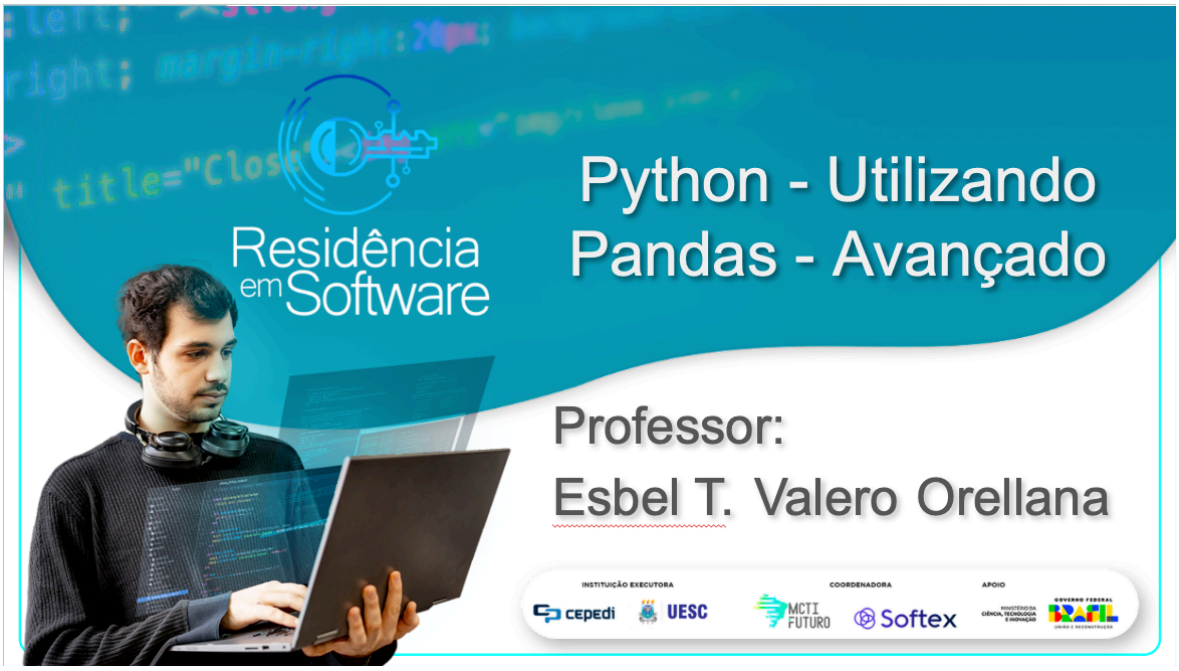
Módulo de Programação Python

Trilha Python - Aula 21: Utilizando Pandas - Avançado



Objetivo: Trabalhar com pacotes e módulos disponíveis em python: Pandas: Recursos **Pandas** de alto desempenho. Persistência de dados com **Pandas**.

Recursos Pandas de alto desempenho: eval() e query()

Como já vimos nas seções anteriores, o poder da **Python** na análise e processamento de dados é construído sobre a capacidade do **NumPy** e do **Pandas** de enviar operações básicas para C por meio de uma sintaxe intuitiva. A modo de exemplos temos as operações vetorizadas/transmitidas em **NumPy** e operações do tipo agrupamento em **Pandas**.

Embora essas abstrações sejam eficientes e eficazes para muitos casos de uso comuns, elas geralmente dependem da criação de objetos intermediários temporários, o que pode causar sobrecarga indevida no tempo computacional e no uso da memória.

A partir da versão 0.13 (lançada em janeiro de 2014), o **Pandas** inclui algumas ferramentas experimentais que permitem acessar diretamente operações C-speed sem alocação dispendiosa de arrays intermediários. Estas são as funções `eval()` e `query()`.

Motivando `query()` e `eval()`: Expressões Compostas

Vimos anteriormente que **NumPy** e **Pandas** suportam operações vetorizadas rápidas como, por exemplo, ao adicionar os elementos de duas matrizes.

In [1]:
```
import numpy as np
rng = np.random.RandomState(42)
x = rng.rand(1000000)
y = rng.rand(1000000)
%timeit x + y
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
448 µs ± 29.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)
```

Conforme discutido em anteriormente, este método é muito mais rápido do que fazer a adição por meio de um loop **Python**.

Mas esta abstração pode se tornar menos eficiente ao calcular expressões compostas. Por exemplo, considere a seguinte expressão.

In [2]:
```
mask = (x > 0.5) & (y < 0.5)
```

Como o **NumPy** avalia cada subexpressão, isso é aproximadamente equivalente a:

In [3]:
```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

Em outras palavras, cada etapa intermediária é alocada explicitamente na memória. Se os arrays `x` e `y` forem muito grandes, isso pode levar a memória significativa e sobrecarga computacional.

Por outro lado, a biblioteca **Numexpr** oferece a capacidade de calcular esse tipo de expressão composta elemento por elemento, sem a necessidade de alocar matrizes intermediárias completas.

A documentação do Numexpr (https://github.com/pydata/numexpr) tem mais detalhes, mas por enquanto é suficiente dizer que a biblioteca aceita uma *string* fornecendo a expressão no estilo NumPy que você deseja calcular.

In [4]:
```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)
```

Out[4]:
```
True
```

A função `allclose` retorna `True` se duas matrizes forem iguais elemento por elementos dentro de uma margem de tolerância.

```
    numpy.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=
False)
```

A vantagem aqui é que o **Numexpr** avalia a expressão de uma forma que não usa arrays temporários de tamanho normal e, portanto, pode ser muito mais eficiente que o **NumPy**, especialmente para arrays grandes.

As ferramentas `eval()` e `query()` do Pandas que discutiremos aqui são conceitualmente semelhantes e dependem do pacote **Numexpr**.

`pandas.eval()` para operações eficientes

A função `eval()` no **Pandas** usa expressões de *string* para calcular operações de forma eficiente usando `DataFrame`s. Por exemplo, considere os seguintes `DataFrame`s:

In [5]:
```
import pandas as pd
nrows, ncols = 100000, 100
rng = np.random.RandomState(42)
df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                      for i in range(4))
```

Para calcular a soma de todos os quatro `DataFrame`s usando a abordagem típica do **Pandas**, podemos simplesmente escrever a soma.

In [6]:
```
%timeit df1 + df2 + df3 + df4
```

```
21.2 ms ± 1.52 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

O mesmo resultado pode ser calculado via `pd.eval` construindo a expressão como uma string.

In [7]:
```
%timeit pd.eval('df1 + df2 + df3 + df4')
```

```
10.7 ms ± 488 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

A versão `eval()` desta expressão é cerca de 50% mais rápida (e usa muito menos memória), dando o mesmo resultado.

In [8]:
```
np.allclose(df1 + df2 + df3 + df4, pd.eval('df1 + df2 + df3 + df4'))
```

Out[8]:
```
True
```

Operações suportadas por `pd.eval()`

A partir do Pandas v0.16, `pd.eval()` suporta uma ampla gama de operações. Para demonstrá-los, usaremos os seguintes inteiros `DataFrame` s

In [10]:
```
df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100,
                           for i in range(5))
```

Operadores aritméticos

`pd.eval()` suporta todos os operadores aritméticos.

In [11]:
```
result1 = -df1 * df2 / (df3 + df4) - df5
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
np.allclose(result1, result2)
```

Out[11]: True

Operadores de comparação

`pd.eval()` suporta todos os operadores de comparação, incluindo expressões encadeadas.

In [12]:
```
result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
result2 = pd.eval('df1 < df2 <= df3 != df4')
np.allclose(result1, result2)
```

Out[12]: True

Operadores bit a bit

`pd.eval()` suporta os operadores bit a bit `&` e `|`

In [13]:
```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
np.allclose(result1, result2)
```

Out[13]: True

Além disso, ele suporta o uso dos literais `and` e `or` em expressões booleanas.

In [14]:
```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
np.allclose(result1, result3)
```

Out[14]: True

Atributos e índices de objetos

`pd.eval()` suporta acesso a atributos de objetos através da sintaxe `obj.attr` e índices através da sintaxe `obj[index]`

In [15]:
```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
```

Out[15]: True

`DataFrame.eval()` para operações em colunas

Assim como o **Pandas** tem uma função `pd.eval()` de nível superior, os `DataFrame` s têm um método `eval()` que funciona de maneira semelhante.

A vantagem do método `eval()` é que as colunas podem ser referenciadas pelo nome.

In [16]:
```
df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
df.head()
```

Out[16]:

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

Usando `pd.eval()` como acima, podemos calcular expressões com as três colunas.

In [17]:
```
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
```

Out[17]: True

O método `DataFrame.eval()` permite uma avaliação muito mais sucinta de expressões com as colunas.

In [18]:
```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
```

Out[18]:
```
True
```

Repare aqui que tratamos nomes de colunas como variáveis dentro da expressão avaliada, e o resultado é o que estávamos esperando.

Atribuição em DataFrame.eval()

Além das opções que acabamos de discutir, `DataFrame.eval()` também permite atribuição a qualquer coluna.

Vamos usar o `DataFrame` de antes, que tem as colunas `'A'`, `'B'` e `'C'`

In [19]:
```
df.head()
```

Out[19]:

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

Podemos usar `df.eval()` para criar uma nova coluna `'D'` e atribuir a ela um valor calculado a partir das outras colunas.

In [20]:
```
df.eval('D = (A + B) / C', inplace=True)
df.head()
```

Out[20]:

	A	B	C	D
0	0.375506	0.406939	0.069938	11.187620
1	0.069087	0.235615	0.154374	1.973796
2	0.677945	0.433839	0.652324	1.704344
3	0.264038	0.808055	0.347197	3.087857
4	0.589161	0.252418	0.557789	1.508776

Da mesma forma, qualquer coluna existente pode ser modificada.

In [21]:
```
df.eval('D = (A - B) / C', inplace=True)
df.head()
```

Out[21]:

	A	B	C	D
0	0.375506	0.406939	0.069938	-0.449425
1	0.069087	0.235615	0.154374	-1.078728
2	0.677945	0.433839	0.652324	0.374209
3	0.264038	0.808055	0.347197	-1.566886
4	0.589161	0.252418	0.557789	0.603708

Variáveis locais em DataFrame.eval()

O método `DataFrame.eval()` suporta uma sintaxe adicional que permite trabalhar com variáveis locais do **Python**.

In [22]:
```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
```

Out[22]:
```
True
```

O caractere `@` aqui marca um nome de variável em vez de um nome de coluna, e permite avaliar eficientemente expressões envolvendo os dois "namespaces": o namespace de colunas e o namespace de objetos **Python**.

Observe que este caractere `@` só é suportado pelo método `DataFrame.eval()`, e não pela função `pandas.eval()`. Isto porque o `pandas.eval()` função só tem acesso ao namespace **Python**.

Método DataFrame.query()

O `DataFrame` possui outro método baseado em avaliação de *strings*, chamado método `query()`.

In [23]:
```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
np.allclose(result1, result2)
```

Out[23]: True

Assim como no exemplo usado em nossa discussão sobre `DataFrame.eval()`, esta é uma expressão envolvendo colunas do `DataFrame`. Entretanto, ele não pode ser expresso usando a sintaxe `DataFrame.eval()`!

Em vez disso, para este tipo de operação de filtragem, você pode usar o método `query()`

In [24]:
```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
```

Out[24]: True

Além de ser um cálculo mais eficiente, comparado à expressão de mascaramento é muito mais fácil de ler e entender. Observe que o método `query()` também aceita a flag `@` para marcar variáveis locais.

In [25]:
```
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
```

Out[25]: True

Desempenho: quando usar essas funções

Ao ponderar se essas funções devem ser usadas, há duas considerações: *tempo de computação* e *uso de memória*.

O uso da memória é o aspecto mais previsível. Como já mencionado, toda expressão composta envolvendo arrays **NumPy** ou `DataFrame`s do **Pandas** resultará na criação implícita de arrays temporários.

In [26]:
```
# Isto ...
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

In [27]:
```
# ... é aproximadamente equivalente a isso
tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
x = df[tmp3]
```

Se o tamanho dos `DataFrame` s temporários for significativo, comparado à memória disponível do sistema, então é uma boa ideia usar uma expressão `eval()` ou `query()` .

Você pode verificar o tamanho aproximado do seu array em bytes.

In [28]:
```
df.values.nbytes
```

Out[28]:
```
32000
```

Do lado do desempenho, `eval()` pode ser mais rápido mesmo quando você não está maximizando a memória do sistema.

A questão é como seus `DataFrame` s temporários são em relação ao tamanho do cache L1 ou L2 da CPU em seu sistema. Se eles forem muito maiores, então `eval()` pode evitar algum movimento potencialmente lento de valores entre os diferentes caches de memória.

Na prática, acho que a diferença no tempo de computação entre os métodos tradicionais e o método `eval` / `query` geralmente não é significativa – na verdade, o método tradicional é mais rápido para arrays menores!

O benefício de `eval` / `query` está principalmente na memória salva e na sintaxe às vezes mais limpa que eles oferecem.

Cobrimos a maioria dos detalhes de `eval()` e `query()` aqui. Para obter mais informações, você pode consultar a documentação do **Pandas**, em particular veja a seção "Aprimorando o desempenho" (http://pandas.pydata.org/pandas-docs/dev/enhancingperf.html).