LICENCIATURA EM ENGENHARIA INFORMÁTICA

FUNDAMENTOS DE PROGRAMAÇÃO

BIBLIOTECA PANDAS

FILIPE PINTO / JOÃO RODRIGUES

GAIA
2025

Índice

1. Introdução

No âmbito da unidade curricular de Fundamentos de Programação da Licenciatura em Engenharia Informática, foi proposto a realização de um trabalho de investigação sobre bibliotecas da linguagem Python. O grande potencial e a popularidade do Python devem-se, em grande parte, à vasta quantidade de bibliotecas e módulos produzidos pela comunidade, que permitem reduzir o código necessário para realizar tarefas, simples ou complexas. O presente relatório tem como objetivo apresentar a biblioteca Pandas, uma ferramenta essencial no ecossistema Python, e ao longo deste documento, faremos a descrição da biblioteca e das suas áreas de utilização, detalhado o seu processo de instalação e, por fim, demonstrada uma aplicação prática simples, desenvolvida para ilustrar as suas funcionalidades. A escolha desta biblioteca justifica-se pela sua relevância na área de Ciência de Dados e pela sua utilidade prática na manipulação de ficheiros de dados, uma competência transversal a vários projetos.

Neste relatório procuramos descrever a arquitetura, a filosofia e a aplicação prática da biblioteca **pandas**, utilizando o um conjunto de dados, sobre o histórico naufrágio do Titanic.

O objetivo é fornecer uma base teórica para leitores sem experiência prévia, simulando de forma pedagógica, e recorrendo a um ambiente Jupyter Notebook interativo, as funcionalidades básicas que esta biblioteca disponibiliza.

2. Apresentação da biblioteca

Criação da biblioteca Pandas

A biblioteca Pandas é uma das ferramentas mais populares e poderosas de código aberto (*open source*) para a linguagem Python, no que a Análise e Ciência de Dados diz respeito. É uma biblioteca construída sobre a biblioteca **NumPy**, e o seu nome deriva do termo "Panel Data" (dados em painel), um termo técnico para conjuntos de dados estruturados multidimensionais. Foi criado em 2008, pela AQR Capital Management (fonte: https://www.aqr.com/), idealizado e concebido para resolver dificuldades específicas na manipulação de dados financeiros, uma vez que os analistas financeiros enfrentavam um dilema: utilizar a linguagem **R**, que possuía o objeto **data.frame**, e suporte estatístico, mas era difícil de integrar em ambientes de produção, ou utilizar o Python com a biblioteca **NumPy**, que, apesar de rápida para cálculos numéricos puros, falhava categoricamente ao lidar com dados heterogêneos (mistura de *strings*, datas e números), e não possuía o conceito de "rótulos" de dados (*labels*). Nessa época, a analise era frequentemente realizada, como já foi referido em **R**, **Numpy**, **SQL** para a gestão de bases de dados e o Excel para a manipulação visual.

No final do ano de 2009, era já uma biblioteca *open source*, ativamente suportada pela comunidade "pyhton".

Em 2015, a biblioteca pandas passou a fazer parte de um projeto global chamado NumFOCUS (fonte: https://numfocus.org/sponsored-projects), cuja missão é promover código e trabalho open source, nas áreas de investigação, ciência de dados e ciência da computação, servindo de "fiscalizador" (no sentido de garantia de qualidade) desses projetos, e organizando programas educativos na comunidade python, em

particular, o programa **PyData**, como fórum internacional de partilha de experiências para utilizadores e programadores de ferramentas de analise de dados.

Desde que se tornou *open-source*, em 2009, e se juntou ao projeto **NumFOCUS** em 2015, evoluiu para se tornar a ferramenta "de facto" para a manipulação de dados, suportando atualmente mais de 100 milhões de downloads mensais (fonte: https://www.datacamp.com/pt/tutorial/pandas) e servindo como a espinha dorsal de pipelines de dados em indústrias que vão desde a finança à biomedicina.

Timeline (fonte: https://pandas.pydata.org/about/)

- 2008: Inicio do desenvolvimento da biblioteca pandas
- 2009: pandas torna-se *open sorce*
- 2012: Primeira edição do pandas para analise de dados é lançada
- 2015: pandas torna-se um projeto patrocinado pela NumFOCUS

Para utilizadores familiarizados com **SQL**, Linguagem **R** ou **Excel**, o pandas tem funcionalidades equivalentes que permitem analisar e tratar dados. O site do pandas, tem na sua documentação a "tradução" dessas funcionalidades. A título de exemplo:

- SQL
- EXCEL
- Linguagem R

O Papel do Pandas na "Data Stack" atual (2025)

No panorama atual de 2025, a relevância do Pandas não só se mantem, como podemos dizer que é crescente, apesar do surgimento de novas bibliotecas, como é exemplo o **Polars**, que surge como resposta para trabalhar com grandes volumes de dados (fonte: https://realpython.com/polars-

vs-pandas/). Para entender melhor a comparação entre o **Pandas** e o **Polaris**, conselho a leiutra "Polars vs. pandas: What's the Difference?" (fonte: https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/).

O Pandas mantém o seu domínio global em conjuntos de dados de pequena e média dimensão (aqueles que cabem na memória RAM) devido à sua maturidade, à riqueza da sua API e à sua integração eficaz com o restante ecossistema (outras bibliotecas Python) para uso científico, como são exemplo:

- **NumPy**: O Pandas é construído sobre o NumPy, aproveitando a velocidade dos *arrays* otimizados em C para computação numérica.
- **Scikit-Learn**: A maioria dos pipelines de aprendizagem automática (*machine learning*) em Python aceita DataFrames do Pandas como entrada padrão para o treino de modelos.
- **Matplotlib** e **Seaborn**: As bibliotecas de visualização são desenhadas para ingerir estruturas Pandas diretamente, facilitando a criação de gráficos complexos com poucas linhas de código.

NOTA: Abaixo deixamos links relacionados, que ajudam no entendimento da biblioteca:

- Quick overview: Video com funcionalidades e usos
- Livro: Python for Data Analysis
- Cheat Sheet: Data Wrangling
- Try pandas in your browser: link

Estruturas de Dados Fundamentais

O problema central que o pandas se propôs a resolver foi o do alinhamento de dados intrínseco. Em ferramentas baseadas em arrays puros, como o NumPy a referência aos dados é **estritamente posicional**. No mundo real, e especificamente em finanças, dados raramente chegam perfeitamente alinhados ou limpos. Uma série temporal de preços de ações de uma empresa pode não ter registos nos mesmos dias

que outra devido a feriados locais ou falhas de coleta. O pandas introduziu a filosofia de que os dados devem ser carregar com os seus metadados (o índice), permitindo que operações binárias respeitem os rótulos em vez da posição física na memória, garantindo a integridade da análise mesmo na presença de dados ausentes ou dessincronizados.

Para compreender "**como funciona**" o pandas, é necessário analisar para além da sintaxe e examinar a engenharia (arquitetura) subjacente que permite as suas operações.

O pandas não é uma ferramenta isolada; ele é construído sobre a base do **NumPy** (como já referido anteriormente), mas estende as suas capacidades através de uma camada de abstração sofisticada que gere a complexidade dos tipos de dados mistos, enquanto que no **NumPy**, a estrutura de dados primária é o **ndarray** (N-dimensional array), que exige **homogeneidade** de **tipos** para garantir performance.

O pandas resolve essa incompatibilidade entre a necessidade de heterogeneidade dos dados do mundo real e a exigência de homogeneidade do NumPy através de um componente interno denominado **BlockManager**. O BlockManager atua como um orquestrador de memória. Quando um usuário cria um DataFrame com múltiplas colunas de tipos diferentes, o BlockManager não aloca cada coluna separadamente na memória, nem tenta forçar tudo em um único array de tipo genérico (o que seria lento). Em vez disso, ele agrupa colunas de tipos idênticos em "blocos".

Por exemplo, se um DataFrame possui dez colunas, sendo três de números inteiros, cinco de valores decimais e duas de texto, o BlockManager manterá, em teoria, três blocos de memória distintos: uma matriz NumPy 3xN para os inteiros, uma 5xN para os floats e um array de objetos para as strings. Essa arquitetura permite que o pandas delegue operações

matemáticas para o NumPy dentro de cada bloco homogêneo, mantendo a velocidade de execução.

A arquitetura teórica do Pandas assenta em dois tipos de objetos primários: a Series e o DataFrame. A compreensão da distinção e da relação entre estes dois objetos é conceptualmente importante.

Outro elemento estrutural com elevada importância, e potencial, são os Índices.

O Índice é o que separa funcionalmente o Pandas do NumPy. Pandas usa uma técnica chamada tabela hash para guardar os índices. O Indice representa a capacidade de **"lookup"** (procurar, como usamos por exemplo no Excel). No conjunto de dados do Titanic, o índice padrão será um **"RangeIndex"**(0, 1, 2...), mas teoricamente poderíamos definir o índice como sendo o **PassengerId** ou até mesmo o **Name**. Isto permite a chamada de dados baseada em indice significativos em vez de posições arbitrárias, aproximando a manipulação de dados da lógica de bases de dados relacionais (SQL).

Indice

É como um "endereço" ou "etiqueta" para cada linha da tabela. Em vez de procurar os dados linha por linha, o Pandas usa este índice para encontrar rapidamente o que precisa.

Exemplo: Se tens uma lista de nomes, o índice pode ser os números (0, 1, 2…) ou os próprios nomes.

Existem 3 tipos de índices que respondem a lógicas distintas e específicas:

- **RangeIndex**: Uma representação otimizada de uma sequência de inteiros (0, 1, 2...), semelhante à função range do Python, que

consome memória insignificante independentemente do tamanho do DataFrame.

- **DatetimeIndex**: Talvez a contribuição mais significativa do pandas para a analises temporais. Este índice armazena *timestamps* com precisão de nanossegundos e permite "*slicing* inteligente". Um utilizador pode solicitar dados com *strings* parciais, como **df ['2023-01']**, e o índice resolverá automaticamente para selecionar todas as linhas correspondentes a janeiro de 2023, sem a necessidade de operações de filtragem explícitas complexas.
- **MultiIndex** (*Hierarchical Indexing*): Permite armazenar dados de dimensões superiores, numa estrutura tabular bidimensional. Através de múltiplos níveis de indexação (ex: Ano, depois Trimestre, depois Região),o pandas oferece uma alternativa flexível aos painéis multidimensionais sem a complexidade de gerenciar estruturas N-dimensionais reais.

Series

Uma **Series** é a unidade atómica do pandas, é essencialmente um *array* unidimensional indexado. Em termos de folha de cálculo, representaria uma única coluna. No entanto, ao contrário de uma lista simples em Python, uma **Series** possui um Índice (Index) — um conjunto de *rótulos* intrínsecos aos dados que permite uma recuperação eficiente e alinhamento automático.

(fonte: https://pandas.pydata.org/docs/reference/series.html),

Se uma lista Python é um "*saco*" de valores acedidos pela sua posição arbitrária (0, 1, 2...), uma **Series** do Pandas é como um dicionário Python onde as chaves (o índice) são ordenadas e os valores são "tipados" e homogéneos.

Esta dualidade "array-dicionário" é o que confere à **Series** sua flexibilidade:

- Como Array: Suporta slicing (s[0:5]), operações vetorizadas (s * 2) e funções universais do NumPy (np.log(s)).
- Como Dicionário: Permite recuperação de valores por chave especifica (s['indice_a']).

Características das **Series**:

- **Homogeneidade de tipos**: Uma Series armazena tipicamente dados de um único tipo (inteiro, float, string/objeto). Isto permite as chamadas operações vetorizadas — a aplicação de uma função a toda a coluna simultaneamente sem a necessidade de escrever um ciclo for explícito, resultando numa performance computacional superior. Análogo ao que fazemos hoje no Excel ou em SQL.
- **Imutabilidade de tamanho**: Embora os valores individuais possam ser alterados, o tamanho de uma Series não pode ser modificado "*in-place*" sem criar um novo objeto. Isto significa que posso editar os valores existentes, mas não consigo adicionar novos valores a essa Serie, e se necessito de o fazer, tenho de criar uma nova **Series** com base na original, adicionando os novos valores (explicado no **Anexo A**). Esta característica deriva da sua base em arrays **NumPy**.
- **Alinhamento de dados**: As operações entre **Series** alinham-se pelo índice, não pela posição (também demonstrado no **Anexo A**). Se somarmos duas Series, o Pandas fará a correspondência dos rótulos do índice, não dos números das linhas. Isto previne erros comuns no Excel onde o desalinhamento de células leva a cálculos errados "ocultos".
- Pode ser criada a partir de listas ou dicionários.

Exemplos (nota: com fundo cinza os inputs de código, e fundo azul os respetivos outputs)

```
import pandas as pd # a instalação veremos no capitulo 3, mas precisamos de i
mportar para representar os exemplos

s = pd.Series([0.1, 0.2, 0.3, 0.4])
s
```

```
0    0.1
1    0.2
2    0.3
3    0.4
dtype: float64
```

```
s.values # podemos visualizar os valores que constituiem a Series
```

```
array([0.1, 0.2, 0.3, 0.4])
```

```
s.index # Vamos verificar que é atribuido um indice automaticamente (RegeInde
x) a cada um dos elementos da serie, conforme explicado no ponto 2.1.1
```

```
RangeIndex(start=0, stop=4, step=1)
```

```
# Usando esse index podemos "chamar" um elemento especifico, e devolve o valo
r, mas também informação do tipo de dados
s[0]
```

```
np.float64(0.1)
```

Até aqui o comportamento é similar ao **Numpy**, mas ao contrário de um array **NumPy**, esta indexação permite que seja outros tipos de dados que não inteiros:

```
import numpy as np

# vamos alterar o indice da nossa serie, criando uma nova serie com os indice
s (a,b,c,d)
s_index = pd.Series(np.arange(4), index=['a', 'b', 'c', 'd'])
s_index
```

```
a    0
b    1
c    2
d    3
dtype: int64
```

```
# Usando esse index podemos "chamar" um elemento especifico, agora usando o n
ovo indice
s_index['d']
```

```
np.int64(3)
```

Como podemos ver, os objetos ***Series***, podem ser vistos como tendo similaridade com o mapeamento usado nos dicionários python.

E de facto, é possível criar uma ***Series*** diretamente a partir de um dicionário python:

```
populacao_dicionario = {'Portugal': 11.3, 'Italia': 58.9, 'Alemanha': 83.9, '
Grécia': 10.9, 'Espanha': 49.3}
populacao = pd.Series(populacao_dicionario)
populacao
```

```
Portugal    11.3
Italia      58.9
Alemanha    83.9
Grécia      10.9
Espanha     49.3
dtype: float64
```

Podemos aceder, via índice, a informação tal como acedemos nos dicionários:

```
populacao['Grécia']
```

```
np.float64(10.9)
```

Mas com as capacidades de um array de Numpy, algo que diretamente nos dicionários não conseguimos. Vamos neste exemplo aplicar a multiplicação a todos os valores de "população":

```
populacao * 1000
```

```
Portugal    11300.0
Italia      58900.0
Alemanha    83900.0
Grécia      10900.0
Espanha     49300.0
dtype: float64
```

DataFrame

O **DataFrame** é a estrutura de dados principal do Pandas. É uma estrutura tabular **bidimensional**, mutável em tamanho, e potencialmente heterogénea, com eixos indexados (linhas e colunas). É uma coleção de Series que partilham o mesmo índice de linhas, no entanto, a sua implementação física no BlockManager garante que ele é mais eficiente do que um simples dicionário de Series (fonte: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame).

Bidimensional, pois o DataFrame possui dois eixos:

- **Eixo 0 (Index):** O eixo vertical, representando as linhas.
- **Eixo 1 (Columns):** O eixo horizontal, representando as colunas.

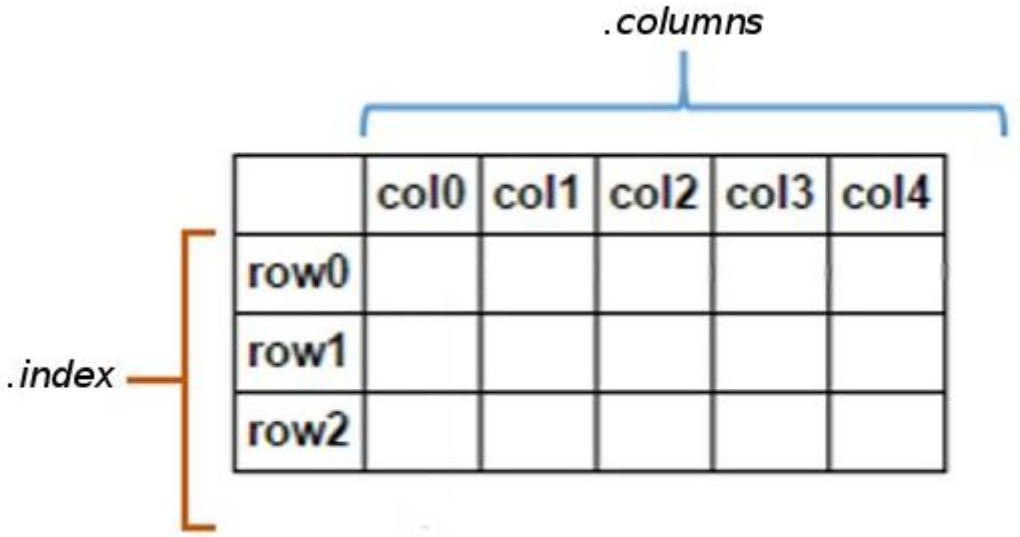

Figura 1 - DataFrame, eixos. (fonte: https://github.com/jorisvandenbossche/pandas-tutorial/blob/master/02%20-%20Data%20structures.ipynb)

A maioria das operações de agregação ou estatística exige a especificação do eixo. Por exemplo, o método ***.mean(axis=0)*** calcula a média de cada coluna (agregando o valor das linhas), enquanto ***.mean(axis=1)*** calculará a média de cada linha (agregando através das colunas).

Características dos **DataFrame**:

- É a estrutura mais utilizada.
- É **bidimensional** (semelhante a uma tabela inteira ou folha de cálculo).
- Possui **índices** para as linhas e **cabeçalhos** para as colunas:
- Colunas: Cada coluna num DataFrame é, na realidade, uma Series.
- Linhas: As linhas partilham um rótulo de Índice.
- Células: A interseção de uma linha e uma coluna.

- Pode ser criado a partir de dicionários, listas ou leitura de ficheiros externos.

Distinguir os dois conceitos **Series/DataFrame**, é fundamental. Enquanto uma coluna específica (uma Series) deva ser consistente no seu tipo de dados (por exemplo, todos os valores são inteiros), o DataFrame como um todo pode conter colunas de inteiros ao lado de colunas de texto e colunas de datas.

Esta estrutura flexível permite representar conjuntos de dados complexos como vamos ver nos dados sobre os passageiros do Titanic, que contém nomes (*strings*), idades (*floats*) e estatuto de sobrevivência (inteiros/booleanos).

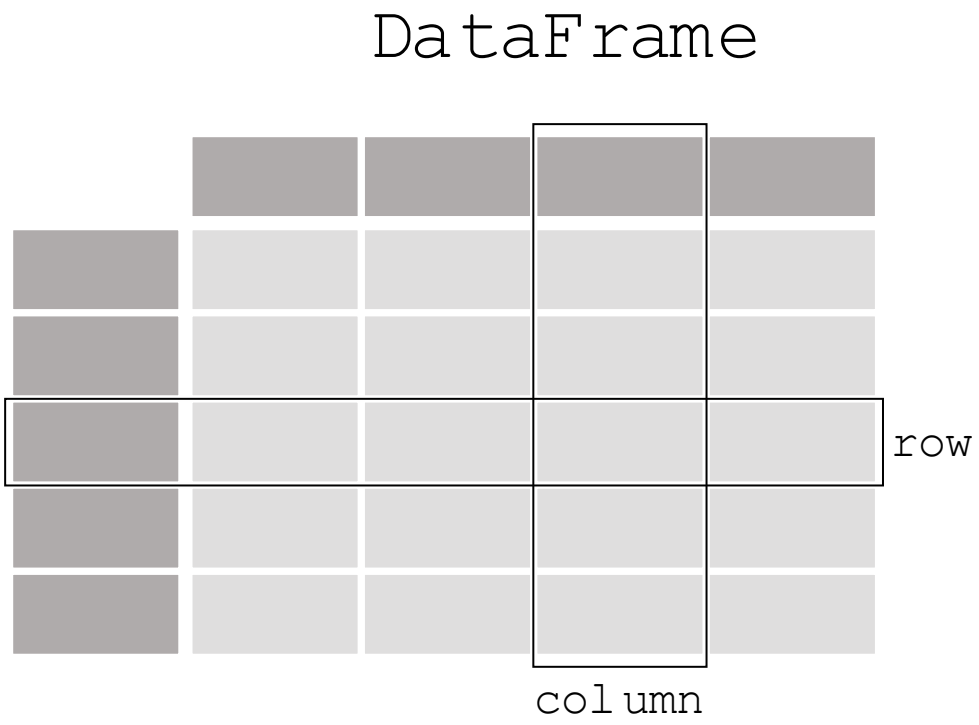

Figura 2 - DataFrame: Linhas, Colunas e Células

Exemplos:

```
# Uma das formas mais comuns de criar DataFrames é um dicionário de listas ou arrays.
dados = {'Pais': ['Portugal', 'Itália', 'Alemanha', 'Grécia', 'Espanha'],
# Lista de Paises
        'populacao': [11.3, 58.9, 83.9, 10.9, 49.3],
# Lista de valores de população
        'area': [92.212 , 302.073, 357.683, 131.957, 506.030],
# Lista de valores para àrea
        'capital': ['Lisboa', 'Roma', 'Berlim', 'Atenas', 'Madrid']}
# Lista de Capitais
paises = pd.DataFrame(dados)
paises
```

```
       Pais  populacao     area capital
0  Portugal       11.3   92.212  Lisboa
1    Itália       58.9  302.073    Roma
2  Alemanha       83.9  357.683  Berlim
3    Grécia       10.9  131.957  Atenas
4   Espanha       49.3  506.030  Madrid
```

Vamos verificar os atributos descritos, dos *DataFrame*.

Um DataFrame, além de um atributo de **.index**, tem também um atributo **.columns**:

```
paises.index # Este atributo comporta-se da mesma forma que nas Series
```

```
RangeIndex(start=0, stop=5, step=1)
```

```
paises.columns # Este atributo descreve o nome das colunas que compõem o data set
```

```
Index(['Pais', 'populacao', 'area', 'capital'], dtype='object')
```

E para provar que cada coluna pode ter o seu tipo de dados conforme falamos acima:

```
paises.dtypes
```

```
Pais          object
populacao    float64
area         float64
capital       object
dtype: object
```

Como vimos nas *Series*, e para demonstrar o potencial do pandas no tratamento de dados, podemos facilmente realizar operações com os valores das colunas, e criar novas colunas (ou *features* como se designam na engenharia de dados).

```
paises['populacao'] / paises['area']
```

```
0    0.122544
1    0.194986
2    0.234565
3    0.082603
4    0.097425
dtype: float64
```

Principais Funcionalidades (O que permite fazer?)

Leitura e Escrita de Dados (I/O):

- Capacidade ler e escrever ficheiros em vários formatos, sendo o CSV (read_csv) e Excel os mais comuns, mas também em sql, json, parquet...
- Permite definir cabeçalhos, índices e tratar caracteres especiais durante a importação.

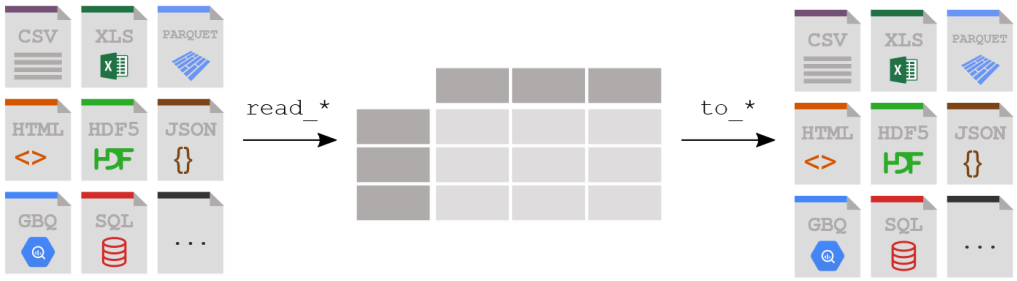

Figura 3 - Pandas, tipo de ficheiros de leitura e escrita

Limpeza e Tratamento de Dados:

- Tratamento de valores nulos: Identificação (*isnull*), remoção ou substituição de dados em falta *(missing values* - NaN).
- Filtrar: Seleção de dados baseada em condições booleanas (ex: "mostrar apenas filmes lançados após 1990").
- Remoção de duplicados: Identificar e limpar registos repetidos.

Manipulação e Análise:

- *Slicing* e Indexação: Seleção precisa de linhas e colunas (loc, ix).
- Operações de *String*: Manipulação de texto dentro das colunas.

- Agrupamento (GroupBy): Agrupar dados por categorias para realizar cálculos (somas, médias, contagens), semelhante às tabelas dinâmicas (*pivot tables*).
- Fusão (Merge/Join): Combinar diferentes DataFrames baseados em chaves comuns (como os JOINs em SQL).

Séries Temporais (Time Series):

- O Pandas é excecionalmente forte, tendo suporte nativo, para trabalhar com datas e horas (date_range, conversão de *strings* para datas), particularmente útil em dados financeiros de ações, registos meteorológicos, gestão logística...
- Ferramentas para reamostragem (*resampling*) e janelas móveis (*moving windows*) para análise de tendências ao longo do tempo.

Visualização:

- Integração direta com a bibliotecas como **Seaborn** e **Matplotlib**, para criar gráficos (linhas, barras, histogramas) diretamente a partir dos DataFrames.

Em resumo, quando é que precisamos de usar o **pandas**? Quando estamos a trabalhar com dados tabelares ou estruturados, e queremos:

- Importar dados
- Limpar dados
- Explorar dados e obter informação rápida desses dados
- Processar e preparar dados para analise
- Analisar dados (isto em conjunto com outras bibliotecas como matplotlib, scikit-learn, etc...)

3. Processo de instalação

A biblioteca Pandas está alojada no Python Package Index (**PyPI**), que é o repositório oficial de software para a linguagem de programação Python.

Sendo uma biblioteca externa, não vem instalada por defeito com o interpretador base do Python e necessita de ser descarregada e configurada no ambiente de desenvolvimento. Uma vez que estamos no ambiente de um Jupiter, temos de usar o "!" antes do pip, algo que não é necessário se instalado diretamente na consola.

```
!pip install pandas
```

Ao importarmos a biblioteca, convenciona-se a utilização do alias ***pd***, que não sendo obrigatória, ajuda depois na geração e manutenção do código. Esta abreviatura não é apenas uma questão estética ou de redução de tempo ou escrita, mas um padrão da comunidade que torna o código legível e partilhável universalmente.

Nota: vamos também importar o numpy, que como referimos acima é a base do pandas, e será útil para visualizar funcionalidades que o pandas irá permitir realizar. ***Mas não é de todo obrigatório*** *importar o numpy para que o pandas seja totalmente*

```
import pandas as pd
import numpy as np # Importamos também o Numpy pois será útil mais à frente

print(pd.__version__) #Verificar se temos o pandas corretamente instaladpo,
e em que versão
```

```
2.3.3
```

4. Projeto

Para consolidar os conceitos teóricos de Series, DataFrames, Indexação e Limpeza de Dados, realizaremos uma análise do conjunto de dados do Titanic. Este dataset é considerado o "Hello World" da ciência de dados devido à sua mistura de tipos de dados (numéricos, categóricos, texto livre) e problemas de qualidade de dados (valores nulos, outliers), oferecendo um ambiente perfeito dos desafios reais enfrentados por analistas e engenheiros de dados.

O objetivo deste projeto não é construir um modelo preditivo de *Machine Learning* (embora fosse o passo seguinte natural), mas sim realizar uma Análise Exploratória de Dados (EDA), demonstrando como o Pandas serve como ferramenta de "Data Wrangling" e inferência estatística.

Preparação do Ambiente e Importação de Bibliotecas

A primeira etapa em qualquer análise é a importação das ferramentas necessárias.

Além do Pandas e NumPy para manipulação, importaremos Matplotlib e Seaborn para visualização de dados. O Seaborn é particularmente relevante, porque "fala a língua" do Pandas, aceitando DataFrames diretamente como input e interpretando os rótulos das colunas automaticamente.

Instalamos também estas duas bibliotecas, que vão ser necessárias para a visualização gráfica

```
!pip install seaborn
!pip install matplotelib
```

Importamos as bibliotecas que vamos precisar.

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

```
import seaborn as sns
import matplotlib.pyplot as plt
```

Ingestão de Dados (ETL: *Extraction*)

O Pandas oferece uma suite de funções de leitura, função **read_csv** (read_csv, read_excel, read_sql, read_json, etc.). Utilizaremos read_csv para carregar os dados diretamente de um repositório Kaggle.

Teoria por trás do read_csv: Quando esta função é chamada, o Pandas realiza um varrimento inicial (buffer) para determinar o delimitador (neste caso a virgula) e os tipos de dados de cada coluna. Se uma coluna contiver misturas de números e *strings*, o Pandas adota o tipo da coluna para o tipo mais genérico (object), o que consome mais memória. Definir tipos explicitamente ou usar conversores pode otimizar esse processo em grandes datasets.

Nota: esta inferência é útil, mas pode ser computacionalmente dispendiosa e ter um consumo de memória relevante, quandos se trata de ficheiros com milhões de registos. Nesse tipo de cenários, é possível especificar os tipos de dados (dtype) manualmente pode otimizar este processo.

(Fonte: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html).

```
# Vamos ter de instalar e importar a biblioteca Kagglehub, para termos acesso via API ao ficheiro CSV mais recente.
!pip install Kagglehub
import kagglehub

# Vamos importar a biblioteca os, para nos permitir construir o link atraves do path fornecido pela API
import os

# Download da última versão do dataset
path = kagglehub.dataset_download("yasserh/titanic-dataset")

# Verificar que temos um caminho define através do print
print("Path to dataset files:", path)

# Construir o caminho para o CSV
csv_path = os.path.join(path, "Titanic-Dataset.csv")
```

```
# Exemplo de ingestão de dados num ambiente Jupyter
url = "https://raw.githubusercontent.com/jorisvandenbossche/pandas-tutorial/master/data/titanic.csv"

# Carregar o conjunto de dados Titanic no nosso DataFrame do pandas, de nome "titanic"
titanic = pd.read_csv(csv_path)

# Verificar o tipo de dados
print(type(titanic)) # Deve retornar <class 'pandas.core.frame.DataFrame'>
```

```
<class 'pandas.core.frame.DataFrame'>
```

A função **to_csv**, exporta o nosso dataframe no formato que pretendermos (a mesma lista de formato que temos para o *read*). Imaginemos que pretendemos exportar para formato Excel, o ficheiro que vamos importar, para registo de auditoria, com o mesmo nome e data da ação.

Nota: Função **to_csv**. A função de exportação, tal como a de importação suporta nativamente ficheiros .CSV, como vimos no exemplo anterior. Se pretrendermos importar ou exportar em outros formatos precisamos sempre de immporetr bibliotecas adicionais como *openpyxl* no caso de ficheiros .xlsx, ou *pdfkit* no caso de ficheiros PDF.

Analise Exploratória de Dados

Após o carregamento, é geralmente realizada uma inspeção os dados. Compreender o significado de cada variável é um pré-requisito para qualquer análise teórica ou modelação de dados, assim como para analises estatísticas.

Não devemos assumimos que os dados estão corretos, e para isso vamos realizamos algumas "questões" ao nosso dataset. O Pandas oferece métodos essenciais para esta Análise Exploratória de Dados (EDA), que são altamente simples, mas representam informação valiosa para esta análise.

Nota: "df" representa DataFrame, e é usado na documentação, mas no nosso código será substituído pelo nome do nosso DataFrame, que no nosso caso, será "titanic"

Funções head and tail

Estes métodos retornam as primeiras ou últimas, respetivamente, 5 linhas do DataFrame. A sua função teórica é confirmar se o cabeçalho foi analisado corretamente e se os dados parecem consistentes com o esperado.

Vamos ver a tabela, que representa o nosso DataFrame, no primeiro caso com os 5 primeiros registos (pacientes) e respetivas colunas de atributos, e no segundo caso os últimos 5 registos. Nota, os primeiros e últimos de acordo com o índice, que por omissão é o RegeIndex.

```
print("--- Primeira 5 linhas do DataFrame Titanic ---")
titanic.head()
```

```
--- Primeira 5 linhas do DataFrame Titanic ---

   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

```
print("--- Últimas 5 linhas do DataFrame Titanic ---")
titanic.tail()
```

```
--- Últimas 5 linhas do DataFrame Titanic ---

     PassengerId  Survived  Pclass                                      Name
\
886          887         0       2                     Montvila, Rev. Juozas
887          888         1       1              Graham, Miss. Margaret Edith
888          889         0       3  Johnston, Miss. Catherine Helen "Carrie"
889          890         1       1                     Behr, Mr. Karl Howell
890          891         0       3                       Dooley, Mr. Patrick

        Sex   Age  SibSp  Parch      Ticket   Fare Cabin Embarked
886    male  27.0      0      0      211536  13.00   NaN        S
887  female  19.0      0      0      112053  30.00   B42        S
888  female   NaN      1      2  W./C. 6607  23.45   NaN        S
889    male  26.0      0      0      111369  30.00  C148        C
890    male  32.0      0      0      370376   7.75   NaN        Q
```

Função df.shape

Retorna um tuplo (linhas, colunas). A verificação da dimensionalidade é o primeiro passo para garantir a integridade da ingestão de dados, e percebermos se foram corretamente "lidos" pelo pandas.

Ficamos a saber que o nosso DataFrame tem 891 linhas (em teoria 891 passageiros) e cada linha tem 12 atributos (colunas).

```
print("--- Formato (Linhas, colunas) DataFrame Titanic ---")
titanic.shape
```

```
--- Formato (Linhas, colunas) DataFrame Titanic ---

(891, 12)
```

Função df.info()

Uma função que nos dá informação relevante para a gestão de memória (tipos de dados ocupam recursos de memória diferentes) e verificação de tipos de dados. Revela por exemplo, quais as colunas que têm valores nulos, e permite ao analista decidir estratégias de limpeza antes de qualquer cálculo ou analise.

Podemos inferir que as colunas `Age`, `Embarked` e `Cabin` tem vários valores nulos, uma vez que temos menos valores em cada uma dessas colunas, do que temos linhas (891). Mais à frente ajudará a decidir estratégias de limpeza e preparação dos dados.

```
print("--- Informação DataFrame Titanic ---")

titanic.info()
```

```
--- Informação DataFrame Titanic ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

df.describe():

Fornece estatísticas descritivas (média, desvio padrão, min, max, quartis) para as colunas com valores numéricos.

Concluímos que apenas para as colunas com valores numéricos, o pandas vai apresentar um resumo com métricas estatísticas gerais de cada uma dessas colunas. Observamos que a idade média é de cerca de 29.7 anos, mas o desvio padrão é elevado. Além disso, a coluna `Survived` (0 ou 1) tem uma média de 0.38, o que nos diz imediatamente que a taxa de sobrevivência global foi de aproximadamente 38%, sem a necessidade de realizar quaisquer cálculos.

```
titanic.describe()
```

```
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

Limpeza e Tratamento de Dados (Data Wrangling)

A limpeza, tratamento e enriquecimento dos dados, ou Data Wrangling (fonte: https://www.ibm.com/think/topics/data-wrangling), consome a maior parte do tempo de um cientista de dados, e o pandas fornece ferramentas bastante eficazes para essa tarefa.

A **limpeza de dados** é guiada por decisões sobre a natureza da ausência dos dados. Conforme vimos através da função `info()`, temos colunas com valores nulos. Vamos analisar as **Estratégias de Resolução** para valores nulos:

- **Eliminação** (Dropping): `titanic.dropna(subset=['Age'])` remove as linhas onde a idade é desconhecida. Embora simples, isto reduz o tamanho da amostra e pode introduzir viés se os dados em falta não forem aleatórios.
- **Imputação** (Imputation): Preencher os valores em falta usando `titanic.fillna()`.
    - Imputação **Simples**: Preencher com a idade média global (`titanic['Age'].mean()`).
    - Imputação **Contextual**: Preencher com a mediana da idade do género e classe específicos. Por exemplo, preencher a idade em falta de um homem da 3ª classe com a mediana das idades de outros homens da 3ª classe. Esta abordagem preserva a distribuição estatística e a integridade dos dados muito melhor do que uma média global.

No nosso dataframe:

- **Coluna Cabin**: Com mais de 75% dos dados ausentes, a imputação (atribuir um valor) introduziria um desvio estatístico significativo. A decisão mais conservadora, será a remoção da característica (*Feature Dropping*).
- **Coluna Age**: A idade é um valor preditivo geralmente de grande valor para sobrevivência. A remoção das linhas reduziria significativamente o tamanho da amostra (perda de informação). A imputação é preferível. Usaremos a mediana em vez da média, pois a média é sensível a *outliers* (passageiros muito idosos), enquanto a mediana é uma medida de tendência central com maior significado.
- **Coluna Embarked**: Possui apenas 2 valores nulos. Remover essas duas linhas (observações) tem impacto estatístico insignificante, pelo que faremos o drop mas apenas dessas linhas.

```
# Vamos criar o Pipeline de Limpeza

# Criar uma cópia para preservar o raw data (boa prática)
titanic_clean = titanic.copy()

# 1. Remoção de coluna com excesso de nulos: Coluna Cabine
titanic_clean.drop(columns=['Cabin'], axis=1, inplace=True)  # axis=1 indica
operação na coluna vertical

# 2. Imputação da Idade com a Mediana das idades
mediana_idade = titanic_clean['Age'].median()
titanic_clean['Age'].fillna(mediana_idade, inplace=True)

# 3. Remoção de linhas Embarked com valor nulo
titanic_clean.dropna(subset=['Embarked'], inplace=True)  # axis=0 (padrão) i
ndica operação na linha

# Verificação do nosso datafram após limpeza
print("Valores nulos restantes no dataset limpo:", titanic_clean.isnull().su
m().max())
print("-------------------------------------")
```

```
Valores nulos restantes no dataset limpo: 0
-------------------------------------
```

Podemos recorrer ao `info()`, aplicado ao dataset `titanic_clean` para verificar se funcionou:

```
titanic_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 889 entries, 0 to 890
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  889 non-null    int64
 1   Survived     889 non-null    int64
 2   Pclass       889 non-null    int64
 3   Name         889 non-null    object
 4   Sex          889 non-null    object
 5   Age          889 non-null    float64
 6   SibSp        889 non-null    int64
 7   Parch        889 non-null    int64
 8   Ticket       889 non-null    object
 9   Fare         889 non-null    float64
 10  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(4)
memory usage: 83.3+ KB
```

Verificamos que a coluna **`Age`** já não tem valores nulos! Verificamos que já não existe a coluna **`Cabin`**! E verificamos que temos no total 889 registos, pois 2 foram eliminados devido ao valor nulo da coluna **`Embarked`**.

Feature engeneering (ou engenharia de variáveis): O Feature Engineering, ou engenharia de variáveis é o processo de usar conhecimento de domínio para criar novas variáveis com valor para a analise ou pipeline de dados.. Pode ser por exemplo pela

- **Manipulação de Texto**, .str expõe métodos vetorizados para manipulação de *strings* (ex: df['nome'].str.upper(), df['email'].str.contains('@')). Isso permite aplicar operações de string a colunas inteiras sem a penalidade de performance de loops explícitos em Python.
- Pode ser através de **Colunas calculadas**, onde com base em outras colunas já existente criamos uma nova coluna.

Manipulação de Texto

Vamos criar campos separados para "UltNome" (sobrenome) e "Nome" (restantes nomes) para análise de famílias e normalização de nomes, e poder enriquecer os nossos dados com novas "features".

```
titanic_clean['UltNome'] = titanic_clean['Name'].str.split(',').str[0].str.s
trip()         # Como os nomes estão no padrão Sobrenome, Título. PrimeiroNom
e…. Para UltNome ficamos com o que está antes e da virgula
titanic_clean['Nome'] = titanic_clean['Name'].str.split(',').str[1].str.stri
p()            # Neste caso ficamos com o que está após a virgula.
titanic_clean[['Name','UltNome','Nome']].head(5)
# Vamos ver as primeiras 5 linhas para verificar as duas novas colunas.
```

```
                                                Name    UltNome  \
0                            Braund, Mr. Owen Harris     Braund
1  Cumings, Mrs. John Bradley (Florence Briggs Th...    Cumings
2                             Heikkinen, Miss. Laina  Heikkinen
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)   Futrelle
4                           Allen, Mr. William Henry      Allen

                                    Nome
0                          Mr. Owen Harris
1  Mrs. John Bradley (Florence Briggs Thayer)
2                               Miss. Laina
3         Mrs. Jacques Heath (Lily May Peel)
4                          Mr. William Henry
```

Outro exemplo será detetar possíveis famílias por sobrenome repetido, e para isso criamos uma nova "feature" com um booleano, se o nome é repetido ou não.

```
UltNome_counts = titanic_clean['UltNome'].value_counts()
titanic_clean['NomeFamiliaRepetido?'] = titanic_clean['UltNome'].map(lambda
s: UltNome_counts.get(s, 0) > 1)
titanic_clean[['Name','UltNome','NomeFamiliaRepetido?']].head(8)
```

```
                                                Name    UltNome  \
0                            Braund, Mr. Owen Harris     Braund
1  Cumings, Mrs. John Bradley (Florence Briggs Th...    Cumings
2                             Heikkinen, Miss. Laina  Heikkinen
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)   Futrelle
4                           Allen, Mr. William Henry      Allen
5                                   Moran, Mr. James      Moran
6                            McCarthy, Mr. Timothy J   McCarthy
7                  Palsson, Master. Gosta Leonard     Palsson

   NomeFamiliaRepetido?
0                  True
1                 False
2                 False
3                  True
4                  True
5                  True
6                 False
7                  True
```

Podemos, também com recurso à extração de texto, relacionar o título (presente no nome) com a taxa de sobrevivência.

```
# Usando **titanic_clean['Name'].str.extract(' ([A-Za-z]+)\.')**, esta extraç
ão baseada em expressões regulares (_RegEx_) permite-nos agrupar não apenas p
or género, mas por título social.
titanic_clean['Title'] = titanic_clean['Name'].str.extract(r' ([A-Za-z]+)\.
')

# Calculamos a media de sobrevivência por Titulo
titanic_clean.groupby('Title')['Survived'].mean()
```

```
Title
Capt        0.000000
Col         0.500000
Countess    1.000000
Don         0.000000
Dr          0.428571
Jonkheer    0.000000
Lady        1.000000
Major       0.500000
Master      0.575000
Miss        0.696133
Mlle        1.000000
Mme         1.000000
Mr          0.156673
Mrs         0.790323
Ms          1.000000
Rev         0.000000
Sir         1.000000
Name: Survived, dtype: float64
```

Ficamos a saber, por exemplo, que "Master" (um título vitoriano para jovens rapazes) tem uma taxa de sobrevivência muito superior a "Mr." (homens adultos), e parece também confirmando a aplicação estrita da "regra social" ***mulheres e crianças primeiro***.

Colunas calculadas

Vamos criar uma nova coluna `TamanhoFamilia` somando SibSp (irmãos/cônjuges) e Parch (pais/filhos).

```
titanic_clean['TamanhoFamilia'] = titanic_clean['SibSp'] + titanic_clean['Parch']     #Somamos duas colunas para originar a nova coluna
display(titanic_clean[['Name', 'TamanhoFamilia']].head(3))                  #Vamos ver as primeiras 3 linhas para verificar a nova coluna, calculada.
```

```
                                                Name  TamanhoFamilia
0                            Braund, Mr. Owen Harris               1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...               1
2                             Heikkinen, Miss. Laina               0
```

Agora que temos a nova coluna (feature) "TamanhoFamilia" já podemos relacionar com a coluna "Survive", calculando a média por cada "tamanho":

```
# RELACIONAR familysize com sobrevivencia
sobrevivencia_por_familia = titanic_clean.groupby('TamanhoFamilia')['Survived'].mean()
sobrevivencia_por_familia
```

```
TamanhoFamilia
0     0.300935
1     0.552795
2     0.578431
3     0.724138
4     0.200000
5     0.136364
6     0.333333
7     0.000000
10    0.000000
Name: Survived, dtype: float64
```

Conclusão possível destes novos dados, seria que as maiores taxas de sobrevivência foram para agregados de 3 ou menos pessoa, ou seja, que famílias com mais que 3 elementos dificilmente sobreviveram. Inferência possível seria, que por tentarem reunir ou procurar os elementos do seu agregado familiar, perderam tempo crucial à sobrevivência, ou que não se colocaram a salvo em botes de salva-vidas, por tentarem ir juntos.

Indexação, Seleção e Filtragem

Uma vez carregados os dados, feita uma análise inicial, limpeza e engenharia de variáveis, outra tarefa comum é selecionar subconjuntos de interesse, para responder a diferentes questões. O pandas oferece múltiplos paradigmas de seleção, o que historicamente gerou alguma confusão, e levou à padronização atual fas funções **.loc** e **.iloc**.

Um dos obstáculos mais comuns quando nos iniciamos no usa da biblioteca é a confusão entre selecionar dados por rótulo (label) versus selecionar dados por posição (Integer Location), que são utilizados respetivamente pelos comandos **.loc** e **.iloc**.

- **Seleção Baseada em Rótulos (.loc)**: É a forma mais explícita e recomendada. **df.loc['linha_a', 'coluna_b']** recupera dados baseados estritamente nos nomes do índice e das colunas. Uma particularidade do *slicing* neste caso, é que é inclusiva em ambas as extremidades (ao contrário da regra padrão do Python).
- **Seleção Baseada em Posição (.iloc)**: Funciona como a indexação de listas ou arrays NumPy. **df.iloc[0:5, 1]** seleciona as primeiras cinco linhas da segunda coluna. Aqui, o slicing segue a regra padrão do Python (exclui o último valor, neste caso o 5).
- **Indexação Booleana**: É a ferramenta essencial para filtragem exploratória. Expressões como **df[df['Idade'] > 18]** criam uma máscara booleana (uma Series de True/False) que é usada para filtrar o DataFrame, retornando apenas as linhas onde a condição é verdadeira. Permite ainda o uso de operadores **(&, |, ~)** para combinar múltiplas condições.
- **Filtros**: Uma das grandes funcionalidades do pandas, é a filtragem de dados baseada em condições lógicas. Isto utiliza a lógica vetorizada, permitindo perguntas complexas aos dados.

iloc (Localização por Posição)

O indexador .iloc é puramente baseado em posições inteiras (de 0 a comprimento-1). O limite superior com iloc é exclusivo (o índice final não é incluído), mantendo a consistência com a indexação do Python.

Sintaxe: **df.iloc[índice_linha, índice_coluna]**

Exemplo: **titanic.iloc** retorna os valores da primeira linha do conjunto de dados Titanic (Sr. Owen Harris Braund).

Exemplo de aplicação a um intervalo: **titanic.iloc[0:5, 0:2]** retorna as primeiras 5 linhas e as primeiras 2 colunas, que são "PassangerId" e "Survived".

```
titanic_clean.iloc[0]  # Primeira linha do DataFrame
```

```
PassengerId                                  1
Survived                                     0
Pclass                                       3
Name                       Braund, Mr. Owen Harris
Sex                                         male
Age                                         22.0
SibSp                                          1
Parch                                          0
Ticket                                 A/5 21171
Fare                                        7.25
Embarked                                       S
UltNome                                  Braund
Nome                               Mr. Owen Harris
NomeFamiliaRepetido?                          True
Title                                           Mr
TamanhoFamilia                                 1
Name: 0, dtype: object
```

```
titanic_clean.iloc[0:5, 0:2]  # Primeiras 5 linhas e primeiras 2 colunas do DataFrame
```

```
   PassengerId  Survived
0            1         0
1            2         1
2            3         1
3            4         1
4            5         0
```

loc: Label Location (Localização por *Rótulo*)

O indexador .loc é baseado em rótulos.

Sintaxe: **df.loc[rótulo_linha, nome_coluna]**

- Exemplo: **df.loc[0, 'Name']** retorna o nome associado ao rótulo de índice 0.

O limite superior no *slicing* com loc é inclusivo. Esta é a maior diferença com o iloc: titanic.loc[0:5] retorna 6 linhas (os rótulos 0, 1, 2, 3, 4 e 5).

A razão para isto é que, se os rótulos fossem strings (ex: "Jan", "Fev", "Mar"), não seria intuitivo saber qual é o "próximo" elemento para fazer a exclusão, pelo que o Pandas opta por incluir o último rótulo explicitamente.

Aparentemente, titanic.loc e titanic.iloc parecem fazer a mesma coisa. No entanto, se ordenássemos o DataFrame por Idade (titanic.sort_values('Age')), a linha na posição 0 (a primeira linha física) poderia ter o rótulo 630. Nesse caso:

- **iloc** retornaria o passageiro mais jovem (que está na primeira posição física).
- **loc** retornaria o passageiro com o ID 0 (que agora pode estar no meio da tabela).

```
titanic_clean.loc[0:5]  # Primeiras 6 linhas do DataFrame usando loc
```

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3
5            6         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0
5                                   Moran, Mr. James    male  28.0      0

   Parch            Ticket     Fare Embarked    UltNome  \
0      0         A/5 21171   7.2500        S     Braund
1      0          PC 17599  71.2833        C    Cumings
2      0  STON/O2. 3101282   7.9250        S  Heikkinen
3      0            113803  53.1000        S   Futrelle
4      0            373450   8.0500        S      Allen
5      0            330877   8.4583        Q      Moran
```

```
                                     Nome  NomeFamiliaRepetido? Title  \
0                             Mr. Owen Harris                 True    Mr
1  Mrs. John Bradley (Florence Briggs Thayer)                False   Mrs
2                                Miss. Laina                False  Miss
3         Mrs. Jacques Heath (Lily May Peel)                 True   Mrs
4                         Mr. William Henry                 True    Mr
5                                  Mr. James                 True    Mr

   TamanhoFamilia
0               1
1               1
2               0
3               1
4               0
5               0
```

Indexação Booleana

```
# Criar a máscara booleana (Series de True/False) para nos indicar os pacient
e maiores de idade
mascara_adulto = titanic_clean['Age'] > 18

# Visualizar a máscara (apenas para entendimento do funcionamento da indexaçã
o booleana)

print(mascara_adulto.head())
```

```
0    True
1    True
2    True
3    True
4    True
Name: Age, dtype: bool
```

Filtros

Permite criar perguntas de investigação:

- **Pergunta**: Quero identificar todas as passageiras do sexo feminino que viajavam em 1ª Classe?
- **Método**: Criamos uma "máscara" — uma Series booleana de valores True/False — e aplicamo-la ao DataFrame.

NOTA: As palavras-chave do Python and ou or, no contexto do Pandas, não funcionam porque o Python tenta avaliar a "verdade" de toda a Series de uma só vez. Em vez disso, deve-se obrigatoriamente usar os operadores & (para E) e | (para OU), e encapsular cada condição individual entre parênteses para garantir a precedência correta das operações.

```
# Todas as passageitas do sexo feminino que viajavam na primeira classe
mulhares_primeira_classtitanic = titanic[(titanic['Pclass'] == 1) & (titanic[
'Sex'] == 'female')]
mulhares_primeira_classtitanic
```

```
     PassengerId  Survived  Pclass  \
1              2         1       1
3              4         1       1
11            12         1       1
31            32         1       1
52            53         1       1
..           ...       ...     ...
856          857         1       1
862          863         1       1
871          872         1       1
879          880         1       1
887          888         1       1

                                                  Name     Sex   Age  SibSp
\
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
11                            Bonnell, Miss. Elizabeth  female  58.0      0
31      Spencer, Mrs. William Augustus (Marie Eugenie)  female   NaN      1
52            Harper, Mrs. Henry Sleeper (Myna Haxtun)  female  49.0      1
..                                                 ...     ...   ...    ...
856         Wick, Mrs. George Dennick (Mary Hitchcock)  female  45.0      1
862  Swift, Mrs. Frederick Joel (Margaret Welles Ba...  female  48.0      0
871   Beckwith, Mrs. Richard Leonard (Sallie Monypeny)  female  47.0      1
879      Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)  female  56.0      0
887                        Graham, Miss. Margaret Edith  female  19.0      0

     Parch    Ticket      Fare Cabin Embarked
1        0  PC 17599   71.2833   C85        C
3        0    113803   53.1000  C123        S
11       0    113783   26.5500  C103        S
31       0  PC 17569  146.5208   B78        C
52       0  PC 17572   76.7292   D33        C
..     ...       ...       ...   ...      ...
856      1     36928  164.8667   NaN        S
862      0     17466   25.9292   D17        S
871      1     11751   52.5542   D35        S
879      1     11767   83.1583   C50        C
887      0    112053   30.0000   B42        S

[94 rows x 12 columns]
```

Combinando a "indexação boleana" com os filtros, e realizamos *queries* mais complexas, como por exemplo que passageiros com mais de 30 anos, sobreviveram.

```
print("--- Filtrando Passageiros que Sobreviveram e Têm Mais de 30 Anos ---"
)
titanic[(titanic['Survived'] == 1) & (titanic['Age'] > 30)]
```

```
--- Filtrando Passageiros que Sobreviveram e Têm Mais de 30 Anos ---

     PassengerId  Survived  Pclass  \
1              2         1       1
3              4         1       1
```

```
11            12          1       1
15            16          1       2
21            22          1       2
..           ...        ...     ...
857           858         1       1
862           863         1       1
865           866         1       2
871           872         1       1
879           880         1       1
```

```
                                                 Name     Sex   Age  SibSp
\
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
11                             Bonnell, Miss. Elizabeth  female  58.0      0
15                   Hewlett, Mrs. (Mary D Kingcome)   female  55.0      0
21                                Beesley, Mr. Lawrence    male  34.0      0
..                                                 ...     ...   ...    ...
857                              Daly, Mr. Peter Denis     male  51.0      0
862  Swift, Mrs. Frederick Joel (Margaret Welles Ba...  female  48.0      0
865                            Bystrom, Mrs. (Karolina)  female  42.0      0
871   Beckwith, Mrs. Richard Leonard (Sallie Monypeny)  female  47.0      1
879      Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)  female  56.0      0
```

```
     Parch    Ticket     Fare Cabin Embarked
1        0  PC 17599  71.2833   C85        C
3        0    113803  53.1000  C123        S
11       0    113783  26.5500  C103        S
15       0    248706  16.0000   NaN        S
21       0    248698  13.0000   D56        S
..     ...       ...      ...   ...      ...
857      0    113055  26.5500   E17        S
862      0     17466  25.9292   D17        S
865      0    236852  13.0000   NaN        S
871      1     11751  52.5542   D35        S
879      1     11767  83.1583   C50        C
[124 rows x 12 columns]
```

Agregação

Group by

O método **groupby** é uma função fundamental na análise exploratória. Ele segue um conceito conhecido como **Split-Apply-Combine** (Dividir-Aplicar-Combinar).

- **Split** (Dividir): Os dados são particionados em grupos com base em chaves (ex: dividir os passageiros por Pclass).
- **Apply** (Aplicar): Uma função é aplicada a cada grupo independentemente (ex: calcular a média de Survived).
- **Combine** (Combinar): Os resultados são fundidos novamente numa única estrutura de dados.

Relembrar: Eixos (0 e 1). Muitas funções (como drop, mean, sum) exigem um argumento axis (eixo).

- **Axis 0** (Índice/Linhas): As operações movem-se para baixo ao longo do eixo vertical. titanic.mean(axis=0) calcula a média de cada coluna. É a direção do índice.
- **Axis 1** (Colunas): As operações movem-se através do eixo horizontal. titanic.drop('col_name', axis=1) remove uma coluna.

Com o `group by`, de forma simples e rápida conseguimos entender correlações, como por exemplo a sobrevivência por classe.

```
sobrevivencia_classe = titanic.groupby('Pclass')[['Survived']].mean()
display(sobrevivencia_classe)
```

```
        Survived
Pclass
1       0.629630
2       0.472826
3       0.242363
```

Ao analisarmos o output, descobrimos uma estratificação clara:

- Classe 1: ~63% de sobrevivência
- Classe 2: ~47% de sobrevivência

- Classe 3: ~24% de sobrevivência

Esta simples linha de código Pandas revela a desigualdade, que podemos classificar de social, confirmando que o estatuto socioeconómico foi um determinante na probabilidade de vida ou morte.

Podemos usar o **group by** associando o **.mean**, e saber por exemplo a média de idades dos passageiros, por género.

```
print("--- Agrupando por Sexo e Calculando a Média de Idade ---")
titanic.groupby('Sex')['Age'].mean()
```

```
--- Agrupando por Sexo e Calculando a Média de Idade ---

Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

Agrupamento Multi-Nível

Podemos agrupar por múltiplas colunas para obter outputs mais complexos.

Por exemplo agrupar por Classe e Género, isto vai revelar que, embora os passageiros da 1ª Classe tenham sobrevivido a taxas elevadas, as Mulheres da 1ª Classe tiveram uma taxa de sobrevivência de quase 97%, enquanto os Homens da 3ª Classe tiveram a mais baixa de todas.

Este cruzamento de variáveis demonstra como o Pandas facilita a análise multivariável.

```
titanic.groupby(['Pclass', 'Sex'])['Survived'].mean()
```

```
Pclass  Sex
1       female    0.968085
        male      0.368852
2       female    0.921053
        male      0.157407
3       female    0.500000
        male      0.135447
Name: Survived, dtype: float64
```

.agg(): O método **.agg()** permite aplicar múltiplas funções estatísticas simultaneamente.

Por exemplo, conseguimos ver não apenas a taxa de sobrevivência, mas também o número (count) de passageiros em cada classe, fornecendo contexto estatístico (tamanho da amostra) para as taxas calculadas.

```
# Do dataframe, vamos usar a coluna Pclass e Survived, e agrupar por Pclass, calculando a média e o count de Survived
titanic[['Pclass', 'Survived']].groupby('Pclass').agg(['mean', 'count'])
```

```
        Survived
            mean count
Pclass
1       0.629630   216
2       0.472826   184
3       0.242363   491
```

A análise revela uma relação (não linear), onde parece que estar sozinho (FamilySize=0) tem baixa sobrevivência, ter uma família pequena (1-3) aumenta a sobrevivência, mas famílias grandes (>4) têm taxas de sobrevivência muito baixas, provavelmente devido à dificuldade logística de reunir todos os membros durante a evacuação caótica.

Visualização gráfica

Embora o Pandas seja uma ferramenta de manipulação, ele integra-se diretamente, por exemplo, com o Matplotlib, permitindo a visualização rápida de estruturas de dados. Esta capacidade de "plotagem rápida" é essencial para a Análise Exploratória de Dados (EDA).

Análise Univariada

- **Histogramas**: titanic['Age'].plot(kind='hist') ajuda a visualizar a distribuição das idades. No Titanic, isto revela tipicamente uma distribuição bimodal, com picos para bebés/crianças pequenas e jovens adultos.

```
# 1. Histograma da distribuição das idades dos passageiros do Titanic
import matplotlib.pyplot as plt
titanic['Age'].plot(kind='hist', bins=30, color='skyblue', edgecolor='black')
plt.xlabel('Idade')
```

```
plt.ylabel('Frequência')
plt.title('Distribuição das Idades dos Passageiros do Titanic')
plt.show()
```

Nota: no Jupiter usamos este mesmo código mas com o dataframe titanic_clean, propositalmente para poder ser comparado.

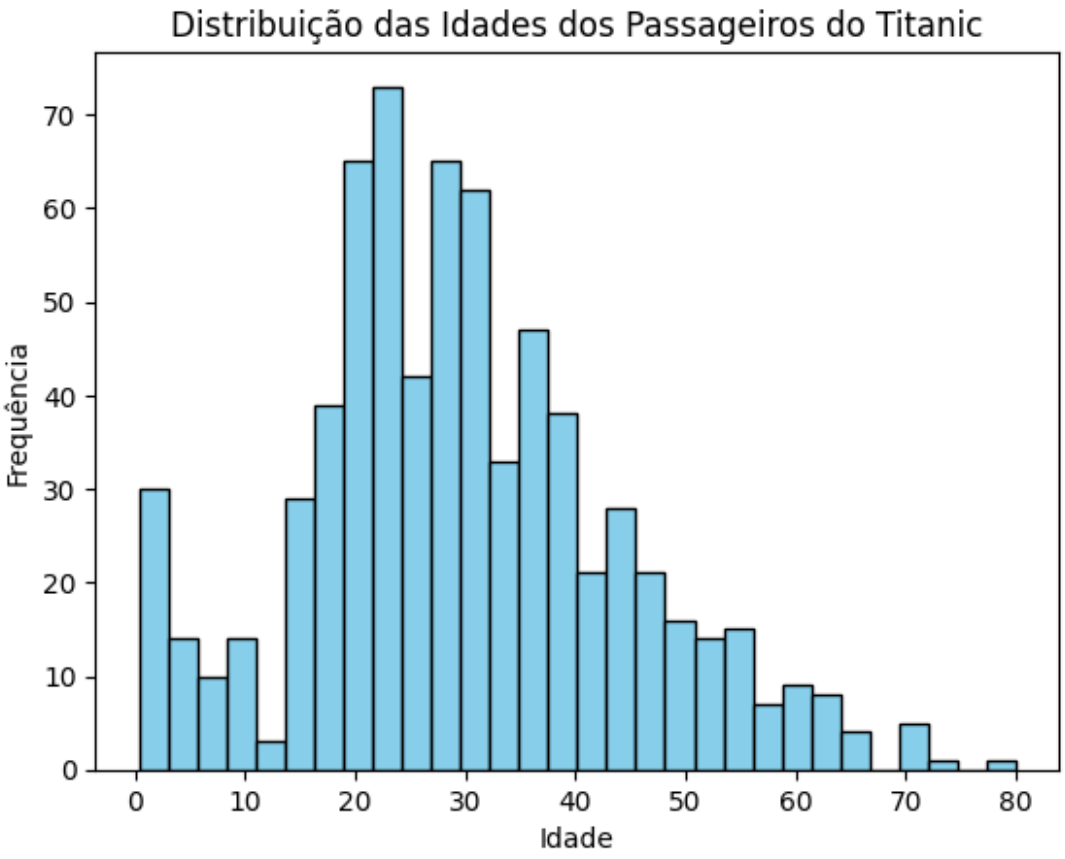

Figura 4 Histograma

- **Value Counts**: titanic['Embarked'].value_counts().plot(kind='bar') mostra graficamente que a vasta maioria dos passageiros partiu de Southampton (S), permitindo uma compreensão imediata da demografia geográfica da viagem.

Legenda do gráfico:

S → Southampton (Inglaterra)

C → Cherbourg (França)

Q → Queenstown (atual Cobh, Irlanda)

```
# 1. Value counts, criando um gráfico de barras para o número de passageiros por porto de embarque
titanic['Embarked'].value_counts().plot(kind='bar')
plt.xlabel('Porto de Embarque')
plt.ylabel('Número de Passageiros')
plt.title('Número de Passageiros por Porto de Embarque')
plt.show()
```

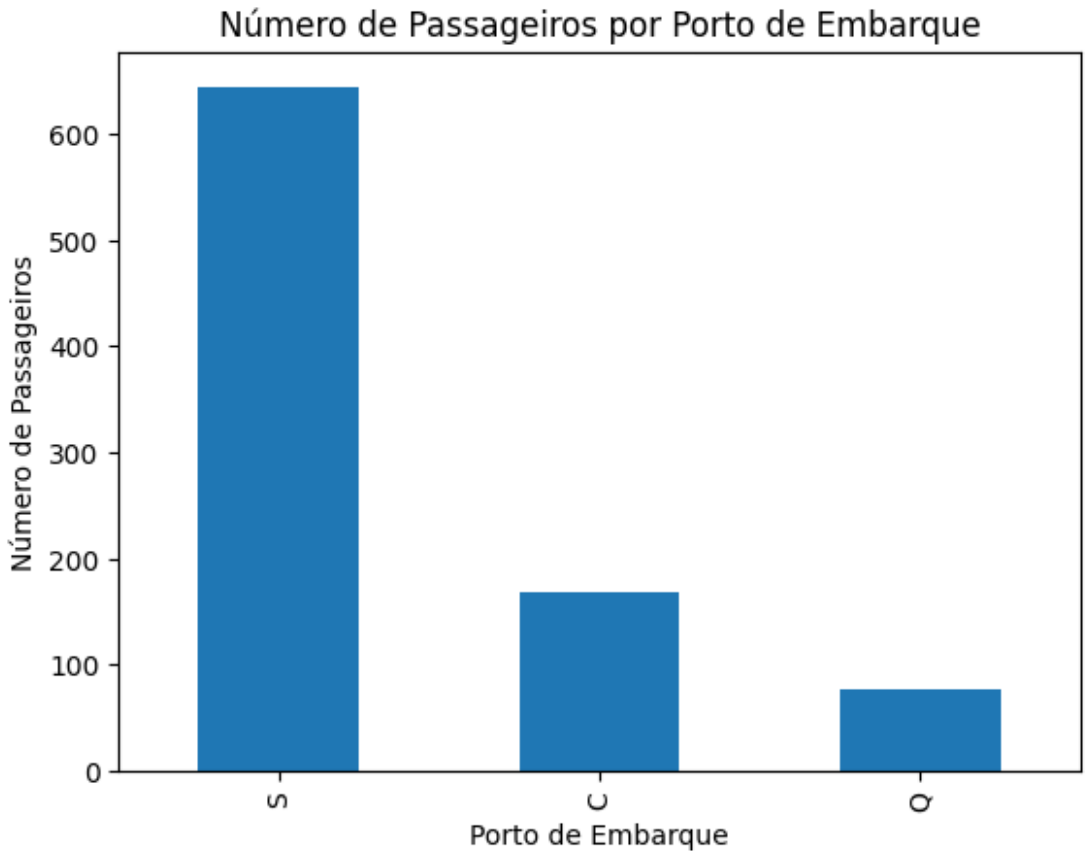

Figura 5 Gráfico de barras

Análise Bivariada

Box Plots: titanic.boxplot(column='Fare', by='Pclass') visualiza a dispersão dos preços dos bilhetes entre as classes. Isto destaca os "outliers" extremos — por exemplo, os poucos passageiros que pagaram mais de $500 por um bilhete, uma fortuna na época, ilustrando a disparidade económica extrema a bordo.

```
# 2. Box Plots, mostrando a dispersão dos preços dos bilhetes por classe
titanic.boxplot(column='Fare', by='Pclass')
plt.xlabel('Classe')
plt.ylabel('Preço do Bilhete (Fare)')
plt.title('Dispersão dos Preços dos Bilhetes por Classe')
plt.show()
```

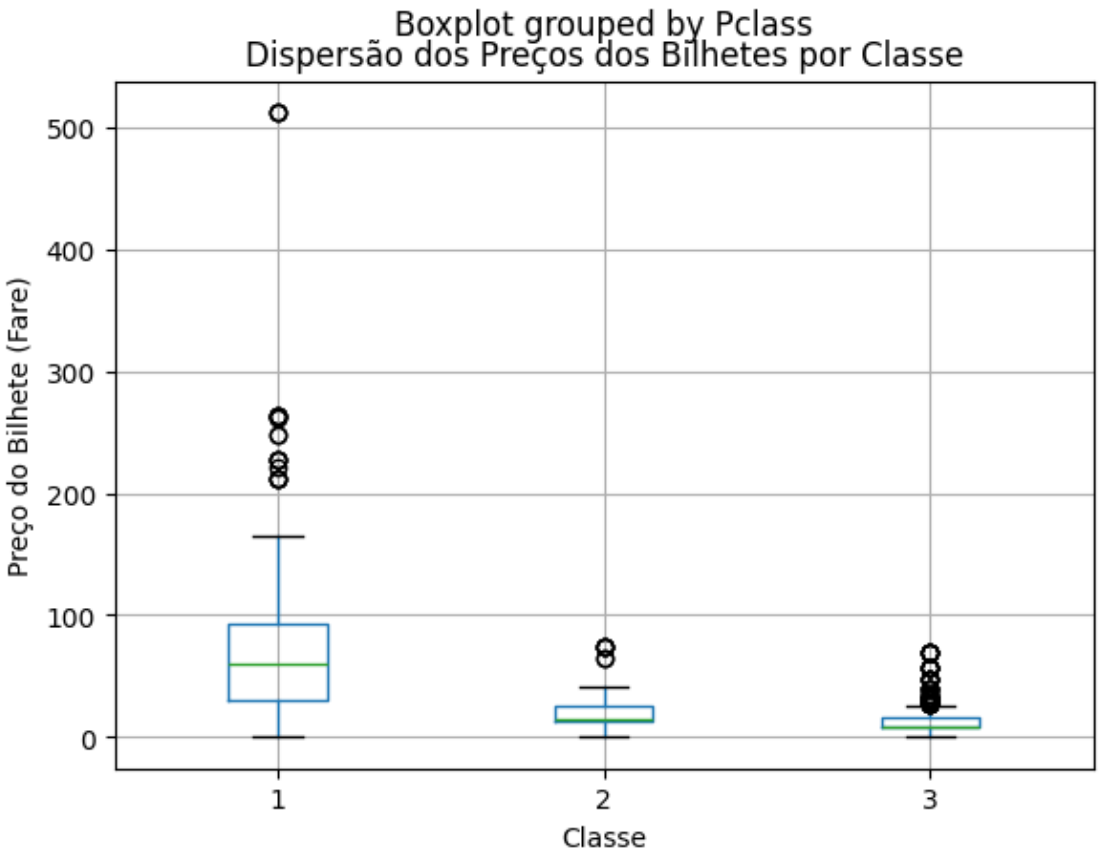

Figura 6 Box plot

Fusão e Relacionamento de Dados

A capacidade de combinar múltiplos datasets é onde o pandas substitui muitas das funções tradicionais de bancos de dados relacionais.

- **Merge**: A função `pd.merge()` implementa a lógica de álgebra relacional (joins). Diferente do SQL, o merge do pandas pode usar tanto colunas quanto índices como chaves de junção. Suporta junções do tipo:
    - inner (interseção de chaves),
    - outer (união de chaves),
    - left (preserva chaves da esquerda)
    - right.
- **Concat**: A função `pd.concat()` é utilizada para "colar" datasets ao longo de um eixo, seja empilhando linhas (axis=0) ou adicionando

colunas (axis=1). O alinhamento automático de índices é preservado aqui: se concatenarmos dois DataFrames horizontalmente, o pandas alinhará as linhas com base em seus índices, introduzindo valores nulos onde não houver correspondência, prevenindo o desalinhamento acidental de dados.

Não iremos explorar neste projeto de analise de dados do Titanic, mas deixamos no **Anexo B**, a forma como poderíamos implementar esse código em pandas.

Conclusão

A análise realizada demonstra inequivocamente os motivos do Pandas se ter tornado a ferramenta principal em ciência de dados na atualidade. Ele abstrai a complexidade da computação numérica (arrays C, alocação de memória, ...) e oferece uma interface de alto nível centrada na semântica dos dados.

Através do projeto Titanic, evidenciamos como a biblioteca gerência o ciclo completo de inteligência:

- Flexibilidade de Ingestão: Lendo dados remotos via HTTP com tratamento de protocolos de segurança.
- Eficácia e Simplicidade Estrutural: Gerenciando dados faltantes (NaN) e tipos heterogêneos dentro da mesma estrutura tabular.
- Poder Analítico: Os métodos como groupby e a indexação hierárquica permitiram encontrar rápidamente padrões complexos com poucas linhas de código.
- Interoperabilidade: A integração fluida com Seaborn e Matplotlib para visualização.

O futuro do Pandas aponta para uma integração ainda maior com tecnologias de "Big Data" através do projeto Apache Arrow (fonte: https://arrow.apache.org/), prometendo resolver atuais problemas de gestão de memória e permitir operações multithreaded nativas (capacidade de um programa dividir tarefas em "threads" (tarefas) que o sistema operacional gerência para correr em simultâneo). No entanto, para a vasta maioria das aplicações de análise de dados teórica e prática, a arquitetura atual baseada em DataFrames e Series, conforme detalhada neste relatório, permanecerá o ***gold standard*** por mais alguns anos.

Referências

- pandas (software) - Wikipedia
- pandas - Python Data Analysis Library
- 10 minutes to pandas
- Python pandas Tutorial: The Ultimate Guide for Beginners
- Pandas vs Polars in 2025: Choosing the Best Python Tool for Big Data
- Titanic Data Science Solutions

Anexos

Anexo A

Como mudar um valor de uma Series (funciona "in-place").

Como tentar mudar o tamanho e perceber que isso gera uma nova Series (novo objeto).

```
import pandas as pd
import numpy as np

# 1) Criar uma Series. Vemos claramente que permite uma logica similar a um d
icionário em Python, ao ter a possibilidade de definir o indice de cada eleme
nto da serie. Se não definesse atribuia automaticamente.
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print("Series original:\n", s) # imprimimos a serie que acabamos de criar pa
ra confirmar a sua composição.

# 2) Agora pretendemos mostrar que podemos mudar um valor (isto altera 'in-pl
ace')
s['b'] = 99  #usando o tal indice que existem sempre numa serie, vamos mudar
o indice "b" para 99.
print("\nDepois de alterar o valor em 'b':\n", s) # imprimimos a serie novam
ente para visualizar que assume o novo valor para o indice "b".

# 3) Tentar "mudar o tamanho" usando operações que NÃO modificam o objeto ori
ginal. Ex.: tentamos um drop, e este retorna uma NOVA Series, e se não guarda
r o resultado, "s" continua igual
s_sem_a = s.drop('a')  # cria uma nova Series em que fazemos drop do indice "
a"
print("\nResultado de drop('a') (nova Series):\n", s_sem_a)
print("\nA Series original continua igual:\n", s)

# 4) Para de facto mudar o tamanho (adicionar/remover), precisa criar/receber
um NOVO objeto Ex.: adicionar um elemento via CONCAT (gera uma nova Series)
novo_elemento = pd.Series([40], index=['d'])
s_expandida = pd.concat([s, novo_elemento])
print("\nSeries expandida (nova Series):\n", s_expandida)

# Se quiser que 's' fique com o novo tamanho, tem de reatribuir a nova serie
à variavel "s"
s = s_expandida
print("\nReatribuímos a 's' para ficar com o novo tamanho:\n", s)

# 5) Exemplo relacionado com NumPy (a base de uma Series):
arr = np.array([1, 2, 3])
print("\nNumPy array original:", arr)

# Alterar um valor é in-place
arr[1] = 99
```

```
print("NumPy array após alterar valor:", arr)

# "Redimensionar" o array não é in-place; np.append cria um NOVO array
arr2 = np.append(arr, 4)
print("Novo NumPy array após append (arr2):", arr2)
print("O array original continua:", arr)
```

```
Series original:
 a    10
b    20
c    30
dtype: int64

Depois de alterar o valor em 'b':
 a    10
b    99
c    30
dtype: int64

Resultado de drop('a') (nova Series):
 b    99
c    30
dtype: int64

A Series original continua igual:
 a    10
b    99
c    30
dtype: int64

Series expandida (nova Series):
 a    10
b    99
c    30
d    40
dtype: int64

Reatribuímos a 's' para ficar com o novo tamanho:
 a    10
b    99
c    30
d    40
dtype: int64

NumPy array original: [1 2 3]
NumPy array após alterar valor: [ 1 99  3]
Novo NumPy array após append (arr2): [ 1 99  3  4]
O array original continua: [ 1 99  3]
```

Anexo B

```
# MERGE

# DataFrame A
df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'nome': ['Ana', 'Bruno', 'Carlos']
})

# DataFrame B
df2 = pd.DataFrame({
    'id': [2, 3, 4],
    'cidade': ['Lisboa', 'Porto', 'Coimbra']
})

# Merge INNER (apenas ids comuns)
inner_merge = pd.merge(df1, df2, on='id', how='inner')

# Merge LEFT (todos do df1)
left_merge = pd.merge(df1, df2, on='id', how='left')

# Merge OUTER (todos os ids)
outer_merge = pd.merge(df1, df2, on='id', how='outer')

print(inner_merge)
print(left_merge)
print(outer_merge)
```

```
   id    nome  cidade
0   2   Bruno  Lisboa
1   3  Carlos   Porto
   id    nome  cidade
0   1     Ana     NaN
1   2   Bruno  Lisboa
2   3  Carlos   Porto
   id    nome   cidade
0   1     Ana      NaN
1   2   Bruno   Lisboa
2   3  Carlos    Porto
3   4     NaN  Coimbra
```

```
# CONCAT

# DataFrames com mesmo índice
df3 = pd.DataFrame({'idade': [25, 30, 35]}, index=['Ana', 'Bruno', 'Carlos'])
df4 = pd.DataFrame({'cidade': ['Lisboa', 'Porto', 'Coimbra']}, index=['Ana',
'Bruno', 'Carlos'])

# Concat horizontal (axis=1)
concat_horizontal = pd.concat([df3, df4], axis=1)

# Concat vertical (axis=0)
concat_vertical = pd.concat([df3, df3], axis=0)

print(concat_horizontal)
print(concat_vertical)
```

```
        idade   cidade
Ana        25   Lisboa
Bruno      30    Porto
Carlos     35  Coimbra
        idade
Ana        25
Bruno      30
Carlos     35
Ana        25
Bruno      30
Carlos     35
```