Estudo de Python e dados

Sergio Pedro Rodrigues Oliveira

26 September 2025

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

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 reprodutibilidade.

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 contéudo.

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;
# Nosso 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 esta 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 tupla[1] na qual o primeiro valor é o número de linhas e o segundo é a quantidade de colunas.

  Com base nesse resultado anteior, 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)
```

[1] Uma 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:

Table 1: Informações do método info() do Pandas

Informação	Discrição
Tipo de índice	Informações sobre o índice (por exemplo, RangeIndex).
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. Isso é crucial para identificar dados faltantes.
Dtype (tipo de dado) de cada coluna	Semelhante ao atributo dtype, 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
```

Table 2: Tipos do Pandas versus tipos de Python

Tipo do Pandas	Tipo de Python	Discrição
object	string	Cadeia de caracteres, usado para representar texto.
int64	int	Números inteiros.
float64	float	Números com decimais.
datetime64	datetime	`datetime` 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 específicar 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 Table 3 apresenta uma visão geral rápida dos diversos métodos.

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

Método para obtenção de subconjuntos	Discrição
`loc`	Subconjunto baseado no rótulo do índice (**nome** da linha).
`iloc`	Subconjunto baseada no índice da linha (**número** da linha).
`ix` (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.

```python
# 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
```

```python
# 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 entro o início e o fim (**inclusive à esquerda, não inclusive a direita**) será criado, a comenos 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 achamado, 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:

```
# Fatia 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:

```
# Fatia 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 facil 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 subconjuntode 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 dataframe 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 0x7666ccb8a0d0>
```

- 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 0x7666ccb8bb10>
```

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 váriavel? 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-continente, 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 linhaque os "nomes das colunas" de expectativa de vida e GPD. 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 Condatodes 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 descobretas 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()
```

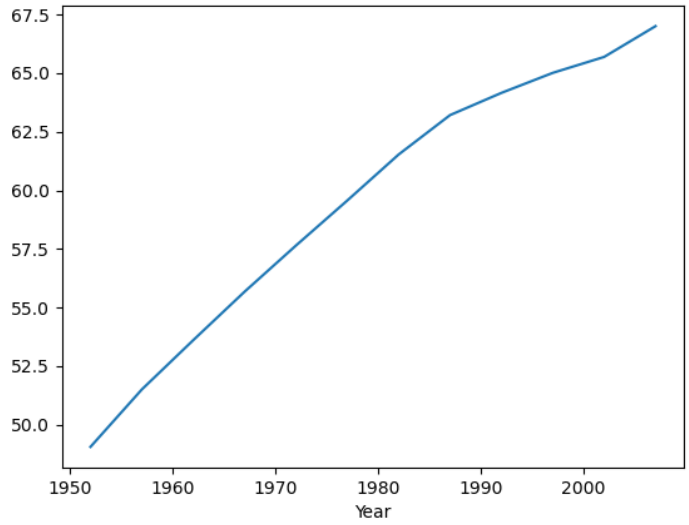

Figure 1: Plotagem básica no Pandas mostrando a expectativa de vida média no tempo.

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 Table 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

1. O que acontecerá se você usar outros contêineres Python como `list`, `tuple`, `dict` ou ate 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 específicar 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 Table 4.

Table 4: Alguns dos atributos de uma Series

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

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 cientifíco 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 Table 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) aum cálculo ou uma observação em particular.

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

Métodos de `Series`	Descrição
`append`	Concatena duas ou mais `Series`.
`corr`	Calcula uma correlação com outra `Series`.*
`cov`	Calcula uma covariância com outra `Series`.*
`describe`	Calcula estatísticas resumidas.*
`drop_duplicates`	Devolve uma `Series` sem duplicações.
`equals`	Determina se uma `Series` tem os mesmos elementos.
`get_values`	Obtém valores da `Series`; o mesmo que o atributo `values`.
`hist`	Desenha um histograma.
`isin`	Verifica se valores estão contidos em uma `Series`.
`min`	Devolve o valor mínimo.
`max`	Devolve o valor máximo.
`mean`	Devolve a média aritmética.
`median`	Devolve a mediana.
`mode`	Devolve a(s) moda(s).
`quantile`	Devolve o valor em um dado quantil.
`replace`	Substitui valores da `Series` por um valor especificado.
`sample`	Devolve uma amostra aleatória de valores da `Series`.
`sort_values`	Ordena valores.
`to_frame`	Converte uma `Series` em um `DataFrame`.
`transpose`	Devolve a transposta.
`unique`	Devolve um `numpy.ndarray` 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 `bool`s 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,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 Table 6 resumo os diversos tipos de métodos para obtenção de subconjuntos.

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

Sintaxe	Resultado da seleção
`df[column_name]`	Única coluna.
`df[[column1,column2,...]]`	Várias colunas.
`df.loc[row_label]`	Linha pelo rótulo do índice da linha (nome da linha).
`df.loc[[label1,label2,...]]`	Várias linhas pelos rótulos do índice.
`df.iloc[row_number]`	Linha pelo número da linha.
`df.iloc[[row1,row2,...]]`	Várias linhas pelos números das linhas.
`df.ix[label_or_number]`	Linha pelo rótulo do índice ou pelo número.
`df.ix[lab_num1,lab_num2,...]`	Várias linhas pelos rótulos de índice ou pelos números.
`df[bool]`	Linha baseada em `bool`.
`df[bool1,bool2,...]`	Várias linhas baseadas em `bool`.
`df[start:stop:step]`	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 assoaciado 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 FranklinRosaline Franklin  1920-07-251920-07-25
1              William GossetWilliam Gosset  1876-06-131876-06-13
2  Florence NightingaleFlorence Nightingale  1820-05-121820-05-12
3                    Marie CurieMarie Curie  1867-11-071867-11-07
4                Rachel CarsonRachel Carson  1907-05-271907-05-27
5                       John SnowJohn Snow  1813-03-151813-03-15
6                    Alan TuringAlan Turing  1912-06-231912-06-23
7                  Johann GaussJohann Gauss  1777-04-301777-04-30

                   Died  Age                          Occupation
0  1958-04-161958-04-16   74                      ChemistChemist
1  1937-10-161937-10-16  122           StatisticianStatistician
2  1910-08-131910-08-13  180                        NurseNurse
3  1934-07-041934-07-04  132                      ChemistChemist
4  1964-04-141964-04-14  112                  BiologistBiologist
5  1858-06-161858-06-16   90                  PhysicianPhysician
6  1954-06-071954-06-07   82  Computer ScientistComputer Scientist
7  1855-02-231855-02-23  154          MathematicianMathematician
```

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
```

```python
# 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 Table 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 `datatime` 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, poderiamos 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 descartar 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
# Específique 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

3.6.3 Excel

3.6.4 Formato feather para interface com R

3.6.5 Outros tipos de saída de dados

3.7 Conclusão

4 Introdução à plotagem