Manipulação de DataFrames em Python

Nesta apresentação vamos criar e manipular um baralho usando DataFrames do pandas. Um DataFrame é uma tabela com colunas nomeadas e linhas indexadas, onde cada linha representa uma carta do baralho.

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

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

CONCEITO

Construindo o Baralho (Produto Cartesiano)

Criamos todas as combinações entre faces e naipes usando o produto cartesiano. Isso gera exatamente 52 cartas — as 13 faces multiplicadas pelos 4 naipes.

O resultado é armazenado diretamente em um DataFrame com colunas nomeadas.

```
import itertools

faces = ["ás","dois","três","quatro","cinco","seis",
        "sete","oito","nove","dez","valete","dama","rei"]
naipes = ["ouros","copas","paus","espadas"]

baralho = pd.DataFrame(
    itertools.product(faces, naipes),
    columns=["face", "naipe"]
)

baralho.head()
```

Saída esperada:

```
 face naipe
0 ás ouros
1 ás copas
2 ás paus
3 ás espadas
4 dois ouros
```

OPERAÇÃO

Criando e Atribuindo uma Nova Coluna

Podemos criar novas colunas atribuindo valores diretamente ao DataFrame. A função `np.tile` repete os valores de 1 a 13 quatro vezes, criando o valor numérico para cada carta.

Essa abordagem é simples e eficiente para adicionar informações estruturadas.

```
baralho["valor"] = np.tile(np.arange(1, 14), 4)

baralho.head()
```

Saída esperada:

```
face naipe valor
0 ás ouros 1
1 ás copas 1
2 ás paus 1
3 ás espadas 1
4 dois ouros 2
```

TÉCNICAS

Formas de Acesso a Tabelas

iloc: Posição

Acesso baseado em índices numéricos de linhas e colunas

```
baralho.iloc[0, 1]
```

loc: Nome

Acesso usando rótulos de índice e nomes de colunas

```
baralho.loc[0, "naipe"]
```

Condição

Filtragem usando expressões booleanas

```
baralho[baralho["valor"] > 10]
```

Esses três métodos cobrem os principais padrões de indexação no pandas, permitindo flexibilidade no acesso aos dados.

Saídas esperadas: 'ouros', 'ouros', e DataFrame com valetes, damas e reis

ESTRUTURA

Colunas como Vetores (Séries)

Quando acessamos uma única coluna de um DataFrame, o resultado é uma **Série** — uma estrutura unidimensional que se comporta como um vetor.

Séries mantêm o índice original e permitem operações vetorizadas eficientes.

```
baralho.iloc[0, 0]
baralho["face"].iloc[[0, 1]]
baralho["face"].head()
```

Saída esperada:

```
'ás'

0 ás
1 dois

Name: face, dtype: object
```

Múltiplas Formas de Acessar Colunas

O pandas oferece diferentes sintaxes equivalentes para acessar os mesmos dados. Podemos escolher a forma mais clara e legível conforme o contexto do código.

1

loc com nomes

```
baralho.loc[[10, 13], "face"]
```

2

iloc com posições

```
baralho.iloc[[10, 13], 0]
```

3

Coluna + iloc

```
baralho["face"].iloc[[10, 13]]
```

Resultado: Todas retornam a mesma Série com 'valete' e 'ás'

SELEÇÃO

Acessando Múltiplas Colunas

Quando selecionamos **mais de uma coluna**, o pandas retorna um DataFrame em vez de uma Série. Isso preserva a estrutura tabular e a relação entre as variáveis.

Use listas duplas de colchetes para garantir que o resultado seja sempre um DataFrame.

```
baralho.iloc[10:12, 0:2]
baralho.loc[[10, 13], ["face", "naipe"]]
```

Saída esperada:

```
 face naipe
10 valete ouros
11 valete copas

 face naipe
10 valete ouros
13 ás copas
```

Série vs DataFrame

Uma coluna: Série

Colchetes simples retornam uma Série unidimensional

```
type(baralho["face"])
```

Resultado: `pandas.core.series.Series`

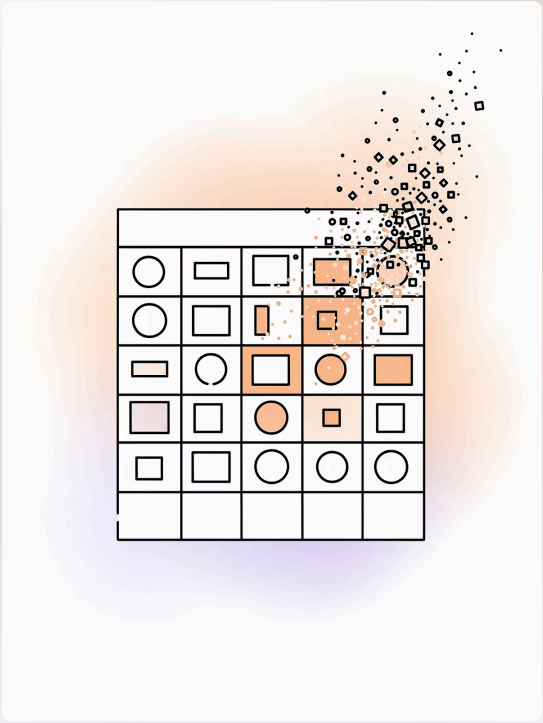
Uma coluna: DataFrame

Colchetes duplos preservam a estrutura de tabela

```
type(baralho[["face"]])
```

Resultado: `pandas.core.frame.DataFrame`

Essa distinção é importante ao encadear operações que esperam um formato específico.



REMOÇÃO

Exclusão de Linhas e Colunas

No pandas, usamos o método `drop()` para remover dados. Isso substitui os índices negativos usados em outras linguagens como R.

Remover linhas

```
baralho.drop(index=range(3))
```

Remove as três primeiras linhas

Remover colunas

```
baralho.drop(columns=["face"])
```

Remove a coluna 'face'

Por padrão, `drop()` retorna uma nova cópia. Use `inplace=True` para modificar o original.

DIMENSÕES

Seleção Total e Índices Vazios

Podemos acessar dimensões completas da tabela: uma linha inteira, uma coluna inteira ou o DataFrame completo.

Essas operações são fundamentais para análise exploratória e transformações de dados.

```
baralho.iloc[0]
baralho["naipe"].head()
```

Saídas:

```
face ás
naipe ouros
valor 1
Name: 0, dtype: object

0 ouros
1 copas
2 paus
3 espadas
4 ouros
Name: naipe, dtype: object
```

FILTROS

Indexação Lógica para Filtrar Colunas

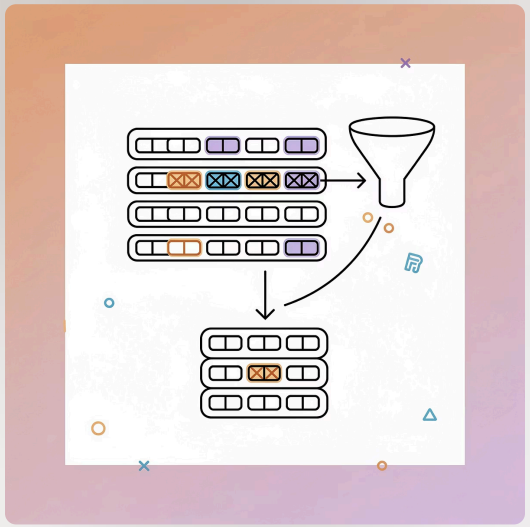
Podemos usar **vetores booleanos** para escolher quais colunas manter. Cada valor `True` indica uma coluna que deve aparecer no resultado.

Essa técnica permite filtros dinâmicos baseados em condições complexas.

```
mask = [True, True, False]
baralho.loc[:, mask]
```

Resultado: DataFrame contendo apenas as colunas 'face' e 'naipe'

```
 face naipe
0 ás ouros
1 ás copas
2 ás paus
3 ás espadas
4 dois ouros
...
```



FILTROS

Indexação Lógica para Filtrar Linhas

Criamos uma **máscara booleana** com base em uma condição. Essa máscara determina quais linhas permanecem no DataFrame resultante.

É a forma mais comum e poderosa de filtrar dados no pandas.

```
filtro = baralho["valor"] < 3
baralho[filtro]
```

Resultado: Apenas cartas com valor menor que 3 (ases e dois)

```
 face naipe valor
0 ás ouros 1
1 ás copas 1
4 dois ouros 2
5 dois copas 2
...
```

ALEATORIZAÇÃO

Embaralhando o Baralho

Criamos uma permutação aleatória dos índices usando NumPy. Depois, usamos essa ordem para reorganizar completamente o DataFrame.

01

Gerar permutação

Cria uma sequência aleatória de índices

02

Aplicar ao DataFrame

Usa iloc para reordenar as linhas

03

Obter resultado

Retorna o baralho embaralhado

```
ordem = np.random.permutation(len(baralho))
cartas = baralho.iloc[ordem]
cartas.head()
```

Saída: Cartas em ordem aleatória (varia a cada execução)

FUNÇÃO

Encapsulando em uma Função

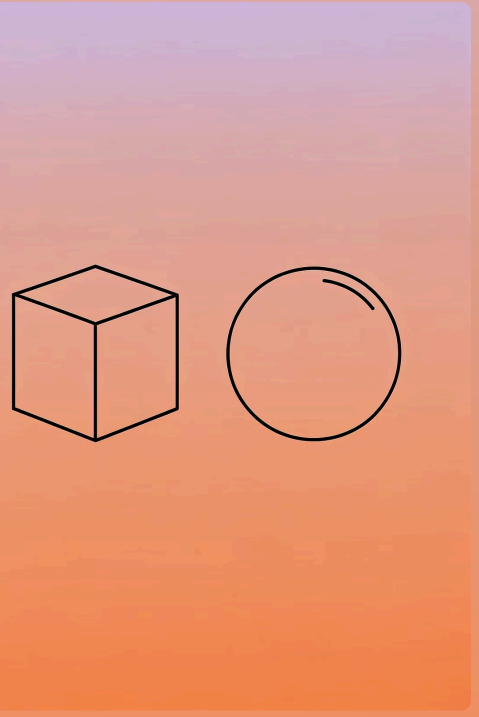
Transformamos o código em uma **função reutilizável** que aceita qualquer DataFrame e retorna uma versão embaralhada.

Isso promove o princípio DRY (Don't Repeat Yourself) e facilita manutenção.

```
def embaralhar(df):
    ordem = np.random.permutation(len(df))
    return df.iloc[ordem]

embaralhar(baralho).head()
```

Vantagens: Código modular, testável e reutilizável em diferentes contextos



TIPOS

Tipos de Dados (dtype)

Arrays NumPy e colunas pandas possuem tipos internos que determinam como os dados são armazenados na memória e qual a precisão numérica disponível.

int64

Números inteiros

```
a = np.array([1, 2, 3])
a.dtype
# dtype('int64')
```

float64

Números com casas decimais

```
b = np.array([1.0, 2.0, 3.0])
b.dtype
# dtype('float64')
```

Conhecer os tipos ajuda a otimizar memória e evitar erros de precisão em cálculos.

CONCEITOS AVANÇADOS

Broadcasting, Visão e Cópia

Broadcasting

NumPy combina vetores de tamanhos diferentes automaticamente, repetindo valores conforme necessário para realizar operações elemento a elemento.

View (Visão)

Algumas indexações criam uma **visão** dos dados originais. Modificar a visão afeta o array original.

Copy (Cópia)

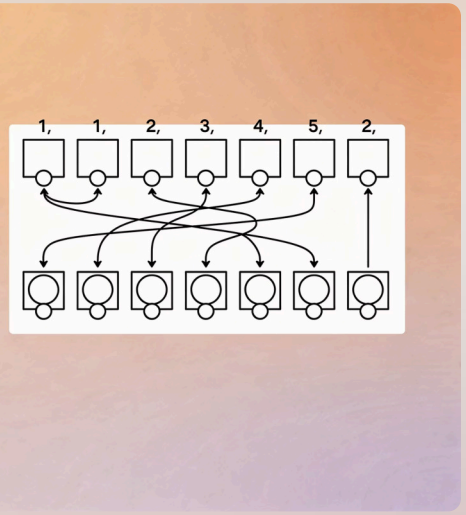
Outras indexações criam uma **cópia real** dos dados. Modificações não afetam o array original.

```
x = np.array([1, 2, 3, 4])
y = x[0:2]        # view
z = x[0:2].copy()  # copy

y[0] = 99
x, y, z
```

Resultado: `(array([99, 2, 3, 4]), array([99, 2]), array([1, 2]))`

Note que modificar `y` alterou `x`, mas `z` permaneceu inalterado.



TÉCNICA

Indexação Avançada e Referências

Podemos selecionar e reordenar dados usando **vetores de índices**. Isso permite amostragem personalizada e reordenação controlada dos elementos.

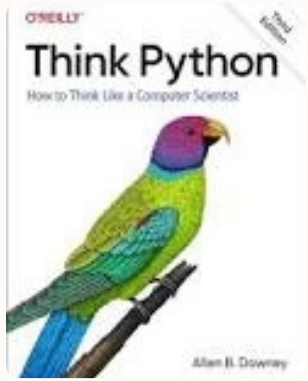
Essa técnica é poderosa para criar subconjuntos específicos ou reorganizar dados conforme necessário.

```
x = np.array([10, 20, 30, 40, 50])
idx = [0, 2, 4]
x[idx]
```

Saída esperada: `array([10, 30, 50])`

Os elementos nas posições 0, 2 e 4 foram extraídos, criando um novo array com apenas esses valores.

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.