

# Aula 03| PosTech | Análise Exploratória de Dados

Anotações sobre a terceira aula da PosTech FIAP ✨✨

<https://on.fiap.com.br/mod/conteudoshtml/view.php?id=307787&c=8729&sesskey=AlUOg2UtXh>

## Temas abordados:

- Análise envolvendo índice, plots, aleatoriedade e série temporal em dataframes;
- Como os números pseudoaleatórios são gerados;
- Como obter uma amostra aleatória dos dados;

## Dependências:

- Baixar o arquivo zip:

<https://github.com/alura-tech/pos-datascience-analise-e-exploracao-de-dados/tree/aula1>

- Documentação Pandas:

<https://pandas.pydata.org/>

- Documentação Matplotlib:

<https://matplotlib.org/>

- Google Colab:

<https://colab.research.google.com/>

- TabNet:

<https://datasus.saude.gov.br/informacoes-de-saude-tabnet/>

- Leitura de artigo:

<https://www.cambridge.org/engage/api-gateway/coe/assets/orp/resource/item/61b410fadcbea24f839f0235/original/itamaraca-a-novel-simple-way-to-generate-pseudo-random-numbers.pdf>

## Aula 3 - Manipulação de Dados

**Biblioteca Pandas** → nos traz muito conteúdo para visualizar dados e as funções que facilitam a nossa vida.

**Python** → Uma das vantagens é usar a tipagem dinâmica para guardar os Dataframes com o que realmente importa e também perceber que a função .plot() do Pandas nos permite trazer um gráfico sem rodar uma biblioteca gráfica previamente declarada com a palavra reservada “import”.

O uso de bibliotecas como o **Seaborn, Matplotlib e Plotly** ainda é importante. Elas são bibliotecas completas e que nos dão **completude em análises gráficas**, então não as descarte e analise com cuidado, já que o .plot() do Pandas é excelente para análises rápidas dentro do seu Dataframe a ser analisado.

Outro ponto importante a ser analisado no tipo de base que estamos lidando é o fato da análise obrigatoriamente nos conduzir à um problema de **cunho temporal**, e isso será o princípio do que mais tarde chamaremos de **séries temporais**.

Outros pontos para uma análise importante em experimentos que a computação traz do método científico são a **reprodutibilidade** e **aleatoriedade**.

Reprodutibilidade → quando falamos em análise de dados que tem como um dos seus pilares a reprodutibilidade, significa simplesmente que, se por um acaso você, ao analisar uma base, obter o resultado “X”, se outra pessoa em outro lugar do planeta quiser fazer a mesma coisa que você, **ela deve obter o mesmo resultado “X”**. Pois uma análise com resultados que não pode ser reproduzida por outra pessoa, é no mínimo suspeita e um tanto enviesada, não acha?

Aleatoriedade → é importante para que, ao analisar os dados, independente do dado em questão, não nos apeguemos com especificidades, ou seja, a análise deve servir para qualquer valor.

Ordenação de dados analisados → “Por que eu deveria organizar as coisas, já que antes foram mencionados os dados aleatórios?” → Há uma diferença entre dados aleatórios e dados desorganizados → a escolha dos dados para serem analisados pode ser aleatória, mas é interessante que você os ordene conforme sua necessidade e problema de negócio.

No Python, existe uma solução dentro da biblioteca Numpy, onde chamamos uma linha de código antes de começar o resto do programa:

```
np.random.seed(valor)
```

Esse trecho garante um estado de aleatoriedade salvo, ou seja, se chamarmos essa função com um valor fixo, várias vezes, o computador exibirá os mesmos números aleatórios. Porém, usar isso globalmente pode ser ruim em projetos grandes, já que isso pode *afetar o módulo numpy.random.\* como um todo*.

O ideal, nesse caso, é criar um **objeto gerador** e utilizar na medida do possível. Essa convenção foi criada por Robert Kern e se encontra em uma normativa da documentação do Numpy chamada NEP19:

<https://numpy.org/neps/nep-0019-rng-policy.html>

Imagine que você tenha o seguinte trecho de código:

```
import numpy as np

np.random.seed(1)
array = np.random.rand(5)
np.random.seed(1)
array2 = np.random.rand(5)
print(array)
print(array2)
```

Como nosso 'seed' foi repetido entre as duas variáveis array, é de se esperar que encontremos um valor igual:

```
[4.17022005e-01  7.20324493e-01  1.14374817e-04  3.02332573e-01  1.46755891e-01]
[4.17022005e-01  7.20324493e-01  1.14374817e-04  3.02332573e-01  1.46755891e-01]
```

Aqui está bem evidente a garantia de reprodutibilidade. Contudo, olhemos agora para uma questão onde apenas chamamos o seed uma vez:

```
import numpy as np

np.random.seed(1)
```

```
array = np.random.rand(5)

array2 = np.random.rand(5)
print(array)
print(array2)
```

O output será:

```
[4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01 1.46755891e-01]
[0.09233859 0.18626021 0.34556073 0.39676747 0.53881673]
```

No primeiro array, era de se esperar que mantivéssemos o valor do bloco de código anterior, mas veja que a **variável array2 assume um novo valor**, já que pela lei de formação de dados pseudoaleatórios, **o valor será diferente**, porque a aleatoriedade se garantiu apenas na primeira chamada após o código.

Computadores são incapazes de gerar um número verdadeiramente aleatório, porque os computadores são determinísticos e seguem consistentemente um determinado conjunto de instruções.

A ideia por trás disso é que sempre obteremos o mesmo conjunto de números aleatórios para a mesma “semente” (seed) em qualquer máquina.

Historicamente, temos algoritmos geradores de números pseudoaleatórios, desde John von Neumann, em 1946, até um algoritmo (Itamaracá) criado por um brasileiro chamado Daniel Henrique Pereira, em 2021.

⚠ O mais importante nisso tudo é ter em mente que o processo aleatório em máquinas jamais será 100%, já que o próprio hardware é determinístico. Em contrapartida, a pseudoaleatoriedade nos traz os benefícios de **algo que podemos gerar reprodutibilidade**, sem nos preocuparmos se garantiremos isso de forma escalável. 💣

### Criando números aleatórios na unha #01

Sempre fui curioso em relação a aleatoriedade. Já havia escutado que em geral nossos números aleatórios são, na verdade, pseudo aleatórios.

▶ <https://www.youtube.com/watch?v=p5-uRvEK5WE>



### Criando números aleatórios na unha #02

Continuando com a sugestão que peguei na conversa com as professoras do Peixe Babel, fui agora analisar a distribuição dos números pseudo aleatórios gerados e tentar resolver o problema

▶ <https://www.youtube.com/watch?v=NuQIJcEOjBY>




## 1 | MANIPULAÇÃO DE DADOS 01

Continuando com a mesma base da aula anterior...


Agora, considerando que a gente queira **mais de um mês**, ou seja, duas colunas.

□ `dados[["2008/Ago", "2008/Set"]].head()`

0s  `dados[['2008/Ago', '2008/Set']].head()`

	2008/Ago	2008/Set
0	2938286.29	2843930.91
1	1886871.84	2084884.80
2	8331763.97	8148089.92
3	1045291.11	1003740.89
4	23259148.28	23667690.01

Se a gente for dar uma olhada nos dados usando `dados.head()`, podemos notar que tem meses que não tem valores. Quero trabalhar com meses que tem valores de todas as unidades, sem faltar nenhuma.

0s  `dados.head()`

	Unidade da Federação	1992/Mar	1992/Abr	1993/Mai	1993/Dez	1994/Jan	1994/Fev	1994/Mai	1994/Ago	1994/Nov	...	2022/Ago	2022/Set	2022/Out
0	11 Rondônia	-	-	-	-	-	-	-	-	-	...	9525352.64	10309106.47	10026702.19
1	12 Acre	-	-	-	-	-	-	-	-	-	...	3894498.82	3529504.71	4040340.45
2	13 Amazonas	-	-	-	-	-	-	-	-	-	...	21958050.33	20685834.98	20322950.82
3	14 Roraima	-	-	-	-	-	-	-	-	-	...	4618378.13	4651406.91	4406762.46
4	15 Pará	-	-	-	-	-	-	-	-	-	...	46369977.08	45602504.20	45458406.27

5 rows x 312 columns

Se eu quiser todas as colunas:

☐ `dados.columns`

```
0s dados.columns
Index(['Unidade da Federação', '1992/Mar', '1992/Abr', '1993/Mai', '1993/Dez',
      '1994/Jan', '1994/Fev', '1994/Mai', '1994/Ago', '1994/Nov',
      ...,
      '2022/Ago', '2022/Set', '2022/Out', '2022/Nov', '2022/Dez', '2023/Jan',
      '2023/Fev', '2023/Mar', '2023/Abr', 'Total'],
      dtype='object', length=312)
```

Como já foi dito em aulas anteriores, para tirar a média dos dados:

☐ `dados.mean()`


```
0s dados.mean()
<ipython-input-30-99af3ba13e28>:1: FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future version, it will
dados.mean()
2007/Ago      588123.21
2007/Set      1519994.43
2007/Out      4706028.57
2007/Nov      10640611.38
2007/Dez      23793093.17
...
2023/Jan      111302102.69
2023/Fev      90164782.52
2023/Mar      87925024.90
2023/Abr      40539206.93
Total        15399758873.16
Length: 190, dtype: float64
```

Se olhar bem pra média, ela começa em Agosto de 2007, porque tem meses que tem valor vazio, que não foi preenchido. Quando não é preenchido, não dá para tirar a média. Portanto, o `dados.mean()` devolve uma série, mas isso só funciona quando temos todos os valores, além do índice na série ter o nome do índice da coluna em que estamos interessados.

Se transformarmos esse índice em uma lista do python...

☐ `dados.mean().index.tolist()`



```
0s  dados.mean().index.tolist()

<ipython-input-31-29c7e08b1f6>
dados.mean().index.tolist()
['2007/Ago',
'2007/Set',
'2007/Out',
'2007/Nov',
'2007/Dez',
'2008/Jan',
'2008/Fev',
'2008/Mar',
'2008/Abr',
'2008/Mai',
'2008/Jun',
'2008/Jul',
'2008/Ago',
'2008/Set',
'2008/Out',
'2008/Nov',
'2008/Dez',
'2009/Jan',
'2009/Fev',
'2009/Mar',
'2009/Abr',
'2009/Mai',
'2009/Jun',
'2009/Jul',
'2009/Ago',
'2009/Set',
'2009/Out',
'2009/Nov',
'2009/Dez',
```

Isso é uma lista em python. 😊

Passos:

- 1 | determinamos que tem campos vazios e como lidar com isso: queremos ou não?
- 2 | descobri quais são as colunas totalmente preenchidas através de `.mean()`;
- 3 | peguei o índice que são os nomes dessas colunas;
- 4 | transformei elas em uma lista de python;


Agora, vou declarar uma variável para receber esse valor:

☐ `colunas_usaveis = dados.mean().index.tolist()`



☐ `colunas_usaveis.insert(0, "Unidade da Federação")`

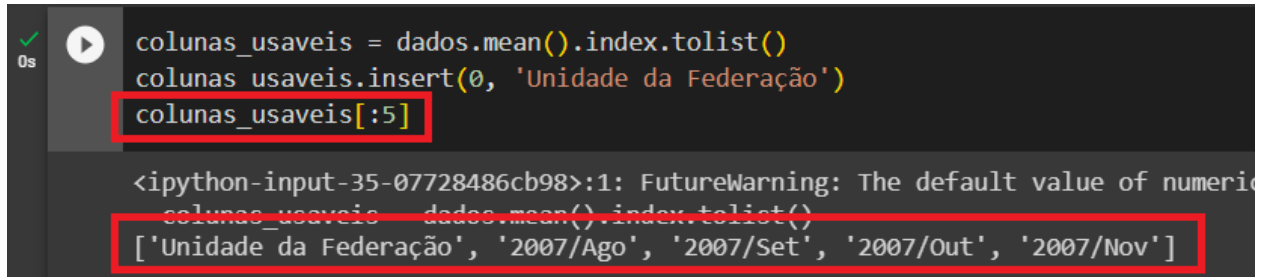
```
✓ 0s ▶ colunas_usaveis = dados.mean().index.tolist()  
colunas_usaveis.insert(0, 'Unidade da Federação')  
colunas_usaveis
```

 <ipython-input-34-154a372ef4f9>:1: FutureWarning: The  
colunas\_usaveis = dados.mean().index.tolist()  
['Unidade da Federação',  
'2007/Ago',  
'2007/Set',  
'2007/Out',  
'2007/Nov',  
'2007/Dez',  
'2008/Jan',  
'2008/Fev',  
'2008/Mar',  
'2008/Abr',  
'2008/Mai',  
'2008/Jun',  
'2008/Jul',  
'2008/Ago',  
'2008/Set',  
'2008/Out',  
'2008/Nov',  
'2008/Dez',  
'2009/Jan',  
'2009/Fev',  
'2009/Mar',  
'2009/Abr',  
'2009/Mai',  
'2009/Jun',  
'2009/Jul',  
'2009/Ago',  
'2009/Set',  
'2009/Out',  
'2009/Nov',

Agora, temos todas as colunas, inclusive “Unidade da Federação”.

Na hora de imprimir, vamos apenas escolher 5 elementos da lista de python (não de série):

☐ `colunas_usaveis[:5]`



```
colunas_usaveis = dados.mean().index.tolist()
colunas_usaveis.insert(0, 'Unidade da Federação')
colunas_usaveis[:5]
```

```
<ipython-input-35-07728486cb98>:1: FutureWarning: The default value of numeric
colunas_usaveis = dados.mean().index.tolist()
['Unidade da Federação', '2007/Ago', '2007/Set', '2007/Out', '2007/Nov']
```

Detalhe que **.tolist()** é muito diferente de **.head()**! não estamos chamando uma série do pandas. Uma lista de python tem o método insert.

Agora que temos `colunas_usaveis`, podemos usar elas:

☐ `dados[colunas_usaveis]`

Os

☐ `dados_usaveis = dados[colunas_usaveis]`

☐ `dados_usaveis.head()`

	Unidade da Federação	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr	...	2022/Ago	2022/Dez
0	11 Rondônia	4209.37	16397.03	133645.19	346912.84	711758.31	1829559.71	1940792.63	1955721.68	2143353.81	...	9525352.64	103091
1	12 Acre	10470.07	14001.71	86200.85	301323.68	769612.36	1731744.62	1743978.66	2057439.02	2057829.69	...	3894498.82	35295
2	13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13	8531576.49	...	21958050.33	206858
3	14 Roraima	4785.91	11858.63	43852.67	369328.51	470676.43	808448.39	771687.83	876091.18	896952.53	...	4618378.13	46514
4	15 Pará	181159.29	433414.74	1893197.50	5105406.44	13162823.43	21762104.16	20126081.01	22149375.82	23436682.75	...	46369977.08	456025

5 rows x 191 columns

Agora, eu espero que todos esses valores tenham algo preenchido. 😊

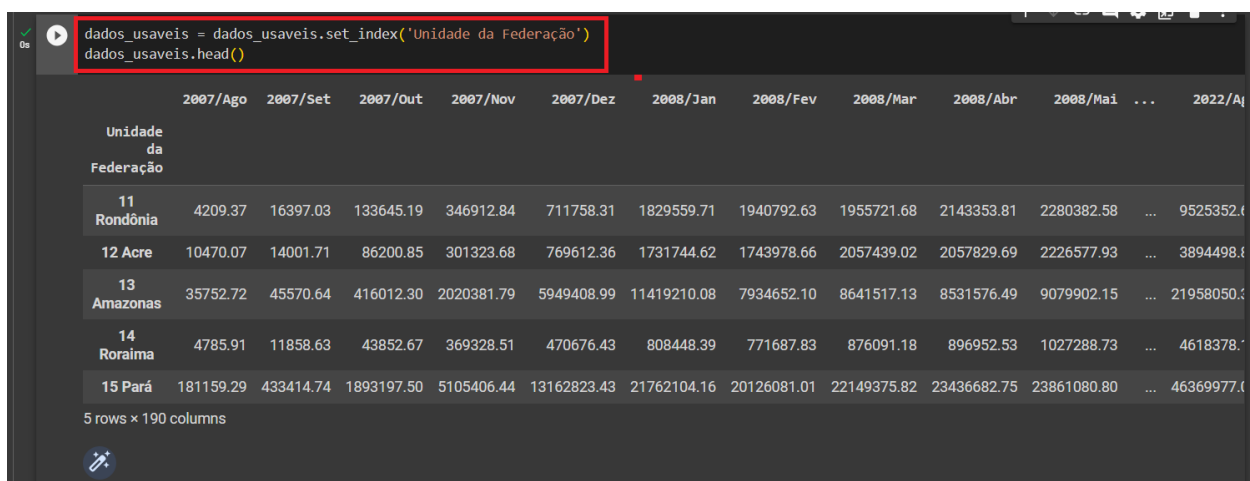
As séries do pandas tem um vetor de índice e um vetor de valor, enquanto que as listas são só um vetor de valores. Isso tem várias implicações porque, como objetos de series do pandas assim como objetos do python tem uma série de métodos distintos, têm aplicações diferentes.

O índice em uma série é quem coordena tudo, era a unidade da federação. Agora, não, é 0 1 2 3 4... Quero que cada linha represente uma unidade da federação:

☐ `dados_usaveis = dados_usaveis.set_index('Unidade da Federação')`

☐ `dados_usaveis.head()`

Agora, a unidade da federação virou o índice! o 0 1 2 3 4 (...) desapareceu.



The screenshot shows a Jupyter Notebook interface. The code cell contains two lines of Python code: `dados_usaveis = dados_usaveis.set_index('Unidade da Federação')` and `dados_usaveis.head()`. The output is a pandas DataFrame with 5 rows and 190 columns. The index column is labeled 'Unidade da Federação' and contains values 11, 12, 13, 14, and 15. The columns are labeled with months and years, starting from '2007/Ago' and ending with '2022/Ago'. The first five rows of data are shown, corresponding to the index values 11 through 15.

	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr	2008/Mai	...	2022/Ago
11 Rondônia	4209.37	16397.03	133645.19	346912.84	711758.31	1829559.71	1940792.63	1955721.68	2143353.81	2280382.58	...	9525352.6
12 Acre	10470.07	14001.71	86200.85	301323.68	769612.36	1731744.62	1743978.66	2057439.02	2057829.69	2226577.93	...	3894498.8
13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13	8531576.49	9079902.15	...	21958050.3
14 Roraima	4785.91	11858.63	43852.67	369328.51	470676.43	808448.39	771687.83	876091.18	896952.53	1027288.73	...	4618378.7
15 Pará	181159.29	433414.74	1893197.50	5105406.44	13162823.43	21762104.16	20126081.01	22149375.82	23436682.75	23861080.80	...	46369977.6

5 rows x 190 columns

☐ `dados_usaveis['2019/Ago']`

```
✓ 0s dados_usaveis['2019/Ago']  
[ ]> Unidade da Federação  
11 Rondônia  
8909111.89  
12 Acre  
2981072.98  
13 Amazonas  
16493719.52  
14 Roraima  
3181321.82  
15 Pará  
36462011.47  
16 Amapá  
2173060.89  
17 Tocantins  
7140062.79  
21 Maranhão  
31374933.66  
22 Piauí  
17896496.59  
23 Ceará  
51672585.01  
24 Rio Grande do Norte  
23073542.40
```

Agora, quero imprimir apenas o head:

☐ `dados_usaveis['2019/Ago'].head()`



```
dados_usaveis['2019/Ago'].head()
```

Unidade da Federação	
11 Rondônia	8909111.89
12 Acre	2981072.98
13 Amazonas	16493719.52
14 Roraima	3181321.82
15 Pará	36462011.47

Name: 2019/Ago, dtype: float64

E se eu quiser localizar um elemento pelo índice?

☐ `dados_usaveis.loc['12 Acre']`

```
dados_usaveis.loc['12 Acre']
```

2007/Ago	10470.07
2007/Set	14001.71
2007/Out	86200.85
2007/Nov	301323.68
2007/Dez	769612.36
...	
2023/Jan	3915317.49
2023/Fev	3381011.42
2023/Mar	3394939.31
2023/Abr	1401760.07
Total	550167804.59

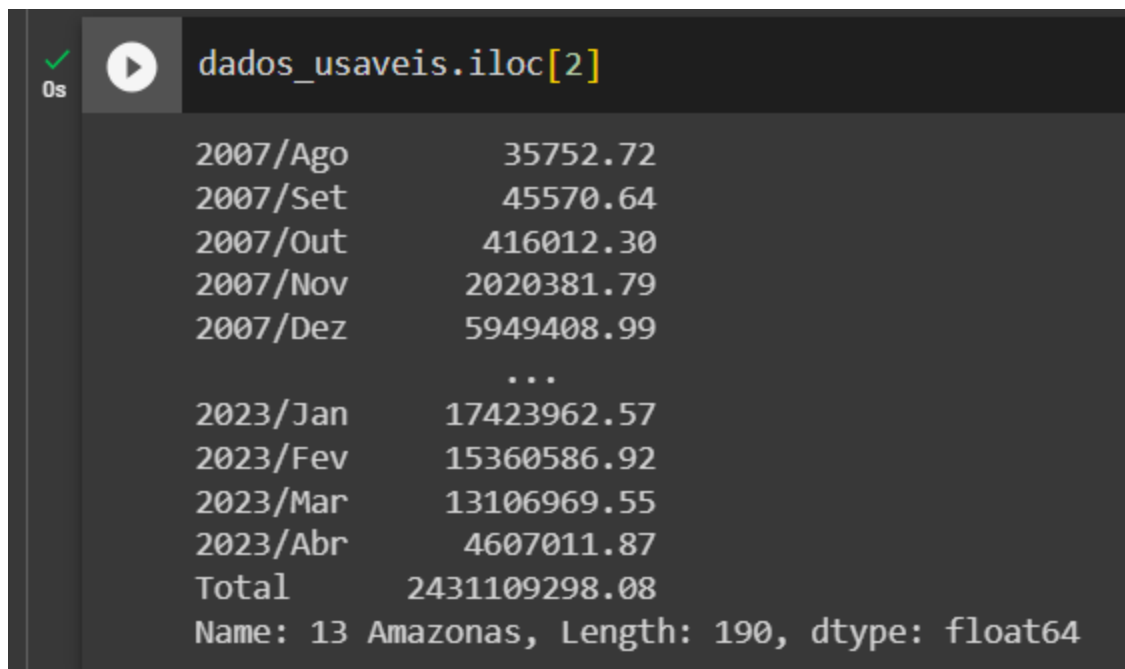
Name: 12 Acre, Length: 190, dtype: float64

Agora sim temos uma série que começa em Agosto de 2007 até Abril de 2023 e temos também o total. 😊

Então, `.loc` é a busca pelo índice puro, pela coluna. Há eixos diferentes de buscas.

Considerando que Acre seja a linha 2, será que não dá pra buscar pela linha 2? Sim!

☐ dados\_usaveis.iloc[2]

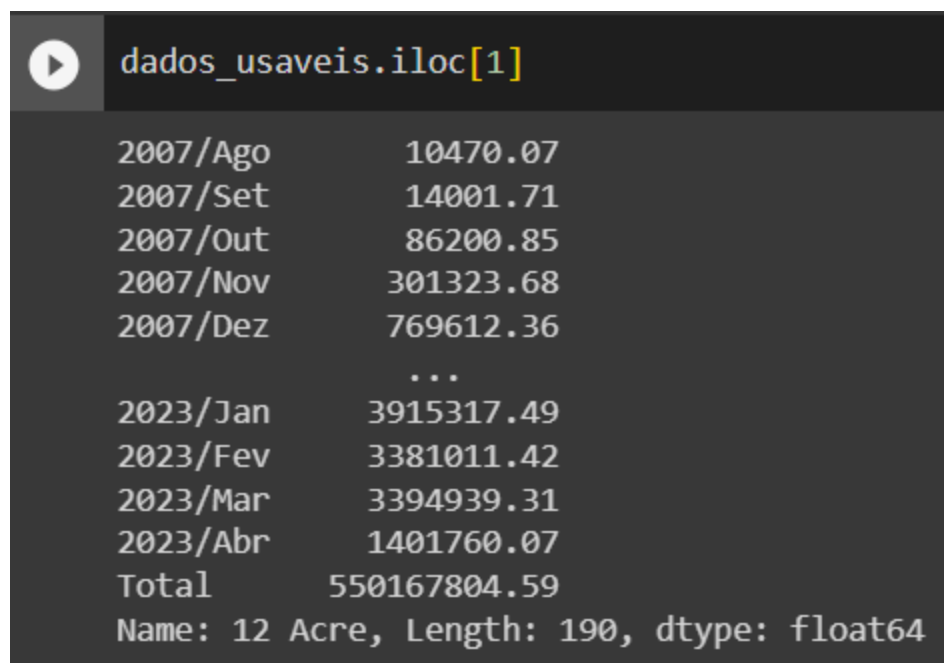


A screenshot of a Jupyter Notebook cell showing the execution of `dados_usaveis.iloc[2]`. The cell has a green checkmark and a play button icon. The output is a Series with a DatetimeIndex. The values are displayed in two columns: the date and the corresponding float64 value. The index ranges from 2007 to 2023, with a 'Total' row at the end. The dtype is float64 and the length is 190.

2007/Ago	35752.72
2007/Set	45570.64
2007/Out	416012.30
2007/Nov	2020381.79
2007/Dez	5949408.99
...	
2023/Jan	17423962.57
2023/Fev	15360586.92
2023/Mar	13106969.55
2023/Abr	4607011.87
Total	2431109298.08

Name: 13 Amazonas, Length: 190, dtype: float64

Mas a contagem é a partir do 0, então colocamos [1]:



A screenshot of a Jupyter Notebook cell showing the execution of `dados_usaveis.iloc[1]`. The cell has a play button icon. The output is a Series with a DatetimeIndex. The values are displayed in two columns: the date and the corresponding float64 value. The index ranges from 2007 to 2023, with a 'Total' row at the end. The dtype is float64 and the length is 190.

2007/Ago	10470.07
2007/Set	14001.71
2007/Out	86200.85
2007/Nov	301323.68
2007/Dez	769612.36
...	
2023/Jan	3915317.49
2023/Fev	3381011.42
2023/Mar	3394939.31
2023/Abr	1401760.07
Total	550167804.59

Name: 12 Acre, Length: 190, dtype: float64

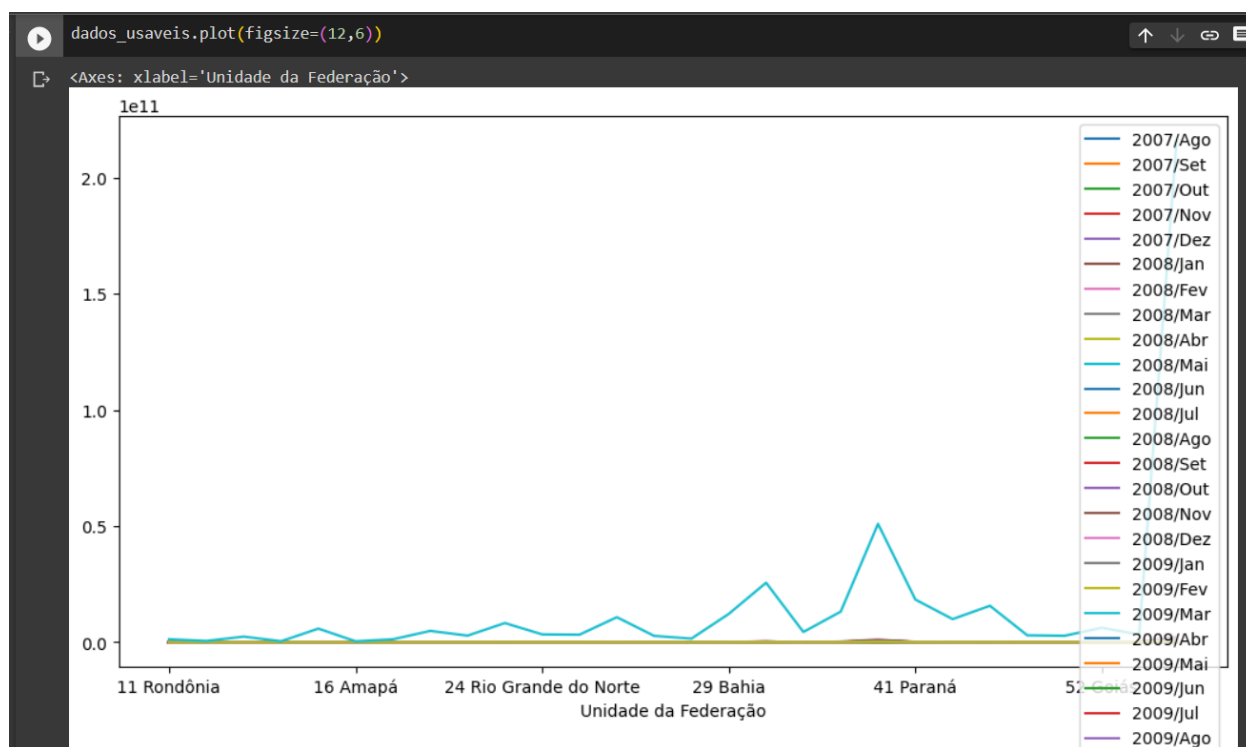
Agora sim temos o Acre! ;)

Portanto, cuidado, o `.iloc` é um localizador através do índice! através de um contador 0 1 2 3 (...). Sempre começando do [0].

Agora vamos usar os dados:

```
dados_usaveis.plot(figsize=(12,6))
```

Para cada uma das colunas o plot desenhou uma linha diferente:

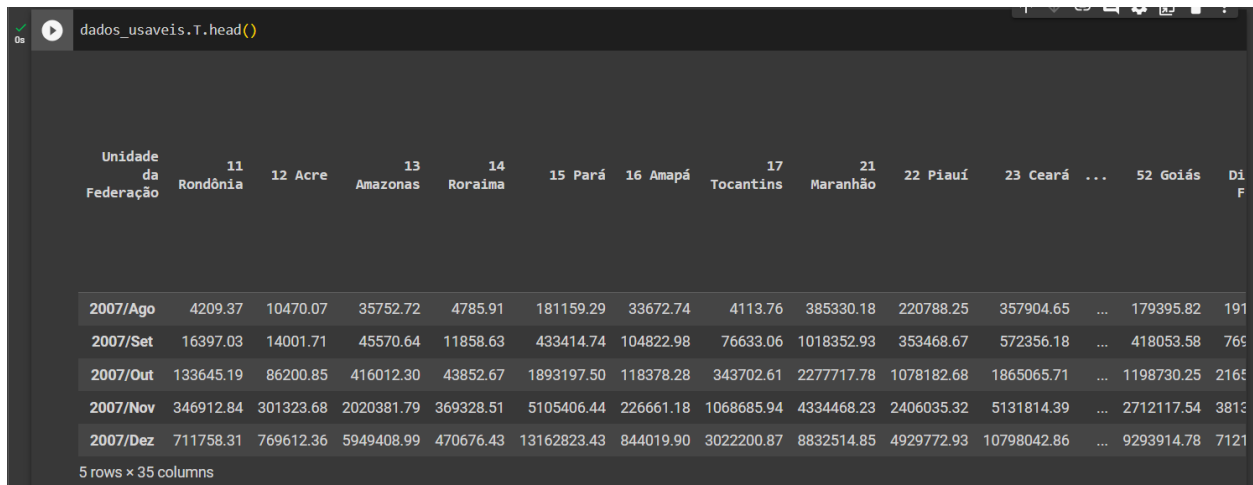


Eu queria que o eixo x fosse o tempo, não os estados. Além do gráfico em si não ser bom para uma série...

Nós podemos transpor uma matriz (T)!

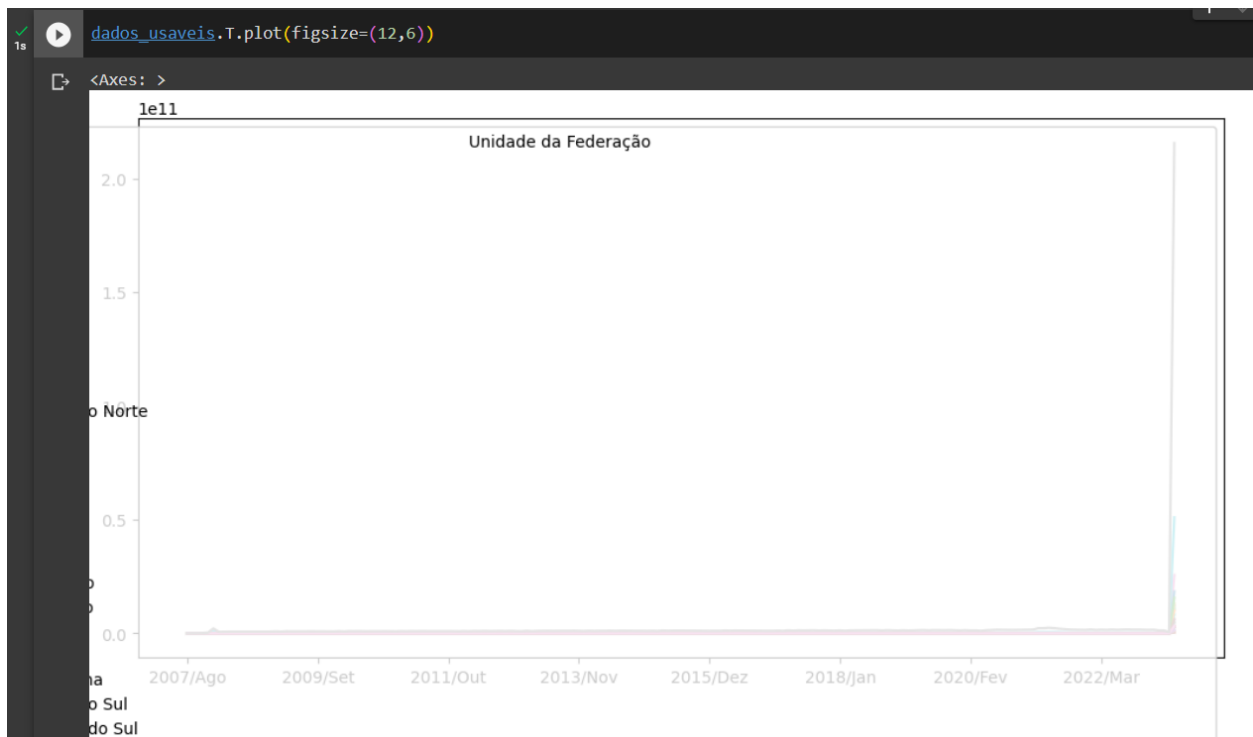
☐ `dados_usaveis.T.head()`

O que era linha virou coluna e o que era coluna virou linha!



	Unidade da Federação	11 Rondônia	12 Acre	13 Amazonas	14 Roraima	15 Pará	16 Amapá	17 Tocantins	21 Maranhão	22 Piauí	23 Ceará	...	52 Goiás	D.F.
2007/Ago		4209.37	10470.07	35752.72	4785.91	181159.29	33672.74	4113.76	385330.18	220788.25	357904.65	...	179395.82	191
2007/Set		16397.03	14001.71	45570.64	11858.63	433414.74	104822.98	76633.06	1018352.93	353468.67	572356.18	...	418053.58	769
2007/Out		133645.19	86200.85	416012.30	43852.67	1893197.50	118378.28	343702.61	2277717.78	1078182.68	1865065.71	...	1198730.25	2165
2007/Nov		346912.84	301323.68	2020381.79	369328.51	5105406.44	226661.18	1068685.94	4334468.23	2406035.32	5131814.39	...	2712117.54	3813
2007/Dez		711758.31	769612.36	5949408.99	470676.43	13162823.43	844019.90	3022200.87	8832514.85	4929772.93	10798042.86	...	9293914.78	7121

☐ `dados_usaveis.T.plot(figsize=(12,6))`

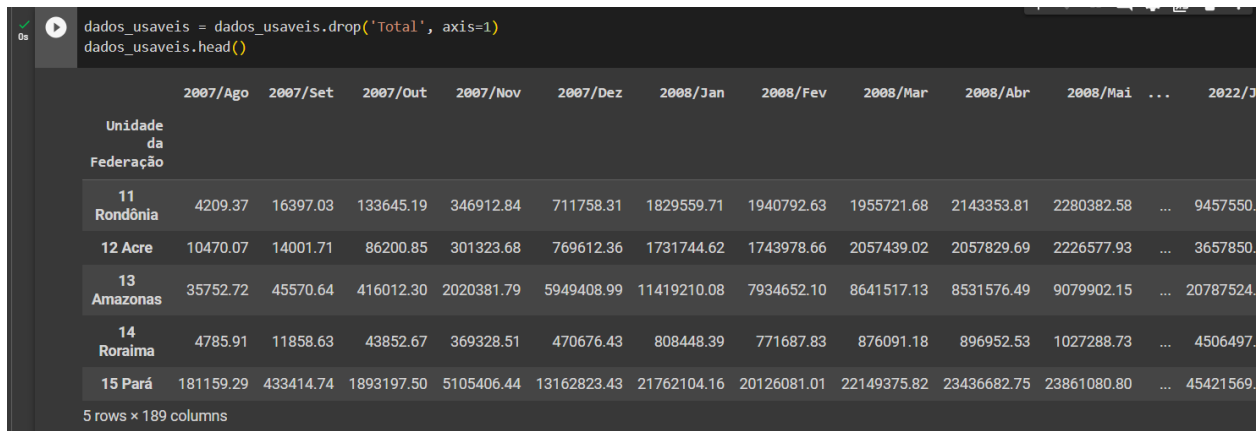


Melhor! Mas tem algo de errado acontecendo aqui... Quando chega no finalzinho, tem um imenso salto no gráfico, isso é por causa do “total”. 😊

☐ `dados_usaveis = dados_usaveis.drop("Total", axis=1)`

☐ `dados_usaveis.head()`

1 é para procurar coluna!

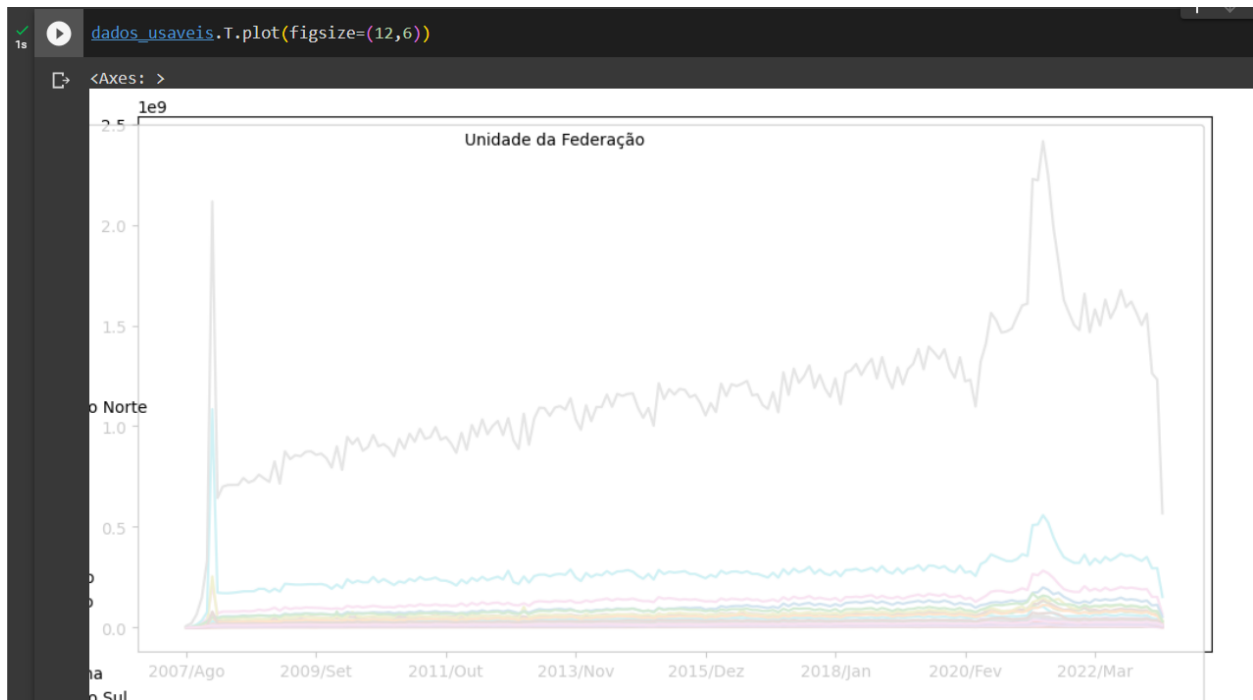


```
dados_usaveis = dados_usaveis.drop('Total', axis=1)
dados_usaveis.head()
```

	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr	2008/Mai	...	2022/J
Unidade da Federação												
11 Rondônia	4209.37	16397.03	133645.19	346912.84	711758.31	1829559.71	1940792.63	1955721.68	2143353.81	2280382.58	...	9457550.
12 Acre	10470.07	14001.71	86200.85	301323.68	769612.36	1731744.62	1743978.66	2057439.02	2057829.69	2226577.93	...	3657850.
13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13	8531576.49	9079902.15	...	20787524.
14 Roraima	4785.91	11858.63	43852.67	369328.51	470676.43	808448.39	771687.83	876091.18	896952.53	1027288.73	...	4506497.
15 Pará	181159.29	433414.74	1893197.50	5105406.44	13162823.43	21762104.16	20126081.01	22149375.82	23436682.75	23861080.80	...	45421569.

5 rows x 189 columns

☐ `dados_usaveis.T.plot(figsize=(12,6))`



Agora temos um gráfico bem mais razoável e temos uma legenda com todos os estados.

**Desafio 01: Reposicionar a legenda. Dentro? Fora? Onde?**

**Desafio 02: Retocar a nossa visualização**

**Desafio 03: Colocar títulos nos dois eixos**

**1 | MANIPULAÇÃO DE DADOS 02**

Entretanto, esse gráfico tem informação demais. Mesmo que a gente posicione, tem muitas linhas. Tem 3 azuis muito parecidos, inclusive.

Posso selecionar 5 colunas:

☐ `dados_usaveis.T.columns[:5]`

☐ `dados_usaveis.index[:5]`



☐ `dados_usaveis[:5]` para buscar as 5 primeiras linhas

The screenshot shows a Jupyter Notebook interface with a code cell containing the following code:

```
dados_usaveis[:5]
```

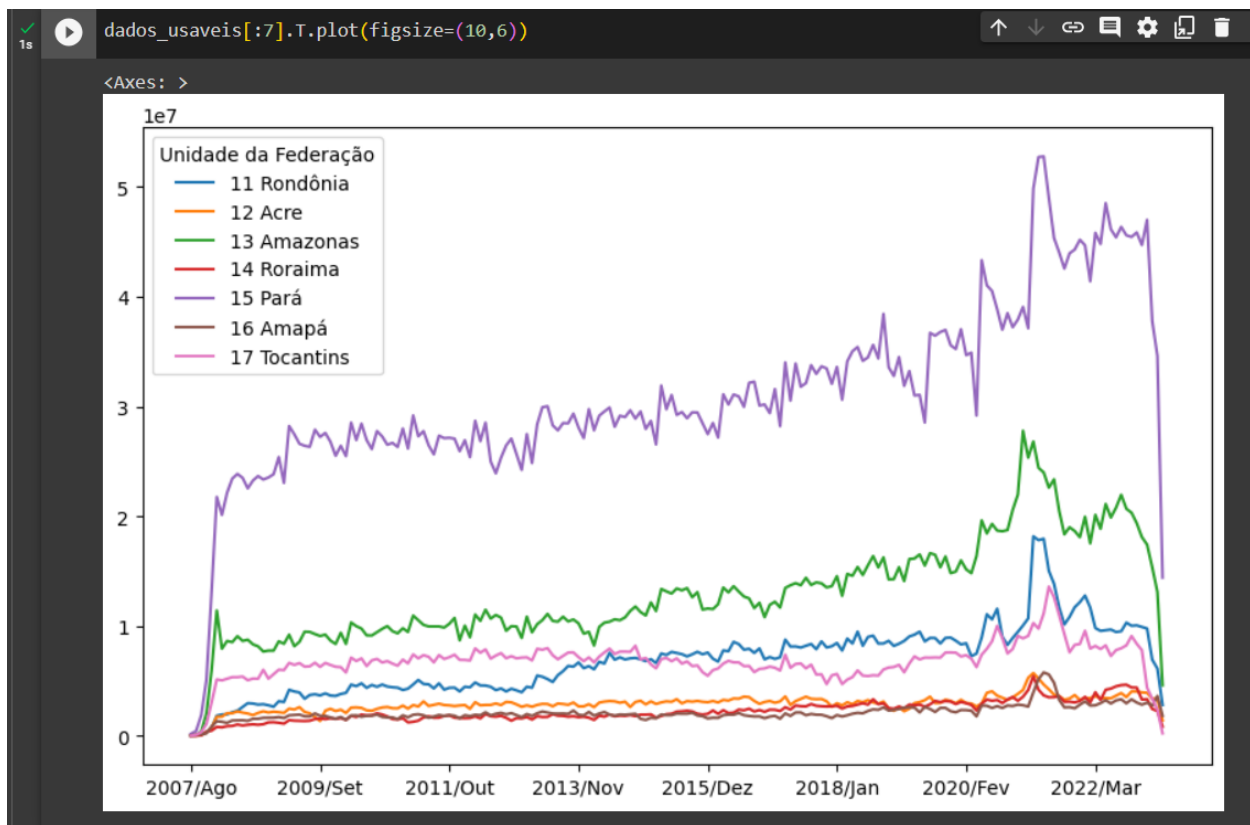
The output of the code cell is a table with 5 rows and 10 columns. The columns are labeled with dates from 2007/Ago to 2008/Mar. The rows are labeled with the names of the states: 11 Rondônia, 12 Acre, 13 Amazonas, 14 Roraima, and 15 Pará. The table shows numerical values for each state across the different dates.

	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr
Unidade da Federação									
11 Rondônia	4209.37	16397.03	133645.19	346912.84	711758.31	1829559.71	1940792.63	1955721.68	21433
12 Acre	10470.07	14001.71	86200.85	301323.68	769612.36	1731744.62	1743978.66	2057439.02	20578
13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13	85315
14 Roraima	4785.91	11858.63	43852.67	369328.51	470676.43	808448.39	771687.83	876091.18	8969
15 Pará	181159.29	433414.74	1893197.50	5105406.44	13162823.43	21762104.16	20126081.01	22149375.82	234366

5 rows x 189 columns

E se eu quiser plotar os 7 primeiros estados?

☐ `dados_usaveis[:7].T.plot(figsize=(10,6))`



E se eu quiser 7 estados aleatórios? Posso escolher uma amostra “aleatória” dos nossos dados.

☐ `dados_usaveis.sample()`

amostra é `sample()`, que chama uma amostra (pode ser qualquer estado).

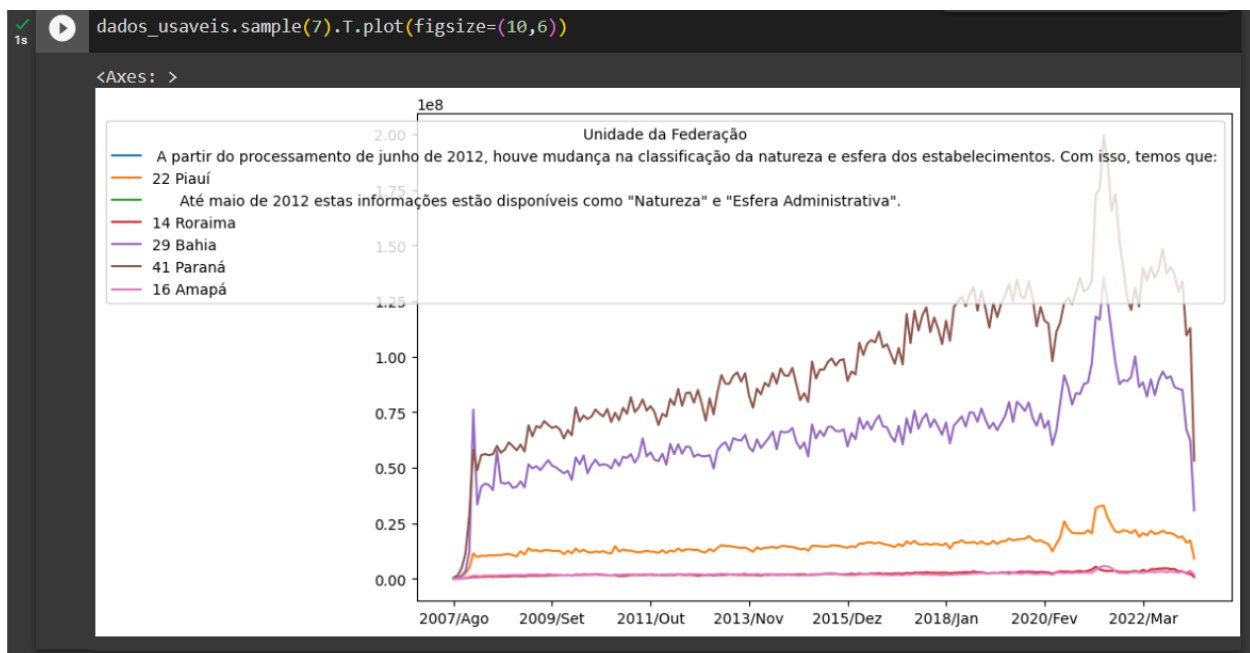
☐ `dados_usaveis.sample(n=7)` ou `dados_usaveis.sample(7)`



`dados_usaveis.sample(n=7)`

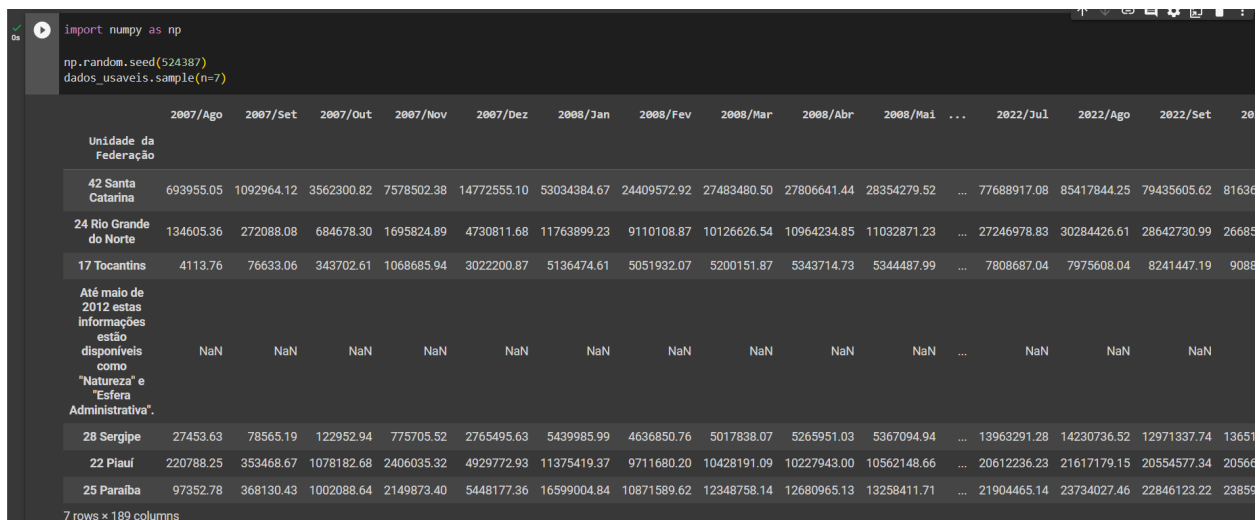
	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar
Unidade da Federação								
13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13
A partir do processamento de junho de 2012, houve mudança na classificação da natureza e esfera dos estabelecimentos. Com isso, temos que:								
16 Amapá	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
29 Bahia	33672.74	104822.98	118378.28	226661.18	844019.90	1329876.61	1275063.72	1177745.93
29 Bahia	135146.36	448349.75	1382038.11	3348779.62	11948984.56	76061864.60	33410124.06	41370676.80
51 Mato Grosso	63562.50	232444.62	927219.59	1740439.60	5394225.43	8213438.64	7929894.93	9416367.68
Até maio de 2012 estas informações estão disponíveis como "Natureza" e "Esfera								
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

`dados_usaveis.sample(7).T.plot(figsize=(10,6))`



A aleatoriedade precisa ser fixa. Vai ser aleatório, mas o mesmo aleatório que eu tenho é o aleatório que outra pessoa pode ter também. Aleatorizar uma vez e a partir daí rodar outras vezes igual. Como funcionam os **algoritmos de aleatoriedade**?

- ☐ `import numpy as np`
- ☐ `np.random.seed(524387)` # colocar ali um número aleatório
- ☐ `dados_usaveis.sample(n=7)`



```
import numpy as np
np.random.seed(524387)
dados_usaveis.sample(n=7)
```

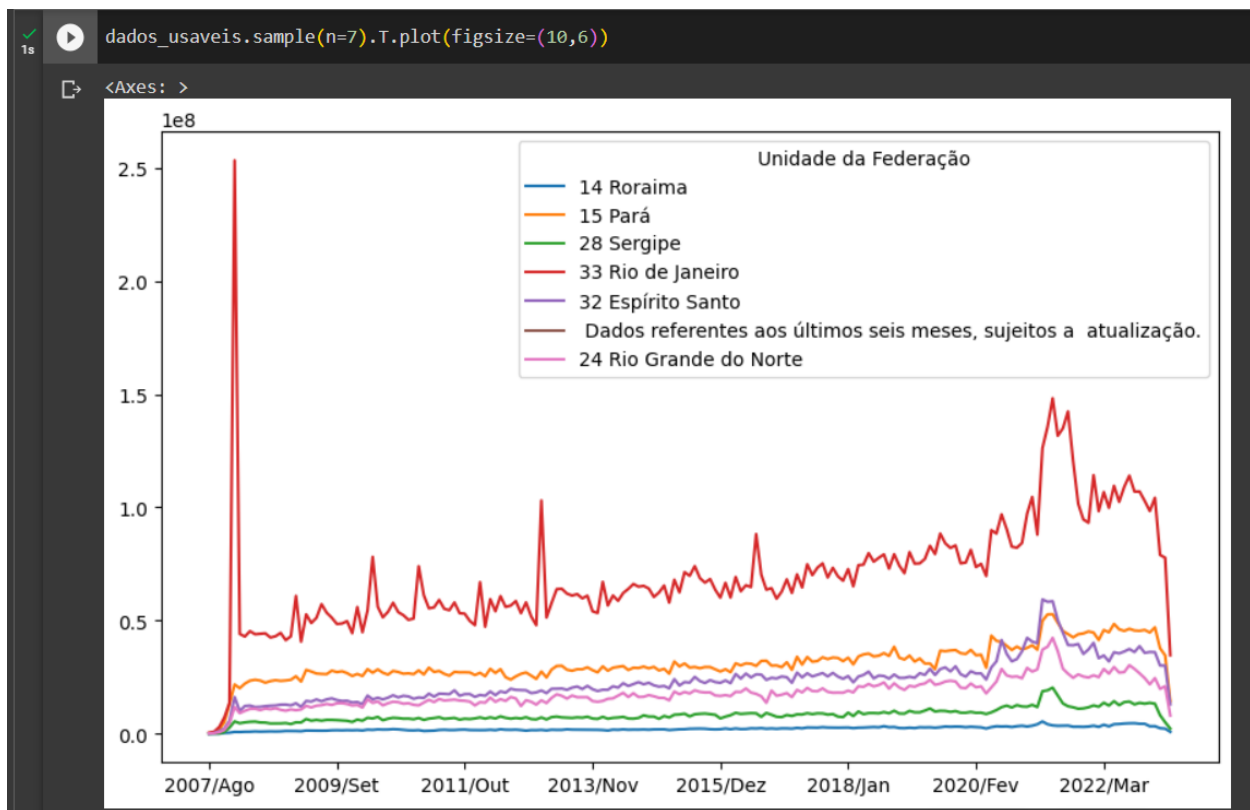
	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr	2008/Mai	...	2022/Jul	2022/Ago	2022/Set	2022/Oct
Unidade da Federação															
42 Santa Catarina	693955.05	1092964.12	3562300.82	7578502.38	14772555.10	53034384.67	24409572.92	27483480.50	27806641.44	28354279.52	...	77688917.08	85417844.25	79435605.62	81636...
24 Rio Grande do Norte	134605.36	272088.08	684678.30	1695824.89	4730811.68	11763899.23	9110108.87	10126626.54	10964234.85	11032871.23	...	27246978.83	30284426.61	28642730.99	26685...
17 Tocantins	4113.76	76633.06	343702.61	1068685.94	3022200.87	5136474.61	5051932.07	5200151.87	5343714.73	5344487.99	...	7808687.04	7975608.04	8241447.19	9088...
Até maio de 2012 estas informações estão disponíveis como "Natureza" e "Esfera Administrativa".	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
28 Sergipe	27453.63	78565.19	122952.94	775705.52	2765495.63	5439985.99	4636850.76	5017838.07	5265951.03	5367094.94	...	13963291.28	14230736.52	12971337.74	13651...
22 Piauí	220788.25	353468.67	1078182.68	2406035.32	4929772.93	11375419.37	9711680.20	10428191.09	10227943.00	10562148.66	...	20612236.23	21617179.15	20554577.34	20566...
25 Paraíba	97352.78	368130.43	1002088.64	2149873.40	5448177.36	16599004.84	10871589.62	12348758.14	12680965.13	13258411.71	...	21904465.14	23734027.46	22846123.22	23859...

7 rows x 189 columns

Se rodar de novo esse trecho de código, a gente vai ter de novo a mesma aleatoriedade!

Agora precisamos plotar...

- ☐ `dados_usaveis.sample(n=7).T.plot(figsize=(10,6))`



☐ dados\_usaveis["Total"] = dados\_usaveis.sum(axis = 1)

☐ dados\_usaveis.head()

dados\_usaveis["Total"] = dados\_usaveis.sum(axis = 1)  
dados\_usaveis.head()

	2007/Ago	2007/Set	2007/Out	2007/Nov	2007/Dez	2008/Jan	2008/Fev	2008/Mar	2008/Abr	2008/Mai	...	2022/Ago	2022/Set	2022/Out	2022/Nov
Unidade da Federação															
11 Rondônia	4209.37	16397.03	133645.19	346912.84	711758.31	1829559.71	1940792.63	1955721.68	2143353.81	2280382.58	...	9525352.64	10309106.47	10026702.19	10061542.23
12 Acre	10470.07	14001.71	86200.85	301323.68	769612.36	1731744.62	1743978.66	2057439.02	2057829.69	2226577.93	...	3894498.82	3529504.71	4040340.45	4029748.15
13 Amazonas	35752.72	45570.64	416012.30	2020381.79	5949408.99	11419210.08	7934652.10	8641517.13	8531576.49	9079902.15	...	21958050.33	20685834.98	20322950.82	19374330.26
14 Roraima	4785.91	11858.63	43852.67	369328.51	470676.43	808448.39	771687.83	876091.18	896952.53	1027288.73	...	4618378.13	4651406.91	4406762.46	4391756.17
15 Pará	181159.29	433414.74	1893197.50	5105406.44	13162823.43	21762104.16	20126081.01	22149375.82	23436682.75	23861080.80	...	46369977.08	45602504.20	45458406.27	45863296.96

5 rows x 190 columns

**Desafio 01: Agora que a gente tem o total, a gente pode ordenar esse total. Ou seja, ordenar o nosso Dataframe**

para que, na primeira linha, tenha a linha com maior gasto, e na última com menor gasto (ordenação).

**Desafio 02:** Adicionar uma coluna com a região de cada estado.

**Desafio 03:** Adicionar seu estado nessa lista de 7 estados (dados\_dos\_7\_estados).

```
dados_dos_7_estados = dados_dos_7_estados.sample(n=7).T.plot(figsize=(10,6))
```