

Programação II

+

Estruturas de Dados para

Bioinformática

Hugo Pacheco

DCC/FCUP

23/24

Análise de dados tabulares (pandas)

NumPy

- Desenhado para programação numérica sobre arrays
- Otimizado para álgebra de arrays, operações elemento-a-elemento
- Tem algumas limitações para dados menos estruturados:
 - Não permite atribuir nomes a linhas/colunas (e.g., cabeçalhos CSV)
 - Não suporta elementos de tipos diferentes (inteiros, datas, strings, etc)
 - Pouco suporte para transformar a estrutura dos arrays
 - Pouco suporte para lidar com dados mal formatados ou em falta

Arrays \rightsquigarrow Tabelas

- Matrizes = Arrays 2D
- Tabelas \simeq folhas de cálculo \simeq bases de dados

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

- Colunas potencialmente de diferentes tipos
name : `str`, age : `int`, gender : `str`, rating : `float`

Pandas

- Biblioteca construída em cima do *numpy*
- Generalização de arrays 2D para tabelas:
 - Permite atribuir nomes a linhas/colunas
 - Permite diferentes tipos por coluna
 - Oferece operações de manipulação e transformação de tabelas
- Vasto suporte para lidar com dados em falta

Pandas

- Dois tipos principais:
 - *Series*: um vetor de elementos do mesmo tipo, ao estilo de um array 1D *numpy*, com índices (`[0,1,...[` se não definidos)

A	B	C	D	A
10	50	23	70	34

- *DataFrame*: uma tabela com colunas de tipos diferentes; primeira coluna define índices; (`[0,1,...[` se não definidos)
- nomes de colunas únicos; nomes de índices podem ser repetidos

Índice	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Steve	45	Male	3.9
Katie	38	Female	2.78

Pandas (criação)

- Uma *Series* pode ser criada a partir de vários tipos de sequências (listas, dicionários, array *numpy*, etc)

```
>>> s = pd.Series((1,2,3))
>>> s
0    1
1    2
2    3
dtype: int64
>>> type(s.values)
<class 'numpy.ndarray'>
>>> s.index
RangeIndex(start=0, stop=3, step=1)

>>> s =
pd.Series([1,2,3],index=[1,2,5])
>>> type(s.values)
<class 'numpy.ndarray'>
>>> s.index
Int64Index([1, 2, 5],
dtype='int64')
```

```
>>> s = pd.Series({'a':1,'b':2})
>>> s
a    1
b    2
dtype: int64
>>> s.index
Index(['a', 'b'],
dtype='object')
```

Pandas (criação)

- Um *DataFrame* pode ser criado a partir de vários tipos de sequências (listas, dicionários, array *numpy*, *Series*, *DataFrame*, etc)

```
>>> import pandas as pd
>>> pd.DataFrame([1, 2, 3, 4])
   0
0  1
1  2
2  3
3  4
>>> pd.DataFrame([[1, 2], [3, 4],
[5, 6]], index=list("abc"), columns=list("ab"))
   a  b
a  1  2
b  3  4
c  5  6
```

```
>>> pd.DataFrame({'name':
['steve', 'lia'], 'age':
[25, 38], 'sex':
['Male', 'Female']})
   name  age  sex
0  steve   25  Male
1   lia   38  Female
>>>
pd.DataFrame([{'a': 1, 'b':
2},
{'a': 5, 'b': 10, 'c': 20}])
   a  b  c
0  1  2 NaN
1  5 10 20.0
```


NumPy \leq *Pandas*

- Projeção e slices de *Series* funcionam tal como para arrays 1D *numpy*, com a generalização que índices não têm que ser numéricos (*.iloc* para índices numéricos)

```
>>> s =  
pd.Series(range(5), index=1  
ist("abcde"))
```

```
>>> s  
a      0  
b      1  
c      2  
d      3  
e      4  
dtype: int64
```

```
>>> s['a']  
0  
>>> s.iloc[1]  
1
```

```
>>> s['b':'d'] = [10, 20, 30]
```

```
>>> s
```

```
a      0  
b     10  
c     20  
d     30  
e      4
```

```
>>> s[s > 2]
```

```
d      3  
e      4  
dtype: int64
```

NumPy \leq *Pandas*

- Projeção e slices de *DataFrame* funcionam quase como para arrays 2D *numpy*; 1ª dimensão para colunas, 2ª dimensão ou *.loc* para linhas

```
>>> df = pd.DataFrame([[1,2],  
[3,4]],index=list("ab"),columns  
=list("cd"))
```

```
>>> df
```

```
   c  d
```

```
a  1  2
```

```
b  3  4
```

```
>>> df['c']
```

```
a    1
```

```
b    3
```

```
>>> df['c']['a'] = 10
```

```
>>> df
```

```
   c  d
```

```
a 10  2
```

```
b  3  4
```

```
>>> df[df['c'] > 2]
```

```
   c  d
```

```
b  3  4
```

```
>>> df.loc['b']
```

```
c    3
```

```
d    4
```

```
Name: b, dtype: int64
```

```
>>> df.loc['a'] = [5,6]
```

```
>>> df
```

```
   c  d
```

```
a  5  6
```

```
b  3  4
```

```
>>> df[:].loc['b'] = [0,0]
```

```
>>> df
```

```
   c  d
```

```
a  5  6
```

```
b  0  0
```

NumPy \leq *Pandas*

- Operações matemáticas elemento-a-elemento e de agregação sobre *Series* funcionam tal como arrays 1D *numpy*

```
>>> s = pd.Series({'a':1, 'b':2, 'c':3})
>>> s
a      1
b      2
c      3
dtype: int64
>>> s + 2
a      3
b      4
c      5
dtype: int64
>>> s + s
a      2
b      4
c      6
dtype: int64
>>> np.sum(s)
6
```

NumPy \leq *Pandas*

- Operações matemáticas elemento-a-elemento e de agregação sobre *DataFrame* funcionam quase como arrays 2D *numpy*

```
>>> df = pd.DataFrame([(1, 2),
(3, 4)], index=list("ab"), columns=list("cd"))
>>> df
   c  d
a  1  2
b  3  4
>>> (df + df) / 3
   c          d
a  0.666667  1.333333
b  2.000000  2.666667
>>> df.max()
c    3
d    4
dtype: int64
>>> df.max().max()
4
```

NumPy \leq *Pandas*

- Funções universais *numpy* podem ser aplicadas a todos os elementos de uma *Series* ou um *DataFrame*

```
>>> s =  
pd.Series(range(1,3))  
>>> s  
0    1  
1    2  
dtype: int64  
>>> np.power(s,2)  
0    1  
1    4  
dtype: int64
```

```
>>> df = pd.DataFrame([[1,2],  
[4,5]], columns=['A','B'])  
>>> df  
   A  B  
0  1  2  
1  4  5  
>>> np.sin(df * np.pi / 4)  
   A          B  
0  7.071068e-01  1.000000  
1  1.224647e-16 -0.707107
```

Pandas (mapeamento)

- Podemos também aplicar uma função:

```
>>> df = pd.DataFrame([[1, 2],  
[4, 5]], columns=['A', 'B'])
```

```
>>> df
```

```
   A  B  
0  1  2  
1  4  5
```

```
f = lambda x: x.max() -  
x.min()
```

- por colunas

```
>>> df.apply(f, axis=0)
```

```
A      3
```

```
B      3
```

```
dtype: int64
```

- por linhas

```
>>> df.apply(f, axis=1)
```

```
0      1
```

```
1      1
```

```
dtype: int64
```

Pandas (colunas)

- Podem-se renomear as colunas de um *DataFrame*

```
>>> df = pd.DataFrame([[1, 2], [4, 5]],  
columns=['A', 'B'])
```

```
>>> df
```

	A	B
0	1	2
1	4	5

```
>>> df.columns=[True, False]
```

```
>>> df
```

	True	False
0	1	2
1	4	5

Pandas (colunas)

- Podem-se renomear as colunas de um *DataFrame* aplicando uma função ou um mapeamento (dicionário) às colunas existentes, criando um novo objeto

```
>>> df = pd.DataFrame([[1, 2], [4, 5]],  
columns=['A', 'B'])
```

```
>>> df = df.rename(columns={'A': 'a'})
```

```
   a  B  
0  1  2  
1  4  5
```

```
>>> df = df.rename(columns=lambda s: s.lower())
```

```
   a  b  
0  1  2  
1  4  5
```


Pandas (índices)

- Podem-se reordenar os índices de uma *Series* ou de um *DataFrame*, criando um novo objeto

```
>>> s = pd.Series((1,2,3))
>>> s
0      1
1      2
2      3
dtype: int64
>>> s = s.reindex((2,2,1,3))
>>> s
2      3
2      3
1      2
3      NaN
dtype: int64
```

```
>>> df =
pd.DataFrame([{'a':1, 'b':2}, {'a':3, 'b':4}])
>>> df
   a  b
0  1  2
1  3  4
>>> df =
df.reindex([1,0])
>>> df
   a  b
1  3  4
0  1  2
```

Pandas (índices)

- Podem-se alterar os índices de uma *Series* ou de um *DataFrame* aplicando uma função ou um mapeamento (dicionário) aos índices existentes, criando um novo objeto

```
>>> s = pd.Series((1,2,3))
>>> s
0      1
1      2
2      3
dtype: int64
>>> s = s.rename(lambda
i : i+1)
>>> s
1      1
2      2
3      3
dtype: int64
```

```
>>> df =
pd.DataFrame([{'a':1, 'b'
:2}, {'a':3, 'b':4}])
>>> df
   a  b
0  1  2
1  3  4
>>> df =
df.rename({0:'a', 1:'b'})
>>> df
   a  b
a  1  2
b  3  4
```

Pandas (índices)

- Podem-se converter todos os índices em colunas

```
>>> s =  
pd.Series((1, 2, 3), index=  
list("abc"))  
>>> s  
a      1  
b      2  
c      3  
dtype: int64  
>>> s.reset_index()  
   index  a  b  
0      a  1  2  
1      b  2  3  
2      c  3  4
```

```
>>> df =  
pd.DataFrame([{'a': 1, 'b': 2}  
,  
{ 'a': 3, 'b': 4}], index=list("cd"))  
>>> df  
   a  b  
c  1  2  
d  3  4  
>>> df.reset_index()  
   index  a  b  
0      c  1  2  
1      d  3  4
```

Pandas (índices)

- Podem-se converter colunas de um *DataFrame* em índices

```
>>> s =  
pd.Series((1, 2, 3))  
>>> s  
0      1  
1      2  
2      3  
>>> s.index =  
list("abc")  
>>> s  
a      1  
b      2  
c      3  
dtype: int64
```

```
>>> df =  
pd.DataFrame([{'a': 1, 'b': 2},  
               {'a': 3, 'b': 4}])  
>>> df  
   a  b  
0  1  2  
1  3  4  
>>>  
df.set_index(keys='a', inplace=True)  
>>> df  
   b  
a  
1  2  
3  4
```

Exemplo *Pandas*

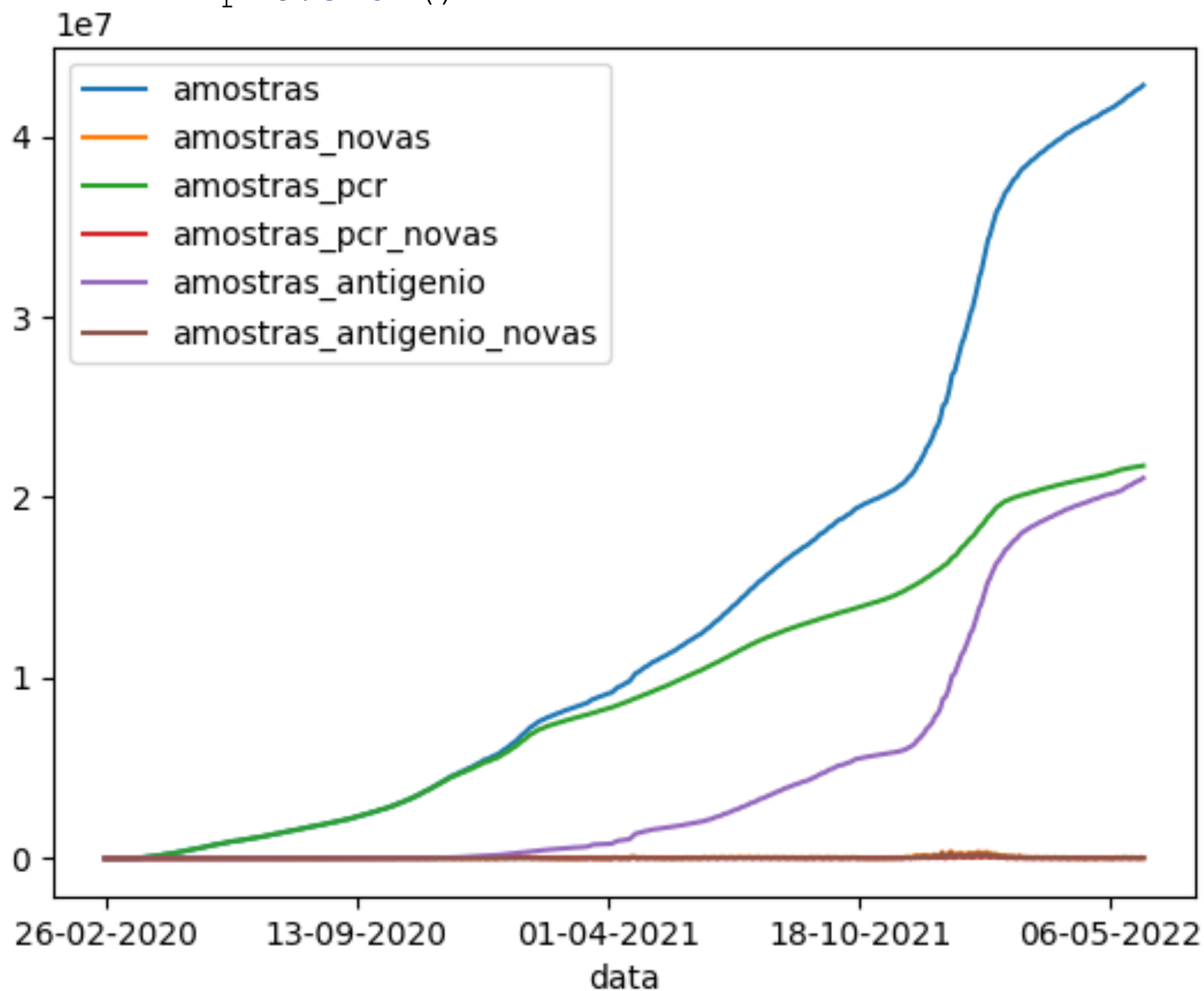
- Ler ficheiro CSV com o número de testes COVID-19 diários, disponível [aqui](#)

```
>>> import pandas as pd
>>> amostras = pd.read_csv('amostras.csv', index_col='data')
>>> amostras.fillna(0, inplace=True)
>>> amostras
```

	amostras	...	amostras_antigenio_novas
data		...	
26-02-2020	0.0	...	0.0
27-02-2020	0.0	...	0.0
28-02-2020	0.0	...	0.0
29-02-2020	0.0	...	0.0
01-03-2020	25.0	...	0.0
...

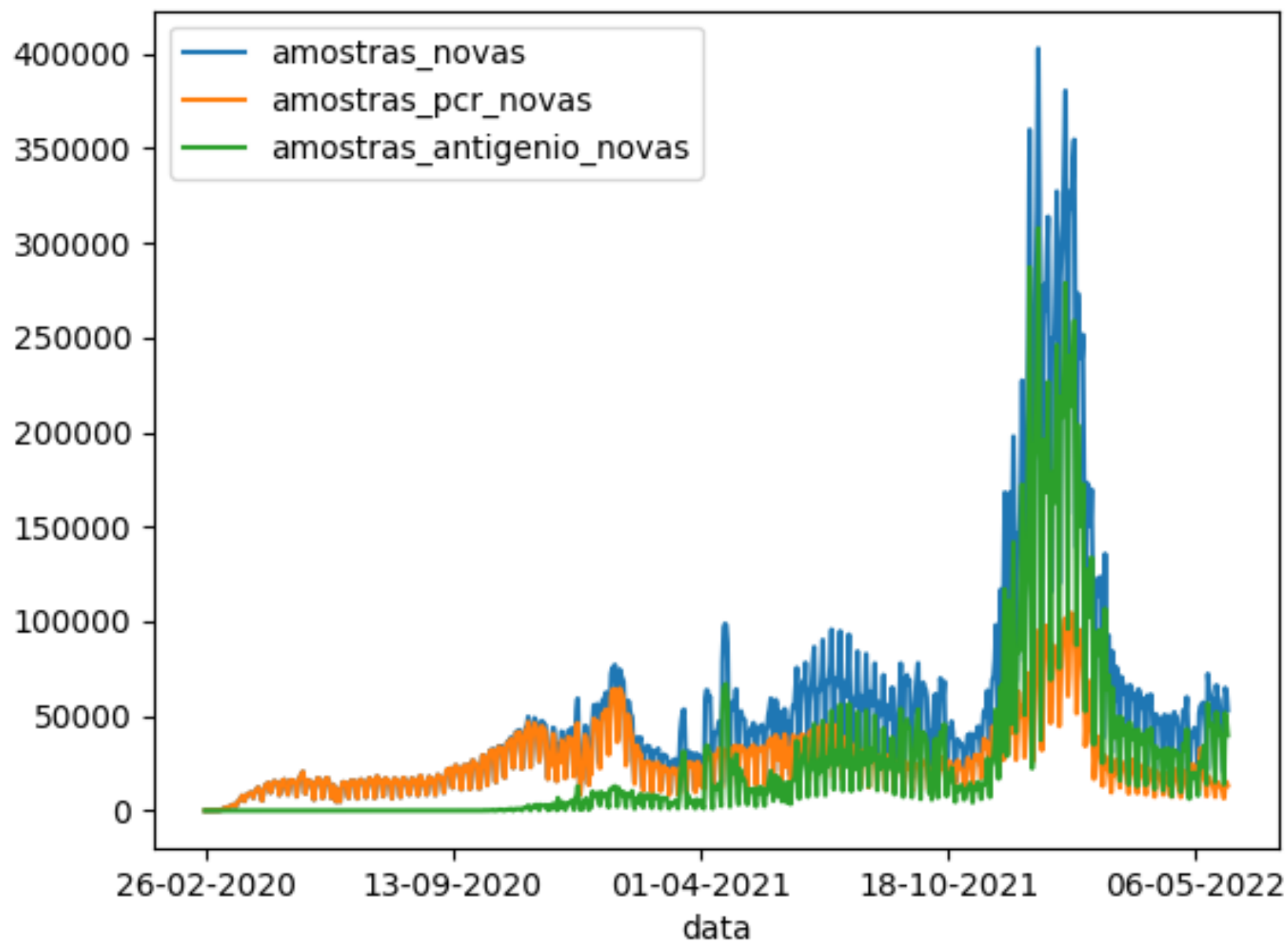
Exemplo *Pandas*

```
import matplotlib.pyplot as plt  
amostras.plot()  
plt.show()
```



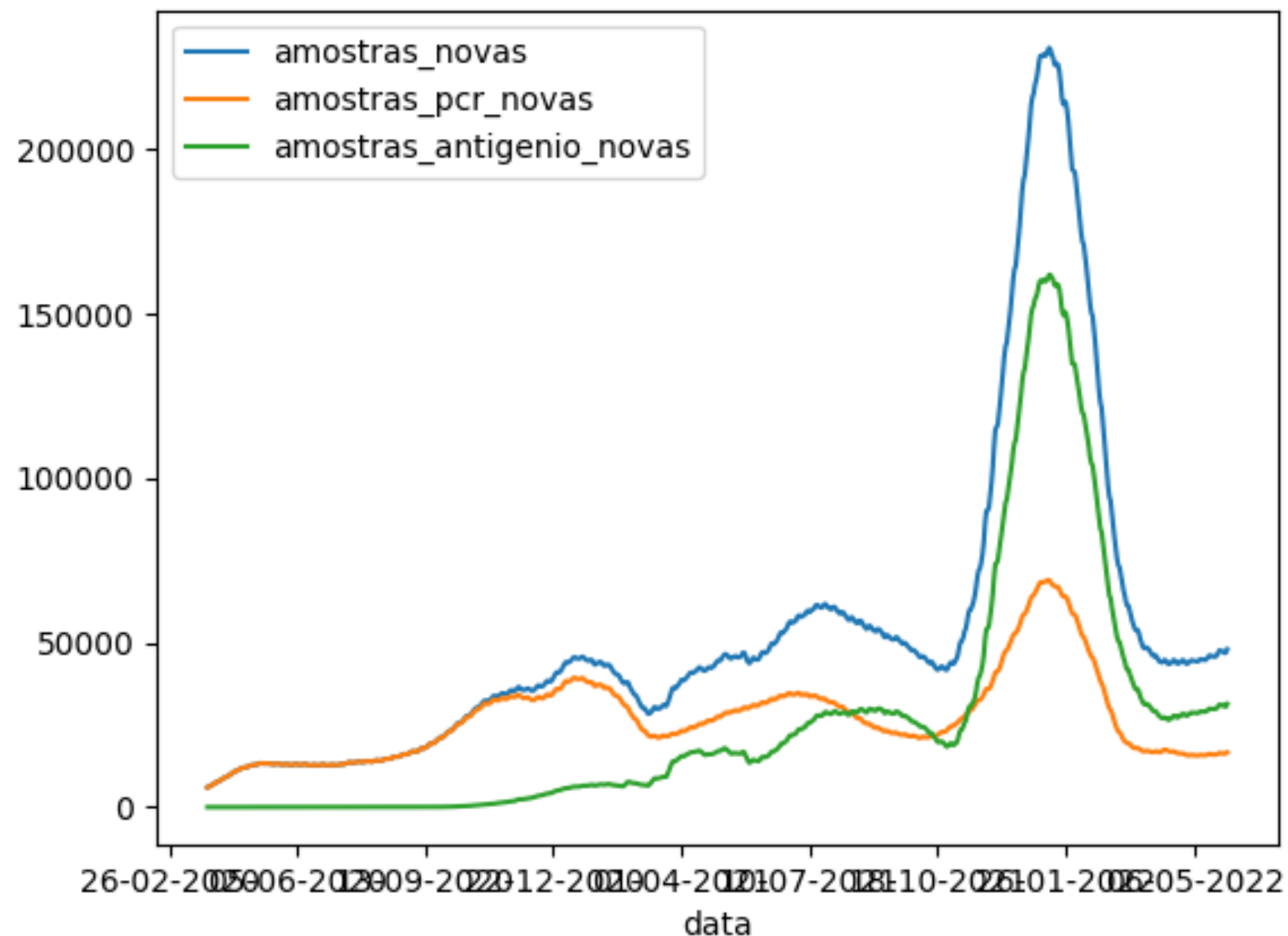
Exemplo *Pandas*

```
novas_colunas = [c for c in amostras.columns if c.endswith('_novas')]  
novas_amstras = amostras[novas_colunas].copy()  
novas_amstras.plot()  
plt.show()
```



Exemplo *Pandas*

```
smooth = novas_amstras.copy()
for i in range(len(smooth)) :
    smooth.iloc[i] = novas_amstras.iloc[i-30:i+30].mean()
smooth.plot()
plt.show()
```



Exemplo *Pandas*

- Algumas estatísticas

```
#1º dia antigénio
```

```
>>> amostras[amostras['amostras_antigenio'] > 0].index[0]
```

```
# dias e valores dos máximos de novas amostras por categoria
```

```
>>> max_datas = novas_amostras.idxmax()
```

```
>>> { k : (data,novas_amostras[k][data]) for k,data in  
dict(max_datas).items() }
```

```
# dias de Dezembro de 2021 acima da média de novas amostras de 2021
```

```
>>> novas = novas_amostras.rename(lambda str :  
pd.to_datetime(str,infer_datetime_format=True))
```

```
>>> novas2021 = novas [ novas.index.year == 2021 ]
```

```
>>> means2021 = novas2021.mean()
```

```
>>> above_mean2021 = (novas2021['amostras_novas'] >=  
means2021['amostras_novas'])
```

```
>>> dec_above_mean2021 = novas2021[above_mean2021 &  
(novas2021.index.month == 12)]
```

Pandas (Dados em falta)

- Dados em falta são assinalados com *NaN* para valores numéricos ou *None* para objetos
- Podemos verificar se há dados em falta
- Podemos apagar todas as linhas com dados em falta
- Podemos apagar todas as colunas com dados em falta

```
>>> df = pd.DataFrame({'age':  
{'John':28,'Anne':35},'status':  
{'John':'Married','Anne':None,'  
Mary':'Single'}})
```

```
>>> df
```

	age	status
John	28.0	Married
Anne	35.0	None
Mary	NaN	Single

```
>>> df.isnull()
```

	age	status
John	False	False
Anne	False	True
Mary	True	False

```
>>> df.dropna()
```

	age	status
John	28.0	Married

```
>>> df.dropna(axis=0)
```

	age	status
John	28.0	Married

```
>>> df.dropna(axis=1)
```

	age	status
--	-----	--------

Pandas (Dados em falta)

- Podemos substituir dados em falta:
- Por um valor fixo
- A partir dos dados próximos
- Utilizando uma função ou mapeamento

```
>>> df.fillna(0)
John    28.0    Married
Anne    35.0         0
Mary     0.0     Single
```

```
>>> df.fillna(method='ffill')
      age    status
John  28.0    Married
Anne  35.0    Married
Mary  35.0     Single
```

```
>>>
df.fillna({'age':0, 'status':'
Unknown'})
John  28.0    Married
Anne  35.0    Unknown
Mary   0.0     Single
```

Pandas (Adicionar colunas)

- Pode-se adicionar colunas:

```
>>> df = pd.DataFrame([{'one':1},  
                        {'one':5, 'two':10}])  
>>> df
```

	one	two
0	1	NaN
1	5	10.0

- com uma nova sequência

```
>>> df['three'] = pd.Series([True, False])  
>>> df
```

	one	two	three
0	1	NaN	True
1	5	10.0	False

- a partir de colunas existentes

```
>>> df['four'] = df['one'] + df['two']  
>>> df
```

	one	two	three	four
0	1	NaN	True	NaN
1	5	10.0	False	15.0

Pandas (Adicionar colunas)

- Pode-se adicionar colunas:
- Juntando um ou mais *DataFrames*

```
>>> df = pd.DataFrame([{'one':1},  
                        {'one':5, 'two':10}])
```

```
>>> df  
   one  two  
0    1  NaN  
1    5 10.0
```

```
>>> df2 = pd.DataFrame([{'three':3}, {}])
```

```
>>> df3 = pd.DataFrame([{}, {'four':4}])
```

```
>>> df = df.join([df2, df3])
```

```
>>> df  
   one  two  three  four  
0    1  NaN   1.0  NaN  
1    5 10.0   NaN   5.0
```

Pandas (Remover colunas)

- Pode-se remover colunas:
 - “in-place”
 - Criando um novo *DataFrame*

```
>>> df = pd.DataFrame([{'one':1, 'two':2},  
                        {'one':5, 'three':10, 'four':4}])
```

```
>>> df  
      one  two  three  four  
0       1  2.0   NaN   NaN  
1       5  NaN  10.0   4.0
```

```
>>> del df['one']
```

```
>>> df  
      two  three  four  
0  2.0     NaN   NaN  
1  NaN    10.0   4.0
```

```
>>> df = df.drop(columns=['two', 'three'])
```

```
>>> df  
      four  
0     NaN  
1     4.0
```

Pandas (Adicionar linhas)

- Pode-se adicionar linhas:

```
>>> df =  
pd.DataFrame([[1, 2, 3]], index=list("a"))  
   0  1  2  
a  1  2  3
```

- Juntando uma sequência

```
>>> df = df.append([[7, 8, 9]])  
   0  1  2  
a  1  2  3  
0  7  8  9
```

- Alterando/criando um índice “in-place”

```
>>> df.loc[0] = [10, 11, 12]  
   0  1  2  
a  1  2  3  
0 10 11 12.0
```

- Concatenando com outro *DataFrame*

```
>>> df2 = pd.DataFrame([[0, 0],  
[7, 8]], index=list("ab"))  
>>> df = pd.concat([df, df2])  
   0  1  2  
a  1  2  3.0  
0 10 11 12.0  
a  0  0  NaN  
b  7  8  NaN
```

Pandas (Remover linhas)

- Pode-se remover linhas:
- Com uma sequência de índices
- Eliminando valores duplicados

```
>>> df = pd.DataFrame([[1,2],[3,4],  
                        [3,4],[5,6]],index=list("abca"))
```

```
>>> df  
      0  1  
a     1  2  
b     3  4  
c     3  4  
a     5  6
```

```
>>> df.drop(index='a')
```

```
      0  1  
b     3  4  
c     3  4
```

```
>>> df.drop_duplicates()
```

```
      0  1  
b     3  4
```


NumPy \leq Pandas

- Algumas operações sobre matrizes *NumPy* têm similares em *Pandas*
- E.g., transposta de matrizes troca colunas com índices

```
>>> df = pd.DataFrame({"A": [3, 4],  
                        "B": [5, 6]}, index=['a', 'b'])
```

```
>>> df  
      A  B  
a     3  5  
b     4  6
```

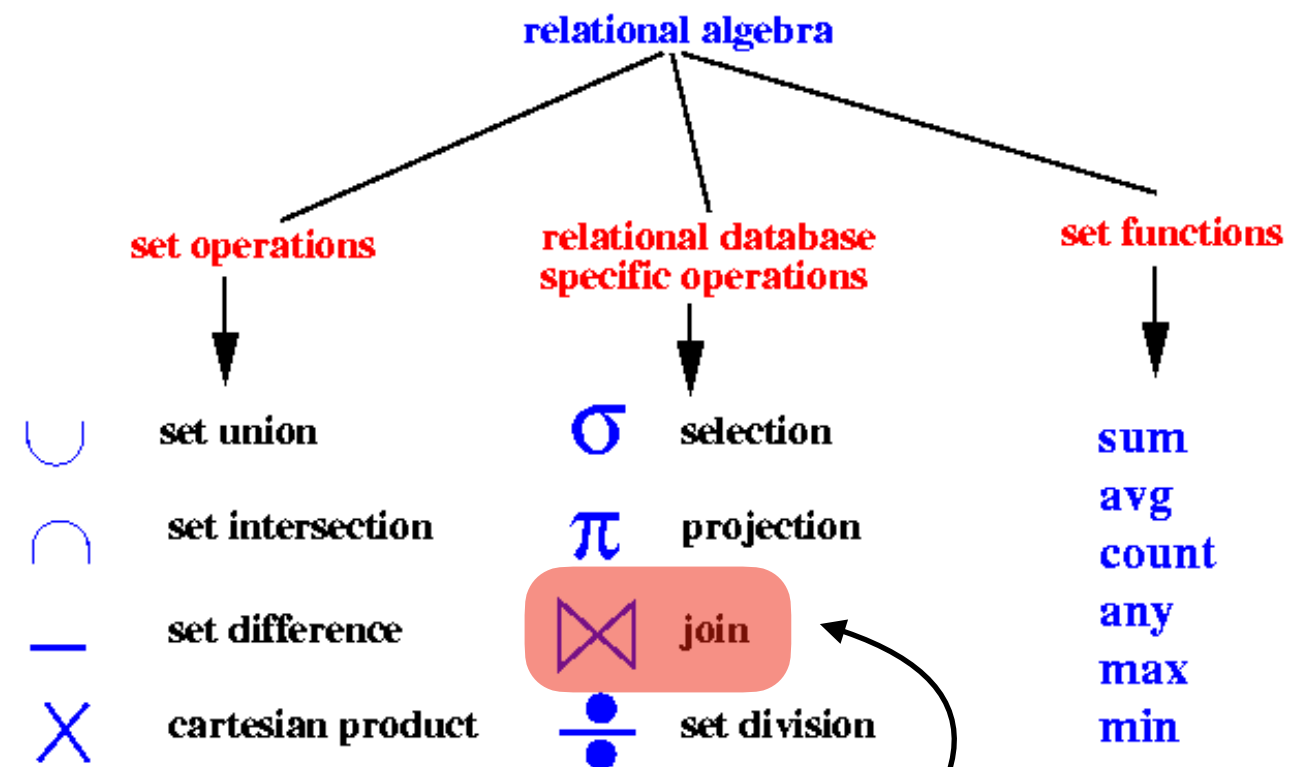
```
>>> df.transpose()  
      a  b  
A     3.0  4.0  
B     5.0  6.0
```

Pandas (álgebra relacional)

- A biblioteca *pandas* também suporta algumas operações de transformação de *DataFrames* reminescentes da **álgebra relacional**

- permite relacionar e combinar dados de diferentes *DataFrames*

- está no cerne das ditas bases de dados relacionais (SQL)



- Vamos olhar de forma informal para algumas delas

Pandas (join)

- Podemos juntar dois *DataFrames* utilizando um índice ou coluna em comum como “chave”
 - Embora não obrigatório, é conveniente assumir que os valores da coluna em cada lado são únicos, daí a nomenclatura “chave”
 - Em álgebra relacional é o chamado “inner join”: o resultado corresponde à união para os elementos da coluna que existam nos dois lados

```
>>> df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa',  
'Mary'], 'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})  
>>> df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],  
'hire_date': [2004, 2008, 2012, 2014]})  
>>> df1; df2
```

	employee	group		employee	hire_date
0	Bob	Accounting	0	Lisa	2004
1	Jake	Engineering	1	Bob	2008
2	Lisa	Engineering	2	Jake	2012
3	Mary	HR	3	Sue	2014

```
>>> pd.merge(df1, df2)
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004

Pandas (join)

- Podemos controlar o critério que determina quando uma linha aparece no resultado
 - “inner join” (por defeito) : resultado tem só os elementos existentes nos dois lados
 - “outer join”: resultado tem elementos existentes em qualquer um dos lados, com valores em falta
 - “left join”: resultado tem elementos existentes do lado esquerdo, com valores em falta do lado direito
 - “right join”: resultado tem elementos existentes do lado direito, com valores em falta do lado esquerdo

```
>>> pd.merge(df1, df2, how='inner')
   employee      group  hire_date
0      Bob  Accounting    2008
1     Jake  Engineering    2012
2     Lisa  Engineering    2004
```

```
>>> pd.merge(df1, df2, how='outer')
   employee      group  hire_date
0      Bob  Accounting    2008.0
1     Jake  Engineering    2012.0
2     Lisa  Engineering    2004.0
3     Mary          HR         NaN
4      Sue          NaN    2014.0
```

```
>>> pd.merge(df1, df2, how='left')
   employee      group  hire_date
0      Bob  Accounting    2008.0
1     Jake  Engineering    2012.0
2     Lisa  Engineering    2004.0
3     Mary          HR         NaN
```

```
>>> pd.merge(df1, df2, how='right')
   employee      group  hire_date
0     Lisa  Engineering    2004
1      Bob  Accounting    2008
2     Jake  Engineering    2012
3      Sue          NaN    2014
```

Pandas (group)

- Podemos agrupar linhas de um *DataFrame* por categorias

```
>>> teams = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',  
    'Kings', 'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',  
    'Riders'], 'Rank': [1, 2, 2, 3, 3, 4, 1, 1, 2, 4, 1, 2], 'Year':  
    [2014, 2015, 2014, 2015, 2014, 2015, 2016, 2017, 2016, 2014, 2015, 2017], 'Po  
ints': [876, 789, 863, 673, 741, 812, 756, 788, 694, 701, 804, 690]}  
>>> df = pd.DataFrame(teams)  
>>> df
```

	Team	Rank	Year	Points
0	Riders	1	2014	876
1	Riders	2	2015	789
2	Devils	2	2014	863
3	Devils	3	2015	673
4	Kings	3	2014	741
5	kings	4	2015	812
6	Kings	1	2016	756
7	Kings	1	2017	788
8	Riders	2	2016	694
9	Royals	4	2014	701
10	Royals	1	2015	804
11	Riders	2	2017	690

Pandas (group)

- Resultado é um “*DataFrame de DataFrames*”

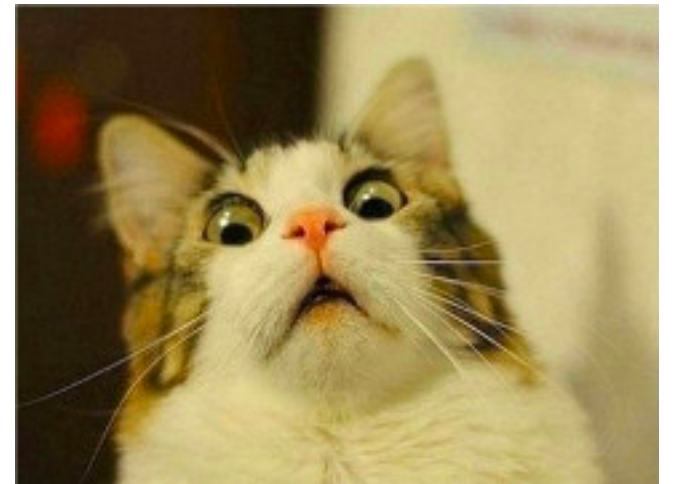
```
>>> ddf = df.groupby('Team')
>>> ddf
<pandas.core.groupby.generic.DataFrameGroupBy
object at 0x10d406070>
>>> for i, df in ddf:
>>>     print(i)
>>>     print(df)
```

Devils

	Team	Rank	Year	Points
2	Devils	2	2014	863
3	Devils	3	2015	673

Kings

	Team	Rank	Year	Points
4	Kings	3	2014	741
6	Kings	1	2016	756
7	Kings	1	2017	788



Pandas (group)

- Podemos agrupar linhas de um *DataFrame* por categorias

```
# ranking e pontos médios por equipa  
>>> df.drop(columns='Year').groupby('Team').mean()
```

	Rank	Points
Team		
Devils	2.500000	768.000000
Kings	1.666667	761.666667
Riders	1.750000	762.250000
Royals	2.500000	752.500000
kings	4.000000	812.000000

```
# número de equipas por ano  
>>> df.groupby('Year').size()
```

Year	
2014	4
2015	4
2016	2
2017	2

Pandas (group)

- Podemos agrupar linhas de um *DataFrame* por categorias

```
# apenas anos com mais de 3
equipas
>>>
df.groupby('Year').filter(lambda
x : len(x) >= 3)
```

	Team	Rank	Year	Points
0	Riders	1	2014	876
1	Riders	2	2015	789
2	Devils	2	2014	863
3	Devils	3	2015	673
4	Kings	3	2014	741
5	kings	4	2015	812
9	Royals	4	2014	701
10	Royals	1	2015	804

```
# máximo de pontos por equipa por
rank
>>>
df.drop(columns='Year').groupby(['Te
am', 'Rank']).max()
```

Team	Rank	
Devils	2	863
	3	673
Kings	1	788
	3	741
Riders	1	876
	2	789
Royals	1	804
	4	701
kings	4	812

Pandas (outras operações)

- Podemos criar uma nova coluna com a soma cumulativa de uma *Series* (e.g., copiada ou calculada a partir de outras colunas)

```
>>> standings = {'Matchday': [1, 2, 3, 4], 'Match':  
['Win', 'Lose', 'Draw', 'Win'], 'Matchpoints': [3, 0, 1, 3]}  
>>> df = pd.DataFrame(standings)  
>>> df.set_index('Matchday', inplace=True, drop=True)  
>>> df
```

	Match	Matchpoints
Matchday		
1	Win	3
2	Lose	0
3	Draw	1
4	Win	3

```
>>> df['Points'] = df['Matchpoints'].cumsum()  
>>> df
```

	Match	Matchpoints	Points
Matchday			
1	Win	3	3
2	Lose	0	3
3	Draw	1	4
4	Win	3	7

Pandas (outras operações)

- Podemos criar uma nova coluna com a diferença de uma *Series* (e.g., copiada ou calculada a partir de outras colunas)

```
>>> standings = {'Matchday': [1, 2, 3, 4], 'Match':  
['Win', 'Lose', 'Draw', 'Win'], 'Matchpoints': [3, 0, 1, 3]}  
>>> df = pd.DataFrame(standings)  
>>> df.set_index('Matchday', inplace=True, drop=True)  
>>> df
```

	Match	Matchpoints
Matchday		
1	Win	3
2	Lose	0
3	Draw	1
4	Win	3

```
>>> df['Form'] = df['Matchpoints'].diff()  
>>> df
```

	Match	Matchpoints	Form
Matchday			
1	Win	3	NaN
2	Lose	0	-3.0
3	Draw	1	1.0
4	Win	3	2.0

Pandas (outras operações)

- Podemos ordenar um *DataFrame* por uma ou mais colunas ou índices.

```
>>> cars = {'Brand': ['Honda Civic', 'Toyota Corolla', 'Ford Focus', 'Audi A4'], 'Price': [22000, 25000, 27000, 35000], 'Year': [2015, 2013, 2018, 2018]}
>>> df = pd.DataFrame(cars, columns=['Brand', 'Price', 'Year'])
>>> df
```

	Brand	Price	Year
0	Honda Civic	22000	2015
1	Toyota Corolla	25000	2013
2	Ford Focus	27000	2018
3	Audi A4	35000	2018

```
>>> df.sort_values(by=['Year', 'Price'])
```

	Brand	Price	Year
1	Toyota Corolla	25000	2013
0	Honda Civic	22000	2015
2	Ford Focus	27000	2018
3	Audi A4	35000	2018

Pandas (outras operações)

- Podemos “derreter” colunas por várias linhas



```
>>> data = pd.DataFrame({'Name': ['José', 'Maria', 'Ana'], 'House':  
['A', 'B', 'A'], 'Age': [32, 46, 25], 'Books': [100, 30, 20], 'Movies': [10, 20, 40]})  
>>> data
```

	Name	House	Age	Books	Movies
0	José	A	32	100	10
1	Maria	B	46	30	20
2	Ana	A	25	20	40

```
>>> data.melt(id_vars=['Name', 'House'], value_vars=['Age', 'Books', 'Movies'])
```

	Name	House	variable	value
0	José	A	Age	32
1	Maria	B	Age	46
2	Ana	A	Age	25
3	José	A	Books	100
4	Maria	B	Books	30
5	Ana	A	Books	20
6	José	A	Movies	10
7	Maria	B	Movies	20
8	Ana	A	Movies	40

Exemplo *Pandas*

- Relembrando uma das tabelas de novos testes diários COVID-19

```
>>> novas
```

```

amostras_novas  amostras_pcr_novas  ...
data
2020-02-26      0.0                0.0  ...
2020-02-27      0.0                0.0  ...
2020-02-28      0.0                0.0  ...
2020-02-29      0.0                0.0  ...
2020-01-03     25.0               25.0  ...
...           ...                ...  ...

```

```
>>> novas.info()
```

```
DatetimeIndex: 827 entries, 2020-02-26 to 2022-06-01
```

```
Data columns (total 3 columns):
```

#	Column	Non-Null Count	Dtype
0	amostras_novas	784 non-null	float64
1	amostras_pcr_novas	784 non-null	float64
2	amostras_antigenio_novas	784 non-null	float64

Exemplo *Pandas*

- Calcular a média de novos testes por mês

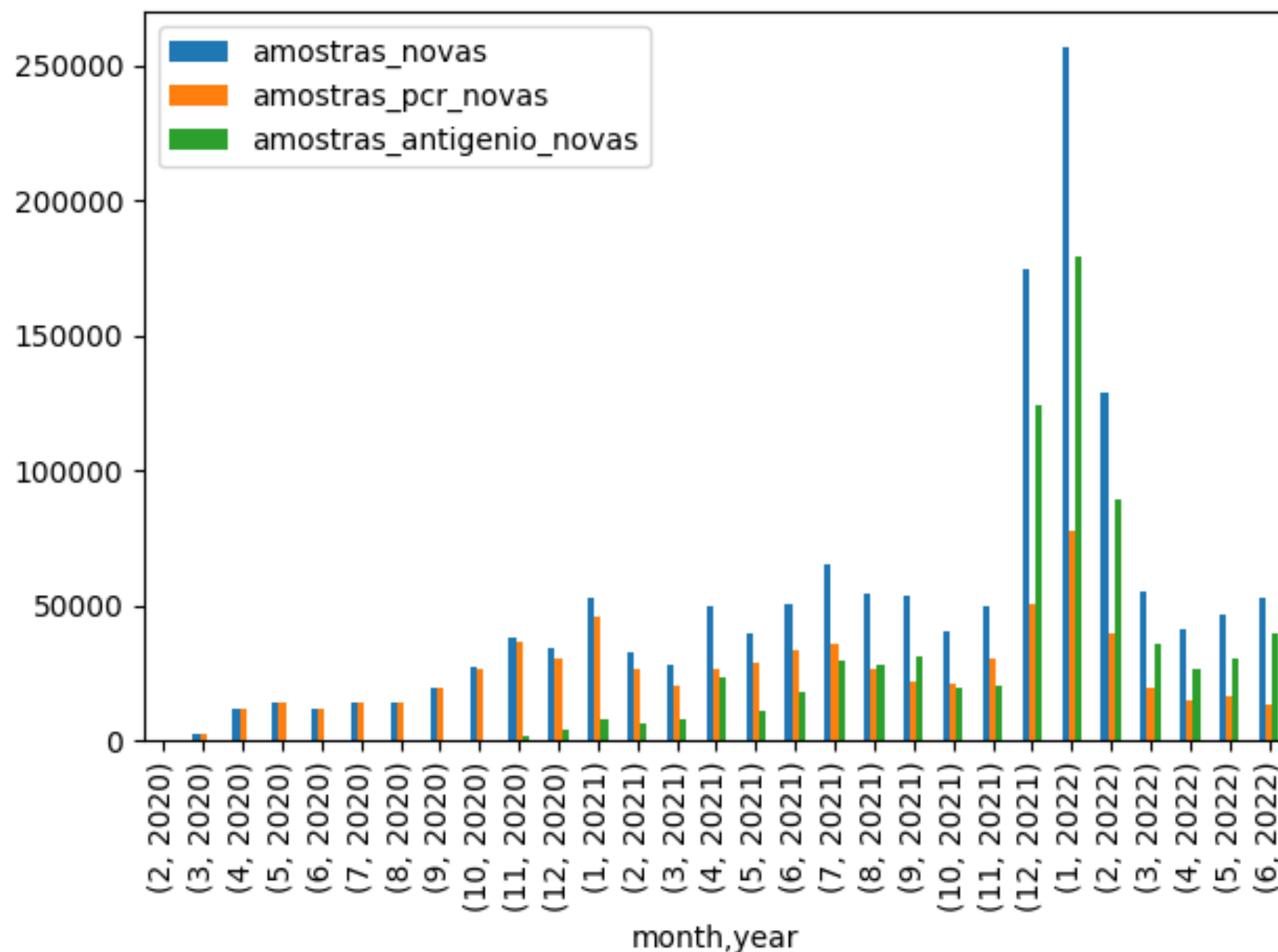
```
# adicionar duas novas colunas para mês e ano
>>> novas['month'] = novas.index.month
>>> novas['year'] = novas.index.year
# agrupar linhas por (mês,ano)
>>> meses = novas.groupby(['month', 'year']).mean()
# ordenar por data crescente
>>> meses.sort_values(by=['year', 'month'], inplace=True)
>>> meses
```

month	year	amostras_novas	amostras_pcr_novas	amostras_antigenio_novas
1	2020	15182.000000	15113.200000	68.800000
2	2020	12055.642857	11757.214286	298.428571
3	2020	9034.862069	8901.413793	133.448276
4	2020	15008.285714	14843.142857	165.142857
...

Exemplo *Pandas*

- Calcular a média de novos testes por mês

```
meses.plot(kind='bar')  
plt.tight_layout()  
plt.show()
```



Exemplo *Pandas*

- Cruzar os dados de números de testes e o número de casos confirmados, disponível aqui

```
>>> amostras = pd.read_csv('amostras.csv', index_col='data')
>>> amostras = amostras['amostras_novas']
>>> amostras.fillna(0, inplace=True)

>>> dados = pd.read_csv('data.csv', index_col='data')
>>> confirmados = dados['confirmados_novos']

>>> amostras_confirmados =
pd.merge(amostras, confirmados, how='inner', left_index=True, right_index=True)
>>> amostras_confirmados
```

	amostras_novas	confirmados_novos
data		
26-02-2020	0.0	0.0
27-02-2020	0.0	0.0
...

Exemplo *Pandas*

```
>>> amostras_confirmados.corr()  
amostras_novas      confirmados_novos  
amostras_novas      1.000000      0.816434  
confirmados_novos    0.816434      1.000000
```

- Verificar que existe uma forte correlação entre testagem e casos COVID-19

