

# Ciência de Dados

---

## Aula 2.1 - Engenharia de Dados - Introdução

Prof. Wellington Franco



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS DE CRATEÚS



Laboratório de Engenharia de  
Software e Sistemas

# Agenda

1. Introdução
2. Extração;
3. Tratamento;
4. Limpeza;
5. Manipulação de Dados;

# Engenharia de Dados

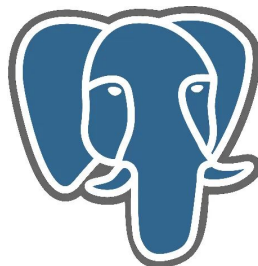
- Principais Atividades
  - Extração;
  - Tratamento;
  - Limpeza;
  - Manipulação de Dados;

# **Configurando o Ambiente**

# Ambiente de Desenvolvimento

Para a realização dos experimentos propostos nas seguintes aulas, é necessário ter instalado em sua máquina:

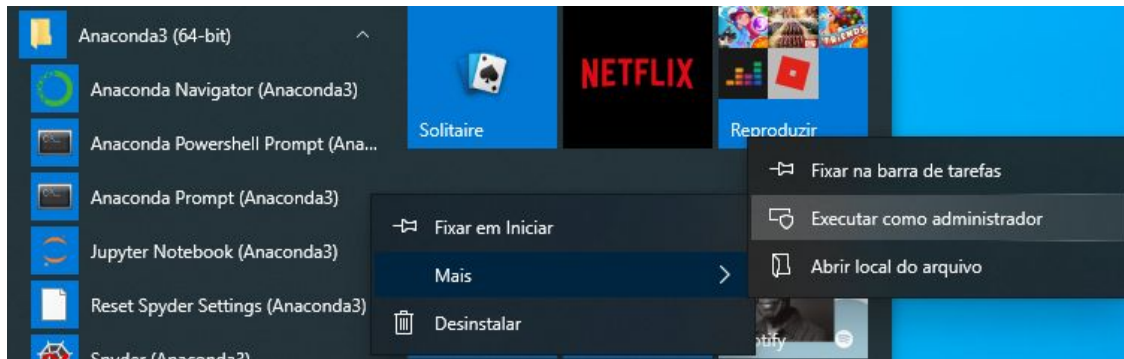
- **Anaconda:** <https://www.anaconda.com/products/individual>
  - Usaremos *Jupyter Notebook* como ambiente de desenvolvimento;
  - Utilizaremos as seguintes bibliotecas para desenvolvimento:
    - *NumPy, Pandas, Scrappy, PyPDF4, Spacy, Pdfminer, db-sqlite3*
- **PgAdmin 4:** <https://www.pgadmin.org/download/>
  - Usaremos a IDE do *PgAdmin 4* para fazer experimentos com banco de dados *plsql*.
- Colab



# Ambiente de Desenvolvimento: DICAS

Para instalar qualquer **biblioteca** que iremos utilizar no **Jupyter Notebook**:

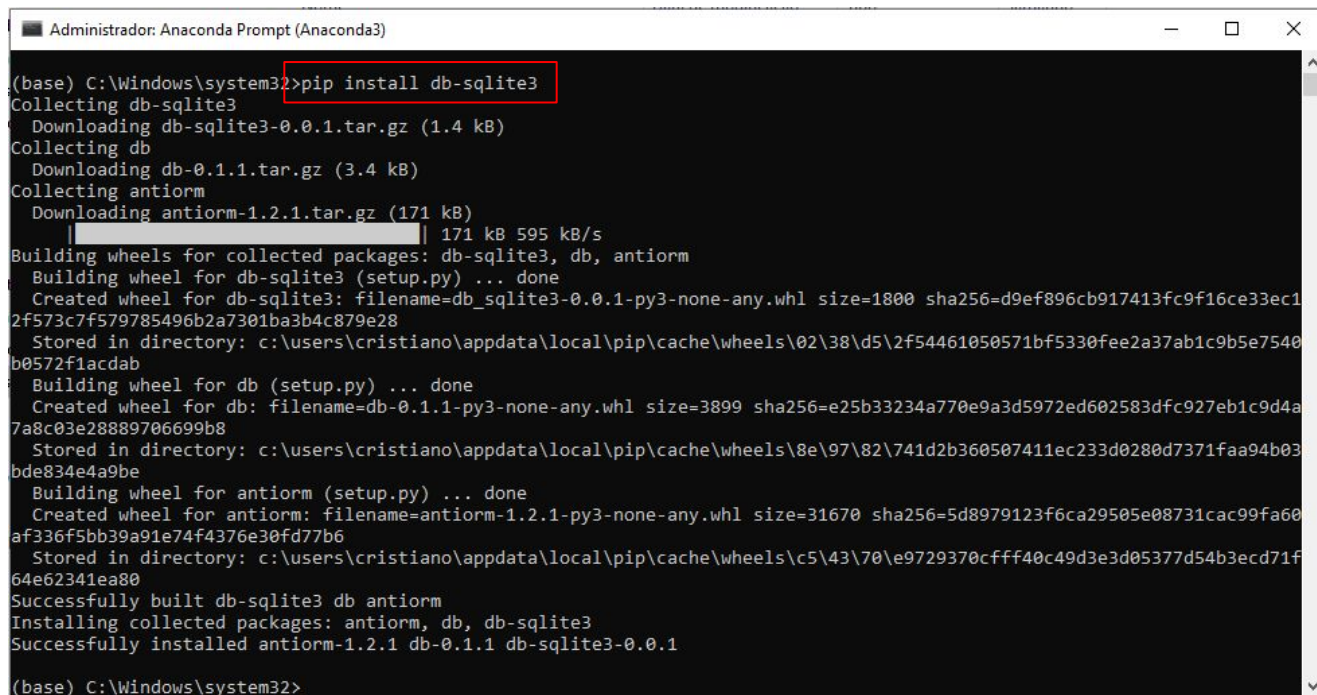
- No Windows:
  - Vá em INICIAR > Anaconda > clique com o botão direito em *Anaconda Prompt* e execute como administrador.



# Ambiente de Desenvolvimento: DICAS

Com o *prompt* aberto, digite o comando sugerido pela documentação da biblioteca desejada.

Ex:



```
Administrador: Anaconda Prompt (Anaconda3)

(base) C:\Windows\system32>pip install db-sqlite3
Collecting db-sqlite3
  Downloading db-sqlite3-0.0.1.tar.gz (1.4 kB)
Collecting db
  Downloading db-0.1.1.tar.gz (3.4 kB)
Collecting antiorm
  Downloading antiorm-1.2.1.tar.gz (171 kB)
    |#####| 171 kB 595 kB/s
Building wheels for collected packages: db-sqlite3, db, antiorm
  Building wheel for db-sqlite3 (setup.py) ... done
  Created wheel for db-sqlite3: filename=db_sqlite3-0.0.1-py3-none-any.whl size=1800 sha256=d9ef896cb917413fc9f16ce33ec1
2f573c7f579785496b2a7301ba3b4c879e28
  Stored in directory: c:\users\cristiano\appdata\local\pip\cache\wheels\02\38\d5\2f54461050571bf5330fee2a37ab1c9b5e7540
b0572f1acdab
  Building wheel for db (setup.py) ... done
  Created wheel for db: filename=db-0.1.1-py3-none-any.whl size=3899 sha256=e25b33234a770e9a3d5972ed602583dfc927eb1c9d4a
7a8c03e28889706699b8
  Stored in directory: c:\users\cristiano\appdata\local\pip\cache\wheels\8e\97\82\741d2b360507411ec233d0280d7371faa94b03
bde834e4a9be
  Building wheel for antiorm (setup.py) ... done
  Created wheel for antiorm: filename=antiorm-1.2.1-py3-none-any.whl size=31670 sha256=5d8979123f6ca29505e08731cac99fa60
af336f5bb39a91e74f4376e30fd77b6
  Stored in directory: c:\users\cristiano\appdata\local\pip\cache\wheels\c5\43\70\e9729370cfff40c49d3e3d05377d54b3ecd71f
64e62341ea80
Successfully built db-sqlite3 db antiorm
Installing collected packages: antiorm, db, db-sqlite3
Successfully installed antiorm-1.2.1 db-0.1.1 db-sqlite3-0.0.1

(base) C:\Windows\system32>
```

**Colab**



# **Definições Preliminares**

# Definindo Ambiente de Programação

Para realização de nossos experimentos, utilizaremos **Jupyter Notebook**.



Motivos:

- *Python 3*;
- Organização;
- Mantém o *log* das execuções;
- Melhor controle do fluxo de trabalho;

```
In [1]: import pandas as pd  
import numpy as np
```

```
In [2]: a = 2
```

```
In [3]: a
```

```
Out[3]: 2
```

```
In [4]: b = a*a
```

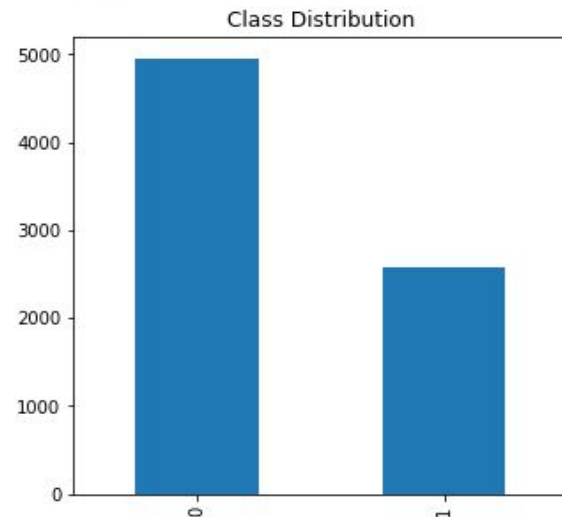
```
In [5]: b
```

```
Out[5]: 4
```

# Definindo Ambiente de Programação

- Permite uso de gráficos em células intermediárias

```
Out[4]: will_change  
0      4954  
1      2582  
Name: will_change, dtype: int64
```



# Bibliotecas



- **Pandas**

- ***Panel datas*** (“dados em painel”);
- Manipulação e análise de dados em Python;
- Torna mais fácil manipulação de diferentes formatos de arquivo (ex: csv);
- *Dataframe*.

# Diferença entre usar lista e *dataframe*

```
In [1]: import pandas as pd
```

```
In [2]: # generating lists  
list_people = [['Gil', 35], ['Gal', 32], ['Zé', 45]]
```

```
In [3]: list_people
```

```
Out[3]: [['Gil', 35], ['Gal', 32], ['Zé', 45]]
```

Para um grande volume de dados, utilizar lista é mais trabalhoso

```
In [4]: # generating dataframe  
list_people_df = pd.DataFrame(list_people)
```

```
In [5]: list_people_df
```

```
Out[5]:
```

	0	1
0	Gil	35
1	Gal	32
2	Zé	45

Visualmente mais intuitivo!

```
In [6]: list_people_df.columns = ['Nome', 'Idade']
```

```
In [7]: list_people_df
```

```
Out[7]:
```

	Nome	Idade
0	Gil	35
1	Gal	32
2	Zé	45

Mais fácil de manipular dado

# Bibliotecas



- NumPy
  - Manipulação algébrica de forma mais fácil:
    - Ordenar vetor, pegar o maior elemento, o menor elemento;
    - Função inversa, transposta, produto interno.
  - Realização de cálculo numérico em operações de *Machine Learning*;
  - “Facilidade” em manipulação de matrizes multidimensionais.

# Obtendo valores estatísticos

```
In [7]: list_people_df
```

```
Out[7]:
```

	Nome	Idade
0	Gil	35
1	Gal	32
2	Zé	45

```
In [8]: list_people_df[['Idade']]
```

```
Out[8]:
```

	Idade
0	35
1	32
2	45

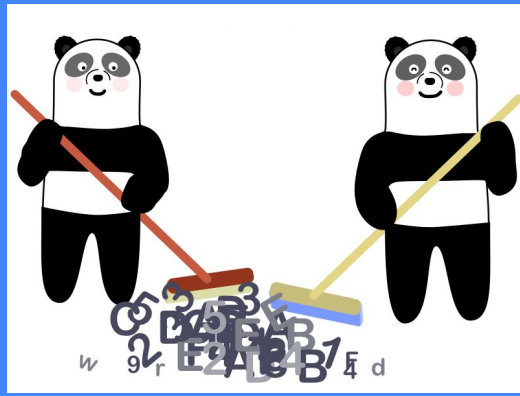
```
In [9]: import numpy as np
```

```
In [10]: mean = np.mean(list_people_df['Idade'])
```

```
In [11]: mean
```

```
Out[11]: 37.333333333333336
```

← usando a biblioteca *numpy*  
para obter a média das idades



# Manipulação e Limpeza de Dados



# Manipulando os Dados

- Como já mencionamos anteriormente, iremos trabalhar utilizando *DataFrames* por serem mais organizados na hora de visualizar, manipular e limpar dados.
- A biblioteca responsável por gerar *DataFrames* em *Python* é a Pandas

```
In [1]: import pandas as pd
```



# Primeiros passos

Ao declararmos um *DataFrame*, podemos fazer de duas maneiras:

1) De forma **default**, no qual apenas declaramos os dados dentro da função `pd.DataFrame`:

```
In [2]: # Dataframe default  
matrix_default_df = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]])
```

```
In [3]: matrix_default_df
```

Out[3]:

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

# Primeiros passos

2) Inserindo os dados e nomeando as linhas (index) e colunas (columns)

```
In [5]: # DataFrame personalizado  
matrix_df = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=(0,'A','$'), columns=['colA', 'colB', 'colC'])
```

```
In [6]: matrix_df
```

Out[6]:

	colA	colB	colC
0	1	2	3
A	4	5	6
\$	7	8	9

# Acessando Linhas e Colunas no DataFrame

- **Linhas:**

Podemos fazer manipulação na linha de duas formas: pelo *index original* ou pelos rótulos dados. Para isso, utilizamos:

- `loc[]`: no qual retorna a linha de acordo com o rótulo dado
- `iloc[]`: faz a busca de acordo com o *index original* do DataFrame

	colA	colB	colC
0	0	1	2
1	A	4	5
2	\$	7	8

```
In [7]: print(matrix_df.loc['A'])
```

```
colA    4
colB    5
colC    6
Name: A, dtype: int64
```

```
In [8]: print(matrix_df.iloc[2])
```

```
colA    7
colB    8
colC    9
Name: $, dtype: int64
```

# Acessando Linhas e Colunas no DataFrame

- **Colunas:**

Podemos fazer manipulação nas colunas simplesmente das seguintes formas:

- Simplesmente digitando o nome da coluna entre colchetes;
- `iloc[]`: utilizando a função `iloc` e usando dois argumentos `[ arg1, arg2 ]`:
  - `arg1`) dois-pontos: para trazer todas as linhas
  - `arg2`) o *index* da coluna

```
In [9]: matrix_df['colA']
```

```
Out[9]: 0    1  
A      4  
$      7  
Name: colA, dtype: int64
```

```
In [10]: matrix_df.iloc[:,0]
```

```
Out[10]: 0    1  
A      4  
$      7  
Name: colA, dtype: int64
```

# Adicionando Linhas e Colunas no DataFrame

- Seja o seguinte DataFrame:

```
✓ [14] df = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=[2.5,12.6, 4.8], columns=[48, 49, 50])  
0s
```

✓  df  
0s



	48	49	50
2.5	1	2	3
12.6	4	5	6
4.8	7	8	9



OBS: Iremos utilizar rótulos “estranhos” para linhas e colunas no intuito de não confundir com os index originais de cada uma.

# Adicionando Linhas e Colunas no DataFrame

- Adicionando Linha:

Se utilizar a função `loc`, ele gerará uma linha nova contendo os valores passados. O rótulo da linha será o valor passado como argumento:

Rótulo da linha

elementos da linha

```
In [47]: df.loc[3] = [11, 12, 13]
df
```

Out[47]:

	48	49	50
2.5	1	2	3
12.6	4	5	6
4.8	60	50	40
3.0	11	12	13

É importante que a quantidade de elementos inseridos seja na mesma quantidade e na respectiva ordem para cada coluna!

# Adicionando Linhas e Colunas no DataFrame

- **Adicionando Linha:**

Se utilizar a função `iloc`, ele apenas irá substituir os valores da linha do index passado como argumento

```
In [13]: df.iloc[2] = [60, 50, 40]  
df
```

Out[13]:

	48	49	50	
0	2.5	1	2	3
1	12.6	4	5	6
2	4.8	60	50	40

  
iloc[]

Como a função `iloc[]` serve para acessar o index original do DataFrame, se você tentar inserir `df.iloc[3]` com esses valores, ele não irá encontrar o index 3 e exibirá mensagem de erro!



# Adicionando Linhas e Colunas no DataFrame

- **Adicionando Coluna:**

Para adicionar uma coluna, basta inserir o nome dentro do argumento e atribuir valores ao dataframe. Neste exemplo, adicionamos uma coluna D com os seus valores iguais aos do index do Dataframe

```
In [15]: df['D'] = df.index  
df
```

Out[15]:

	48	49	50	D
2.5	1	2	3	2.5
12.6	4	5	6	12.6
4.8	60	50	40	4.8
3.0	11	12	13	3.0

# Resetando o index do DataFrame

- Essa função serve para tornar os *index* do DataFrame com formato padrão:

```
In [16]: #Veja os index atuais do seu dataframe  
df
```

Out[16]:

	48	49	50	D
2.5	1	2	3	2.5
12.6	4	5	6	12.6
4.8	60	50	40	4.8
3.0	11	12	13	3.0

Valores de index que  
setamos inicialmente

```
In [17]: # Use `reset_index()` para usar valores padrões  
df_reset = df.reset_index(level=0, drop=True)  
df_reset
```

Out[17]:

	48	49	50	D
0	1	2	3	2.5
1	4	5	6	12.6
2	60	50	40	4.8
3	11	12	13	3.0

Se `drop=False`, ele não descarta a coluna de index antiga, ele gera uma nova coluna no DataFrame e a insere

# Removendo Linhas e Colunas do DataFrame

- Seja o seguinte DataFrame:

```
In [18]: df1 = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=(3,'AA','BB'), columns=['A', 'B', 'C'])  
df1
```

Out[18]:

	A	B	C
3	1	2	3
AA	4	5	6
BB	7	8	9

# Removendo Linhas e Colunas do DataFrame

- **Linhas:** para remover linhas, é possível remover de duas formas:
  - Removendo pelo nome da linha
  - Removendo pelo seu index
- O argumento `axis` serve para indicar o que se deseja remover: 0 para linha e 1 para coluna. O *default* é zero!
- O argumento *inplace* serve para que a remoção seja efetuada sem precisar atribuir o resultado em uma nova variável. Por *default* ela é *False*, e serve apenas para “simular” como ficará o resultado após a remoção.

```
In [19]: df1.drop('AA', axis=0, inplace=True)  
df1
```

Out[19]:

	A	B	C
3	1	2	3
BB	7	8	9

```
In [20]: df1.drop(df1.index[1], axis=0, inplace=True)  
df1
```

Out[20]:

	A	B	C
3	1	2	3

# Removendo Linhas e Colunas do DataFrame

- Seja o seguinte DataFrame:

```
In [21]: df2 = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=(3,'AA','BB'), columns=['A', 'B', 'C'])  
df2
```

Out[21]:

	A	B	C
3	1	2	3
AA	4	5	6
BB	7	8	9

# Removendo Linhas e Colunas do DataFrame

- **Colunas:** para remover colunas, é possível remover de duas formas:
  - Removendo pelo nome da coluna;
  - Removendo pelo seu `columns` (index da coluna).
- Aqui o argumento `axis` deve ser explicitado, visto que o *default* é zero;

```
In [22]: df2.drop('A', axis=1, inplace=True)  
df2
```


Out[22]:

	B	C
3	2	3
AA	5	6
BB	8	9

```
In [23]: df2.drop(df2.columns[0], axis=1, inplace=True)  
df2
```

Out[23]:

	C
3	3
AA	6
BB	9



Cuidado! O index da coluna é obtido através de `.columns[]`, não de `.index[]`

# Renomeando *index* ou *columns*

Seja o seguinte DataFrame:

```
In [24]: df3 = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=(0,0,1), columns=['A', 'B', 'C'])  
df3
```

Out[24]:

	A	B	C
0	1	2	3
0	4	5	6
1	7	8	9

# Renomeando *index* ou *columns*

Renomeando Linhas:

```
In [25]: # Renomeie o index  
df3.rename(index={1: 'a'}, inplace=True)  
df3
```

Out[25]:

	A	B	C
0	1	2	3
0	4	5	6
a	7	8	9

Matriz original

	A	B	C
0	1	2	3
0	4	5	6
1	7	8	9



# Renomeando *index* ou *columns*

Renomeando Colunas:

```
In [26]: # Defina o nome das colunas
newcols = {
    'A': 'new_column_1',
    'B': 'new_column_2',
    'C': 'new_column_3'
}
```

```
In [27]: # Use `rename()` para renomear
df3.rename(columns=newcols, inplace=True)
df3
```

Out[27]:

	new_column_1	new_column_2	new_column_3
0	1	2	3
0	4	5	6
a	7	8	9

Matriz antes de ter a coluna renomeada

	A	B	C
0	1	2	3
0	4	5	6
a	7	8	9

# Formatar dados no DataFrame

Seja o seguinte DataFrame:

```
[43] df4 = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=[2.5,12.6, 4.8], columns=[48, 49, 50])
```

df4



	48	49	50
2.5	1	2	3
12.6	4	5	6
4.8	7	8	9



# Formatar dados no DataFrame

Podemos alterar elementos dentro do DataFrame das seguintes formas:

- 1) Alterando uma lista de elementos:

```
In [29]: # Substituir números por string  
df4.replace([1,2,3,4,5],['Awful', 'Poor', 'OK', 'Acceptable', 'Perfect'])
```

Out[29]:

	48	49	50
2.5	Awful	Poor	OK
12.6	Acceptable	Perfect	6
4.8	7	8	9

# Formatar dados no DataFrame

Podemos alterar elementos dentro do DataFrame das seguintes formas:

2) Utilizando estrutura de repetição na linha:

```
In [30]: for i in range(3):  
         df4.iloc[2,i] = 0  
         df4
```

Out[30]:

	48	49	50
2.5	1	2	3
12.6	4	5	6
4.8	0	0	0

# Formatar dados no DataFrame

Podemos alterar elementos dentro do DataFrame das seguintes formas:

3) Utilizando estrutura de repetição na coluna:

```
In [31]: for i in range(3):  
         df4[48] = 1  
         df4
```

Out[31]:

	48	49	50
2.5	1	2	3
12.6	1	5	6
4.8	1	0	0

# Aplicando função *lambda* no DataFrame

- O que é uma função lambda?
  - Todas as características de uma função lambda são muito parecidas com as funções comuns que já vimos, com exceção de duas coisas: elas não possuem uma definição em código, ou seja, são declaradas como variáveis e não possuem um def próprio; e elas são funções de **uma** linha, que funcionam como se houvesse a instrução *return* antes do comando.
  - Exemplo de uma função tradicional:

```
In [32]: def timesTwo(numero):  
         return numero*2
```

# Aplicando função *lambda* no DataFrame

- Seja o seguinte DataFrame:

```
In [33]: df5 = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], index=(0,1,2), columns=['A', 'B', 'C'])
```

```
In [34]: df5
```

Out[34]:

	A	B	C
0	1	2	3
1	4	5	6
2	7	8	9

# Aplicando função *lambda* no DataFrame

- Para aplicarmos a função definida anteriormente em cada elemento do DataFrame, precisamos fazer da seguinte forma:

```
In [35]: for i in range(len(df5)):
          for j in range(len(df5)):
            df5.iloc[i,j] = timesTwo(df5.iloc[i,j])
```

```
In [36]: df5
```

```
Out[36]:
```

	A	B	C
0	2	4	6
1	8	10	12
2	14	16	18



# Aplicando função *lambda* no DataFrame

- Entretanto, podemos definir em Python uma função Lambda da seguinte forma:

```
In [37]: # seja a seguinte função  
doubler = lambda x: x*2
```

- Para aplicá-la em cada elemento, o DataFrame do Pandas possui uma função chamada ApplyMap:

```
In [38]: df5 = df5.applymap(doubler)  
df5
```

Out[38]:

	A	B	C
0	4	8	12
1	16	20	24
2	28	32	36

# Aplicando função *lambda* no DataFrame

- O DataFrame também possui uma função Apply() para aplicar apenas em elementos desejados em linhas ou colunas:

```
In [39]: # Vamos aplicar a função na coluna 'A'  
df5['A'] = df5['A'].apply(doubler)  
df5
```

Out[39]:

	A	B	C
0	8	8	12
1	32	20	24
2	56	32	36

```
In [40]: # Aplicando na linha de rótulo 0  
df5.loc[0] = df5.loc[0].apply(doubler)  
df5
```

Out[40]:

	A	B	C
0	16	16	24
1	32	20	24
2	56	32	36

# Dúvidas?

Email: [wellington@crateus.ufc.br](mailto:wellington@crateus.ufc.br)



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS DE CRATEÚS

