desafio 10

Introdução ao Polars

```
from datetime import datetime

print("Executado em:", datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
```

```
Executado em: 2025-10-02 10:59:39
```

- Uma biblioteca Python de código aberto para manipulação e análise de dados.
- Conhecida por seu alto desempenho, especialmente em tarefas que envolvem grandes volumes de dados.
- Construída em Rust, o que contribui para sua velocidade e eficiência.
- Oferece uma API intuitiva e expressiva, similar à do Pandas, facilitando a adoção por usuários familiarizados com essa biblioteca.
- Velocidade: Polars é significativamente mais rápido que o Pandas em muitas operações, graças à sua arquitetura otimizada e ao uso de Rust.
- Eficiência de Memória: Utiliza a memória de forma mais eficiente, permitindo trabalhar com conjuntos de dados maiores sem sobrecarregar o sistema.
- Paralelismo: Aproveita o poder de processamento de múltiplos núcleos para executar tarefas em paralelo, acelerando ainda mais o processamento.
- Flexibilidade: Suporta uma ampla gama de operações de manipulação de dados, como filtragem, agregação, ordenação e junção.
- Integração: Funciona bem com outras bibliotecas populares do ecossistema Python, como NumPy e PyArrow.

- Cenários com Grandes Conjuntos de Dados: Se você trabalha com datasets que não cabem confortavelmente na memória ou que exigem processamento rápido, Polars é uma excelente opção.
- Tarefas de Análise Exploratória de Dados (EDA): A velocidade e eficiência do Polars permitem realizar análises exploratórias de forma mais ágil e interativa.
- Aplicações de Ciência de Dados: A capacidade de lidar com grandes volumes de dados e realizar operações complexas torna o Polars adequado para projetos de aprendizado de máquina e outras áreas da ciência de dados.

Instalação Polars

- O modo mais simples de instalar o Polars é utilizando o terminal do seu computador.
- A instalação pressupõem que o Python já esteja instalado.

```
reticulate::py_install("polars")
```

```
Using virtual environment "\\smb/ra277230/Documentos/.virtualenvs/r-reticulate" ...

+ "\\smb/ra277230/Documentos/.virtualenvs/r-reticulate/Scripts/python.exe" -m pip install --u
```

```
reticulate::py_install("pyarrow")
```

```
Using virtual environment "\\smb/ra277230/Documentos/.virtualenvs/r-reticulate" ...

+ "\\smb/ra277230/Documentos/.virtualenvs/r-reticulate/Scripts/python.exe" -m pip install --u
```

Carregando a biblioteca Polars

- Para utilizar as funções disponibilizadas pela biblioteca, você deve importá-la no início da sua sessão.
- No Python, com o intuito de proteger-se contra conflitos com os nomes de funções disponibilizadas por diferentes bibliotecas, utilizamos frequentemente o conceito de *namespace*.
- Assim, ao importarmos uma biblioteca, recomenda-se atribuir um nome curto (*alias*) à mesma.
- Sempre que formos empregar uma função daquela biblioteca, utilizamos a notação `alias.funcao()`.

```
import polars as pl
```

```
#!pip install polars
#!pip install fastexcel
```

```
import polars as pl
```

Arquivos Tabulares

Leitura de Arquivos Tabulares

- Ao importar arquivos tabulares, o Polars os representa como um objeto de classe `DataFrame` no Python.
- Os tipos de arquivo abaixo podem ser importados da seguinte maneira:
  - Arquivos delimitados (como CSV e TSV): `pl.read_csv()`.
  - Arquivos Parquet: `pl.read_parquet()`.
  - Arquivos JSON: `pl.read_json()`.
  - Arquivos Excel: `pl.read_excel()`.
    - Necessita da biblioteca `fastexcel`!

Argumentos Importantes

- `file`: Caminho para o arquivo ou objeto file-like (para URLs).
  - `pl.read_csv("dados.csv")`
- `columns`: Lista de colunas a serem lidas.
  - `pl.read_csv("dados.csv", columns=["coluna1", "coluna3"])`
- `dtypes`: Dicionário especificando os tipos de dados das colunas.
  - `pl.read_csv("dados.csv", dtypes={"idade": pl.Int32})`
- `infer_schema_length`: Número de linhas para inferir o esquema.
  - `pl.read_csv("dados.grande.csv", infer_schema_length=1000)`

- `has_header`: Indica se o arquivo possui cabeçalho.
  - `pl.read_csv("dados_sem_header.csv", has_header=False)`
- `delimiter`: Delimitador usado no arquivo (padrão é ‘,’).
  - `pl.read_csv("dados.tsv", delimiter="\t")`

Exemplo `airports.csv`

```
getwd()
```

```
[1] "H:/Documentos/me315"
```

```
aeroportos = pl.read_csv("H:/Documentos/me315/airports.csv",
                         columns = ["IATA_CODE", "CITY", "STATE"])
aeroportos.head(2)
```

```
shape: (2, 3)

 IATA_CODE   CITY        STATE
 ---         ---         ---
 str         str         str

 ABE         Allentown   PA
 ABI         Abilene     TX
```

Exemplo `WDIEXCEL.xlsx`

```
wdi = pl.read_excel("H:/Documentos/me315/WDIEXCEL.xlsx",
sheet_name = "Country",
columns = ["Short Name", "Region"])
wdi.head(2)
```

```
shape: (2, 2)

 Short Name    Region
 ---           ---
 str           str

 Aruba         Latin America & Caribbean
 Afghanistan   South Asia
```

Operações em DataFrames

Um Exemplo Simples

```
df = pl.DataFrame({
    "grupo": ["A", "A", "B", "B", "C"],
    "valor1": [10, 15, 10, None, 25],
    "valor2": [5, None, 20, 30, None]
})
df
```

```
shape: (5, 3)

 grupo   valor1   valor2
 ---     ---      ---
 str     i64      i64

 A       10       5
 A       15       null
 B       10       20
 B       null     30
 C       25       null
```

Operando em `valor1`?

```
df["valor1"]
```

```
shape: (5,)
Series: 'valor1' [i64]
[
    10
    15
    10
    null
    25
]
```

```
df["valor1"].mean()
```

```
15.0
```

```
df["valor1"].drop_nulls()
```

```
shape: (4,)
Series: 'valor1' [i64]
[
    10
    15
    10
    25
]
```

```
df["valor1"].drop_nulls().mean()
```

```
15.0
```

Operando em Colunas

```
df.select([
  pl.col("valor1").mean().alias("media_v1"),
  pl.col("valor2").mean()
])
```

```
shape: (1, 2)

 media_v1   valor2
 ---        ---
 f64        f64

 15.0       18.333333
```

Exemplo

Quais são as médias da variável `valor1` e o valor mínimo da variável `valor2` para cada um dos grupos definidos por `grupo`?

```
df.group_by("grupo").agg([
  pl.col("valor1").mean().alias("media_valor1"),
  pl.col("valor2").min().alias("min_valor2")
]).sort("grupo")
```

```
shape: (3, 3)

 grupo   media_valor1   min_valor2
 ---     ---            ---
 str     f64            i64

 A       12.5           5
 B       10.0           20
 C       25.0           null
```

De volta ao `flights.csv`

Calcule o percentual de vôos das cias. aéreas "AA" e "DL" que atrasaram pelo menos 30 minutos nas chegadas aos aeroportos "SEA", "MIA" e "BWI".

```
voos = pl.read_csv("H:/Documentos/me315/flights.csv",
                   columns = ["AIRLINE", "ARRIVAL_DELAY", "DESTINATION_AIRPORT"],
                   dtypes = {"AIRLINE": pl.Utf8,
                             "ARRIVAL_DELAY": pl.Int32,
                             "DESTINATION_AIRPORT": pl.Utf8})
```

```
<string>:1: DeprecationWarning: the argument `dtypes` for `read_csv` is deprecated. It was re
```

```
voos.shape
```

```
(5819079, 3)
```

```
voos.head(3)
```

```
shape: (3, 3)

 AIRLINE   DESTINATION_AIRPORT   ARRIVAL_DELAY
 ---       ---                   ---
 str       str                   i32

 AS        SEA                   -22
 AA        PBI                   -9
 US        CLT                   5
```

Calcule o percentual de vôos das cias. aéreas “AA” e “DL” que atrasaram pelo menos 30 minutos nas chegadas aos aeroportos “SEA”, “MIA” e “BWI”.

```
resultado = (
  voos.drop_nulls(["AIRLINE", "DESTINATION_AIRPORT", "ARRIVAL_DELAY"])
  .filter(
    pl.col("AIRLINE").is_in(["AA", "DL"]) &
    pl.col("DESTINATION_AIRPORT").is_in(["SEA", "MIA", "BWI"])
    )
    .group_by(["AIRLINE", "DESTINATION_AIRPORT"])
    .agg([
      (pl.col("ARRIVAL_DELAY") > 30).mean().alias("atraso_medio")
      ])
)
```

```
resultado.sort("atraso_medio")
```

```
shape: (6, 3)

 AIRLINE   DESTINATION_AIRPORT   atraso_medio
 ---       ---                   ---
```

```
str       str                 f64

DL        BWI                 0.069455
DL        SEA                 0.072967
DL        MIA                 0.090467
AA        MIA                 0.117894
AA        SEA                 0.124212
AA        BWI                 0.127523
```

Dados Relacionais com Polars

Estruturas Relacionais com Duas Tabelas

- Em bancos de dados relacionais, informações frequentemente estão organizadas em múltiplas tabelas que se relacionam entre si através de chaves.
- A chave é uma coluna (ou conjunto de colunas) que permite identificar e associar os dados de uma tabela com os de outra.

```python
import polars as pl

# Criando DataFrames
clientes = pl.DataFrame({
    "cliente_id": [1, 2, 3, 4],
    "nome": ["Ana", "Bruno", "Clara", "Daniel"]
})

print(clientes)
```

```
shape: (4, 2)

 cliente_id  nome
 ---         ---
 i64         str

 1           Ana
 2           Bruno
 3           Clara
 4           Daniel
```

Dados Compras

```
pedidos = pl.DataFrame({
    "pedido_id": [101, 102, 103, 104, 105],
    "cliente_id": [1, 2, 3, 1, 5],
    "valor": [100.50, 250.75, 75.00, 130.00, 79.00]
})

print(pedidos)
```

```
shape: (5, 3)

 pedido_id   cliente_id   valor
 ---         ---          ---
 i64         i64          f64

 101         1            100.5
 102         2            250.75
 103         3            75.0
 104         1            130.0
 105         5            79.0
```

JOINs

- Um `JOIN` combina registros de duas tabelas com base em uma coluna comum (chave).
- É amplamente utilizado em manipulação de dados relacionais.
- Deve ser empregado para combinar informações de duas (ou mais) tabelas que compartilham chaves entre si.

`INNER JOIN`

Retorna apenas as linhas que têm correspondências (de chaves) nas duas tabelas.

Exemplo `INNER JOIN`

```
res_ij = clientes.join(pedidos, on="cliente_id", how="inner")
print(res_ij)
```

```
shape: (4, 4)

 cliente_id  nome   pedido_id  valor
 ---         ---    ---        ---
 i64         str    i64        f64

 1           Ana    101        100.5
 2           Bruno  102        250.75
 3           Clara  103        75.0
 1           Ana    104        130.0
```

LEFT/RIGHT JOIN

Retorna todas as linhas da tabela à esquerda [direita] e as correspondentes da direita [esquerda] (se houver).

Exemplo `LEFT JOIN`

```
res_lj = clientes.join(pedidos, on="cliente_id", how="left")
print(res_lj)
```

```
shape: (5, 4)

 cliente_id  nome    pedido_id  valor
 ---         ---     ---        ---
 i64         str     i64        f64

 1           Ana     101        100.5
 1           Ana     104        130.0
 2           Bruno   102        250.75
 3           Clara   103        75.0
 4           Daniel  null       null
```

Exemplo `RIGHT JOIN`

```
res_rj = clientes.join(pedidos, on="cliente_id", how="right")
print(res_rj)
```

```
shape: (5, 4)

 nome    pedido_id   cliente_id   valor
 ---     ---         ---          ---
 str     i64         i64          f64

 Ana     101         1            100.5
 Bruno   102         2            250.75
 Clara   103         3            75.0
 Ana     104         1            130.0
 null    105         5            79.0
```

`OUTER JOIN`

Retorna todas as linhas quando há uma correspondência em uma das tabelas.

Exemplo `OUTER JOIN`

```
res_oj = clientes.join(pedidos, on="cliente_id", how="outer")
```

```
<string>:1: DeprecationWarning: use of `how='outer'` should be replaced with `how='full'`.
(Deprecated in version 0.20.29)
```

```
print(res_oj)
```

```
shape: (6, 5)

 cliente_id   nome     pedido_id   cliente_id_right   valor
 ---          ---      ---         ---                ---
 i64          str      i64         i64                f64
```

```
1           Ana     101         1                   100.5
2           Bruno   102         2                   250.75
3           Clara   103         3                   75.0
1           Ana     104         1                   130.0
null        null    105         5                   79.0
4           Daniel  null        null                null
```

`CROSS JOIN`

Retorna o produto cartesiano de ambas as tabelas.

Exemplo `CROSS JOIN`

```
res_cj = clientes.join(pedidos, how="cross")
print(res_cj)
```

```
shape: (20, 5)
```

```
 cliente_id  nome    pedido_id  cliente_id_right  valor
 ---         ---     ---        ---               ---
 i64         str     i64        i64               f64

 1           Ana     101        1                 100.5
 1           Ana     102        2                 250.75
 1           Ana     103        3                 75.0
 1           Ana     104        1                 130.0
 1           Ana     105        5                 79.0
 …           …       …          …                 …
 4           Daniel  101        1                 100.5
 4           Daniel  102        2                 250.75
 4           Daniel  103        3                 75.0
 4           Daniel  104        1                 130.0
 4           Daniel  105        5                 79.0
```

Operações em Tabelas Combinadas

- Ao final de uma operação do tipo `JOIN`, a tabela resultante continua sendo um `DataFrame`.

- Todas as operações para `DataFrame` podem ser aplicadas:
  - Filtros
  - Seleções
  - Operações Agregadas

P1: Qual é o valor médio das compras realizadas para cada cliente identificado?

```
print(clientes)
```

```
shape: (4, 2)

 cliente_id   nome
 ---          ---
 i64          str

 1            Ana
 2            Bruno
 3            Clara
 4            Daniel
```

```
print(pedidos)
```

```
shape: (5, 3)

 pedido_id   cliente_id   valor
 ---         ---          ---
 i64         i64          f64

 101         1            100.5
 102         2            250.75
 103         3            75.0
 104         1            130.0
 105         5            79.0
```

```
res = res_ij.group_by(["nome", "cliente_id"]).agg(pl.col("valor").mean())
print(res)
```

```
shape: (3, 3)

 nome    cliente_id   valor
 ---     ---          ---
 str     i64          f64

 Clara   3            75.0
 Ana     1            115.25
 Bruno   2            250.75
```

P2: Informe os nomes e a quantidade de compras com valor mínimo de $100.00 realizadas por cada cliente.

```
print(clientes)
```

```
shape: (4, 2)

 cliente_id   nome
 ---          ---
 i64          str

 1            Ana
 2            Bruno
 3            Clara
 4            Daniel
```

```
print(pedidos)
```

```
shape: (5, 3)

 pedido_id   cliente_id   valor
 ---         ---          ---
```

```
 i64         i64          f64

 101         1            100.5
 102         2            250.75
 103         3            75.0
 104         1            130.0
 105         5            79.0
```

```
res = (res_oj.with_columns(pl.col("valor") > 100)
       .group_by("nome")
       .agg(pl.col("valor").sum()))
print(res)
```

```
shape: (5, 2)

 nome     valor
 ---      ---
 str      u32

 Bruno    1
 null     0
 Ana      2
 Clara    0
 Daniel   0
```

`JOIN` com Múltiplas Colunas como Chave

```
vendas = pl.DataFrame({
    "id_venda": [1, 2, 3],
    "id_cl": [1, 2, 1],
    "id_prod": [101, 102, 103],
    "qtde": [2, 1, 1]
})

detalhes_pedidos = pl.DataFrame({
    "id_ped": [201, 202, 203],
    "cl_id": [1, 2, 1],
    "id_prod": [101, 102, 104],
```

```
    "valor": [50.00, 75.00, 100.00]
})
```

```
print(vendas)
```

```
shape: (3, 4)

 id_venda   id_cl   id_prod   qtde
 ---        ---     ---       ---
 i64        i64     i64       i64

 1          1       101       2
 2          2       102       1
 3          1       103       1
```

```
print(detalhes_pedidos)
```

```
shape: (3, 4)

 id_ped   cl_id   id_prod   valor
 ---      ---     ---       ---
 i64      i64     i64       f64

 201      1       101       50.0
 202      2       102       75.0
 203      1       104       100.0
```

Realizando um `JOIN` com Múltiplas Colunas

```
final = vendas.join(detalhes_pedidos,
                    left_on = ["id_cl", "id_prod"],
                    right_on = ["cl_id", "id_prod"],
                    how = "inner")
print(final)
```

```
shape: (2, 6)

 id_venda   id_cl   id_prod   qtde   id_ped   valor
 ---        ---    ---       ---    ---      ---
 i64        i64    i64       i64    i64      f64

 1          1      101       2      201      50.0
 2          2      102       1      202      75.0
```