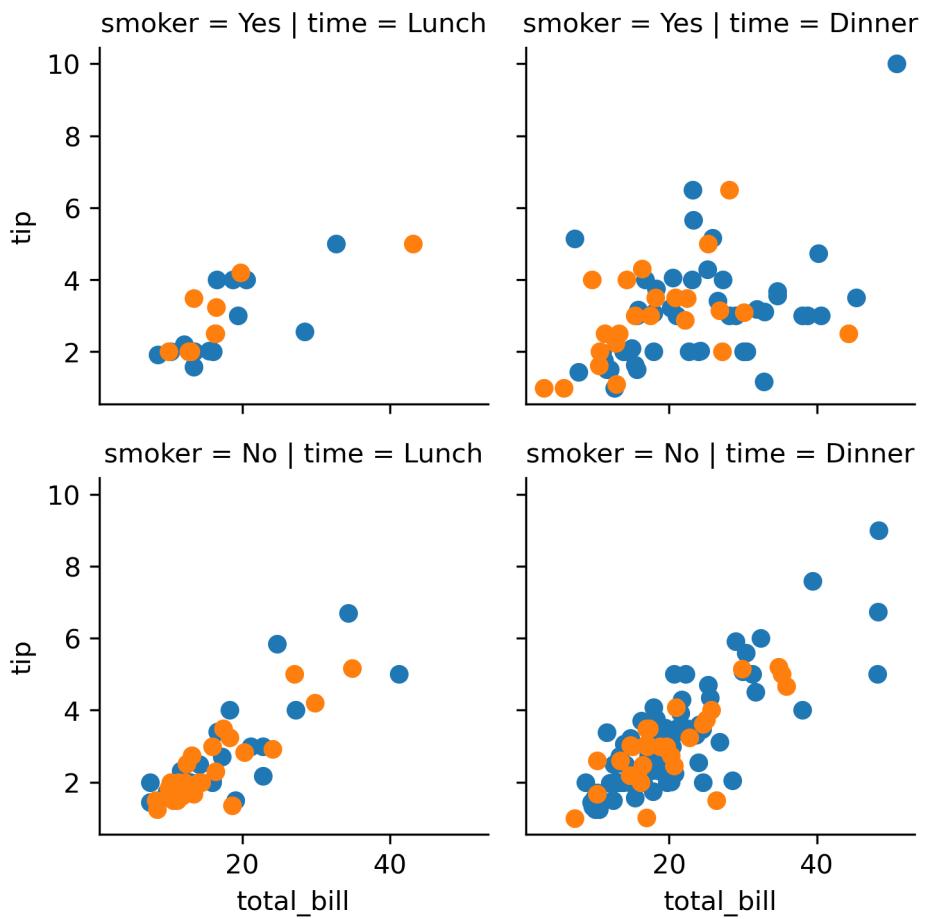


Figura 5: Plotagem do seaborn com facetas criadas manualmente com duas variáveis.

O `col` e o `row` funcionam como segmentadores automáticos do seu DataFrame:

- `col (Column)`: Filtra os dados em colunas verticais (cria subplots lado a lado).
- `row (Row)`: Filtra os dados em linhas horizontais (cria subplots um abaixo do outro).

O `seaborn` aplica um `groupby` automático nas variáveis escolhidas e desenha um gráfico para cada subconjunto resultante dessa interseção.

Se você não quiser que todos os elementos `hue` se sobreponham (isto é, quer esse comportamento nos gráficos de dispersão, mas não nas plotagens de `violin` ou `boxplot`), use a função `sns.factorplot`.

A função `factorplot` foi renomeada para `catplot` em versões recentes do `seaborn` para maior clareza, mas os parâmetros permanecem os mesmos.

- `catplot` para plotagem `violin`:

```
facet = sns.catplot(  
    x='day',           # Eixo X: categorias (dias)  
    y='total_bill',   # Eixo Y: valores numéricos  
    hue='sex',         # Cores por gênero  
    data=tips,         # Fonte dos dados  
    row='smoker',      # Linhas: filtro por fumante  
    col='time',         # Colunas: filtro por horário  
    kind='violin',     # Tipo de gráfico: violino  
    height=2.5,        # Altura de cada subplot (polegadas)  
    aspect=1           # Define a largura de cada gráfico  
)
```

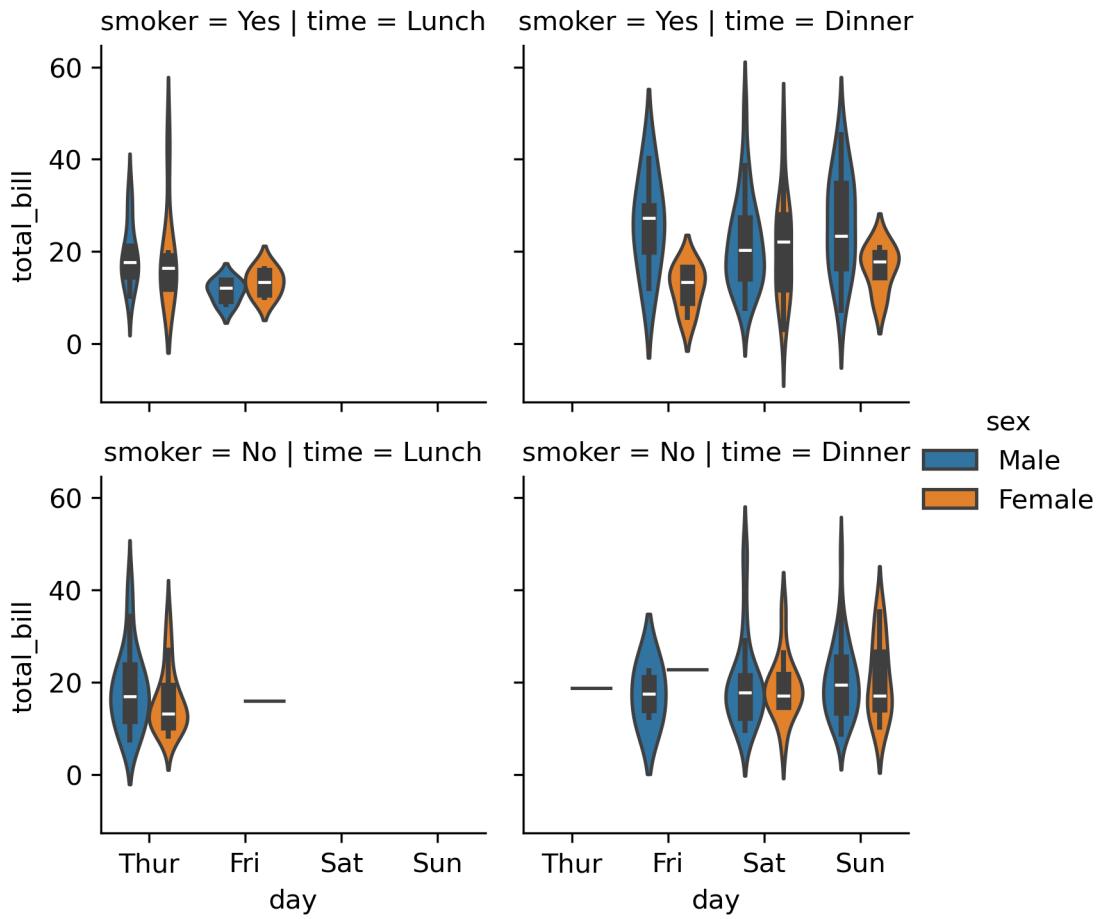


Figura 6: Plotagem do seaborn com facetas criadas manualmente com duas variáveis violin.

- catplot para plotagem boxplot:

```

facet = sns.catplot(
    x='day',                      # Eixo X: categorias
    y='total_bill',                # Eixo Y: valores
    hue='sex',                     # Cores por gênero
    data=tips,                     # Dados
    row='smoker',                  # Filtro em linhas
    col='time',                    # Filtro em colunas
    kind='box',                     # Tipo de gráfico: Boxplot
    height=2.5,                    # Altura de cada subplot (polegadas)
    aspect=1                        # Define a largura de cada gráfico
)

```

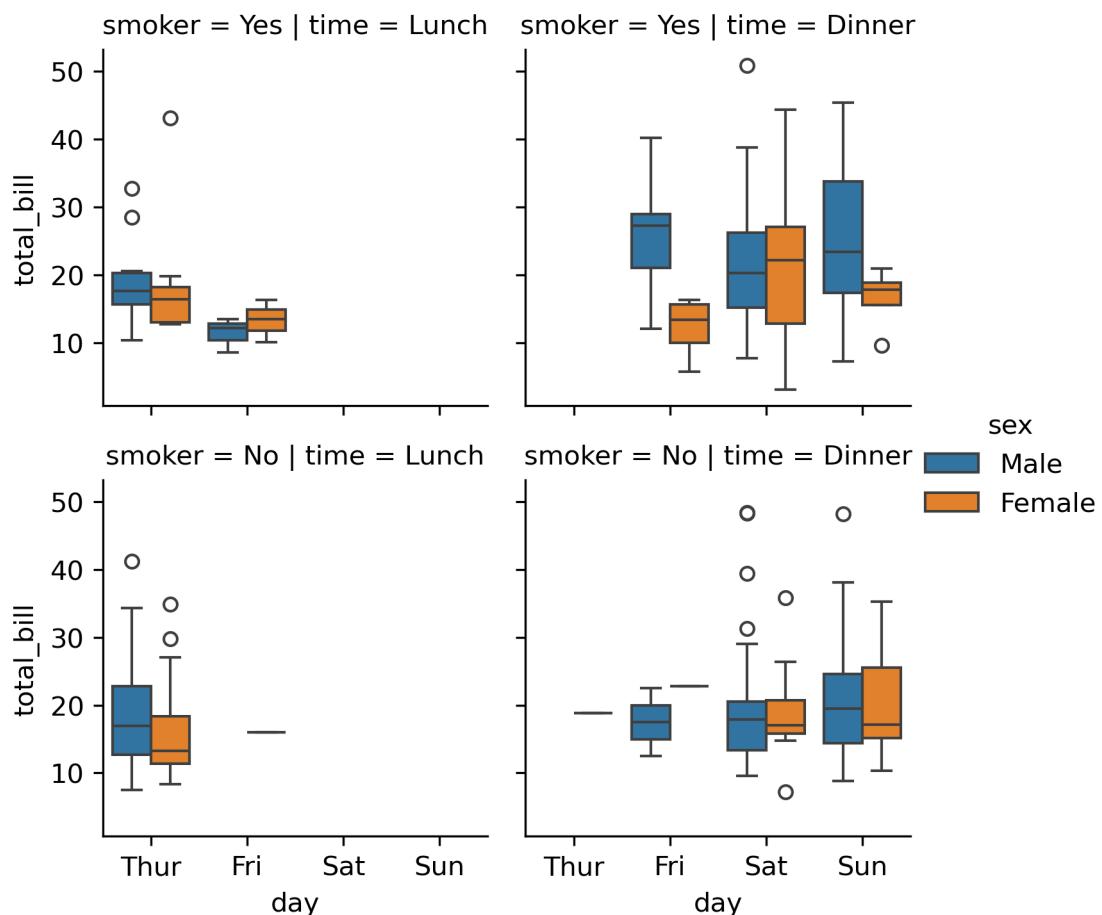


Figura 7: Plotagem do seaborn com facetas criadas manualmente com duas variáveis boxplot.

4.5 Objetos do Pandas

Os objetos do Pandas também tem as próprias funções de plotagem.

Assim como `seaborn`, as funções de plotagem incluídas no Pandas são apenas *wrappers* em torno da `matplotlib`, com valores predefinidos.

Em geral, a plotagem usando o Pandas é feita de acordo com as funções `DataFrame.plot.TIPO_DA_PLOTAGEM` ou `Series.plot.TIPO_DA_PLOTAGEM`.

4.5.1 Histogramas

Os histogramas podem ser criados usando a função `Series.plot.hist` ou a função `DataFrame.plot.hist`.

- Em uma `Series`:

```
fig, ax = plt.subplots()
ax = tips['total_bill'].plot.hist()
```

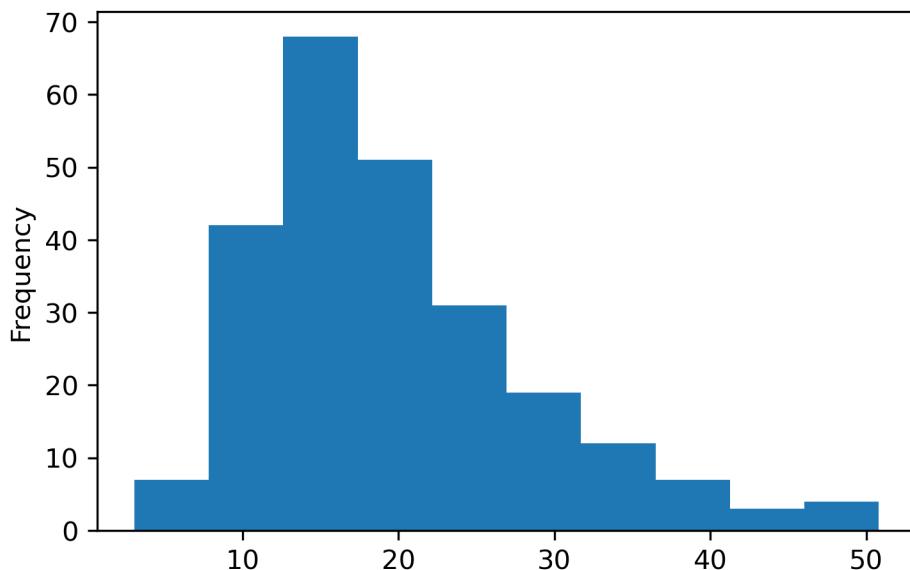


Figura 8: Histograma de uma Series do Pandas.

- Em um DataFrame:

```
# Define um canal de transparéncia alpha,
#de modo que possamos ver através das barras que se sobrepõem.

fig, ax = plt.subplots()
ax = tips[['total_bill','tip']].plot.hist(
    alpha=0.5,
    bins=20,
    ax=ax
)
```

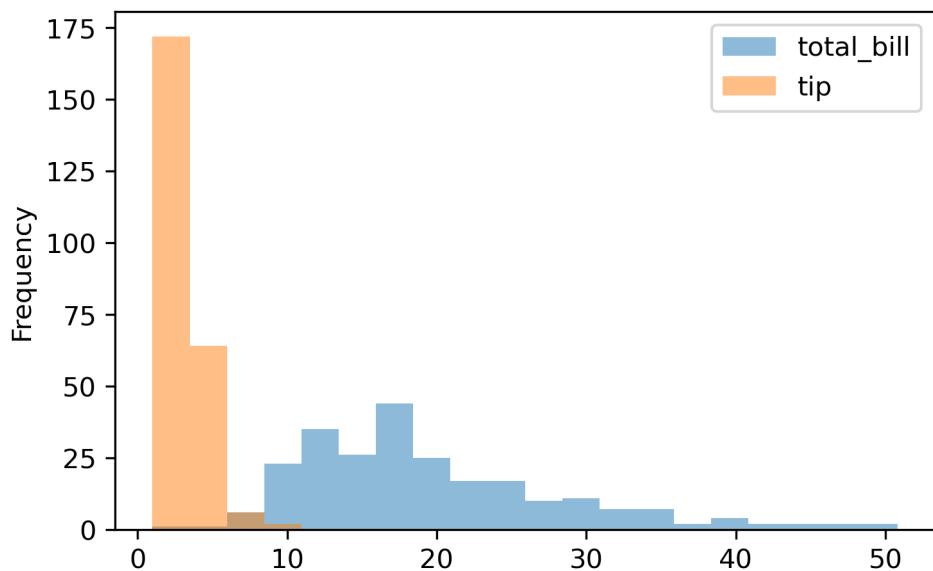


Figura 9: Histograma de um DataFrame do Pandas.

4.5.2 Plotagem de densidade

A plotagem de estimativa de densidade por kernel (densidade) pode ser criada com a função `DataFrame.plot.kde()`.

```
fig, ax = plt.subplots()
ax = tips['tip'].plot.kde()
```

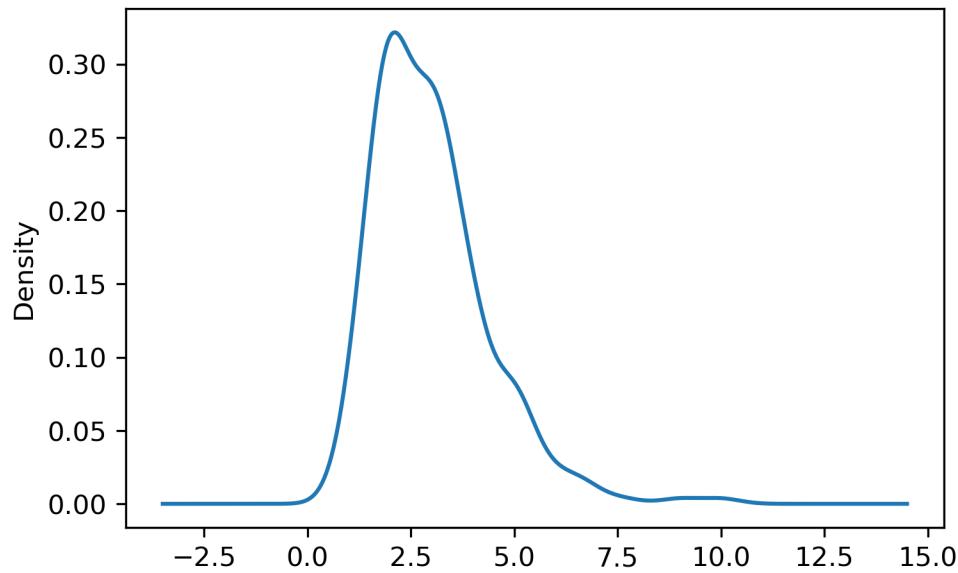


Figura 10: Plotagem de KDE do Pandas.

4.5.3 Gráfico de dispersão

Os gráficos de dispersão são criados usando a função `DataFrame.plot.scatter`.

```
fig, ax = plt.subplots()
ax = tips.plot.scatter(
    x='total_bill',
    y='tip',
    ax=ax
)
```

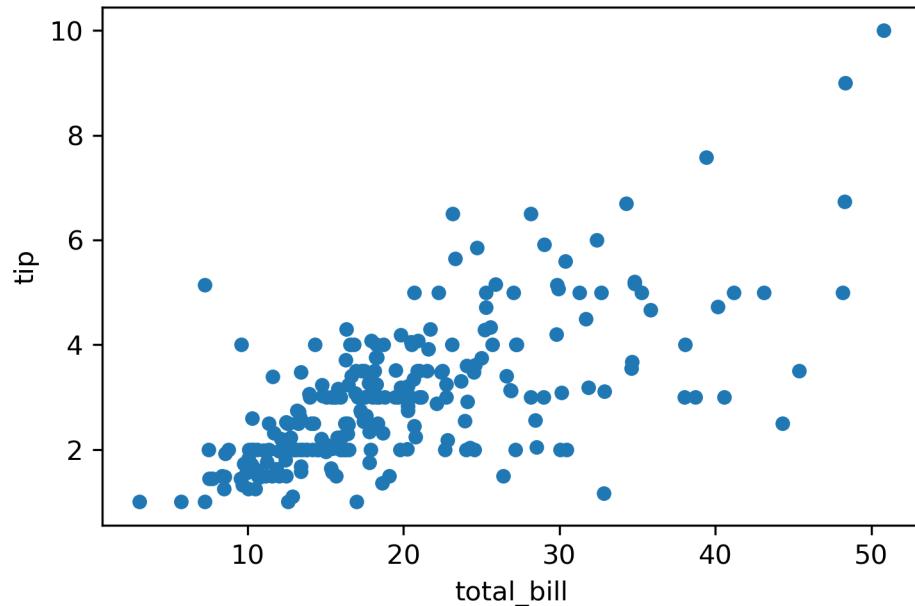


Figura 11: Gráfico de dispersão do Pandas.

4.5.4 Plotagem hexbin

As plotagens hexbin são criadas usando a função `DataFrame.plot.hexbin`.

```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(
    x='total_bill',
    y='tip',
    ax=ax
)
```

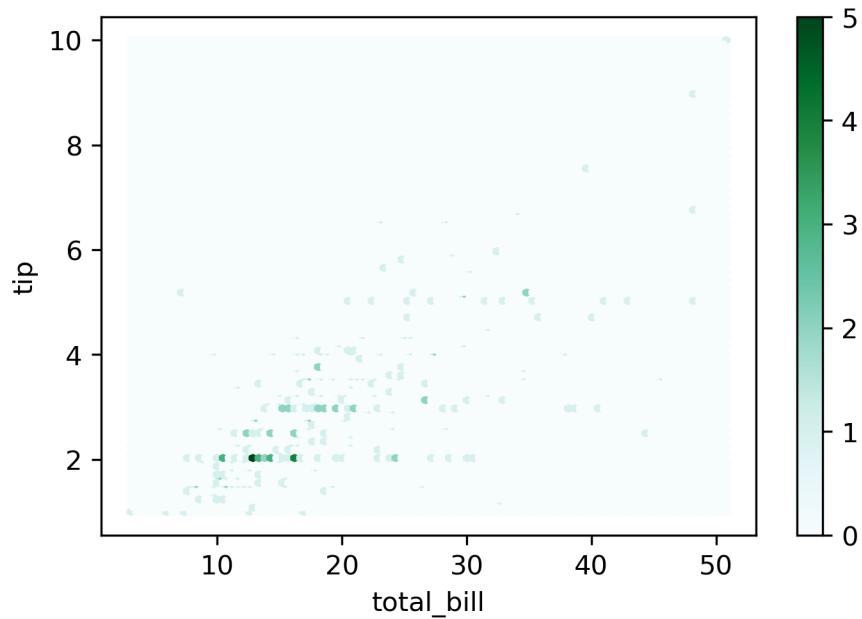


Figura 12: Plotagem hexbin Pandas.

O tamanho da grade pode ser ajustado com o parâmetro `gridsize`.

```
fig,ax = plt.subplots()
ax = tips.plot.hexbin(
    x='total_bill',
    y='tip',
    gridsize=10,
    ax=ax
)
```

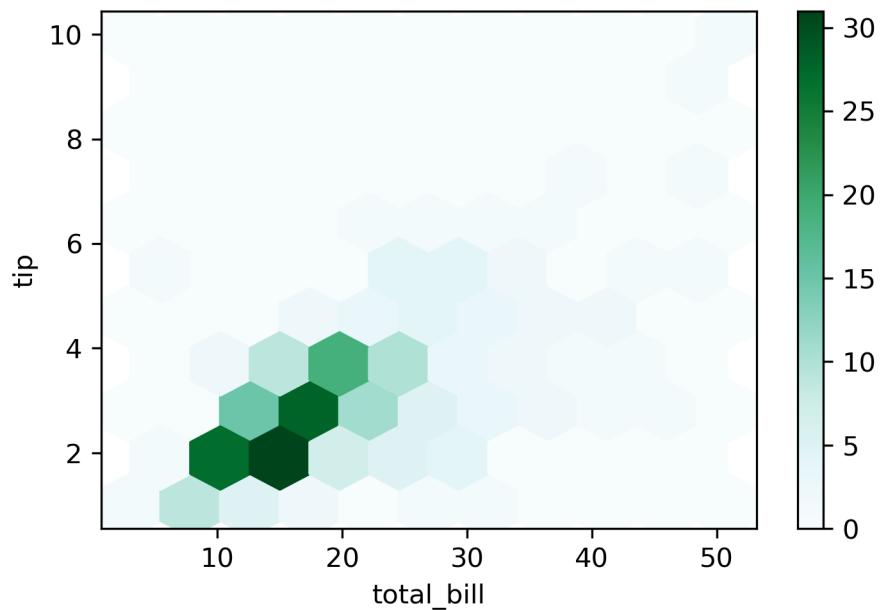


Figura 13: Plotagem hexbin do Pandas com tamanho de grade modificado.

4.5.5 Gráfico de caixa

Os gráficos de caixa (boxplot) são criados com a função `DataFrame.plot.box`.

```
fig, ax = plt.subplots()
ax = tips.plot.box(
    ax=ax
)
```

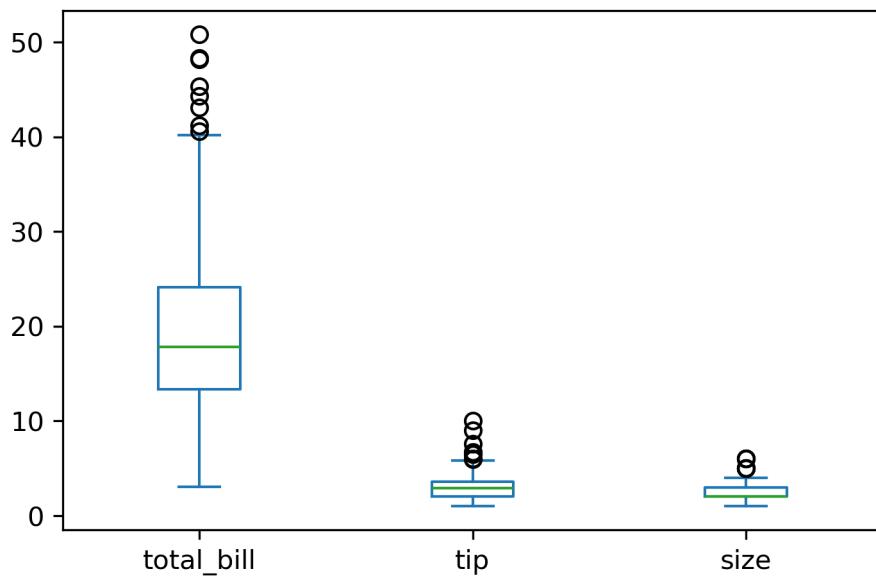


Figura 14: Plotagem de caixas do Pandas.

4.6 Temas e estilos do seaborn

Todas as plotagens do `seaborn` mostradas nesse capítulo usaram os estilos de plotagem default.

Podemos alterar o estilo de plotagem com a função `sns.set_style`.

Em geral, essa função é executada comente uma vez, no início de seu código; todas as plotagens subsequentes usarão o mesmo conjunto de estilos.

Os estilos incluídos no `seaborn` são: * `darkgrid` * `whitegrid` * `dark` * `white` * `ticks`

A Figura 15 mostra uma plotagem básica, enquanto a Figura 16 exibe uma plotagem com estilo `whitegrid`.

```
# Plotagem inicial para comparação
fig, ax = plt.subplots()
ax = sns.violinplot(
    x='time',
    y='total_bill',
    hue='sex',
    data=tips,
    split=True
)
```

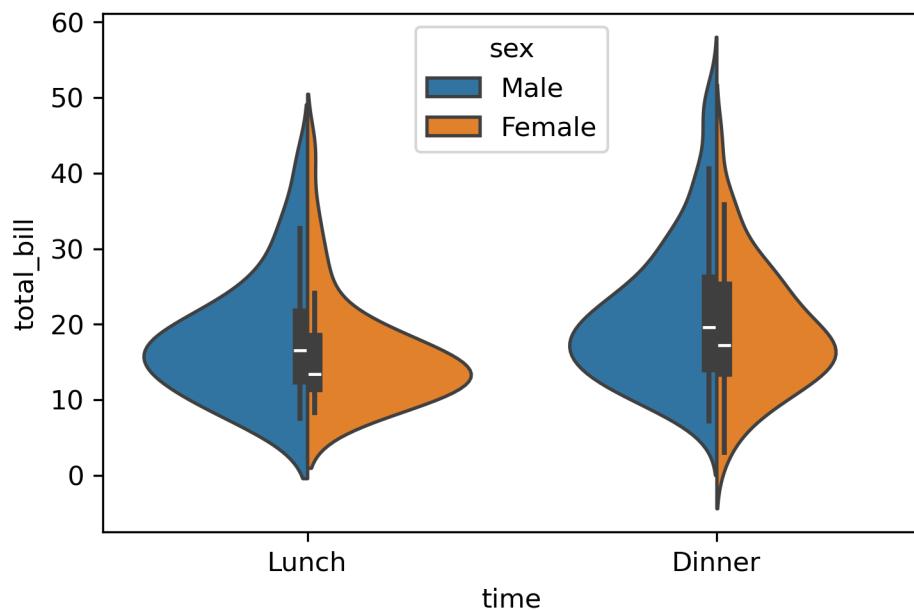


Figura 15: Estilo básico do seaborn.

```
# Define o estilo e faz a plotagem
sns.set_style('whitegrid')
fig, ax = plt.subplots()
ax = sns.violinplot(
    x='time',
    y='total_bill',
    hue='sex',
    data=tips,
    split=True
)
```

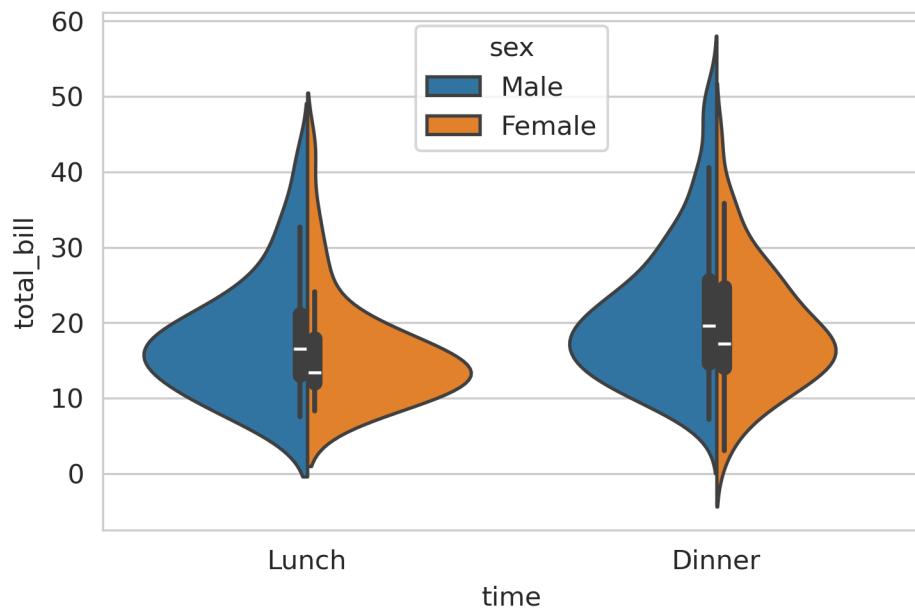


Figura 16: Estilo whitegrid do seaborn.

O código a seguir mostra a aparência de todos os estilos:

```
fig=plt.figure()
seaborn_styles=['darkgrid','whitegrid',
'dark','white','ticks']
for idx, style in enumerate(seaborn_styles):
    plot_position = idx + 1
    with sns.axes_style(style):
        ax = fig.add_subplot(2,3,plot_position)
        violin = sns.violinplot(
            x='time',
            y='total_bill',
            data=tips,
            ax=ax
        )
        violin.set_title(style)
fig.tight_layout()
```

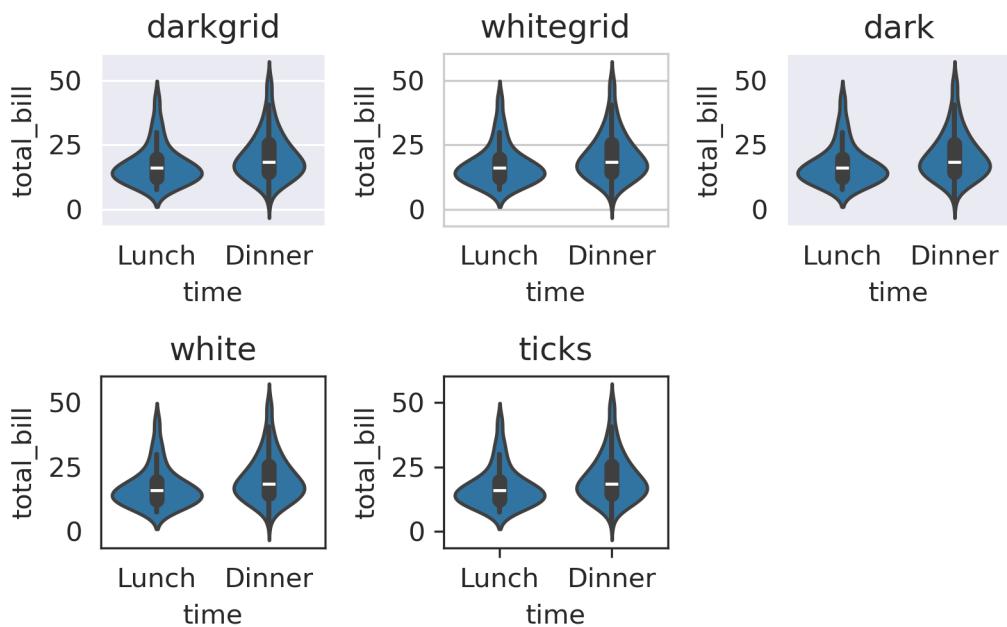


Figura 17: Todos os estilos seaborn.

4.7 Conclusão

A visualização de dados é parte integrante da análise de dados exploratória e da apresentação dos dados.

Este capítulo apresentou uma introdução das diversas maneiras de explorar e apresentar seus dados. À medida que avançamos no livro, conheceremos visualizações mais complexas.

Há uma variedade de recursos de plotagem e visualização disponíveis na internet. A documentação `seaborn`, a documentação de visualização `Pandas` e a documentação da `matplotlib` descrevem maneiras de ajustar melhor as suas plotagens (por exemplo, cores, espessura das linhas, posicionamento de legendas, anotações na figura). Outros recursos incluem o `colorbrewer` para ajudar a selecionar bons esquemas de cores. As bibliotecas de plotagem mencionadas neste capítulo também têm diversos esquemas de cores que podem ser usados para dar ênfase ao conteúdo de suas visualizações.