

Estudo de Python e dados II

Manipulação de dados: Preparação de dados, dados ausentes e Tidy data

Sergio Pedro Rodrigues Oliveira

12 fevereiro 2026

SUMÁRIO

1	Objetivo	1
2	Preparação dos dados	2
2.1	Introdução	2
	Mapa conceitual	2
	Objetivos	2
2.2	Tidy data	3
	2.2.1 Combinando conjuntos de dados	4
2.3	Concatenação	5
	2.3.1 Adicionando linhas	5
	2.3.1.1 Concatenar Series	8
	2.3.1.2 Ignorando o índice	11
	2.3.2 Adicionando colunas	12
	Concatenando colunas - axis	12
	Subconjuntos colunas	12
	Adicionar uma única coluna	13
	Reiniciando índices colunas	14
	2.3.3 Concatenação com índices diferentes	15
	2.3.3.1 Concatenando linhas com colunas diferentes	15
	2.3.3.2 Concatenando colunas com linhas diferentes	18
2.4	Combinando vários conjuntos de dados	20
	2.4.1 merge um a um (one-to-one)	24
	2.4.2 merge de muitos para um (many-to-one)	25
	2.4.3 merge de muitos para muitos (many-to-many)	26
	2.4.4 Resumo	29
2.5	Conclusão preparação dos dados	31
3	Dados Ausentes	32
3.1	Introdução	32
	Mapa conceitual	32
	Objetivos	32
3.2	O que é um valor NaN?	33
3.3	De onde vêm os valores ausentes?	35
	3.3.1 Carga de dados	35
	na_values	36
	keep_default_na	37
	na_filter	38
	3.3.2 Dados combinados	39
	3.3.3 Valores de entrada do usuário	41
	3.3.4 Reindexação	43

3.4	Trabalhando com dados ausentes	46
3.4.1	Encontrando e contando dados ausentes	46
Linhas		46
Colunas		48
3.4.2	Limpando dados ausentes	50
3.4.2.1	Recodificar/substituir	51
3.4.2.2	Preenchimento para a frente (<i>forward</i>)	52
3.4.2.3	Preenchimento para trás (<i>backward</i>)	53
3.4.2.4	Interpolação	54
3.4.2.5	Descartando valores ausentes	55
3.4.3	Cálculos com dados ausentes	57
3.5	Conclusão dados ausentes	59
4	Tidy data (dados organizados)	60
4.1	Introdução	60
Mapa conceitual		60
Objetivos		61
4.2	Coluna contém valores, e não variáveis	62
4.3	Colunas contendo diversas variáveis	62
4.4	Variáveis tanto em linhas quanto em colunas	62
4.5	Várias unidades de observação em uma tabela (normalização)	62
4.6	Unidade de observação em várias tabelas	62
4.7	Conclusão <i>Tidy data</i>	62

LISTA DE FIGURAS

1	Preparação de dados, principais tópicos	2
---	---	---

LISTA DE TABELAS

1	Como o parâmetro <i>how</i> do Pandas se relaciona com o SQL	22
2	Comportamento do parâmetro <i>how</i> no merge	22
3	Parâmetros do <code>merge</code>	23
4	Métodos para limpeza de dados.	50

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 Preparação dos dados

2.1 Introdução

A essa altura, você deverá ser capaz de carregar dados no Pandas e fazer algumas visualização básica. Essa parte do livro tem como foco várias tarefas de limpeza dos dados. Começaremos com a preparação de um conjunto de dados para análise por meio da combinação de diversos conjuntos.

Mapa conceitual

1. Conhecimento prévio
 - a) Carga de dados;
 - b) Obtenção de subconjuntos de dados;
 - c) Funções e métodos de classe.

Objetivos

Este capítulo abordará:

1. *Tidy data* (dados organizados);
2. Concatenação de dados;
3. Combinação (merge) de conjunto de dados.

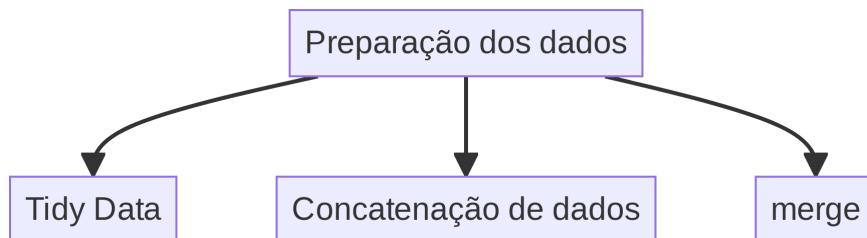


Figura 1: Preparação de dados, principais tópicos.

2.2 Tidy data

Hadley Wickham, um dos mais proeminentes membros da comunidade **R**, fala sobre a ideia de *tidy data* (dados organizados).

Com efeito, ele escreveu um artigo sobre esse conceito no *Journal of Statistical Software*. *Tidy data* é um framework para estruturar conjuntos de dados a fim de que sejam facilmente analisados. É usado principalmente como um objetivo a que devemos visar quando limpamos os dados. Depois que você compreender o que é o conceito de *tidy data*, esse conhecimento fará com que a coleta de dados seja muito mais fácil.

Então o que é *tidy data*? O artigo de Hadley Wickham o define como um conceito que atende aos seguintes critérios:

1. “Cada observação deve formar uma linha” (Observation)

Uma observação é o conjunto de todas as medidas feitas em uma única unidade (ex: uma pessoa em um exame, um país em um ano específico).

- O erro comum: Repetir a mesma observação em várias colunas ou espalhar os dados de uma mesma pessoa em tabelas diferentes sem necessidade.
- O modo Tidy: Se você está analisando a saúde de pacientes, cada linha deve representar um paciente em um momento específico.

2. “Cada variável deve formar uma coluna” (Variable)

Uma variável é um atributo que você mede (ex: Peso, Data, Temperatura).

- O erro comum: Ter colunas chamadas “Janeiro”, “Fevereiro” e “Março”. Aqui, o nome da variável é “Mês”, e Janeiro/Fevereiro são apenas valores.
- O modo Tidy: Criar uma coluna única chamada Mes onde os valores são listados.

3. “Cada tipo de unidade observacional forma uma tabela”

Esta regra foca na organização macro.

Sobre o terceiro critério do tidy data, as observações devem ser coerentes com a tabela, tornando-a objetiva quanto ao tipo de informação que deve armazenar. A ideia é que a tabela tenha um propósito único; ao misturar assuntos diferentes em uma mesma estrutura, fere-se a normalização dos dados.

- O erro comum: Misturar dados de “Clientes” com dados de “Vendas” na mesma tabela, causando redundância (ex: repetir o endereço do cliente toda vez que ele compra algo).
- O modo Tidy: Ter uma tabela para Clientes e outra para Vendas, relacionando-as por um ID. Isso facilita a manutenção e evita erros de digitação.

2.2.1 Combinando conjuntos de dados

Começaremos com o último critério de Hadley Wickham para *tidy data*: “cada tipo de unidade de observação forma uma tabela”.

Quando os dados estão organizados, é necessário combinar várias tabelas para responder a uma pergunta. Por exemplo, pode haver uma tabela separada que armazene informações de empresas e outra tabela contendo preços de ações. Se quisermos observar os preços de todas as ações no mercado de tecnologia, talvez antes tenhamos de encontrar todas as empresas de tecnologia na tabela de informações sobre empresas e então combinar esses dados com os preços das ações a fim de obter as informações de que precisamos para responder à nossa pergunta.

Os dados podem ter sido separados em tabelas distintas para reduzir a quantidade de informações redundantes (não precisamos armazenar informações sobre as empresas em cada entrada de preço das ações), mas essa organização implica que, como analistas de dados, teremos de combinar os dados relevantes por conta própria para responder à nossa pergunta.

Em outras ocasiões, um único conjunto de dados pode estar dividido em várias partes. Por exemplo, em dados de séries temporais, cada data pode estar em um arquivo separado. Em outro caso, um arquivo pode ter sido separado em partes para que os arquivos individuais fossem menores. Talvez você precise combinar dados de diversas origens para responder a uma pergunta (por exemplo, como combinar latitudes em longitudes com CEPs). Nos dois casos, você terá que combinar dados em um único dataframe de análise.

2.3 Concatenação

Uma das maneiras mais (conceitualmente) fáceis de combinar dados é por meio da concatenação.

Podemos pensar na concatenação como uma junção de linhas ou colunas em seus dados. Essa abordagem é possível se seus dados estiverem separados em partes ou se você fez um cálculo que queira concatenar ao seu conjunto de dados existente.

A concatenação é feita usando a função concat do Pandas.

2.3.1 Adicionando linhas

Vamos começar com alguns conjuntos de dados de exemplo para que você veja o que realmente acontece.

```
import pandas as pd

df1 = pd.read_csv("Cap_04-Preparacao_dados/01-Concatenacao(concat_1.csv")
df2 = pd.read_csv("Cap_04-Preparacao_dados/01-Concatenacao(concat_2.csv")
df3 = pd.read_csv("Cap_04-Preparacao_dados/01-Concatenacao(concat_3.csv")
```

```
print("dataframe_csv_1:")
print(df1)
print("dataframe_csv_2:")
print(df2)
print("dataframe_csv_3:")
print(df3)
```

```
dataframe_csv_1:
     A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3
dataframe_csv_2:
     A    B    C    D
0   a4   b4   c4   d4
1   a5   b5   c5   d5
2   a6   b6   c6   d6
3   a7   b7   c7   d7
dataframe_csv_3:
```

	A	B	C	D
0	a8	b8	c8	d8
1	a9	b9	c9	d9
2	a10	b10	c10	d10
3	a11	b11	c11	d11

A operação de empilhar dataframes uns sobre os outros é feita com a função `concat` do Pandas. Todos os dataframes a serem concatenados são passados em uma `list`.

```
row_concat = pd.concat([df1,df2,df3])
print(row_concat)
```

	A	B	C	D
0	a0	b0	c0	d0
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
0	a4	b4	c4	d4
1	a5	b5	c5	d5
2	a6	b6	c6	d6
3	a7	b7	c7	d7
0	a8	b8	c8	d8
1	a9	b9	c9	d9
2	a10	b10	c10	d10
3	a11	b11	c11	d11

Como podemos ver, `concat` empilha cegamente os dataframes. Se observar os nomes das linhas (isto é, seus índices), verá que eles são apenas uma versão empilhada dos índices originais das linhas.

Se aplicarmos os diversos métodos para obtenção de subconjuntos, os subconjuntos serão obtidos conforme esperado.

```
#Obtém o subconjunto da quarta linha do dataframe concatenado
print(row_concat.iloc[3,:])
```

```
A    a3
B    b3
C    c3
D    d3
Name: 3, dtype: object
```

O que acontece se você usar `loc` para obter o subconjunto do novo dataframe?

A função `loc` pega os subconjuntos pelo rótulo (nome da linha), logo pega todos os rótulos com mesmo nome, pegando assim 3 linhas diferentes com mesmo nome.

```
print(row_concat.loc[3])
```

	A	B	C	D
3	a3	b3	c3	d3
3	a7	b7	c7	d7
3	a11	b11	c11	d11

2.3.1.1 Concatenar Series

Anteriormente apresentamos o processo de criar uma **Series**. No entanto, se criássemos uma nova série para concatenar em um dataframe, ela não seria concatenada corretamente.

```
#Criar uma nova linha de dados
new_row_series = pd.Series(['n1','n2','n3','n4'])
print(new_row_series)
```

```
0    n1
1    n2
2    n3
3    n4
dtype: object
```

```
#Tentando adicionar a nova linha em um dataframe
print(pd.concat([df1,new_row_series]))
```

	A	B	C	D	0
0	a0	b0	c0	d0	NaN
1	a1	b1	c1	d1	NaN
2	a2	b2	c2	d2	NaN
3	a3	b3	c3	d3	NaN
0	NaN	NaN	NaN	NaN	n1
1	NaN	NaN	NaN	NaN	n2
2	NaN	NaN	NaN	NaN	n3
3	NaN	NaN	NaN	NaN	n4

O primeiro detalhe que você perceba são os valores NaN. É simplesmente o modo Python de representar um “valor ausente”.

Esperávamos concatenar nossos novos valores como uma linha, mas isso não aconteceu. De fato, nosso código não só não concatenou os valores como uma linha, como também criou uma nova coluna totalmente desalinhada em relação ao restante dos dados.

Se paramos para pensar no que está acontecendo nesse caso, poderemos ver que o resultado, na verdade, faz sentido. Em primeiro lugar, se observarmos os novos índices adicionados, percebemos que são muito semelhantes aos resultados que obtivemos quando concatenamos dataframes antes. Os índices do objeto **new_row_series** são análogos aos números das linhas do dataframe.

Além disso, como nossa série não tem uma coluna correspondente, nosso **new_row_series** foi adicionado em uma nova coluna.

Para corrigir esse problema, podemos transformar a nossa série em um dataframe. Esse dataframe contém uma linha de dados, e os nomes das colunas são aqueles com as quais os dados serão associados.

```
#Observe os colchetes duplos
new_row_df = pd.DataFrame([['n1','n2','n3','n4']],columns=['A','B','C','D'])
print(new_row_df)
```

```
A      B      C      D
0  n1  n2  n3  n4
```

```
print(pd.concat([df1,new_row_df]))
```

```
A      B      C      D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
0  n1  n2  n3  n4
```

`concat` é uma função genérica capaz de concatenar vários dados de uma só vez.

Se você tiver de concatenar um único objeto a um dataframe existente, a função `append` poderá cuidar dessa tarefa.

O método `.append()` foi descontinuado nas versões mais recentes do Pandas (acima da 2.0). Atualmente, a forma correta e recomendada de juntar DataFrames é usar o `pd.concat()`.

- Usando um DataFrame:

```
print(df1.append(df2))
```

- Usando um DataFrame com uma só linha:

```
print(df1.append(new_row_df))
```

- Usando um dicionário Python:

```
data_dict = {  
    'A': 'n1',  
    'B': 'n2',  
    'C': 'n3',  
    'D': 'n4'  
}
```

```
print(df1.append(data_dict, ignore_index=True))
```

2.3.1.2 Ignorando o índice

No último exemplo, quando adicionamos um `dict` em um dataframe, tivemos que usar o parâmetro `ignore_index`. Se observarmos com mais atenção, veremos que o índice da linha também foi incrementado em 1, e não houve repetição de um valor de índice anterior.

Se simplesmente queremos concatenar os dados, podemos usar o parâmetro `ignore_index` para reiniciar o índice da linha após a concatenação.

```
row_concat_i = pd.concat([df1,df2,df3],  
                         ignore_index=True)  
print(row_concat_i)
```

	A	B	C	D
0	a0	b0	c0	d0
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5
6	a6	b6	c6	d6
7	a7	b7	c7	d7
8	a8	b8	c8	d8
9	a9	b9	c9	d9
10	a10	b10	c10	d10
11	a11	b11	c11	d11

2.3.2 Adicionando colunas

Concatenando colunas - axis

Concatenar colunas é muito semelhante a concatenar linhas. A principal diferença esta no parâmetro `axis` da função `concat`.

O valor default de `axis` é 0, portanto ele concatenará os dados nas linhas. Entretanto, se passarmos `axis=1` para a função, os dados serão concatenados nas colunas.

```
col_concat = pd.concat([df1,df2,df3],  
                      axis=1)  
print(col_concat)
```

	A	B	C	D	A	B	C	D	A	B	C	D
0	a0	b0	c0	d0	a4	b4	c4	d4	a8	b8	c8	d8
1	a1	b1	c1	d1	a5	b5	c5	d5	a9	b9	c9	d9
2	a2	b2	c2	d2	a6	b6	c6	d6	a10	b10	c10	d10
3	a3	b3	c3	d3	a7	b7	c7	d7	a11	b11	c11	d11

Subconjuntos colunas

Se tentarmos obter um subconjunto de dados com base nos nomes das colunas, teremos um resultado similar aquele obtido se concatenássemos por linha e gerássemos um subconjunto pelo índice da linhas.

```
print(col_concat['A'])
```

	A	A	A
0	a0	a4	a8
1	a1	a5	a9
2	a2	a6	a10
3	a3	a7	a11

Adicionar uma única coluna

Adicionar uma única coluna em um dataframe pode ser feito diretamente, sem usar nenhuma função específica do Pandas.

Basta especificar um novo nome de coluna e o vetor que você quer que seja atribuído a essa nova coluna.

```
col_concat['new_col_list'] = ['n1','n2','n3','n4']
print(col_concat)
```

	A	B	C	D	A	B	C	D	A	B	C	D	new_col_list
0	a0	b0	c0	d0	a4	b4	c4	d4	a8	b8	c8	d8	n1
1	a1	b1	c1	d1	a5	b5	c5	d5	a9	b9	c9	d9	n2
2	a2	b2	c2	d2	a6	b6	c6	d6	a10	b10	c10	d10	n3
3	a3	b3	c3	d3	a7	b7	c7	d7	a11	b11	c11	d11	n4

```
col_concat['new_col_series'] = pd.Series(['n1','n2','n3','n4'])
print(col_concat)
```

	A	B	C	D	A	B	C	D	A	B	C	D	new_col_list \
0	a0	b0	c0	d0	a4	b4	c4	d4	a8	b8	c8	d8	n1
1	a1	b1	c1	d1	a5	b5	c5	d5	a9	b9	c9	d9	n2
2	a2	b2	c2	d2	a6	b6	c6	d6	a10	b10	c10	d10	n3
3	a3	b3	c3	d3	a7	b7	c7	d7	a11	b11	c11	d11	n4

	new_col_series
0	n1
1	n2
2	n3
3	n4

Usar a função `concat` continua funcionando, desde que você lhe passe um dataframe. Essa abordagem exige um pouco mais de código desnecessário.

Reiniciando índices colunas

Por fim, podemos reiniciar os índices das colunas para que não tenhamos nomes duplicados.

```
print(pd.concat([df1,df2,df3], axis=1, ignore_index=True))
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	a0	b0	c0	d0	a4	b4	c4	d4	a8	b8	c8	d8
1	a1	b1	c1	d1	a5	b5	c5	d5	a9	b9	c9	d9
2	a2	b2	c2	d2	a6	b6	c6	d6	a10	b10	c10	d10
3	a3	b3	c3	d3	a7	b7	c7	d7	a11	b11	c11	d11

2.3.3 Concatenação com índices diferentes

Os exemplos apresentados até agora partiram de pressuposto de que estávamos executando uma concatenação simples de linha ou de coluna. Também foi suposto que a(s) nova(s) linha(s) tinha(m) os mesmos nomes de colunas ou que a(s) coluna(s) tinha(m) os mesmos índices de linha.

Esta seção aborda o que acontece quando os índices das linhas e das colunas não estão alinhados.

2.3.3.1 Concatenando linhas com colunas diferentes

Vamos modificar nossos dataframes para os próximos exemplos:

```
df1.columns = ['A', 'B', 'C', 'D']
df2.columns = ['E', 'F', 'G', 'H']
df3.columns = ['A', 'C', 'F', 'H']
```

```
print(df1)
```

```
      A    B    C    D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```
print(df2)
```

```
      E    F    G    H
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
```

```
print(df3)
```

```
      A    C    F    H
0  a8  b8  c8  d8
1  a9  b9  c9  d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

Se tentarmos concatenar esses dataframes como fizemos anteriormente, os dataframes agora serão muito mais do que simplesmente empilhados uns sobre os outros. As colunas se alinharão e NaN preencherá qualquer área que estaja faltando.

```
row_concat = pd.concat([df1,df2,df3])
print(row_concat)
```

	A	B	C	D	E	F	G	H
0	a0	b0	c0	d0	NaN	NaN	NaN	NaN
1	a1	b1	c1	d1	NaN	NaN	NaN	NaN
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN
3	a3	b3	c3	d3	NaN	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	a4	b4	c4	d4
1	NaN	NaN	NaN	NaN	a5	b5	c5	d5
2	NaN	NaN	NaN	NaN	a6	b6	c6	d6
3	NaN	NaN	NaN	NaN	a7	b7	c7	d7
0	a8	NaN	b8	NaN	NaN	c8	NaN	d8
1	a9	NaN	b9	NaN	NaN	c9	NaN	d9
2	a10	NaN	b10	NaN	NaN	c10	NaN	d10
3	a11	NaN	b11	NaN	NaN	c11	NaN	d11

Uma maneira de evitar a inclusão de valores NaN é manter somente as colunas que sejam compartilhadas pela lista de objetos a serem concatenados. Um parâmetro chamado `join` faz isso.

Por padrão, seu valor é `outer`, o que significa que todas as colunas serão mantidas.

Porém, [podemos definir `join='inner'` para manter somente as colunas que sejam compartilhadas entre os conjuntos de dados]{underline}.

Se tentarmos manter apenas as colunas de todos os três dataframes, teremos um dataframe vazio, pois não há nenhuma coluna em comum.

```
print(pd.concat([df1,df2,df3], join='inner'))
```

```
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
```

Se usarmos os dataframes que tenham colunas em comum, somente aquelas que sejam com-
partilhadas por todos serão devolvidas.

```
print(pd.concat([df1,df3],ignore_index=False, join='inner'))
```

	A	C
0	a0	c0
1	a1	c1
2	a2	c2
3	a3	c3
0	a8	b8
1	a9	b9
2	a10	b10
3	a11	b11

2.3.3.2 Concatenando colunas com linhas diferentes

Vamos pegar nossos dataframes e modificá-los novamente de modo que tenham índices de linha diferente. Nesse caso, estamos tomando como base as mesmas modificações de dataframe feitas anteriormente.

```
df1.index = [0,1,2,3]
df2.index = [4,5,6,7]
df3.index = [0,2,5,7]
```

```
print(df1)
```

```
      A    B    C    D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```
print(df2)
```

```
      E    F    G    H
4  a4  b4  c4  d4
5  a5  b5  c5  d5
6  a6  b6  c6  d6
7  a7  b7  c7  d7
```

```
print(df3)
```

```
      A    C    F    H
0  a8  b8  c8  d8
2  a9  b9  c9  d9
5  a10  b10  c10  d10
7  a11  b11  c11  d11
```

Quando concatenamos ao longo de `axis=1`, temos o mesmo resultado de concatenar ao longo de `axis=0`. Os novos dataframes serão somados por coluna, havendo correspondência entre seus respectivos índices de linha. Indicadores de valores ausentes aparecerão nas áreas em que os índices não se alinham.

```
col_concat = pd.concat([df1,df2,df3], axis=1)
print(col_concat)
```

	A	B	C	D	E	F	G	H	A	C	F	H
0	a0	b0	c0	d0	NaN	NaN	NaN	NaN	a8	b8	c8	d8
1	a1	b1	c1	d1	NaN							
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN	a9	b9	c9	d9
3	a3	b3	c3	d3	NaN							
4	NaN	NaN	NaN	NaN	a4	b4	c4	d4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	a5	b5	c5	d5	a10	b10	c10	d10
6	NaN	NaN	NaN	NaN	a6	b6	c6	d6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	a7	b7	c7	d7	a11	b11	c11	d11

Assim como fizemos quando concatenamos por linha, podemos optar por manter o resultado somente quando houver índices correspondentes usando `join='inner'`.

```
print(pd.concat([df1,df3],axis=1,join='inner'))
```

	A	B	C	D	A	C	F	H
0	a0	b0	c0	d0	a8	b8	c8	d8
2	a2	b2	c2	d2	a9	b9	c9	d9

2.4 Combinando vários conjuntos de dados

A seção anterior fez alusão a alguns conceitos de banco de dados. Os parâmetros `join='inner'` e o default `join='outer'` têm origem no modo de trabalhar com banco de dados quando queremos combinar tabelas.

Em vez de simplesmente ter um índice de linha ou de coluna que queremos usar para concatenar valores, às vezes podemos ter dois ou mais dataframes que queremos combinar com base em valores de dados comuns. Essa tarefa é conhecida no mundo dos bancos de dados como a execução de uma “junção” (`join`).

O Pandas tem um comando `pd.join` que usa `pd.merge` internamente. `join` fará a combinação (`merge`) de objetos com base em um índice, mas o comando `merge` é muito mais explícito e flexível. Se você planeja combinar dataframes pelo índice da linha, por exemplo, poderá consultar a função `join`.

Usaremos conjuntos de dados de pesquisa nessa série de exemplos.

```
person = pd.read_csv('./Data/Cap_04/survey_person.csv')
site = pd.read_csv('./Data/Cap_04/survey_site.csv')
survey = pd.read_csv('./Data/Cap_04/survey_survey.csv')
visited = pd.read_csv('./Data/Cap_04/survey_visited.csv')
```

```
print(person)
```

```
      ident   personal    family
0     dyer     William      Dyer
1       pb        Frank  Pabodie
2     lake     Anderson      Lake
3      roe  Valentina  Roerich
4  danforth        Frank  Danforth
```

```
print(site)
```

```
      name     lat    long
0  DR-1 -49.85 -128.57
1  DR-3 -47.15 -126.72
2  MSK-4 -48.87 -123.40
```

```
print(survey)
```

	taken	person	quant	reading
0	619	dyer	rad	9.82
1	619	dyer	sal	0.13
2	622	dyer	rad	7.80
3	622	dyer	sal	0.09
4	734	pb	rad	8.41
5	734	lake	sal	0.05
6	734	pb	temp	-21.50
7	735	pb	rad	7.22
8	735	NaN	sal	0.06
9	735	NaN	temp	-26.00
10	751	pb	rad	4.35
11	751	pb	temp	-18.50
12	751	lake	sal	0.10
13	752	lake	rad	2.19
14	752	lake	sal	0.09
15	752	lake	temp	-16.00
16	752	roe	sal	41.60
17	837	lake	rad	1.46
18	837	lake	sal	0.21
19	837	roe	sal	22.50
20	844	roe	rad	11.25

```
print(visited)
```

	ident	site	dated
0	619	DR-1	1927-02-08
1	622	DR-1	1927-02-10
2	734	DR-3	1939-01-07
3	735	DR-3	1930-01-12
4	751	DR-3	1930-02-26
5	752	DR-3	NaN
6	837	MSK-4	1932-01-14
7	844	DR-1	1932-03-22

No momento, nossos dados estão separados em várias partes, cada uma sendo uma unidade de observação. Se quiséssemos observar as datas de cada local, junto com as informações de latitude e longitude desse local, teríamos de combinar (e fazer um `merge`) de vários dataframes. Isso pode ser feito com a função `merge` do Pandas. `merge` é, na verdade, um método de `dataframe`.

Quando chamamos esse método, o dataframe chamado será referenciado como ‘`left`’. Na função `merge`, o primeiro parâmetro é o dataframe ‘`right`’. O próximo parâmetro indica como o resultado final combinado se parecerá (`how`). A Tabela 1 apresenta mais detalhes.

Em seguida, definimos o parâmetro `on`. Ele especifica com quais colunas será feita a correspondência. Se as colunas a esquerda e à direita não tiverem o mesmo nome, poderemos usar os parâmetros `left_on` e `right_on` em seu lugar.

Tabela 1: Como o parâmetro `how` do **Pandas** se relaciona com o **SQL**

Pandas	SQL	Descrição
<code>left</code>	<code>left outer</code>	Mantém todas as chaves da esquerda .
<code>right</code>	<code>right outer</code>	Mantém todas as chaves da direita .
<code>outer</code>	<code>full outer</code>	Mantém todas as chaves tanto da esquerda quanto da direita .
<code>inner</code>	<code>inner</code>	Mantém somente as chaves que existem tanto na esquerda quanto na direita .

Tabela 2: Comportamento do parâmetro `how` no `merge`

Tipo de Join	Conceito	Resultado Prático
<code>inner</code> (padrão)	Interseção	Retorna apenas as linhas que possuem chaves em ambos os DataFrames.
<code>left</code>	Prioriza a Esquerda	Mantém todos os dados do DataFrame da esquerda e traz o que houver match da direita.
<code>right</code>	Prioriza a Direita	Mantém todos os dados do DataFrame da direita e traz o que houver match da esquerda.
<code>outer</code>	União	Retorna todos os registros de ambos. Onde não houver correspondência, preenche com <code>NaN</code> .

Tabela 3: Parâmetros do `merge`

Parâmetro	Função
<code>left</code>	O primeiro DataFrame (base principal no caso de <code>left join</code>).
<code>right</code>	O segundo DataFrame (o que será “anexado”).
<code>how</code>	O tipo de união: ‘ <code>left</code> ’, ‘ <code>right</code> ’, ‘ <code>inner</code> ’ (padrão) ou ‘ <code>outer</code> ’.
<code>on</code>	Nome da coluna usada como chave (quando o nome é igual em ambos).
<code>left_on</code>	Nome da coluna chave no DataFrame da esquerda .
<code>right_on</code>	Nome da coluna chave no DataFrame da direita .
<code>suffixes</code>	Sufixos para diferenciar colunas com nomes iguais que não são chaves (ex: <code>_x</code> , <code>_y</code>).

2.4.1 merge um a um (*one-to-one*)

No tipo mais simples de `merge`, temos dois dataframes em que queremos fazer a junção de uma coluna com outra, e as colunas que queremos juntar não contém nenhum valor duplicado.

Nesse exemplo, modificamos o dataframe `visited` de modo que não haja valores duplicados de `site`.

```
visited_subset = visited.loc[[0,2,6]]  
print(visited_subset)
```

```
      ident   site       dated  
0     619  DR-1  1927-02-08  
2     734  DR-3  1939-01-07  
6     837  MSK-4 1932-01-14
```

Podemos fazer nosso `merge` um a um (*one-to-one*):

```
#O valor default de 'how' é 'inner',  
# portanto, não precisa ser especificado.  
o2o_merge = site.merge(  
    visited_subset,  
    left_on='name',  
    right_on='site'  
)  
  
print(o2o_merge)
```

```
      name    lat   long  ident   site       dated  
0  DR-1 -49.85 -128.57    619  DR-1  1927-02-08  
1  DR-3 -47.15 -126.72    734  DR-3  1939-01-07  
2  MSK-4 -48.87 -123.40    837  MSK-4 1932-01-14
```

Como pode ver, criamos agora um novo dataframe a partir de dois dataframes diferentes, em que houve uma correspondência de linhas com base em um conjunto particular de colunas. No jargão de SQL, as colunas usadas na correspondência são chamadas de “**chaves**”.

2.4.2 merge de muitos para um (*many-to-one*)

Se optarmos por fazer o mesmo `merge`, mas dessa vez sem usar o dataframe `visited` como subconjunto, fariamos um `merge` de muitos para um (*many-to-one*). Nesse tipo de merge, um dos dataframes tem valores de chave que se repetem.

Os dataframes contendo as observações únicas serão então duplicados no `merge`.

```
m2o_merge = site.merge(  
    visited,  
    left_on='name',  
    right_on='site'  
)  
  
print(m2o_merge)
```

	name	lat	long	ident	site	dated
0	DR-1	-49.85	-128.57	619	DR-1	1927-02-08
1	DR-1	-49.85	-128.57	622	DR-1	1927-02-10
2	DR-1	-49.85	-128.57	844	DR-1	1932-03-22
3	DR-3	-47.15	-126.72	734	DR-3	1939-01-07
4	DR-3	-47.15	-126.72	735	DR-3	1930-01-12
5	DR-3	-47.15	-126.72	751	DR-3	1930-02-26
6	DR-3	-47.15	-126.72	752	DR-3	NaN
7	MSK-4	-48.87	-123.40	837	MSK-4	1932-01-14

Como podemos ver, as informações de `site(name, lat e long)` foram duplicadas e houve correspondência com os dados de `visited`.

2.4.3 merge de muitos para muitos (many-to-many)

Por fim, há ocasiões em que vamos querer efetuar uma correspondência baseada em várias colunas. Como exemplo, suponha que haja dois dataframes, provenientes de um `merge` entre `person` e `survey` e de outro entre `visited` com `survey`.

```
ps = person.merge(
    survey,
    left_on='ident',
    right_on='person'
)

vs = visited.merge(
    survey,
    left_on='ident',
    right_on='taken'
)

print(ps)
```

	ident	personal	family	taken	person	quant	reading
0	dyer	William	Dyer	619	dyer	rad	9.82
1	dyer	William	Dyer	619	dyer	sal	0.13
2	dyer	William	Dyer	622	dyer	rad	7.80
3	dyer	William	Dyer	622	dyer	sal	0.09
4	pb	Frank	Pabodie	734	pb	rad	8.41
5	pb	Frank	Pabodie	734	pb	temp	-21.50
6	pb	Frank	Pabodie	735	pb	rad	7.22
7	pb	Frank	Pabodie	751	pb	rad	4.35
8	pb	Frank	Pabodie	751	pb	temp	-18.50
9	lake	Anderson	Lake	734	lake	sal	0.05
10	lake	Anderson	Lake	751	lake	sal	0.10
11	lake	Anderson	Lake	752	lake	rad	2.19
12	lake	Anderson	Lake	752	lake	sal	0.09
13	lake	Anderson	Lake	752	lake	temp	-16.00
14	lake	Anderson	Lake	837	lake	rad	1.46
15	lake	Anderson	Lake	837	lake	sal	0.21
16	roe	Valentina	Roerich	752	roe	sal	41.60
17	roe	Valentina	Roerich	837	roe	sal	22.50
18	roe	Valentina	Roerich	844	roe	rad	11.25

```
print(vs)
```

	ident	site	dated	taken	person	quant	reading
0	619	DR-1	1927-02-08	619	dyer	rad	9.82
1	619	DR-1	1927-02-08	619	dyer	sal	0.13
2	622	DR-1	1927-02-10	622	dyer	rad	7.80
3	622	DR-1	1927-02-10	622	dyer	sal	0.09
4	734	DR-3	1939-01-07	734	pb	rad	8.41
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	NaN	752	lake	rad	2.19
14	752	DR-3	NaN	752	lake	sal	0.09
15	752	DR-3	NaN	752	lake	temp	-16.00
16	752	DR-3	NaN	752	roe	sal	41.60
17	837	MSK-4	1932-01-14	837	lake	rad	1.46
18	837	MSK-4	1932-01-14	837	lake	sal	0.21
19	837	MSK-4	1932-01-14	837	roe	sal	22.50
20	844	DR-1	1932-03-22	844	roe	rad	11.25

Podemos executar um merge de muitos para muitos (*many-to-many*) passando as várias colunas com as quais a correspondência será feita, usando uma lista Python.

```
ps_vs = ps.merge(  
    vs,  
    left_on=['ident','taken','quant','reading'],  
    right_on=['person','ident','quant','reading'])
```

Vamos observar apenas a primeira linha:

```
print(ps_vs.loc[0,])
```

```
ident_x      dyer  
personal     William  
family       Dyer  
taken_x      619  
person_x     dyer  
quant        rad  
reading      9.82  
ident_y      619  
site         DR-1  
dated        1927-02-08  
taken_y      619  
person_y     dyer  
Name: 0, dtype: object
```

O Pandas acrescentará automaticamente um sufixo no nome de uma coluna se houver colisões no nome. Na saída, _x refere-se aos valores do dataframe à esquerda, enquanto o sufixo _y é proveniente dos valores do dataframe à direita.

2.4.4 Resumo

O `pd.merge()` funciona de forma similar aos JOINs do **SQL**. O que define o tipo de relação é a frequência com que as chaves de ligação aparecem em cada DataFrame.

1. *One-to-One* (Um-para-Um - o2o)

Ocorre quando a chave de ligação é única em ambos os DataFrames. É como unir duas tabelas de informações diferentes sobre os mesmos indivíduos.

- **A lógica:** Cada linha da Tabela A encontra exatamente uma linha correspondente na Tabela B.
- **Impacto nas linhas:** O número de linhas do DataFrame resultante será igual ao número de chaves que coincidem. Nenhuma linha é duplicada; as colunas são apenas “esticadas” para o lado.

Exemplo: Uma tabela com `ID` e `Nome`, e outra com `ID` e `Cargo`.

2. *Many-to-One* (Muitos-para-Um - m2o)

Ocorre quando uma das tabelas possui chaves repetidas, mas a outra possui apenas chaves únicas. O pandas preserva as repetições e propaga os dados da tabela “única”.

- **A lógica:** Uma linha da Tabela B (o lado “One”) corresponde a várias linhas da Tabela A (o lado “Many”).
- **Impacto nas linhas:** As linhas da Tabela A são preservadas. O conteúdo da linha correspondente na Tabela B é copiado para cada uma dessas linhas.
- **Exemplo visual:** Se o “Vendedor 1” aparece em 10 linhas de vendas, os dados dele (nome, região) serão repetidos nessas 10 linhas no resultado final.

Exemplo: Uma tabela de **Vendas** (onde o `ID_Vendedor` se repete) e uma tabela de **Vendedores** (onde cada `ID_Vendedor` é único).

3. *Many-to-Many* (Muitos-para-Muitos - m2m)

Ocorre quando a chave de ligação se repete em ambos os DataFrames. O resultado é um produto cartesiano das linhas correspondentes.

- **A lógica:** Uma chave aparece N vezes na Tabela A e M vezes na Tabela B.
- **Impacto nas linhas:** O Pandas cria todas as combinações possíveis entre elas. O número de linhas resultantes para aquela chave específica será $N \times M$.
- **Atenção:** Este é o caso mais perigoso, pois pode gerar um aumento explosivo no tamanho do seu DataFrame se você não estiver esperando por essas combinações.

Exemplo: Uma tabela de Produtos em promoção e uma tabela de Lojas que vendem esses produtos. Se um produto aparece 3 vezes na Tabela A e 2 vezes na Tabela B, o resultado terá $3 \times 2 = 6$ linhas para essa chave.

2.5 Conclusão preparação dos dados

As vezes, será preciso combinar diversas partes ou dados ou vários conjuntos de dados, conforme a pergunta que tiver tentado responder. Tenha em mente, porém, que os dados de que você precisar nas análises não estarão necessariamente no melhor formato usado para armazená-los.

Os dados de pesquisa usados no último exemplo estavam separados em quatro partes que precisam ser combinadas. Depois que fizemos o `merge` das tabelas, muitas informações redundantes apareceram nas linhas. Do ponto de vista da armazenagem de dados e de sua entrada, cada uma dessas duplicações pode levar a erros e inconsistência nos dados. É isso que Hadley quis dizer quando afirmou que, nos dados organizados (*Tidy data*), “cada tipo de unidade de observação forma uma tabela”.

3 Dados Ausentes

3.1 Introdução

Raramente você receberá um conjunto de dados sem valor ausente. Há muitas representações para dados ausentes. Nos bancos de dados, são valores NULL; certas linguagens de programação usam NA, e, dependendo da origem de seus dados, valores ausentes podem ser uma string vazia, '', ou até mesmo valores numéricos como 88 ou 99. O Pandas exibe valores ausentes como NaN.

Mapa conceitual

1. Conhecimento prévio:
 - a. Importação de bibliotecas;
 - b. Fatiamento e indexação de dados;
 - c. Uso de funções e métodos;
 - d. Uso de parâmetros de funções.

Objetivos

Este capítulo abordará:

1. O que é um valor ausente
2. Como os valores ausentes são criados
3. Como recodificar e fazer cálculos com valores ausentes

3.2 O que é um valor NaN?

O valor NaN no Pandas é proveniente do numpy. Valores ausentes podem ser usados ou exibidos de algumas maneiras em Python - NaN, NAN ou nan -, mas são todos equivalentes.

```
# Basta importar os valores ausentes do numpy
```

```
from numpy import NaN, NAN, nan
```

Valores ausentes são diferentes de outros tipos de dados, pois não são realmente iguais a nada. O dado está ausente, portanto não há um conceito de igualdade. NaN não é equivalente a 0 nem a uma string vazia, ''.

Podemos demonstrar essa ideia em Python fazendo testes de igualdade:

```
print(np.nan == True)
```

```
False
```

```
print(np.nan == False)
```

```
False
```

```
print(np.nan == 0)
```

```
False
```

```
print(np.nan == '')
```

```
False
```

Valores ausentes também não são iguais a outros valores ausentes:

```
print(np.nan == np.nan)
```

```
False
```

O Pandas tem métodos embutidos para testar um valor ausente:

```
print(pd.isnull(np.nan))
```

True

O Pandas também tem métodos para testar valores que não são ausentes:

```
print(pd.notnull(np.nan))
```

False

```
print(pd.notnull(42))
```

True

```
print(pd.notnull('missing'))
```

True

3.3 De onde vêm os valores ausentes?

Podemos obter valores ausentes quando carregamos um conjunto de dados com esses valores, ou do processo de manipulação dos dados (*data munging*).

3.3.1 Carga de dados

Os dados de pesquisa que usamos anteriormente incluíam um conjunto de dados, `visited`, que continham dados ausentes. Quando os valores foram carregados, o Pandas encontrou automaticamente as células com dados ausentes e nos deu um dataframe com o valor `NaN` nas células apropriadas.

Na função `read_csv`, três parâmetros estão relacionados com a leitura de valores ausentes:

- `na_values`;
- `keep_default_na`;
- `na_filter`.

na_values

O parâmetro **na_values** nos permite especificar valores ausentes ou **NaN** adicionais. Você pode passar uma str Python ou um objeto do tipo lista para que seja automaticamente codificados como valores ausentes quando o arquivo é lido.

É claro que valores ausentes default, como **NA**, **NaN**, ou **nan**, já estão disponíveis, e é por isso que esse parâmetro nem sempre é usado.

Alguns dados saudáveis podem codificar **99** como valor ausente; para especificar o uso desse valor, defina **na_values=[99]**.

na_values: Permite que você adicione seus próprios marcadores (ex: '999', 'FALTA', 'INTERROGAÇÃO').

```
# Define o local em que estão os dados
visited_file = 'Data/Cap_04/survey_visited.csv'

# Carregar os dados com valores default
print(pd.read_csv(visited_file))
```

	ident	site	dated
0	619	DR-1	1927-02-08
1	622	DR-1	1927-02-10
2	734	DR-3	1939-01-07
3	735	DR-3	1930-01-12
4	751	DR-3	1930-02-26
5	752	DR-3	NaN
6	837	MSK-4	1932-01-14
7	844	DR-1	1932-03-22

keep_default_na

O parâmetro keep_default_na é um bool que permite especificar se algum valor adicional deve ser considerado como ausente.

Esse valor é True por padrão, ou seja, qualquer valor ausente adicional especificado com o parâmetro na_values será concatenado à lista de valores ausentes. No entanto, keep_default_na também pode ser definido com keep_default_na=False; nesse caso, somente os valores ausentes especificados em na_values serão usados.

keep_default_na=False decide se a lista padrão do Pandas (ex: ‘NaN’, ‘NULL’, ‘N/A’) deve ser usada.

```
# Carrega os dados sem valores ausentes default
print(pd.read_csv(visited_file,keep_default_na=False))
```

```
    ident    site      dated
0     619   DR-1  1927-02-08
1     622   DR-1  1927-02-10
2     734   DR-3  1939-01-07
3     735   DR-3  1930-01-12
4     751   DR-3  1930-02-26
5     752   DR-3
6     837  MSK-4  1932-01-14
7     844   DR-1  1932-03-22
```

```
# Especifica manualmente os valores ausentes
print(pd.read_csv(
    visited_file,
    na_values=[''],
    keep_default_na=False))
```

```
    ident    site      dated
0     619   DR-1  1927-02-08
1     622   DR-1  1927-02-10
2     734   DR-3  1939-01-07
3     735   DR-3  1930-01-12
4     751   DR-3  1930-02-26
5     752   DR-3      NaN
6     837  MSK-4  1932-01-14
7     844   DR-1  1932-03-22
```

`na_filter`

Por fim, `na_filter` é um `bool` que especificará se algum valor será lido como ausente.

O valor default de `na_filter=True` implica que valores ausentes serão codificados como `NaN`. Se atribuirmos `na_filter=False`, então nada será recodificado como ausente. Liga e desliga a busca por dados faltantes, se for `False`, os outros dois parâmetros são ignorados..

Podemos pensar nesse parâmetro como uma forma de desativar todos os parâmetros definidos para `na_values` e `keep_default_na`, mas é mais provável que ele seja usado se quisermos ter um ganho de desempenho carregando dados sem valores ausentes.

Como ele funciona:

- **True** (Padrão): O pandas analisa o conteúdo de todas as colunas em busca de strings que representem dados faltantes e as converte para `NaN`.
- **False**: O pandas ignora essa busca. Tudo o que estiver no arquivo será lido literalmente como string ou número, sem conversão para “ausente”.

Por que usar `na_filter=False`?

- Velocidade: Em arquivos gigantes onde você sabe que não há dados faltantes, desativar o filtro acelera consideravelmente o carregamento.
- Integridade: Se o seu arquivo contém a string “NA” (por exemplo, o código de um país como a Namíbia) e você não quer que o pandas a transforme em um buraco vazio (`NaN`).

3.3.2 Dados combinados

No capítulo Preparação dos dados mostrou como combinar conjunto de dados (`merge`). Alguns dos exemplos naquele capítulo incluíam valores ausentes na saída. Se recriarmos a tabela combinada, veremos valores ausentes na saída resultante.

```
visited = pd.read_csv('Data/Cap_04/survey_visited.csv')
survey = pd.read_csv('Data/Cap_04/survey_survey.csv')
```

```
print(visited)
```

	ident	site	dated
0	619	DR-1	1927-02-08
1	622	DR-1	1927-02-10
2	734	DR-3	1939-01-07
3	735	DR-3	1930-01-12
4	751	DR-3	1930-02-26
5	752	DR-3	NaN
6	837	MSK-4	1932-01-14
7	844	DR-1	1932-03-22

```
print(survey)
```

	taken	person	quant	reading
0	619	dyer	rad	9.82
1	619	dyer	sal	0.13
2	622	dyer	rad	7.80
3	622	dyer	sal	0.09
4	734	pb	rad	8.41
5	734	lake	sal	0.05
6	734	pb	temp	-21.50
7	735	pb	rad	7.22
8	735	NaN	sal	0.06
9	735	NaN	temp	-26.00
10	751	pb	rad	4.35
11	751	pb	temp	-18.50
12	751	lake	sal	0.10
13	752	lake	rad	2.19
14	752	lake	sal	0.09
15	752	lake	temp	-16.00
16	752	roe	sal	41.60

```

17    837    lake   rad     1.46
18    837    lake   sal     0.21
19    837    roe    sal    22.50
20    844    roe    rad    11.25

```

```

# Default how = 'inner'
vs = visited.merge(
    survey,
    left_on='ident',
    right_on='taken'
)
print(vs)

```

	ident	site	dated	taken	person	quant	reading
0	619	DR-1	1927-02-08	619	dyer	rad	9.82
1	619	DR-1	1927-02-08	619	dyer	sal	0.13
2	622	DR-1	1927-02-10	622	dyer	rad	7.80
3	622	DR-1	1927-02-10	622	dyer	sal	0.09
4	734	DR-3	1939-01-07	734	pb	rad	8.41
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	NaN	752	lake	rad	2.19
14	752	DR-3	NaN	752	lake	sal	0.09
15	752	DR-3	NaN	752	lake	temp	-16.00
16	752	DR-3	NaN	752	roe	sal	41.60
17	837	MSK-4	1932-01-14	837	lake	rad	1.46
18	837	MSK-4	1932-01-14	837	lake	sal	0.21
19	837	MSK-4	1932-01-14	837	roe	sal	22.50
20	844	DR-1	1932-03-22	844	roe	rad	11.25

3.3.3 Valores de entrada do usuário

O usuário também pode criar valores ausentes - por exemplo, criar um vetor de valores a partir de um cálculo ou um vetor preparado manualmente.

Criaremos nossos próprios dados com valores ausentes. NaNs são valores válidos tanto quanto para `Series` quanto para `DataFrames`.

```
# Valores ausentes em uma Series  
# Também poderia usar 'None' ao invés de pd.nan  
num_legs = pd.Series({'goat':4, 'amoeba': None})
```

```
print(num_legs)
```

```
goat      4.0  
amoeba    NaN  
dtype: float64
```

```
# Valores ausentes em um dataframe  
scientists = pd.DataFrame({  
    'Name':['Rosaline Franklin','William Gosset'],  
    'Occupation':['Chemist','statistician'],  
    'Born':['1920-07-25','1876-06-13'],  
    'Died':['1958-04-16','1937-10-16'],  
    'missing': [None, np.nan]  
})
```

```
print(scientists)
```

	Name	Occupation	Born	Died	missing
0	Rosaline Franklin	Chemist	1920-07-25	1958-04-16	NaN
1	William Gosset	statistician	1876-06-13	1937-10-16	NaN

Você também pode atribuir diretamente uma coluna de valores ausentes a um dataframe.

```
# Cria um novo dataframe
scientists = pd.DataFrame({
    'Name': ['Rosaline Franklin', 'William Gosset'],
    'Occupation': ['Chemist', 'statistician'],
    'Born': ['1920-07-25', '1876-06-13'],
    'Died': ['1958-04-16', '1937-10-16']
})

# Atribui uma coluna de valores ausentes
scientists['missing']=np.nan

print(scientists)
```

	Name	Occupation	Born	Died	missing
0	Rosaline Franklin	Chemist	1920-07-25	1958-04-16	NaN
1	William Gosset	statistician	1876-06-13	1937-10-16	NaN

3.3.4 Reindexação

Outra maneira de introduzir valores ausentes em seus dados é reindexando o seu dataframe. Isso é conveniente quando queremos adicionar novos indices em nosso dataframe, mas queremos preservar seus valores originais. Um uso comum é quando o índice representa algum intervalo de tempo e queremos adicionar mais datas.

Em termos simples:

- O que acontece: Você define uma nova lista de índices (ex: novas datas).
- A consequência: O Pandas mantém os dados que já existiam e, para os novos índices que não possuem valor correspondente, ele insere automaticamente um NaN (valor ausente).

Se quiséssemos observar somente os anos 2000 a 2010 da plotagem de dados Gapminder, poderíamos executar asmesmas operações agrupadas, obter um subconjunto dos dados e então os reindexar.

```
# Carregando arquivo CSV
gapminder = pd.read_csv('Data/Cap_01/gapminder.tsv', sep='\t')

# Agrupamento de dados
life_exp = gapminder.groupby(['year'])['lifeExp'].mean()

print(life_exp)
```

```
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 reindexar fatiando os dados.

```
# Você pode continuar a encadear o 'loc' do código anterior
# Esse código não funciona mais!
print(life_exp.loc[range(2000,2010),])
```

No Pandas moderno (versão 1.0 ou superior), se você tentar usar o `.loc` com uma lista de rótulos onde alguns não existem no índice, o código gerará um erro de `KeyError`.

Antigamente, o Pandas criava linhas com `NaN` automaticamente, mas isso mudou para evitar erros silenciosos que poderiam causar confusão.

```
# Em vez de .loc, use .reindex
indices_desejados = range(2000, 2010)
print(life_exp.\n    reindex(indices_desejados))
```

```
year
2000      NaN
2001      NaN
2002  65.694923
2003      NaN
2004      NaN
2005      NaN
2006      NaN
2007  67.007423
2008      NaN
2009      NaN
Name: lifeExp, dtype: float64
```

Como alternativa, podemos obter o subconjunto dos dados separadamente e usar o método `reindex`.

```
# Obtém subconjunto
y2000 = life_exp[life_exp.index > 2000]
print(y2000)
```

```
year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
# Reindexa
print(y2000.reindex(range(2000,2010)))
```

```
year
2000      NaN
2001      NaN
2002    65.694923
2003      NaN
2004      NaN
2005      NaN
2006      NaN
2007    67.007423
2008      NaN
2009      NaN
Name: lifeExp, dtype: float64
```

3.4 Trabalhando com dados ausentes

Agora que sabemos como os dados ausentes podem ser criados, vamos ver como eles se comportam quando estamos trabalhando com dados.

3.4.1 Encontrando e contando dados ausentes

```
ebola = pd.read_csv('Data/Cap_04/country_timeseries.csv')
```

Linhas

Uma maneira de ver o número de valores ausentes é contá-los com `count`.

```
print(ebola.count())
```

```
Date          122
Day           122
Cases_Guinea    93
Cases_Liberia     83
Cases_SierraLeone  87
Cases_Nigeria      38
Cases_Senegal       25
Cases_UnitedStates   18
Cases_Spain        16
Cases_Mali         12
Deaths_Guinea      92
Deaths_Liberia      81
Deaths_SierraLeone   87
Deaths_Nigeria       38
Deaths_Senegal        22
Deaths_UnitedStates    18
Deaths_Spain         16
Deaths_Mali          12
dtype: int64
```

Você também pode subtrair o número de linhas não ausentes do número total de linhas.

```
num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print("Número de linhas ausentes:")
print(num_missing)
```

Número de linhas ausentes:

```
Date          0
Day           0
Cases_Guinea  29
Cases_Liberia 39
Cases_SierraLeone 35
Cases_Nigeria 84
Cases_Senegal  97
Cases_UnitedStates 104
Cases_Spain    106
Cases_Mali     110
Deaths_Guinea 30
Deaths_Liberia 41
Deaths_SierraLeone 35
Deaths_Nigeria 84
Deaths_Senegal 100
Deaths_UnitedStates 104
Deaths_Spain   106
Deaths_Mali    110
dtype: int64
```

Colunas

Se quiser contar o número total de valores ausentes em seus dados, ou contar o número de valores ausentes em uma coluna em particular, você poderá usar a função `count_nonzero` do `numpy` em conjunto com o método `isnull`.

A função `np.count_nonzero()` conta quantos valores são verdadeiros (`True`). Como o método `.isnull()` retorna `True` para cada dado ausente, a combinação dos dois resulta no total de nulos.

```
print(np.count_nonzero(ebola.isnull()))
```

1214

```
print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))
```

29

Outra forma de obter os contadores de dados ausentes é usar o método `value_counts` em uma série. Uma tabela de frequência de valores será exibida. Se o parâmetro `dropna` for usado, você também poderá obter um contador de valores ausentes.

O `value_counts` serve para contar quantas vezes cada valor aparece em uma coluna. Imagine uma coluna com as frutas: [Maçã, Maçã, Banana, NaN]. O que acontece na prática:

- Sem o `dropna=False` (Comportamento padrão): O Pandas ignora os dados ausentes. Ele diria que você tem 2 Maçãs e 1 Banana. O NaN fica invisível.
- Com o `dropna=False`: O Pandas passa a tratar o dado ausente como uma categoria comum. O resultado seria:
 - Maçã: 2
 - Banana: 1
 - NaN: 1

```
# Obtém os 5 primeiros contadores da coluna Cases_Guinea
print(ebola.Cases_Guinea.value_counts(dropna=False).head())
```

```
Cases_Guinea
NaN      29
86.0     3
112.0    2
495.0    2
390.0    2
Name: count, dtype: int64
```

```
# Também poderia ser
print(ebola['Cases_Guinea'].value_counts(dropna=False).head())
```

```
Cases_Guinea
NaN      29
86.0     3
112.0    2
495.0    2
390.0    2
Name: count, dtype: int64
```

3.4.2 Limpando dados ausentes

Há muitas maneiras com as quais podemos lidar com dados ausentes. Por exemplo, podemos substituir os dados ausentes por outro valor, preenchê-los usando dados existentes ou descartá-los de nosso conjunto de dados.

Tabela 4: Métodos para limpeza de dados.

Método	Descrição	Exemplo
<code>fillna</code>	Recodificar/substituir valores ausentes.	<code>df.fillna(0)</code>
<code>method='ffill'</code>	Método embutido para fazer preenchimento para a frente (forward).	<code>.fillna(method='ffill')</code>
<code>method='bfill'</code>	Método embutido para fazer preenchimento para trás (backward).	<code>.fillna(method='bfill')</code>
<code>interpolate</code>	A interpolação usa valores existentes para preencher dados ausentes.	<code>.interpolate()</code>
<code>dropna</code>	Descartar observações ou variáveis com esses dados ausentes.	<code>.dropna()</code>

3.4.2.1 Recodificar/substituir

Podemos usar o método `fillna` para recodificar os valores ausentes, transformando-os em outro valor.

Por exemplo, suponha que quiséssemos que os valores ausentes fossem recodificados como 0.

```
print(ebola.fillna(0).iloc[0:10,0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	0.0	10030.0
1	1/4/2015	288	2775.0	0.0	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	0.0	8157.0	0.0
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	0.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	0.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

Ao usar `fillna`, podemos recodificar os valores para um valor específico.

Se consultar a documentação, você descobrirá que `fillna`, como muitas outras funções Pandas, tem um parâmetro `inplace`. Isso significa que os dados subjacentes serão automaticamente alterados para você; não é necessário criar uma nova cópia com as alterações. Você poderá usar esse parâmetro quando seus dados ficarem maiores e quiser que o código seja mais eficiente no que diz respeito à memória.

Em termos simples:

- Comportamento padrão: Quando você usa `df.fillna(0)`, o Pandas cria uma cópia inteira da sua tabela na memória com os novos valores. O original permanece intocado a menos que você o sobrescreva (`df = df.fillna(0)`).
- Com `inplace=True`: O Pandas altera diretamente a tabela original. Ele não cria uma cópia nova.

Sintaxe do `inplace`:

```
df.fillna(0, inplace=True)
```

3.4.2.2 Preenchimento para a frente (*forward*)

Podemos usar métodos embutidos para fazer preenchimentos para a frente (*forward*) ou para trás (*backward*). Quando preenchemos os dados para a frente, o último valor conhecido é usado para o próximo valor ausente. Desse modo, os valores ausentes são substituídos pelo último valor conhecido/registrado.

```
print(ebola.fillna(method='ffill').\n      iloc[0:10,0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	NaN	10030.0
1	1/4/2015	288	2775.0	NaN	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	2769.0	8157.0	9722.0
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	8018.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7977.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

Se a coluna começar com um valor ausente, esse dado permanecerá ausente, pois não haverá um valor anterior com o qual fazer o preenchimento.

Lógica: O valor da linha n é copiado para a linha $n + 1$ se esta for NaN.

3.4.2.3 Preenchimento para trás (*backward*)

Também podemos fazer com que o Pandas preencha dados para trás. Quando preenchemos dados para trás, o valor mais recente é usado para substituir os dados ausentes. Desse modo, os valores ausentes são substituídos pelo valor mais recente.

```
print(ebola.fillna(method='bfill')\
      .iloc[:,0:5]\n      .tail())
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
117	3/27/2014	5	103.0	8.0	6.0
118	3/26/2014	4	86.0	NaN	NaN
119	3/25/2014	3	86.0	NaN	NaN
120	3/24/2014	2	86.0	NaN	NaN
121	3/22/2014	0	49.0	NaN	NaN

Se uma coluna terminar com um valor ausente, esse dado permanecerá ausente, pois não haverá nenhum valor novo com o qual fazer o preenchimento.

Lógica: O valor da linha n é preenchido pelo valor da linha $n + 1$.

3.4.2.4 Interpolação

A interpolação usa valores existentes para preencher valores ausentes. Embora haja muitas maneiras de preencher valores ausentes, a interpolação no Pandas preenche os valores ausentes de modo linear. Especificamente, os valores ausentes são tratados como se devessem estar igualmente espaçados.

```
print(ebola.interpolate()\
      .iloc[0:10,0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	NaN	10030.0
1	1/4/2015	288	2775.0	NaN	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	2749.5	8157.0	9677.5
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	7997.5	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7919.5	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

O método `interpolate` tem um parâmetro `method` que pode alterar o método de interpolação.

Principais Parâmetros:

- `method`: Define a técnica matemática.
 - ‘linear’: (Padrão) Ignora o índice e trata os valores como igualmente espaçados.
 - ‘time’: Funciona com séries temporais, considerando o intervalo real de tempo entre as datas.
 - ‘polynomial’ ou ‘spline’: Requer que você passe uma ordem (ex: ordem 2 ou 3) para curvas mais suaves.
- `limit_direction`: Controla a direção do preenchimento.
 - ‘forward’;
 - ‘backward’;
 - ‘both’.
- `inplace`: Se `True`, altera o DataFrame original.

3.4.2.5 Descartando valores ausentes

O último modo de trabalhar com dados ausentes é descartar observações ou variáveis com esses dados.

Podemos usar o método `dropna` para descartar dados ausentes e especificar parâmetros para que esse método que controlem como os dados são descartados.

Por exemplo, o parâmetro `how` permite especificar se uma linha (ou coluna) será descartada quando algum dado ('`any`') ou todos ('`all`') estiverem ausentes.

O parâmetro `thresh` permite especificar quantos valores diferentes de `NaN` devemos ter para descartar a linha ou a coluna.

```
print(ebola.shape)
```

(122, 18)

Se somente os casos de dados completos forem mantidos em nosso conjunto de dados Ebola, ficaremos com apenas uma linha de dados.

```
ebola_dropna = ebola.dropna()  
print(ebola_dropna.shape)
```

(1, 18)

```
print(ebola_dropna)
```

```
          Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \  
19 11/18/2014  241        2047.0        7082.0        6190.0  
  
      Cases_Nigeria  Cases_Senegal  Cases_UnitedStates  Cases_Spain  Cases_Mali  \  
19            20.0           1.0            4.0           1.0           6.0  
  
      Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone  Deaths_Nigeria  \  
19         1214.0        2963.0        1267.0           8.0  
  
      Deaths_Senegal  Deaths_UnitedStates  Deaths_Spain  Deaths_Mali  
19            0.0            1.0            0.0           6.0
```

Principais parâmetros:

- **axis**:

Define se você remove a linha (0 ou ‘index’) ou a coluna (1 ou ‘columns’). O padrão é remover a linha.

- **how**:

- ‘any’: Remove se houver pelo menos um valor nulo (**padrão**).
- ‘all’: Só remove se todos os valores da linha/coluna forem nulos.

- **subset**:

Permite focar em colunas específicas.

Ex: `df.dropna(subset=['idade'])` só removerá a linha se o nulo estiver na coluna “idade”.

- **inplace=True**:

Altera o DataFrame original diretamente, sem precisar atribuir a uma nova variável.

Cuidado:

Dependendo da quantidade de dados que estiver faltando, manter somente os casos com dados completos pode deixar você com um conjunto de dados inútil. Talvez os dados ausentes não sejam aleatórios, de modo que descartar valores ausentes fará com que você fique com um conjunto de dados tendenciosos ou, quem sabe, manter somente os dados completos o deixará com dados insuficientes para efetuar a sua análise.

3.4.3 Cálculos com dados ausentes

Suponha que quiséssemos ver os contadores de casos para várias regiões. Podemos somar vários regiões a fim de obter uma nova coluna que armazene os contadores de casos.

```
ebola['Cases_multiple'] = ebola['Cases_Guinea']+\
    ebola['Cases_Liberia']+\
    ebola['Cases_SierraLeone']
```

Vamos observer as dez primeiras linhas de cálculo.

```
ebola_subset = ebola.loc[:,\
    ['Cases_Guinea','Cases_Liberia',\
     'Cases_SierraLeone','Cases_multiple']]
print(ebola_subset.head(n=10))
```

	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	Cases_multiple
0	2776.0	NaN	10030.0	NaN
1	2775.0	NaN	9780.0	NaN
2	2769.0	8166.0	9722.0	20657.0
3	NaN	8157.0	NaN	NaN
4	2730.0	8115.0	9633.0	20478.0
5	2706.0	8018.0	9446.0	20170.0
6	2695.0	NaN	9409.0	NaN
7	2630.0	7977.0	9203.0	19810.0
8	2597.0	NaN	9004.0	NaN
9	2571.0	7862.0	8939.0	19372.0

Podemos ver que um valor para `Cases_multiple` foi calculado somente quando não havia valores ausentes para `Cases_Guinea`, `Cases_Liberia` e `Cases_SierraLeone`. Cálculos com valor ausente em geral devolverão um valor ausente, a menos que a função ou método chamado tenha um modo de ignorar os valores ausentes em seus cálculos.

Exemplos de métodos embutidos capazes de ignorar valores ausentes incluem `mean` e `sum`. Essas funções geralmente têm um parâmetro `skipna` permitindo que um valor ainda seja calculado, ignorando os valores ausentes.

```
# Ignorar valores ausentes é True por padrão
print(ebola.Cases_Guinea.sum(skipna=True))
```

84729.0

```
print(ebola.Cases_Guinea.sum(skipna=False))
```

nan

3.5 Conclusão dados ausentes

É raro ter um conjunto de dados sem qualquer valor ausente. É importante saber como trabalhar com valores ausentes porque, mesmo quando estiver trabalhando com dados completos, valores ausentes ainda poderão surgir durante a sua própria manipulação de dados.

Neste capítulo, analisamos alguns dos métodos básicos usados no processo de análise de dados pertinentes à validade dos dados. Ao observar seus dados e tabular os valores ausentes, podemos dar início ao processo de avaliar se os dados têm uma qualidade suficientemente boa para tomada de decisões e inferências.

4 *Tidy data* (dados organizados)

4.1 Introdução

Conforme mencionamos anteriormente, Hadley Wickham, um dos membros mais proeminentes da comunidade R, introduziu o conceito de *tidy data* (dados organizados) em um artigo do *Journal of Statistical Software*.

Tidy data é um framework para estruturar conjuntos de dados de modo que sejam facilmente analisados e visualizados. Podemos pensar nele como um objetivo a que devemos visar quando limpamos os dados.

Depois que você comprehende o que é *tidy data*, esse conhecimento fará com que a análise, a visualização e a coleta de dados sejam muito mais fáceis.

Então o que é *tidy data*? O artigo de Hadley Wickham o define como um conceito que atende aos seguintes critérios:

- Cada linha é uma observação. (*observation*)
- Cada coluna é uma variável. (*variable*)
- Cada tipo de unidade de observação forma uma tabela.

Este capítulo descreve as diversas maneiras de organizar os dados conforme identificadas pelo artigo de Wickham.

Mapa conceitual

Conhecimento prévio:

- a. Chamadas de função e de métodos;
- b. Obtenção de subconjuntos de dados;
- c. Laços;
- d. `list comprehension` (abrangência de listas).

Neste capítulo:

- Refatoração de dados:
 - a. `unpivot/melt/gather` (“despivotear”);
 - b. `pivot/cast/spread`;
 - c. Obtenção de subconjuntos;
 - d. Combinação:
 1. Uso da biblioteca `glob`
 2. Concatenação

Objetivos

Este capítulo abordará:

1. Operação de *unpivoting/melting/gathering* de colunas em linhas (“despivotar”);
2. Operação de *pivoting/casting/spreading* de linhas em colunas (“pivotar”);
3. Normalização de dados separando um dataframe em várias tabelas;
4. Reunião de dados de várias partes.

4.2 Coluna contêm valores, e não variáveis

4.3 Colunas contendo diversas variáveis

4.4 Variáveis tanto em linhas quanto em colunas

4.5 Várias unidades de observação em uma tabela (normalização)

4.6 Unidade de observação em várias tabelas

4.7 Conclusão *Tidy data*