ENGENHARIA DE *SOFTWARE* PARA INTELIGÊNCIA ARTIFICIAL

UNIDADE III

MANIPULAÇÃO DE DADOS (COM PYTHON)

Elaboração

Diogo Santos Silva da Costa

Atualização

Alexander Francisco Vargas Salgado

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

SUMÁRIO

CAPÍTULO 1

NUMPY

Numpy, o que é?

Numpy é um pacote criado principalmente para computação científica, que auxilia no tratamento de vetores e matrizes multidimensionais, possui diversas funções úteis na álgebra linear, estatística e matemática em geral.

Primeiro, importamos o pacote Numpy com o alias `np`. Chamamos de alias a um nome, nesse caso `np`, que damos a um objeto invocado, por exemplo, de uma biblioteca, neste caso.

```
import numpy as np
```

A estrutura básica do Numpy é seu *array*, que, matematicamente, são tensores (como vetores matrizes ou de ordem superior).

Criação de tensores

Criaremos um vetor e uma matriz.

» Vetor:

```
vetor = np.array([2,4,6,8,10,12,14,16,18,20,22,24])
print(vetor)
```

```
[ 2  4  6  8 10 12 14 16 18 20 22 24]
```

» Matriz:

```
matriz = np.array([
 [2,4,6],
 [8,10,12],
 [14,16,18],
 [20,22,24]])
print(matriz)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]
 [20 22 24]]
```

Vamos criar agora um tensor de ordem superior aos criados (ordem três). Veja a seguir:

```
tensor = np.array([[[1,2],
 [3,4]],
 [[5,6],
 [7,8]],
 [[9,10],
 [11,12]]])
print(tensor)
```

```
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
```

Vamos explorar um pouco mais as dimensões dos objetos já criados.

O método *ndim* fornece a ordem do objeto. Por exemplo, o objeto `vetor` possui ordem um:

```
vetor.ndim
```

```
1
```

Já o objeto `matriz` possui ordem dois:

```
matriz.ndim
```

```
2
```

Quanto ao objeto `tensor`, possui ordem três:

```
tensor.ndim
```

```
3
```

Apesar de o nome `ndim` lembrar o número de dimensões, o método se refere à ordem do objeto.

Vamos entender um pouco sobre as ordens dos tensores:

```
vetor[0]
```

```
2
```

Para obter algum elemento do `vetor`, é necessário um índice. Já a matriz, que possui ordem dois, precisa de dois índices:

```
matriz[2]
```

```
array([14, 16, 18])
```

```
matriz[2,1]
```

```
16
```

Perceba que `matriz [2]` é, na verdade, um vetor. Ou seja, quando identificamos apenas uma ordem, temos como resposta um vetor. Para termos apenas um elemento na saída, temos de especificar dois índices.

No tensor de ordem três, se utilizarmos apenas um índice, teremos uma matriz:

```
1 tensor[1]
```

```
array([[5, 6],
       [7, 8]])
```

Com dois índices, teremos um vetor:

```
1 tensor[0,1]
```

```
array([3, 4])
```

Com três índices, teremos um elemento do objeto `tensor`.

```
1 tensor[0,1,1]
```

```
4
```

Podemos obter o número de elementos em cada ordem do objeto a partir da propriedade `shape`. O vetor possui uma ordem com 12 elementos:

```
1 vetor.shape
```

```
(12,)
```

A `matriz` possui duas ordens, uma com quatro elementos e outra com três elementos:

```
1 matriz.shape
```

```
(4, 3)
```

O `tensor` possui três ordens, a primeira com três, a segunda com dois e a terceira com dois elementos:

```
1 tensor.shape
```

```
(3, 2, 2)
```

A propriedade `size` retorna o número de elementos considerando todas as ordens.

O `vetor` possui 12 elementos totais:

```
1 vetor.size
```

```
12
```

A `matriz` possui 12 elementos totais:

```
1 matriz.size
```

```
12
```

No caso do `tensor`, possui 12 elementos totais:

```
1 tensor.size
```

```
12
```

A propriedade `size` retorna o produto entre os valores do `shape`.

Criaremos alguns tensores especiais a partir de funções do numpy.

Usamos a função `zeros(n)` do numpy para criar um vetor nulo de shape `n`.

```
1 vetor_nulo = np.zeros(3)
2 vetor_nulo
```

```
array([0., 0., 0.])
```

Faremos algo similar para criar uma matriz de elementos nulos. Aqui o argumento `n` será uma lista.

```
1 matriz_nula = np.zeros([3,3])
2 matriz_nula
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

A criação de um tensor de ordem três nulo pode ser feita de forma análoga.

```
1 tensor_nulo = np.zeros([3,4,2])
2 tensor_nulo
```

```
array([[[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]]])
```

Em vez de zeros, se quisermos ter elementos repetidos dentro do tensor, usamos a função `full` especificando o `shape` e o valor a ser repetido:

```
1 vetor = np.full((1,5), 7)
2 vetor
```

```
array([[7, 7, 7, 7, 7]])
```

```
1 matriz = np.full((4,5),3)
2 matriz
```

```
array([[3, 3, 3, 3, 3],
       [3, 3, 3, 3, 3],
       [3, 3, 3, 3, 3],
       [3, 3, 3, 3, 3]])
```

```
tensor = np.full((4,3,2,2), 9)
tensor
```

```
array([[[[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]]],


       [[[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]]],


       [[[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]]],


       [[[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]],

        [[9, 9],
         [9, 9]]]])
```

Nesta etapa, demonstramos como pode ser criada uma matriz diagonal com a função `diag`. Basta inserir os elementos da diagonal, assim:

```
diagonal = np.array([1,2,3,4,5])
matriz_diagonal = np.diag(diagonal)
matriz_diagonal
```

```
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

Caso queiramos uma matriz identidade, só é preciso que a diagonal tenha elementos unitários repetidos:

```
matriz_identidade = np.diag([1,1,1,1])
matriz_identidade
```

```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

Seria mais fácil, porém, utilizando a função `eye`:

```
matriz_identidade = np.eye(4)
matriz_identidade
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Podemos criar tensores de números aleatórios. Para isso, importamos o pacote `random`:

```
from numpy import random as rd
```

A função `rand` retorna uma matriz com entradas criadas aleatoriamente a partir de uma distribuição uniforme. Necessita apenas que sejam fornecidos os parâmetros do `shape`, que, nesse caso, foram duas linhas e três colunas:

```
matriz_aleatorio = rd.rand(2,3)
matriz_aleatorio
```

```
array([[0.59289349, 0.04803127, 0.36907076],
       [0.06311559, 0.10814737, 0.84065277]])
```

Nesse caso, temos uma matriz gerada a partir de uma distruibição normal padrão de valores em seus elementos:

```
matriz_normal = rd.randn(2,3)
matriz_normal
```

```
array([[-1.10631744,  0.49249688,  0.51958046],
       [-0.35421301, -0.70240616,  0.97679291]])
```

A função `randint` retorna apenas valores inteiros, dentro de uma faixa preestabelecida. Foi estabelecido que a faixa seria de números de 0 a 100, para um tensor de ordem três com `shape(2,3,3)`.

```
matriz_truncada = rd.randint(0, 100,(2,3,3))
matriz_truncada
```

```
array([[[ 7, 54, 21],
        [51, 15, 46],
        [ 2, 87, 15]],

       [[35, 56, 77],
        [ 0,  8, 51],
        [78, 36, 18]]])
```

Agora, vamos ver alguns exemplos de como o Numpy pode ser utilizado para operar cálculos matemáticos com tensores. Para isso, definiremos alguns vetores e matrizes:

```
vetor_1 = np.array([1,2,5,7,11])
vetor_1
```

```
array([ 1,  2,  5,  7, 11])
```

```
vetor_2 = np.array([2,4,6,8,10])
vetor_2
```

```
array([ 2,  4,  6,  8, 10])
```

```
matriz_1 = np.eye(3)
matriz_1
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
matriz_2 = np.array([[1,2,3],[4,5,6],[7,8,9]])
matriz_2
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Operações matemáticas

Após definidos os tensores, vamos começar com as operações:

```
1 vetor_1 + vetor_2
```

```
array([ 3,  6, 11, 15, 21])
```

```
1 matriz_1 + matriz_2
```

```
array([[ 2.,  2.,  3.],
       [ 4.,  6.,  6.],
       [ 7.,  8., 10.]])
```

Acima, temos a soma de dois vetores e de duas matrizes, respectivamente. As outras operações básicas da matemática são semelhantes, basta trocar o operador matemático da soma por subtração (–), divisão (/) ou multiplicação (*). A seguir, temos a multiplicação de um tensor por um escalar:

```
1 vetor_1 * 2
```

```
array([ 2,  4, 10, 14, 22])
```

No caso da função `dot`, é responsável por realizar o produto interno entre as matrizes e os vetores:

```
1 np.dot(vetor_1, vetor_2)
```

```
206
```

```
1 np.dot(matriz_1, matriz_2)
```

```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

Note que há uma diferença entre realizar o produto interno das matrizes e a operação de multiplicação. A multiplicação é feita elemento a elemento:

```
1 matriz_1 * matriz_2
```

```
array([[1., 0., 0.],
       [0., 5., 0.],
       [0., 0., 9.]])
```

Tanto no vetor quanto na matriz podemos trabalhar com seus elementos de forma individual como uma lista:

```
1 matriz_2[0,2] * vetor_1[3]
```

```
21
```

É possível comparar tensores entrada por entrada:

```
1 matriz_1 > matriz_2
```

```
array([[False, False, False],
       [False, False, False],
       [False, False, False]])
```

Funções matemáticas

O Numpy fornece uma série de funções matemáticas que podem ser aplicadas tanto em escalares quanto em tensores. Vamos ver alguns exemplos a seguir.

Assim, a partir da função `exp`, é possível calcular a função exponencial de todos os elementos dentro do objeto fornecido:

```
1 np.exp(1)
```

```
2.718281828459045
```

```
1 np.exp(vetor_1)
```

```
array([2.71828183e+00, 7.38905610e+00, 1.48413159e+02, 1.09663316e+03,
       5.98741417e+04])
```

```
1 np.exp(matriz_1)
```

```
array([[2.71828183, 1.        , 1.        ],
       [1.        , 2.71828183, 1.        ],
       [1.        , 1.        , 2.71828183]])
```

De forma bastante intuitiva, também podemos determinar o seno dos elementos dados com a função `sin`. Para cosseno, é utilizado `cos` e, para tangente, `tan`.

```
1 np.sin(matriz_2)
```

```
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ],
       [ 0.6569866 ,  0.98935825,  0.41211849]])
```

```
1 np.cos(matriz_1)
```

```
array([[0.54030231, 1.        , 1.        ],
       [1.        , 0.54030231, 1.        ],
       [1.        , 1.        , 0.54030231]])
```

```
1 np.tan(vetor_1)
```

```
array([   1.55740772,   -2.18503986,   -3.38051501,    0.87144798,
       -225.95084645])
```

É possível usar também funções estatísticas, conforme exemplos a seguir.

A função `mean` calcula a média de um determinado conjunto de valores:

```
1 np.mean(vetor_1)
```

```
5.2
```

A função `std` calcula o desvio padrão dos valores:

```
1 np.std(vetor_1)
```

```
3.6
```

Manipulação de matrizes

Neste momento, faremos algumas manipulações nas matrizes. Dada a matriz abaixo:

```
1 matriz = np.array([[14, 14, 6],[-17, 0, -1],[15, -5, -6]])
2 matriz
```

```
array([[ 14,  14,   6],
       [-17,   0,  -1],
       [ 15,  -5,  -6]])
```

Vamos transformar essa matriz 3 x 3 em uma linha. Ela se tornará um vetor com nove elementos. A função `ravel` faz com que os elementos sejam distribuídos em uma única linha, de forma ordenada da direita para a esquerda, linha por linha:

```
1 matriz.ravel()
```

```
array([ 14,  14,   6, -17,   0,  -1,  15,  -5,  -6])
```

É possível mudar a quantidade de linhas e colunas em uma matriz como quisermos utilizando o método `shape`. A única coisa que deve ser respeitada é a quantidade de elementos na matriz. O novo `shape` deve respeitar a propriedade `size`:

```
1 matriz.shape = (9,1)
2 matriz
```

```
array([[ 14],
       [ 14],
       [  6],
       [-17],
       [  0],
       [ -1],
       [ 15],
       [ -5],
       [ -6]])
```

```
1 matriz.shape = (3,3)
2 matriz
```

```
array([[ 14,  14,   6],
       [-17,   0,  -1],
       [ 15,  -5,  -6]])
```

Podemos, por fim, transpor a matriz trocando linhas por colunas com o método `transpose` ou, simplesmente, a propriedade `T`:

```
1 matriz.transpose()
```

```
array([[ 14, -17,  15],
       [ 14,   0,  -5],
       [  6,  -1,  -6]])
```

```
1 matriz.T
```

```
array([[ 14, -17,  15],
       [ 14,   0,  -5],
       [  6,  -1,  -6]])
```

CAPÍTULO 2

PANDAS

Pandas, o que é?

Pandas é uma biblioteca para *Python* pensada para facilitar a gestão de dados. Com esse recurso, podemos carrega-los em arquivos de diferentes formatos, aplicar transformações, junções com outros conjuntos de dados e remover aqueles que estiverem incorretos ou desnecessários, permitindo gerar um novo conjunto de dados como resultado final, dados esses que podem ser analisados ou, como é de nosso interesse, usados como entrada para os sistemas de aprendizado de máquina com os quais trabalharemos.

No nosso exemplo, vamos usar o Pandas para processar *bad data*, que é o termo empregado para dados que não estão organizados em forma útil para o trabalho em questão.

Reparando dados ruins

Faremos a seguir um passo a passo sobre as boas práticas no tratamento de dados (limpeza de dados de pouco valor para a nossa análise).

Carregando dados

Vamos usar dados de uma fonte pública, o livro vermelho, que contém ocorrências de espécies em extinção no Brasil. Esse é um exemplo de dados organizados para leitura humana; mas que, quando usados para leitura em máquina, oferecem muitos problemas. Primeiro passo é carregar e importar o módulo pandas. Vamos dar um alias a ele, o mais comum nos exemplos é o `pd`.

```
1 import pandas as pd
```

A seguir, vamos carregar a planilha. Para carregar esse formato, usamos a função `read_excel` do módulo pandas, assim:

```
1 df = pd.read_excel("D:\Documentos\especiesavaliadaslv2013.xlsx")
```

O objeto `df` é do tipo `DataFrame`, é o equivalente a uma tabela de banco de dados. Para ter uma ideia dos dados que carregamos, usamos o método `head(n)` para ter uma amostra do `DataFrame`, vendo as n primeiras linhas. A tabela 1 demonstra a saída parcial do comando a seguir:

```
1 df.head(3)
```

Tabela 1. Saída parcial do conjunto de dados.

	familia	nome cientifico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Podemos ter uma amostra olhando as últimas n linhas também, usando o método `tail(n)`. A tabela 2 exibe a saída:

```
1 df.tail(5)
```

Tabela 2. Saída dos dados com o método tail(n).

	familia	nome cientifico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
4612	XYRIDACEAE	Xyris vacillans	Malme	EN	A2c	Rafael Augusto Xavier Borges	Miguel d'Avila de Moraes	As subpopulações de Xyris vacillans estão dist...	4/20/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Cerrado, Mata Atlântica
4613	XYRIDACEAE	Xyris wawrae	Heimerl	EN	A2c,B2ab(iii)	Rafael Augusto Xavier Borges	Miguel d'Avila de Moraes	Xyris wawrae ocorre em habitats severamente fr...	4/20/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Cerrado, Mata Atlântica
4614	ZAMIACEAE	Zamia ulei	Dammer	LC	NaN	Luiz Antonio Ferreira dos Santos Filho	Tainan Messina	Zamia ulei conhecida popualrmente por "batatad...	9/27/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia
4615	ZINGIBERACEAE	Renealmia brasiliensis	K.Schum.	EN	B2ab(ii,iii,iv)	Luiz Antonio Ferreira dos Santos Filho	Tainan Messina	Renealmia brasiliensis conhecida popularmente ...	9/27/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
4616	ZINGIBERACEAE	Renealmia petasites	Gagnep.	LC	NaN	Luiz Antonio Ferreira dos Santos Filho	Tainan Messina	Renealmiapetasites conhecida popularmente como...	9/27/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Em ambos os casos, o valor padrão para o número de linhas `n` é `5`, com isso:

Existem diversos outros tipos de arquivos que podem ser carregados nativamente com o Pandas. Alguns formatos e a função de leitura: `CSV: read_csv, JSON: read_json, HTML: read_html, Área de transferência: read_clipboard, HDF5: read_hdf, Parquet: read_parquet`

Preenchendo lacunas

Uma olhada no `DataFrame` mostra que existem muitos valores `NaN`, isto é, `Not a Number`, que, caso não conheça, é o valor universalmente atribuído a erros numéricos. Pandas usa o `NaN` para identificar campos não preenchidos. Quando alguém analisa os dados visualmente, consegue distinguir com facilidade células vazias de células não preenchidas. `NaN` atrapalha os trabalhos porque, se temos colunas do tipo texto, não podemos operá-las com facilidade, visto que temos de nos preocupar com os `NaN`, os quais são tipos numéricos especiais. Do mesmo modo, em campos numéricos, `NaN` causa erros em adições e produtos, assim temos de converter isso para valores que façam sentido no negócio. Para tanto, usamos a função `fillna`. Primeiramente, acessamos uma coluna, por exemplo, família, que é aceita:

```
1 df['familia'].head(10)
```

```
0    ACANTHACEAE
1    ACANTHACEAE
2    ACANTHACEAE
3    ACANTHACEAE
4    ACANTHACEAE
5    ACANTHACEAE
6    ACANTHACEAE
7    ACANTHACEAE
8    ACANTHACEAE
9    ACANTHACEAE
Name: familia, dtype: object
```

Fica claro que ela possui valores indefinidos, mas vamos ver onde ela possui valores definidos:

```
1 df['familia'][df['familia'].notnull()]
```

Tabela 3 – Exemplo de dataframe.

```
0         ACANTHACEAE
1         ACANTHACEAE
2         ACANTHACEAE
3         ACANTHACEAE
4         ACANTHACEAE
             ...
4612       XYRIDACEAE
4613       XYRIDACEAE
4614        ZAMIACEAE
4615    ZINGIBERACEAE
4616    ZINGIBERACEAE
Name: familia, Length: 4617, dtype: object
```

Fonte: Elaboração própria do autor, 2020.

Essa sintaxe pode ser complicada, então vamos explicar. As colunas de um `DataFrame` são do tipo Series, tanto uma Series quanto um `DataFrame` podem ter elementos acessados pelo índice, como:

```
1 df['familia'][4614]
```

```
'ZAMIACEAE'
```

```
1 df['familia'][3]
```

```
'ACANTHACEAE'
```

Ou por uma lista de índices:

```
df['familia'][[4614, 3, 4, 4616]]
```

```
4614        ZAMIACEAE
3         ACANTHACEAE
4         ACANTHACEAE
4616    ZINGIBERACEAE
Name: familia, dtype: object
```

Quando usamos a função `notnull` em uma *Series*, estamos recebendo uma nova *Series* que só possui os elementos não nulos (e não `NaN`) da *Series* original:

```
df['familia'].notnull()
```

Tabela 4 – Exemplo de dataframe lógico.

```
0       True
1       True
2       True
3       True
4       True
        ...
4612    True
4613    True
4614    True
4615    True
4616    True
Name: familia, Length: 4617, dtype: bool
```

Fonte: Elaboração própria do autor, 2020.

Essa *Series* tem valores verdadeiro e falso e índices. Quando usamos isso como parâmetro de entrada, teremos os elementos da *Series* cujo resultado foi verdadeiro na listagem de entrada, o que nos retorna ao valor original:

```
df['familia'][df['familia'].notnull()]
```

Tabela 5 – Exemplo de dataframe do tipo "object".

```
0         ACANTHACEAE
1         ACANTHACEAE
2         ACANTHACEAE
3         ACANTHACEAE
4         ACANTHACEAE
             ...
4612       XYRIDACEAE
4613       XYRIDACEAE
4614        ZAMIACEAE
4615    ZINGIBERACEAE
4616    ZINGIBERACEAE
Name: familia, Length: 4617, dtype: object
```

Fonte: Elaboração própria do autor, 2020.

Essa forma de filtrar elementos é uma das maiores utilidades do Pandas.

Agora vamos preencher os vazios das diversas colunas, conforme visto a seguir. 'Família aceita' é uma coluna de texto, então vamos preencher com uma `string` vazia e verificar a saída dos dados na tabela 6:

```
df['familia'] = df['familia'].fillna('')
df.head()
```

Tabela 6. Saída dos dados com o método *head*.

	familia	nome científico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

O campo nome científico também é textual, então vamos aplicar-lhe a mesma transformação e observer sua saída na Tabela 7.

```
df['nome científico'] = df['nome científico'].fillna('')
df.head()
```

Tabela 7. Saída dos dados com o método *head*.

	familia	nome científico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Fazemos o mesmo com o 'bioma'. Observemos na tabela 8:

```
df['bioma'] = df['bioma'].fillna('')
df.head()
```

Tabela 8. Saída dos dados com o método *head*.

```
df['bioma'] = df['bioma'].fillna('')
df.head()
```

	familia	nome científico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

O mesmo para o campo 'avaliador' e sua saída na tabela 9.

```
df['avaliador'] = df['avaliador'].fillna('')
df.head()
```

Tabela 9. Saída dos dados com o método *head*.

	familia	nome científico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Outro campo com valores vazios é 'justificativa'. Novamente, é um campo de texto com valores `NaN`, vamos resolver isso usando a técnica anterior. Podemos observar sua saída na tabela 10.

```
df['justificativa'] = df['justificativa'].fillna('')
df.head()
```

Tabela 10. Saída dos dados com o método *head*.

	familia	nome científico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	link	bioma
0	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Mudando o campo 'id'

Um *DataFrame* do Pandas tem um campo identificador, o index.

```
1 df.index
```

```
RangeIndex(start=0, stop=4617, step=1)
```

A princípio, ele é uma sequência e isso poderia nos atender. Entretanto, se olharmos o campo 'link', vemos que ele tem o fomato de um identificador único, um `uuid`. Podemos usar esse campo como identificador das linhas do `DataFrame`, até porque esse é o identificador fornecido pelos gestores do dado. Para ter certeza, precisamos conferir se os valores são únicos, uma forma de fazer isso é contar o número de valores diferentes e ver se é o mesmo número de valores total:

```
1 df['link'].shape[0] == df['link'].unique().shape[0]
```

```
True
```

Certo, uma vez que temos a comprovação de que os identificadores são únicos, podemos substituir o índice do conjunto de dados pelo identificador usando a função `set_index`, dessa maneira:

```
1 df.set_index('link', inplace=True)
```

Perceba o parâmetro `inplace=True`, por padrão o seu valor é False. Esse parâmetro é comum em muitas funções do tipo `DataFrame`, indicando que as alterações devem ser feitas no `DataFrame` atual, e não em uma cópia. Por padrão, o `DataFrame` é imutável, o que significa que alterações realizadas só terão efeito em uma cópia dele, mas podemos burlar isso em alguns casos com o método `inplace`.

Agora, é possível analisar o `DataFrame df` e veremos que o índice mudou. A saída é exibida na Tabela 11:

```
df.head()
```

Tabela11. Saída dos dados com o método head.

link	familia	nome cientifico	autor	categoria	critério	avaliador	revisor	justificativa	data da avaliacao	bioma
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20espirito-santensis	ACANTHACEAE	Aphelandra espirito-santensis	Profice & Wassh.	EN	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Como o epíteto específico sugere, é uma espéci...	3/14/2012	Mata Atlântica
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20gigantea	ACANTHACEAE	Aphelandra gigantea	(Rizzini) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Também conhecida como Crista-de-Galinha, a esp...	2012-04-04 00:00:00	Mata Atlântica
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20hirta	ACANTHACEAE	Aphelandra hirta	(Klotzsch) Wassh.	LC	NaN	Julia Caram Sfair	Tainan Messina	A. hirta possui ampla distribuição (EOO = 99.5...	2012-04-04 00:00:00	Mata Atlântica
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20longiflora	ACANTHACEAE	Aphelandra longiflora	(Lindl.) Profice	LC	NaN	Julia Caram Sfair	Tainan Messina	Espécie de ampla distribuição e é encontrada e...	2012-04-04 00:00:00	Amazônia, Mata Atlântica, Cerrado
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20margaritae	ACANTHACEAE	Aphelandra margaritae	E.Morren	VU	B1ab(iii)	Julia Caram Sfair	Tainan Messina	Apesar de amplamente coletada e de ocorrer em ...	2012-04-04 00:00:00	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Limpando *categoria*

Analisemos agora o campo `categoria`.

```
df['categoria'].unique()
```

```
array(['EN', 'LC', 'VU', 'CR', 'NT', 'DD'], dtype=object)
```

O campo 'categoria' tem seis valores possíveis. Podemos associar o valor EN a True, o valor DD a False e, assim, estaremos lidando com um campo booleano, o que faz muito mais sentido nesse caso. Quanto aos campos NaN, vamos associar o valor None, que, em Python, é o equivalente a um valor vazio, não preenchido, em linguagens como Java ou C, esse seria o valor null.

Vamos usar a função `apply` e uma função `lambda`, que é uma função sem nome, definida em uma linha:

```
lambda v: True if v == 'EN' else (False if v == 'DD' else None)
```

```
<function __main__.<lambda>(v)>
```

Essa função recebe como parâmetro o valor da célula em questão e compara com `'EN'`; em caso positivo, devolve `True`, se não, avalia se o valor é `'DD'`. Nessa situação, associa `False`, e, caso não seja, associa `None`. Nesse sentido:

```
df['categoria'] = df['categoria'].apply(lambda v: True if v == 'EN' else (False if v == 'DD' else None))
df['categoria']
```

Tabela 12. Tabela de visualização de dataframe

link	
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20espirito-santensis	True
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20gigantea	None
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20hirta	None
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20longiflora	None
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20margaritae	None
	...
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Xyris%20vacillans	True
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Xyris%20wawrae	True
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Zamia%20ulei	None
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Renealmia%20brasiliensis	True
http://cncflora.jbrj.gov.br/portal/pt-br/profile/Renealmia%20petasites	None

Name: categoria, Length: 4617, dtype: object

Fonte: Elaboração própria do autor, 2020.

Datas

Vamos dar uma olhada no campo `'data da avaliacão'`:

```
1 data = df['data da avaliacao'].unique()
2 data
```

```
array([0, nan, datetime.datetime(2012, 4, 4, 0, 0),
       datetime.datetime(2012, 1, 10, 0, 0),
       datetime.datetime(2012, 6, 8, 0, 0), '8/29/2012',
       datetime.datetime(2012, 8, 5, 0, 0), '12/31/1969',
       datetime.datetime(2012, 2, 4, 0, 0), '4/18/2012',
       datetime.datetime(2012, 9, 4, 0, 0),
       datetime.datetime(2012, 11, 4, 0, 0),
       datetime.datetime(2012, 5, 4, 0, 0),
       datetime.datetime(2012, 10, 4, 0, 0),
       datetime.datetime(2012, 12, 4, 0, 0), '7/19/2012', '8/13/2012',
       '8/22/2012', '5/25/2012', '5/17/2012',
       datetime.datetime(2012, 6, 6, 0, 0),
       datetime.datetime(2012, 4, 6, 0, 0),
       datetime.datetime(2012, 7, 6, 0, 0),
       datetime.datetime(2012, 12, 6, 0, 0), '5/15/2012',
       datetime.datetime(2012, 8, 6, 0, 0),
       datetime.datetime(2012, 9, 11, 0, 0),
       datetime.datetime(2012, 11, 6, 0, 0),
       datetime.datetime(2012, 1, 6, 0, 0),
       datetime.datetime(2012, 6, 2, 0, 0), '2/17/2012', '2/16/2012',
       datetime.datetime(2012, 10, 5, 0, 0), '9/25/2012',
       datetime.datetime(2012, 10, 10, 0, 0), '3/14/2012',
       datetime.datetime(2012, 7, 3, 0, 0), '2/29/2012',
       datetime.datetime(2012, 6, 3, 0, 0),
       datetime.datetime(2012, 8, 3, 0, 0), '10/23/2012', '5/21/2012',
       datetime.datetime(2012, 2, 5, 0, 0), '5/31/2012', '5/24/2012',
       datetime.datetime(2012, 12, 7, 0, 0), '6/29/2012',
       datetime.datetime(2012, 5, 6, 0, 0), '9/24/2012', '4/26/2012',
       '3/19/2012', '4/25/2012', datetime.datetime(2012, 9, 5, 0, 0),
       '3/29/2012', '4/27/2012', '3/22/2012', '3/20/2012', '3/21/2012',
       '5/14/2012', datetime.datetime(2012, 3, 5, 0, 0),
```

```
'3/29/2012', '4/27/2012', '3/22/2012', '3/20/2012', '3/21/2012',
'5/14/2012', datetime.datetime(2012, 3, 5, 0, 0),
datetime.datetime(2012, 7, 5, 0, 0),
datetime.datetime(2012, 11, 5, 0, 0),
datetime.datetime(2012, 7, 8, 0, 0),
datetime.datetime(2011, 3, 10, 0, 0), '8/20/2012', '8/28/2012',
'8/27/2012', '8/17/2012', '8/24/2012', '8/21/2012', '8/23/2012',
'9/13/2012', datetime.datetime(2012, 10, 9, 0, 0), '9/17/2012',
'9/18/2012', '1/31/2012', '4/16/2012', '4/13/2012',
datetime.datetime(2011, 1, 11, 0, 0),
datetime.datetime(2012, 3, 4, 0, 0), '4/20/2012', '4/17/2012',
datetime.datetime(2012, 1, 2, 0, 0), '4/28/2012', '4/29/2012',
'5/22/2012', datetime.datetime(2012, 3, 7, 0, 0),
datetime.datetime(2012, 5, 7, 0, 0),
datetime.datetime(2012, 6, 7, 0, 0), '9/27/2012', '9/28/2012',
'3/15/2012', datetime.datetime(2012, 10, 2, 0, 0),
datetime.datetime(2012, 9, 2, 0, 0), '3/13/2012', '3/16/2012',
'3/26/2012', datetime.datetime(2012, 8, 2, 0, 0), '7/26/2012',
datetime.datetime(2012, 3, 10, 0, 0),
datetime.datetime(2012, 2, 10, 0, 0), '8/14/2012',
datetime.datetime(2012, 12, 9, 0, 0),
datetime.datetime(2012, 11, 9, 0, 0), '9/19/2012',
datetime.datetime(2012, 12, 11, 0, 0), '4/19/2012', '8/30/2012',
'7/25/2012', '8/31/2012', '9/30/2011',
datetime.datetime(2011, 4, 10, 0, 0),
datetime.datetime(2012, 2, 3, 0, 0), '10/19/2011', '10/21/2011',
'9/26/2012', datetime.datetime(2012, 4, 9, 0, 0), '6/13/2012',
'3/23/2012', '5/29/2012', '6/21/2012', '6/26/2012', '7/16/2012',
'3/27/2012', '3/30/2012', '4/24/2012', '3/28/2012', '5/16/2012',
...
datetime.datetime(2012, 9, 3, 0, 0),
datetime.datetime(2012, 12, 3, 0, 0),
datetime.datetime(2011, 7, 11, 0, 0),
datetime.datetime(2012, 10, 8, 0, 0),
datetime.datetime(2012, 9, 8, 0, 0),
datetime.datetime(2012, 8, 8, 0, 0), '8/16/2012',
datetime.datetime(2012, 3, 8, 0, 0), '6/20/2012', '6/14/2012',
'10/14/2012', '2/27/2012', datetime.datetime(2012, 5, 3, 0, 0),
datetime.datetime(2012, 11, 7, 0, 0), '5/23/2012',
datetime.datetime(2012, 3, 9, 0, 0),
datetime.datetime(2012, 2, 8, 0, 0), '7/30/2012', '6/28/2013',
'9/21/2012', datetime.datetime(2012, 5, 9, 0, 0), '8/15/2012',
'10/28/2011', '10/25/2011', '10/31/2011', '10/27/2011',
'10/24/2011', datetime.datetime(2011, 3, 11, 0, 0), '4/30/2012',
'2/13/2012', '9/28/2011', '9/29/2011', '5/30/2012',
datetime.datetime(2012, 4, 7, 0, 0), '6/22/2012', '7/23/2012',
'5/28/2012', datetime.datetime(2012, 10, 7, 0, 0),
datetime.datetime(2012, 5, 10, 0, 0),
datetime.datetime(2012, 4, 10, 0, 0), '6/15/2012',
datetime.datetime(2012, 2, 7, 0, 0), '6/27/2012', '9/20/2012',
datetime.datetime(2012, 1, 3, 0, 0), '2/28/2012', '10/17/2011',
'10/18/2011', '9/15/2011', '9/16/2011', '9/19/2011', '9/20/2011',
datetime.datetime(2011, 11, 10, 0, 0), '5/18/2012',
datetime.datetime(2012, 3, 2, 0, 0)], dtype=object)
```

"Data da avaliacao" possui um valor NaN, mas também um valor zero, ambos não fazem sentido como valor de uma data, mas o ideal é escolher um deles. Escolheremos o porque, sendo um valor numérico adequado, pode ser ordenado com os demais, além disso faz sentido agrupar e ordenar elementos de acordo com o mês de coleta. Outro ponto é que os meses são números de ponto flutuante, isso não tende a criar muitos problemas, mas idealmente trabalhamos com valores inteiros.

A seguir, vamos substituir NaN por 0 (inteiro):

```
1 df['data da avaliacao'].fillna(0, inplace=True)
2 df['data da avaliacao'].tail(10)
```

```
4607    2012-08-05 00:00:00
4608    2012-08-05 00:00:00
4609    2012-08-05 00:00:00
4610    2012-08-05 00:00:00
4611              4/20/2012
4612              4/20/2012
4613              4/20/2012
4614              9/27/2012
4615              9/27/2012
4616              9/27/2012
Name: data da avaliacao, dtype: object
```

Agora, transformamos os demais em `strings`:

```
df['data da avaliacao'] = df['data da avaliacao'].apply(str)
df['data da avaliacao'].tail(10)
```

```
4607    2012-08-05 00:00:00
4608    2012-08-05 00:00:00
4609    2012-08-05 00:00:00
4610    2012-08-05 00:00:00
4611              4/20/2012
4612              4/20/2012
4613              4/20/2012
4614              9/27/2012
4615              9/27/2012
4616              9/27/2012
Name: data da avaliacao, dtype: object
```

Alguns dados têm meses de coleta maiores que 12, a princípio isso não faz sentido, mas não conhecemos os dados o suficiente para afirmar. De qualquer forma, vamos aproveitar a oportunidade para aprender algo novo e modificar a coluna "válido" que identifica as linhas válidas ou não. Comecemos invalidando, então, aquelas que têm valor no campo mês da coleta 0 ou maior que 12:

```
def invalidate_by_month(row):
 return False if (row['data da avaliacao'] > '01/07/12' or row['data da avaliacao'] == 0) else row.valido
 df.valido = df.T.apply(invalidate_by_month)
```

Visualizemos os resultados. Primeiro, os meses de coleta com valor adequado, de 1 a 12, na Tabela 13.

```
df[df['data da avaliacao'] <= 12][df['data da avaliacao'] > 0][['data da avaliacao', 'válido']].head(8)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1:
UserWarning: Boolean Series key will be reindexed to match DataFrame index.
 """Entry point for launching an IPython kernel.
```

Tabela 13. Saída dos dados com o método *head*.

id da ocorrência	Data da avaliacao	válido
54c7ca1f3d7b2	5	True
54c7ca1f3d839	11	True
54c7ca1f3d8bb	11	True
54c7ca1f3d93b	12	True
54c7ca1f3d9bc	9	True
54c7ca1f3da3d	9	True
54c7ca1f3dabc	5	True
54c7ca5212870	7	True

Fonte: Elaboração própria do autor, 2020.

Agora, os meses de coleta com valor maior que 12, com a saída dos dados na Tabela 10.

```
df[df['data da avaliacao'] > 12][['data da avaliacao', 'válido']].head(8)
```

Tabela 14. Saída dos dados com o método *head*.

id da ocorrência	data da avaliacao	válido
54c7ca2a440b7	13	False
54c7ca35d504f	20	False
54c7ca3eddae6	23	False
54c7ca423e80b	60	False
54c7ca4bf2766	71	False
54c7cb01d6f16	71	False

Fonte: Elaboração própria do autor, 2020.

Por fim, os meses de coleta com valor 0, sendo demonstrados na Tabela 11.

```
In [35]: 1 df[df['data da avaliacao'] == 0][['data da avaliacao', 'válido']].head(8)
```

Tabela 15. Saída dos dados com o método *head*.

id da ocorrência	data da avaliacao	válido
54c7ca1f3de39	0	False
54c7cb5c3610c	0	False
54c7ca1f3e2ca	0	False
54c7ca1f3e3c6	0	False
54c7ca1f3e443	0	False
54c7ca1f3e4c0	0	False
54c7ca67f2090	0	False
54c7ca692dfa3	0	False

Fonte: Elaboração própria do autor, 2020.

Outro campo que merece nossa atenção é dia da coleta. Primeiro, vamos olhar os valores únicos.

```
1 dias = df['dia da coleta'].unique()
2 dias.sort()
3 dias
```

```
array([0.000e+00, 1.000e+00, 2.000e+00, 3.000e+00, 4.000e+00, 5.000e+00,
 6.000e+00, 7.000e+00, 8.000e+00, 9.000e+00, 1.000e+01, 1.100e+01,
 1.200e+01, 1.300e+01, 1.400e+01, 1.500e+01, 1.600e+01, 1.700e+01,
 1.800e+01, 1.900e+01, 2.000e+01, 2.100e+01, 2.200e+01, 2.300e+01,
 2.400e+01, 2.500e+01, 2.600e+01, 2.700e+01, 2.800e+01, 2.900e+01,
 3.000e+01, 3.100e+01, 3.200e+01, 1.934e+03, 1.982e+03,    nan])
```

Temos dias iguais a 0, esses registros devem ter a coluna de validade nula, bem como os que apresentam valores acima de 31.

Precisamos, neste momento, utilizar uma função do pacote numpy, a função *isnan*. Ela retorna um valor booleano, `True` se o valor for `NaN` ou `False` para quaisquer outros valores de entrada.

Com isso, testemos se o valor, para cada uma das células, é igual a 0, maior que 31 ou `NaN`.

```
import numpy as np
def invalidate_by_day(row):
 if (row['data da avaliacao'] > 31 or row['data da avaliacao'] == 0):
  return False
 elif np.isnan(row['data da avaliacao']):
  return False
 else:
  return row.válido
```

Até então usamos a função `apply` apenas em uma *Series* que só contém linhas, empregando o apply em um `DataFrame`. Ele itera nas colunas, mas precisamos que essa iteração ocorra nas linhas, então usamos a transposta com a propriedade `T`.

Utilizar a propriedade T é equivalente à função transpose().

```
df.válido = df.T.apply(invalidate_by_day)
```

Verifiquemos, então, o que foi feito. Primeiro, ao olhar a coluna 'dia da coleta' com valores iguais a 0 ou acima de 31, esperamos que a coluna "válido" exiba resultados False na Tabela 16.

```
df[(df['data da avaliacao'] > 31) | (df['data da avaliacao'] == 0)][['data da avaliacao', 'válido']].head(10)
```

Tabela 16. Saída dos dados com o método *head*.

id da ocorrência	data da avaliacao	válido
54c7ca870687f	0.0	False
54c7ca8704319	0.0	False
54c7ca8705b2e	0.0	False
54c7ca8ef2489	0.0	False
54c7cab0f3e78	0.0	False
54c7cb6fdc54e	1934.0	False
54c7cab7e66a1	0.0	False
54c7cab0f3c5d	0.0	False
54c7caa82cab6	0.0	False
54c7caa82cb32	0.0	False

Fonte: Elaboração própria do autor, 2020.

Agora, examinamos o campo “válido” atribuído às linhas com a coluna “dia da coleta” com valores no intervalo de 1 até 31, como é possível ver na tabela 17.

```
In [ ]: 1 df[(df['data da avaliacao'] <= 31) & (df['data da avaliacao'] != 0)][['data da avaliacao', 'válido']].head(10)
```

Tabela 17. Saída dos dados com o método *head*.

id da ocorrência	data da avaliacao	válido
54c7ca1f3d7b2	15.0	True
54c7ca1f3d839	23.0	True
54c7ca1f3d8bb	21.0	True
54c7ca1f3d93b	1.0	True
54c7ca1f3d9bc	26.0	True
54c7ca1f3da3d	26.0	True
54c7ca1f3dabc	15.0	True
54c7ca5212870	17.0	True
54c7ca722d1e6	26.0	True
54c7ca1f3db3d	23.0	True

Fonte: Elaboração própria do autor, 2020.

Podemos fazer uma rápida visualização das colunas presentes no conjunto de dados:

```
1 df.columns
```

```
Index(['familia', 'nome científico', 'autor', 'categoria', 'critério',
       'avaliador', 'revisor', 'justificativa', 'data da avaliacao', 'link',
       'bioma'],
      dtype='object')
```

Vamos observar a coluna “revisor” e sua relação com a coluna “autor” na saída da Tabela 18.

```
1 df[['autor', 'revisor']].head(10)
```

Tabela 18. Saída dos dados com o método *head*.

	autor	revisor
0	Profice & Wassh.	Tainan Messina
1	(Rizzini) Profice	Tainan Messina
2	(Klotzsch) Wassh.	Tainan Messina
3	(Lindl.) Profice	Tainan Messina
4	E.Morren	Tainan Messina
5	(Nees) Benth.	Tainan Messina
6	Nees & Mart.	Tainan Messina
7	(Vell.) Hiern	Tainan Messina
8	(Nees) Profice	Tainan Messina
9	Nees	Tainan Messina

Fonte: Elaboração própria do autor, 2020.

Caso fosse identificado problemas na coluna autor, por exemplo, a qual também poderia apresentar células `NaN`. Vamos corrigir essas células substituindo por `string` vazia. A saída é exibida na tabela 19.

```
df['autor'] = df['autor'].fillna('')
df[['autor']].head(10)
```

Tabela 19. Saída dos dados com o método *head*.

	autor
0	Profice & Wassh.
1	(Rizzini) Profice
2	(Klotzsch) Wassh.
3	(Lindl.) Profice
4	E.Morren
5	(Nees) Benth.
6	Nees & Mart.
7	(Vell.) Hiern
8	(Nees) Profice
9	Nees

Fonte: Elaboração própria do autor, 2020.

Vamos ver os valores possíveis para a coluna `'autor'`:

```
df['autor'].unique()
```

```
array(['Profice & Wassh.', '(Rizzini) Profice', '(Klotzsch) Wassh.', ...,
       'Kral & L.B.Sm.', 'L.A.Nilsson ex Malme', 'Gagnep.'], dtype=object)
```

Suponhamos que observarmos que temos o valor 0, que seria o nome de um identificador na coluna autor. Precisamos substituir também por *string* vazia.

Vamos usar o método *apply* para isso. Utilizaremos uma função anônima (lambda) que retorna *string* vazia caso o valor seja 0 ou o próprio valor, caso contrário. A saída do comando é exibida na Tabela 20.

```
df['autor'] = df['autor'].apply(lambda v: "" if v == 0 else v)
df[['autor', 'revisor']].head(8)
```

Tabela 20. Saída dos dados com o método head.

	autor	revisor
0	Profice & Wassh.	Tainan Messina
1	(Rizzini) Profice	Tainan Messina
2	(Klotzsch) Wassh.	Tainan Messina
3	(Lindl.) Profice	Tainan Messina
4	E.Morren	Tainan Messina
5	(Nees) Benth.	Tainan Messina
6	Nees & Mart.	Tainan Messina
7	(Vell.) Hiern	Tainan Messina

Fonte: Elaboração própria do autor, 2020.

Analisemos os dados contidos nas colunas `bioma` e `link`, exibidos na tabela 21.

```
df[['link','bioma']].head(8)
```

Tabela 21. Saída dos dados com o método *head*.

	link	bioma
0	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
1	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
2	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
3	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Amazônia, Mata Atlântica, Cerrado
4	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
5	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
6	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica
7	http://cncflora.jbrj.gov.br/portal/pt-br/profi...	Mata Atlântica

Fonte: Elaboração própria do autor, 2020.

Notamos várias células repetidas, vejamos os valores únicos.

```
df['link'].unique()
```

```
array(['http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20espirito-santensis',
       'http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20gigantea',
       'http://cncflora.jbrj.gov.br/portal/pt-br/profile/Aphelandra%20hirta',
       ...,
       'http://cncflora.jbrj.gov.br/portal/pt-br/profile/Zamia%20ulei',
       'http://cncflora.jbrj.gov.br/portal/pt-br/profile/Renealmia%20brasiliensis',
       'http://cncflora.jbrj.gov.br/portal/pt-br/profile/Renealmia%20petasites'],
      dtype=object)
```

Na coluna `link`, temos mais informações, portanto, caso existam, podemos substituir esses valores indeterminados por `string` vazia, exibidos na Tabela 22.

```
df.link = df.link.fillna('')
df[['link']].head(10)
```

Tabela 22. Saída dos dados com o método head.

	link
0	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
1	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
2	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
3	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
4	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
5	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
6	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
7	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
8	http://cncflora.jbrj.gov.br/portal/pt-br/profi...
9	http://cncflora.jbrj.gov.br/portal/pt-br/profi...

Fonte: Elaboração própria do autor, 2020.

Agora a coluna código da instituição.

```
df['critério'].unique()
```

```
array(['B1ab(iii)', nan, 'B1ab(iii,v)', 'D2', 'B1ab(v)',
       'B1ab(iii)+2ab(iii)', 'B2ab(iii)', 'A2c', 'B2ab(ii,iii,iv)',
       'B1ab(ii,iii,iv)+2ab(ii,iii,iv)', 'B1ab(i,iii,iv)',
       'B1ab(iii)+2ab(i,iii)', 'B1ab(i,ii,iii)', 'B1ab(i,iii)',
       'B1ab(ii,iii)', 'A4cd,B2ab(iii,iv)', 'B2ab(ii,iii)', 'B2ab(iii,v)',
       'A4c,B1ab(iii,iv)+2ab(iii,iv),C2a(i)',
       'A4cd,B1ab(iii,iv)+2ab(iii,iv)', 'B2ab(iii,iv)', 'B2ab(iii,iv,v)',
       'B1ab(iii,iv,v)+2ab(iii,iv,v)', 'A4acd,C2a(ii)',
       'B1ab(iii,iv)+2ab(iii,iv)', 'B1b(i,iii)+2ab(iii)',
       'B1ab(i,ii,iii,iv)', 'B1ab(i,ii,iii)+2ab(i,ii,iii)',
       'B1ab(iii,iv)', 'B2ab(i,ii,iii,iv,v)', 'B2ab(ii,iii,iv,v)',
       'B1ab(ii,iii,iv,v)', 'B1ab(ii,iii)+2ab(ii,iii)', 'B2ab(i,ii,iii)',
       'C1', 'B1ab(i,ii,iii,iv)+2ab(i,ii,iii,iv)',
       'B1b(i,ii,iii)+2b(i,ii,iii)', 'A2abcde', 'B1ab(iii,v)+2ab(iii,v)',
       'A4cd', 'B1ab(i,ii,iii,iv,v)', 'A1acd', 'B2ab(ii,iii,v)',
       'B2ab(iii)c(ii)', 'B1ab(i,ii,iii,iv)+2b(i,ii,iii,iv)',
       'B2ab(i,ii,iii,iv)', 'B1ab(i,iii,iv,v)+2ab(ii,iii,iv,v)',
```

Continua...

Como não temos mais interesse na coluna, vamos somente apagá-la, para isso usamos a função `drop`.

```
df.drop('critério', axis=1, inplace=True)
```

Usamos o parâmetro `axis` para indicar que queremos apagar as colunas, e não as linhas.

Agora vemos que a coluna critério não está mais entre as colunas.

```
df.columns
```

```
Index(['familia', 'nome científico', 'autor', 'categoria', 'avaliador',
       'revisor', 'justificativa', 'data da avaliacao', 'link', 'bioma'],
      dtype='object')
```

A próxima coluna a ser analisada é “categoria”.

```
df['categoria'].unique()
```

```
array(['EN', 'LC', 'VU', 'CR', 'NT', 'DD'], dtype=object)
```

Nesse sentido, notamos que há diversas siglas para as categorias. Vamos representá-las com suas siglas dobradas. A lista de estados é pequena.

Tabela 23 – Lista de valores do tipo “string”.

```
categoria_replaces=[('EN','EENN'),('LC','LLCC'),('VU','VVUU'),('CR','CCRR'),('NT','NNTT'),('DD','DDDD')]
```

Fonte: Elaboração própria do autor, 2020.

Vamos iterar sobre a lista e substituir cada um dos valores, os antigos para os novos.

```
for antigo, novo in categoria_replaces:
 df.categoria = df.categoria.str.replace(antigo, novo)
```

Assim, observamos que os valores únicos para a coluna `'categoria'` estão coerentes.

```
categoria_replaces=[('EN','EENN'),('LC','LLCC'),('VU','VVUU'),('CR','CCRR'),('NT','NNTT'),('DD','DDDD')]
```

Por fim, vamos salvar os dados corrigidos em alguns formatos.

```
df.to_csv('dados_ajustados.csv')
```

Além da função `to_csv`, que salva em csv, podemos gerar outros formatos de saída de arquivo. Alguns deles estão listados a seguir:

```
Excel: to_excel
JSON: to_json
HTML: to_html
Área de transferência: to_clipboard
HDF5: to_hdf
Parquet: to_parquet
```

CAPÍTULO 3

MAP E REDUCE

Inteligência Artificial (IA) exige dados, muitos deles. No mundo real, trabalhar com esses dados exige um grande poder de processamento ou muito tempo, de modo que a melhor forma de equilibrar ambos é gerar instâncias de processamento menor, as quais podem ser executadas em *hardwares* baratos e simples. Quando precisamos de mais poder de processamento, em lugar de ampliar a capacidade de processamento do *hardware* em questão, adicionamos novas unidades de processamento, novos nós.

Hoje em dia é comum uma máquina *desktop* ter quatro, oito ou mesmo 12 núcleos. Esse poder de processamento pode ser empregado ao menos para introduzir a ideia de um algoritmo de `MapReduce`. Ao longo do material complementar, vamos expandir os conceitos estudados aqui para sistemas com múltiplas máquinas em lugar de múltiplos processadores.

Para uma estrutura `map`, criamos uma função convenientemente chamada de `mapper` (mapeadora), e, para uma estrutura `reduce`, uma função `reducer`. Ambas são funções que atuam em listas, isto é, conjuntos de dados, porém de forma bem diferente. Vamos a elas.

Matematicamente, um mapa é, assim, uma função que leva cada elemento de um conjunto de origem a elementos de um novo conjunto em que temos os elementos transformados. Já computacionalmente, um mapa é uma função que atua em uma lista transformando os elementos individualmente, isto é, dado um tensor:

$$U = (u_0, \ldots, u_n)$$

A transformação:

$$y = f(U)$$

A transformação *f* é uma estrutura *map* somente se existe um *mapper g(u)* tal que:

$$f(U) = (g(u_1), \ldots, g(u_n))$$

Ou seja, a transformação `map` implica que a função `mapper` é aplicada de forma independente em cada coordenada.

Para ilustrar, vamos a um exemplo muito simples de `map`:

Como sempre, o primeiro passo é carregar as bibliotecas que vamos utilizar:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Agora, vamos carregar um conjunto de dados de salários do funcionalismo público do Brasil.

```
df = pd.read_excel("D:\Documentos\salarios_mensais_funcionalismo_publico_Brasil.xlsx")
df.head(10)
```

Tabela 24. Tabela gerada pelo código.

	Unnamed: 0	id	job	sector	month_salary	13_salary	eventual_salary	indemnity	extra_salary	discount_salary	total_salary
0	0	0	OFICIAL ADMINISTRATIVO	DETRAN	2315.81	0.00	0.0	0.0	73.85	0.0	1929.34
1	1	1	SD 2C PM	PM	3034.05	0.00	0.0	0.0	651.82	0.0	2265.96
2	2	2	1TEN PM	PM	8990.98	0.00	0.0	0.0	626.75	0.0	6933.04
3	3	3	MAJ PM	SPPREV	13591.02	0.00	0.0	0.0	0.00	0.0	10568.36
4	4	4	AG.TEC. DE ASSIT. A SAUDE	HCFMUSP	4203.67	0.00	0.0	0.0	0.00	0.0	3561.88
5	5	5	SD 1C PM	PM	4373.69	0.00	0.0	0.0	451.26	0.0	4152.98
6	6	6	1CFO PM	PM	2311.76	0.00	0.0	0.0	0.00	0.0	1967.28
7	7	7	BENEFICIARIO DE SERVIDOR ESTADUAL-IPESP	SPPREV	1699.19	0.00	0.0	0.0	0.00	0.0	1699.19
8	8	8	CABO PM	PM	3767.78	0.00	0.0	0.0	626.75	0.0	2703.47
9	9	9	SD 1C PM	PM	3821.99	603.57	0.0	0.0	601.68	0.0	3010.04

Fonte: Elaboração própria do autor, 2020.

Antes de mais nada, vamos ordernar esse conjunto de dados pelo campo `total_salary` e, apenas para visualização, tomaremos os 10 menores salários.

```
df.sort_values("total_salary").head(10)
```

Tabela 20. Tabela gerada pelo código.

	Unnamed: 0	id	job	sector	month_salary	13_salary	eventual_salary	indemnity	extra_salary	discount_salary	total_salary
7	7	7	BENEFICIARIO DE SERVIDOR ESTADUAL-IPESP	SPPREV	1699.19	0.00	0.0	0.0	0.00	0.0	1699.19
0	0	0	OFICIAL ADMINISTRATIVO	DETRAN	2315.81	0.00	0.0	0.0	73.85	0.0	1929.34
6	6	6	1CFO PM	PM	2311.76	0.00	0.0	0.0	0.00	0.0	1967.28
1	1	1	SD 2C PM	PM	3034.05	0.00	0.0	0.0	651.82	0.0	2265.96
8	8	8	CABO PM	PM	3767.78	0.00	0.0	0.0	626.75	0.0	2703.47
9	9	9	SD 1C PM	PM	3821.99	603.57	0.0	0.0	601.68	0.0	3010.04
4	4	4	AG.TEC. DE ASSIT. A SAUDE	HCFMUSP	4203.67	0.00	0.0	0.0	0.00	0.0	3561.88
5	5	5	SD 1C PM	PM	4373.69	0.00	0.0	0.0	451.26	0.0	4152.98
2	2	2	1TEN PM	PM	8990.98	0.00	0.0	0.0	626.75	0.0	6933.04
3	3	3	MAJ PM	SPPREV	13591.02	0.00	0.0	0.0	0.00	0.0	10568.36

Fonte: Elaboração própria do autor, 2020.

Vemos que os menores salários ficam na faixa de R$10 mil. Agora ordenaremos de forma decrescente e avaliaremos os maiores salários:

```
df.sort_values("total_salary", ascending=False).head(10)
```

Tabela 21. Tabela gerada pelo código.

	Unnamed: 0	id	job	sector	month_salary	13_salary	eventual_salary	indemnity	extra_salary	discount_salary	total_salary
3	3	3	MAJ PM	SPPREV	13591.02	0.00	0.0	0.0	0.00	0.0	10568.36
2	2	2	1TEN PM	PM	8990.98	0.00	0.0	0.0	626.75	0.0	6933.04
5	5	5	SD 1C PM	PM	4373.69	0.00	0.0	0.0	451.26	0.0	4152.98
4	4	4	AG.TEC. DE ASSIT. A SAUDE	HCFMUSP	4203.67	0.00	0.0	0.0	0.00	0.0	3561.88
9	9	9	SD 1C PM	PM	3821.99	603.57	0.0	0.0	601.68	0.0	3010.04
8	8	8	CABO PM	PM	3767.78	0.00	0.0	0.0	626.75	0.0	2703.47
1	1	1	SD 2C PM	PM	3034.05	0.00	0.0	0.0	651.82	0.0	2265.96
6	6	6	1CFO PM	PM	2311.76	0.00	0.0	0.0	0.00	0.0	1967.28
0	0	0	OFICIAL ADMINISTRATIVO	DETRAN	2315.81	0.00	0.0	0.0	73.85	0.0	1929.34
7	7	7	BENEFICIARIO DE SERVIDOR ESTADUAL-IPESP	SPPREV	1699.19	0.00	0.0	0.0	0.00	0.0	1699.19

Fonte: Elaboração própria do autor, 2020.

Temos salários totais, aquele junto de benefícios, na ordem de R$10 mil, ou seja, a amplitude do nosso conjunto é pequena. Há desde valores pequenos a valores médios. Vamos calcular, com isso, a mediana do campo `total_salary`, isto é, o valor de centro do conjunto de dados:

```
1 salario_mediano = df.total_salary.sort_values().median()
2 salario_mediano
```

```
2856.755
```

Percebemos que a mediana está na faixa de 2.900,00, mais próximo aos menores salários que aos maiores, ou seja, poucos recebem salários tão altos, havendo muitos salários mais baixos. Aproveitando, vamos também calcular a média:

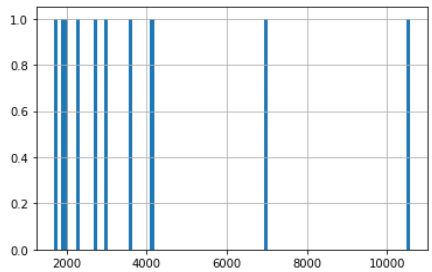
```
1 salario_medio = df.total_salary.mean()
2 salario_medio
```

```
3879.154
```

Novamente, apesar da amplitude do conjunto, o salário médio está bem mais próximo ao limite inferior do que ao superior, o que nos sugere que, mesmo com salários tão altos, há um número muito grande de salários baixos que puxa o conjunto médio pra valores menores. Tudo isso é mais interessante se for visto em um gráfico. Nesse sentido:

Gráfico 1. Imagem gerada pelo código.

```
1 ax = df.total_salary.sort_values().hist(bins=100)
2 ax.get_figure().savefig("D:\imagens\distribuicao-salarios.svg")
```



Fonte: Elaboração própria do autor, 2020.

Temos uma noção dos dados com os quais estamos trabalhando, vamos partir efetivamente para a nossa estrutura `map`.

Uma etapa importante na preparação dos dados, para a maioria dos algoritmos de IA, é escalar os dados. Como os valores de salários variam muito, é importante calculá-los e torná-los mais centralizados. Uma forma eficiente de fazer isso é deslocando-os da média e dividindo-os pelo valor máximo, isto é: se o valor médio do conjunto é , o valor máximo e o mínimo , fazemos a seguinte transformação em cada elemento :

Com essa transformação, os algoritmos tendem a resultados melhores, a transformações de escala nos dados fica por conta da disciplina de *Matemática para Inteligência Artificial*. Continuando, escreveremos a função que faz essa transformação:

```
def escala_salario(salario, media, minimo, maximo):
 return (salario-media)/(maximo-minimo)
```

Para usar essa função, são necessários, ainda, outros parâmetros além do valor, são eles: média, mínimo e máximo. A média já temos, vamos agora obter o mínimo e o máximo:

```
salario_menor = df.total_salary.min()
salario_menor
```

```
1699.19
```

Esse é o menor valor, o mesmo que obtivemos quando ordenamos o conjunto de dados de forma ascendente.

```
salario_maior = df.total_salary.max()
salario_maior
```

```
10568.36
```

Já esse é o maior valor, o mesmo que obtivemos ordenando o conjunto de forma descendente.

Temos tudo que precisamos para nossa função mapper. Ela atua isoladamente em um valor de salário e, por isso, podemos executar a transformação de maneira sequencial ou paralela.

A maneira sequencial você já conheceu ao longo do curso.

```
salarios_escalados_sequencias = [escala_salario(salario, salario_medio, salario_menor, salario_maior)
 for salario in df.total_salary]
```

Como o conjunto é grande, vamos ver apenas alguns:

```
salarios_escalados_sequencias[:20]
```

```
[-0.21984176647871223,
 -0.18188782039356557,
 0.34432601923291584,
 0.7542087929310184,
 -0.035772682223928494,
 0.03087391492101285,
 -0.21556402684805906,
 -0.24579120706898164,
 -0.13255851449459197,
 -0.09799270957710812]
```

Perceba que agora temos valores entre `0` e `1`, ou seja, nossa transformação funcionou. No entanto, poderíamos fazer o mesmo de forma paralela; além disso, é possível usar os vários núcleos do computador como forma de estudar o paralelismo que posteriormente pode ser feito em clusters ou supercomputadores.

Vamos importar mais algumas bibliotecas, essas são referentes ao processamento em paralelo:

```
import multiprocessing as mp
```

Nesse sentido, a biblioteca `multiprocessing` nos permite executar partes do código em paralelo. Vamos focar aqui apenas no método `map`. Para executar o método, é preciso criar um `pool` de módulos de execução, no caso um `pool` de processadores. Para usar todos os processadores da máquina, é possível fazê-lo de forma muito simples com a classe `Pool` do pacote mp:

```
pool = mp.Pool()
```

Pronto, o `pool` de processadores está usando todos os núcleos disponíveis em sua máquina. Se está executando esse código em um sistema como mesos, pode ofertar todos os processadores do cluster para uma tarefa complexa. Vamos aplicar a transformação usando o método `map` do objeto `pool`.

O primeiro argumento é a função a ser executada em cada elemento, a função `mapper`, o segundo, a lista na qual ele deve atuar:

```
salarios_escalados_paralelo = pool.map(lambda salario: escala_salario(salario,
 salario_medio, salario_menor, salario_maior),
 df.total_salary)
salarios_escalados_paralelo
```

```
---------------------------------------------------------------------------
PicklingError                             Traceback (most recent call last)
<ipython-input-329-1f691ef7c10d> in <module>
      1 salarios_escalados_paralelo = pool.map(lambda salario: escala_salario(salario,
      2  salario_medio, salario_menor, salario_maior),
----> 3  df.total_salary)
      4 salarios_escalados_paralelo

D:\Anaconda3\lib\multiprocessing\pool.py in map(self, func, iterable, chunksize)
    266         in a list that is returned.
    267         '''
--> 268         return self._map_async(func, iterable, mapstar, chunksize).get()
    269
    270     def starmap(self, func, iterable, chunksize=None):

D:\Anaconda3\lib\multiprocessing\pool.py in get(self, timeout)
    655             return self._value
    656         else:
--> 657             raise self._value
    658
    659     def _set(self, i, obj):

D:\Anaconda3\lib\multiprocessing\pool.py in _handle_tasks(taskqueue, put, outqueue, pool, cache)
    429                         break
    430                     try:
--> 431                         put(task)
    432                     except Exception as e:
    433                         job, idx = task[:2]


D:\Anaconda3\lib\multiprocessing\connection.py in send(self, obj)
    204         self._check_closed()
    205         self._check_writable()
--> 206         self._send_bytes(_ForkingPickler.dumps(obj))
    207
    208     def recv_bytes(self, maxlength=None):

D:\Anaconda3\lib\multiprocessing\reduction.py in dumps(cls, obj, protocol)
     49     def dumps(cls, obj, protocol=None):
     50         buf = io.BytesIO()
---> 51         cls(buf, protocol).dump(obj)
     52         return buf.getbuffer()
     53

PicklingError: Can't pickle <function <lambda> at 0x000002642334A318>: attribute lookup <lambda> on __main__ failed
```

Sim, você recebeu um erro. Pode parecer surpreendente, mas, acredite, errar e entender o cenário é a melhor forma de ser convencido.

O erro em questão é `Can't pickle <function at 0x7fa2998999d8>: attribute lookup`. Isso quer dizer que a função anônima que passamos para o map:

```
1 lambda salario: escala_salario(salario, media, minimo, maximo)
```

```
<function __main__.<lambda>(salario)>
```

Ou seja, a função lambda não pode ser distribuída entre os processadores. O motivo disso é simples: ela usa variáveis globais! A variável salário é o argumento de lambda, até aí tudo bem, mas as variáveis `salario_medio`, `minimo` e `maximo` são conhecidas do código como um todo, de forma que isso é um problema quando enviamos para unidades de processamento que não as conhecem. Em outras palavras, a função `mapper` (no nosso caso a função lambda) precisa ter acesso a todas as variáveis necessárias na lista.

Uma forma de resolver esse problema é com uma função parcial. Uma função `partial` é aquela em que já definimos alguns dos parâmetros, no nosso caso, vamos usar a função `escala_salario (salario, salario_medio, minimo, maximo)` e definir os parâmetros `salario_medio, minimo e maximo`.

Comecemos importando a função `partial` do pacote `functools`, que nos ajudará com essa missão:

```
1 from functools import partial
```

Criamos, neste momento, uma nova função de escala que convenientemente vamos chamar de parcial. A criação é simples:

```
1 parcial = partial(escala_salario, media = salario_medio, minimo=salario_menor, maximo=salario_maior)
```

Agora que definimos três dos quatro parâmetros, temos uma função de um único parâmetro, vamos usá-la na paralelização do nosso `pool`.

```
1 salarios_escalados_paralelo = pool.map(parcial, df.total_salary)
```

Dessa vez, não tivemos erros. Vamos ver os primeiros 20 como fizemos com os resultados sequenciais:

```
[-0.01804080282025261,
 -0.014613373858304789,
 0.0329063372691607,
 0.0699207780449558,
 -0.001418454541664603,
 0.004600062656978837,
 -0.017654501784020952,
 -0.020384165590118603,
 -0.010158693669540633,
 -0.007037230604766277,
 -0.00435267258650485,
 -0.026307380266381224,
 0.07454997588688313,
 -0.012749374630499584,
 -0.0030591648468403882,
 -0.02448258132823053,
 -0.016237080401474594,
 -0.02069461152039544,
 -0.017909252757308144,
 -0.0024335893153150058]
```

Agora que você já conhece a ideia do map, vamos ao *reduce*.

Se o `map` tem por objetivo transformar um conjunto de dados e devolver cada elemento transformado individualmente, o `reducer` tem a função de misturar todos os elementos

em alguma função especial. Na linguagem tensorial, o reducer tem como objetivo eliminar uma das ordens do tensor agregando os elementos dela.

Vamos começar com um exemplo, o primeiro passo é importar a função *reduce* que, assim como a *partial*, também faz parte do pacote `functools`:

```
from functools import reduce
```

A título de curiosidade, existe também uma função `map` que faz o mesmo que a compreensão de lista, diferendo-se por devolver um gerador, também conhecido como *generator*. Um gerador é bem parecido com a lista, mas de forma "preguiçosa" (o termo "técnico" é `lazy`). Um `generator` sabe a lista que usará como base para a transformação e sabe como fazer isso, mas só fará o cálculo do elemento `n` quando ele for solicitado. Você é encorajado a substituir a compreensão de lista pela função `map` e trabalhar com o `generator`, em muitas vezes isso é útil. Discutiremos mais sobre `generators` no material complementar.

Voltando à função `reduce`, vamos começar conhecendo a estrutura dela. Uma função `reduce` tem de receber dois valores, o valor total, que é o resultado parcial do `reduce`, e o valor atual. Podemos especificar o valor inicial; mas, se não for dito, o `reduce` usará os dois primeiros valores. Por exemplo, uma função que somará todos os valores de uma lista:

```
reduce(lambda total, atual: total + atual, [1, 2, 3, 4, 5, 6])
```

```
21
```

É possível fazer algo mais interessante: como já foi dito, algumas vezes é viável reduzir uma ordem do tensor, um dos cenários em que isso é mais importante é em um processo chamado de `planificação` de uma lista de listas. Para diversos procedimentos em IA, não usamos matrizes como entradas, preferimos empregar vetores por motivos que não cabe discutir aqui. Então, transformamos matrizes em vetores. Vamos começar com uma matriz simples:

```
m = [[0, 1, 3],
 [6, 9, 1],
 [9, 1, 4],
 [3, 2, 7]]
m
```

```
[[0, 1, 3], [6, 9, 1], [9, 1, 4], [3, 2, 7]]
```

Vamos transformar essa matriz em um vetor usando um *reducer*. Primeiro passo, criaremos a função que concatena listas:

```
def concatena_lista(lista1, lista2):
 return lista1+lista2
```

Agora, aplicamos esse resultado na matriz `m` usando a função *reduce*:

```
1 reduce(concatena_lista, m)
```

```
[0, 1, 3, 6, 9, 1, 9, 1, 4, 3, 2, 7]
```

Temos, nesse sentido, uma lista planificada. O que o `reduce` fez foi pegar a primeira e a segunda listas e concatenar, a seguir, o resultado à próxima lista, assim por diante até toda a lista ser percorrida.

Outro exemplo: vamos calcular o maior salário do conjunto usando uma função reducer. A ideia é receber dois valores e dizer qual deles é o maior, o maior valor fará o papel do total, e então será comparado a todos os demais valores.

Vamos primeiro à função que retorna o maior valor:

```
1 def retorna_maior_salario(salario1, salario2):
2   if salario1 > salario2:
3      return salario1
4   return salario2
```

Agora aplicamos a função:

```
1 reduce(retorna_maior_salario, df.total_salary)
```

```
10568.36
```

Sobre o `reduce` não há mais o que falar neste momento, tivemos o mesmo resultado, como era de se esperar. No material complementar, vamos conhecer alguns casos reais de `map reduce`, o importante aqui é fundamentar o conceito.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. **Introduction to Algorithms**. 3. ed. Massachusetts, MA: MIT Press, 1.292 p. 2009.

ELKNER, J.; DOWNEY, A.; MEYERS, C. **How to think like a computer scientist**: learning with Python. Wickford, UK: Samurai Media Limited, 306 p. 2016.

GRUS, J. **Data science from scratch**: first principles with python. Sebastopol, CA: O'Reilly, 330 p. 2015.

HERSTEIN, N.; WINTER, D. J. **Matrix Theory and Linear Algebra**. USA: Macmillan Pub. Co., 508 p. 1988.

JOLLY, K. **Hands-on data visualization with Bokeh**: interactive web plotting for Python using Bokeh. Packt Publishing Ltd, 174 p. 2018.

MADHAVAN, S. **Mastering Python for Data Science**. USA: Packt Publishing, 294 p. 2015.

MÜLLER, A. C.; GUIDO, S. **Introduction to machine learning with Python**: a guide for data scientists. 1. ed. O'Reilly Media, 392 p. 2016.

SEDGEWICK, R.; WAYNE, K. **The textbook Algorithms**. 4, USA: Ed. Addison-Wesley Professional, 992 p. 2011.

TATSUOKA, Maurice M.; LOHNES, Paul R. **Multivariate analysis:** Techniques for educational and psychological research. USA: Macmillan Publishing Co. Inc, 479 p. 1988.