CAPÍTULO 2

A ESTRUTURA DE DADOS SERIES

Neste capítulo, iniciamos a nossa jornada pelo mundo da programação pandas apresentando a ED **Series**, estrutura básica e fundamental do pacote. Se você pretende utilizar a pandas de forma profissional, é muito importante que você aprenda a trabalhar muito bem com esta ED, compreendendo as suas propriedades e dominando os tipos de operações que podem ser realizadas sobre dados que estejam nela estruturados. O capítulo aborda os seguintes temas:

- Criação e propriedades básicas
- Técnicas para consulta e modificação de dados
- Computação vetorizada
- Índices datetime
- Indexação hierárquica

2.1 COMO CRIAR SERIES?

Series é uma ED composta por um vetor de dados e um vetor associado de rótulos, este último chamado de índice (*index*). Ou seja: **Series = vetor de dados + vetor de rótulos**.

A figura 2.1 mostra dois exemplos de Series. Considere que ambas armazenam dados provenientes de um sistema acadêmico utilizado por uma instituição de ensino hipotética. A primeira Series chama-se `notas`, cujos rótulos são inteiros e os dados valores reais (notas médias dos alunos de um dado curso). A segunda chama-se `alunos`, cujos rótulos são strings (matrículas dos alunos do curso) e os dados também strings (nomes).

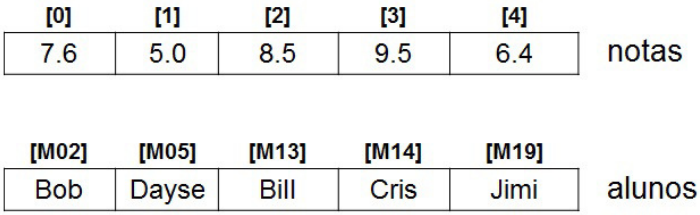

Figura 2.1: Dois exemplos de Series contendo dados de alunos.

O programa a seguir apresenta o código para criar ambas as Series utilizando o método construtor padrão. Tanto neste exemplo, como nos demais a serem apresentados ao longo do livro, mostra-se primeiro o código do programa e imediatamente depois, o resultado de sua execução. Todos os exemplos foram elaborados para funcionar de forma independente, então nenhum programa depende da execução de algum outro que tenha sido apresentado previamente. Desta forma, para testar o código de qualquer programa exemplo em seu computador, basta que você o digite e execute no ambiente Python de sua preferência.

```
#P01: Hello Series!
import pandas as pd

#cria a Series notas
notas = pd.Series([7.6, 5.0, 8.5, 9.5, 6.4])

#cria a Series alunos
```

```
lst_matriculas = ['M02','M05','M13','M14','M19']
lst_nomes = ['Bob','Dayse','Bill','Cris','Jimi']
alunos = pd.Series(lst_nomes,index=lst_matriculas)

#imprime as duas Series
print(notas); print("---------------"); print(alunos)
```

Veja a saída do programa:

```
0    7.6
1    5.0
2    8.5
3    9.5
4    6.4
dtype: float64
---------------
M02      Bob
M05    Dayse
M13     Bill
M14     Cris
M19     Jimi
dtype: object
```

Observe que, para criar a Series `notas`, foi necessário apenas definir uma **lista** com a relação de notas finais. Uma vez que não especificamos um índice para esta Series, a pandas utilizará inteiros de 0 a N-1 (onde N é o tamanho da lista, neste exemplo, N=5). Por outro lado, ao criar `alunos`, além da lista de valores (nomes dos alunos), foi também especificada uma lista de índices (matrículas dos alunos), passada para o método construtor com o uso do parâmetro `index`.

Existem outras formas para criar Series. Um exemplo é apresentado no trecho de código seguinte, onde a Series é criada a partir de um **dicionário**. Neste caso, as chaves do dicionário são automaticamente transformadas em rótulos.

```
dic_alunos = {'M02':'Bob','M05':'Dayse','M13':'Bill',
              'M14':'Cris','M19':'Jimi'}
```

```
alunos = pd.Series(dic_alunos)
```

Listas e dicionários são EDs nativas da linguagem Python, ou seja, elas fazem parte da linguagem padrão (ao contrário da Series, que é específica do pacote pandas). Se você tem pouca experiência em Python e não tem intimidade com essas estruturas, não se preocupe, pois aqui vai uma breve explicação sobre ambas.

Uma lista é uma **sequência ordenada de elementos**, cada qual associado a um número responsável por indicar a sua posição (índice). O primeiro índice de uma lista é sempre 0, o segundo, 1, e assim por diante. Para criar uma lista, basta especificar uma sequência de valores entre colchetes `[ ]`, onde os valores devem estar separados por vírgula. Por exemplo, `['BR','FR','US']` é a lista com as siglas dos países Brasil (posição 0), França (posição 1) e Estados Unidos (posição 2).

Por sua vez, um dicionário é uma ED em que os **elementos são pares chave:valor**. A chave (*key*) identifica um item e o valor armazena seu conteúdo. Qualquer valor armazenado pode ser recuperado de forma extremamente rápida através de sua chave. Para criar um dicionário, você deve utilizar chaves `{ }` e, dentro delas, especificar uma relação de elementos do tipo par *chave:valor*, separados por vírgula. Por exemplo, `{'BR':'Real', 'FR': 'Euro', 'US':'Dólar'}` é um dicionário onde a chave é a sigla de um país e o valor, o nome de sua moeda.

Existem duas diferenças fundamentais entre os dicionários e as listas. A primeira é que, em uma lista, os índices que determinam a posição dos elementos precisam ser inteiros, enquanto em um dicionário os índices podem ser não apenas inteiros, mas também de qualquer tipo básico, como strings. A segunda diferença

encontra-se no fato de que em um dicionário não existe o conceito de ordem, ou seja, ele é uma coleção não ordenada de pares chave:valor.

A partir dessas definições, podemos concluir que a Series une "o melhor dos dois mundos". Isso porque, assim como uma lista, a Series armazena uma sequência ordenada de elementos; no entanto, ao mesmo tempo, permite que cada elemento também seja acessado de forma simples e rápida através de um rótulo (similar ao acesso por *keys* inerente aos dicionários).

Quais as propriedades elementares das Series?

Você deve ter notado que, além de ter mostrado os índices e valores de cada Series, o comando `print()` do programa anterior também exibiu o `dtype` delas. Trata-se de uma das propriedades básicas das Series, que corresponde ao tipo dos elementos do vetor de dados. O `dtype` de `notas` é `float64`, que é utilizado para armazenar números reais com dupla precisão (64 bits). Por sua vez, o `dtype` de `alunos` é `object`, utilizado pela pandas para armazenar dados alfanuméricos (strings). Vale a pena deixar claro que o vetor de dados de uma Series sempre conterá valores do mesmo tipo, ou seja, com o mesmo `dtype`.

A tabela a seguir apresenta os cinco principais `dtypes` da pandas. Na primeira coluna mostramos o nome do `dtype`, na segunda, o tipo de dado que ele armazena e na terceira, o nome do tipo equivalente no Python padrão.

dtype	Utilização	Tipo Python
int64	números inteiros	int
float64	números reais	float

bool	True/False	bool
object	texto	str
datetime64	data/hora	datetime

É necessário ressaltar que `dtype` não é a mesma coisa que tipo do objeto (*type*)! O tipo de qualquer objeto Python representa a classe deste objeto e pode ser sempre obtido com o uso da função `type()`. Se você aplicar a função `type()` sobre qualquer Series, sempre receberá como resposta o tipo `pandas.core.series.Series`. Outra consideração importante é que toda Series está associada a outras propriedades elementares além do `dtype`. São elas:

- `values`: vetor de dados;
- `index`: vetor de rótulos;
- `name`: nome do vetor de dados;
- `size`: tamanho da Series (número de elementos);
- `index.name`: nome do vetor de rótulos;
- `index.dtype`: `dtype` do vetor de rótulos.

No programa seguinte, apresentamos a forma para trabalhar com essas propriedades.

```
#P02: Propriedades básicas das Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#atribui nomes p/ os vetores de dados e rótulos
alunos.name = "alunos"
alunos.index.name = "matrículas"

#recupera e imprime as propriedades
```

```
print(alunos)
print('-----------')
tamanho = alunos.size
dados   = alunos.values
rotulos = alunos.index
alunos_tipo = type(alunos)
alunos_dtype = alunos.dtype
alunos_idx_dtype = alunos.index.dtype

print('número de elementos: ', tamanho)
print('vetor de dados: ', dados)
print('vetor de rótulos: ', rotulos)
print('tipo (type): ', alunos_tipo)
print('dtype da Series:', alunos_dtype)
print('dtype do vetor de rótulos:', alunos_idx_dtype)
```

Saída do programa:

```
matrículas
M02      Bob
M05    Dayse
M13     Bill
M14     Cris
M19     Jimi
Name: alunos, dtype: object
-----------
número de elementos:  5
vetor de dados:  ['Bob' 'Dayse' 'Bill' 'Cris' 'Jimi']
vetor de rótulos:  Index(['M02', 'M05', 'M13', 'M14', 'M19'], dty
pe='object', name='matrículas')
tipo (type):  <class 'pandas.core.series.Series'>
dtype da Series: object
dtype do vetor de rótulos: object
```

Inicialmente, o programa mostra como nomear tanto o vetor de dados como o de rótulos da Series `alunos`, utilizando as propriedades `name` e `index.name`, respectivamente. Veja que quando mandamos imprimir a Series estes nomes passam a ser exibidos. Em seguida, mostramos como recuperar e imprimir as demais propriedades.

2.2 TÉCNICAS PARA CONSULTA E MODIFICAÇÃO DE DADOS

No início do capítulo, vimos que a Series é uma ED que mistura características das listas e dos dicionários. Sendo assim, não é surpreendente que a forma para consultar e modificar dados de Series seja bem parecida com a utilizada por estas EDs. As próximas subseções abordam este tema.

Indexação

Utilizamos colchetes `[ ]` para indexar (acessar e obter) elementos de uma Series. A pandas permite o emprego de três diferentes técnicas: indexação tradicional, fatiamento e indexação booleana. Nos parágrafos a seguir, essas técnicas são explicadas através de exemplos baseados nas Series mostradas na figura 2.1.

Vamos começar pela **indexação tradicional**, a mais simples de todas. Esta técnica de indexação deve ser utilizada quando você quiser recuperar apenas **um elemento** da Series. Para fazer a indexação tradicional, você deve especificar um número inteiro que corresponde ao índice do elemento que você deseja acessar. Se você especificar um número negativo, a pandas fará a indexação de trás para frente, isto é, -1 recupera o último elemento, -2, o penúltimo etc. Caso a sua Series possua um vetor de rótulos associado (como é o caso de `alunos`), você também poderá indexá-la pelos rótulos (como 'M13', 'M02' etc.). Na tabela a seguir, são apresentados diversos exemplos:

Exemplo	Resultado	Explicação
`alunos[0]`	Bob	primeiro aluno

`alunos[1]`	Dayse	segundo aluno
`alunos['M14']`	Cris	aluno de matrícula 'M14'
`alunos[alunos.size-1]`	Jimi	último aluno
`alunos[-1]`	Jimi	outra forma de pegar o último aluno

A técnica de indexação baseada em **fatiamento** (*slicing*) deve ser utilizada quando você quiser recuperar **mais de um elemento** da Series. Você pode fatiar de duas diferentes formas:

- Por intervalos (*ranges*) definidos por dois pontos `:`;
- Por listas.

A tabela seguinte apresenta a sintaxe da operação de fatiamento, onde as cinco primeiras linhas referem-se ao fatiamento por intervalos e a última, ao fatiamento com listas. Na notação utilizada, considere que `s` é o nome de uma Series.

Sintaxe	Explicação
`s[i:j]`	do elemento de índice `i` ao de índice `j-1`
`s[i:]`	do elemento de índice `i` até o último
`s[:j]`	do primeiro elemento ao de índice `j-1`
`s[-k:]`	últimos `k` elementos
`s[i:j:k]`	do elemento `i` ao `j-1`, utilizando o passo `k`
`s[[lista]]`	todos os elementos especificados na lista

Veja alguns exemplos:

Exemplo	Resultado
`alunos[0:2]`	{M02:Bob, M05:Dayse}
`alunos[2:4]`	{M13:Bill, M14:Cris}

`alunos[:2]`	{M02:Bob, M05:Dayse}
`alunos[2:]`	{M13:Bill, M14:Cris, M19:Jimi}
`alunos[-2:]`	{M14:Cris, M19:Jimi}
`alunos[1:5:2]`	{M05:Dayse, M14:Cris}
`alunos[[2,0,4]]`	{M13:Bill, M02:Bob, M19:Jimi}
`alunos[['M13','M02','M19']]`	{M13:Bill, M02:Bob, M19:Jimi}

Caso você esteja em dúvida com relação aos três últimos exemplos (que são realmente menos intuitivos!), aqui vai uma explicação detalhada. Quando fazemos `alunos[1:5:2]`, estamos pedindo o seguinte para a pandas: a partir do elemento de índice `i=1` até o elemento de índice anterior a `j=5` (ou seja, elemento de índice 4), recupere todos os elementos pulando os índices de 2 em 2 (`k=2`). Por este motivo, o resultado inclui `M05:Dayse` (índice 1) e também `M14:Cris` (índice 3).

Já no exemplo `alunos[[2,0,4]]` (fatiamento com lista), estamos pedindo para a pandas recuperar os elementos de índice 2, 0 e 4, **nesta ordem**. De maneira análoga, `alunos[['M13','M02','M19']]` (outro exemplo de fatiamento através de lista) faz com que sejam recuperados os alunos de rótulo 'M13', 'M02' e 'M19', nesta ordem. Veja que a lista especificada deve possuir colchetes, isto é, o certo é usar a notação `alunos[[2,0,4]]` e não `alunos[2,0,4]`.

Uma importante diferença entre as operações de indexação e fatiamento diz respeito ao tipo do resultado retornado por cada uma das operações:

- A indexação tradicional sempre retorna um único elemento, cujo tipo será o tipo básico Python

correspondente ao `dtype` do vetor de dados. Por exemplo, `alunos[0]` retorna uma string (tipo str) e `notas[0]`, um float.

- A operação de fatiamento sempre retorna uma Series, ou seja, um objeto do tipo `pandas.core.series.Series`.

Para encerrar a subseção, vamos falar sobre a **indexação booleana**, a terceira e última técnica para indexar Series. No presente momento, realizaremos apenas uma introdução ao assunto, que será revisitado e mais bem detalhado em capítulos posteriores. Neste modo de indexação, subconjuntos de dados são selecionados com base nos **valores da Series** (valores do vetor de dados) e não em seus rótulos/índices.

No próximo programa, apresentamos um exemplo em que a indexação booleana é aplicada para determinar os nomes de todos os alunos com nota igual ou superior a 7.0. Explicações detalhadas são apresentadas após o código.

```
#P03: Indexação booleana
import pandas as pd

#cria as Series "notas" e "alunos"
notas = pd.Series([7.6, 5.0, 8.5, 9.5, 6.4])
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#obtém os índices dos alunos aprovados
idx_aprovados = notas[notas >= 7].index

#imprime os alunos aprovados
print('relação de alunos aprovados:')
print('---------------------------')
print(alunos[idx_aprovados])
```

Saída:

```
relação de alunos aprovados:
--------------------------
M02    Bob
M13    Bill
M14    Cris
dtype: object
```

Como esse programa funciona? O maior segredo está no comando `idx_aprovados = notas[notas >= 7].index` . Ele é responsável por gerar um vetor de índices (objeto do tipo `pandas.core.indexes.numeric.Int64Index`) que armazenará os índices de todos os elementos com valor igual ou superior a 7.0 na Series `notas` . Sendo assim, o comando retorna o vetor `[0, 2, 3]` . Ao utilizarmos esse vetor em `alunos[idx_aprovados]` , executa-se a operação de fatiamento de `alunos` , recuperando os seus elementos 0, 2 e 3 (Bob, Bill e Cris, os alunos que tiraram mais de 7.0).

Busca

O próximo programa mostra como verificar se determinados rótulos ou valores estão presentes em uma Series.

```
#P04: Busca em Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#testa se rótulos fazem parte de uma Series
tem_M13 = 'M13' in alunos
tem_M99 = 'M99' in alunos
print("existe o rótulo 'M13'? -> ",tem_M13)
print("existe o rótulo 'M99'? -> ",tem_M99)
print('--------------------------')

#testa se valor faz parte de uma Series
```

```
tem_Bob = alunos.isin(['Bob'])
print("existe o valor 'Bob'")
print(tem_Bob)
```

Resultado:

```
existe o rótulo 'M13'? ->  True
existe o rótulo 'M99'? ->  False
---------------------------
existe o valor 'Bob'
M02     True
M05    False
M13    False
M14    False
M19    False
dtype: bool
```

Para testar se um rótulo existe em uma Series (mais precisamente, se faz parte de seu vetor de rótulos), a coisa é bem simples: basta utilizar o operador `in` do Python padrão (que retornará `True` ou `False`).

Se, por outro lado, o que você deseja é verificar se um ou mais valores estão em uma Series (mais precisamente, se fazem parte de seu vetor de dados), precisará utilizar o método `isin()` da pandas. Este método recebe como entrada uma lista contendo um ou mais valores (em nosso exemplo, utilizamos uma lista contendo apenas um valor, 'Bob'). Como saída, é gerada uma Series com `dtype bool` que terá o valor `True` para todos os rótulos associados a valores da lista. Veja que em nosso exemplo, apenas 'M2' recebeu `True` , pois este é o rótulo de `alunos` que possui o valor 'Bob'.

Modificação

No próximo exemplo, apresentamos a forma básica para

inserir, modificar e excluir elementos.

```
#P05: Inserindo, Alterando e Removendo elementos de Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

print('Series original:')
print(alunos)

#insere o aluno de matrícula M55, Rakesh
alunos['M55'] = 'Rakesh'

#altera os nomes Bill, Cris e Jimi para Billy, Cristy e Jimmy
alunos['M13'] = 'Billy'
alunos[['M14','M19']] = ['Cristy','Jimmy']

#remove o aluno de matrícula M02 (Bob)
alunos = alunos.drop('M02')

print('---------------------------')
print('Series após as alterações:')
print(alunos)
```

Saída:

```
Series original:
M02      Bob
M05    Dayse
M13     Bill
M14     Cris
M19     Jimi
dtype: object
---------------------------
Series após as alterações:
M05     Dayse
M13     Billy
M14    Cristy
M19     Jimmy
M55    Rakesh
dtype: object
```

Nesse exemplo, primeiro criamos `alunos` com os mesmos dados dos exemplos anteriores. Em seguida, inserimos um novo aluno de matrícula `M55`, denominado `Rakesh`. Para tal, basta utilizar um comando de atribuição comum: `alunos['M55'] = 'Rakesh'`. A modificação também é feita de forma parecida. Podemos modificar um elemento por vez (`alunos['M13'] = 'Billy'`) ou mais de um, neste caso, usando uma lista de rótulos e uma de valores, como foi feito em `alunos[['M14','M19']] = ['Cristy','Jimmy']`. Por fim, mostramos como remover elementos com o uso do método `drop()` da pandas: `alunos = alunos.drop('M02')`. Vale ressaltar que não apenas a modificação, mas também a inclusão e remoção suportam o uso de listas para que seja possível inserir ou remover muitos elementos de uma vez.

MODIFICAÇÃO DE ÍNDICES

Além de alterar valores, podemos alterar os índices de uma Series, utilizando a propriedade `index`. Por exemplo, se o comando adiante for executado, o primeiro rótulo de alunos será alterado para 'M91', o segundo, para 'M92', e assim sucessivamente.

```
alunos.index = ['M91','M92','M93','M94','M95']
```

2.3 COMPUTAÇÃO VETORIZADA

É possível utilizar a tradicional estrutura `for ... in` para iterar sobre uma Series (ou seja, para percorrer "de cabo a rabo" o vetor

de dados ou o de rótulos):

```
#P06: Iteração
import pandas as pd
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#itera sobre os dados (nomes dos alunos)
for aluno in alunos: print(aluno)

#itera sobre os índices (matrículas)
for indice in alunos.index: print(indice)
```

Entretanto, em grande parte das situações práticas, não precisaremos fazer uso desse recurso. Isso porque, em geral, as operações da pandas podem ser executadas através do mecanismo conhecido como **computação vetorizada** (*vectorization*). Neste processo, as operações são realizadas sobre cada elemento da Series automaticamente, sem a necessidade de programar um laço com o comando `for`. Alguns exemplos:

- Se `s` é uma Series com valores numéricos, e fazemos `s * 2`, obteremos como resultado uma Series que conterá todos os elementos de `s` multiplicados por 2.
- Ao efetuarmos uma soma de duas Series `s1` e `s2`, teremos como resultado uma nova Series em que o valor de um rótulo (ou índice) `i` será igual a `s1[i] + s2[i]`. E da mesma forma ocorrerá para todos os demais rótulos.

O programa a seguir demonstra estes conceitos na prática. Uma outra novidade introduzida no exemplo é a importação da biblioteca **NumPy** e a utilização de um de seus métodos, denominado `sqrt()` (explicações detalhadas são apresentadas após o exemplo).

```
#P07: Operações aritméticas com computação vetorizada
import pandas as pd
import numpy as  np

#cria as Series s1 e s2
s1 = pd.Series([2,4,6])
s2 = pd.Series([1,3,5])
print('s1:'); print(s1)
print('s2:'); print(s2)

#efetua as operações aritméticas
print('---------------------------')
print('s1 * 2')
print(s1 * 2)
print('---------------------------')
print('s1 + s2')
print(s1 + s2)
print('---------------------------')
print('raiz quadrada dos elementos de s1')
print(np.sqrt(s1))   #com a Numpy!
```

Observe com calma os resultados gerados:

```
s1:
0    2
1    4
2    6
dtype: int64
s2:
0    1
1    3
2    5
dtype: int64
---------------------------
s1 * 2
0     4
1     8
2    12
dtype: int64
---------------------------
s1 + s2
0     3
1     7
2    11
```

```
dtype: int64
--------------------------
raiz quadrada dos elementos de s1
0    1.414214
1    2.000000
2    2.449490
dtype: float64
```

Neste programa, primeiro realizamos a importação das bibliotecas pandas (como sempre, apelidada de `pd`) e da biblioteca `numpy` (que recebe o apelido de `np`). Logo depois declaramos duas Series `s1` e `s2`. Então, três operações são efetuadas:

- Multiplicação de todos os elementos de `s1` por 2, obtida por `s1 * 2`;
- Soma de `s1` e `s2`, obtida por `s1 + s2`. Veja que a soma é feita entre pares de elementos de `s1` e `s2` que ocupam a mesma posição (compartilham o mesmo índice);
- Raiz quadrada dos elementos de `s1`, obtida por `np.sqrt(s1)`.

Conforme mencionado anteriormente, o cálculo da raiz quadrada foi feito com o uso de um método da NumPy, uma biblioteca direcionada para a computação de alto desempenho sobre arrays (vetores e matrizes). A NumPy é considerada a "pedra fundamental" da computação científica em Python pelo fato de as suas propriedades e métodos terem sido utilizados como base para o desenvolvimento de diversas outras bibliotecas importantes para ciência de dados, como a nossa querida pandas.

De fato, uma Series é na verdade um array NumPy de dados associado a um array de rótulos. Por este motivo, normalmente é possível aplicar qualquer função matemática disponibilizada pela

NumPy sobre os dados de uma Series. Para consultar uma relação completa das funções NumPy, veja https://docs.scipy.org/doc/numpy/reference/ufuncs.html.

Se você estiver utilizando uma distribuição Python voltada para ciência de dados, não precisará se preocupar em instalar a NumPy. Caso contrário, poderá instalar a biblioteca através do utilitário `pip` (apresentado no capítulo anterior):

```
pip install -U numpy
```

O valor NaN

A figura a seguir mostra duas Series, denominadas `verde` e `azul`. Considere que a primeira indica se a cor verde faz parte das bandeiras do Brasil (BR), França (FR), Itália (IT) e Reino Unido (UK), onde o valor 1 representa sim e 0, não. Por sua vez, a Series `azul` indica se a cor azul faz parte das bandeiras dos mesmos países anteriores com o acréscimo da Argentina (AR).

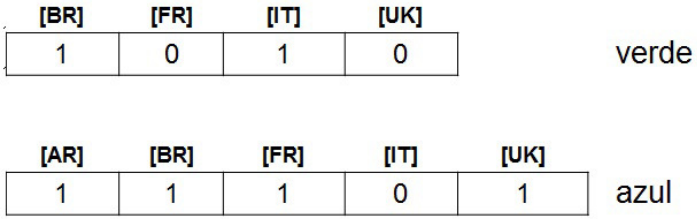

Figura 2.2: Series verde e azul, ambas contendo informações sobre cores de bandeiras de diferentes nações.

O programa a seguir examina o comportamento da pandas ao realizar uma operação aritmética (soma) envolvendo estas duas Series **que não possuem os rótulos inteiramente compatíveis**.

```
#P08: o valor NaN
import pandas as pd

verde = pd.Series({'BR':1, 'FR': 0, 'IT':1, 'UK': 0})
azul = pd.Series({'AR':1, 'BR':1, 'FR': 1, 'IT':0, 'UK': 1})

soma = verde + azul
print("soma:")
print(soma)
print('---------------------------')

print("isnull(soma):")
print(pd.isnull(soma))
```

Aqui está o resultado:

```
soma:
AR    NaN
BR    2.0
FR    1.0
IT    1.0
UK    1.0
dtype: float64
---------------------------
isnull(soma):
AR     True
BR    False
FR    False
IT    False
UK    False
dtype: bool
```

Como você deve ter notado, o resultado da soma para os rótulos existentes nas duas Series ('BR', 'FR', 'IT' e 'UK') ficou perfeitinho. No entanto, para o rótulo 'AR', que só existe em `azul`, o resultado da soma aparece como **`NaN`** (*not a number*). Mas o que é isso? Bem, `NaN` é o conceito utilizado pela pandas para marcar valores nulos/ausentes (*missing*) ou desconhecidos.

Em nosso exemplo, como 'AR' existe em `azul`, mas não tem um correspondente em `verde`, a pandas interpreta que o valor de

'AR' em `verde` é desconhecido. E o resultado de qualquer operação aritmética envolvendo um número conhecido e um desconhecido resultará sempre em desconhecido (ou seja, em `NaN`). Parece estranho, mas veja como faz sentido. Se eu fizer para você a seguinte pergunta: "Qual o resultado da soma do número 1 com outro número desconhecido". Certamente, você me responderia: "Eu não sei! Se o segundo valor é desconhecido, como posso responder?". E a pandas faz da mesma maneira: **1 + desconhecido = desconhecido**.

No mesmo programa, também aproveitamos para apresentar mais um método da pandas, denominado `isnull()`. Este método recebe como entrada uma Series `s`, gerando como saída uma outra Series com `dtype bool`, que indica quais dos rótulos de `s` estão associados a valores nulos (`NaN`). Existe também o método `notnull()` que faz o oposto, retornando `False` para todos os valores nulos.

Os exemplos apresentados evidenciaram as vantagens da computação vetorizada em processos de Data Wrangling. Por exemplo, no programa recém-apresentado, utilizamos apenas uma linha de código para implementar um processo de transformação de dados (criamos uma Series a partir de duas outras), ao fazermos `soma = verde + azul`. De maneira análoga, apenas uma linha de código foi suficiente para testarmos por valores ausentes na Series `soma`: `pd.isnull(soma)`.

2.4 ÍNDICES DATETIME

Como o seu próprio nome indica, a estrutura Series foi originalmente projetada para lidar com **séries temporais**

(https://www.datacamp.com/community/tutorials/time-series-analysis-tutorial), um dos temas mais estudados no campo da Estatística. Uma série temporal consiste em uma série de dados coletados em incrementos sucessivos de tempo ou algum outro tipo de indicador de sequência. Um exemplo bem simples é apresentado na figura a seguir: uma Series em que cada elemento representa um determinado dia do ano (rótulo) acompanhado da temperatura máxima registrada no dia em questão em uma determinada cidade (dado).

[10/02/2019]	[11/02/2019]	[12/02/2019]	[13/02/2019]	[14/02/2019]	[15/02/2019]
31	35	34	28	27	27

Figura 2.3: Exemplo simples de série temporal.

O programa a seguir mostra como podemos criar essa série temporal, configurando explicitamente o vetor de índices para que ele possua o **tipo data** (`dtype datetime64`). Afinal de contas, essa é a semântica correta da nossa série de registro temperaturas!

```
#P09: Índices datetime
import pandas as pd

#(1)-cria a série temporal
dias = ['10/02/2019', '11/02/2019','12/02/2019','13/02/2019',
        '14/02/2019','15/02/2019']
temp_max = [31,35,34,28,27,27]

serie_temporal = pd.Series(temp_max,index=dias)

#(2)-converte o tipo do índice para datetime e imprime a série
serie_temporal.index = pd.to_datetime(serie_temporal.index,
                                      format='%d/%m/%Y')
print(serie_temporal)
```

Esta é a saída do programa:

```
2019-02-10    31
2019-02-11    35
2019-02-12    34
2019-02-13    28
2019-02-14    27
2019-02-15    27
dtype: int64
```

Veja que no programa os índices foram originalmente passados como uma lista de strings: `dias = ['10/02/2019', '11/02/2019','12/02/2019','13/02/2019','14/02/2019','15/02/2019']` . O truque para transformar os valores deste índice para `datetime` consistiu no emprego do método `to_datetime()` com dois parâmetros:

- O nome do vetor de índices a ser convertido (em nosso caso, `serie_temporal.index`).
- `format='%d/%m/%Y'` : o parâmetro `format` é utilizado para que possamos informar a pandas o formato da data (**date string**). Para indicar este formato, foi preciso utilizar as máscaras `%d` (dia), `%m` (mês) e `%Y` (ano com quatro dígitos) combinadas com o caractere `/` . No mundo Python, essas máscaras são chamadas de diretivas (*directives*). A lista completa das diretivas pode ser consultada em http://strftime.org.

2.5 INDEXAÇÃO HIERÁRQUICA

A indexação hierárquica é um recurso oferecido pela pandas para permitir que você trabalhe com mais de um **nível de indexação**. Para que o conceito fique claro, o exemplo desta seção mostra como utilizar este recurso para criar uma Series com informações sobre os nomes das moedas dos cinco países

mostrados na figura a seguir. Desta vez, a Series poderá ser indexada não apenas pela sigla do país, mas também pelo nome de seu continente.

[América] [AR]	[América] [BR]	[Europa] [FR]	[Europa] [IT]	[Europa] [UK]
Peso	Real	Euro	Euro	Libra

Figura 2.4: Série com dois níveis de indexação.

```
#P10: Índexação hierárquica
import pandas as pd

moedas  = ['Peso', 'Real', 'Euro', 'Euro', 'Libra']
paises = [['América','América','Europa','Europa','Europa'],
          ['AR','BR','FR','IT','UK']]


paises = pd.Series(moedas, index=paises)

print(paises)                    #imprime toda a Series
print('----------------------')
print(paises['América'])         #{AR: Peso, BR:Real}
print('----------------------')
print(paises[:,'IT'])            #{Europa: Euro}
print('----------------------')
print(paises['Europa','IT'])     #Euro
```

Saída:

```
América  AR     Peso
         BR     Real
Europa   FR     Euro
         IT     Euro
         UK    Libra
dtype: object
----------------------
AR    Peso
BR    Real
dtype: object
----------------------
```

```
Europa    Euro
dtype: object
----------------------
Euro
```

Neste programa, a Series `paises` foi criada com um vetor de rótulos indexado em dois níveis. No primeiro nível temos o nome do continente ('América' ou 'Europa') e, no segundo, a sigla do país ('AR', 'BR', 'FR', 'IT' e 'UK'). Com isso, torna-se possível indexar os dados de três formas: apenas pelo nível 1, apenas pelo nível 2 ou por ambos. E foi exatamente o que fizemos no programa:

- `paises['América']` : este é um exemplo de indexação pelo nível 1 (nome do continente). O resultado é uma Series contendo as siglas e moedas dos países da América.
- `paises[:,'IT']` : aqui foi realizada a indexação pelo nível 2 (sigla do país). Como resultado, é retornada uma Series com um único elemento, em que o rótulo é o nome do continente e o valor é o nome da moeda da Itália.
- `paises['Europa','IT']` : este é um exemplo de indexação por ambos os níveis. Como resultado, será retornado 'Euro' (valor string), pois este é o valor do elemento cujo nível 1 do índice é 'Europa' e o nível 2 'IT'.

Capítulo 3

A ESTRUTURA DE DADOS DATAFRAME

DataFrame é a ED pandas utilizada para representar dados tabulares em memória, isto é, dados dispostos em **linhas** e **colunas**. Trata-se da ED mais importante para ciência de dados, responsável por disponibilizar um amplo e sofisticado conjunto de métodos para a importação e o pré-processamento de grandes bases de dados. São tantos métodos, que precisaremos de quatro capítulos para apresentá-los! Para começar, o presente capítulo cobre os seguintes tópicos:

- Como criar DataFrames?
- Técnicas para consulta e modificação de dados
- Trabalhando com arquivos

Adicionalmente, na seção final do capítulo veremos como empregar DataFrames para importar e explorar a base de dados **flags**, iniciando o processo de construção do classificador de cores de bandeiras.

3.1 COMO CRIAR DATAFRAMES?

O DataFrame é uma ED especialmente projetada para tornar o

processo de manipulação de **dados tabulares** mais rápido, simples e eficaz. A figura a seguir apresenta um exemplo: o DataFrame `paises`, que possui cinco linhas e quatro colunas e armazena informações sobre cinco diferentes nações.

	nome	*continente*	*extensão*	*corVerde*
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

Figura 3.1: Exemplo de DataFrame contendo dados de países.

Em sua aparência, o DataFrame é igual a uma planilha Excel, uma vez que possui linhas e colunas. No entanto, considerando a forma como a pandas organiza os DataFrames, cada coluna é, na verdade, uma Series. Mais especificamente, o DataFrame é um **dicionário de Series**, todas do mesmo tamanho (`size`). Tanto as suas linhas como as suas colunas podem ser indexadas e rotuladas.

No DataFrame `paises` as linhas são indexadas pela sigla do país. Este DataFrame possui as seguintes colunas: `nome` (nome do país), `continente` (nome do continente onde se localiza o país), `extensao` (extensão territorial em milhares de quilômetros quadrados) e `corVerde` (indica se a cor verde está presente na bandeira do país; o valor 1 representa sim e 0, não). O programa a seguir, apresenta o código para criar este DataFrame utilizando o método construtor padrão.

```
#P11: Hello DataFrame!
import pandas as pd
```

```
#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]
         }

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#imprime o DataFrame
print(paises)
```

Veja a saída do programa:

```
           nome continente  extensao  corVerde
AR    Argentina    América      2780         0
BR       Brasil    América      8511         1
FR       França     Europa       644         0
IT       Itália     Europa       301         1
UK  Reino Unido     Europa       244         0
```

Há várias formas para criar DataFrames. Neste exemplo, fizemos a partir de um **dicionários de listas** chamado `dados`. Observe que cada chave do dicionário foi transformada em uma coluna. As listas associadas às diferentes chaves possuem todas o mesmo tamanho (cinco elementos) e foram utilizadas para estabelecer os dados de cada coluna. Veja ainda que, em nosso exemplo, além de passar os dados do DataFrame, também criamos uma lista de índices de linhas (`siglas`), que foi passada para o método construtor com o uso do parâmetro `index`.

Todo DataFrame é um objeto do tipo `pandas.core.frame.DataFrame` que possui as seguintes propriedades básicas:

- `shape` : formato do DataFrame, ou seja, o seu número de linhas (`shape[0]`) e de colunas (`shape[1]`);
- `index` : lista com os rótulos das linhas;
- `columns` : lista com os rótulos das colunas;
- `dtypes` : retorna uma Series com os `dtypes` de cada coluna;
- `index.dtype` : `dtype` dos rótulos das linhas.

O programa listado a seguir mostra como trabalhar com estas propriedades.

```
#P12: Propriedades básicas dos DataFrames
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#recupera e imprime as propriedades
print('-----------')
num_linhas = paises.shape[0]
num_colunas = paises.shape[1]
indices = paises.index
colunas = paises.columns
paises_tipo = type(paises)
paises_dtypes = paises.dtypes
paises_idx_dtype = paises.index.dtype

print('número de linhas: ', num_linhas)
print('número de colunas: ', num_colunas)
print('rótulos das linhas: ', indices)
print('rótulos das colunas: ', colunas)
```

```
print('tipo (type): ',paises_tipo)
print('dtypes das colunas:\n', paises_dtypes)
print('dtype dos rótulos das linhas:', paises_idx_dtype)
```

Saída do programa:

```
número de linhas:  5
número de colunas:  4
rótulos das linhas:  Index(['AR', 'BR', 'FR', 'IT', 'UK'], dtype=
'object')
rótulos das colunas:  Index(['nome', 'continente', 'extensao', 'c
orVerde'], dtype='object')
tipo (type):  <class 'pandas.core.frame.DataFrame'>
dtypes das colunas:
 nome          object
continente    object
extensao       int64
corVerde       int64
dtype: object
dtype dos rótulos das linhas: object
```

O shape de um DataFrame

A forma de trabalhar com as propriedades dos DataFrames é muito parecida com a das Series. No entanto, a maneira de obter o número de linhas e colunas, com o uso da propriedade `shape`, pode causar alguma confusão. O `shape` de um DataFrame corresponde ao seu formato e é representado como uma tupla bidimensional (uma tupla é uma ED praticamente igual a uma lista). O número de linhas é armazenado em `shape[0]` e o de colunas, em `shape[1]`.

3.2 TÉCNICAS PARA CONSULTA E MODIFICAÇÃO DE DADOS

Indexação

Utilizamos colchetes `[ ]` para indexar elementos de um DataFrame. Assim como ocorre com as Series, é possível empregar três técnicas de indexação: indexação tradicional, fatiamento e indexação booleana. Nesta seção, abordaremos as duas primeiras, deixando a indexação booleana para a seção final do capítulo.

No quadro adiante, mostramos um resumo das opções disponíveis para indexação básica de células (operações que retornam um único elemento do DataFrame). Na notação utilizada, considere que `d` é o nome de um DataFrame em memória. Veja que a pandas disponibiliza quatro diferentes métodos para esta operação: `iloc()`, `iat()`, `loc()` e `at()`. Os dois primeiros são utilizados para indexação baseada na posição da linha (número inteiro), enquanto os dois últimos são para acessar linhas através de seus rótulos.

Sintaxe	Explicação
`d.iloc[i][j]`	retorna o valor da célula que ocupa a linha i, coluna j
`d.iat[i,j]`	retorna o valor da célula que ocupa a linha i, coluna j
`d.iloc[i]['col']`	retorna o valor da célula que ocupa a linha i, coluna denominada 'col'
`d.loc['idx'][j]`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna j
`d.loc['idx']['col']`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna denominada 'col'
`d.at['idx','col']`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna denominada 'col'

Algumas observações importantes:

- No vocabulário adotado pela pandas, o termo **index** é sempre utilizado para índices das linhas, enquanto o termo **column** é utilizado para os índices das colunas.
- Não esqueça que a primeira coluna está na posição 0, a segunda, na posição 1 etc. Da mesma forma, a primeira linha está na posição 0, a segunda, na posição 1 etc.

Veja a seguir alguns exemplos de indexação de células do DataFrame `paises`:

Exemplo	Resultado
`paises.iloc[1][0]`	Brasil
`paises.iat[1,0]`	Brasil
`paises.iloc[3]['corVerde']`	1
`paises.loc['IT'][1]`	Europa
`paises.loc['FR']['nome']`	França
`paises.at['FR','nome']`	França

O fatiamento de DataFrames pode ser realizado com o uso dos métodos `iloc()` (por posição) e `loc()` (por rótulo). Na tabela a seguir, são apresentadas as diferentes sintaxes que podem ser empregadas para fatiar colunas ou linhas inteiras. Em todos os casos, o resultado será retornado em objeto do tipo Series.

Sintaxe	Explicação
`d['col']`	retorna a coluna de nome 'col' (toda a coluna)
`d.col`	outra forma para retornar a coluna de nome 'col'

`d.loc['idx']`	retorna a linha associada ao índice de rótulo 'idx' (linha inteira)
`d.iloc[i]`	retorna a linha que ocupa a posição i (linha inteira)

Observe os exemplos:

Exemplo	Resultado
`paises['extensao']`	{AR: 2780, BR: 8511, FR: 644, IT: 301, UK: 244}
`paises.corVerde`	{AR: 0, BR: 1, FR: 0, IT: 1, UK: 0}
`paises.loc['BR']`	{nome: Brasil, continente: América, extensao: 8511, corVerde: 1}
`paises.iloc[2]`	{nome: França, continente: Europa, extensao: 644, corVerde: 0}

Agora serão apresentados exemplos de fatiamentos capazes de "recortar" pedaços do DataFrame que sejam diferentes de toda uma linha ou coluna. Neste caso, as "fatias" de linhas e colunas desejadas devem ser definidas dentro de colchetes `[ ]`, com o uso do operador dois-pontos (`:`) e separadas por vírgula (`,`). Primeiro deve-se definir a fatia de linhas e depois a de colunas. Para que o conceito fique claro, as figuras seguintes apresentam, de maneira visual, o resultado obtido por três diferentes operações de fatiamento sobre o DataFrame `paises`:

	nome	*continente*	*extensão*	*corVerde*
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

paises.iloc[:3, :2]

Figura 3.2: Primeiro exemplo de fatiamento.

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

```
paises.loc[['BR','IT'],'corVerde']
```

Figura 3.3: Segundo exemplo de fatiamento.

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

```
paises.iloc[-2:,1:3]
```

Figura 3.4: Terceiro exemplo de fatiamento.

Explicação:

- `paises.iloc[:3, :2]` : retorna uma fatia contendo as três primeiras linhas e duas primeiras colunas.
- `paises.loc[['BR','IT'],'corVerde']` : para as linhas rotuladas com `'BR'` e `'IT'` , retorna o valor armazenado na coluna `corVerde` .
- `paises.iloc[-2:,1:3]` : retorna uma fatia contendo as duas últimas linhas e as colunas de índice 1 e 2.

Você deve ter notado que a técnica para fatiar DataFrames é muito similar à que é empregada no fatiamento de Series. O que muda é apenas o fato de que você precisa especificar uma fatia para linhas e outra para colunas. Com relação ao tipo de objeto retornado, temos que o resultado do fatiamento de um DataFrame

será sempre um objeto do tipo DataFrame, exceto quando a operação resultar em uma única linha ou uma única coluna retornada. Neste caso, o tipo de objeto resultante é uma Series.

Busca

O próximo programa mostra:

1. Como verificar se um determinado rótulo de linha ou coluna existe em um DataFrame;
2. Como verificar se um determinado valor está armazenado em alguma coluna do DataFrame.

```
#P13: Busca em DataFrames
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#testa se um dado rótulo de linha existe
tem_BR = 'BR' in paises.index
tem_US = 'US' in paises.index
print("existe o rótulo 'BR? -> ",tem_BR)
print("existe o rótulo 'US'? -> ",tem_US)
print('---------------------------')

#testa se um dado rótulo de coluna existe
tem_corVerde = 'corVerde' in paises.columns
tem_corAzul = 'corAzul' in paises.columns
print("existe o rótulo 'corVerde? -> ",tem_corVerde)
print("existe o rótulo 'corAzul'? -> ",tem_corAzul)
```

```
print('--------------------------')

#testa se valor faz parte de uma coluna
tem_Brasil = paises['nome'].isin(['Brasil'])
print("existe o valor 'Brasil' na coluna 'nome'?")
print(tem_Brasil)
```

Saída do programa:

```
existe o rótulo 'BR? ->  True
existe o rótulo 'US'? ->  False
--------------------------
existe o rótulo 'corVerde? ->  True
existe o rótulo 'corAzul'? ->  False
--------------------------
existe o valor 'Brasil' na coluna 'nome'?
AR    False
BR     True
FR    False
IT    False
UK    False
Name: nome, dtype: bool
```

Para verificar se um rótulo de linha ou de coluna existe em um DataFrame, você deve aplicar o operador `in` sobre a lista de rótulos de linha ou de colunas, respectivamente (propriedades `index` e `columns`).

Se você quiser testar se um valor está armazenado em uma coluna, precisará utilizar o método `isin()`. Em nosso exemplo, passamos para o método uma lista com um único elemento (`'Brasil'`) e mandamos checar a coluna `nome`. Como resultado, o método indicou `True` para a sigla `BR`.

Modificação

Este tópico apresenta as técnicas básicas para inserir e alterar linhas de um DataFrame e para modificar o conteúdo de alguma

célula. Em capítulos posteriores voltaremos a abordar o assunto, introduzindo técnicas mais avançadas, capazes de realizar a transformação de valores de colunas inteiras.

```
#P14: Modificação de DataFrame
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#insere o país Japão (JP)
paises.loc['JP']= {'nome': 'Japão',
      'continente': 'Ásia',
      'extensao': 372,
      'corVerde': 0}

#altera a extensão do Brasil
paises.at['BR','extensao']=8512

#remove a Argentina e o Reino Unido
paises = paises.drop(['AR','UK'])

print('DataFrame após as alterações:')
print(paises)
```

A seguir, a saída do programa, que imprime `paises` após três diferentes operações de modificação: inserção do país Japão, alteração da extensão territorial do Brasil (de 8511 para 8512) e remoção dos países associados às siglas `'AR'` e `'UK'`.

```
DataFrame após as alterações:
      nome continente  extensao  corVerde
BR  Brasil    América      8512         1
```

```
FR  França     Europa       644         0
IT  Itália     Europa       301         1
JP   Japão       Ásia       372         0
```

- Para inserir o Japão, bastou indicar os dados desse país em um dicionário e realizar a atribuição com o uso do método `loc()`. Caso o DataFrame não possuísse rótulos de linha, seria preciso utilizar o método `iloc()` com o índice da linha a ser inserida.
- Para alterar uma célula, no caso, a extensão do Brasil, utilizou-se um comando de atribuição simples que empregou o método `at()` com a indicação dos rótulos de linha e coluna da célula a ser alterada. Caso o DataFrame não possuísse rótulos de linha, seria preciso utilizar o método `iat()` com o índice da linha a ser alterada.
- Por fim, para remover linhas, basta utilizar o método `drop()` indicando a lista de rótulos de linha a serem removidos.

3.3 TRABALHANDO COM ARQUIVOS

Tipicamente, os dados dos DataFrames são obtidos a partir de arquivos ou tabelas de banco de dados. Nesta seção, serão apresentadas as técnicas básicas para importar dados destas fontes para DataFrames pandas. Os exemplos utilizam pequenas bases de dados que se encontram disponíveis no **repositório de bases de dados** de nosso livro. O endereço do repositório é: https://github.com/edubd/pandas.

Importação de arquivo CSV

Podemos realizar a leitura de arquivos CSV (*comma-separated*

values — valores separados por vírgula) e de outros tipos de arquivos baseados em **caracteres delimitadores** utilizando o método `read_csv()`. Um exemplo de arquivo deste tipo é `paises.csv`, listado a seguir (note que o arquivo contém os mesmos dados usados nos exemplos anteriores):

```
sigla,nome,continente,extensao,corVerde
AR,Argentina,América,2780,0
BR,Brasil,América,8511,1
FR,França,Europa,644,0
IT,Itália,Europa,301,1
UK,Reino Unido,Europa,244,0
```

Suponha que este arquivo esteja armazenado na pasta `C:\bases` de seu computador (utilizaremos essa suposição para todos os exemplos de agora em diante — modifique o código se você preferir usar outra pasta). A seguir, apresenta-se um programa que importa o conteúdo do arquivo para um DataFrame, definindo ainda que a coluna `sigla` será utilizada como índice de linha.

```
#P15: Importação de CSV padrão para um DataFrame
import pandas as pd

paises = pd.read_csv("c:/bases/paises.csv",index_col="sigla")
print(paises)
```

Simples, não? A saída do programa demonstra que a importação foi perfeita:

```
             nome continente  extensao  corVerde
sigla
AR       Argentina    América      2780         0
BR          Brasil    América      8511         1
FR          França     Europa       644         0
IT          Itália     Europa       301         1
UK     Reino Unido     Europa       244         0
```

Nesse caso, o programa ficou bem pequeno porque o formato do arquivo de entrada estava idêntico ao que método `read_csv()` espera como padrão: os dados estão separados por vírgula e o arquivo possui cabeçalho. O único parâmetro que utilizamos foi `index_col` , para permitir que a coluna `sigla` fosse transformada no índice de linha.

O método `read_csv()` é extremamente flexível, possuindo uma série de parâmetros que podem ser utilizados para permitir a importação de arquivos CSV estruturados de diferentes maneiras. Os principais parâmetros são relacionados e explicados a seguir:

- `sep` : caractere ou expressão regular utilizada para separar campos em cada linha;
- `skiprows` : número de linhas no início do arquivo que devem ser ignoradas;
- `skip_footer` : número de linhas no final do arquivo que devem ser ignoradas;
- `encoding` : padrão de codificação do arquivo. A codificação default da pandas é **utf-8**. Se o seu arquivo estiver codificado no formato ANSI, você deverá utilizar `encoding='ansi'` . Para obter uma lista completa dos encodings, consulte https://docs.python.org/3/library/codecs.html#standard-encodings;
- `header` : número da linha que contém o cabeçalho (default=0). Se não houver cabeçalho, deve-se especificar `header=None` ou passar uma lista de nomes através do parâmetro `names` ;
- `names` : permite especificar uma lista de nomes para as colunas;

- `index_col` : permite que uma das colunas seja transformada em índice de linhas;
- `na_values` : sequência de valores que deverão ser substituídos por `NA` . Útil para transformação de dados;
- `thousands` : definição do separador de milhar, por exemplo `.` ou `,` ;
- `squeeze` : caso o arquivo CSV possua apenas uma coluna, é possível fazer com que ele seja importado para uma Series em vez de um DataFrame, bastando para isso especificar `squeeze=True` .

A seguir, serão apresentados exemplos que demonstram a utilização destes parâmetros. Vamos começar pelos mais utilizados na prática: `sep` e `names` . Será apresentado um programa capaz de importar e formatar o arquivo `notas.csv` , cujo conteúdo armazena as matrículas de quatro alunos e as notas por eles obtidas em duas diferentes provas.

```
M0012017;9.8;9.5
M0022017;5.3;4.1
M0032017;2.5;8.0
M0042017;7.5;7.5
```

Veja que o arquivo usa ponto e vírgula (`;`) como separador e não possui uma linha de cabeçalho. Para importá-lo e definir uma linha de cabeçalho, basta fazer da seguinte forma:

```
#P16: Importação de CSV sem cabeçalho e com ";" como separador
import pandas as pd

notas = pd.read_csv("c:/bases/notas.csv", sep=";",
                    names=['matricula','nota1','nota2'])
print(notas)
```

Saída:

```
  matricula  nota1  nota2
0  M0012017    9.8    9.5
1  M0022017    5.3    4.1
2  M0032017    2.5    8.0
3  M0042017    7.5    7.5
```

O parâmetro `sep` foi utilizado para definir `;` como separador, enquanto `names` foi empregado para definir os cabeçalhos das colunas. Desta vez, não utilizamos o parâmetro `index_col` e, por consequência, os índices de linha foram definidos como números inteiros. Para que fosse possível colocar matrículas como índices de linha, bastaria ter feito: `index_col='matricula'`.

Para demonstrar a utilização do parâmetro `squeeze`, considere o arquivo `gols.txt`, que armazena informações sobre uma sequência de sete jogos realizados por um time de futebol no mês de junho de 2019. Para cada jogo, registra-se a sua data de realização e o número de gols que o time marcou (observe que as duas informações são separadas por um espaço em branco).

```
dia gols
05/06/2019 1
09/06/2019 0
16/06/2019 5
19/06/2019 2
23/06/2019 1
27/06/2019 3
30/06/2019 0
```

Veja que esses dados correspondem a uma sequência de valores registrados em incrementos de tempo. Ou seja, eles representam uma série temporal. Sendo assim, nada mais justo do que importá-los para uma Series em vez de um DataFrame. Esta operação é bem simples com a pandas:

```
#P17: Importação de arquivo com série temporal
```

```
import pandas as pd

#importa o arquivo para uma Series
serie_gols = pd.read_csv("c:/bases/gols.txt", sep=" ",
                     squeeze=True, index_col=0)

#converte o tipo do índice para datetime e imprime a série
serie_gols.index = pd.to_datetime(serie_gols.index,
                                       format='%d/%m/%Y')
print(serie_gols)
```

Saída:

```
dia
2019-06-05    1
2019-06-09    0
2019-06-16    5
2019-06-19    2
2019-06-23    1
2019-06-27    3
2019-06-30    0
Name: gols, dtype: int64
```

Desta vez, três parâmetros foram utilizados no método `read_csv()`:

- `sep=" "` : para indicar que o separador é espaço em branco.
- `squeeze=True` : para retornar uma Series em vez de um DataFrame, já que, por padrão, o método `read_csv()` sempre retorna um DataFrame.
- `index_col=0` : para indicar que a primeira coluna do arquivo (`dia`) deverá ser usada para formar o vetor de rótulos da Series. Por consequência, os gols serão usados no vetor de rótulos.

MÉTODO `READ_TABLE()`

A pandas disponibiliza ainda um outro método para a leitura de arquivos separados por delimitador, denominado `read_table()`. Este método possui os mesmos parâmetros de `read_csv()`. A única diferença entre os dois métodos é que o `read_csv()` tem a vírgula `,` como separador padrão, enquanto `read_table()` utiliza a tabulação `\t`.

Importação de planilha Excel

Considere a planilha Excel `capitais.xlsx`, mostrada na figura a seguir. Ela contém a relação das capitais dos estados brasileiros localizados nas regiões Sul, Sudeste e Norte. Para cada capital, indica-se o seu nome (coluna A), região (coluna B) e a população estimada de acordo com o IBGE (coluna C). Os dados foram obtidos no site https://cidades.ibge.gov.br.

	A	B	C
1	**capital**	**região**	**população**
2	Belém	Sudeste	1446042
3	Belo Horizonte	Sudeste	2513451
4	Boa Vista	Norte	326419
5	Curitiba	Sul	1893977
6	Florianópolis	Sul	477798
7	Macapá	Norte	465495
8	Manaus	Norte	2094391
9	Palmas	Norte	279856
10	Porto Alegre	Sul	1481019
11	Porto Velho	Norte	511219
12	Rio Branco	Norte	377057
13	São Paulo	Sudeste	12038175
14	Rio de Janeiro	Sudeste	6498837
15	Vitória	Sudeste	359555

Figura 3.5: Planilha Excel com dados de capitais do Brasil.

Podemos importar esta planilha para um DataFrame pandas de maneira trivial, utilizando o método `read_excel()`.

```
#P18: Importação de planilha Excel
import pandas as pd

cidades = pd.read_excel("c:/bases/capitais.xlsx")
print(cidades)
```

Saída do programa:

```
          capital   região  população
0   Belém           Sudeste    1446042
1   Belo Horizonte  Sudeste    2513451
2   Boa Vista         Norte     326419
3   Curitiba            Sul    1893977
4   Florianópolis       Sul     477798
5   Macapá            Norte     465495
6   Manaus            Norte    2094391
```

```
7   Palmas             Norte     279856
8   Porto Alegre         Sul    1481019
9   Porto Velho        Norte     511219
10  Rio Branco         Norte     377057
11  São Paulo        Sudeste   12038175
12  Rio de Janeiro   Sudeste    6498837
13  Vitória          Sudeste     359555
```

Importação de arquivo JSON

JSON (*JavaScript Object Notation*) é um modelo para armazenamento e transmissão de informações no formato texto. Apesar de muito simples, é o mais utilizado por aplicações Web devido à sua capacidade de estruturar informações de forma compacta e autodescritiva. A listagem a seguir apresenta um exemplo de arquivo no formato JSON. O arquivo, denominado `notas.json`, contém as matrículas de quatro alunos e as notas por eles obtidas em duas provas (os dados são os mesmos do arquivo `notas.csv`, apresentado previamente, porém agora estruturados no formato JSON).

```
[
    {
     "matricula": "M0012017",
     "notas": [9.8, 9.5]
    },
    {
     "matricula": "M0022017",
     "notas": [5.3, 4.1]
    },
    {
     "matricula": "M0032017",
     "notas": [2.5, 8.0]
    },
    {
     "matricula": "M0042017",
     "notas": [7.5, 7.5]
    }
]
```

O arquivo lembra um pouco um dicionário, no sentido de que também é formado por chaves e valores. Neste exemplo, as chaves são `matricula` e `notas`. A primeira está associada a um valor String (matrícula do aluno). Já a segunda chave é um pouco mais interessante, pois está associada a um **vetor** de números reais, contendo duas notas. É exatamente aí que está o poder do JSON! Ao contrário do formato CSV, o JSON consegue representar não apenas dados tabulares, mas também dados complexos, como vetores, listas, hierarquias etc.

Neste livro, não apresentaremos uma explicação detalhada sobre o padrão JSON, pois trata-se de um assunto bastante abrangente e que não é específico da linguagem Python. No entanto, se você não conhece muito sobre o tema, a sugestão é consultar o tutorial introdutório disponibilizado em https://www.devmedia.com.br/json-tutorial/25275. O que será coberto aqui é a receita básica para importar dados de arquivos JSON para DataFrames pandas. Essa receita consiste em dois passos:

- Importar o arquivo JSON para um objeto em memória, utilizando o pacote `json` do Python padrão.
- Transferir os dados do objeto para um DataFrame.

```
#P19: Importação de arquivo JSON
import pandas as pd
import json

#(1)-Importa o arquivo JSON para memória
with open("c:/bases/notas.json") as f:
    j_notas = json.load(f)

#(2)-Transfere as informações para um DataFrame
notas = pd.DataFrame(j_notas,
                     columns=['matricula','notas'])
```

```
print(notas)
```

Saída do programa:

```
  matricula       notas
0  M0012017  [9.8, 9.5]
1  M0022017  [5.3, 4.1]
2  M0032017  [2.5, 8.0]
3  M0042017  [7.5, 7.5]
```

Conforme mencionado anteriormente, o processo tem dois passos:

- Na primeira parte temos a "receita de bolo" para a leitura de arquivos JSON, que consiste em utilizar o método `load()` do pacote `json`. Ao ser chamado, este método faz com que o arquivo JSON inteiro seja carregado para um objeto em memória (que no nosso programa, foi chamado de `j_notas`).
- Na segunda parte, transfere-se o conteúdo do objeto JSON em memória para um DataFrame. É preciso passar duas informações para o construtor padrão: o objeto JSON (`j_notas`) e a relação de chaves do objeto que desejamos mapear para colunas no DataFrame (utilizando o parâmetro `columns`).

Veja que, como resultado, foi gerado um DataFrame que é "fiel" ao formato original do arquivo JSON. Ele possui duas colunas, `matricula`, do tipo String, e `notas`, do tipo **lista de `floats`**. Interessante, não é? Um DataFrame suporta a definição de colunas que armazenam tipos complexos, como listas e outras EDs.

Gravação de arquivos

Se você quiser gravar o conteúdo de um DataFrame em memória para um arquivo CSV, deverá utilizar o método `to_csv()`.

```
#P20: Salva o conteúdo de um DataFrame em um CSV.
import pandas as pd

#cria o DataFrame
dados = {'codigo': [1001,1002,1003,1004,1005],
         'nome': ['Leite','Café','Biscoito','Chá',
                  'Torradas']
         }

produtos = pd.DataFrame(dados)

#salva o seu conteúdo para um arquivo
produtos.to_csv("C:/bases/produtos.csv", sep="\t", index=False)
```

Neste exemplo, realiza-se a gravação do arquivo `produtos.csv` a partir do conteúdo do DataFrame `produtos`. Nos parâmetros do método `to_csv()`, a tabulação `\t` foi adotada como caractere delimitador (parâmetro `sep`) e a opção `index=False` foi utilizada para evitar que os rótulos dos índices (números inteiros) fossem gravados no arquivo. O formato do arquivo produzido é este:

```
codigo    nome
1001    Leite
1002    Café
1003    Biscoito
1004    Chá
1005    Torradas
```

Se você quiser exportar o DataFrame para uma planilha Excel, poderá utilizar o método `to_excel()`. Sua forma básica de utilização é bem simples, como mostra-se a seguir:

```
produtos.to_excel("C:/bases/produtos.xlsx", index=False)
```

Para obter a relação com as opções avançadas do método `to_excel()`, consulte o link: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html.

4.3 RANQUEAMENTO E ORDENAÇÃO

Uma das provas de natação mais apreciadas pelo público é a dos 50m nado livre. Nos Jogos Olímpicos do Rio de Janeiro, em 2016, a final desta prova na categoria masculino, foi considerada uma das mais empolgantes. A figura a seguir mostra o resultado final dos oito nadadores que a disputaram. Para cada nadador, apresenta-se o número raia na qual ele realizou a prova, o seu nome, nacionalidade e tempo obtido.

raia	nadador	nacionalidade	tempo
1	Simonas Bilis	Lituânia	22.08
2	Benjamin Proud	Grã-Bretanha	21.68
3	Anthony Ervin	Estados Unidos	21.40
4	Florent Manaudou	França	21.41
5	Andriy Hovorov	Ucrânia	21.74
6	Nathan Adrian	Estados Unidos	21.49
7	Bruno Fratus	Brasil	21.79
8	Brad Tandy	África do Sul	21.79

Figura 4.7: DataFrame contendo o resultado da final dos 50m nado livre, masculino, nas Olimpíadas do Rio de Janeiro.

Note que o DataFrame está ordenado pelo número da raia, o que atrapalha a identificação do vencedor da prova! No próximo programa, mostramos como os métodos `sort_values()` e `rank()` podem ser utilizados para nos ajudar a resolver este problema. O primeiro método serve para ordenar o DataFrame e o segundo para produzir um ranqueamento.

```
#P27 Métodos sort_values() e rank()
import pandas as pd

#1-cria o DataFrame da prova de 50m
```

```
dados = {"nadador": ["Simonas Bilis",
                     "Benjamin Proud",
                     "Anthony Ervin",
                     "Florent Manaudou",
                     "Andriy Hovorov",
                     "Nathan Adrian",
                     "Bruno Fratus",
                     "Brad Tandy"],
         "nacionalidade": ["Lituânia",
                           "Grã-Bretanha",
                           "Estados Unidos",
                           "França",
                           "Ucrânia",
                           "Estados Unidos",
                           "Brasil",
                           "África do Sul"],
         "tempo": [22.08,
                   21.68,
                   21.40,
                   21.41,
                   21.74,
                   21.49,
                   21.79,
                   21.79]
         }

raias = list(range(1,9))

prova50m = pd.DataFrame(dados, index=raias)
prova50m.index.name = 'raia'

#2-ordena pelo tempo de forma crescente
prova50m = prova50m.sort_values(by="tempo")
print("* * resultado final ordenado por tempo:")
print(prova50m)

#3 - gera os rankings
resultado_por_raia = prova50m['tempo'].rank(method="min")
print("\n* * posição de cada nadador (por raia):")
print(resultado_por_raia)
```

Observe com atenção a saída do programa:

```
* * resultado final ordenado por tempo:
```

```
               nadador   nacionalidade  tempo
raia
3        Anthony Ervin  Estados Unidos  21.40
4     Florent Manaudou          França  21.41
6        Nathan Adrian  Estados Unidos  21.49
2       Benjamin Proud    Grã-Bretanha  21.68
5       Andriy Hovorov         Ucrânia  21.74
7         Bruno Fratus          Brasil  21.79
8           Brad Tandy   África do Sul  21.79
1        Simonas Bilis        Lituânia  22.08

* * posição de cada nadador (por raia):
      tempo
raia
3       1.0
4       2.0
6       3.0
2       4.0
5       5.0
7       6.0
8       6.0
1       8.0
```

Agora uma explicação detalhada sobre o programa. Mais uma vez, ele está dividido em três partes. A primeira trata simplesmente de criar e imprimir o DataFrame. O comando `prova50m.index.name = 'raia'` é usado para atribuir o rótulo `raia` à coluna que armazena o índice.

A segunda parte apresenta o método `sort_values()`. Este método serve para ordenar o DataFrame por uma ou mais colunas, que devem ser especificadas no parâmetro `by`. Neste exemplo, `by="tempo"` foi utilizado para gerar uma ordenação ascendente pela coluna `tempo`. Uma observação muito importante é que se você quiser mudar de fato o DataFrame (ou seja, ordená-lo de verdade em vez de apenas exibir as suas linhas ordenadas), precisará fazer uso de um comando de **atribuição**:

```
prova50m = prova50m.sort_values(by="tempo")
```

Alguns comentários adicionais sobre o método `sort_values()`:

- A ordenação default é ascendente. Para ordenar de forma descendente, você deve utilizar o parâmetro `ascending=False`.
- Para ordenar por mais de uma coluna, é preciso especificá-las em uma lista. Por exemplo, `by=["tempo","nacionalidade"]` faria com que a pandas realizasse a ordenação por tempo (primeiro critério) e, em casos de empate, por país (segundo critério). Da mesma maneira, para indicar explicitamente o tipo de ordenação de cada campo, use uma lista, como `ascending=[True,False]`.

A terceira e última parte do programa apresenta o método `rank()` que serve para gerar uma Series contendo ranking de valores:

```
resultado_por_raia = prova50m['tempo'].rank(method="min")
```

- O comando apresentado cria o ranqueamento com base nos valores da coluna `tempo`. Veja que a saída do programa mostra que o nadador da raia 3 ficou em primeiro no ranqueamento (menor tempo), o da raia 4 em segundo e assim por diante.
- O parâmetro `method="min"` serve para especificar o procedimento a ser adotado para o tratamento de empates. Neste exemplo, os nadadores das raias 7 e 8 empataram em

sexto lugar e `method="min"` faz com que a posição 6 seja atribuída para ambos. Outros valores possíveis seriam `method="max"` (7 seria atribuído para ambos), `method="average"` (colocaria 6.5 em ambos).

- Para gerar um ranqueamento por ordem decrescente de valores, basta empregar o parâmetro `ascending=False` .