



INSTITUTO DE GESTÃO E
TECNOLOGIA DA INFORMAÇÃO

Python para a Análise de Dados

Bootcamp Desenvolvedor Python

Matheus de Oliveira Mendonça

2020

Python para a Análise de Dados

Bootcamp Desenvolvedor Python

Matheus de Oliveira Mendonça

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

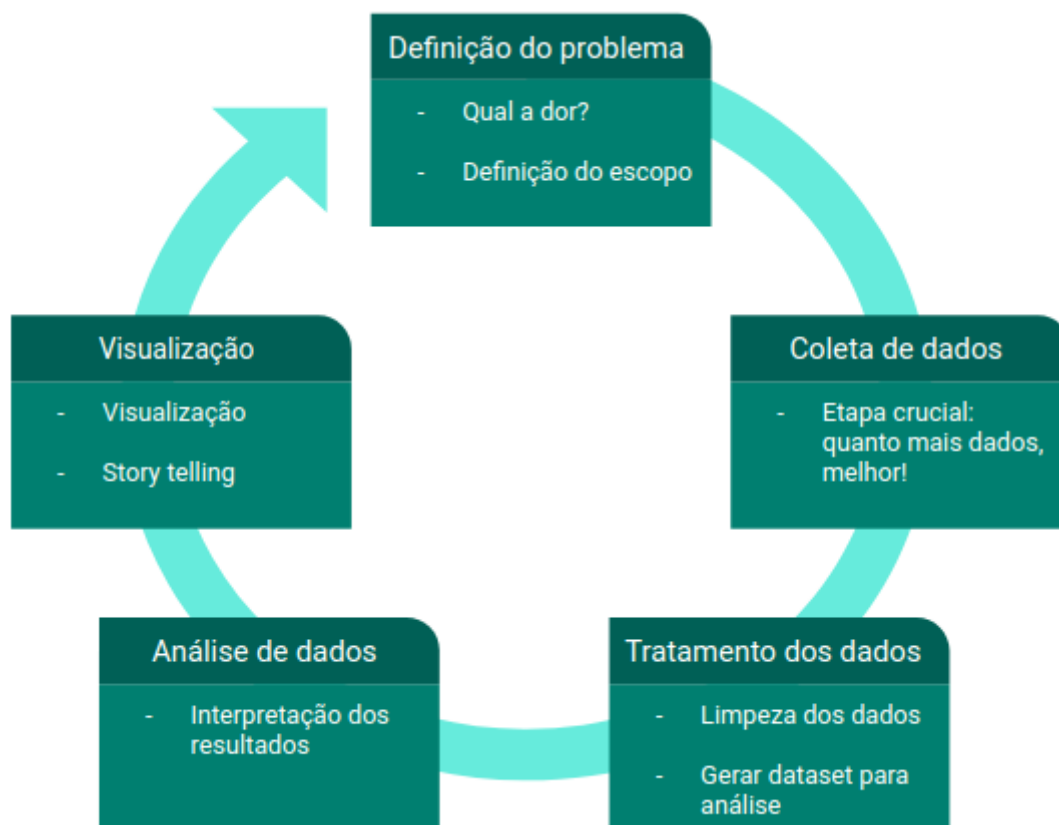
Capítulo 1.	Introdução à análise de dados	4
Capítulo 2.	Pandas e numpy para análise de dados	6
	Numpy para análise de dados	6
	Pandas para análise de dados.....	15
Capítulo 3.	Introdução ao scikit-learn	24
	Introdução ao machine learning.....	25
Capítulo 4.	Conclusão	28
Referências	29

Capítulo 1. Introdução à análise de dados

A análise de dados pode ser definida como processo de coleta, tratamento, análise e apresentação de dados, de forma a trazer **novas informações e agregar valor** ao processo de **tomada de decisão** de qualquer negócio (ver Figura 1).

Esse processo nasce a partir de uma dor do negócio, na qual os conhecimentos empíricos não são suficientes para uma tomada de decisão assertiva e imparcial. Assim, faz-se necessária a utilização de um processo metodológico bem definido para munir o tomador de decisão com informações adicionais relevantes, muitas vezes desconhecidas até então.

Figura 1 - Ciclo de um trabalho de análise de dados.



Existem diversas ferramentas para análise de dados, mas sem dúvida alguma o Python, em conjunto com diversas bibliotecas disponíveis, é uma ferramenta poderosíssima que vem ganhando cada vez mais popularidade entre a comunidade

científica e os desenvolvedores. A Figura 2 mostra o resultado expressivo da pesquisa conduzida pelo [StackOverflow](https://stackoverflow.com/) que mostra o Python figurando entre as 5 linguagens de programação mais populares, desbancando o Java.

Figura 2 - Popularidade das linguagens de programação em 2019 no StackOverflow.



Este curso dedica-se à introdução - de maneira prática - de algumas das ferramentas fundamentais de análise de dados em Python, a saber:

1. NumPy;
2. Pandas;
3. Scikit-learn.

Capítulo 2. Pandas e numpy para análise de dados

Numpy para análise de dados

O [numpy](#) é uma das principais bibliotecas para computação científica em Python. Ela disponibiliza um objeto de array multidimensional de alta performance e diversas ferramentas para se trabalhar com esses objetos.

Como a maioria das bibliotecas em Python, a instalação do numpy é bem simples e pode ser executada através dos comandos:

Figura 3 - Instalação do numpy.

CONDA

If you use `conda`, you can install it with:

```
conda install numpy
```

PIP

If you use `pip`, you can install it with:

```
pip install numpy
```

Fonte: <https://numpy.org/install/>.

Para utilizá-la, é necessário, inicialmente, importar o pacote com o comando:

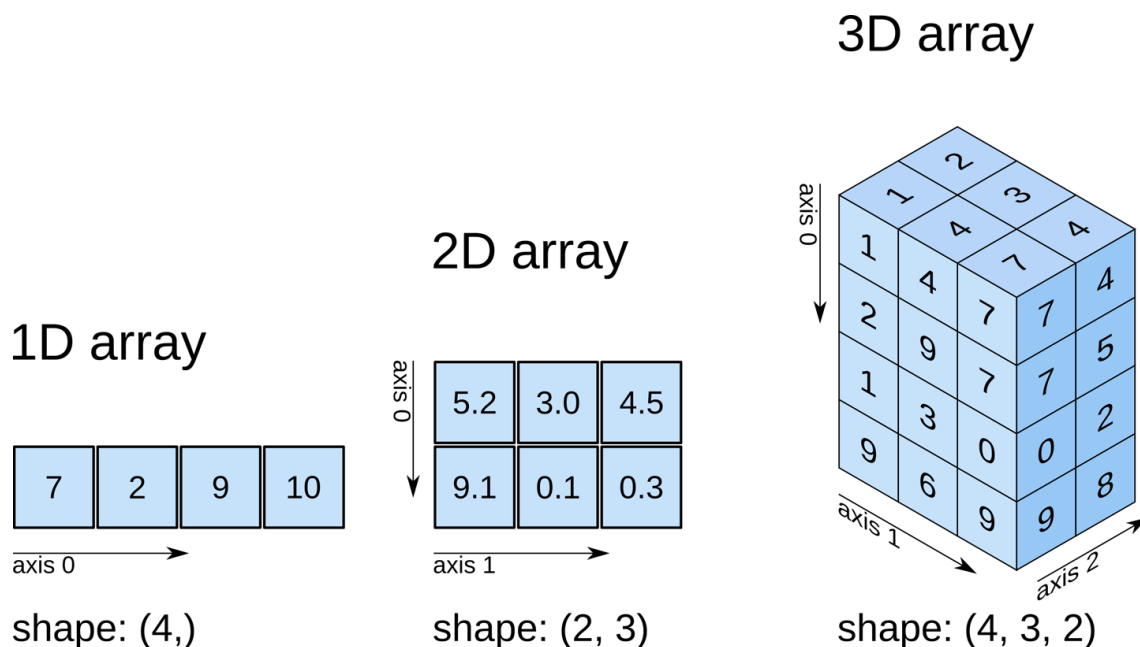
```
import numpy as np
```

- **Arrays**

Uma array em numpy é uma grade de valores, todos do mesmo tipo, indexada por uma tupla de inteiros não negativos. O número de dimensões de uma array é chamado de *rank* do array; o *shape* de uma array é representada através de uma

tupla de inteiros, que indicam o tamanho da array em cada dimensão. A Figura a seguir ilustra alguns exemplos de arrays.

Figura 4 - Ilustração de arrays multidimensionais.



Fonte: https://fgnt.github.io/python_crashkurs_doc/include/numpy.html.

É possível criar arrays em numpy utilizando listas de Python aninhadas, e o acesso dos elementos é feito utilizando colchetes:

```
# cria um array de 2 dimensões: matrix 3x3
a = np.array([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
print("Array criado:\n", a)
print("shape:", a.shape)
```

```
Array criado:
[[1 2 3]
 [2 3 4]
 [3 4 5]]
shape: (3, 3)
```

A biblioteca numpy também oferece várias funções para a criação de arrays:

- **np.zeros(tuple):** cria uma array com todos os valores iguais a 0. As dimensões da array são definidas pela tupla passada por parâmetro.

- **`np.ones(tuple)`**: semelhante à função acima, porém cria uma array com todos os valores iguais a 1.
- **`np.eye(n)`**: cria uma matriz identidade de tamanho **$n \times n$** . O tipo de **n** deve ser **`int`**.
- **`np.random.random(tuple)`**: cria uma matriz com valores aleatórios. As dimensões são definidas pela tupla passada por parâmetro.
- **`np.linspace(start, stop, num)`**: cria um vetor contendo **`num`** elementos, linearmente espaçados dentro do intervalo **`[start, stop]`**.

Alguns exemplos de implementação estão listados a seguir:

```
# criação de uma matriz 3x2 de 0's
print("Criação de uma matriz 3x2 de 0's:")
print(np.zeros((3, 2)))

# criação de uma matriz 3x2 de 1's
print("Criação de uma matriz 3x2 de 1's:")
print(np.ones((3, 2)))

# criação de uma matriz identidade 3x3
print("Criação de uma matriz identidade 3x3:")
print(np.eye(3))

# criação de uma matriz 3x3 com números aleatórios
print("Criação de uma matriz 3x3 com números aleatórios:")
print(np.random.random((3, 3)))
```

Criação de uma matriz 3x2 de 0's:

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

Criação de uma matriz 3x2 de 1's:

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

Criação de uma matriz identidade 3x3:

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

Criação de uma matriz 3x3 com números aleatórios:

```
[[0.91566385 0.41521502 0.3004463 ]
 [0.94635743 0.40210197 0.58536861]
 [0.17914514 0.75828708 0.83239962]]
```


• Indexação de arrays

Assim como listas em Python, arrays em numpy podem ser fatiadas (*slicing*, termo comum em inglês). Dado que arrays podem ser multidimensionais, é necessário especificar uma fatia para cada uma das dimensões da array:

```
# Criação de uma matriz bidimensional de tamanho (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
print("A:")
A = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(A)

# indexação do array A para extração de um sub-array consistindo
# das primeiras 2 linhas de A e das colunas de índice 1 e 2,
# resultando em um novo array B de tamanho (2, 2):
# [[2 3]
#  [6 7]]
print("B:")
B = A[:2, 1:3]
print(B)
```

```
A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
B:
[[2 3]
 [6 7]]
```

Repare que um slice de uma array é uma visualização do mesmo dado, ou seja, ao alterar um slice, o **dado original também será alterado**:

```
# B[0, 0] aponta para a mesma posição de memória de A[0, 1]
print("A[0,1] antes:")
print(A[0, 1])
B[0, 0] = 77
print("A[0,1] depois:")
print(A[0, 1])
```

```
A[0,1] antes:
2
A[0,1] depois:
77
```

Para a criação de um sub-array que **não** compartilha memória com o array original, faz-se necessária a utilização do método `copy()` durante a indexação (*slicing*):

```
# Criação de uma matriz bidimensional de tamanho (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
A = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# indexação do array A para extração de um sub-array consistindo
# das primeiras 2 linhas de A e das colunas de índice 1 e 2,
# resultando em um novo array B de tamanho (2, 2):
# [[2 3]
#  [6 7]]
B = A[:2, 1:3].copy() # slicing com cópia do objeto

# B[0, 0] agora NÃO aponta para a mesma posição de memória de A[0, 1]
print("A[0,1] antes:")
print(A[0, 1])
B[0, 0] = 77
print("A[0,1] depois:")
print(A[0, 1])

A[0,1] antes:
2
A[0,1] depois:
2
```

- **Funções aritméticas**

Funções aritméticas básicas operam sobre cada elemento em arrays, e estão disponíveis tanto como sobrecarga de operadores quanto como funções no módulo `numpy`. Elas podem ser implementadas tanto entre arrays quanto entre um array e um escalar (exemplo: `int` e `float`). Exemplos:

- Soma:

```
# arrays
x = np.array([[1,2], [3,4]])
y = np.array([[5,6], [7,8]])

# Soma por elemento;
print("Sobrecarga de operador:")
print(x + y)
print("Função do módulo:")
print(np.add(x, y))
print("Soma entre um array e um escalar:")
print(x + 10)
```

```
Sobrecarga de operador:
[[ 6  8]
 [10 12]]
Função do módulo:
[[ 6  8]
 [10 12]]
Soma entre um array e um escalar:
[[11 12]
 [13 14]]
```

- Subtração:

```
# Diferença por elemento
print (x - y)
print (np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

- Multiplicação:

```
# Produto por elemento
print (x * y)
print (np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

Repare que o operador `*` representa a multiplicação por elemento, e não a multiplicação de matrizes. Para calcular o produto interno de vetores, multiplicar um

vetor por uma matriz ou multiplicar matrizes, a função utilizada é *dot*, conforme exemplificado a seguir:

```
# matrizes
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

# vetores
v = np.array([5, 5])
w = np.array([2, 3])

# Produto interno de vetores
print(v.dot(w))
print(np.dot(v, w))
```

25
25

```
# Produto de um vetor e uma matriz
print (x.dot(v))
print (np.dot(x, v))
```

[15 35]
[15 35]

```
# Produto de matrizes
print (x.dot(y))
print (np.dot(x, y))
```

[[19 22]
[43 50]]
[[19 22]
[43 50]]

- Divisão:

```
# Divisão por elemento
print (x / y)
print (np.divide(x, y))
```

[[0.2 0.33333333]
[0.42857143 0.5]]
[[0.2 0.33333333]
[0.42857143 0.5]]

- Outras operações:

```
# Raiz quadrada por elemento
print(np.sqrt(x))
```

```
[[1.          1.41421356]
 [1.73205081 2.          ]]
```

```
# Exponenciação por elemento
print(x**2)
```

```
[[ 1  4]
 [ 9 16]]
```

```
# Logarítmo por element
print(np.log(x))
```

```
[[0.          0.69314718]
 [1.09861229 1.38629436]]
```

• Comparações

Comparações *booleanas* também são possíveis em numpy arrays e são executadas elemento por elemento, retornando um outro numpy array com o resultado da comparação. A seguir, alguns exemplo de comparações que podem ser executadas:

- Maior/Maior ou igual:

```
# maior
print("Comparação maior:")
print(A > B)
print(A > s)

# maior ou igual
print("Comparação maior ou igual:")
print(A >= B)
print(A >= s)
```

```
Comparação maior:
[False True True]
[False False False]
Comparação maior ou igual:
[False True True]
[False False True]
```

- Menor/Menor ou igual:

```
# comparações booleanas
A = np.array([1, 2, 3])
B = np.array([2, 0, 2])
s = 3

# menor
print("Comparação menor:")
print(A < B)
print(A < s)

# menor ou igual
print("Comparação menor ou igual:")
print(A <= B)
print(A <= s)
```

```
Comparação menor:
[ True False False]
[ True  True False]
Comparação menor ou igual:
[ True False False]
[ True  True  True]
```

- Igualdade:

```
# igual
print("Comparação de igualdade:")
print(A == B)
print(A == s)
```

```
Comparação de igualdade:
[False False False]
[False False  True]
```

- Indexação *booleana*:

```
# indexação booleana: um novo subarray contendo uma
# cópia dos elementos em que a condição de verificação se aplica
cond = A <= 2
D = A[cond]
print("A:", A)
print("condição:", cond)
print("D:", D)
```

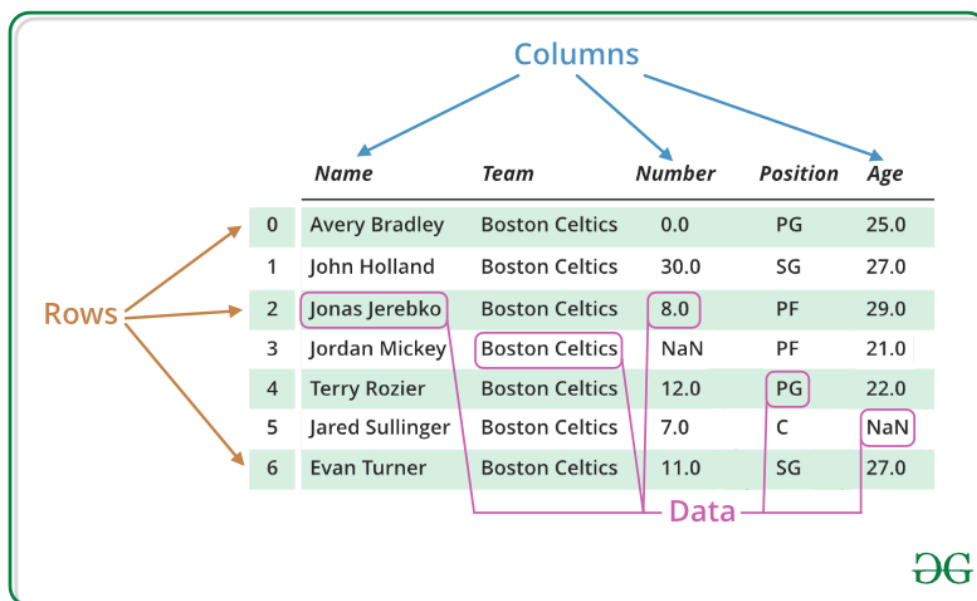
```
A: [1 2 3]
condição: [ True  True False]
D: [1 2]
```

Pandas para análise de dados

Pandas é um pacote em Python desenvolvido para disponibilizar estruturas de dados rápidas e flexíveis para se trabalhar com dados "relacionais" ou "rotulados" (ver Figura 5). Ele é adequado para diversos tipos de dados:

- Dados tabulares com colunas de tipos heterogêneos, como em tabelas SQL ou planilhas Excel;
- Dados de séries temporais ordenados ou não ordenados;
- Dados matriciais arbitrários, com linhas e colunas rotuladas;
- Qualquer outro tipo de conjunto de dados estatísticos ou observados. Os dados não necessariamente precisam estar rotulados para serem utilizados com a estrutura de dados do Pandas.

Figura 5 - Exemplo de um *DataFrame*.



The diagram illustrates a DataFrame structure. It features a table with 7 rows and 6 columns. The columns are labeled 'Name', 'Team', 'Number', 'Position', and 'Age'. The rows are indexed from 0 to 6. The data is as follows:

	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Annotations in the diagram include: 'Columns' pointing to the column headers, 'Rows' pointing to the row indices, and 'Data' pointing to the data cells. A green logo is visible in the bottom right corner.

Fonte: <https://www.geeksforgeeks.org/python-Pandas-dataframe/>.

O Pandas utiliza dois tipos principais de estruturas de dados: **Series** (unidimensional) e **DataFrame** (bidimensional), que são abstrações de vetores e matrizes, respectivamente, assim como no numpy, porém com características mais versáteis e mais próximas dos dados do mundo real. Essas duas estruturas são

capazes de representar a maioria dos casos de uso em finanças, em estatística, em ciências sociais e em várias áreas da engenharia. A próxima Figura ilustra esse conceito:

Figura 6 - Exemplo de um *DataFrame*.

Series 1		Series 2		Series 3		DataFrame
Mango		Apple		Banana		Mango Apple Banana
0 4		0 5		0 2		0 4 5 2
1 5		1 4		1 3		1 5 4 3
2 6		2 3		2 5		2 6 3 5
3 3		3 0		3 2		3 3 0 2
4 1		4 2		4 7		4 1 2 7

Fonte: <http://www.datasciencemadesimple.com/create-series-in-python-Pandas/>.

Algumas das tarefas que o Pandas faz com eficiência são:

- Tratamento de dados faltantes (representados por NaN);
- Tamanhos mutáveis: colunas podem ser inseridas e excluídas de *DataFrames* com facilidade;
- Grupo de funcionalidades poderoso e flexível para operações de split-apply-combine, para agregar e transformar conjuntos de dados;
- Ferramentas de IO robustas para leitura de dados de arquivos como CSV, Excel, bancos de dados, além da possibilidade de se utilizar o formato HDF5;
- Entre outros.

Para leitura dos dados, existem diversas funções, a depender do formato do dado de entrada. Algumas das mais usadas estão listados abaixo:

- `read_csv`: leitura de arquivos CSV;
- `read_json`: leitura de arquivos JSON;
- `read_html`: leitura de arquivos HTML;
- `read_clipboard`: leitura de dados da área de transferência (CTRL + C, por exemplo);
- `read_hdf`: leitura de arquivos HDF5;

- read_sql: leitura de arquivos SQL;
- read_excel: leitura de arquivos Excel.

Uma das principais características do Pandas é a possibilidade de lidar com diferentes formatos de uma maneira muito simples e similar ao que já está implementado no numpy (slicing, indexação, comparações, etc). Entre os tipos de dados suportados e como eles se relacionam com os formatos nativos do Python, têm-se:

Tabela 1 - Tipos de dados suportados no Pandas.

Pandas dtype	Python type	Uso
object	str ou mixed	Texto ou valores mistos numéricos e não-numéricos
int64	int	Números inteiros
float64	float	Números ponto flutuantes
bool	bool	Valores True/False
datetime64	NA	Valores em formato de data e hora
timedelta[ns]	NA	Diferença de dois datetimes
category	NA	Lista finita de texto

A instalação do Pandas é análoga à instalação do numpy e, para sua utilização, basta a importação da biblioteca no ambiente de desenvolvimento, conforme descrito à seguir:

```
# importando as bibliotecas
import numpy as np
import pandas as pd
```

Para carregar uma base de dados em memória, basta utilizar um dos métodos de leitura disponíveis conforme o formato do arquivo que contém os dados a serem

analisados. Segue um exemplo de leitura de um arquivo com extensão .csv:

```
# leitura dos dados
df = pd.read_csv("https://pycourse.s3.amazonaws.com/temperature.csv")

# visualizando as primeiras 3 linhas
df.head(3)
```

	date	temperatura	classification
0	2020-01-01	29.1	quente
1	2020-02-01	31.2	muito quente
2	2020-03-01	28.5	quente

Esse *DataFrame* possui 3 colunas dos seguintes tipos:

```
# tipos de dados
df.dtypes

date                object
temperatura         float64
classification      object
dtype: object
```

Note que a coluna **date** claramente é uma representação de datas, mas como não explicitamos na leitura do arquivo quais os tipos de cada coluna, o Pandas inferiu que essa coluna é do tipo object. Para que possamos usufruir das funcionalidades de comparações de datetimes, precisamos forçar a conversão da coluna **date** para o tipo datetime:

```
# transformando o tipo da coluna date para datetime
df['date'] = pd.to_datetime(df['date'])
```

Também é conveniente definir qual coluna do *DataFrame* será utilizada como “referência” para as demais. No Pandas, essa “referência” é denominada **index** e é especialmente útil quando temos uma coluna de datetime, pois ela serve para determinar os labels do eixo de todos os outros objetos do *DataFrame*:

```
# setando o índice
df = df.set_index('date')
```

```
# visualizando o índice
print(df.index)
```

```
DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01', '2020-04-01',
                '2020-05-01', '2020-06-01'],
              dtype='datetime64[ns]', name='date', freq=None)
```

Algumas das manipulações mais comuns são listadas a seguir:

- Estatísticas básicas:

```
# estatísticas básicas de dados numéricos
df.describe()
```

temperatura	
count	6.000000
mean	26.800000
std	4.075782
min	20.000000
25%	25.000000
50%	28.250000
75%	28.950000
max	31.200000

- Indexação por índice (método *iloc*):

```
# indexação por índice
# selecionado todas as linhas e a coluna 1
# coluna 1: temperatura
df.iloc[:, 1]
```

```
0    29.1
1    31.2
2    28.5
3    28.0
4    24.0
5    20.0
Name: temperatura, dtype: float64
```

- Indexação por nome (método *loc*):

```
# indexação por nome
# selecionado todas as linhas e a coluna 1
df.loc[:, 'temperatura']
```

```
0    29.1
1    31.2
2    28.5
3    28.0
4    24.0
5    20.0
Name: temperatura, dtype: float64
```

- Ordenação por coluna:

```
# ordenando por uma coluna
df.sort_values(by='temperatura')
```

	temperatura	classification
date		
2020-06-01	20.0	frio
2020-05-01	24.0	confortavel
2020-04-01	28.0	quente
2020-03-01	28.5	quente
2020-01-01	29.1	quente
2020-02-01	31.2	muito quente

- Ordenação por índice:

```
# ordenando pelo índice
df.sort_index(ascending=False)
```

	temperatura	classification
date		
2020-06-01	20.0	frio
2020-05-01	24.0	confortavel
2020-04-01	28.0	quente
2020-03-01	28.5	quente
2020-02-01	31.2	muito quente
2020-01-01	29.1	quente

- Indexação booleana:

```
# indexação booleana
# seleção de exemplos acima de 25 graus
df[df['temperatura'] >= 25]
```

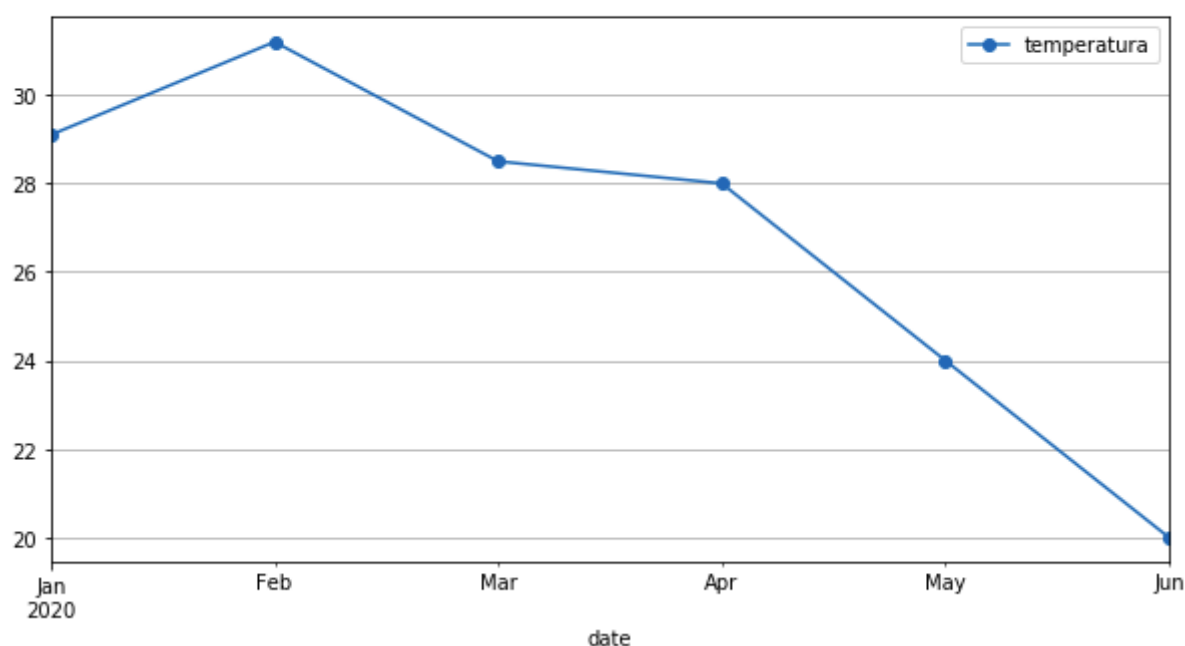
	temperatura	classification
date		
2020-01-01	29.1	quente
2020-02-01	31.2	muito quente
2020-03-01	28.5	quente
2020-04-01	28.0	quente

```
# indexação booleana considerando datetime
# seleção de entradas até Março de 2020
df[df.index <= '2020-03-01']
```

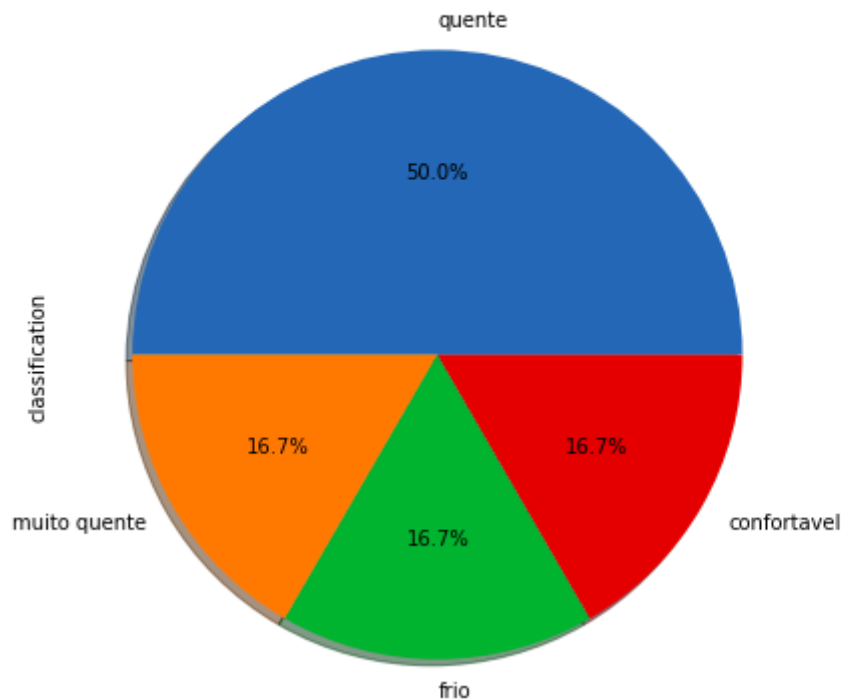
	temperatura	classification
date		
2020-01-01	29.1	quente
2020-02-01	31.2	muito quente
2020-03-01	28.5	quente

- Visualização: além de ser escrito em cima do numpy, o Pandas também herda os métodos de visualização do matplotlib, uma biblioteca de visualização de dados muito versátil e utilizada. Alguns *plots* podem ser feitos com apenas uma linha de código no Pandas:

```
# plot de linhas
df.plot(style='-o', figsize=(10, 5), grid=True);
```



```
# pie plot
df['classification'].value_counts().plot.pie(autopct='%1.1f%%',
                                             shadow=True,
                                             figsize=(10, 7));
```



Capítulo 3. Introdução ao scikit-learn

O [scikit-learn](#) é um dos mais utilizados *frameworks* de aprendizado de máquinas em Python. Ele possui interfaces para a execução de diversas atividades inerentes às atividades de um cientista de dados:

- Classificação: identificação de qual categoria um novo exemplo pertence.
- Regressão: predição de um valor contínuo associado a um determinado exemplo.
- Agrupamento: agrupamento automático de exemplos em conjuntos.
- Redução de dimensionalidade: redução do número de variáveis presentes em um *dataset*.
- Seleção de modelos: comparação, validação e calibração de parâmetros de modelos.
- Pré-processamento: extração/seleção de atributos, normalização e tratamento de dados faltantes.

Para exemplificação, resolveremos um problema simples de machine learning baseado no dataset que estamos utilizando até o momento:

temperatura classification

20.0	frio
24.0	confortavel
28.0	quente
28.5	quente
31.2	muito quente
29.1	quente

Baseado nesse conjunto de seis exemplos de pares (temperatura, classification), iremos treinar um modelo para nos dizer qual será a classificação de uma temperatura que não está presente nessa tabela. Exemplo: para a temperatura de 9°C, qual classificação o modelo irá retornar? Esperamos que seja frio ...

O modelo matemático irá **aprender**, a partir dessa pequena base de dados, a **inferir** (generalizar) a classificação de uma temperatura nunca vista antes pelo modelo. Daí o nome aprendizado de máquinas.

Introdução ao machine learning

No scikit-learn, é comum adotar a nomenclatura **x** para variáveis preditoras e **y** para a variável alvo. No nosso exemplo, **x** é a temperatura e **y** é a classificação. Sendo assim, o seguinte trecho de código executa esse *slicing*:

```
# extração de x e y
x, y = df[['temperatura']].values, df[['classification']].values
print("x:\n", x)
print("y:\n", y)
```

```
x:
[[29.1]
 [31.2]
 [28.5]
 [28. ]
 [24. ]
 [20. ]]
y:
[['quente']
 ['muito quente']
 ['quente']
 ['quente']
 ['confortavel']
 ['frio']]
```

Observe que a variável resposta é uma *string*, mas modelos matemáticos necessitam de valores numéricos para funcionarem. Sendo assim, umas das funcionalidades presentes no scikit-learn é a codificação de variáveis categóricas em variáveis numéricas, que pode ser feita pelo seguinte trecho:

```
# pré-processamento
from sklearn.preprocessing import LabelEncoder

# conversão de y para valores numéricos
le = LabelEncoder()
y = le.fit_transform(y.ravel())
print("y:\n", y)

y:
[3 2 3 3 0 1]
```

Após o pré-processamento, partiremos para o treinamento do modelo. (Existem outras etapas em um fluxo normal de machine learning. Aqui, para fins de exemplificação, não as realizaremos):

```
# modelo
from sklearn.linear_model import LogisticRegression

# classificador
clf = LogisticRegression()
clf.fit(x, y)
```

Com o modelo treinado, podemos inferir a classificação de novas temperaturas. Para isso, iremos gerar uma sequência de 100 valores de temperatura entre 0 e 45 para avaliarmos o resultado da generalização do modelo:

```
# gerando 100 valores de temperatura
# linearmente espaçados entre 0 e 45
predição em novos valores de temperatura
x_test = np.linspace(start=0., stop=45., num=100).reshape(-1, 1)

# predição desses valores
y_pred = clf.predict(x_test)
```

De posse da predição, podemos realizar a conversão inversa dos valores numéricos de **y** para os seus valores originais (frio, confortável, quente, muito quente):

```
# conversão de y_pred para os valores originais
y_pred = le.inverse_transform(y_pred)
```

Salvando os resultados em um *DataFrame*:

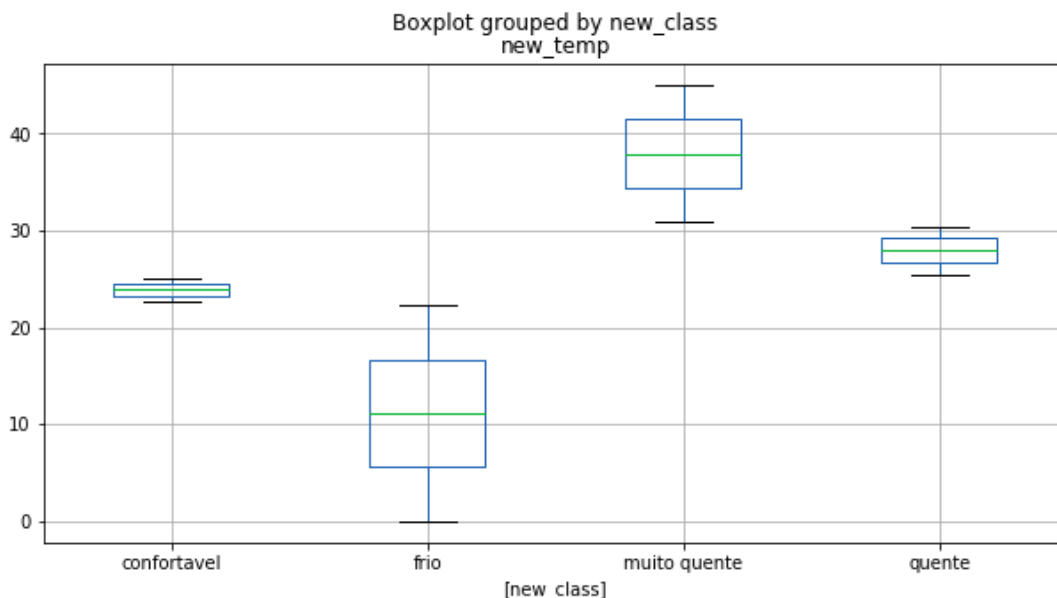
```
# output
output = {'new_temp': x_test.ravel(),
          'new_class': y_pred.ravel()}
output = pd.DataFrame(output)

# estatísticas
output.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 2 columns):
new_temp      100 non-null float64
new_class      100 non-null object
dtypes: float64(1), object(1)
memory usage: 1.7+ KB
```

De posse dos resultados, vamos visualizar as classificações inferidas pelo modelo através de um plot de caixa (*boxplot*, em inglês), que nos mostra a distribuição dos valores de cada uma das classes para o novo conjunto de valores de temperatura gerados. Observe que o comportamento está como o esperado e que o modelo conseguiu aprender corretamente partindo de uma base de dados bem pequena.

```
# distribuição do output produzido
# conseguimos inferir a classificação novas temperaturas
# a partir de um dataset com 6 exemplos
output.boxplot(by='new_class', figsize=(10, 5));
```



Capítulo 4. Conclusão

Esse módulo dedicou-se à introdução de conceitos fundamentais da análise de dados e apresentou 2 das bibliotecas mais utilizadas no cotidiano de um profissional de dados: Pandas e numpy. Além disso, foi apresentada de forma simplificada a ideia de geral de um problema de aprendizado de máquinas, através da resolução de um problema de classificação com a biblioteca scikit-learn.

Referências

JAMES, G.; WITTEN, D.; HASTIE, T.; TIBISHIRANI, R. *An introduction to statistical learning*. New York: Springer, 2013.

NumPy. Disponível em: <<https://numpy.org/>>. Acesso em: 14 de jul. de 2020.

Pandas. Disponível em: <<https://Pandas.pydata.org/>>. Acesso em: 14 de jul. de 2020.

Scikit-learn. Disponível em: <<https://scikit-learn.org/stable/>>. Acesso em: 14 de jul. de 2020