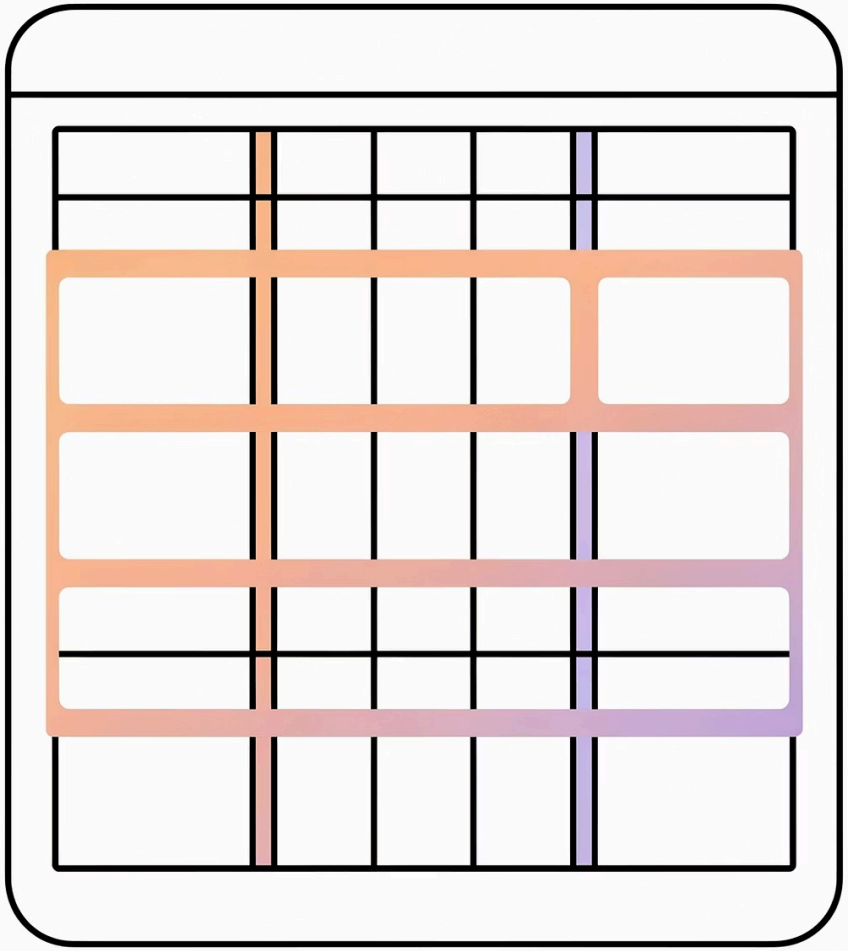


CEFET/RJ

DataFrame em Python (pandas)

Um **DataFrame** é uma estrutura de dados tabular bidimensional, composta por linhas e colunas. Cada coluna pode armazenar um tipo diferente de dado (números, textos, datas). Em Python, utilizamos a biblioteca **pandas** para criar e manipular DataFrames de forma eficiente e intuitiva.

```
import pandas as pd
import numpy as np
```

Eduardo Ogasawara
eduardo.ogasawara@cefet-rj.br
https://eic.cefet-rj.br/~eogasawara

CONCEITOS BÁSICOS

Criando vetores básicos com NumPy

Para construir uma tabela, começamos criando vetores de dados. Em Python, esses vetores são representados por **arrays NumPy**. Cada vetor corresponderá a uma futura coluna da tabela.

Arrays NumPy são estruturas de dados eficientes e fundamentais para computação numérica em Python.

```
import numpy as np

weight = np.array([60, 72, 57, 90, 95, 72])
height = np.array([1.75, 1.80, 1.65, 1.90, 1.74, 1.91])
subject = np.array(["A", "B", "C", "D", "E", "F"])

print(weight)
print(height)
print(subject)
```

Saída:

```
[60 72 57 90 95 72]

[1.75 1.8 1.65 1.9 1.74 1.91]

['A' 'B' 'C' 'D' 'E' 'F']
```

Criando um DataFrame e visualizando

Os vetores são combinados em uma estrutura tabular usando `pd.DataFrame`. Cada vetor se torna uma coluna com um nome específico. O método `head()` exibe as primeiras linhas da tabela, útil para visualização rápida dos dados.

```
d = pd.DataFrame({
    "weight": weight,
    "height": height,
    "subject": subject
})

print(d.head())
```

Saída:

0	60	1.75	A
1	72	1.80	B
2	57	1.65	C
3	90	1.90	D
4	95	1.74	E

MANIPULAÇÃO

Adicionando uma coluna calculada

01

Identificar colunas existentes

Usamos as colunas `weight` e `height` já presentes no DataFrame

02

Criar operação matemática

Aplicamos a fórmula do IMC: peso dividido pela altura ao quadrado

03

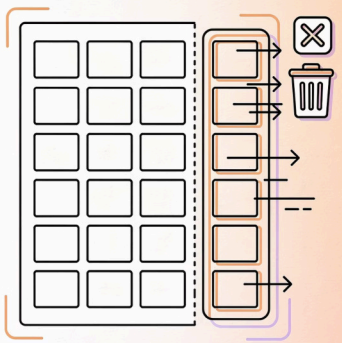
Atribuir nova coluna

O resultado é automaticamente aplicado a todas as linhas

Podemos criar novas colunas a partir de cálculos entre colunas existentes. A operação é aplicada automaticamente a todas as linhas — isso é chamado de **operação vetorizada**.

```
d["bmi"] = d["weight"] / (d["height"] ** 2)
print(d.head())
```

> **Saída:** Agora o DataFrame contém a coluna `bmi` com o Índice de Massa Corporal calculado para cada registro.

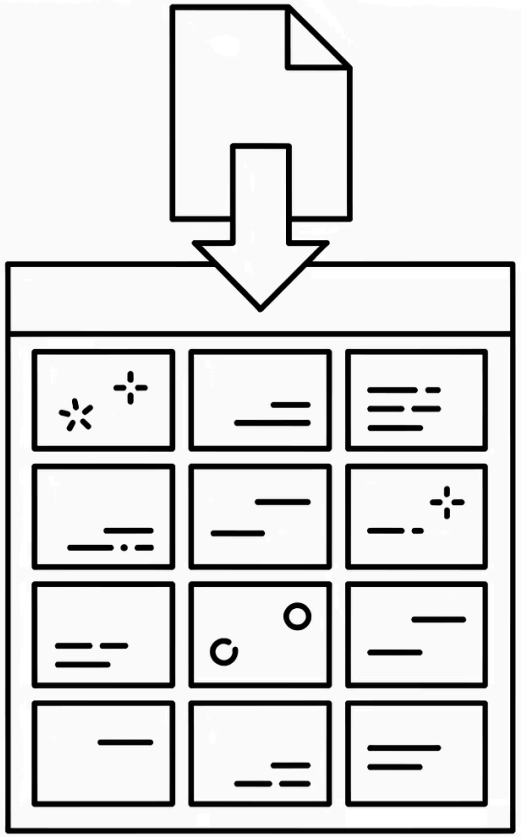


Removendo uma coluna do DataFrame

Uma coluna pode ser removida quando não é mais necessária. Isso simplifica a estrutura da tabela e economiza memória. Em pandas usamos o método `drop` com o parâmetro `columns`.

```
d = d.drop(columns=["subject"])
print(d.head())
```

Resultado: A coluna "subject" foi removida. O DataFrame agora contém apenas weight, height e bmi.



IMPORTAÇÃO

Importando dados de um arquivo CSV

Arquivos CSV (Comma-Separated Values) armazenam tabelas em formato texto simples. Em pandas, usamos `read_csv` para carregar dados de arquivos locais ou URLs remotas. O resultado é um DataFrame pronto para análise e manipulação.

```
import pandas as pd

url = "http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"
wine = pd.read_csv(url, header=None)
print(wine.head())
```

Flexibilidade

Carregue dados de arquivos locais ou URLs

Automático

Pandas detecta automaticamente a estrutura dos dados

Configurável

Controle cabeçalhos, separadores e tipos de dados

Persistência de dados em Python (pickle)

Em Python, usamos **pickle** para salvar e restaurar DataFrames completos. Ele preserva tipos de dados, índices e toda a estrutura do objeto, sendo o formato nativo para persistir objetos pandas.

Esse método é ideal para trabalhos intermediários e compartilhamento de dados processados.

```
wine.to_pickle("wine.pkl")
del wine
wine = pd.read_pickle("wine.pkl")
print(wine.head(3))
```

1

Salvar

`to_pickle()`

2

Armazenar

Arquivo .pkl

3

Restaurar

read_pickle()

EXPORTAÇÃO

Exportando dados para CSV

CSV é o formato universal para trocar dados entre diferentes programas e plataformas. Em pandas usamos `to_csv` para salvar o DataFrame. Podemos escolher se o índice será incluído ou não no arquivo final.

```
wine.to_csv("wine.csv", index=False)
```

Resultado: O arquivo `wine.csv` é criado no diretório do projeto, pronto para ser aberto no Excel, Google Sheets ou qualquer ferramenta de análise de dados.

Compatibilidade universal

Abra em qualquer software de análise de dados

Formato legível

Texto simples que pode ser visualizado e editado

Compartilhamento fácil

Ideal para colaboração entre equipes

FILTRAGEM

Filtrando linhas com máscara booleana

Podemos filtrar linhas criando uma condição lógica. A condição gera um vetor de `True` e `False`. Apenas as linhas marcadas como `True` são mantidas no resultado final.

```
mask = d["height"] > 1.7
print(mask)
filtered = d[mask]
print(filtered)
```

Máscara booleana:

```
0    True
1    True
2   False
3    True
4    True
5    True
Name: height, dtype: bool
```

DataFrame filtrado:

Apenas as linhas onde `height > 1.7` são mantidas. As linhas 0, 1, 3, 4 e 5 aparecem no resultado.

DESEMPENHO

Operações vetorizadas e desempenho

Velocidade

Cálculos sobre colunas inteiras são extremamente rápidos

Otimização

NumPy e pandas usam código compilado em C

Processamento em lote

Todos os valores processados simultaneamente

Cálculos feitos diretamente sobre colunas inteiras são muito rápidos. Isso ocorre porque pandas e NumPy usam **operações vetorizadas** — todos os valores são processados de uma só vez, sem loops Python.

```
import numpy as np
import pandas as pd
from time import perf_counter

rheight = np.random.normal(1.8, 0.2, 100_000)
rweight = np.random.normal(72, 15, 100_000)

t0 = perf_counter()
hw = pd.DataFrame({"height": rheight, "weight": rweight})
hw["bmi"] = hw["weight"] / (hw["height"] ** 2)
t1 = perf_counter()

print("Tempo vetorizado:", t1 - t0)
```

> **Saída típica:** Tempo vetorizado: 0.0021 segundos para processar 100.000 linhas!

Tipos de dados das colunas (dtype)

Em pandas, cada coluna tem um **tipo interno** chamado `dtype`. O tipo influencia desempenho, uso de memória e o comportamento das operações. Conferir dtypes ajuda a evitar erros e resultados inesperados.

```
print(hw.dtypes)
```

Saída (exemplo):

```
height    float64
weight    float64
bmi       float64
dtype: object
```

float64

Números decimais de alta precisão

int64

Números inteiros

object

Texto ou dados mistos

bool

Valores True/False

⚠ ANTIPADRÃO

Antipadrão: calcular com loop no DataFrame

Atribuir valores linha a linha em um DataFrame é **muito lento**. Esse padrão faz acessos repetidos à estrutura complexa do pandas. Sempre que possível, prefira operações vetorizadas.

```
from time import perf_counter
import numpy as np
import pandas as pd

rheight = np.random.normal(1.8, 0.2, 100_000)
rweight = np.random.normal(72, 15, 100_000)

t0 = perf_counter()
hw = pd.DataFrame({"height": rheight, "weight": rweight})
hw["bmi"] = np.nan
for i in range(len(hw)):
    hw.loc[i, "bmi"] = hw.loc[i, "weight"] / (hw.loc[i, "height"] ** 2)
t1 = perf_counter()

print("Tempo com loop:", t1 - t0)
```

❌ Evite loops

Iteração linha a linha é extremamente lenta

✅ Use vetorização

Operações em colunas inteiras são muito mais rápidas

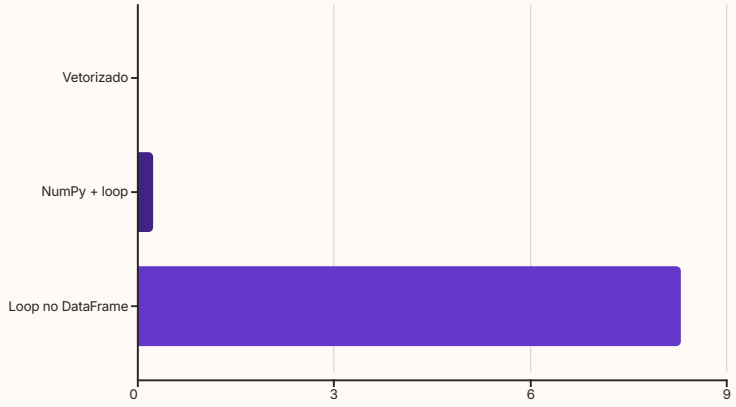
> **Saída típica:** Tempo com loop: ~~8.3 segundos~~ — 3900x mais lento que a versão vetorizada!

OTIMIZAÇÃO INTERMEDIÁRIA

Converter para NumPy e voltar

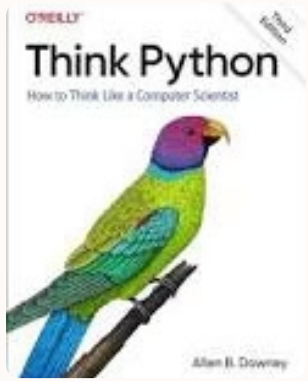
DataFrames são construídos sobre arrays NumPy. Converter para NumPy dá acesso mais direto e rápido aos valores. Depois do processamento, colocamos o resultado de volta no DataFrame. Essa abordagem é intermediária: mais rápida que loops no DataFrame, mas mais lenta que vetorização pura.

```
from time import perf_counter
import numpy as np
import pandas as pd

t0 = perf_counter()
hw = pd.DataFrame({"height": rheight, "weight": rweight})
hwm = hw.to_numpy()
bmi = np.empty(len(hwm))
for i in range(len(hwm)):
    bmi[i] = hwm[i, 1] / (hwm[i, 0] ** 2)
hw["bmi"] = bmi
t1 = perf_counter()

print("Tempo NumPy + loop:", t1 - t0)
print(hw.head())
```



Conclusão: A conversão para NumPy melhora significativamente o desempenho comparada ao loop direto no DataFrame, mas a vetorização pura continua sendo a melhor escolha sempre que possível.

Referências



Think Python

Downey, A. *Think Python: How to Think Like a Computer Scientist.* O'Reilly Media.

An essential introduction to programming fundamentals and computational thinking using Python.

Python Data Science Handbook

VanderPlas, J. *Python Data Science Handbook.* O'Reilly Media.

A comprehensive guide to essential tools for working with data in Python, including NumPy, Pandas, and visualization libraries.

Data Science from Scratch

Grus, J. *Data Science from Scratch.* O'Reilly Media.

Learn data science fundamentals by building algorithms and tools from the ground up using Python.