

```
import numpy as np
```

2021

```
class Vetor:
```

```
    def __init__(self, tamanho):
```

```
        self.tamanho = tamanho
```

```
        self.ultimo = -1
```

```
        self.elementos = np.empty(self.tamanho, dtype = float)
```

```
    def exibe_em_tela(self):
```

```
        if self.ultimo == -1:
```

```
            print("Vetor vazio!")
```

```
        else:
```

```
            for i in range(self.ultimo + 1):
```

```
                print(i, self.elementos[i])
```

```
    def insere_elemento(self, elemento):
```

```
        if self.ultimo == self.tamanho - 1:
```

```
            print("Capacidade máxima atingida")
```

```
        else:
```

```
            self.ultimo += 1
```

```
            self.elementos[self.ultimo] = elemento
```

```
    def pesquisa_elemento(self, elemento):
```

```
        for i in range(self.ultimo + 1):
```

```
            if elemento == self.elementos[i]:
```

```
                return i
```

```
        return -1
```

```
    def exclui_elemento(self, elemento):
```

```
        posicao = self.pesquisa_elemento(elemento)
```

```
        if posicao == -1:
```

```
            return
```

```
        else:
```

```
            for i in range(posicao, self.ultimo):
```

```
                self.elementos[i] = self.elementos[i + 1]
```

```
            self.ultimo -= 1
```

ANÁLISE EXPLORATÓRIA DE DADOS COM PYTHON, PANDAS E NUMPY

FERNANDO FELTRIN


```
import numpy as np
```

```
class Vetor:
```

2021

```
    def __init__(self, tamanho):
```

```
        self.tamanho = tamanho
```

```
        self.ultimo = -1
```

```
        self.elementos = np.empty(self.tamanho, dtype = float)
```

```
    def exibe_em_tela(self):
```

```
        if self.ultimo == -1:
```

```
            print('Vetor vazio!!!')
```

```
        else:
```

```
            for i in range(self.ultimo + 1):
```

```
                print(i, self.elementos[i])
```

```
    def insere_elemento(self, elemento):
```

```
        if self.ultimo == self.tamanho - 1:
```

```
            print('Capacidade máxima atingida')
```

```
        else:
```

```
            self.ultimo += 1
```

```
            self.elementos[self.ultimo] = elemento
```

```
    def pesquisa_elemento(self, elemento):
```

```
        for i in range(self.ultimo + 1):
```

```
            if elemento == self.elementos[i]:
```

```
                return i
```

```
        return -1
```

```
    def exclui_elemento(self, elemento):
```

```
        posicao = self.pesquisa_elemento(elemento)
```

```
        if posicao == -1:
```

```
            print('Elemento não encontrado')
```

```
        else:
```

```
            for i in range(posicao, self.ultimo):
```

```
                self.elementos[i] = self.elementos[i + 1]
```

```
            self.ultimo -= 1
```

ANÁLISE EXPLORATÓRIA DE DADOS COM PYTHON, PANDAS E NUMPY

FERNANDO FELTRIN

ANÁLISE EXPLORATÓRIA DE DADOS COM PYTHON, PANDAS E NUMPY

Fernando Feltrin

AVISOS

Este livro é um compilado de:



Todo conteúdo foi reordenado de forma a não haver consideráveis divisões dos conteúdos entre um livro e outro, cada tópico está rearranjado em posição contínua, formando assim um único livro. Quando necessário, alguns capítulos sofreram pequenas alterações a fim de deixar o conteúdo melhor contextualizado.

Este livro conta com mecanismo antipirataria Amazon Kindle Protect DRM. Cada cópia possui um identificador próprio rastreável, a distribuição ilegal deste conteúdo resultará nas medidas legais cabíveis.

É permitido o uso de trechos do conteúdo para uso como fonte desde que dados os devidos créditos ao autor.

Análise Exploratória de Dados com Python, Pandas e Numpy

v1.03

SOBRE O AUTOR



Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 10 livros sobre programação de computadores e responsável pelo desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia (diagnóstico por imagem).

LIVROS



Disponível em: [Amazon.com.br](https://www.amazon.com.br)

CURSO

Desenvolvimento > Linguagens de programação > Python

Python do ZERO à Programação Orientada a Objetos

Aprenda programação em Python de forma rápida e efetiva.


Mais bem cotados 4,7 ★★★★★ (79 classificações) 1.445 alunos

Criado por [Fernando Belomé Feltrin](#)

Última atualização em 10/2020 Português Português [Automático]

[Lista de Favoritos](#) [Compartilhar](#) [Presentear este curso](#)


Fernando Belomé Feltrin
Professor



★ 4,7 Classificação do instrutor
👤 79 Avaliações
👥 1.445 Alunos
🎓 1 Cursos

4.7
★★★★★
Classificação do Curso

★★★★★	68%
★★★★☆	25%
★★★☆☆	5%
★★☆☆☆	1%
★☆☆☆☆	1%



Python do ZERO à Programação Orientada a Objetos
Aprenda programação em Python de forma rápida e efetiva.
Fernando Belomé Feltrin
4,7 ★★★★★ (79)
15,5 horas no total • 340 aulas • Iniciante
[Classificação mais alta](#)

Pré-visualizar este curso

R\$ [redacted] R\$ [redacted]
38% de desconto
🕒 Só mais 5 horas por este preço!

[Adicionar ao carrinho](#)

[Comprar agora](#)

Garantia de devolução do dinheiro em 30 dias

Este curso inclui:

- 📺 15,5 horas de vídeo sob demanda
- 📄 2 artigos
- ∞ Acesso total vitalício
- 📱 Acesso no dispositivo móvel e na TV
- 📜 Certificado de Conclusão

[Aplicar cupom](#)

[Curso Python do ZERO à Programação Orientada a Objetos](#)

Mais de 15 horas de videoaulas que lhe ensinarão programação em linguagem Python de forma simples, prática e objetiva.

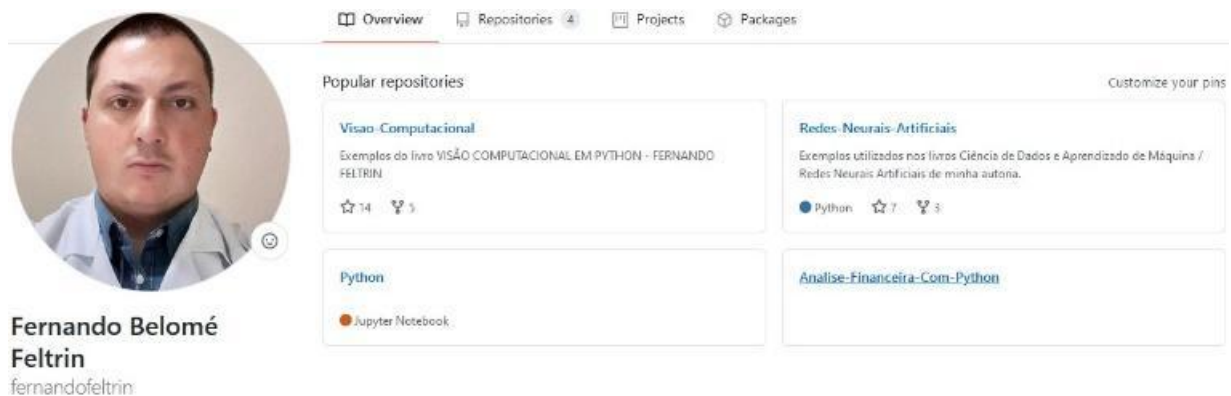
REDES SOCIAIS



 **Prof. Fernando Feltrin - Python**
@fernandofeltrinpython · Site educacional

[WhatsApp](#)

<https://github.com/fernandofeltrin>



Fernando Belomé Feltrin
fernandofeltrin

Overview Repositories (4) Projects Packages

Popular repositories

- Visao-Computacional**
Exemplos do livro VISÃO COMPUTACIONAL EM PYTHON - FERNANDO FELTRIN
☆ 14 🍴 5
- Redes-Neurais-Artificiais**
Exemplos utilizados nos livros Ciência de Dados e Aprendizado de Máquina / Redes Neurais Artificiais de minha autoria.
Python ☆ 7 🍴 3
- Python**
Jupyter Notebook
- Analise-Financeira-Com-Python**

<https://github.com/fernandofeltrin>

ÍNDICE

[CAPA](#)

[AVISOS](#)

[SOBRE O AUTOR](#)

[LIVROS](#)

[CURSO](#)

[REDES SOCIAIS](#)

[ÍNDICE](#)

[INTRODUÇÃO](#)

[BIBLIOTECA NUMPY](#)

[Sobre a biblioteca Numpy.](#)

[Instalação e importação das dependências](#)

[ARRAYS](#)

[Criando uma array numpy.](#)

[Criando uma array gerada com números ordenados](#)

[Array gerada com números do tipo float](#)

[Array gerada com números do tipo int](#)

[Array gerada com números zero](#)

[Array gerada com números um](#)

[Array gerada com espaços vazios](#)

[Criando uma array de números aleatórios, mas com tamanho predefinido em variável](#)

[Criando arrays de dimensões específicas](#)

[Array unidimensional](#)

[Array bidimensional](#)

[Array tridimensional](#)

[Verificando o tamanho e formato de uma array.](#)

[Verificando o tamanho em bytes de um item e de toda a array.](#)

[Verificando o elemento de maior valor de uma array.](#)

[Consultando um elemento por meio de seu índice](#)

[Consultando elementos dentro de um intervalo](#)

[Modificando manualmente um dado/valor de um elemento por meio de seu índice.](#)

[Criando uma array com números igualmente distribuídos](#)

[Redefinindo o formato de uma array.](#)

[Usando de operadores lógicos em arrays](#)

[OPERAÇÕES MATEMÁTICAS EM ARRAYS](#)

[Criando uma matriz diagonal](#)

[Criando padrões duplicados](#)

[Somando um valor a cada elemento da array.](#)

[Realizando soma de arrays](#)

[Subtração entre arrays](#)

[Multiplicação e divisão entre arrays](#)

[OPERAÇÕES LÓGICAS EM ARRAYS](#)

[Transposição de arrays](#)

[Salvando uma array no disco local](#)

[Carregando uma array do disco local](#)

[Exemplo de aplicação da biblioteca Numpy.](#)

[BIBLIOTECA PANDAS](#)

[Sobre a biblioteca Pandas](#)

[Instalação das Dependências](#)

[Tipos de dados básicos Pandas](#)

[ANÁLISE EXPLORATÓRIA DE DADOS](#)

[Importando a biblioteca Pandas](#)

SERIES

[Criando uma Serie](#)

[Visualizando o cabeçalho](#)

[Definindo uma origem personalizada](#)

[Definindo um índice personalizado](#)

[Integrando uma Array Numpy em uma Serie](#)

[Qualquer estrutura de dados como elemento de uma Serie](#)

[Integrando funções com Series](#)

[Verificando o tamanho de uma Serie](#)

[Criando uma Serie a partir de um dicionário](#)

[Unindo elementos de duas Series](#)

DATAFRAMES

[Criando um DataFrame](#)

[Extraindo dados de uma coluna específica](#)

[Criando colunas manualmente](#)

[Removendo colunas manualmente](#)

[Ordenando elementos de uma coluna](#)

[Extraindo dados de uma linha específica](#)

[Extraindo dados de um elemento específico](#)

[Extraindo dados de múltiplos elementos](#)

[Buscando elementos via condicionais](#)

OPERAÇÕES MATEMÁTICAS EM DATAFRAMES

[Usando de funções embutidas do sistema](#)

[Aplicando uma operação matemática a todos elementos](#)

[Usando de funções matemáticas personalizadas](#)

ESTRUTURAS CONDICIONAIS APLICADAS A DATAFRAMES

[Aplicação de estruturas condicionais simples](#)

[Aplicação de estruturas condicionais compostas](#)

[Atualizando um DataFrame de acordo com uma condicional](#)

[INDEXAÇÃO DE DATAFRAMES](#)

[Manipulando o índice de um DataFrame](#)

[Índices multinível](#)

[ENTRADA DE DADOS](#)

[Importando dados de um arquivo para um DataFrame](#)

[TRATAMENTO DE DADOS](#)

[Tratando dados ausentes / faltantes](#)

[Agrupando dados de acordo com seu tipo](#)

[MÉTODOS APLICADOS](#)

[Alterando o formato de um DataFrame](#)

[Concatenando dados de dois DataFrames](#)

[Mesclando dados de dois DataFrames](#)

[Agregando dados de um DataFrame em outro DataFrame](#)

[Filtrando elementos únicos de uma Serie de um DataFrame](#)

[Processamento em paralelo de um Dataframe](#)

[CONSIDERAÇÕES FINAIS](#)

INTRODUÇÃO

BIBLIOTECA NUMPY



Quando estamos usando da linguagem Python para criar nossos códigos, é normal que usemos de uma série de recursos internos da própria linguagem, uma vez que como diz o jargão, Python vem com baterias inclusas, o que em outras palavras pode significar que Python mesmo em seu estado mais básico já nos oferece uma enorme gama de funcionalidades prontas para implementação.

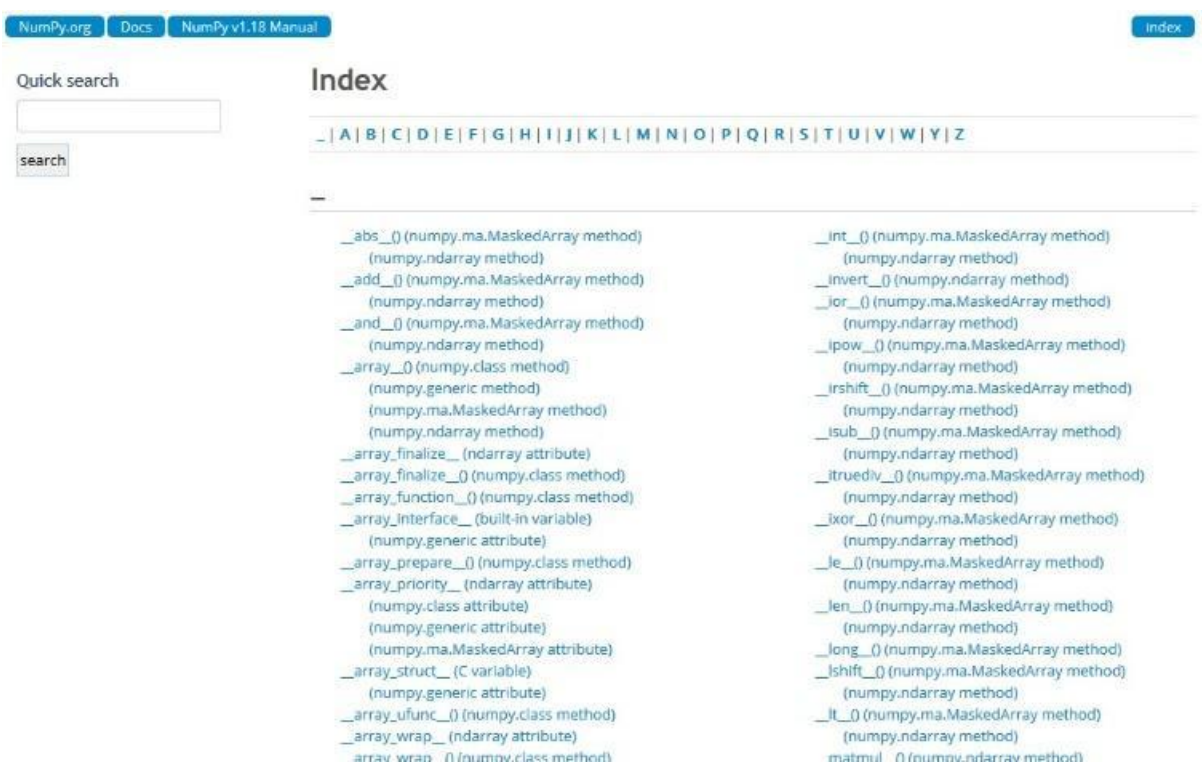
No âmbito de nichos específicos como, por exemplo, computação científica, é bastante comum o uso tanto das ferramentas nativas da linguagem Python como de bibliotecas desenvolvidas pela comunidade para propósitos bastante específicos.

Com uma rápida pesquisa no site <https://pypi.org> é possível ver que existem, literalmente, milhares de bibliotecas, módulos e pacotes desenvolvidos pela própria comunidade, a fim de adicionar novas funcionalidades ou aprimorar funcionalidades já existentes no núcleo da linguagem Python.

Uma das bibliotecas mais usadas, devido a sua facilidade de implementação e uso, é a biblioteca Numpy. Esta por sua vez oferece uma grande variedade de funções aritméticas de fácil aplicação, de modo que para certos tipos de operações as funções da biblioteca Numpy são muito mais intuitivas e eficientes do que as funções nativas de mesmo propósito presentes nas built-ins do Python.

Sobre a biblioteca Numpy

Como mencionado anteriormente, a biblioteca Numpy, uma das melhores, se não a melhor, quando falamos de bibliotecas dedicadas a operações aritméticas, possui em sua composição uma série de funções matemáticas pré-definidas, de modo que bastando importar a biblioteca em nosso código já temos à disposição as tais funções prontas para execução.



E quando digo uma enorme variedade de funções predefinidas, literalmente temos algumas centenas delas prontas para uso, bastando chamar a função específica e a parametrizar como é esperado. Por meio da documentação online da biblioteca é possível ver a lista de todas as funções.

Neste pequeno livro, iremos fazer uma abordagem simples das principais funcionalidades que a biblioteca Numpy nos oferece, haja visto que a aplicação de tais funções se dá de acordo com a necessidade, e em suas rotinas você notará que a grande maioria das funções disponíveis nem mesmo é usada, porém, todas elas estão sempre à disposição prontas para uso.

Também será visto nos capítulos finais um exemplo de funções da biblioteca Numpy sendo aplicadas em machine learning.

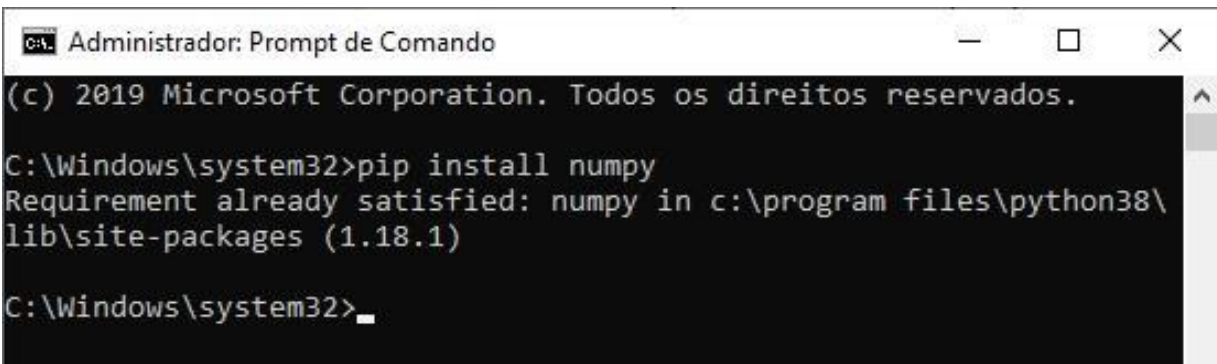
Instalação e importação das dependências

Usando do Google Colab, podemos realizar a instalação e atualização da biblioteca Numpy por meio do gerenciador de pacotes pip, de forma muito parecida como quando instalamos qualquer outra biblioteca localmente.

```
1 !pip install numpy
2
```

Uma vez que o processo de instalação tenha sido finalizado como esperado, sem erros, podemos imediatamente fazer o uso das ferramentas disponíveis da biblioteca Numpy.

O processo de instalação local também pode ser feito via Prompt de Comando, por meio de `pip install numpy`.

A screenshot of a Windows Command Prompt window titled "Administrador: Prompt de Comando". The window shows the command `pip install numpy` being executed. The output indicates that the requirement is already satisfied, as numpy version 1.18.1 is already installed in the local environment at `c:\program files\python38\lib\site-packages`. The prompt is currently at `C:\Windows\system32>`.

```
Administrador: Prompt de Comando
(c) 2019 Microsoft Corporation. Todos os direitos reservados.
C:\Windows\system32>pip install numpy
Requirement already satisfied: numpy in c:\program files\python38\
lib\site-packages (1.18.1)
C:\Windows\system32>
```

Para darmos início a nossos exemplos, já com nossa IDE aberta, primeiramente é necessário realizar a importação da biblioteca Numpy, uma vez que a mesma é uma biblioteca externa que por padrão não vem pré-carregada em nossa IDE.

O processo de importação é bastante simples, bastando criar a linha de código referente a importação da biblioteca logo no início de nosso código para que a biblioteca seja carregada prioritariamente antes das demais estruturas de código, de acordo com a leitura léxica de nosso interpretador.

```
1 import numpy as np
2
```

Uma prática comum é referenciar nossas bibliotecas por meio de alguma sigla, para que simplesmente fique mais fácil instanciar a mesma em nosso código. Neste caso, importamos a biblioteca numpy e a referenciamos como np apenas por convenção.

ARRAYS

Criando uma array numpy

Estando todas as dependências devidamente instaladas e carregadas, podemos finalmente dar início ao entendimento das estruturas de dados da biblioteca Numpy. Basicamente, já que estaremos trabalhando com dados por meio de uma biblioteca científica, é necessário contextualizarmos os tipos de dados envolvidos assim como as possíveis aplicações dos mesmos.

Sempre que estamos trabalhando com dados por meio da biblioteca Numpy, a primeira coisa que devemos ter em mente é que todo e qualquer tipo de dado em sua forma mais básica estará no formato de uma Array Numpy.

Quando estamos falando de arrays de forma geral basicamente estamos falando de dados em formato de vetores / matrizes, representados normalmente com dimensões bem definidas, alocando dados nos moldes de uma tabela com suas respectivas linhas, colunas e camadas.

A partir do momento que estamos falando de uma array “do tipo numpy”, basicamente estamos falando que tais dados estão carregados pela biblioteca Numpy de modo que possuem um sistema de indexação próprio da biblioteca e algumas métricas de mapeamento dos dados para que se possam ser aplicados sobre os mesmos uma série de funções, operações lógicas e aritméticas dependendo a necessidade.

```
1 data = np.array([2, 4, 6, 8, 10])
2
3 print(data)
4 print(type(data))
5 print(data.shape)
6
```

```
[ 2  4  6  8 10]
<class 'numpy.ndarray'>
(5,)
```

Inicialmente criamos uma variável / objeto de nome `data`, que recebe como atributo a função `np.array()` parametrizada com uma simples lista de caracteres. Note que instanciamos a biblioteca `numpy` por meio de `np`, em seguida chamando a função `np.array()` que serve para transformar qualquer tipo de dado em array do tipo `numpy`.

Uma vez criada a array `data` com seus respectivos dados/valores, podemos realizar algumas verificações simples.

Na primeira linha vemos o retorno da função `print()` simplesmente exibindo o conteúdo de `data`. Na segunda linha o retorno referente a função `print()` parametrizada para verificação do tipo de dado por meio da função `type()` por sua vez parametrizado com `data`, note que, como esperado, trata-se de uma `'numpy.ndarray'`.

Por fim, na terceira linha temos o retorno referente ao tamanho da array por meio do método `.shape`, nesse caso, 5 elementos em uma linha, o espaço vazio após a vírgula indica não haver nenhuma coluna adicional.

Criando uma array gerada com números ordenados

Dependendo do propósito, pode ser necessário a criação de uma array composta de números gerados ordenadamente dentro de um intervalo numérico.

Nesse contexto, a função `np.arange()` foi criada para este fim, bastando especificar o número de elementos que irão compor a array.

```
1 data = np.arange(15)
2
3 print(data)
4
```



```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

Nesse caso, quando estamos falando em números ordenados estamos falando em números inteiros gerados de forma sequencial, do zero em diante.

Novamente por meio da função `print()` parametrizada com `data`, é possível ver o retorno, nesse caso, uma array sequencial de 15 elementos gerados de forma ordenada.

Array gerada com números do tipo float

```
1 data = np.random.rand(15)
2
3 print(data)
4
```



```
[0.46767506 0.33941593 0.24069971 0.14360643 0.69268799 0.98486848
 0.13006424 0.27738541 0.90083945 0.69435152 0.00873169 0.238581
 0.80114636 0.94970789 0.37813807]
```

Outra possibilidade que temos é a de criar uma array numpy de dados aleatórios tipo float, escalonados entre 0 e 1, por meio da função `np.random.rand()`, novamente parametrizada com o número de elementos a serem gerados.

Da mesma forma, por meio da função `print()` é possível visualizar tais números gerados randomicamente. Note que neste caso os números são totalmente aleatórios, não sequenciais, dentro do intervalo de 0 a 1.

Array gerada com números do tipo int

```
1 data = np.random.randint(10, size = 10)
2 # números de 0 a 10, size define quantos elementos serão gerados
3
4 print(data)
5
```

```
[5 9 3 3 0 6 9 1 6 3]
```

Outro meio de criar uma array numpy é gerando números inteiros dentro de um intervalo específico definido manualmente para a função `np.random.randint()`.

Repare que o primeiro parâmetro da função, nesse caso, é o número 10, o que em outras palavras nos diz que estamos gerando números de 0 até 10, já o segundo parâmetro, `size`, aqui definido em 10, estipula que 10 elementos serão gerados aleatoriamente.

Mais uma vez por meio da função `print()` podemos visualizar a array gerada, como esperado, 10 números inteiros gerados aleatoriamente.

```
1 data = np.random.random((2, 2))
2
3 print(data)
4
```

```
[[0.3105231  0.46127506]
 [0.25691878 0.31584648]]
```

De forma parecida com o que fizemos anteriormente por meio da função `np.random.rand()`, podemos via `np.random.random()` gerar arrays de dados tipo float, no intervalo entre 0 e 1, aqui com mais de uma dimensão.

Note que para isso é necessário colocar o número de linhas e colunas dentro de um segundo par de parênteses.

Como de costume, por meio da função `print()` visualizamos os dados gerados e atribuídos a `data`.

Array gerada com números zero

```
1 data = np.zeros((3, 4))
2 # 3 linhas e 4 colunas
3
4 print(data)
5
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Dependendo da aplicação, pode ser necessário gerar arrays numpy previamente preenchidas com um tipo de dado específico. Em machine learning, por exemplo, existem várias aplicações onde se cria uma matriz inicialmente com valores zerados a serem substituídos após algumas funções serem executadas e gerarem retornos.

Aqui, por meio da função `np.zeros()` podemos criar uma array com as dimensões que quisermos, totalmente preenchida com números 0. Nesse caso, uma array com 3 linhas e 4 colunas deverá ser gerada.

Por meio da função `print()` novamente vemos o resultado, neste caso, uma matriz composta apenas de números 0.

Array gerada com números um

```
1 data = np.ones((3, 4))
2 # 3 linhas e 4 colunas
3
4 print(data)
5
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
```

Da mesma forma como realizado no exemplo anterior, de acordo com as particularidades de uma aplicação pode ser necessário realizar a criação de uma array numpy inicialmente preenchida com números 1. Apenas substituindo a função `np.zeros()` por `np.ones()` criamos uma array nos mesmos moldes da anterior.

Por meio da função `print()` visualizamos o resultado, uma matriz de números 1.

Array gerada com espaços vazios

```
1 data = np.empty((3, 4))
2
3 print(data)
4
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[2.85964880e-316 2.88902396e-316 3.95252517e-323 2.88902396e-316]
 [3.95252517e-323 2.88902396e-316 2.79437377e+276 0.00000000e+000]
 [2.88904926e-316 2.88904926e-316 2.88904926e-316 0.00000000e+000]]
```

Apenas como curiosidade, teoricamente é possível criar uma array numpy de dimensões pré definidas e com valores vazios por meio da função `np.empty()`.

Porém o retorno gerado, como você pode visualizar na imagem acima, é uma array com números em notação científica, representando algo muito próximo de zero, porém não zero absoluto. Este tipo de array normalmente é criado como uma estrutura esqueleto a ter seus dados /valores atualizados assim que tal array é instanciada por alguma variável ou alimentada como retorno de alguma função.

```
1 data = np.arange(10) * -1
2
3 print(data)
4
```

```
[ 0 -1 -2 -3 -4 -5 -6 -7 -8 -9]
```

Concluindo essa linha de raciocínio, novamente dependendo da aplicação, pode ser necessário a geração de uma array composta de números negativos. Nesse caso não há uma função específica para tal fim,

porém há o suporte da linguagem Python em permitir o uso do operador de multiplicação por um número negativo.

Por meio da função `print()`, como esperado, dessa vez temos uma array composta de 10 elementos negativos.

Criando uma array de números aleatórios, mas com tamanho predefinido em variável

Uma última possibilidade a ser explorada é a de associar a criação de uma array com um tamanho predefinido através de uma variável. Por meio da função `np.random.permutation()` podemos definir um tamanho que pode ser modificado conforme a necessidade,

```
1 tamanho = 10
2
3 data6 = np.random.permutation(tamanho)
4
5 print(data6)
6
```



```
[9 2 1 3 5 8 4 0 7 6]
```

Para esse fim, inicialmente criamos uma variável de nome `tamanho` que recebe como atributo o valor 10, este será o número a ser usado como referência de tamanho para a array, em outras palavras, o número de elementos da mesma.

Na sequência criamos nossa array `data6` que recebe a função `np.random.permutation()` por sua vez parametrizada com o valor de `tamanho`. Por meio da função `print()` podemos ver o resultado desse gerador.

O resultado, como esperado, é uma array unidimensional composta por 10 elementos aleatoriamente gerados.

Criando arrays de dimensões específicas

Uma vez que estamos trabalhando com dados em forma matricial (convertidos e indexados como arrays numpy), uma característica importante a observar sobre os mesmos é sua forma representada, seu número de dimensões.

Posteriormente estaremos entendendo como realizar diversas operações entre arrays, mas por hora, raciocine que para que seja possível o cruzamento de dados entre arrays, as mesmas devem possuir formato compatível. Nessa lógica, posteriormente veremos que para determinadas situações estaremos inclusive realizando a alteração do formato de nossas arrays para aplicação de certas funções.

Por hora, vamos entender da maneira correta como são definidas as dimensões de uma array numpy.

Array unidimensional

```
1 data = np.random.randint(5, size = 10)
2
3 print(data)
4 print(data.shape)
5
```



```
[4 4 1 2 3 3 1 2 3 0]
(10,)
```

Como já feito anteriormente, por meio da função `np.random.randint()` podemos gerar uma array, nesse caso, de 10 elementos com valores distribuídos entre 0 a 5.

Por meio da função `print()` parametrizada com `data` vemos os valores da array, e parametrizando a mesma com `data.shape` podemos inspecionar de forma simples o formato de nossa array. Neste caso, `[10,]` representa uma array com 10 elementos em apenas uma linha. Uma array unidimensional.

Array bidimensional

```
1 data2 = np.random.randint(5, size = (3, 4))
2
3 print(data2)
4 print(data2.shape)
5
```

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

Para o próximo exemplo criamos uma variável de nome `data2` que recebe, da mesma forma que o exemplo anterior, a função `np.random.randint()` atribuída para si, agora substituindo o valor de `size` por dois valores (3, 4), estamos definindo manualmente que o tamanho dessa array deverá ser de 3 linhas e 4 colunas, respectivamente. Uma array bidimensional.

Como esperado, por meio da função `print()` vemos a array em si com seu conteúdo, novamente via `data2.shape` vemos o formato esperado.

Array tridimensional

```
1 data3 = np.random.randint(5, size = (5, 3, 4))
2
3 print(data3)
4 print(data3.shape)
5
```



```
[[[0 2 4 4]
  [0 1 1 2]
  [1 2 2 0]]

 [[1 4 4 3]
  [0 4 3 3]
  [1 1 2 4]]

 [[2 1 1 2]
  [1 1 3 2]
  [3 0 2 4]]

 [[2 4 1 3]
  [0 0 3 4]
  [0 4 3 0]]

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

Seguindo com a mesma lógica, criamos a variável `data3` que chama a função `np.random.randint()`, alterando novamente o número definido para o parâmetro `size`, agora com 3 parâmetros, estamos definindo uma array de formato tridimensional.

Nesse caso, importante entender que o primeiro dos três parâmetros se refere ao número de camadas que a array terá em sua terceira dimensão, enquanto o segundo parâmetro define o número de linhas e o terceiro parâmetro o número de colunas.

Como esperado, 5 camadas, cada uma com 3 linhas e 4 colunas, preenchida com números gerados aleatoriamente entre 0 e 5.

```
1 data4 = np.array([[1, 2, 3, 4], [1, 3, 5, 7]])
2
3 print(data4)
4
```

```
[[1 2 3 4]
 [1 3 5 7]]
```

Entendidos os meios de como gerar arrays com valores aleatórios, podemos agora entender também que é perfeitamente possível criar arrays manualmente, usando de dados em formato de lista para a construção da mesma. Repare que seguindo uma lógica parecida com o que já foi visto anteriormente, aqui temos duas listas de números, conseqüentemente, teremos uma array bidimensional.

Exibindo em tela o resultado via função `print()` temos nossa array com os respectivos dados declarados manualmente. Uma última observação a se fazer é que é imprescindível que se faça o uso dos dados declarados da maneira correta, inclusive na sequência certa.

Verificando o tamanho e formato de uma array

Quando estamos trabalhando com dados em formato de vetor ou matriz, é muito importante eventualmente verificar o tamanho e formato dos mesmos, até porquê para ser possível realizar operações entre tais tipos de dados os mesmos devem ter um formato padronizado.

```
1 data3 = np.random.randint(5, size = (5, 3, 4))
2
3 print(data3.ndim)
4 print(data3.shape)
5 print(data3.size)
6
```



```
3
(5, 3, 4)
60
```

Indo diretamente ao ponto, nossa variável `data3` é uma array do tipo numpy como pode ser observado. A partir daí é possível realizar algumas verificações, sendo a primeira delas o método `.ndim`, que por sua vez retornará o número de dimensões presentes em nossa array. Da mesma forma `.shape` nos retorna o formato de nossa array (formato de cada dimensão da mesma) e por fim `.size` retornará o tamanho total dessa matriz.

Como retorno das funções `print()` criadas, logo na primeira linha temos o valor 3 referente a `.ndim`, que em outras palavras significa tratar-se de uma array tridimensional. Em seguida, na segunda linha temos os valores 5, 3, 4, ou seja, 5 camadas, cada uma com 3 linhas e 4 colunas. Por fim, na terceira linha temos o valor 60, que se refere a soma de todos os elementos que compõem essa array / matriz.

Verificando o tamanho em bytes de um item e de toda a array

Uma das questões mais honestas que todo estudante de Python (não somente Python, mas outras linguagens) possui é do porquê utilizar de bibliotecas externas quando já temos a disposição ferramentas nativas da linguagem.

Pois bem, toda linguagem de programação de código fonte aberto tende a receber forte contribuição da própria comunidade de usuários, que por sua vez, tentam implementar novos recursos ou aperfeiçoar os já existentes.

Com Python não é diferente, existe uma infinidade de bibliotecas para todo e qualquer tipo de aplicação, umas inclusive, como no caso da Numpy, mesclam implementar novas ferramentas assim como reestruturar ferramentas padrão em busca de maior performance.

Uma boa prática é manter em nosso código apenas o necessário, assim como levar em consideração o impacto de cada elemento no processamento de nossos códigos. Não é porque você vai criar um editor de texto que você precise compilar junto todo o sistema operacional. Inclusive se você já tem alguma familiaridade com a programação em Python sabe muito bem que sempre que possível importamos apenas os módulos e pacotes necessários de uma biblioteca, descartando todo o resto.

```
1 print(f'Cada elemento possui {data3.itemsize} bytes')
2 print(f'Toda a matriz possui {data3.nbytes} bytes')
3
```



```
Cada elemento possui 8 bytes
Toda a matriz possui 480 bytes
```

Dado o contexto acima, uma das verificações comumente realizada é a de consultar o tamanho de cada elemento que faz parte da array assim como o tamanho de toda a array. Isso é feito diretamente por meio dos métodos `.itemsize` e `.nbytes`, respectivamente,

Aqui, apenas para variar um pouco, já implementamos essas verificações diretamente como parâmetro em nossa função `print()`, porém é perfeitamente possível associar esses dados a uma variável.

Como esperado, junto de nossa mensagem declarada manualmente por meio de f'Strings temos os respectivos dados de cada elemento e de toda a matriz

Verificando o elemento de maior valor de uma array

Iniciando o entendimento das possíveis verificações que podemos realizar em nossas arrays, uma das mais comuns é buscar o elemento de maior valor em relação aos demais da array.

```
1 data = np.arange(15)
2
3 print(data)
4 print(data.max())
5
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
14
```

Criada a array data com 15 elementos ordenados, por meio da função `print()` parametrizando a mesma com os dados de data veremos tais elementos. Aproveitando o contexto, por meio da função `max()` teremos acesso ao elemento de maior valor associado.

Como esperado, na primeira linha temos todos os elementos de data, na segunda linha temos em destaque o número 14, lembrando que a array tem 15 elementos, mas os mesmos iniciam em 0, logo, temos números de 0 a 14, sendo 14 o maior deles.

Consultando um elemento por meio de seu índice

Como visto anteriormente, a organização e indexação dos elementos de uma array se assemelha muito com a estrutura de dados de uma lista, e como qualquer lista, é possível consultar um determinado elemento por meio de seu índice, mesmo que esse índice não seja explícito.

Inicialmente vamos ver alguns exemplos com base em array unidimensional.

```
1 data = np.arange(15)
2
3 print(data)
4 print(data[8])
5
```



```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
8
```

Criada a array data da mesma forma que o exemplo anterior, por meio da função `print()` parametrizada com `data[]` especificando um número de índice, é esperado que seja retornado o respectivo dado/valor situado naquela posição.

Se não houver nenhum erro de sintaxe, é retornado o dado/valor da posição 8 do índice de nossa array data. Nesse caso, o retorno é o próprio número 8 uma vez que temos dados ordenados nesta array.

```
1 data
2
3 print(data)
4 print(data[-5])
5

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
10
```

Da mesma forma, passando como número de índice um valor negativo, será exibido o respectivo elemento a contar do final para o começo dessa array.

Nesse caso, o 5º elemento a contar do fim para o começo é o número 10.

Partindo para uma array multidimensional, o processo para consultar um elemento é feito da mesma forma dos exemplos anteriores, bastando agora especificar a posição do elemento levando em conta a

```
1 data2 = np.random.randint(5, size = (3, 4))
2
3 print(data2)
4 print(data2[0, 2])
5

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

Note que é criada a array data2, de elementos aleatórios entre 0 e 5, tendo 3 linhas e 4 colunas em sua forma.

Parametrizando print com data2[0, 2] estamos pedindo que seja exibido o elemento situado na linha 0 (primeira linha) e coluna 2 (terceira coluna pois a primeira coluna é 0).

Podemos visualizar via console a array em si, e o respectivo elemento situado na primeira linha, terceira coluna. Neste caso, o número 4.

Consultando elementos dentro de um intervalo

```
1 data
2
3 print(data)
4
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

Reutilizando nossa array `data` criada anteriormente, uma simples array unidimensional de 15 elementos ordenados, podemos avançar com o entendimento de outras formas de consultar dados/valores, dessa vez, definindo intervalos específicos.

```
1 print(data)
2 print(data[:5])
3 print(data[3:])
4 print(data[4:8])
5 print(data[::2])
6 print(data[3::])
7
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[0 1 2 3 4]
[ 3  4  5  6  7  8  9 10 11 12 13 14]
[4 5 6 7]
[ 0  2  4  6  8 10 12 14]
[ 3  4  5  6  7  8  9 10 11 12 13 14]
```

Como mencionado anteriormente, a forma como podemos consultar elementos via índice é muito parecida (senão igual) a forma como realizávamos as devidas consultas em elementos de uma lista. Sendo assim, podemos usar das mesmas notações aqui.

Na primeira linha simplesmente por meio da função `print()` estamos exibindo em tela os dados contidos em `data`.

Na sequência, passando `data[:5]` (mesmo que `data[0:5]`) como parâmetro para nossa função `print()` estaremos exibindo os 5 primeiros elementos dessa array. Em seguida via `data[3:]` estaremos exibindo do terceiro elemento em diante todos os demais. Na sequência, via `data[4:8]` estaremos exibindo do quarto ao oitavo elemento.

Da mesma forma, porém com uma pequena diferença de notação, podemos por meio de `data[::2]` exibir de dois em dois elementos, todos os elementos. Por fim, via `data[3:]` estamos pulando os 3 primeiros elementos e exibindo todos os demais.

Como esperado, na primeira linha temos todos os dados da array, na segunda linha apenas os 5 primeiros, na terceira linha do terceiro elemento em diante, na quarta linha os elementos entre 4 e 8, na quinta linha os elementos pares e na sexta linha todos os dados exceto os três primeiros.

Modificando manualmente um dado/valor de um elemento por meio de seu índice.

Assim como em listas podíamos realizar modificações diretamente sobre seus elementos, bastando saber seu índice, aqui estaremos realizando o mesmo processo da mesma forma.

```
1 data2
2
3 print(data2)
4
5 data2[0, 0] = 10
6
7 print(data2)
8
```

```
[[2 3 0 3]
 [4 1 1 4]
 [1 3 0 1]]
[[10 3 0 3]
 [ 4 1 1 4]
 [ 1 3 0 1]]
```

Reutilizando nossa array data 2 criada anteriormente, podemos realizar a alteração de qualquer dado/valor desde que se faça a referência correta a sua posição na array de acordo com seu índice. Nesse caso, apenas como exemplo, iremos substituir o valor do elemento situado na linha 0 e coluna 0 por 10.

Repare que como esperado, o valor do elemento [0, 0] foi substituído de 1 para 10 como mostra o retorno gerado via print().

```

1 data2
2
3 data2[1, 1] = 6.82945
4
5 print(data2)
6
7

```

```

[[10  3  0  3]
 [ 4  6  1  4]
 [ 1  3  0  1]]

```

Apenas como curiosidade, atualizando o valor de um elemento com um número do tipo float, o mesmo será convertido para int para que não hajam erros por parte do interpretador.

```

[[10  3  2  3]
 [ 4  6  4  0]
 [ 0  2  4  0]]

```

Elemento [1, 1] em formato int, sem as casas decimais declaradas manualmente.

```

1 data2 = np.array([8, -3, 5, 9], dtype = 'float')
2
3 print(data2)
4 print(type(data2[0]))
5

```

```

[ 8. -3.  5.  9.]
<class 'numpy.float64'>

```

Supondo que você realmente precise dos dados em formato float dentro de uma array, você pode definir manualmente o tipo de dado por meio do parâmetro `dtype = 'float'`. Dessa forma, todos os elementos desta array serão convertidos para float.

Aqui como exemplo estamos criando novamente uma array unidimensional, com números inteiros em sua composição no momento da declaração.

Analisando o retorno de nossa função `print()` podemos ver que de fato são números, agora com casas decimais, de tipo `float64`.

Criando uma array com números igualmente distribuídos

Dependendo mais uma vez do contexto, pode ser necessária a criação de uma array com dados/valores distribuídos de maneira uniforme na mesma. Tenha em mente que os dados gerados de forma aleatória não possuem critérios definidos quanto à sua organização. Porém, por meio da função `linspace()` podemos criar dados uniformemente arranjados dentro de uma array.

```
1 data = np.linspace(0, 1, 7)
2
3 print(data)
4
```

```
[0.          0.16666667 0.33333333 0.5          0.66666667 0.83333333
1.          ]
```

Criamos uma nova array de nome `data`, onde por meio da função `np.linspace()` estamos gerando uma array composta de 7 elementos igualmente distribuídos, com valores entre 0 e 1.

O retorno como esperado é uma array de números float, para que haja divisão igual dos valores dos elementos.

Redefinindo o formato de uma array

Um dos recursos mais importantes que a biblioteca Numpy nos oferece é a de poder livremente alterar o formato de nossas arrays. Não que isso não seja possível sem o auxílio da biblioteca Numpy, mas a questão mais importante aqui é que no processo de reformatação de uma array numpy, a mesma se reajustará em todos os seus níveis, assim como atualizará seus índices para que não se percam dados entre dimensões.

```
1 data5 = np.arange(8)
2
3 print(data5)
4
```

```
[0 1 2 3 4 5 6 7]
```

Inicialmente apenas para o exemplo criamos nossa array data5 com 8 elementos gerados e ordenados por meio da função `np.arange()`.

```
1 data5 = data5.reshape(2, 4)
2
3 print(data5)
4
```

```
[[0 1 2 3]
 [4 5 6 7]]
```

Em seguida, por meio da função `reshape()` podemos alterar livremente o formato de nossa array. Repare que inicialmente geramos uma array unidimensional de 8 elementos, agora estamos transformando a mesma para uma array bidimensional de 8 elementos, onde os mesmos serão distribuídos em duas linhas e 4 colunas.

É importante salientar que você pode alterar livremente o formato de uma array desde que mantenha sua proporção de distribuição de dados.

Aqui, nesse exemplo bastante simples, 8 elementos dispostos em uma linha passarão a ser 8 elementos dispostos em duas linhas e 4 colunas. O número de elementos em si não pode ser alterado.

Na primeira linha, referente ao primeiro `print()` temos a array `data5` em sua forma inicial, na segunda linha o retorno obtido pós `reshape`.

Usando de operadores lógicos em arrays

Dependendo do contexto pode ser necessário fazer uso de operadores lógicos mesmo quando trabalhando com arrays numpy. Respeitando a sintaxe Python, podemos fazer uso de operadores lógicos sobre arrays da mesma forma como fazemos com qualquer outro tipo de dado.

```
1 data7 = np.arange(10)
2
3 print(data7)
4 print(data7 > 3)
5
```



```
[0 1 2 3 4 5 6 7 8 9]
[False False False False  True  True  True  True  True  True]
```

Para esse exemplo criamos nossa array data7 de 10 elementos ordenados gerados aleatoriamente. Por meio da função print() podemos tanto exibir seu conteúdo quanto realizar proposições baseadas em operadores lógicos. Note que no exemplo a seguir, estamos verificando quais elementos de data7 tem seu valor maior que 3.

Na primeira linha todo o conteúdo de data7, na segunda marcados como False os elementos menores que 3, assim como marcados como True os de valor maior que 3.

```
1 print(data7)
2 print(data7[4] > 5)
3
```



```
[0 1 2 3 4 5 6 7 8 9]
False
```

Da mesma forma como visto anteriormente, fazendo a seleção de um elemento por meio de seu índice, também podemos aplicar um operador lógico sobre o mesmo.

Nesse caso, o elemento da posição 4 do índice, o número 3, não é maior que 5, logo, o retorno sobre essa operação lógica é False.

OPERAÇÕES MATEMÁTICAS EM ARRAYS

Dentro das possibilidades de manipulação de dados em arrays do tipo numpy está a de realizar, como esperado, operações aritméticas entre tais dados. Porém recapitulando o básico da linguagem Python, quando estamos trabalhando com determinados tipos de dados temos de observar seu tipo e compatibilidade com outros tipos de dados, sendo em determinados casos necessário a realização da conversão entre tipos de dados.

Uma array numpy permite realizar determinadas operações que nos mesmos tipos de dados em formato não array numpy iriam gerar exceções ou erros de interpretação.

```
1 data8 = [1, 2, 3, 4, 5, 6, 7, 8]
2
3 print(data8 + 10)
4
```

```
-----
TypeError                                Traceback (most recent call
<ipython-input-21-76a4de19f583> in <module>()
      1 data8 = [1, 2, 3, 4, 5, 6, 7, 8]
      2
----> 3 print(data8 + 10)

TypeError: can only concatenate list (not "int") to list
```

Para entendermos melhor, vamos usar do seguinte exemplo, temos uma array data8 com uma série de elementos dispostos em formato de lista, o que pode passar despercebido aos olhos do usuário, subentendendo que para este tipo de dado é permitido realizar qualquer tipo de operação. Por meio da função print() parametrizada com data8 + 10, diferentemente do esperado, irá gerar um erro.

Repare que o interpretador lê os dados de data8 como uma simples lista, tentando por meio do operador + concatenar os dados com o valor 10 ao invés de realizar a soma.

```
1 data8 = [1, 2, 3, 4, 5, 6, 7, 8]
2
3 data8 = np.array(data8)
4
5 print(data8 + 10)
6

[11 12 13 14 15 16 17 18]
```

Realizando a conversão de data8 para uma array do tipo numpy, por meio da função `np.array()` parametrizada com a própria variável data8, agora é possível realizar a operação de soma dos dados de data8 com o valor definido 10.

Como retorno da função `print()`, na primeira linha temos os valores iniciais de data8, na segunda linha os valores após somar cada elemento ao número 10.

Criando uma matriz diagonal

Aqui mais um dos exemplos de propósito específico, em algumas aplicações de machine learning é necessário a criação de uma matriz diagonal, e a mesma pode ser feita por meio da função `np.diag()`.

```
1 data9 = np.diag((1, 2, 3, 4, 5, 6, 7, 8))
2
3 print(data9)
4
```

```
[[1 0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0]
 [0 0 3 0 0 0 0 0]
 [0 0 0 4 0 0 0 0]
 [0 0 0 0 5 0 0 0]
 [0 0 0 0 0 6 0 0]
 [0 0 0 0 0 0 7 0]
 [0 0 0 0 0 0 0 8]]
```

Para esse exemplo criamos a array `data9` que recebe como atributo uma matriz diagonal de 8 elementos definidos manualmente, gerada pela função `np.diag()`.

O retorno é uma matriz gerada com os elementos predefinidos compondo uma linha vertical, sendo todos os outros espaços preenchidos com 0.

Algo parecido pode ser feito através da função `np.eye()`, porém, nesse caso será gerada uma matriz diagonal com valores 0 e 1, de tamanho definido pelo parâmetro repassado para função.

```
1 data9 = np.eye(4)
2
3 print(data9)
4
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Neste caso, para a variável `data9` está sendo criada uma array diagonal de 4 linhas e colunas conforme a parametrização realizada em `np.eye()`.

O retorno é uma matriz diagonal, com números 1 dispostos na coluna diagonal

Criando padrões duplicados

Por meio da simples notação de “encapsular” uma array em um novo par de colchetes e aplicar a função `np.tile()` é possível realizar a duplicação/replicação da mesma quantas vezes for necessário.

```
1 data10 = np.tile(np.array([[9, 4], [3, 7]]), 4)
2
3 print(data10)
4
```

```
[[9 4 9 4 9 4 9 4]
 [3 7 3 7 3 7 3 7]]
```

Aqui criada a array `data10`, note que atribuído para a mesma está a função `np.tile()` parametrizada com uma array definida manualmente assim como um último parâmetro 4, referente ao número de vezes a serem replicados os dados/valores da array.

No retorno podemos observar os elementos declarados na array, em sua primeira linha 9 e 4, na segunda, 3 e 7, repetidos 4 vezes.

```
1 data10 = np.tile(np.array([[9, 4], [3, 7]]), (2,2))
2
3 print(data10)
4
```

```
[[9 4 9 4]
 [3 7 3 7]
 [9 4 9 4]
 [3 7 3 7]]
```

Mesmo exemplo anterior, agora parametrizado para replicação em linhas e colunas.

Note que, neste caso, a duplicação realizada de acordo com a parametrização ocorre sem sobrepor a ordem original dos elementos.

Somando um valor a cada elemento da array

Como visto anteriormente, a partir do momento que uma matriz passa a ser uma array do tipo numpy, pode-se perfeitamente aplicar sobre a mesma qualquer tipo de operador lógico ou aritmético. Dessa forma, é possível realizar tais operações, desde que leve em consideração que a operação realizada se aplicará a cada um dos elementos da array.

```
1 data11 = np.arange(0, 15)
2
3 print(data11)
4
5 data11 = np.arange(0, 15) + 1
6
7 print(data11)
8
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
```

Para esse exemplo criamos uma nova array de nome data11, por sua vez gerada com números ordenados. Em seguida é realizada uma simples operação de somar 1 a própria array, e como dito anteriormente, essa soma se aplicará a todos os elementos.

Observando os retornos das funções print() na primeira linha temos a primeira array com seus respectivos elementos, na segunda, a mesma array onde a cada elemento foi somado 1 ao seu valor original.

```
1 data11 = np.arange(0, 30, 3) + 3
2
3 print(data11)
4
```

```
[ 3  6  9 12 15 18 21 24 27 30]
```

Apenas como exemplo, usando da notação vista anteriormente é possível realizar a soma de um valor a cada elemento de uma array, nesse caso, somando 3 ao valor de cada elemento, de 3 em 3 elementos de acordo com o parâmetro passado para função `np.arange()`.

Como esperado, o retorno é uma array gerada com números de 0 a 30, com intervalo de 3 entre cada elemento.

Realizando soma de arrays

Entendido boa parte do processo lógico das possíveis operações em arrays não podemos nos esquecer que é permitido também a soma de arrays, desde que as mesmas tenham o mesmo formato ou número de elementos.

```
1  d1 = np.array([ 3,  6,  9, 12, 15])
2  d2 = np.array([ 1,  2,  3,  4,  5])
3
4  d3 = d1 + d2
5
6  print(d3)
7
```

[4 8 12 16 20]

Para esse exemplo criamos duas arrays d1 e d2, respectivamente, cada uma com 5 elementos distintos. Por meio de uma nova variável de nome d3 realizamos a soma simples entre d1 e d2, e como esperado, cada elemento de cada array será somado ao seu elemento equivalente da outra array.

O resultado obtido é a simples soma do primeiro elemento da primeira array com o primeiro elemento da segunda array, assim como todos os outros elementos com seu equivalente.

Subtração entre arrays

```
1 d1 = np.array([ 3,  6,  9, 12, 15])
2 d2 = np.array([ 1,  2,  3,  4,  5])
3
4 d3 = d1 - d2
5
6 print(d3)
7
```

[2 4 6 8 10]

Exatamente da mesma forma é possível realizar a subtração entre arrays.

Como esperado, é obtido os valores das subtrações entre cada elemento da array d1 com o seu respectivo elemento da array d2.

Multiplicação e divisão entre arrays

```
1  d1 = np.array([ 3,  6,  9, 12, 15])
2  d2 = np.array([ 1,  2,  3,  4,  5])
3
4  d3 = d1 / d2
5
6  d4 = d1 * d2
7
8  print(d3)
9  print(d4)
10
```

```
[3.  3.  3.  3.  3.]
[ 3 12 27 48 75]
```

Exatamente da mesma forma é possível realizar a divisão e a multiplicação entre os elementos das arrays.

Obtendo assim, na primeira linha o retorno da divisão entre os elementos das arrays, na segunda linha o retorno da multiplicação entre os elementos das mesmas.

OPERAÇÕES LÓGICAS EM ARRAYS

Da mesma forma que é possível realizar o uso de operadores aritméticos para seus devidos cálculos entre elementos de arrays, também é uma prática perfeitamente possível fazer o uso de operadores lógicos para verificar a equivalência de elementos de duas ou mais arrays.

```
1 d1 = np.array([ 3,  6,  9, 12, 15, 18, 21])
2 d2 = np.array([ 1,  2,  3,  4,  5, 20, 25])
3
4 d3 = d2 > d1
5
6 print(d3)
7
```

```
[False False False False False  True  True]
```

Neste exemplo, duas arrays d1 e d2 novamente, agora com algumas pequenas mudanças em alguns elementos, apenas para tornar nosso exemplo mais interessante.

Em nossa variável d3 estamos usando um operador para verificar se os elementos da array d2 são maiores que os elementos equivalentes em d1. Novamente, o retorno será gerado mostrando o resultado dessa operação lógica para cada elemento.

Repare que por exemplo o 4º elemento da array d2 se comparado com o 4º elemento da array d1 retornou False, pois 4 não é maior que 12.

Da mesma forma, o último elemento de d2 se comparado com o último elemento de d1 retornou True, pois obviamente 25 é maior que 22.

Transposição de arrays

Em determinadas aplicações, principalmente em machine learning, é realizada a chamada transposição de matrizes, que nada mais é do que realizar uma alteração de forma da mesma, transformando linhas em colunas e vice-versa por meio da função `transpose()`.

```
1  arr1 = np.array([[1, 2, 3], [4, 5, 6]])
2
3  print(arr1)
4  print(arr1.shape)
5
6  arr1_transposta = arr1.transpose()
7
8  print(arr1_transposta)
9  print(arr1_transposta.shape)
10
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
[[1 4]
 [2 5]
 [3 6]]
(3, 2)
```

Note que inicialmente é criada uma array de nome `arr1`, bidimensional, com elementos distribuídos em 2 linhas e 3 colunas.

Na sequência é criada uma nova variável de nome `arr1_transposta` que recebe o conteúdo de `arr1`, sob a função `transpose()`. Como sempre, por meio da função `print()` realizaremos as devidas verificações.

Como esperado, na primeira parte de nosso retorno temos a array `arr1` em sua forma original, com shape `(2, 3)` e logo abaixo a array `arr1_transposta`, que nada mais é do que o conteúdo de `arr1` reorganizados no que diz respeito às suas linhas e colunas, nesse caso com o shape `(3, 2)`.

Salvando uma array no disco local

```
1 data = np.array([ 3,  6,  9, 12, 15])
2 np.save('minha_array', data)
3
```

Encerrando nossos estudos, vamos ver como é possível salvar nossas arrays localmente para reutilização. Para isso, simplesmente criamos uma array comum atribuída a variável `data`.

Em seguida, por meio da função `np.save()` podemos de fato exportar os dados de nossa array para um arquivo local. Note que como parâmetros da função `np.save()` em primeiro lugar repassamos em forma de string um nome para nosso arquivo, seguido do nome da variável que possui a array atribuída para si.

O arquivo gerado será, nesse caso, `minha_array.npy`.

Carregando uma array do disco local

Da mesma forma que era possível salvar nossa array em um arquivo no disco local, outra possibilidade que temos é de justamente carregar este arquivo para sua utilização.

```
1 np.load('minha_array.npy')  
2
```

Através da função `np.load()`, bastando passar como parâmetro o nome do arquivo com sua extensão, o carregamento será feito, inclusive já alocando em memória a array para que se possam realizar operações sobre a mesma.

Exemplo de aplicação da biblioteca Numpy

```
1  import numpy as np
2
3  class Vetor:
4      def __init__(self, tamanho):
5          self.tamanho = tamanho
6          self.ultimo = -1
7          self.elementos = np.empty(self.tamanho, dtype = int)
8
9      def exibe_em_tela(self):
10         if self.ultimo == -1:
11             print('Vetor vazio!!!')
12         else:
13             for i in range(self.ultimo + 1):
14                 print(i, self.elementos[i])
15
16     def insere_elemento(self, elemento):
17         if self.ultimo == self.tamanho - 1:
18             print('Capacidade máxima atingida')
19         else:
20             self.ultimo += 1
21             self.elementos[self.ultimo] = elemento
22
23     def pesquisa_elemento(self, elemento):
24         for i in range(self.ultimo + 1):
25             if elemento == self.elementos[i]:
26                 return i
27         return -1
28
29     def exclui_elemento(self, elemento):
30         posicao = self.pesquisa_elemento(elemento)
31         if posicao == -1:
32             return -1
33         else:
34             for i in range(posicao, self.ultimo):
35                 self.elementos[i] = self.elementos[i + 1]
36             self.ultimo -= 1
37
```

Partindo para prática, vamos simular uma simples aplicação, toda implementada fazendo o uso de recursos da biblioteca Numpy, onde é criada uma estrutura de vetor não ordenado.

Inicialmente, é necessário sempre realizar as devidas importações das bibliotecas, módulos e pacotes que iremos utilizar ao longo de nosso código. Nesse caso, bastando importar a biblioteca numpy, por convenção a referenciando como np.

Em seguida é criada uma classe de nome Vetor, dentro de seu corpo/escopo, é criado um método construtor/inicializador `__init__()` que define o escopo dessa classe para seus objetos, e que por sua vez receberá obrigatoriamente um dado/valor a ser utilizado como referência para o tamanho fixo deste vetor.

Sendo assim, são criados alguns objetos de classe, onde o primeiro deles é `self.tamanho`, que recebe o valor atribuído a `tamanho`. Também é criado um outro objeto de classe de nome `self.ultimo`, que recebe como atributo o valor inicial -1, que por sua vez define um gatilho a ser acionado quando o vetor atingir sua capacidade máxima de elementos.

Por fim, é criado um último objeto de classe chamado `self.elementos`, que recebe como atributo inicial uma array do tipo numpy vazia, porém já com tamanho definido baseado no valor do objeto `tamanho`, assim como já é aproveitada a deixa para estabelecer o tipo de dado que irá obrigatoriamente compor este vetor.

Na sequência é criado um novo método de classe, dessa vez de nome `exibe_em_tela()`, que como o próprio nome já sugere, quando instanciado e inicializado, retornará a própria estrutura array desse vetor.

Para isso, inicialmente é criada uma estrutura condicional que verifica se o valor de `self.ultimo` for igual a -1, então exibe em tela via função `print()` a mensagem ‘Vetor vazio!!!’.

Caso essa condição não seja válida, por meio de um laço `for` é percorrido cada elemento de nosso vetor, a cada ciclo de repetição exibindo o mesmo assim como seu número de índice.

Dando sequência em nosso código, é criado um novo método de classe, dessa vez chamado `insere_elemento()`, que recebe como parâmetro um elemento.

Dentro do corpo desta função inicialmente é criada uma estrutura condicional onde se o valor atribuído a `self.ultimo` for igual ao valor de `self.tamanho - 1`, é exibido em tela a mensagem ‘Capacidade máxima atingida’.

Caso contrário, `self.ultimo` tem seu valor incrementado em 1 unidade, assim como para `self.elementos` na posição [`self.ultimo`] é inserido o novo elemento anteriormente repassado como parâmetro para nossa função `insere_elemento()`.

Na sequência é criado um novo método de classe, agora chamado `pesquisa_elemento()` que por sua vez receberá como parâmetro um elemento.

No bloco indentado a esta função temos um laço de repetição `for` que percorre cada um dos elementos de nosso vetor, validando se o valor atual de elemento é igual a `self.elementos` na posição [`i`], caso seja igual, é retornado o último valor atribuído a `i`, caso não seja igual, é retornado -1.

Por fim, é criado um último método de classe, dessa vez de nome `exclui_elemento()` que receberá por justaposição um elemento como parâmetro.

Dentro do corpo dessa função é criada uma variável local de nome `posição`, que por sua vez recebe como atributo os dados/valores oriundos da função aninhada `self.pesquisa_elemento(elemento)`.

Em seguida é criada uma estrutura condicional onde se o valor de `posição` for igual a -1, é retornado -1, caso contrário, é feito o uso de um laço de repetição que percorrerá todos os elementos de `posição` equiparados com `self.ultimo`.

A partir deste ponto, `self.elementos` na posição [`i`] tem seu valor atualizado com o último valor atribuído a `self.elementos` em sua posição [`i + 1`], finalizando com o decremento de `self.ultimo` em 1 unidade.

```
1 base = Vetor(10)
2 base.exibe_em_tela()
3
```

```
Vetor vazio!!!
```

De volta ao escopo global do código, é criada uma variável de nome `base` que por sua vez instancia e inicializa a classe `Vetor()`, repassando como argumento para a mesma o valor 10. Em outras palavras, aqui a variável `base` importa toda a estrutura interna da classe `Vetor`, assim como define um tamanho fixo de vetor em 10 elementos.

Usando do método `base.exibe_em_tela()` o retorno gerado neste momento é ‘Vetor vazio!!!’, uma vez que ainda não inserimos elementos no mesmo.

```
1 base.inserir_elemento(9)
2 base.inserir_elemento(3)
3 base.exibe_em_tela()
4
```

```
0 9
1 3
```

Usando do método `insere_elemento()` atrelado a nossa variável `base`, podemos inserir alguns elementos em nosso vetor. Novamente, usando do método `exibe_em_tela()`, nos é retornado o vetor representado por seus elementos.

```
1 print(base.pesquisa_elemento(9))
2
```

```
0
```

Realizando outro tipo de interação com nosso vetor, podemos usar de nosso método `pesquisa_elemento()` parametrizado com o elemento em si

para descobrir sua posição de índice no vetor.

```
1 print(base.ultimo)
2
1
```

Também é possível pesquisar diretamente o último valor atribuído para o objeto ultimo, usando desse retorno, que será um número de índice, para descobrirmos quantos elementos compõem nosso vetor.

```
1 base.insere_elemento(1)
2 base.insere_elemento(4)
3 base.insere_elemento(5)
4 base.insere_elemento(12)
5 base.insere_elemento(20)
6 base.exibe_em_tela()
7
0 9
1 3
2 1
3 4
4 5
5 12
6 20
```

Uma vez que nosso vetor esteja definido, assim como para o mesmo não exista nenhum conflito de interação ou até mesmo de sintaxe, podemos manipular este vetor à vontade.

Inserindo alguns elementos via método `insere_elemento()`, podemos visualizar nosso vetor instanciando e executando o método `exibe_em_tela()` sempre que necessário.

```
1 base.exclui_elemento(12)
2 base.exibe_em_tela()
3
```

```
0 9
1 3
2 1
3 4
4 5
5 20
```

Por fim, testando nosso método `exclui_elemento()` será possível ver que quando excluímos um determinado elemento do vetor, os elementos subsequentes serão realocados automaticamente para que as posições de índice se mantenham íntegras.

BIBLIOTECA PANDAS



Sobre a biblioteca Pandas

A biblioteca Pandas, segundo sua própria documentação (disponível em pandas.pydata.org) é uma biblioteca especificamente criada para análise e manipulação de dados, desenvolvida e integrada totalmente em Python.

Seu projeto base tem início em 2008 pela AQR Capital Management, se tornando open source a partir de 2009, recebendo ao longo dos anos uma série de melhorias feitas tanto pela comunidade quanto por sua mantenedora NumFOCUS, que custeia o projeto desde 2015.

A biblioteca Pandas visa implementar ao Python uma série de ferramentas integradas para o tratamento de dados organizados a partir de tabelas ou bases de dados onde os mesmos estão distribuídos em formato de linhas e colunas mapeadas e indexadas em um padrão.



The screenshot shows a Jupyter Notebook interface with a window titled "data - DataFrame". It displays a Pandas DataFrame with 11 columns and 10 rows of data. The columns are: Index, radius_mean, texture_mean, perimeter_mean, area_mean, compactness, compactness, concavity_mean, vein_points, symmetry_mean, and dimension. The rows are indexed from 0 to 9. The data is presented in a table with alternating red and purple background colors for the rows.

Index	radius_mean	texture_mean	perimeter_mean	area_mean	compactness	compactness	concavity_mean	vein_points	symmetry_mean	dimension
0	17.99	10.38	122.8	1001	0.1184	0.2776	0.3001	0.1471	0.2419	0.07
1	20.57	17.77	132.9	1326	0.08474	0.07864	0.0869	0.07017	0.1812	0.05
2	19.69	21.25	130	1203	0.1096	0.1599	0.1974	0.1279	0.2069	0.05
3	11.42	20.38	77.58	386.1	0.1425	0.2839	0.2414	0.1052	0.2597	0.09
4	20.29	14.34	135.1	1297	0.1003	0.1328	198	0.1043	0.1809	0.05
5	12.45	15.7	82.57	477.1	0.1278	0.17	0.1578	0.08089	0.2087	0.07
6	18.25	19.98	119.6	1040	0.09463	109	0.1127	74	0.1794	0.05
7	13.71	20.83	90.2	577.9	0.1189	0.1645	0.09366	0.05985	0.2196	0.07
8	13	21.82	87.5	519.8	0.1273	0.1932	0.1859	0.09353	235	0.07
9	12.46	24.04	83.97	475.9	0.1186	0.2396	0.2273	0.08543	203	0.08

A integração da biblioteca Pandas com o núcleo da linguagem Python permite realizar de forma fácil, rápida e flexível a análise e manipulação de dados sem que para isso seja necessário o uso de ferramentas externas, tornando assim tal biblioteca uma das melhores (se não a melhor) no quesito de tratamento de dados.

Instalação das Dependências

Para realizar a instalação da biblioteca Pandas no ambiente virtualizado de seu sistema é muito simples, bastando a partir de algum terminal, via gerenciador de pacotes pip (ou conda dependendo de seu ambiente de trabalho) executar o código `pip install pandas`.

```
Terminal: Local × +
Microsoft Windows [versão 10.0.19042.1052]
(c) Microsoft Corporation. Todos os direitos reservados.

(venv) C:\Users\Fernando\PycharmProjects\Pandas>pip install pandas
Collecting pandas
  Downloading pandas-1.2.5-cp39-cp39-win_amd64.whl (9.3 MB)
    |████████████████████████████████████████| 9.3 MB 3.3 MB/s
Collecting numpy>=1.16.5
  Downloading numpy-1.21.0-cp39-cp39-win_amd64.whl (14.0 MB)
    |████████████████████████████████████████| 14.0 MB 1.3 MB/s
Collecting python-dateutil>=2.7.3
  Using cached python_dateutil-2.8.1-py2.py3-none-any.whl (227 kB)
Collecting pytz>=2017.3
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Collecting six>=1.5
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, pytz, python-dateutil, numpy, pandas
Successfully installed numpy-1.21.0 pandas-1.2.5 python-dateutil-2.8.1

(venv) C:\Users\Fernando\PycharmProjects\Pandas>
```

Não havendo nenhum erro no processo de instalação, após um rápido download é instalado no sistema a última versão estável da biblioteca Pandas.

Tipos de dados básicos Pandas

- Series

Dos tipos de dados básicos da biblioteca Pandas, o primeiro que devemos entender é o tipo Serie. A biblioteca Pandas possui alguns tipos de dados particulares (de funcionalidades equivalentes a outros tipos de dados como arrays), onde uma Serie equivale a uma array unidimensional, com mapeamento de elementos e índice próprio, capaz de aceitar em sua composição todo e qualquer tipo de dado Python.

	Index	radius me
	0	17.99
	1	20.57
	2	19.69
	3	11.42
	4	20.29
	5	12.45
	6	18.25
	7	13.71
	8	13
	9	12.46

Por hora, raciocine que as estruturas de dados utilizados via Pandas se assemelham muito a uma planilha Excel, onde o conteúdo é disposto em formato de linhas e colunas. Facilitando assim a aplicação de operações sobre dados desta “tabela” uma vez que cada elemento da mesma possui uma referência de mapeamento de índice.

```
1 base = [1,3,5,7,9,12,15,18,21,24]
2
3 data = pd.Series(base)
4
5 print(data)
6
```


0	1
1	3
2	5
3	7
4	9
5	12
6	15
7	18
8	21
9	24

dtype: int64

Apenas como exemplo, no código acima temos uma variável de nome base que por sua vez recebe como atributo uma lista com diversos elementos.

Em seguida é criada uma nova variável de nome data que instancia e inicializa a função `pd.Series()`, parametrizando a mesma com os dados de base.

Exibindo em tela via função `print()` o conteúdo da variável data, é possível ver uma Serie composta de um índice, uma coluna e 10 linhas com seus respectivos elementos.

- DataFrames

Outro tipo de dado básico usado na biblioteca Pandas é o chamado DataFrame, muito semelhante a uma Serie (inclusive um DataFrame é formado por um conjunto de Series), porém equivalente a uma array multidimensional. Dessa forma, o comportamento e o tratamento dos dados ganham novas funcionalidades que veremos em detalhes nos próximos capítulos.

	Index	radius me	texture me	imeter m	area mea	othness r	actness	ncavity me
	0	17.99	10.38	122.8	1001	0.1184	0.2776	0.3001
	1	20.57	17.77	132.9	1326	0.08474	0.07864	0.0869
	2	19.69	21.25	130	1203	0.1096	0.1599	0.1974
	3	11.42	20.38	77.58	386.1	0.1425	0.2839	0.2414
	4	20.29	14.34	135.1	1297	0.1003	0.1328	198
	5	12.45	15.7	82.57	477.1	0.1278	0.17	0.1578
	6	18.25	19.98	119.6	1040	0.09463	109	0.1127
	7	13.71	20.83	90.2	577.9	0.1189	0.1645	0.09366
	8	13	21.82	87.5	519.8	0.1273	0.1932	0.1859

O ponto chave aqui é que, para dados dispostos em vetores e matrizes, a biblioteca Pandas oferece uma vasta gama de funções aplicáveis a tais dados desde que os mesmos estejam, seja nativos ou convertidos, em formato DataFrame.

```
1 base = {'Nomes':['Ana', 'Carlos', 'Gabriela', 'Maria'],
2         'Fones':[991384562, 981128449, 999510014, 991120991]}
3
4 data = pd.DataFrame(base)
5
6 print(data)
7
```

	Nomes	Fones
0	Ana	991384562
1	Carlos	981128449
2	Gabriela	999510014
3	Maria	991120991

Novamente apenas para fins de exemplo, inicialmente temos uma variável de nome base que recebe como um atributo um dicionário composto de duas chaves e uma lista de valores atribuídos para as mesmas.

Na sequência é criada uma variável de nome data, que instancia e inicializa a função `pd.DataFrame()`, por sua vez parametrizada com o conteúdo da variável base.

Via função `print()`, parametrizada com os dados de data, nos é exibido em tela um Dataframe, onde podemos notar que as chaves do dicionário de origem se tornaram os cabeçalhos das colunas, tendo seus respectivos valores dispostos nas linhas destas colunas, além é claro, do índice interno gerado para esse DataFrame.

ANÁLISE EXPLORATÓRIA DE DADOS

Importando a biblioteca Pandas

```
1 import pandas as pd
2
```

Uma vez que não tenha ocorrido nenhum erro no processo de instalação, para utilizarmos das funcionalidades da biblioteca Pandas é necessário importar a mesma para nosso código.

O processo é extremamente simples, igual ao de qualquer outra importação em Python. Sendo assim, por meio do comando `import pandas` todo seu conteúdo será importado ficando assim seus recursos à disposição do desenvolvedor.

Apenas por convenção podemos referenciar tal biblioteca por alguma sigla, o comum entre a comunidade é instanciar a biblioteca Pandas como simplesmente `pd`.

SERIES

Criando uma Serie

```
1 base = [1,3,5,7,9,12,15,18,21,24]
2
3 data = pd.Series(base)
4
5 print(data)
6
```

0	1
1	3
2	5
3	7
4	9
5	12
6	15
7	18
8	21
9	24

dtype: int64

Uma Serie pode ser criada de diversas formas, sendo a mais comum delas a partir de seu método construtor, utilizando como base dados em forma de lista, uma vez que se tratando de uma Serie teremos como resultado final uma array unidimensional.

No exemplo, dada uma variável de nome base, que por sua vez possui como atributo uma lista de elementos numéricos, podemos usar da mesma para a geração de uma Serie.

Então é criada uma nova variável, dessa vez de nome data, que instancia e inicializa a função `pd.Series()`, repassando como parâmetro para a mesma o conteúdo da variável base.

Exibindo em tela por meio de nossa função `print()`, nos é retornado uma estrutura com características de uma tabela indexada de apenas uma coluna e 10 linhas.

```
1 base = [1,3,5,7,9,12,15,18,21,24]
2
3 data = pd.Series(base)
4
5 print(type(data))
6
<class 'pandas.core.series.Series'>
```

Usando de nossa função `type()` aninhada a função `print()`, por sua vez parametrizada com `data`, é possível ver o tipo de dado de nossa variável `data`.

Como esperado, o conteúdo de `data` é uma Serie criada a partir dos dados de `base`, logo, nos é retornado como tipo de dado `pandas.core.series.Series`.

```
1 data = pd.Series([1,3,5,7,9,12,15,18,21,24])
2
3 print(data)
4
0    1
1    3
2    5
3    7
4    9
5   12
6   15
7   18
8   21
9   24
dtype: int64
```

Outra possibilidade é que criemos uma Serie via construtor, repassando os dados diretamente como parâmetro para a mesma, desde que tais dados estejam em formato de lista.

No exemplo, a variável `data` chama a função `pd.Series()` parametrizando a mesma com a lista `[1,3,5,7,9,12,15,18,21,24]`.

Exibindo novamente o conteúdo de data via função `print()` nos é retornado uma Serie igual as anteriores.

```
1 data2 = pd.Series(['Ana', 'Carlos', 'Diego', 'Fernando', 'Maria', 'Paulo'])
2
3 print(data2)
4
```



```
0      Ana
1    Carlos
2    Diego
3  Fernando
4     Maria
5     Paulo
dtype: object
```

Como mencionado anteriormente, tanto uma Serie quanto um DataFrame suporta todo e qualquer tipo de dado em sua composição.

Declarada uma variável de nome `data2`, é chamada a função `pd.Series()` parametrizando tal função com uma lista de nomes em formato string.

Exibindo em tela via `print()` o conteúdo de `data2`, como esperado é retornado uma Serie, uma tabela unidimensional, agora composta por nomes oriundos da lista de origem.

Visualizando o cabeçalho

```
1 print(data.head())  
2  
  
0    1  
1    3  
2    5  
3    7  
4    9  
dtype: int64
```

Algumas Series / DataFrames que veremos em tópicos futuros podem ter uma grande variedade de dados organizados nas mais diversas formas.

Uma maneira de se ter um primeiro contato com esses dados para entender visualmente sua estrutura é usar do método `head()`.

Aplicando o método `head()` em nossa variável `data`, exibindo a mesma em tela via função `print()`, o retorno será os 5 primeiros elementos de nossa Serie, já com seu índice, para que assim possamos ver a forma como os dados estão dispostos.

Definindo uma origem personalizada

```
1  nomes = ['Ana', 'Carlos', 'Diego', 'Fernando', 'Maria', 'Paulo']
2
3  data3 = pd.Series(data = nomes)
4
5  print(data3)
6
```

```
0      Ana
1    Carlos
2     Diego
3  Fernando
4     Maria
5     Paulo
dtype: object
```

Por justaposição, o primeiro parâmetro de nossa função `pd.Series()` deverá ser a origem dos dados a serem utilizados, essa origem pode ser personalizada (como veremos em exemplos posteriores) assim como definida manualmente pelo parâmetro nomeado `data =` seguido do nome da variável de origem.

Definindo um índice personalizado

```
1  indice = [1,2,3,4,5,6]
2  nomes = ['Ana', 'Carlos', 'Diego', 'Fernando', 'Maria', 'Paulo']
3
4  data4 = pd.Series(data = nomes,
5  | | | | | | | | index = indice)
6
7  print(data4)
8
```

```
1      Ana
2     Carlos
3     Diego
4  Fernando
5     Maria
6     Paulo
dtype: object
```

Explorando os possíveis parâmetros para a função `pd.Series()`, outro bastante útil é o parâmetro `index`, onde pelo qual podemos definir um índice personalizado.

Índices em Python são gerados iniciados em 0, quando não definimos um índice específico para nossa Serie será usado este formato de índice padrão Python, porém, para nossos dados em Series ou DataFrames podemos atribuir índices personalizados (com qualquer tipo de dado como índice, inclusive textos).

No exemplo, inicialmente temos uma variável de nome `indice` que recebe como atributo uma lista com alguns números ordenados.

Da mesma forma temos uma variável de nome `nomes` que recebe como atributo uma lista composta de alguns nomes em formato string.

Na sequência é criada uma variável de nome `data4`, que instancia e inicializa a função `pd.Series()`, definindo para o parâmetro nomeado `data` que os dados de origem serão oriundos da variável `nomes`, assim como para

index que os dados usados como índice deverão ser importados da variável indices.

Note que para esse processo de definir um índice personalizado, o número de elementos da lista deve ser o mesmo número de elementos do índice, caso contrário, será gerada uma exceção pois os dados não são equivalentes.

Via `print()`, parametrizada com `data4`, nos é retornado uma Serie onde o índice inicia pelo número 1 conforme a lista de origem, composta de uma coluna com os respectivos nomes oriundos da variável `nomes`.

Integrando uma Array Numpy em uma Serie

```
1  nomes = np.array(['Ana', 'Carlos', 'Diego', 'Fernando', 'Maria', 'Paulo'])
2
3  data5 = pd.Series(data = nomes)
4
5  print(data5)
6
```

```
0      Ana
1    Carlos
2    Diego
3  Fernando
4    Maria
5    Paulo
dtype: object
```

Uma das características da biblioteca Pandas é sua fácil integração com outras bibliotecas, suportando praticamente todo e qualquer tipo de estrutura de contêineres de dados para sua composição.

Em nosso exemplo, gerando uma array por meio da biblioteca Numpy, podemos perfeitamente transformar esta array numpy em uma Serie ou DataFrame Pandas.

Para isso, inicialmente é criada uma variável de nome `nomes` que por meio da função `np.array()` gera uma array do tipo numpy composta de uma lista de elementos em formato string.

Em seguida é criada uma variável de nome `data5` que por meio da função `pd.Series()`, transforma nossa array numpy de nomes para uma Serie.

Novamente, exibindo em tela o conteúdo de `data5` via função `print()` nos é retornada a Serie.

Qualquer estrutura de dados como elemento de uma Serie

```
1  funcoes = [input, print]
2
3  data6 = pd.Series(data = funcoes)
4
5  print(data6)
6

0  <bound method Kernel.raw_input of <google.colab...
1                                <built-in function print>
dtype: object
```

Uma vez que tudo em Python é objeto, dentro das possibilidades de uso de dados em uma Serie ou DataFrame podemos até mesmo usar de um ou mais métodos como elementos dessa Serie ou DataFrame.

Apenas como exemplo, declarada a variável `funcoes` que recebe uma lista com dois elementos, elementos estes que são palavras reservadas ao sistema para funções de entrada e de saída, respectivamente, tais elementos podem ser perfeitamente incorporados em uma Serie ou em um DataFrame.

Logo em seguida é criada uma variável de nome `data6` que instancia e inicializa a função `pd.Series()`, usando como dados de origem o conteúdo atrelado a variável `funcoes`.

Por fim, exibindo em tela o conteúdo de `data6` via `print()`, são retornadas as referências aos objetos referentes aos métodos `input()` e `print()`.

Integrando funções com Series

```
1  paises = ['Argentina', 'Brasil', 'Canada', 'Estados Unidos', 'Italia', 'Mexico']
2
3  data7 = pd.Series(data = paises, index = np.arange(0,6))
4
5  print(data7)
6
```

```
0      Argentina
1         Brasil
2         Canada
3  Estados Unidos
4         Italia
5         Mexico
dtype: object
```

No processo de geração de uma Serie ou um DataFrame por meio de seus respectivos métodos construtores, se tratando dos parâmetros nomeados utilizados, podemos atribuir funções para tais parâmetros sem que isso gere algum tipo de problema.

Em nosso exemplo, inicialmente é declarada uma variável de nome `paises` que recebe como atributo para si uma lista com alguns nomes de países em formato string.

De modo parecido é criada uma variável de nome `data7`, que chama a função `pd.Series()`, definindo manualmente que como dados o conteúdo associado deverá ser importado da variável `paises`, na sequência para o parâmetro `index` usamos de uma função da biblioteca Numpy `np.arange()`, que irá gerar valores ordenados de 0 até 5, a serem utilizados como índice.

Mais uma vez, via `print()`, agora parametrizada com `data7`, é exibido em tela a Serie composta pelo índice gerado e pela lista de nome de países usada como dados.

Verificando o tamanho de uma Serie

```
1 print(data.index)
2
RangeIndex(start=0, stop=10, step=1)
```

Aproveitando o tópico anterior, um ponto a ser entendido é que o tamanho em si de uma Serie é definido a partir de seu índice.

Em outras palavras, ao consultar o tamanho de uma Serie ou de um DataFrame, como tais estruturas podem ter camadas e mais camadas de dados sobrepostos, a referência para o tamanho sempre será o valor declarado em seu índice.

No exemplo, aplicando o método `index` para a variável `data` gerada em tópicos anteriores, é retornado um `RangeIndex`, dizendo que o conteúdo de `data` tem 10 elementos, iniciados em 0 até 10, contados de um em um elemento.

```
1 print(data7.index)
2
Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

Tentando reproduzir o mesmo retorno a partir de `data7` (variável gerada no tópico imediatamente anterior a este), repare que agora a referência é a própria lista, pois a mesma foi gerada por meio da função `np.arange()`.

O importante a entender aqui é que, ao inspecionar o tamanho de uma Serie ou DataFrame, apenas teremos um resultado útil quando tais estruturas de dados possuírem um índice nativo. Uma vez que tenhamos alterado o índice, personalizando o mesmo de alguma forma, internamente a referência original do índice é sobrescrita pela informação nova, perdendo assim a referência original que retornaria um dado para `index`.

Criando uma Serie a partir de um dicionário

```
1  dicionario = {'Nome':'Fernando', 'Idade': 33, 'Altura':1.90}
2
3  serie_dicio = pd.Series(dicionario)
4
5  print(serie_dicio)
```

Nome	Fernando
Idade	33
Altura	1.9

dtype: object

Como dito em alguns dos tópicos anteriores, uma Serie é uma estrutura de dados equivalente a uma array unidimensional.

Essa estrutura que define o esqueleto / o formato de uma Serie não pode ser alterada, de forma que ao tentarmos usar de dados multidimensionais para criação de uma Serie, tais dados serão remodelados para não alterar a estrutura original da Serie.

Usando de um exemplo, uma Serie pode ser criada a partir de um dicionário ou um contêiner de dados equivalente, porém, nesse caso, os dados das chaves do dicionário serão convertidos para o índice da série, enquanto os valores do dicionário serão utilizados como os dados da Serie.

Partindo para a prática, inicialmente é criada uma variável de nome dicionario que por sua vez, em forma de dicionário, recebe três chaves: 'Nome', 'Idade' e 'Altura' com seus respectivos valores 'Fernando', 33 e 1.90.

Em seguida é criada uma variável de nome serie_dicio que chama a função `pd.Series()` parametrizando a mesma com o conteúdo da variável dicionario.

Exibindo em tela por meio da função `print()` o conteúdo da variável serie_dicio, é possível notar que de fato os dados/valores das chaves se tornaram o índice, assim como os dados/valores dos valores do dicionário de origem se tornaram o conteúdo da Serie.

```
1 meses = {1:'Janeiro', 2:'Fevereiro', 3:'Março', 4:'Abril'}
2
3 serie_meses = pd.Series(meses)
4
5 print(serie_meses)
```

```
1      Janeiro
2      Fevereiro
3          Março
4          Abril
dtype: object
```

Usando de um dicionário origem onde suas chaves são números, o processo de conversão das chaves em índice se torna natural, retornando uma Serie onde seu índice é numérico e ordenado como esperado para maior parte dos contextos.

Unindo elementos de duas Series

```
1  nomes1 = ['Ana', 'Carlos', 'Betina', 'Maria', 'Rafael']
2  nomes2 = ['Ana', 'Alberto', 'Maria', 'Paulo', 'Tania']
3
4  data_1 = pd.Series([1,2,3,4,5], index = nomes1)
5  data_2 = pd.Series([1,2,3,4,5], index = nomes2)
6
7  data_3 = data_1 + data_2
8
9  print(data_3)
10
```

```
Alberto    NaN
Ana         2.0
Betina     NaN
Carlos     NaN
Maria       7.0
Paulo      NaN
Rafael     NaN
Tania      NaN
dtype: float64
```

Ao trabalharmos com Series possuímos uma série de restrições, seja pela estrutura base, seja pelo formato interno, seja por sua indexação, seja por interação com outras Series, entre outras situações, algo que como veremos nos próximos capítulos são limitações apenas de Series, que não ocorrem em DataFrames que são estruturas de dados mais robustas.

Apenas simulando um erro rotineiro, ao simplesmente tentar unir dados de duas Series origem para a criação de uma terceira, já começamos a ter comportamentos que podem não ser o esperado para o contexto geral.

Diretamente ao exemplo, inicialmente temos duas variáveis de nomes nome1 e nome2, respectivamente, onde como atributo para as mesmas temos duas listas com alguns nomes em formato de string.

Na sequência é criada uma variável de nome data_1 que, por meio da função `pd.Series()`, gera uma Serie onde por justaposição seus dados serão

a lista [1,2,3,4,5], definindo que como índice serão utilizados os dados oriundos de nomes1.

Exatamente o mesmo processo é feito para data_2, apenas alterando que para seu índice deverão ser considerados os dados importados de nomes2.

Em seguida é criada uma nova variável de nome data_3, que simplesmente como atributo recebe a soma dos elementos de data_1 e de data_2.

Exibindo em tela o resultado dessa soma via print(), é possível notar que nos é retornado a soma dos índices dos elementos repetidos, por exemplo, Maria em nomes1 possui o valor de índice 4, e em nomes2 o valor de índice 7, pois definimos que o índice da Serie iniciaria em 1. No retorno, Maria aparecerá com valor 7, o que é esperado, porém, onde não houverem dados com valores a serem somados o retorno é Nan.

Dessa forma, ao invés de uma soma ou sobreposição de dados como o contexto exigiria, o que é gerado é um espaço alocado sem dado nenhum. Tal situação certamente acarretaria em todo um tratamento específico para contornar essa ausência de dados, alterando em definitivo a integridade dos dados originais, o que não pode ocorrer.

Sendo assim, é interessante termos em mente sempre que Series são dados bastante básicos, rápidos e eficientes para certas situações, porém muito restritivos quando se tratando de interações de dados em Series distintas.

DATAFRAMES

Partindo para a principal estrutura de dados da biblioteca Pandas, finalmente vamos entender as particularidades dos chamados DataFrames nos mesmos moldes dos tópicos anteriores, ou seja, na prática por meio de exemplos.

Importante salientar que o fato de termos dedicado toda uma parte deste pequeno livro falando especificamente sobre Series não necessariamente acarreta em conhecimento inutilizado, muito pelo contrário, haja visto que um DataFrame é constituído de Series, tudo o que vimos até o momento em Series se aplica a DataFrames.

Criando um DataFrame

```
1 base = {'Nomes':['Ana', 'Carlos', 'Gabriela', 'Maria'],
2         'Fones':[991384562, 981128449, 999510014, 991120991]}
3
4 data = pd.DataFrame(base)
5
6 print(data)
7
```

	Nomes	Fones
0	Ana	991384562
1	Carlos	981128449
2	Gabriela	999510014
3	Maria	991120991

Dando início ao entendimento do uso de DataFrames diretamente na prática, não podemos deixar de entender sua estrutura básica desde sua criação, como fizemos ao dar nossos primeiros passos com Series.

DataFrames como dito anteriormente, são estruturas de dados, normalmente criadas a partir de dicionários, onde teremos dados organizados e dispostos em algo muito semelhante a uma tabela Excel, haja visto que aqui trabalharemos com matrizes multidimensionais, separados em linhas e colunas mapeadas, para que possamos de forma simples manipular dados dessas estruturas sem a necessidade de ferramentas externas.

Partindo para o código, inicialmente declarada uma variável de nome base, que possui como atributo um dicionário simples de dois conjuntos de chaves : valores, podemos nos mesmos moldes anteriores gerar outras estruturas de dados a partir dessa variável.

Para isso, criamos uma variável de nome data que instancia e inicializa o método construtor de DataFrames `pd.DataFrame()`, parametrizando o mesmo com o conteúdo da variável base.

Exibindo em tela via função `print()` o conteúdo de base, temos algo muito semelhante ao que foi visto até o momento em uma Serie, com o

grande diferencial de que agora temos dados em uma tabela equivalente a uma array / matriz multidimensional.

```
1 base = {'Nomes':['Ana', 'Carlos', 'Gabriela', 'Maria'],
2         'Fones':[991384562, 981128449, 999510014, 991120991]}
3
4 data = pd.DataFrame(base)
5
6 print(data)
7 print(type(data))
8
```

	Nomes	Fones
0	Ana	991384562
1	Carlos	981128449
2	Gabriela	999510014
3	Maria	991120991

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

Verificando o tipo de dado por meio de nossa função `type()` aninhada para `print()`, parametrizada com `base`, nos é retornado `pandas.core.frame.DataFrame`.

```
1 print(data.info())
2
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0   Nomes   4 non-null          object
1   Fones   4 non-null          int64
dtypes: int64(1), object(1)
memory usage: 192.0+ bytes
None
```

Usando do método `info()` podemos obter um resumo gerado para nosso DataFrame, uma vez que tal base de dados esteja associada a uma variável, podendo assim aplicar o método `info()` para a mesma, retornando assim informações como tipo de dado, número de elementos, tipo de dado dos elementos, finalizando com tamanho alocado em memória para esse DataFrame.

```
1 print(data.describe())
2
```

	Fones
count	4.000000e+00
mean	9.907860e+08
std	7.524343e+06
min	9.811284e+08
25%	9.886229e+08
50%	9.912528e+08
75%	9.934159e+08
max	9.995100e+08

Outra possível verificação rápida sobre os dados de nosso DataFrame pode ser feita por meio do método `describe()`, que por sua vez retorna (quando aplicável) informações sobre quantidade de elementos, média dos valores dos mesmos, desvio padrão de tais valores, assim como os valores mínimos, 25%, 50%, 75% e valor máximo encontrado para os elementos deste DataFrame.

```

1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6

```

	a	b	c	d	e
1	1.567145	-0.671315	0.557331	0.184358	0.003747
2	0.905385	-2.826999	2.735895	-1.035738	0.267082
3	-0.914936	-0.507777	-0.561623	0.125336	-0.718551
4	1.239431	1.534294	0.544628	0.205202	-1.153436
5	1.378925	-0.641252	-0.581054	-1.172074	0.565122
6	0.274400	0.285501	-0.206488	1.763421	0.264679

Lembrando que para um DataFrame se aplicam todos os métodos utilizados até o momento, como exemplo podemos criar um DataFrame com dados gerados a partir de uma função embutida ou de outra biblioteca, por exemplo `np.random.randn()` que para nosso exemplo irá gerar uma matriz de 6 linhas e 5 colunas de dados entre 0 e 1.

De forma parecida, definimos manualmente para alguns parâmetros nomeados alguns dados/valores para compor nosso DataFrame.

Desses parâmetros, um não explorado até o momento é o `columns`, onde podemos especificar que referências iremos passar como cabeçalho para nossas colunas, uma vez que um DataFrame, diferentemente de uma Serie, terá duas ou mais colunas em sua forma.

Por meio da função `print()`, por sua vez parametrizada com `data`, é possível notar que via método construtor `pd.DataFrame()` geramos uma estrutura de 6 linhas, 5 colunas, índice sequencial de 1 até 6 e cabeçalho de colunas com letras de 'a' até 'e'.

```

1 print(data.columns)
2

```

```

Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

```

Na mesma linha de raciocínio que viemos criando, extraindo informações apenas de uma coluna, a mesma internamente será considerada uma Serie.

Por meio da função `print()`, parametrizada com `data.columns`, é exibido em tela um índice específico para as colunas, pois estruturalmente cada coluna de nosso DataFrame é uma Serie.

```
1 print(data.index)
2
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

Como visto em outros momentos, por meio de `data.index` temos acesso ao índice interno de nosso DataFrame.

O ponto a destacar aqui é que, uma vez que temos um DataFrame, com índices gerados tanto para suas linhas quanto para suas colunas, conseguimos mapear facilmente qualquer dado/valor de qualquer elemento deste DataFrame.

```
1 print(data['c'])
1 -0.365474
2 -0.369080
3 -0.506489
4  1.332194
5  0.033296
6 -0.306698
Name: c, dtype: float64
```

Uma vez que temos uma coluna como uma Serie, e uma Serie tem seus dados lidos como em uma lista, podemos usar da notação de listas em Python para fazer referência a uma coluna ou a um determinado elemento da mesma.

Em nosso exemplo, parametrizando nossa função `print()` com `data` na posição `['c']` (*notação de lista), nos é retornado apenas os dados da coluna `c` de nosso `DataFrame`.

```
1 print(type(data))
2
<class 'pandas.core.frame.DataFrame'>
```

Visualizando o tipo de dado de `data` (todo o `DataFrame`) via `type()` nos é retornado `pandas.core.frame.DataFrame`.

```
1 print(type(data['c']))
<class 'pandas.core.series.Series'>
```

Visualizando o tipo de dado de `data['c']` (apenas a coluna `'c'` do `DataFrame`) nos é retornado `pandas.core.series.Series`.

Extraindo dados de uma coluna específica

```
1 print(data['c'])  
  
1    -0.365474  
2    -0.369080  
3    -0.506489  
4     1.332194  
5     0.033296  
6    -0.306698  
Name: c, dtype: float64
```

Como visto anteriormente, tratando uma coluna como uma Serie podemos extrair dados de uma determinada coluna usando da notação de extração de dados de uma lista.

```
1 print(data.d)  
2  
  
1    -1.278880  
2    -0.773291  
3     0.917642  
4     0.750692  
5     1.571643  
6     1.065574  
Name: d, dtype: float64
```

Porém, o método usual para extração de dados de uma coluna específica de um DataFrame é simplesmente fazer referência ao nome do cabeçalho da mesma.

Via `print()`, parametrizado com `data.d`, estamos exibindo em tela apenas o conteúdo da coluna de nome 'd' de nosso DataFrame.

Embora esse seja o método nativo, o mesmo possui como limitação a extração de dados de apenas uma coluna por vez.

```
1 print(data[['c', 'e']])  
2
```

	c	e
1	-0.365474	-0.145414
2	-0.369080	1.001909
3	-0.506489	-0.939375
4	1.332194	0.587649
5	0.033296	-0.070125
6	-0.306698	-0.484401

Retomando o uso da notação de listas (posição em listas) podemos extrair informações de quantas colunas quisermos.

Em nosso exemplo, exibindo em tela por meio de `print()` o conteúdo de `data` nas posições `['c', 'e']` é retornado apenas os conteúdos de tais colunas.

Criando colunas manualmente

```
1 data['f'] = data['a'] + data['e']
2
3 print(data)
4
```

	a	b	c	d	e	f
1	-1.182104	-1.318457	-0.365474	-1.278880	-0.145414	-1.327519
2	-0.309844	0.840820	-0.369080	-0.773291	1.001909	0.692064
3	-0.691325	0.528299	-0.506489	0.917642	-0.939375	-1.630700
4	-0.789610	0.934323	1.332194	0.750692	0.587649	-0.201960
5	-0.228530	0.226750	0.033296	1.571643	-0.070125	-0.298656
6	0.650299	-0.067245	-0.306698	1.065574	-0.484401	0.165897

Uma vez que estamos usando da notação de posição em listas, podemos usar de todos os métodos que se aplicam a esta notação.

Relembrando o básico em Python, ao fazer instância a uma posição de lista inexistente e atribuindo a essa instância um dado/valor, a mesma será criada na lista.

Em nosso exemplo, instanciando a posição ['f'] de data (que até o momento não existe), atribuindo para a mesma os dados da soma entre os dados de data nas posições ['a'] e ['e'], não havendo nenhum erro de sintaxe ou incompatibilidade dos dados de tais colunas, a coluna 'f' será criada.

Exibindo em tela o conteúdo de data agora é possível notar que de fato foi criada a coluna 'f' e seus dados são o resultado da soma de cada elemento de 'a' e 'e' em suas respectivas linhas.

Removendo colunas manualmente

```
1 data = data.drop('f', axis=1)
2
3 print(data)
4
```

	a	b	c	d	e
1	-1.182104	-1.318457	-0.365474	-1.278880	-0.145414
2	-0.309844	0.840820	-0.369080	-0.773291	1.001909
3	-0.691325	0.528299	-0.506489	0.917642	-0.939375
4	-0.789610	0.934323	1.332194	0.750692	0.587649
5	-0.228530	0.226750	0.033296	1.571643	-0.070125
6	0.650299	-0.067245	-0.306698	1.065574	-0.484401

Para remover uma determinada coluna de nosso DataFrame temos mais de uma forma, cabendo ao desenvolvedor optar por usar a qual achar mais conveniente.

Em nosso exemplo, aplicando o método `drop()` em nossa variável `data`, podemos remover uma coluna desde que especifiquemos o nome do cabeçalho da mesma, seguido do parâmetro `axis = 1`, para que de fato a função `drop()` exclua todos os dados da coluna, e não da linha.

Lembrando que para tornar o efeito permanente devemos atualizar a variável de origem, caso contrário, apenas será apagada a referência, porém os dados ainda estarão em suas instâncias originais.

```

1 data.drop('e', axis=1, inplace=True)
2
3 print(data)
4

```

	a	b	c	d
1	-1.182104	-1.318457	-0.365474	-1.278880
2	-0.309844	0.840820	-0.369080	-0.773291
3	-0.691325	0.528299	-0.506489	0.917642
4	-0.789610	0.934323	1.332194	0.750692
5	-0.228530	0.226750	0.033296	1.571643
6	0.650299	-0.067245	-0.306698	1.065574

Caso você esteja usando de um notebook ipynb (Google Colab, Jupyter, DataLore, entre outros) para tornar a ação de exclusão de uma coluna permanente se faz necessário o uso do parâmetro `inplace = True`, caso contrário, os dados continuarão presentes, apenas sem referência.

```

1 del data['b']
2
3 print(data)
4

```

	a	c	d
1	1.919214	0.009094	0.879479
2	-0.315871	-0.490806	1.200689
3	0.587436	1.055801	0.274803
4	-1.425324	0.527136	0.236453
5	-0.024379	0.053845	1.046188
6	-1.256685	0.175534	-1.474178

Outra forma de remover uma determinada coluna assim como todo seu conteúdo é por meio da função do sistema `del`, especificando qual a variável origem e a posição de índice ou nome do cabeçalho o qual remover. Por se tratar de uma função do sistema, independentemente do ambiente, essa alteração será permanente.

Exibindo em tela o conteúdo de `data` em nossa função `print()`, é possível notar que a coluna de nome `'b'` foi removida como era esperado.

Ordenando elementos de uma coluna

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2                       index = [1,2,3,4,5,6],
3                       columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6
7 data.sort_values(by = 'b', inplace = True)
8
9 print(data)
10
```

	a	b	c	d	e
1	1.696631	1.045160	0.507513	0.505087	-0.524425
2	0.418782	-0.309066	0.798466	0.015714	-0.492760
3	-1.040263	-0.208836	0.080758	-0.024275	1.639983
4	0.716648	-1.974941	1.628613	-0.454698	1.647022
5	0.130651	-0.087541	-0.103105	-0.004928	-1.015706
6	1.152694	0.137474	-0.246503	-0.473699	0.208946

	a	b	c	d	e
4	0.716648	-1.974941	1.628613	-0.454698	1.647022
2	0.418782	-0.309066	0.798466	0.015714	-0.492760
3	-1.040263	-0.208836	0.080758	-0.024275	1.639983
5	0.130651	-0.087541	-0.103105	-0.004928	-1.015706
6	1.152694	0.137474	-0.246503	-0.473699	0.208946
1	1.696631	1.045160	0.507513	0.505087	-0.524425

Para nosso exemplo, inicialmente vamos gerar um novo DataFrame, haja visto que nos tópicos anteriores realizamos algumas manipulações dos dados do DataFrame anterior.

Para isso a variável `data` instancia e inicializa a função `pd.DataFrame()`, gerando dados aleatórios entre 0 e 1, distribuídos em 6 linhas e 5 colunas, por meio da função `np.random.randn()`, também atribuímos um índice numérico personalizado e nomes para as colunas igual ao DataFrame dos exemplos anteriores. A única diferença de fato serão os valores que foram gerados aleatoriamente novamente para este DataFrame.

Na sequência, aplicando em nossa variável `data` o método `sort_values()` podemos ordenar de forma crescente os dados de uma determinada coluna apenas especificando qual para o parâmetro `by`.

Repare que para a o retorno da primeira função `print()` temos o `DataFrame` com seus dados dispostos em sua forma original, enquanto para o segundo retorno, como esperado, os dados da coluna 'b' foram reordenados de forma crescente.

Extraindo dados de uma linha específica

```
1 print(data.loc[3])
2
a    -1.040263
b    -0.208836
c     0.080758
d    -0.024275
e     1.639983
Name: 3, dtype: float64
```

Continuando dentro da linha de raciocínio de notação em posição de lista, quando estamos manipulando dados de listas a partir desta notação é muito comum usar do método `loc[]` que em notação de posição de índice, retorna dados segundo este parâmetro.

Por meio da função `print()` por sua vez parametrizada como `data.loc[3]`, ao localizar o elemento de posição de índice 3, será retornado o dado/valor do mesmo. Nesse caso, note que o retorno é uma Serie mostrando os dados da linha 3 rearranjados dessa forma, o que apesar de inicialmente confuso, é bastante funcional.

Lembre-se sempre que todos dados retornados de um DataFrame, quando destacados dos demais, serão exibidos em formato de Serie, nem que para isso linhas virem colunas e vice-versa.

Extraindo dados de um elemento específico

```
1 print(data.loc[2, 'b'])  
2  
-0.30906577188384676
```

Ainda na mesma notação, usando `loc[]` passando para o mesmo duas referências, em justaposição a primeira referência será referente a linha, assim como a segunda referência será o identificador para a coluna.

Em nosso exemplo, parametrizando nossa função `print()` com `data.loc[2, 'b']` estamos exibindo em tela apenas o dado/valor do elemento situado na linha 2 da coluna b.

Extraindo dados de múltiplos elementos

```
1 print(data.loc[[2, 3], ['a', 'b', 'c']])
2
```

	a	b	c
2	0.418782	-0.309066	0.798466
3	-1.040263	-0.208836	0.080758

Para extrair uma seleção de elementos, basta na mesma notação, no lugar do primeiro parâmetro de `loc[]` referente as linhas repassar uma lista de linhas, assim como para o parâmetro referente as colunas repassar uma lista de colunas.

Em nosso exemplo, parametrizando `print()` com `data.loc[[2, 3], ['a', 'b', 'c']]`, nos são retornados os elementos das linhas 2 e 3, situados nas colunas a, b e c.

```
1 print(data.iloc[1:3, 0:3])
2
```

	a	b	c
2	0.418782	-0.309066	0.798466
3	-1.040263	-0.208836	0.080758

Outra forma perfeitamente funcional para extração de dados de nossos DataFrames é usar de `iloc[]`, método muito semelhante ao `loc[]` porém mais robusto, suportando a extração de elementos de uma lista baseado em intervalos numéricos de índices.

Exatamente como no exemplo anterior, parametrizando nossa função `print()`, agora com `data.iloc[1:3, 0:3]`, estamos extraindo os elementos situados entre as posições de índice 1 a 3 referente às linhas 2 e 3, do mesmo modo as posições de índice referente às colunas no intervalo 0 a 3, ou seja, a, b e c, nos são retornados os mesmos elementos do exemplo anterior.

Buscando elementos via condicionais

```
1 print(data > 0)
2

```

	a	b	c	d	e
4	True	False	True	False	True
2	True	False	True	True	False
3	False	False	True	False	True
5	True	False	False	False	False
6	True	True	False	False	True
1	True	True	True	True	False

Se tratando de dados em DataFrames, uma forma rápida de filtrar os mesmos é definindo alguma estrutura condicional simples.

Por exemplo, via função `print()`, parametrizada com `data > 0`, será retornado para cada elemento uma referência `True` ou `False` de acordo com a condição imposta. Em nosso caso, cada elemento `True` faz referência a um valor maior que 0, enquanto cada elemento exibido como `False` não valida tal condição.

Lembrando que nesse caso, via função `print()`, não estamos alterando os dados de origem, apenas obtendo um retorno booleano para uma condição imposta. Para alterar os dados de origem devemos atualizar a variável `data`.

OPERAÇÕES MATEMÁTICAS EM DATAFRAMES

Usando de funções embutidas do sistema

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6
7 print(data['b'].sum())
8
```

	a	b	c	d	e
1	-0.939620	-1.134347	-0.209130	-1.235066	-0.745407
2	-2.229876	-2.036550	0.972273	0.041709	-2.553051
3	0.657049	-0.641325	-1.236023	0.335881	0.767927
4	0.072190	-0.163500	1.284958	-1.096049	-0.445813
5	-0.834458	-1.140670	-0.484915	1.459954	0.949029
6	0.471434	0.662064	-0.428318	-0.910454	-1.620817
	-4.454328061670092				

Para os exemplos a seguir, novamente geramos um DataFrame do zero pois anteriormente realizamos algumas manipulações que para os exemplos seguintes poderiam gerar algumas exceções.

Sendo assim, novamente geramos um DataFrame de valores aleatórios, índice numérico e índice de colunas definidos manualmente.

Usando de funções embutidas do sistema, toda e qualquer função pode ser usada normalmente, como quando aplicada a qualquer elemento de qualquer contêiner de dados.

Em nosso exemplo, por meio da função `print()`, agora parametrizada com `data` na posição `['b']`, aplicando sobre essa posição em índice o método `sum()`, o retorno gerado é a soma dos elementos da coluna `b` de nosso DataFrame, nesse caso, `-4.454328061670092`.

Aplicando uma operação matemática a todos elementos

```
1 data = data + 1
2
3 print(data)
4
```

	a	b	c	d	e
1	0.060380	-0.134347	0.790870	-0.235066	0.254593
2	-1.229876	-1.036550	1.972273	1.041709	-1.553051
3	1.657049	0.358675	-0.236023	1.335881	1.767927
4	1.072190	0.836500	2.284958	-0.096049	0.554187
5	0.165542	-0.140670	0.515085	2.459954	1.949029
6	1.471434	1.662064	0.571682	0.089546	-0.620817

Uma vez que temos um DataFrame em uma variável, como mencionado anteriormente, tal estrutura de dados possui uma notação característica identificada normalmente por nosso interpretador como matrizes de dados, ou seja, dados de listas.

Dessa forma, podemos usar de operações matemáticas simples diretamente aplicadas à variável, lembrando que nesses casos, a operação em si terá efeito sobre todos os elementos de nosso DataFrame.

Por exemplo, instanciando nossa variável data, atualizando a mesma com data + 1, será somado o valor 1 a todos os elementos do DataFrame original.

Usando de funções matemáticas personalizadas

```
1  def soma(x):  
2      | return x + x  
3  
4  print(data['b'].apply(soma))  
5  
  
1  -0.268694  
2  -2.073100  
3    0.717351  
4    1.672999  
5   -0.281339  
6    3.324127  
Name: b, dtype: float64
```

Além de usar de funções embutidas do sistema ou funções matemáticas básicas padrão, outra possibilidade é a de usar de funções matemáticas personalizadas, aplicadas a elementos de apenas uma coluna de nosso DataFrame, e isso é feito por meio do método `apply()`.

Em nosso exemplo, inicialmente definimos uma função personalizada de nome `soma()`, que receberá um valor para `x`, retornando tal valor somado com ele próprio. Lembrando que aqui cabe toda e qualquer operação aritmética ou expressão matemática.

Na sequência, por meio de `print()` parametrizado com `data` na posição `['b']`, aplicando o método `apply()` por sua vez parametrizado com nossa função `soma()`, é exibido em tela uma Serie onde cada um dos elementos dessa Serie (cada um dos elementos da coluna `b` de nosso DataFrame) teve seu valor alterado de acordo com nossa função `soma()`.

```

1  def ao_quadrado(num):
2      return num ** 2
3
4  print(data['a'].apply(ao_quadrado))
5

```

```

1      0.003646
2      1.512595
3      2.745811
4      1.149591
5      0.027404
6      2.165118
Name: a, dtype: float64

```

Outro exemplo, via `print()` exibimos em tela os valores dos elementos da coluna `a` de nosso DataFrame elevados ao quadrado, usando do método `apply()` aplicado sobre nossa variável `data` na posição `['a']` a função `ao_quadrado()` criada anteriormente.

```

1  def ao_quadrado(num):
2      return num ** 2
3
4  print(data.iloc[0:1, 1:2].apply(ao_quadrado))
5

```

```

      b
1  0.018049

```

Dentro da mesma notação vista anteriormente em outros exemplos, podemos selecionar um ou mais elementos específicos de nosso DataFrame por meio de `iloc[]` aplicando sobre os mesmos via `apply()` alguma função matemática personalizada.

Em nosso exemplo, é exibido em tela apenas o elemento situado na linha 1, coluna `b`, (intervalo de índice para linha entre 0 e 1, para coluna entre 1 e 2), tendo seu valor elevado ao quadrado de acordo com a função aplicada sobre o mesmo.

```
1 print(data['d'].apply(lambda x: x ** 2))
2
1    0.055256
2    1.085158
3    1.784579
4    0.009225
5    6.051373
6    0.008018
Name: d, dtype: float64
```

Por fim, encerrando essa linha de raciocínio, usando do método `apply()` para aplicar sobre elementos de nosso `DataFrame` funções personalizadas, uma prática comum é usar de expressões `lambda` para tornar o código reduzido nestas aplicações.

Apenas como exemplo, exibindo em tela via função `print()` os elementos da posição `['d']` de nossa variável `data`, elevados ao quadrado via função anônima escrita diretamente como parâmetro do método `apply()`, em forma reduzida.

ESTRUTURAS
DATAFRAMES

CONDICIONAIS

APLICADAS

A

Aplicação de estruturas condicionais simples

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2                       index = [1,2,3,4,5,6],
3                       columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 print(data['a'])
7 print(data['a'] < 0)
8
```

	a	b	c	d	e
1	1.300745	0.450395	0.563108	0.969970	-1.205950
2	1.043816	0.581930	-0.404550	0.146474	0.839320
3	-0.423912	1.900624	1.314251	-1.386419	1.646977
4	1.176944	0.242762	-0.161281	-0.774744	-1.532771
5	-0.184068	0.161966	-1.429594	-1.576289	-0.012632
6	-1.253910	-0.425104	-0.013969	-0.357884	0.614007

```
1 1.300745
2 1.043816
3 -0.423912
4 1.176944
5 -0.184068
6 -1.253910
Name: a, dtype: float64
1 False
2 False
3 True
4 False
5 True
6 True
Name: a, dtype: bool
```

Durante a manipulação de nossos dados, uma prática comum é o uso de estruturas condicionais para, com base em estruturas condicionais definidas, filtrar elementos ou até mesmo definir comportamentos em nosso DataFrame.

Como estrutura condicional simples em Python entendemos que em sua expressão, existe apenas um campo a ser validado como verdadeiro para que assim toda uma cadeia de processos seja executada.

Mais uma vez, apenas para garantir a consistência de nossos exemplos, um novo DataFrame é gerado do mesmo modo como os anteriores.

Partindo para a prática, para as linhas 5, 6 e 7 de nosso código declaramos três funções `print()`, sendo a primeira parametrizada apenas com `data`, exibindo dessa forma todo o conteúdo do DataFrame. Segunda função `print()` parametrizada com `data` na posição `['a']` exibe apenas os elementos da coluna `a` do DataFrame. Terceira função `print()` exibe `data` na posição `['a']` validando uma condição simples onde apenas serão exibidos como `True` os valores menores que 0.

Lembrando novamente que apenas para fins de testes, usar dessas estruturas condicionais em nossa função `print()` não altera os dados originais nem modifica nenhum comportamento de nosso DataFrame. Para esses casos onde se faz necessário a modificação de elementos do DataFrame, a variável ao qual o DataFrame está associado deve ser atualizada.

```

1 data = pd.DataFrame(data = np.random.randn(6,5),
2                      index = [1,2,3,4,5,6],
3                      columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 print(data['a'])
7
8 data2 = data[data['a'] < 0]
9
10 print(data2)
11

```

```

      a         b         c         d         e
1  0.055152  0.020853  0.228524  1.970407 -0.632108
2  0.422964  1.576821  1.490857  0.354183 -0.247402
3  1.315456 -0.886096  0.057999 -1.431807 -0.096454
4 -0.327879  0.051827 -0.820831 -0.056633 -0.842569
5 -0.481679 -1.251238  0.165817  0.648472 -1.163557
6 -1.354135 -1.952505 -0.251335 -0.034813 -1.404097
1    0.055152
2    0.422964
3    1.315456
4   -0.327879
5   -0.481679
6   -1.354135
Name: a, dtype: float64
      a         b         c         d         e
4 -0.327879  0.051827 -0.820831 -0.056633 -0.842569
5 -0.481679 -1.251238  0.165817  0.648472 -1.163557
6 -1.354135 -1.952505 -0.251335 -0.034813 -1.404097

```

De modo parecido com o exemplo anterior, uma vez que temos um DataFrame associado a variável `data`, via função `print()` podemos ver todo o conteúdo do DataFrame.

Para nossa segunda função `print()` repassamos como parâmetro `data` na posição `['a']`, logo, é exibido em tela o conteúdo da coluna `a` de nosso DataFrame.

Na sequência é criada uma variável de nome `data2`, que recebe como atributo um novo DataFrame baseado no anterior, onde para o mesmo serão

mantidos apenas os dados referentes a data na posição ['a'] que forem menores que 0.

Por fim é exibido em tela o conteúdo de data2 via função print().

Repare que como foi feito nos exemplos anteriores, usando dessa notação e modo de estrutura condicional, nos era retornado o mesmo DataFrame com marcadores True e False para os resultados das validações.

Agora, apenas para fins de exemplo, ao atribuir essa mesma estrutura a uma variável, exibindo em tela seu conteúdo é possível notar que os elementos os quais não atingiam a condição imposta simplesmente foram descartados, de modo que o novo DataFrame gerado possui um novo formato de acordo com os elementos presentes.

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 #print(data['a'])
7 print(data[data['a'] < 0])
8 print(data[data['a'] < 0]['e'])
9
```

	a	b	c	d	e
1	-0.975251	0.841130	-1.106314	-0.426608	0.287946
2	0.363689	1.081690	0.624923	2.125624	-2.391818
3	0.172398	-0.196537	0.675685	-0.149475	-1.014085
4	-0.429771	1.153912	0.986318	0.737138	0.255179
5	-0.105551	0.270492	0.117753	-1.270797	-0.154594
6	1.551044	-1.699938	-0.335873	-0.359221	-0.297094

	a	b	c	d	e
1	-0.975251	0.841130	-1.106314	-0.426608	0.287946
4	-0.429771	1.153912	0.986318	0.737138	0.255179
5	-0.105551	0.270492	0.117753	-1.270797	-0.154594

1	0.287946
4	0.255179
5	-0.154594

Name: e, dtype: float64

Outra situação viável é, usando de estruturas condicionais aplicadas a nosso DataFrame, gerar um novo DataFrame composto de dados obtidos a partir de validações em estruturas condicionais.

Como exemplo, usando do mesmo DataFrame do tópico anterior, como já visto anteriormente, para tornar alterações permanentes em um DataFrame estamos acostumados a associá-lo a uma variável, para que atualizando o conteúdo de seus atributos as alterações se tornem permanentes.

Porém, existe um modo de realizar tais alterações de forma permanente a partir de nossa função `print()`. Para isso, note que na linha 7 de nosso código temos como parâmetro de nossa função `print()` `data`, que na posição `[data['a'] < 0]` é aplicada uma estrutura condicional. Nesse caso, será retornado um novo DataFrame sem os elementos da coluna 'a' que forem menores que 0, e esta alteração é permanente.

Outra possibilidade, referente a linha 8 de nosso código, é usar de uma estrutura condicional que valida dados tendo como base duas referências diferentes. Este processo é muito parecido com o anterior, porém com algumas particularidades por parte de sua notação.

Em nossa linha 8 do código, como parâmetro repassado para a função `print()` existe a expressão `data` que na posição `[data['a'] < 0]['e']`, para que nesse caso seja retornado um novo DataFrame usando de uma estrutura condicional aplicada sobre `data` na posição `['a']` validando seus elementos menores que 0, retornando os elementos equivalentes da coluna 'e'.

Por fim, repare que toda essa expressão altera permanentemente os dados de `data` conforme a notação explicada no exemplo anterior.

```
1 data2 = data[data['a'] < 0]['e']
2
3 print(data2)
4
```

```
1    0.287946
4    0.255179
5   -0.154594
Name: e, dtype: float64
```

Como já visto inúmeras vezes, é perfeitamente possível isolar tais dados, extraindo os mesmos para uma nova variável.

Apenas como exemplo, é declarada uma variável de nome `data2` que recebe como atributo os dados de `data` na posição `[data['a'] < 0]['e']` exatamente como feito anteriormente.

Exibindo em tela via função `print()` o conteúdo de `data2` nos é apresentado uma Serie composta dos elementos da coluna `e` que atendiam a condição imposta para `data` na posição `[data['a']]`.

Em resumo, a primeira expressão serve apenas como referência, pois os dados retornados eram os elementos equivalentes situados na coluna `e` do DataFrame de origem.

Aplicação de estruturas condicionais compostas

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 #print(data['a'])
7 print(data[(data['a'] < 0) & (data['b'] > 1)])
8
```

	a	b	c	d	e
1	-0.268008	-0.228818	-0.083881	1.258773	1.629901
2	0.906926	-0.764981	0.981643	-0.258335	-1.142479
3	0.504328	-1.309354	-1.667187	-0.618108	-1.193881
4	-0.523564	-0.125894	-0.748558	-0.790971	-0.534490
5	-1.419431	0.776450	-1.595538	1.597824	-0.036342
6	0.649353	-1.002809	-0.773971	-0.057788	0.125344

Empty DataFrame
Columns: [a, b, c, d, e]
Index: []

Uma vez entendidos os conceitos básicos funcionais da aplicação de estruturas condicionais simples para os dados de nosso DataFrame, podemos avançar um pouco mais entendendo a lógica de aplicação de estruturas condicionais compostas.

Como estruturas condicionais compostas em Python temos expressões com a mesma notação das condicionais simples, porém, um grande diferencial aqui é que temos de usar alguns operadores um pouco diferentes do usual para conseguir de fato validar duas ou mais expressões condicionais.

Revisando o básico em Python, em um ambiente de código padrão, é muito útil usar dos operadores `and` e `or` para criar estruturas condicionais compostas. Por meio do operador `and` definimos que as duas expressões condicionais devem ser verdadeiras para assim desencadear a execução dos blocos de código atrelados a esta estrutura. Da mesma forma, usando do operador `or` criamos estruturas condicionais compostas onde apenas uma

das expressões sendo verdadeira já valida todo o resto, acionando os gatilhos para execução dos seus blocos de código.

Por parte dos operadores, para a biblioteca Pandas temos duas particularidades as quais devemos respeitar, caso contrário serão geradas exceções. Para um DataFrame, no lugar do operador and devemos usar do operador '&', assim como no lugar do operador or temos de usar o operador '|'.

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 #print(data['a'])
7 print(data[(data['a'] < 0) & (data['b'] > 1)])
8
```

	a	b	c	d	e
1	-0.268008	-0.228818	-0.083881	1.258773	1.629901
2	0.906926	-0.764981	0.981643	-0.258335	-1.142479
3	0.504328	-1.309354	-1.667187	-0.618108	-1.193881
4	-0.523564	-0.125894	-0.748558	-0.790971	-0.534490
5	-1.419431	0.776450	-1.595538	1.597824	-0.036342
6	0.649353	-1.002809	-0.773971	-0.057788	0.125344

Empty DataFrame
Columns: [a, b, c, d, e]
Index: []

Para que tais conceitos façam mais sentido, vamos para a prática. Novamente, apenas para evitar inconsistências em nossos exemplos, um novo DataFrame é criado nos mesmos moldes dos DataFrames anteriores.

Em seguida, por meio da função print() exibimos em tela o conteúdo de nosso DataFrame associado a variável data.

Na sequência, diretamente como parâmetro de nossa função print(), criamos uma estrutura condicional composta onde os dados de data na posição ['a'] forem menores que zero e (and &) os dados de data na posição

['b'] forem maiores que 1, um novo DataFrame é gerado, atualizando os dados originais de data.

Observe que a expressão completa foi escrita de modo que a variável data terá sua estrutura atualizada com base nas expressões condicionais declaradas, porém o que foi retornado é um DataFrame vazio, pois ao menos uma das condições impostas não foi atingida.

```
1 data2 = data[(data['a'] < 0) & (data['b'] > 1)]
2
3 print(data2)
4
```

Empty DataFrame
Columns: [a, b, c, d, e]
Index: []

Isolando a expressão condicional e atribuindo a mesma a uma variável, nesse caso de nome data2, exibindo em tela seu conteúdo via print() é possível ver que o conteúdo de data2 é de fato um DataFrame vazio. Diferente de outros contextos onde o DataFrame pode ter seus valores padrão instanciados como NaN, aqui temos um DataFrame realmente vazio de elementos.

```

1 data = pd.DataFrame(data = np.random.randn(6,5),
2                       index = [1,2,3,4,5,6],
3                       columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6 #print(data['a'])
7 print(data[(data['a'] < 0) | (data['b'] > 1)])
8

```

	a	b	c	d	e
1	0.570724	2.130538	1.368224	-1.923536	-0.917176
2	-0.768732	-0.040108	-1.302378	0.490337	-3.798542
3	-0.724144	0.891747	0.844517	0.218337	2.139444
4	0.289821	1.067948	0.115257	0.005286	0.305404
5	-2.130968	0.163764	1.224822	1.430992	0.950014
6	1.508415	0.758609	0.964695	-0.408959	0.505022

	a	b	c	d	e
1	0.570724	2.130538	1.368224	-1.923536	-0.917176
2	-0.768732	-0.040108	-1.302378	0.490337	-3.798542
3	-0.724144	0.891747	0.844517	0.218337	2.139444
4	0.289821	1.067948	0.115257	0.005286	0.305404
5	-2.130968	0.163764	1.224822	1.430992	0.950014

Usando do operador `|`, equivalente a or, podemos criar estruturas condicionais compostas mais flexíveis, onde bastando uma das expressões ser validada como verdadeira para que todo o bloco de código atrelado a mesma seja executado.

Em cima do mesmo exemplo anterior, apenas alterando o operador de `&` para `|` é possível notar que o conteúdo exibido em tela pela função `print()` agora é um DataFrame com dados distintos, apenas descartando destes dados os dados validados como False, mantendo no DataFrame os dados validados como True em uma das expressões condicionais.

No exemplo, o DataFrame retornado é um novo DataFrame sem os elementos da coluna 'a' que forem menores que 0 e sem os elementos da coluna 'b' que forem maiores que 1 de acordo com as respectivas condições impostas anteriormente.

Atualizando um DataFrame de acordo com uma condicional

```
data_positivos = data > 0
data = data[data_positivos]

print(data)
```

a	b	c	d	e
0.570724	2.130538	1.368224	NaN	NaN
NaN	NaN	NaN	0.490337	NaN
NaN	0.891747	0.844517	0.218337	2.139444
0.289821	1.067948	0.115257	0.005286	0.305404
NaN	0.163764	1.224822	1.430992	0.950014
1.508415	0.758609	0.964695	NaN	0.505022

Para alguns contextos específicos podemos atualizar os dados de um DataFrame usando de estruturas condicionais diretamente sobre a variável a qual o DataFrame está associado.

Este processo contorna um problema visto anteriormente onde de acordo com algumas estruturas condicionais um DataFrame tinha seus dados totalmente apagados.

Usando de uma expressão condicional simples aplicada à uma variável, temos a vantagem de gerar um novo DataFrame a partir desse processo, de modo que onde houverem elementos os quais não forem validados pela expressão condicional, seus espaços simplesmente serão preenchidos com NaN.

Em nosso exemplo, criamos uma variável de nome `data_positivos`, que recebe como atributo os elementos da variável `data` (nosso DataFrame) os quais forem maiores que zero. Nesta etapa é como se definíssemos uma regra a ser aplicada para atualizar nosso DataFrame original.

Em seguida, instanciando nosso DataFrame original via variável `data`, podemos atualizar a mesma com `data` na posição `[data_positivos]`, dessa forma, a expressão condicional será aplicada a cada elemento da composição de nosso DataFrame, retornando seu valor para sua posição

original quando o mesmo for validado como True, da mesma forma, retornando NaN para elementos os quais forem validados como False de acordo com a condicional.

INDEXAÇÃO DE DATAFRAMES

Manipulando o índice de um DataFrame

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2                       index = [1,2,3,4,5,6],
3                       columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6
7 data = data.reset_index()
8
9 print(data)
10
```

	a	b	c	d	e
1	-1.752488	-0.566416	0.995477	1.150715	1.176919
2	0.976744	-0.644148	-0.353794	-0.940721	2.049299
3	-1.089132	0.179902	-0.012432	1.429896	-0.734637
4	-1.395264	-0.216148	1.277971	-0.166671	0.781741
5	-0.569799	0.666268	-0.358426	-0.744800	-0.042683
6	1.688413	-1.106840	0.976592	-1.273916	0.058855

index	a	b	c	d	e	
0	1	-1.752488	-0.566416	0.995477	1.150715	1.176919
1	2	0.976744	-0.644148	-0.353794	-0.940721	2.049299
2	3	-1.089132	0.179902	-0.012432	1.429896	-0.734637
3	4	-1.395264	-0.216148	1.277971	-0.166671	0.781741
4	5	-0.569799	0.666268	-0.358426	-0.744800	-0.042683
5	6	1.688413	-1.106840	0.976592	-1.273916	0.058855

Em tópicos anteriores vimos que uma parte essencial de um DataFrame é seu índice, e o mesmo pode ser manipulado de diversas formas de acordo com o propósito da aplicação.

Para certos contextos específicos o índice de um DataFrame pode ser alterado livremente assim como resetado para suas configurações originais. Este processo também pode ser feito de diversas formas, sendo uma delas fazendo o uso de uma função interna da biblioteca Pandas dedicada a este propósito e com uma característica interessante, a de trazer novamente à tona o índice interno de nosso DataFrame.

Atualizando nossa variável `data` (`DataFrame`), aplicando sobre seus próprios dados o método `reset_index()`, sem parâmetros mesmo, é gerado um índice padrão, iniciado em 0 como todo índice em Python, transformando o índice antigo em mais uma coluna de nosso `DataFrame` para que assim não se altere a consistência dos dados.

Exibindo em tela via função `print()` o conteúdo da variável `data` antes das modificações é possível ver seu índice personalizado.

Da mesma forma exibindo em tela via `print()` o conteúdo de `data` após processamento do método `reset_index()`, é possível notar que agora nosso `DataFrame` possui um índice padrão, seguido de uma coluna de nome `index` com o índice personalizado definido anteriormente, assim como todos os demais dados dispostos em suas posições corretas.

```

1 data = pd.DataFrame(data = np.random.randn(6,5),
2                     index = [1,2,3,4,5,6],
3                     columns = ['a', 'b', 'c', 'd', 'e'])
4
5 coluna_V = 'v1 v2 v3 v4 v5 v6'.split()
6 data['IdV'] = coluna_V
7
8 print(data)
9
10 data = data.set_index('IdV')
11
12 print(data)
13

```

	a	b	c	d	e	IdV
1	1.937015	-0.923351	0.098605	-2.186975	0.204915	v1
2	-0.969770	0.103180	-0.165966	-0.276867	0.431477	v2
3	0.615997	0.215184	0.661466	-0.238085	0.805739	v3
4	0.502551	-0.534448	-1.297095	-2.239641	1.434370	v4
5	0.401690	-0.819280	0.149470	0.339620	0.753718	v5
6	1.023197	0.141539	1.374241	-1.612818	0.869067	v6

IdV	a	b	c	d	e
v1	1.937015	-0.923351	0.098605	-2.186975	0.204915
v2	-0.969770	0.103180	-0.165966	-0.276867	0.431477
v3	0.615997	0.215184	0.661466	-0.238085	0.805739
v4	0.502551	-0.534448	-1.297095	-2.239641	1.434370
v5	0.401690	-0.819280	0.149470	0.339620	0.753718
v6	1.023197	0.141539	1.374241	-1.612818	0.869067

Outra possibilidade interessante no que diz respeito a manipulação do índice de um DataFrame é o fato de podermos atribuir como índice toda e qualquer coluna já existente em nosso DataFrame. Este processo é feito por meio do método `set_index()` por sua vez parametrizado com o nome de cabeçalho da coluna a ser transformada em índice.

Em nosso exemplo inicialmente geramos um DataFrame via método construtor de DataFrames `pd.DataFrame()`, dando a ele algumas características.

Em seguida é criada uma variável de nome `coluna_V` que recebe uma string que terá seus elementos desmembrados através da função `split()`.

Na sequência adicionamos uma nova coluna a nosso DataFrame instanciando data na posição ['Idv'] até então inexistente, que será gerada com os dados oriundos de coluna_V.

Logo após essa alteração, instanciando novamente a variável data, aplicando sob a mesma o método `set_index()`, parametrizado com a string 'Idv', estamos definindo manualmente que a coluna de nome Idv passa a ser o índice de nosso DataFrame.

Exibindo o antes e depois das alterações de nosso DataFrame por meio de nossa função `print()`, é possível ver que de fato a última forma de nosso DataFrame é usando Idv como índice para o mesmo.

Índices multinível

```
1  indice1 = ['A1', 'A1', 'A1', 'A2', 'A2', 'A2']
2  indice2 = [1,2,3,1,2,3]
3
4  indice3 = list(zip(indice1, indice2))
5  indice3 = pd.MultiIndex.from_tuples(indice3)
6
7  print(indice3)
8
9  data8 = pd.DataFrame(data = np.random.randn(6,2),
10 | | | | | | | | | index = indice3)
11
12 print(data8)
13
```

```
MultiIndex([('A1', 1),
            ('A1', 2),
            ('A1', 3),
            ('A2', 1),
            ('A2', 2),
            ('A2', 3)],
           )
           0          1
A1 1 -1.667255 -0.614196
   2  1.123413  1.910823
   3 -0.663782 -0.206900
A2 1 -0.053067  0.129858
   2 -0.456319  0.359771
   3  0.352184 -1.535724
```

Avançando nossos estudos com índices, não podemos deixar de ver sobre os chamados índices multinível, pois para algumas aplicações será necessário mapear e indexar nossos DataFrames de acordo com suas camadas de dados ordenados.

Lembrando que tudo o que foi visto até então também é aplicável para os DataFrames que estaremos usando na sequência. Todo material deste livro foi estruturado de modo a usarmos de ferramentas e recursos de

forma progressiva, de forma que a cada tópico que avançamos estamos somente a somar mais conteúdo a nossa bagagem de conhecimento.

Em teoria, assim como em uma planilha Excel temos diferentes guias, cada uma com suas respectivas tabelas, a biblioteca Pandas incorporou a seu rol de ferramentas funções que adicionavam suporte a estes tipos de estruturas de dados em nossos DataFrames.

Para criar um índice multinível, para camadas de dados em nossos DataFrames, devemos em primeiro lugar ter em mente quantas camadas serão mapeadas, para que assim possamos gerar índices funcionais.

Partindo para a prática, inicialmente criamos uma variável de nome `indice1`, que recebe como atributo uma lista com 6 elementos em formato de string.

Da mesma forma é criada outra variável de nome `indice2` que por sua vez possui como atributo uma lista composta de 6 elementos numéricos.

Aqui já podemos deduzir que tais listas serão transformadas em camadas de nosso índice, para isso, antes mesmo de gerar nosso DataFrame devemos associar tais listas de modo que seus elementos sejam equiparados de acordo com algum critério.

Para nosso exemplo inicial, estaremos trabalhando com um índice de apenas dois níveis, porém, a lógica será a mesma independentemente do número de camadas a serem mapeadas para índice.

Se tratando então de apenas duas camadas, podemos associar os elementos de nossas listas criando uma lista de tuplas. O método mais fácil para gerar uma lista de tuplas é por meio da função `zip()`. É então criada uma variável de nome `indice3` que recebe como parâmetro em forma de lista via construtor de listas `list()` a junção dos elementos de `indice1` e `indice2` via `zip()`.

A partir desse ponto já podemos gerar o índice em questão. Para isso, instanciando a variável `indice3`, chamando a função `pd.MultiIndex.from_tuples()`, por sua vez parametrizada com os dados da própria variável `indice3`, internamente tal função usará de uma série de métricas para criar as referencias para as devidas camadas de índice.

Importante salientar que, apenas para esse exemplo, estamos criando um índice multinível a partir de uma tupla, porém, consultando a ajuda inline de `pd.MultiIndex` é possível ver diversos modos de geração de índices a partir de diferentes estrutura de dados.

Uma vez gerado nosso índice, podemos exibir seu conteúdo através da função `print()`, pois o mesmo é uma estrutura de dados independente, que será atrelada a nosso `DataFrame`.

Na sequência criamos uma variável de nome `data8` que via método construtor de `DataFrames` gera um novo `DataFrame` de 6 linhas por 2 colunas, repassando para o parâmetro nomeado `index` o conteúdo da variável `indice3`, nosso índice multinível criado anteriormente.

Exibindo o conteúdo de nossa variável `data8` via função `print()`, é possível ver que de fato temos um `DataFrame` com seus dados distribuídos de modo que suas linhas e colunas respeitam uma hierarquia de índice multinível.

```
1 indice1 = ['Nivel1', 'Nivel1', 'Nivel1', 'Nivel2', 'Nivel2', 'Nivel2']
2 indice2 = ['SubNivel1', 'SubNivel2', 'SubNivel3', 'SubNivel1', 'SubNivel2', 'SubNivel3']
3
4 indice3 = list(zip(indice1, indice2))
5 indice3 = pd.MultiIndex.from_tuples(indice3)
6
7 data8 = pd.DataFrame(data = np.random.randn(6,2),
8                       index = indice3,
9                       columns = ['Coluna1', 'Coluna2'])
10
11 print(data8)
12
```

		Coluna1	Coluna2
Nivel1	SubNivel1	0.400599	-0.618382
	SubNivel2	0.779403	1.153734
	SubNivel3	-0.325612	-0.531926
Nivel2	SubNivel1	-1.300274	-0.130670
	SubNivel2	-0.211998	-1.434109
	SubNivel3	0.701236	1.504282

Lembrando que uma boa prática de programação é prezarmos pela legibilidade de nossos códigos. Sendo assim, uma vez que estamos

manipulando índices personalizados, podemos usar de nomes os quais facilitam a interpretação.

Usando do mesmo exemplo anterior, apenas alterando os nomes usados para o índice multinível de siglas para nomes usuais, tornamos nosso índice mais legível.

Dessa forma podemos visualizar facilmente que, por exemplo, Nivel1 possui uma segunda camada de dados representada por Subnivel1, Subnivel2 e Subnivel3. Posteriormente iremos entender a lógica de manipulação dos dados do DataFrame por meio deste tipo de índice.

```
1 print(data8.loc['Nivel2'])
2
```

	Coluna1	Coluna2
SubNivel1	-1.300274	-0.130670
SubNivel2	-0.211998	-1.434109
SubNivel3	0.701236	1.504282

Conforme mencionado anteriormente, uma vez que podemos usar da notação de manipulação de dados de lista em nossos DataFrames, podemos facilmente instanciar determinados elementos do mesmo por meio de métodos como `loc[]` entre outros.

Apenas como exemplo, repassando como parâmetro de nossa função `print() data8`, aplicando sobre a mesma o método `loc[]` por sua vez alimentado com a string 'Nivel2', o retorno será todo e qualquer elemento que estiver atrelado a esse nível.

Nesse caso, `data8.loc['Nivel2']` retornará todos elementos de Subnivel1, Subnivel2 e Subnivel3, independentemente de suas colunas.

```

1 print(data8.loc['Nivel2'].loc['SubNivel3'])
2

```

Coluna1	0.701236
Coluna2	1.504282

Name: SubNivel3, dtype: float64

Dentro da mesma lógica, para acessar um subnível específico podemos simplesmente usar duas vezes `loc[]` (e quantas vezes fosse necessário de acordo com o número de camadas / níveis).

Por exemplo, exibindo em tela de `data8` as instâncias `.loc['Nivel2'].loc['Subnivel3']` é retornado uma Serie composta dos elementos apenas deste nível.

Note que por se tratar da apresentação de uma Serie, linhas foram convertidas para colunas, mas apenas para apresentação destes dados, nenhuma alteração permanente ocorre neste processo.

```

1 print(data8.xs('Nivel2'))
2

```

	Coluna1	Coluna2
SubNivel1	-1.300274	-0.130670
SubNivel2	-0.211998	-1.434109
SubNivel3	0.701236	1.504282

Uma forma alternativa de acessar, tanto um nível quanto um subnível de um índice multinível (independentemente do número de níveis) é usando do método `xs()`.

Aplicando tal método diretamente sobre uma variável a qual possua como tipo de dado um `DataFrame`, basta parametrizar o método `xs()` com o nome do nível / subnível para assim retornar seus dados.

```

1  indice1 = ['Nivel1', 'Nivel1', 'Nivel1', 'Nivel2', 'Nivel2', 'Niv
2  indice2 = ['SubNivel1', 'SubNivel2', 'SubNivel3', 'SubNivel1', 'SubNi
3
4  indice3 = list(zip(indice1, indice2))
5  indice3 = pd.MultiIndex.from_tuples(indice3)
6
7  data8 = pd.DataFrame(data = np.random.randn(6,2),
8  | | | | | | | | | | index = indice3,
9  | | | | | | | | | | columns = ['Coluna1', 'Coluna2'])
10
11  data8.index.names = ['NIVEIS', 'SUBNIVEIS']
12
13  print(data8)
14

```

		Coluna1	Coluna2
NIVEIS	SUBNIVEIS		
Nivel1	SubNivel1	1.095250	0.145660
	SubNivel2	2.082887	1.153097
	SubNivel3	-0.319348	-0.060429
Nivel2	SubNivel1	-0.817612	0.200159
	SubNivel2	0.536399	0.229161
	SubNivel3	-0.682792	-0.310040

Outra possibilidade ainda se tratando da organização e legibilidade de um índice é atribuir para o mesmo nomes funcionais.

Usando do mesmo exemplo anterior, uma vez criado nosso DataFrame com seu índice multinível, é possível aplicar o método `index.names` para a variável a qual instancia nosso DataFrame, bastando passar para tal método uma lista com os nomes funcionais que quisermos, desde que o número de nomes funcionais seja compatível com o número de níveis / subníveis.

Exibindo via `print()` nosso DataFrame é possível ver que logo acima de `Nivel1` e de `Subnivel1` existe um cabeçalho composto por `NIVEIS` e `SUBNIVEIS` conforme esperado.

```

1 data = pd.DataFrame({'A': ['1', '1', '1', '2', '2', '2'],
2                       'B': ['3', '3', '2', '2', '1', '1'],
3                       'C': ['x', 'y', 'x', 'y', 'x', 'y'],
4                       'D': [1, 2, 3, 9, 8, 7]})
5
6 data.pivot_table(values = 'D', index = ['A', 'B'], columns = ['C'])
7
8 print(data)
9

```

	A	B	C	D
0	1	3	x	1
1	1	3	y	2
2	1	2	x	3
3	2	2	y	9
4	2	1	x	8
5	2	1	y	7

Encerrando este capítulo, uma última possibilidade a se explorar, embora pouco utilizada na prática, é a de gerar um índice multinível por meio do método `pivot_table()`, onde podemos transformar dados do próprio DataFrame como níveis e subníveis de um índice.

Para nosso exemplo, temos um DataFrame criado da maneira tradicional, com seus dados já declarados em forma de dicionário, definindo assim claramente quais serão as colunas (chaves do dicionário) e as linhas (valores do dicionário).

A partir dessa estrutura podemos usar da função `pivot_table()`, que por sua vez exige a parametrização obrigatória para `values`, `index` e `columns`. Nesse caso, como índice definimos que o mesmo será gerado a partir de 'A' e 'B', que aqui se transformarão em nível e subnível, respectivamente. Do mesmo modo para `columns` repassamos 'C', e como valores os elementos de 'D'.

Dessa forma, embora o DataFrame mantenha seu índice interno ativo, visível e funcional, as demais estruturas de dados estarão mapeadas para `index`, `columns` e `values` de modo que podemos acessar seus elementos por meio da notação padrão `loc[]` fazendo simples referência a seus nomes de cabeçalho de coluna.

ENTRADA DE DADOS

Importando dados de um arquivo para um DataFrame

No que diz respeito à aplicação das ferramentas da biblioteca Pandas, algo mais próximo da realidade é importarmos certas bases de dados para realizar a manipulação e tratamento de seus dados. Essas bases de dados podem ser das mais diversas origens, desde que seus dados estejam dispostos em tabelas / matrizes, organizados em linhas e colunas.

```
1 data = pd.read_csv('/content/sample_data/mnist_test.csv')
2
3 print(data)
4
```

	7	0	0.1	0.2	0.3	0.4	...	0.662	0.663	0.664	0.665	0.666
0	2	0	0	0	0	0	...	0	0	0	0	0
1	1	0	0	0	0	0	...	0	0	0	0	0
2	0	0	0	0	0	0	...	0	0	0	0	0
3	4	0	0	0	0	0	...	0	0	0	0	0
4	1	0	0	0	0	0	...	0	0	0	0	0
...
9994	2	0	0	0	0	0	...	0	0	0	0	0
9995	3	0	0	0	0	0	...	0	0	0	0	0
9996	4	0	0	0	0	0	...	0	0	0	0	0
9997	5	0	0	0	0	0	...	0	0	0	0	0
9998	6	0	0	0	0	0	...	0	0	0	0	0

[9999 rows x 785 columns]

A biblioteca Pandas possui uma série de ferramentas dedicadas à importação de arquivos de várias origens, processando os mesmos por uma série de mecanismos internos garantindo a integridade dos dados assim como a compatibilidade para tratamento dos mesmos.

Dos tipos de arquivos mais comuns de serem importados para nossas estruturas de código estão as planilhas Excel e seus equivalentes de outras suítes Office.

Importar dados a partir de um arquivo Excel (extensão .csv) é bastante simples, bastando usar do método correto, associando os dados

importados a uma variável. No próprio processo de importação tais dados serão mapeados e convertidos para a estrutura de um DataFrame.

Em nosso exemplo, é criada uma variável de nome `data`, que por sua vez instancia e inicializa a função `pd.read_csv()`, parametrizando tal função com o caminho de onde os dados serão importados em formato de string.

Lembrando que de acordo com o contexto o caminho a ser usado pode ser relativo (ex: `/pasta/arquivo.csv`) ou absoluto (ex: `c:/Usuários/Fernando/Documentos/pasta/arquivo.csv`).

Exibindo em tela o conteúdo da variável `data` por meio de nossa velha conhecida função `print()`, é nos exibido um DataFrame, nesse caso apenas como exemplo, um DataSet muito conhecido de estudantes de ciências de dados chamado `mnist`, composto de 9999 linhas e 785 colunas de dados.

```
data = pd.re
read_clipboard
read_csv
read_excel
read_feather
read_fwf
read_gbq
read_hdf
read_html
read_json
read_orc
read_parquet
read_pickle
read_sas
read_spss
read_sql
read_sql_query
read_sql_table
read_stata
read_table
```






















Outras possíveis fontes para importação.

```
1 data.to_csv('base_atualizada.csv')
2
```

Realizando o processo inverso, exportando de nosso DataFrame para um arquivo, também temos à disposição algumas ferramentas de exportação de acordo com o tipo de dado.

Apenas como exemplo, diretamente sobre nossa variável `data`, aplicando o método `to_csv()` repassando como parâmetro um nome de arquivo em formato string e extensão `.csv`, um arquivo será devidamente gerado, arquivo este com todas as propriedades de um arquivo Excel.

`data.to`

-  `to_pickle`
-  `to_clipboard`
-  `to_csv`
-  `to_dict`
-  `to_excel`
-  `to_feather`
-  `to_gbq`
-  `to_hdf`
-  `to_html`
-  `to_json`
-  `to_latex`
-  `to_markdown`
-  `to_numpy`
-  `to_parquet`
-  `to_period`
-  `to_records`
-  `to_sql`
-  `to_stata`
-  `to_string`
-  `to_timestamp`
-  `to_xarray`

Outros possíveis formatos de exportação.

TRATAMENTO DE DADOS

Tratando dados ausentes / faltantes

```
1 data = {'Cidades': ['Porto Alegre', 'Curitiba', np.nan],
2         'Estados': ['Paraná', np.nan, 'São Paulo'],
3         'Países': ['Brasil', np.nan, np.nan]}
4
5 data = pd.DataFrame(data)
6
7 print(data)
8
```

	Cidades	Estados	Países
0	Porto Alegre	Paraná	Brasil
1	Curitiba	NaN	NaN
2	NaN	São Paulo	NaN

Um dos principais problemas a se contornar em uma base de dados são seus dados ausentes / faltantes, haja visto que independentemente da aplicação, a integridade dos dados de base é de suma importância para que a partir dos mesmos se faça ciência.

Para este e para os tópicos subsequentes, vamos partir de uma base de dados comum criada para esses exemplos (embora tudo o que será visto aqui se aplique a toda e qualquer base de dados a qual estaremos usando em estrutura de DataFrame), com alguns elementos problemáticos a serem tratados, para que de forma prática possamos entender o uso de algumas ferramentas específicas para tratamento de dados.

Em nosso exemplo, inicialmente temos uma variável de nome data que recebe como atributo um dicionário onde constam como chaves 'Cidades', 'Estados' e 'Países', sendo que como valores para estas chaves temos um misto entre dados presentes e dados faltantes.

Em seguida geramos um DataFrame a partir destes dados como já feito várias vezes anteriormente, usando do método `pd.DataFrame()`.

Por fim, exibindo em tela o conteúdo de data via função `print()` é possível ver nosso DataFrame, composto de 3 linhas e 3 colunas, onde temos NaN representando a ausência de dados em alguns campos.

```

1 data = data.dropna()
2
3 print(data)
4

```

	Cidades	Estados	Países
0	Porto Alegre	Paraná	Brasil

Uma das formas mais básicas que temos para remover dados de nosso DataFrame associados a NaN é por meio da função `dropna()`.

O método `dropna()` quando aplicado diretamente sobre uma variável, irá varrer todo seu conteúdo, e se tratando de um DataFrame, a mesma removerá todas as linhas e colunas onde houverem dados ausentes.

Repare que em nosso exemplo, em algumas posições das linhas 1 e 2 (número de índice 1 e 2) existiam dados faltantes, em função disso toda a referente linha foi removida.

Raciocine que tanto em grandes quanto em pequenas bases de dados, o fato de excluir linhas compostas de elementos válidos e de elementos faltantes pode impactar negativamente a integridade dos dados. Supondo que uma base de dados recuperada com muitos dados corrompidos seja tratada dessa forma, os poucos dados válidos restantes serão removidos neste tipo de tratamento, acarretando em perdas de dados importantes.

```

1 data = data.dropna(thresh = 2)
2
3 print(data)
4

```

	Cidades	Estados	Países
0	Porto Alegre	Paraná	Brasil

Uma forma de usar de `dropna()` de maneira menos agressiva em nossa base de dados é definir um parâmetro `thresh`, onde você pode

especificar um número mínimo de elementos faltantes a ser considerado para que aí sim toda aquela linha seja removida.

Em nosso exemplo, definindo thresh como 2 faria com que apenas as linhas com 2 ou mais elementos faltantes fossem de fato removidas.

```
1 data = {'Cidades': ['Porto Alegre', 'Curitiba', np.nan],
2         'Estados': ['Paraná', np.nan, 'São Paulo'],
3         'Países': ['Brasil', np.nan, np.nan]}
4
5 data = pd.DataFrame(data)
6
7 data['Cidades'] = data['Cidades'].replace({np.nan: 'Santa Maria'})
8
9 print(data)
10
```

	Cidades	Estados	Países
0	Porto Alegre	Paraná	Brasil
1	Curitiba	NaN	NaN
2	Santa Maria	São Paulo	NaN

Apenas lembrando que todo e qualquer método aplicável a um contêiner de dados pode ser usado para estas estruturas de dados perfeitamente.

O tratamento de um dado faltante poderia, por exemplo, substituir tal elemento por outro via função `replace()`, porém o que buscamos como ferramenta são métodos os quais realizem o tratamento de todos os elementos os quais por algum critério ou característica tenha de ser alterado.

```

1 data = {'Cidades': ['Porto Alegre', 'Curitiba', np.nan],
2         'Estados': ['Paraná', np.nan, 'São Paulo'],
3         'Países': ['Brasil', np.nan, np.nan]}
4
5 data = pd.DataFrame(data)
6
7 data = data.fillna(value = 'Dados Ausentes')
8
9 print(data)
10

```

	Cidades	Estados	Países
0	Porto Alegre	Paraná	Brasil
1	Curitiba	Dados Ausentes	Dados Ausentes
2	Dados Ausentes	São Paulo	Dados Ausentes

Um método melhor elaborado, próprio da biblioteca Pandas, em substituição ao `dropna()` é o método `fillna()`, onde por meio deste, podemos tratar dados faltantes substituindo o marcador de dado faltante `Nan` por algum dado/valor padrão, sendo assim uma alternativa muito mais interessante do que a anterior.

Para nosso exemplo, mantendo exatamente a mesma base de dados anterior, agora atualizamos nossa variável `data`, aplicando sobre a mesma o método `fillna()` que recebe como parâmetro para `value` um dado/valor padrão, em nosso caso, a string `'Dados Ausentes'`.

Através da função `print()` por sua vez parametrizada com a variável `data`, é exibido em tela seu conteúdo. Nesse caso nosso `DataFrame`, onde é possível ver que todos os elementos faltantes foram devidamente preenchidos com `Dados Ausentes`.


```

1 data = {'1º Semestre': [8.2, 8.0, np.nan],
2         '2º Semestre': [7.7, np.nan, 8.0],
3         'Recuperação': [7.9, np.nan, np.nan]}
4
5 data = pd.DataFrame(data)
6
7 print(data)
8
9 data['1º Semestre'].fillna(value = data['1º Semestre'].mean())
10 data['2º Semestre'].fillna(value = data['2º Semestre'].mean())
11 data['Recuperação'].fillna(value = data['Recuperação'].mean())
12
13 print(data)
14

```

	1º Semestre	2º Semestre	Recuperação
0	8.2	7.7	7.9
1	8.0	NaN	NaN
2	NaN	8.0	NaN

	1º Semestre	2º Semestre	Recuperação
0	8.2	7.70	7.9
1	8.0	7.85	7.9
2	8.1	8.00	7.9

Usando do método `fillna()` para tratamento de dados em um `DataFrame` composto de dados numéricos, podemos usar de toda e qualquer operação matemática sobre tais valores.

Apenas como exemplo, uma vez que temos um `DataFrame` associado a variável `data`, composto de 3 linhas e 3 colunas, sendo o conteúdo das linhas apenas dados numéricos (e nesse caso, dados faltantes), podemos aplicar sobre tais dados alguma função matemática como, por exemplo, preencher os campos faltantes com a média dos valores antecessores e sucessores por meio do método `mean()`.

Sendo assim, exibindo em tela nosso `DataFrame` inicial é possível constatar os campos de elementos ausentes, assim como após a aplicação do método `mean()` para as respectivas colunas, os espaços foram preenchidos com as médias das notas, uma solução simples porém eficiente para tratar

os dados de origem sem alterar bruscamente a integridade dos dados já presentes quando associados aos novos dados.

```
1 data = {'1º Semestre': [8.2, 8.0, np.nan],
2         '2º Semestre': [7.7, np.nan, 8.0],
3         'Recuperação': [7.9, np.nan, np.nan]}
4
5 data = pd.DataFrame(data)
6
7 print(data)
8
9 data = data.fillna(method = 'ffill')
10
11 print(data)
12
```

	1º Semestre	2º Semestre	Recuperação
0	8.2	7.7	7.9
1	8.0	NaN	NaN
2	NaN	8.0	NaN

	1º Semestre	2º Semestre	Recuperação
0	8.2	7.7	7.9
1	8.0	7.7	7.9
2	8.0	8.0	7.9

Outra possível solução, bastante empregada em bases de dados numéricas, é o preenchimento de dados faltantes pela simples replicação do dado antecessor, e isto é feito apenas usando de um parâmetro nomeado do método `fillna()` chamado `method`, que quando definido como `'ffill'` cumpre esse papel.

No exemplo, atualizando nosso `DataFrame` atrelado a variável `data`, aplicando sobre a mesma o método `fillna()` por sua vez parametrizado com `method = 'ffill'`, temos um simples tratamento onde os elementos demarcados com `NaN` são substituídos pelos valores antecessores já presentes.

Apenas salientando que quando dizemos antecessores, no contexto de um `DataFrame` estamos falando que daquela `Serie` (daquela coluna) o

elemento antecessor é o que está imediatamente acima do elemento atual.

```
1 data = pd.read_csv('/content/sample_data/mnist_test.csv')
2
3 data = pd.DataFrame(data)
4
5 print(data.isnull())
6
```

	7	0	0.1	0.2	0.3	...	0.663	0.664	0.665	0.666
0	False	False	False	False	False	...	False	False	False	False
1	False	False	False	False	False	...	False	False	False	False
2	False	False	False	False	False	...	False	False	False	False
3	False	False	False	False	False	...	False	False	False	False
4	False	False	False	False	False	...	False	False	False	False
...
9994	False	False	False	False	False	...	False	False	False	False
9995	False	False	False	False	False	...	False	False	False	False
9996	False	False	False	False	False	...	False	False	False	False
9997	False	False	False	False	False	...	False	False	False	False
9998	False	False	False	False	False	...	False	False	False	False

[9999 rows x 785 columns]

Podemos também realizar a verificação de elementos ausentes em uma base de dados a qual não conhecemos seus detalhes por meio da função `isnull()`.

A função `isnull()` é uma função embutida do Python justamente para verificar nos dados de qualquer tipo de contêiner de dados, se existem elementos faltantes em sua composição. Por padrão, a função `isnull()` quando aplicada a uma variável qualquer, retornará `True` quando encontrar algum campo com elemento ausente, e `False` para todos os elementos regulares.

Apenas para fins de exemplo, carregando novamente a base de dados `mnist`, atribuindo a mesma para uma variável de nome `data`, e aplicando sobre `data` o método `isnull()`, é retornado um `Dataframe` com todos os elementos dessa base de dados, demarcados com `True` e `False` conforme cada elemento da mesma é verificado e validado.

Agrupando dados de acordo com seu tipo

Levando em consideração a maneira como os dados em um DataFrame são mapeados, existem diversas possibilidades de manipulação dos mesmos conforme vimos ao longo dos capítulos.

Uma das abordagens muito úteis se tratando de dados é os agrupar conforme algum critério para que assim apliquemos alguns filtros ou funções.

Uma das formas de se trabalhar com agrupamentos personalizados de dados via biblioteca Pandas é usando de sua ferramenta GroupBy.

```
1 data = {'Vendedor': ['Ana', 'Paulo', 'Maria', 'Miguel', 'Rafael', 'Ana'],
2         'Item': ['Camisa Nike', 'Tênis Adidas', 'Camisa Nike',
3                 'Meias Speedo', 'Tênis Nike', 'Camisa Nike'],
4         'Valor': [109.90, 299.00, 115.90, 24.99, 289.90, 99.90]}
5
6 data = pd.DataFrame(data)
7
8 print(data)
9
```

	Vendedor	Item	Valor
0	Ana	Camisa Nike	109.90
1	Paulo	Tênis Adidas	299.00
2	Maria	Camisa Nike	115.90
3	Miguel	Meias Speedo	24.99
4	Rafael	Tênis Nike	289.90
5	Ana	Camisa Nike	99.90

Para esses exemplos, vamos fazer uso de um novo DataFrame, dessa vez criado a partir de um dicionário composto de três chaves, ‘Vendedor’, ‘Item’ e ‘Valor’, com suas respectivas listas de elementos no lugar de seus valores de dicionário.

Nos mesmos moldes dos exemplos anteriores, a partir desse dicionário geramos um DataFrame e a partir deste ponto podemos nos focar nos assuntos deste tópico.

Exibindo em tela via `print()` o conteúdo de `data`, nos é retornado um `DataFrame` com seus respectivos dados organizados por categoria, distribuídos em 3 colunas e 6 linhas.

```
1  itens = data.groupby('Item')
2
3  print(itens)
4
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f50b29bf450>
```

Partindo para o entendimento da lógica de agrupamento, uma vez que temos nosso `DataFrame`, vimos anteriormente alguns métodos para localização de colunas, linhas e elementos específicos destes campos para uso dos mesmos.

Uma forma mais elaborada de se trabalhar com esses dados é a partir do agrupamento dos mesmos de acordo com algum critério definido, como alguma característica relevante, assim, podemos extrair informações interessantes a partir dessas manipulações.

Para exemplo, é criada uma variável de nome `itens`, que receberá apenas os dados da “categoria” `itens` de nossa base de dados. Para isso, `itens` instancia e inicializa para `data` o método `groupby()`, parametrizando o mesmo com ‘`Item`’, nome de cabeçalho de coluna de nosso `DataFrame`.

A partir disso, a função irá gerar dados pelo parâmetro de agrupamento definido.

Através da função `print()`, exibindo em tela o conteúdo da variável `itens`, nos é retornado apenas a referência para o objeto criado com os dados filtrados, assim como seu identificador de objeto alocado em memória.

```
1 print(itens.sum())
2
```

	Valor
Item	
Camisa Nike	325.70
Meias Speedo	24.99
Tênis Adidas	299.00
Tênis Nike	289.90

A partir do momento que temos uma variável com dados agrupados a partir de nosso DataFrame, podemos aplicar funções sobre tais dados para que aí sim tenhamos retornos relevantes.

Diretamente via `print()`, usando como parâmetro `itens.sum()` podemos obter como retorno a soma dos valores numéricos associados aos dados agrupados.

De fato, como retorno nos é exibido em tela cada tipo de item com seu valor somado, como era o retorno esperado pois sobre tais dados aplicamos o método embutido do Python para soma `sum()`.

Isto ocorre pois dos dados agrupados, somente é possível somar valores numéricos, haja visto que o método `sum()` não realiza concatenação, e atuando sobre uma Serie ou DataFrame ignora qualquer coluna de texto.

```
1 print(itens.mean())
2
```

	Valor
Item	
Camisa Nike	108.566667
Meias Speedo	24.990000
Tênis Adidas	299.000000
Tênis Nike	289.900000

Podemos também usar de outros métodos já conhecidos, como por exemplo o método `mean()` que retornará a média dos valores, aqui,

respeitando o agrupamento de cada tipo de item.

```
1 print(itens.count())
2
```

	Vendedor	Valor
Item		
Camisa Nike	3	3
Meias Speedo	1	1
Tênis Adidas	1	1
Tênis Nike	1	1

Usando do próprio método embutido do sistema contador `count()`, é retornado um novo DataFrame representando de acordo com cada tipo de item sua quantidade agrupada.

```
1 print(itens.describe())
2
```

	Valor			...			
	count	mean	std	...	50%	75%	max
Item				...			
Camisa Nike	3.0	108.566667	8.082904	...	109.90	112.90	115.90
Meias Speedo	1.0	24.990000	NaN	...	24.99	24.99	24.99
Tênis Adidas	1.0	299.000000	NaN	...	299.00	299.00	299.00
Tênis Nike	1.0	289.900000	NaN	...	289.90	289.90	289.90

[4 rows x 8 columns]

Mesmo um objeto gerado como agrupador de certos tipos de dados de um DataFrame pode possuir algumas características descritivas geradas pelo Pandas.

Aplicando sobre a variável `itens` o método `describe()` nos é retornado de acordo com o agrupamento dos itens alguns dados baseados em seus valores numéricos, que aqui podemos contextualizar como média de preço, menor preço, 25%, 50% e 75% do preço, maior preço, etc...


```
1 vendedores = data.groupby('Vendedor')
2
3 print(vendedores.sum().loc['Ana'])
4
```

```
Valor      209.8
Name: Ana, dtype: float64
```

Por fim, agrupando itens de acordo com características textuais que podem ser, por exemplo, nomes de vendedores em uma base de dados, podemos combinar outros métodos explorados anteriormente para, a partir de um agrupamento, obter novas informações.

Como exemplo, declarada uma variável de nome `vendedores` que recebe o agrupamento de acordo com o critério ‘Vendedor’ de nosso `DataFrame` via `groupby()`, podemos combinar alguns métodos para extrair dados relevantes.

Por exemplo, diretamente como parâmetro de nossa função `print()`, aplicando sobre a variável `vendedores` o método `sum()` para a posição `['Ana']`, iremos obter como retorno exibido em tela apenas os valores das vendas atribuídos a Ana.

Novamente, cada uma das possibilidades exploradas nos tópicos anteriores se aplica aqui também, apenas não replicaremos os mesmos métodos por uma questão de não tornar o conteúdo deste livro muito repetitivo. Realize testes sobre seus dados, praticando os meios e métodos explicados até o momento para sua base de dados atual.

MÉTODOS APLICADOS

Alterando o formato de um DataFrame

Uma das particularidades de um DataFrame é sua forma, haja visto que uma das características principais deste tipo de estrutura de dados é justamente a distribuição de seus elementos em formato de tabela / matriz de modo que temos linhas e colunas bem estabelecidas.

Se tratando do formato de um DataFrame, sua estrutura é mapeada de modo que o interpretador busca suas referências por meio de índices ordenados, em função disso, devemos tomar muito cuidado sempre que formos realizar algum tipo de manipulação que acarrete em alteração do formato do DataFrame.

Em outras palavras, até podemos alterar o formato estrutural de um DataFrame quando necessário, porém, certas regras referentes a sua proporção e distribuição de seus dados devem ser respeitadas a fim de evitar exceções.

```
1 data = pd.DataFrame(data = np.random.randn(6,5),
2 | | | | | | | | | | index = [1,2,3,4,5,6],
3 | | | | | | | | | | columns = ['a', 'b', 'c', 'd', 'e'])
4
5 print(data)
6
7 data = pd.melt(data)
8
9 print(data)
10
```

	a	b	c	d	e
1	-0.290964	0.830812	-1.412909	-0.414070	1.075445
2	1.341879	0.878129	0.341998	0.895333	0.720504
3	0.223654	1.324789	-1.844739	1.422179	0.611117
4	-1.840032	-1.365422	-0.249960	0.436028	1.621377
5	-0.446507	-0.359948	-0.374418	-0.663777	0.970123
6	0.537361	1.506146	-0.391321	1.262124	1.163186

	variable	value
0	a	-0.290964
1	a	1.341879
2	a	0.223654
3	a	-1.840032
4	a	-0.446507
5	a	0.537361
6	b	0.830812
7	b	0.878129
8	b	1.324789
9	b	-1.365422
10	b	-0.359948
11	b	1.506146
12	c	-1.412909
13	c	0.341998
14	c	-1.844739
15	c	-0.249960
16	c	-0.374418
17	c	-0.391321
18	d	-0.414070
19	d	0.895333
20	d	1.422179
21	d	0.436028
22	d	-0.663777
23	d	1.262124
24	e	1.075445
25	e	0.720504
26	e	0.611117
27	e	1.621377
28	e	0.970123
29	e	1.163186

Partindo para prática, como exemplo usamos novamente de um simples DataFrame gerado via construtor `pd.DataFrame()`, dimensionado em 6 linhas e 5 colunas, com números aleatórios gerados via `np.random.randn()`.

Exibindo o conteúdo de nosso DataFrame, repassando a variável `data` como parâmetro para `print()`, é possível identificar seu formato original de distribuição de elementos.

Atualizando a variável `data`, aplicando sobre a mesma o método `pd.melt()`, estamos transformando de forma proporcional linhas em colunas

e vice-versa.

Em nossa segunda função `print()`, exibindo o conteúdo de data como feito anteriormente, agora temos um novo `DataFrame` composto de 29 linhas e 2 colunas, além é claro, de um novo índice gerado para este novo `DataFrame`.

Concatenando dados de dois DataFrames

Anteriormente vimos que é possível aplicar toda e qualquer operação aritmética sobre os dados numéricos de um DataFrame. Também foi dito que desses operadores aritmético a função `sum()` em um DataFrame ignora completamente dados textuais, aplicando suas métricas apenas nos dados numéricos presentes.

Explorando as funcionalidades da biblioteca Pandas podemos ver que contornando essa situação temos o método `concat()`, que irá concatenar / somar os elementos de um DataFrame sem nenhuma distinção e sem sobrepor nenhum elemento.

Pela lógica de agregação de dados do método `pd.concat()`, concatenando elementos de dois ou mais DataFrames, usando da indexação correta, podemos fazer com que os mesmos sejam enfileirados de modo a formar uma só base de dados ao final do processo.

```

1  loja1 = {'Vendedor': ['Ana', 'Paulo', 'Maria', 'Miguel', 'Rafael', 'Ana'],
2          'Item': ['Camisa Nike', 'Tênis Adidas', 'Camisa Nike',
3                  'Meias Speedo', 'Tênis Nike', 'Camisa Nike'],
4          'Valor': [109.90, 299.00, 115.90, 24.99, 289.90, 99.90]}
5
6  loja2 = {'Vendedor': ['Ana', 'Fernando', 'Carlos', 'Carlos', 'Carlos', 'Tânia'],
7          'Item': ['Camisa Fila', 'Camisa Adidas', 'Camisa Nike',
8                  'Camisa Nike', 'Camisa Fila', 'Tênis Mizuno'],
9          'Valor': [109.90, 99.00, 115.90, 109.99, 89.90, 199.90]}
10
11 data1 = pd.DataFrame(loja1, index = [1,2,3,4,5,6])
12 data2 = pd.DataFrame(loja2, index = [7,8,9,10,11,12])
13
14 print(data1)
15 print(data2)
16

```

	Vendedor	Item	Valor
1	Ana	Camisa Nike	109.90
2	Paulo	Tênis Adidas	299.00
3	Maria	Camisa Nike	115.90
4	Miguel	Meias Speedo	24.99
5	Rafael	Tênis Nike	289.90
6	Ana	Camisa Nike	99.90

	Vendedor	Item	Valor
7	Ana	Camisa Fila	109.90
8	Fernando	Camisa Adidas	99.00
9	Carlos	Camisa Nike	115.90
10	Carlos	Camisa Nike	109.99
11	Carlos	Camisa Fila	89.90
12	Tânia	Tênis Mizuno	199.90

Para realizar a concatenação da maneira correta, vamos ao exemplo: Nesse caso, temos de início dois dicionários de nome loja1 e loja2, respectivamente.

Cada um desses dicionários possui em sua composição três chaves onde cada uma delas recebe uma lista de elementos como valores.

Em seguida criamos uma variável de nome data1 que instancia e inicializa a função `pd.DataFrame()`, em justaposição repassando como dados o conteúdo da variável loja1, seguido do parâmetro nomeado `index` que recebe uma lista composta de 6 elementos numéricos ordenados de 1 até 6.

Exatamente da mesma forma, data2 é criada e para a mesma é gerado um DataFrame a partir de loja2, via função `pd.DataFrame`, também

definindo especificamente um índice por meio de uma lista de números entre 7 a 12.

Aqui temos uma característica importante, note que estamos criando índices para nossos DataFrames data1 e data2 que em nenhum momento se sobrepõem.

```
1 lojas = pd.concat([data1, data2])
2
3 print(lojas)
4
```

	Vendedor	Item	Valor
1	Ana	Camisa Nike	109.90
2	Paulo	Tênis Adidas	299.00
3	Maria	Camisa Nike	115.90
4	Miguel	Meias Speedo	24.99
5	Rafael	Tênis Nike	289.90
6	Ana	Camisa Nike	99.90
7	Ana	Camisa Fila	109.90
8	Fernando	Camisa Adidas	99.00
9	Carlos	Camisa Nike	115.90
10	Carlos	Camisa Nike	109.99
11	Carlos	Camisa Fila	89.90
12	Tânia	Tênis Mizuno	199.90

Uma vez criados os DataFrames com índices únicos entre si, podemos dar prosseguimento com o processo de concatenação.

Seguindo com nosso exemplo, é criada uma variável de nome lojas, que usando do método `pd.concat()` parametrizado com data1 e data2 em forma de lista, recebe os dados agrupados e remapeados internamente pela função `concat()`.

Exibindo em tela o conteúdo de lojas, é possível ver que de fato agora temos um novo DataFrame, composto pelos dados dos DataFrames originais, mantendo inclusive sua indexação pré definida.


```

1 lojas = pd.concat([data1, data2], axis = 1)
2
3 print(lojas)
4

```

	Vendedor	Item	Valor	Vendedor	Item	Valor
1	Ana	Camisa Nike	109.90	NaN	NaN	NaN
2	Paulo	Tênis Adidas	299.00	NaN	NaN	NaN
3	Maria	Camisa Nike	115.90	NaN	NaN	NaN
4	Miguel	Meias Speedo	24.99	NaN	NaN	NaN
5	Rafael	Tênis Nike	289.90	NaN	NaN	NaN
6	Ana	Camisa Nike	99.90	NaN	NaN	NaN
7	NaN	NaN	NaN	Ana	Camisa Fila	109.90
8	NaN	NaN	NaN	Fernando	Camisa Adidas	99.00
9	NaN	NaN	NaN	Carlos	Camisa Nike	115.90
10	NaN	NaN	NaN	Carlos	Camisa Nike	109.99
11	NaN	NaN	NaN	Carlos	Camisa Fila	89.90
12	NaN	NaN	NaN	Tânia	Tênis Mizuno	199.90

Reaproveitando a linha de exercício do tópico anterior, onde falávamos sobre o formato de um DataFrame, podemos associar tal conceito no processo de concatenação.

Como dito anteriormente, em casos onde um DataFrame é remodelado, métricas internas são aplicadas para que o mesmo não perca sua proporção original.

Em nosso exemplo, usando do método `pd.concat()`, definindo para o mesmo o valor 1 em seu parâmetro nomeado `axis`, estamos a realizar a concatenação dos DataFrames `data1` e `data2` usando como referência suas colunas ao invés das linhas.

Importante observar que, nesse caso, tendo desproporcionalidade do número de elementos em linhas e colunas, ao invés do método `concat()` gerar alguma exceção, ele contorna automaticamente o problema preenchendo os espaços inválidos com `NaN`, mantendo assim a proporção esperada para o novo DataFrame.

Apenas realizando um pequeno adendo, como observado em tópicos anteriores, `NaN` é um marcador para um campo onde não existem elementos (numéricos), sendo assim necessário tratar esses dados faltantes.

Mesclando dados de dois DataFrames

Diferentemente do processo realizado no tópico anterior, para alguns contextos específicos pode ser necessário realizar não somente a concatenação de elementos e dois ou mais DataFrames, mas a mescla dos mesmos. Entenda por hora mescla como uma concatenação onde é permitido a união de dois elementos, sobrepondo textos e somando valores quando necessário.

```
1  import pandas as pd
2
3  loja1 = {'Vendedor': ['Ana', 'Paulo', 'Maria',
4                        'Miguel', 'Rafael', 'Ana'],
5           'Item': ['Camisa Nike', 'Tênis Adidas',
6                  'Camisa Nike', 'Meias Speedo',
7                  'Tênis Nike', 'Camisa Nike'],
8           'Valor': [109.90, 299.00, 115.90,
9                   24.99, 289.90, 99.90]}
10
11  loja2 = {'Vendedor': ['Ana', 'Fernando', 'Carlos',
12                       'Carlos', 'Carlos', 'Tânia'],
13           'Item': ['Camisa Fila', 'Camisa Adidas',
14                  'Camisa Nike', 'Camisa Nike',
15                  'Camisa Fila', 'Tênis Mizuno'],
16           'Valor': [109.90, 99.00, 115.90,
17                   109.99, 89.90, 199.90]}
18
19  data1 = pd.DataFrame(loja1, index = [1,2,3,4,5,6])
20  data2 = pd.DataFrame(loja2, index = [7,8,9,10,11,12])
21
22  print(data1)
23  print(data2)
24
```

	Vendedor	Item	Valor
1	Ana	Camisa Nike	109.90
2	Paulo	Tênis Adidas	299.00
3	Maria	Camisa Nike	115.90
4	Miguel	Meias Speedo	24.99
5	Rafael	Tênis Nike	289.90
6	Ana	Camisa Nike	99.90
	Vendedor	Item	Valor
7	Ana	Camisa Fila	109.90
8	Fernando	Camisa Adidas	99.00
9	Carlos	Camisa Nike	115.90
10	Carlos	Camisa Nike	109.99
11	Carlos	Camisa Fila	89.90
12	Tânia	Tênis Mizuno	199.90

Entendendo tais conceitos diretamente por meio de nosso exemplo, aqui usaremos de dois DataFrames gerados a partir de dicionários e ordenados de modo que a concatenação de seus dados não sobreponha elementos graças aos índices predefinidos.

```

1 lojas2 = pd.merge(data1, data2)
2
3 print(lojas2)
4

```

Empty DataFrame
Columns: [Vendedor, Item, Valor]
Index: []

Ao tentar aplicar o método `pd.merge()` sobre nosso DataFrame do mesmo modo como feito via `pd.concat()`, é gerada uma situação inesperada.

Repare que aplicando o método `pd.merge()` por sua vez parametrizado com `data1` e `data2`, atribuindo essa possível mescla de dados a variável `lojas2`, ao tentar exibir seu conteúdo via função `print()` é retornado um DataFrame vazio por parte de elementos.

Segundo o retorno, este DataFrame inclusive possui 3 colunas e um índice vazio, e isto ocorre pois o mecanismo de mescla de dados a partir de

DataFrames espera que seja especificado que métrica será considerada para permitir a mescla.

```
1 lojas2 = pd.merge(data1, data2, how = 'inner', on = 'Item')
2
3 print(lojas2)
4
```

	Vendedor_x	Item	Valor_x	Vendedor_y	Valor_y
0	Ana	Camisa Nike	109.9	Carlos	115.90
1	Ana	Camisa Nike	109.9	Carlos	109.99
2	Maria	Camisa Nike	115.9	Carlos	115.90
3	Maria	Camisa Nike	115.9	Carlos	109.99
4	Ana	Camisa Nike	99.9	Carlos	115.90
5	Ana	Camisa Nike	99.9	Carlos	109.99

Contornando esse problema, em nossa função `pd.merge()`, atribuindo regras aos parâmetros nomeados `how` e `on`, definimos em `how` o método de aproximação, seguido de 'Item' para o parâmetro `on`, estipulando assim que serão mesclados as colunas dos DataFrames com base nos vendedores que realizaram vendas sobre o mesmo tipo de item.

Repare que como retorno desse exemplo temos referências a vendedores de `data1` como `Vendedor_x`, valores de `data1` como `Valor_x`, vendedores de `data2` como `Vendedor_y` e valores de `data2` como `Valor_y`, mesclados em função de todos compartilharem do item Camisa Nike.

Outros vendedores os quais não realizavam interações nos dois DataFrames para o mesmo item simplesmente foram descartados.

```

1 lojas2 = pd.merge(data1, data2, how = 'outer', on = 'Item')
2
3 print(lojas2)
4

```

	Vendedor_x	Item	Valor_x	Vendedor_y	Valor_y
0	Ana	Camisa Nike	109.90	Carlos	115.90
1	Ana	Camisa Nike	109.90	Carlos	109.99
2	Maria	Camisa Nike	115.90	Carlos	115.90
3	Maria	Camisa Nike	115.90	Carlos	109.99
4	Ana	Camisa Nike	99.90	Carlos	115.90
5	Ana	Camisa Nike	99.90	Carlos	109.99
6	Paulo	Tênis Adidas	299.00	NaN	NaN
7	Miguel	Meias Speedo	24.99	NaN	NaN
8	Rafael	Tênis Nike	289.90	NaN	NaN
9	NaN	Camisa Fila	NaN	Ana	109.90
10	NaN	Camisa Fila	NaN	Carlos	89.90
11	NaN	Camisa Adidas	NaN	Fernando	99.00
12	NaN	Tênis Mizuno	NaN	Tânia	199.90

Alterando o parâmetro `how` da função `pd.merge()` para `'outer'`, permitimos que outros vendedores entrem para o rol do DataFrame, mesmo sua relevância sendo baixa em função da baixa participação de interação com os elementos em destaque / elementos mais usados no contexto.

```

1 lojas2 = pd.merge(data1, data2, on = ['Item', 'Valor'])
2
3 print(lojas2)
4

```

	Vendedor_x	Item	Valor	Vendedor_y
0	Maria	Camisa Nike	115.9	Carlos

Repassando como argumento para o parâmetro `on` de `pd.merge()` uma lista contendo `'Item'` e `'Valor'`, será filtrado e retornado apenas o item onde os vendedores conseguiram o vender pelo maior valor possível.

Note que aqui um novo DataFrame é apresentado, contendo apenas Maria e Carlos, vendedores os quais conseguiram vender o item Camisa Nike pelo maior valor 115,9.

Agregando dados de um DataFrame em outro DataFrame

```
1 loja1 = {'Vendedor1': ['Ana', 'Paulo', 'Maria',  
2 | | | | | | | | 'Miguel', 'Rafael', 'Ana'],  
3 | | | | | | | | 'Item1': ['Camisa Nike', 'Tênis Adidas', 'Camisa Nike',  
4 | | | | | | | | 'Meias Speedo', 'Tênis Nike', 'Camisa Nike'],  
5 | | | | | | | | 'Valor1': [109.90, 299.00, 115.90,  
6 | | | | | | | | 24.99, 289.90, 99.90]}  
7  
8 loja2 = {'Vendedor2': ['Ana', 'Fernando', 'Carlos',  
9 | | | | | | | | 'Carlos', 'Carlos', 'Tânia'],  
10 | | | | | | | | 'Item2': ['Camisa Fila', 'Camisa Adidas', 'Camisa Nike',  
11 | | | | | | | | 'Camisa Nike', 'Camisa Fila', 'Tênis Mizuno'],  
12 | | | | | | | | 'Valor2': [109.90, 99.00, 115.90,  
13 | | | | | | | | 109.99, 89.90, 199.90]}  
14  
15 data1 = pd.DataFrame(loja1, index = [1,2,3,4,5,6])  
16 data2 = pd.DataFrame(loja2, index = [1,3,5,7,9,11])  
17  
18 print(data1)  
19 print(data2)  
20
```

	Vendedor1	Item1	Valor1
1	Ana	Camisa Nike	109.90
2	Paulo	Tênis Adidas	299.00
3	Maria	Camisa Nike	115.90
4	Miguel	Meias Speedo	24.99
5	Rafael	Tênis Nike	289.90
6	Ana	Camisa Nike	99.90

	Vendedor2	Item2	Valor2
1	Ana	Camisa Fila	109.90
3	Fernando	Camisa Adidas	99.00
5	Carlos	Camisa Nike	115.90
7	Carlos	Camisa Nike	109.99
9	Carlos	Camisa Fila	89.90
11	Tânia	Tênis Mizuno	199.90

Explorando outra possibilidade, semelhante ao processo de concatenação de elementos de dois ou mais DataFrames, podemos também

unir dados de dois DataFrames definindo um como referência para que os dados do outro apenas sejam agregados de acordo com algum critério definido.

O processo de agregação é feito por meio da função `join()`, que por padrão usa como critério unir dados de duas bases de dados os quais compartilham de mesmos números de índice.

Para nosso exemplo, novamente reutilizamos do exemplo anterior, apenas alterando os valores de índice para o DataFrame atribuído a `data2`.

Exibindo em tela por meio da função `print()` os conteúdos de `data1` e de `data2` podemos ver, como esperado, os DataFrames com seus respectivos índices personalizados e elementos organizados em linhas e colunas.

```
1 data3 = data1.join(data2)
2
3 print(data3)
4
```

	Vendedor1	Item1	Valor1	Vendedor2	Item2	Valor2
1	Ana	Camisa Nike	109.90	Ana	Camisa Fila	109.9
2	Paulo	Tênis Adidas	299.00	NaN	NaN	NaN
3	Maria	Camisa Nike	115.90	Fernando	Camisa Adidas	99.0
4	Miguel	Meias Speedo	24.99	NaN	NaN	NaN
5	Rafael	Tênis Nike	289.90	Carlos	Camisa Nike	115.9
6	Ana	Camisa Nike	99.90	NaN	NaN	NaN

Na sequência é criada uma variável de nome `data3` que chama a função `.join()` sobre `data1`, recebendo como seu conteúdo elementos oriundos de `data2` que podem ser mesclados com `data1`.

Em outras palavras, aqui estamos agregando os dados do DataFrame em `data2` aos dados do DataFrame em `data1` que é o referência, de modo que estaremos agregando os elementos que em seus respectivos DataFrames possuíam o mesmo número de índice.

Novamente usando `print()`, podemos ver o conteúdo de `data3` que de fato é retornado um DataFrame composto de 6 linhas, onde nas mesmas

constam apenas os elementos de data1 equivalentes em número de índice com os elementos de data2. Onde tal equivalência não ocorre, os campos simplesmente são preenchidos com NaN para fins de manter a proporção funcional do DataFrame.

```
1 data1 = pd.DataFrame(loja1, index = [1,2,3,4,5,6])
2 data2 = pd.DataFrame(loja2, index = [7,8,9,10,11,12])
3
4 data3 = data1.join(data2)
5
6 print(data3)
7
```

	Vendedor1	Item1	Valor1	Vendedor2	Item2	Valor2
1	Ana	Camisa Nike	109.90	NaN	NaN	NaN
2	Paulo	Tênis Adidas	299.00	NaN	NaN	NaN
3	Maria	Camisa Nike	115.90	NaN	NaN	NaN
4	Miguel	Meias Speedo	24.99	NaN	NaN	NaN
5	Rafael	Tênis Nike	289.90	NaN	NaN	NaN
6	Ana	Camisa Nike	99.90	NaN	NaN	NaN

Simulando uma exceção, alterando os índices de modo que não exista nenhuma sobreposição, usando do método join() tentando unir os dados de data2 aos de data1, atribuindo este resultado a data3, exibindo em tela data3 é possível notar que tal DataFrame é gerado com os dados de data1, seguidos de uma série de NaN para os campos onde deveriam constar os dados de data2, ausentes por conta de não haver equivalência de índices.

```

1 data1 = pd.DataFrame(loja1, index = [1,2,3,4,5,6])
2 data2 = pd.DataFrame(loja2, index = [1,2,3,4,5,6])
3
4 data3 = data1.join(data2)
5
6 print(data3)
7

```

	Vendedor1	Item1	Valor1	Vendedor2	Item2	Valor2
1	Ana	Camisa Nike	109.90	Ana	Camisa Fila	109.90
2	Paulo	Tênis Adidas	299.00	Fernando	Camisa Adidas	99.00
3	Maria	Camisa Nike	115.90	Carlos	Camisa Nike	115.90
4	Miguel	Meias Speedo	24.99	Carlos	Camisa Nike	109.99
5	Rafael	Tênis Nike	289.90	Carlos	Camisa Fila	89.90
6	Ana	Camisa Nike	99.90	Tânia	Tênis Mizuno	199.90

Simulando outra situação, onde os índices são exatamente iguais, ao usar da função `join()` serão feitas todas as agregações dos elementos de índices iguais a partir dos DataFrames de origem, gerando um novo DataFrame composto de tais dados agregados mantendo o formato original do DataFrame.

Filtrando elementos únicos de uma Serie de um DataFrame

Complementar aos métodos de filtragem de dados vistos em tópicos anteriores específicos, podemos avançar um pouco mais no que diz respeito a métodos específicos para extração destes tipos de dados.

```
1  loja1 = {'Vendedor1': ['Ana', 'Paulo', 'Maria',  
2  | | | | | | | | 'Miguel', 'Rafael', 'Ana'],  
3  | | | | | | | | 'Item1': ['Camisa Nike', 'Tênis Adidas', 'Camisa Nike',  
4  | | | | | | | | 'Meias Speedo', 'Tênis Nike', 'Camisa Nike'],  
5  | | | | | | | | 'Valor1': [109.90, 299.00, 115.90,  
6  | | | | | | | | 24.99, 289.90, 99.90]}  
7  
8  loja1 = pd.DataFrame(loja1)  
9  
10 print(loja1['Item1'].unique())  
11 print(loja1['Item1'].nunique())  
12  
['Camisa Nike' 'Tênis Adidas' 'Meias Speedo' 'Tênis Nike']  
4
```

Mantendo o mesmo DataFrame usado como exemplo para o tópico anterior, temos tal estrutura de dados atribuída a nossa variável de nome loja1, podendo assim realizar mais alguns testes sobre a mesma.

Partindo para filtragem específica, podemos por exemplo, via método unique() filtrar um ou mais elementos únicos de um DataFrame, algo bastante útil para certos contextos.

Sendo assim, diretamente via função print() parametrizando a mesma com loja1 na posição ['Item1'] aplicando o método unique(), sem parâmetros mesmo, nos é retornado uma lista composta apenas dos elementos os quais não se repetem em nenhum momento em nossa base de dados.

Complementar a este método, poderíamos verificar o número de elementos únicos via função len(), porém a biblioteca Pandas oferece uma

função específica para este propósito. Por meio da função `nunique()` nos é retornado o número de elementos únicos de nosso DataFrame.

```
1 print(loja1['Item1'].value_counts())
2
```

Camisa Nike	3
Meias Speedo	1
Tênis Nike	1
Tênis Adidas	1

Name: Item1, dtype: int64

Como mencionado em tópicos anteriores, a estrutura base de um DataFrame costuma ser dados oriundos de tabelas, onde sua estrutura de linhas e colunas já é bem estabelecida, ou a partir de dicionários Python, pois a partir deste tipo de dado chaves do dicionário se tornam colunas, assim como valores deste dicionário são convertidos para linhas de nosso DataFrame.

O fato é que, uma vez que temos dados dispostos nesse formato, podemos aplicar métodos comuns a dicionários para obter o mesmo tipo de retorno.

Apenas como exemplo, de nosso DataFrame associado a `loja1`, se aplicarmos sobre a posição `['Item1']` o método `value_counts()` nos será retornado uma lista com os elementos de nosso DataFrame ordenados pelo de maior ocorrência para o de menor ocorrência.

Encerrando nossa linha de raciocínio, repare que o retorno gerado para nosso pequeno DataFrame é uma lista onde Camisa Nike aparece em primeiro lugar pois tal elemento coexiste em 3 campos diferentes de nossa base de dados, seguido dos elementos de menor relevância de acordo com este parâmetro.

Processamento em paralelo de um Dataframe

Como bem sabemos, se tratando de um Dataframe estamos a trabalhar sobre uma estrutura de dados da biblioteca Pandas a qual podemos inserir e manipular dados de forma mais robusta se com parado a estruturas de dados matriciais básicas.

Dataframes por sua vez possuem sistema de indexação próprio para seus dados e suporte a todo e qualquer tipo de dado em Python (inclusive funções aplicadas aos dados), tornando esta estrutura de dados uma das mais relevantes no meio científico graças as suas particularidades no que diz respeito a simplicidade de uso e performance de processamento.

Nesta linha de raciocínio, é comum que para certos projetos tenhamos bases de dados enormes moldadas em Dataframes, e uma boa prática é trabalhar tais dados de maneira eficiente. Neste processo quase sempre temos etapas de tratamento dos dados, removendo dados desnecessários ou inválidos, posteriormente definindo métricas de leitura e processamento desses dados por meio de funções.

Uma possibilidade que é um verdadeiro divisor de águas quando estamos a falar sobre o processamento de grandes volumes de dados é realizar tal feito através de técnicas de paralelismo.

Relembrando rapidamente, sempre que estamos falando em paralelismo estamos a contextualizar a execução de uma determinada função, dividindo a mesma em partes para processamento individual, o que normalmente acarreta em um considerável ganho de performance (ou por outra perspectiva, redução de tempo de processamento).

Se tratando de Dataframes, é perfeitamente possível dividir o conteúdo de um Dataframe para processamento em partes individuais, sejam em clusters ou simplesmente a partir de técnicas de uso de núcleos individuais de um determinado processador. O ponto é que, justamente em casos de processamento de grandes volumes de dados, nesse caso dados estes originários a partir de um Dataframe, o processamento em paralelo pode ser de grande valia para otimização de uma aplicação.

Partindo para a prática, vamos buscar entender o processo de repartição dos dados de um Dataframe assim como seu processamento, mensurando para fins de comparação o tempo de processamento via método convencional e via paralelismo.

```
1 import pandas as pd
2 import numpy as np
3 from multiprocessing import Pool
4
```

Como de costume, todo processo se inicia com as importações das devidas bibliotecas, módulos e pacotes os quais faremos uso ao longo de nosso código.

Nesse caso, importamos inteiramente as bibliotecas Pandas e Numpy que nos serão úteis para a leitura e manipulação dos dados, por fim da biblioteca multiprocessing importamos a ferramenta Pool, que nos possibilitará definir a execução de uma função sobre nossos dados em paralelismo.

```
1 base = "new_york_hotels.csv"
2 df = pd.read_csv(base, encoding = "ISO-8859-1")
3
4 df.head(10)
5
```

Para nosso exemplo, e apenas para fins de exemplo, vamos usar de uma base de dados chamada new_york_hotels que como nome sugere traz um catálogo dos hotéis situados em Nova Iorque, catálogo este bastante detalhado por sinal, contendo dados como nome do hotel, endereço, cidade, código postal, latitude e longitude, entre outras informações.

O processamento que realizaremos sobre esta base de dados será o cálculo de distância entre dois pontos com base em seus valores de latitude e longitude. Não iremos nos ater a explicar detalhadamente a função que realiza este cálculo via fórmula de Haversine, pois nosso foco será o tempo de processamento desta função.

	A	B	C	D	E	F	G	H	
2872	<td>124.4100</td>								
2873	</tr>								
2874	<tr id="LC21" class="js-file-line">								
2875	<td id="L21" class="blob-num js-line-number" data-line-number="21"></td>								
2876	<td>153579</td>								
2877	<td>TownePlace Suites by Marriott Albany University Area</td>								
2878	<td>1379 Washington Ave</td>								
2879	<td>Albany</td>								
2880	<td>NY</td>								
2881	<td>12206</td>								
2882	<td>42.6867400000000</td>								
2883	<td>-73.8151000000000</td>								
2884	<td>2.5</td>								
2885	<td>179.0800</td>								
2886	<td>119.0700</td>								
2887	</tr>								

Apenas exemplificando o formato original da base de dados.

	ean_hotel_id	name	address1	city
0	269955	Hilton Garden Inn Albany/SUNY Area	1389 Washington Ave	Albany
1	113431	Courtyard by Marriott Albany Thruway	1455 Washington Avenue	Albany
2	108151	Radisson Hotel Albany	205 Wolf Rd	Albany
3	254756	Hilton Garden Inn Albany Medical Center	62 New Scotland Ave	Albany
4	198232	CrestHill Suites SUNY University Albany	1415 Washington Avenue	Albany
5	125200	The Desmond Hotel Albany	660 Albany Shaker Rd	Albany
6	109728	Ramada Plaza Albany	3 Watervliet Avenue Ext	Albany
7	235037	Hampton Inn & Suites Albany-Downtown	25 Chapel St	Albany
8	106464	Albany Marriott	189 Wolf Rd	Albany
9	106922	Best Western Sovereign Hotel - Albany	1228 Western Ave	Albany

Aplicando o método `head()` sobre a variável `df`, por sua vez parametrizado com o número de linhas as quais queremos inspecionar, nos é retornado o cabeçalho da base de dados.

Agora sim temos a base de dados em um formato familiar, composta por nomes de colunas antes da primeira linha, índice automático à esquerda, e os dados organizados em seus devidos campos como em uma simples matriz.

*base de dados disponível em:

https://raw.githubusercontent.com/rajeevratan84/datascienceforbusiness/master/new_york_hotels.csv

```

1 def haversine(df):
2     lat1, lon1, lat2, lon2 = 40.671, -73.985, df['latitude'].values, df['longitude'].values
3     MILES = 3959
4     lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
5     dlat = lat2 - lat1
6     dlon = lon2 - lon1
7     a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
8     c = 2 * np.arcsin(np.sqrt(a))
9     total_miles = MILES * c
10    df["distance"] = total_miles
11    return df
12

```

Logo em seguida é criada a função que aplica a fórmula de Haversine, que por sua vez recebe como ponto de partida as coordenadas de latitude e longitude do ponto de origem, seguido das mesmas coordenadas do ponto de destino, e de um valor fixo pelo qual via triangulação destes pontos irá calcular a distância entre origem e destino. Embora esta fórmula seja de fato um tanto quanto complexa, sua correta aplicação retornará a distância entre dois pontos considerando a curvatura da terra, para assim produzir dados relevantes para navegação (tanto aérea quanto aquática).

A nível de código vale destacar alguns pontos: Primeiramente são declaradas 4 variáveis de nomes lat1, lon1, lat2 e lon2 que recebem como atributo os valores 40.671, -73.985, o valor de cada campo 'latitude' e 'longitude' de nossa base de dados, respectivamente.

Em seguida é declarada uma constante (embora não existam de fato constantes em Python, mas adotamos tal sintaxe apenas por convenção) de nome MILES, que será o valor de um ponto de referência para triangulação.

Na sequência instanciamos novamente as variáveis lat1, lon1, lat2, lon2, dessa vez aplicando sobre as mesmas o método np.deg2rad(), aqui convertendo internamente os valores de graus para radianos. Importante salientar que aqui usamos do método np.deg2rad() aninhado ao método map() para que assim tal método seja aplicado individualmente sobre cada variável. Este procedimento poderia ser feito de diversas outras formas, porém, a mais reduzida por hora é via map(np.deg2rad()).

Também são criadas duas novas variáveis, dessa vez de nomes dlat e dlon que recebem como atributo o retorno da subtração das latitudes e longitudes envolvidas no processo anterior.

São criadas duas outras variáveis de nomes `a` e `c`, que em suas respectivas linhas de código usam de diversos métodos para cruzar os dados produzidos anteriormente. Dentre estes métodos vale destacar `np.sin()` para cálculo do seno, `np.cos()` para cálculo de cosseno, `np.arcsin()` para seno invertido, por fim `np.sqrt()` para cálculo da raiz quadrada de um determinado número.

Finalizando, é criada uma variável de nome `total_miles`, que por sua vez recebe como atributo o retorno da multiplicação do último valor atribuído para as variáveis `MILES` e `c`.

O Dataframe `df` é instanciado novamente, mais especificamente seu campo `'distance'`, atualizando seu valor com o dado/valor atribuído a `total_miles`.

Por fim, é retornado o Dataframe após atualizado.

```
1 %%timeit
2
3 distance_df = haversine(df)
4
```

Uma vez que temos os dados carregados e nossa função para testes definida, podemos finalmente partir para a parte que nos interessa neste tópico, o tempo de processamento dos dados de nosso Dataframe.

A nível de código, em meu caso executando via Colab, inserindo o prefixo `%%time` da célula será gerado um retorno referente ao tempo de execução da mesma.

Nesta célula, neste bloco de código em particular, declaramos uma variável de nome `distance_df` que instancia e inicializa o método `haversine()` por sua vez parametrizado com o conteúdo da variável `df`.

Executando esta célula o retorno referente ao método `haversine()` executado de forma convencional será:

The slowest run took 23.12 times longer than the fastest. This could mean that an intermediate result is being cached.

1000 loops, best of 5: 210 µs per loop

Em tradução livre:

A execução mais lenta demorou 23,12 vezes mais que a mais rápida. Isso pode significar que um resultado intermediário está sendo armazenado em cache.

1000 loops, melhor de 5: 210 µs por loop

Como explicado no retorno, o melhor tempo de execução obtido foi de 210 microssegundos, ou em conversão direta, 0,21 milissegundos.

```
1 def parallelize_dataframe(df, func, n_cores = 8):
2     df_split = np.array_split(df, n_cores)
3     pool = Pool(n_cores)
4     df = pd.concat(pool.map(func, df_split))
5     pool.close()
6     pool.join()
7     return df
8
```

Haja visto que já temos um dado / parâmetro referente ao tempo de execução de forma convencional, podemos partir para a elaboração do método que utilizará de paralelismo.

Nesse caso, criamos uma nova função, dessa vez de nome `parallelize_dataframe()`, que receberá obrigatoriamente uma base de dados, uma função a ser aplicada e um número de núcleos de processados manualmente definido.

Indentado ao corpo da função, inicialmente declaramos uma variável de nome `df_split`, que instancia e inicializa o método `np.array_split()`, aqui parametrizado com a base de dados de origem e o número de partições a serem geradas, nesse caso, 8 referente a 8 núcleos de processamento.

Também é declarada uma variável de nome `pool`, que chama a função `Pool()` também parametrizada com um número de núcleos. Lembrando que a ferramenta `Pool()` por sua vez de forma simples e objetiva cria a instância para processamento em paralelo de funções, tudo o que estiver hierarquicamente sob sua instância será processado usando de métricas internas de paralelismo.

É então instanciada a variável `df`, referente a nossa base de dados, aplicando sobre a mesma o método `pd.concat()`, parametrizado por sua vez com o método `map()` aplicado a variável `pool`, por fim parametrizado com a função a ser aplicada e neste caso, as partições da base de dados de origem.

Encerrando o processamento via `Pool()`, são instanciadas e inicializadas as funções `close()` e `join()`, por fim, retornando nosso Dataframe após atualizado.

```
1 %%timeit
2
3 distance_df = parallelize_dataframe(df, haversine)
4
```

Em uma nova célula, instanciamos novamente a variável `distance_df`, dessa vez a partir da mesma fazendo uso da função `parallelize_dataframe()`, aqui parametrizada com nosso Dataframe e a função a ser aplicada sobre cada partição sua.

Executando esse bloco de código, onde a função `haversine()` é aplicada fazendo uso de métricas internas de paralelismo para o processamento de nossos dados, o retorno será:

10 loops, best of 5: 210 µs per loop

Em conversão direta, 0,014 milissegundos, uma notável redução de tempo de processamento se comparado ao método convencional (210 µs / 0.21 ms).

CONSIDERAÇÕES FINAIS

E assim concluímos este pequeno compêndio introdutório sobre os aspectos práticos mais rotineiros se tratando da manipulação de dados através das bibliotecas Pandas e Numpy com suas ferramentas.

Espero que estas páginas tenham sido de prazerosa leitura assim como para mim as foi escrever linha por linha.

Mais importante que isso, espero que de fato este pequeno livro tenha lhe ajudado em seu processo de aprendizagem em mais um tópico dentro das quase infinitas possibilidades que temos quando programamos em Python, e que ao final desta leitura você consiga dar seus primeiros passos no tratamento de dados.

Muito obrigado por adquirir esta obra, sucesso em sua jornada, um forte abraço... Fernando Feltrin.



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se

singlelogin.re

go-to-zlibrary.se

single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>