Programação para Ciência de Dados

Introdução ao Pandas - Series e DataFrames

Arthur Casals

21 de Outubro de 2025

Avisos

- Primeira entrega quinzenal: entregue!
- Recursos adicionais: notebooks e alternativas a NumPy/Pandas
- Presença: Moodle
- Interação via Zoom

Agenda

- Estruturas Pandas: Series e DataFrames
- Visualização e Inspeção de Dados
- Seleção e Indexação
- Filtragem e Ordenação

Recapitulação Aulas 03-04

O que já sabemos sobre NumPy:

- Arrays multidimensionais eficientes
- Operações vetorizadas e broadcasting
- Indexação avançada (boolean, fancy)
- Álgebra linear (dot, solve, inv, eig)
- Manipulação de shape (reshape, transpose)
- Concatenação e stacking
- Persistência (save, load, npz)

Hoje vamos aprender:

- Pandas: biblioteca para análise de dados
- Series e DataFrames: estruturas de alto nível
- Manipulação de dados tabulares

Bloco 1

Estruturas Pandas: Series e DataFrames

O que é Pandas?

Pandas (Python Data Analysis Library):

- Biblioteca para análise e manipulação de dados
- Construída sobre NumPy
- Estruturas de dados de alto nível
- Ferramentas poderosas para trabalhar com dados tabulares
- Integração com Matplotlib para visualização

Principais características:

- **Series:** Array 1D rotulado
- **DataFrame:** Tabela 2D (linhas e colunas rotuladas)
- **Indexação sofisticada:** Labels em vez de apenas posições
- **Operações de grupo:** GroupBy poderoso
- **Merge/Join:** Combinar datasets facilmente
- **Time Series:** Funcionalidades temporais avançadas

Por que Pandas?

NumPy:

- Arrays numéricos homogêneos
- Performance máxima
- Operações matemáticas
- Base para computação científica
- Indexação por posição

Pandas:

- Dados heterogêneos (tipos mistos)
- Operações de análise de dados
- Limpeza e transformação
- Dados tabulares (como Excel/SQL)
- Indexação por rótulos

Nota Importante

Pandas é ideal para análise exploratória de dados (EDA)

Quando usar cada um?

Use NumPy quando:

- Precisar de máxima performance
- Trabalhar com arrays numéricos puros
- Fazer álgebra linear
- Implementar algoritmos matemáticos
- Processar imagens ou sinais

Use Pandas quando:

- Trabalhar com dados tabulares (CSV, Excel, SQL)
- Precisar de rótulos nas linhas/colunas
- Fazer análise exploratória
- Limpar e transformar dados
- Agrupar, agregar e pivotar dados
- Trabalhar com séries temporais

Instalação e Importação

Instalação:

Bash

```bash
# Google Colab: ja vem instalado
# Local (via pip):
pip install pandas

# Verificar versao:
pip show pandas

```

Instalação e Importação

Importação (convenção universal):

Python

```python
import pandas as pd
import numpy as np  # Frequentemente usado junto

# Verificar versao
print(pd.__version__)  # 2.0.3 (ou similar)

```

Nota Importante

Sempre use `import pandas as pd` - é a convenção padrão

Pandas: Camada sobre NumPy

Arquitetura:

`Pandas (DataFrames, Series)`
↓ *construído sobre*
`NumPy (Arrays N-dimensionais)`
↓ *implementado em*
`C/Cython`

Implicações:

- DataFrames internamente usam arrays NumPy
- Pandas herda a performance do NumPy
- Conversão entre Pandas e NumPy é trivial
- Muitas operações vetorizadas disponíveis
- Integração perfeita com ecossistema científico

Series: Array 1D Rotulado

O que é uma Series?

- Array unidimensional com **índice**
- Como um dicionário (chave $\rightarrow$ valor)
- Como uma coluna de planilha
- Pode conter qualquer tipo de dados

Componentes:

- **Values:** Os dados propriamente ditos (array NumPy)
- **Index:** Rótulos para cada valor
- **Name:** Nome da Series (opcional)
- **Dtype:** Tipo de dados

Nota Importante

Series = Array NumPy + Index

Criando Series: A partir de Lista

Python

```python
# Series simples (indice automatico)
s = pd.Series([10, 20, 30, 40, 50])
print(s)
# 0    10
# 1    20
# 2    30
# 3    40
# 4    50
# dtype: int64

print(type(s))  # <class 'pandas.core.series.Series'>

```

Criando Series: Com Índice Customizado

Python

```python
# Series com indice customizado
s = pd.Series([10, 20, 30, 40, 50],
              index=['a', 'b', 'c', 'd', 'e'])
print(s)
# a    10
# b    20
# c    30
# d    40
# e    50
# dtype: int64

```

Criando Series: A partir de Dicionário

</> Python

```python
# Dicionario -> Series
dados = {
    'Ana': 25,
    'Bruno': 30,
    'Carlos': 35,
    'Diana': 28
}
s = pd.Series(dados)
print(s)
# Ana       25
# Bruno     30
# Carlos    35
# Diana     28
# dtype: int64

```

Criando Series: Com Nome

Python

```python
s = pd.Series([25, 30, 35, 28],
              index=['Ana', 'Bruno', 'Carlos', 'Diana'],
              name='Idade')
print(s)
# Ana        25
# Bruno      30
# Carlos     35
# Diana      28
# Name: Idade, dtype: int64

print(s.name)  # 'Idade'

```

Parâmetros de Series

Python

```python
pd.Series(
    data=None,  # Lista, tupla, Array Numpy, etc.
    index=None, # Rótulo das linhas - IMUTÁVEL INDIVIDUALMENTE
    dtype=None, # Tipo de dados (um só)
    name=None, # Nome vira rótulo da coluna em um DF
    copy=None, # Copy-on-write (None: cópia preguiçosa)
    fastpath=False # Pula validação redundante (interno, instável)
)
```

Atributos de Series

</> Python

```python
s = pd.Series([10, 20, 30, 40],
              index=['a', 'b', 'c', 'd'],
              name='valores')

# Acessar valores (array NumPy)
print(s.values)  # array([10, 20, 30, 40])
# Acessar indice
print(s.index)   # Index(['a', 'b', 'c', 'd'])
# Tipo de dados
print(s.dtype)   # int64 - Não é o tipo do índice!
# Tamanho
print(s.shape)   # (4,)
print(len(s))    # 4
```

Indexação em Series: Por Posição

Python

```python
s = pd.Series([10, 20, 30, 40, 50],
              index=['a', 'b', 'c', 'd', 'e'])

# Indexacao por posicao (como NumPy)
print(s[0])      # 10
print(s[-1])     # 50
print(s[1:3])    # Series com 'b' e 'c'

# Slicing
print(s[:3])     # Primeiros 3 elementos

```

Indexação em Series: Por Rótulo

Python

```python
s = pd.Series([10, 20, 30, 40, 50],
              index=['a', 'b', 'c', 'd', 'e'])
# Indexacao por rotulo (uma das principais vantagens do Pandas)
print(s['a'])        # 10
print(s['c'])        # 30
# Multiplos rotulos
print(s[['a', 'c', 'e']])
# a    10
# c    30
# e    50
```

Operações com Series

Python

```python
s = pd.Series([10, 20, 30, 40, 50])

# Operacoes aritmeticas (vetorizadas)
print(s + 10)        # Adiciona 10 a cada elemento
print(s * 2)         # Multiplica por 2
print(s ** 2)        # Eleva ao quadrado

# Funcoes estatisticas
print(s.mean())      # 30.0
print(s.sum())       # 150
print(s.max())       # 50
print(s.std())       # ~15.81

```

Operações entre Series

Python

```python
s1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
s2 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
# Operacoes elemento a elemento
print(s1 + s2)
# a    11
# b    22
# c    33

# Alinhamento automatico por indice!
s3 = pd.Series([1, 2], index=['a', 'c'])
print(s1 + s3)
# a    11.0
# b     NaN
# c    33.0
```

DataFrame: Tabela 2D

O que é um DataFrame?

- Estrutura tabular bidimensional
- Como uma planilha do Excel
- Como uma tabela SQL
- Coleção de Series (cada coluna é uma Series)

Componentes:

- **Columns:** Nomes das colunas (Index)
- **Index:** Rótulos das linhas (Index)
- **Values:** Dados (array NumPy 2D)
- **Dtypes:** Tipo de cada coluna

Nota Importante

DataFrame = Dicionário de Series (colunas)

Anatomia de um DataFrame

</> Python

```python
import pandas as pd

df = pd.DataFrame({
    'Nome': ['Ana', 'Bruno', 'Carlos'],
    'Idade': [25, 30, 35],
    'Cidade': ['SP', 'RJ', 'BH']
})

print(df)
#      Nome  Idade Cidade
# 0     Ana     25     SP
# 1   Bruno     30     RJ
# 2  Carlos     35     BH

```

Criando DataFrames: A partir de Dicionário

</> Python

```python
# Dicionario de listas
dados = {
    'Nome': ['Ana', 'Bruno', 'Carlos', 'Diana'],
    'Idade': [25, 30, 35, 28],
    'Cidade': ['SP', 'RJ', 'BH', 'SP'],
    'Salario': [3000, 4500, 5500, 3800]
}

df = pd.DataFrame(dados)
print(df)

```

Criando DataFrames: Resultado

Python

```
#      Nome  Idade Cidade  Salario
# 0     Ana     25     SP     3000
# 1   Bruno     30     RJ     4500
# 2  Carlos     35     BH     5500
# 3   Diana     28     SP     3800
```

Criando DataFrames: A partir de Lista de Dicionários

Python

```python
# Lista de dicionarios (cada dict = uma linha)
dados = [
    {'Nome': 'Ana', 'Idade': 25, 'Cidade': 'SP'},
    {'Nome': 'Bruno', 'Idade': 30, 'Cidade': 'RJ'},
    {'Nome': 'Carlos', 'Idade': 35, 'Cidade': 'BH'}
]

df = pd.DataFrame(dados)
print(df)
#      Nome  Idade Cidade
# 0     Ana     25     SP
# 1   Bruno     30     RJ
# 2  Carlos     35     BH

```

Criando DataFrames: A partir de Array NumPy

</> Python

```python
import numpy as np

# Array NumPy
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

df = pd.DataFrame(arr,
                  columns=['A', 'B', 'C'],
                  index=['x', 'y', 'z'])
print(df)
#    A  B  C
# x  1  2  3
# y  4  5  6
# z  7  8  9
```

Criando DataFrames: Com Índice Customizado

</> Python

```python
dados = {
    'Nome': ['Ana', 'Bruno', 'Carlos'],
    'Idade': [25, 30, 35]
}

df = pd.DataFrame(dados, index=['emp001', 'emp002', 'emp003'])
print(df)
#           Nome  Idade
# emp001     Ana     25
# emp002   Bruno     30
# emp003  Carlos     35

```

Leitura de Arquivos: CSV

Forma mais comum de criar DataFrames:

</> Python

```python
# Ler CSV
df = pd.read_csv('titanic.csv') # extremamente flexível e poderoso

# Com opcoes
df = pd.read_csv('dados.csv',
                 sep=';',           # Separador
                 encoding='utf-8',  # Codificacao
                 index_col=0,       # Coluna como indice
                 na_values=['NA', '?'])  # Valores faltantes

```

Leitura de Arquivos: Outros Formatos

</> Python

```python
# Excel
df = pd.read_excel('dados.xlsx', sheet_name='Plan1')

# JSON
df = pd.read_json('dados.json')

# SQL
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql('SELECT * FROM tabela', conn)

# Clipboard (copiar do Excel e colar)
df = pd.read_clipboard()

```

Atributos do DataFrame

</> Python

```python
df = pd.read_csv('titanic.csv')
# Shape (linhas, colunas)
print(df.shape)  # (891, 12)

# Colunas
print(df.columns)  # Index(['PassengerId', 'Survived', ...])

# Indice
print(df.index)    # RangeIndex(start=0, stop=891, step=1)

# Tipos de dados
print(df.dtypes)

# Valores (array NumPy)
print(df.values)  # array 2D
```

Parâmetros do DataFrame - set_index()

</> Python

```python
DataFrame.set_index(
    keys,                    # str, Index, Series, list, etc.
    drop=True,               # Remove do DF a coluna que virou index
    append=False,            # Adiciona novo index aos já existentes
    inplace=False,           # False: retorna view
    verify_integrity=False   # Verifica índices duplicados
)
```

Nota Importante

Para mais detalhes: Recursos Adicionais

DataFrame vs NumPy Array

NumPy Array:

</> Python

```python
arr = np.array([[1, 2],
                [3, 4]])
# Indexacao por posicao
print(arr[0, 1])  # 2

# Sem rotulos
# Tipos homogeneos
```

DataFrame:

</> Python

```python
df = pd.DataFrame([[1, 2],
                   [3, 4]],
        columns=['A', 'B'])
# Indexacao por rotulo
print(df.loc[0, 'B'])  # 2

# Com rotulos
# Tipos heterogeneos
```

Conversão: Pandas ↔ NumPy

Python

```python
# DataFrame -> NumPy
df = pd.DataFrame([[1, 2], [3, 4]], columns=['A', 'B'])
arr = df.values  # ou df.to_numpy()
print(type(arr))  # <class 'numpy.ndarray'>

# NumPy -> DataFrame
arr = np.array([[1, 2], [3, 4]])
df = pd.DataFrame(arr, columns=['A', 'B'])
print(type(df))  # <class 'pandas.core.frame.DataFrame'>

# Series -> NumPy
s = pd.Series([1, 2, 3])
arr = s.values

```

Relação: DataFrame e Series

Python

```python
df = pd.DataFrame({
    'Nome': ['Ana', 'Bruno', 'Carlos'],
    'Idade': [25, 30, 35]
})

#DataFrame = Coleção de Series alinhadas pelo índice
# Cada coluna é uma Series
col = df['Nome']
print(type(col))  # <class 'pandas.core.series.Series'>

# Cada linha e uma Series
linha = df.loc[0]
print(type(linha))  # <class 'pandas.core.series.Series'>
```

Exemplo Prático: Carregar Titanic (arquivo local)

Python

```python
import pandas as pd

# Carregar dataset
df = pd.read_csv('titanic.csv')

# Primeiras impressoes
print(f"Linhas: {len(df)}")
print(f"Colunas: {len(df.columns)}")
print(f"Shape: {df.shape}")

# Tipos de dados presentes
print(df.dtypes.value_counts())

```

Exemplo Prático: Carregar Titanic (arquivo remoto)

</> Python

```python
import pandas as pd

# Carregar dataset
url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
df = pd.read_csv(url)

# Primeiras impressoes
print(f"Linhas: {len(df)}")
print(f"Colunas: {len(df.columns)}")
print(f"Shape: {df.shape}")

# Tipos de dados presentes
print(df.dtypes.value_counts())

```

Bloco 2

Visualização e Inspeção de Dados

Explorando DataFrames

Primeiros passos com dados novos:

1. **Visualizar:** Primeiras/últimas linhas
2. **Dimensões:** Quantas linhas e colunas?
3. **Tipos:** Quais tipos de dados?
4. **Valores faltantes:** Existem NaN?
5. **Estatísticas:** Resumo numérico
6. **Distribuições:** Como são os valores?

Funções principais:

- `head()`, `tail()`, `sample()`
- `info()`, `describe()`
- `value_counts()`, `unique()`

head() e tail(): Visualização Rápida

</> Python

```python
df = pd.read_csv('titanic.csv')

# Primeiras 5 linhas (padrao)
print(df.head())

# Primeiras 10 linhas
print(df.head(10))

# Ultimas 5 linhas
print(df.tail())

# Ultimas 3 linhas
print(df.tail(3))

```

head() e tail(): Visualização Rápida

Nota Importante

Sempre comece com `head()` para entender a estrutura

sample(): Amostra Aleatória

Python

```python
df = pd.read_csv('titanic.csv')

# 5 linhas aleatorias
print(df.sample(5))

# 10% dos dados
print(df.sample(frac=0.1))

# Com seed para reproducibilidade
print(df.sample(5, random_state=42))

# Amostragem com reposicao
print(df.sample(10, replace=True))

```

info(): Informações Gerais

Python

```python
df = pd.read_csv('titanic.csv')
df.info()
# <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
# ...
```

info(): O que mostra

Informações fornecidas:

- Tipo da classe (DataFrame)
- Tipo e tamanho do índice
- Número total de colunas
- Para cada coluna:
  - Nome
  - Número de valores não-nulos
  - Tipo de dados (dtype)
- Uso de memória

Nota Importante

`info()` é essencial para identificar valores faltantes rapidamente

describe(): Estatísticas Descritivas

Python

```python
df = pd.read_csv('titanic.csv')

# Estatisticas para colunas numericas
print(df.describe())
#        PassengerId    Survived      Pclass         Age
# count   891.000000  891.000000  891.000000  714.000000
# mean    446.000000    0.383838    2.308642   29.699118
# std     257.353842    0.486592    0.836071   14.526497
# min       1.000000    0.000000    1.000000    0.420000
# 25%     223.500000    0.000000    2.000000   20.125000
# 50%     446.000000    0.000000    3.000000   28.000000
# 75%     668.500000    1.000000    3.000000   38.000000
# max     891.000000    1.000000    3.000000   80.000000

```

describe(): Incluir Todas as Colunas

Python

```python
# Incluir colunas categoricas
print(df.describe(include='all'))

# Apenas colunas categoricas
print(df.describe(include='object'))

# Apenas colunas numericas (padrao)
print(df.describe(include='number'))

# Estatisticas especificas
print(df[['Age', 'Fare']].describe())

```

describe(): Interpretação

Para colunas numéricas:

- **count:** Número de valores não-nulos
- **mean:** Média
- **std:** Desvio padrão
- **min/max:** Valores mínimo e máximo
- **25%/50%/75%:** Percentis (quartis)

Para colunas categóricas:

- **count:** Número de valores não-nulos
- **unique:** Número de valores únicos
- **top:** Valor mais frequente
- **freq:** Frequência do valor mais frequente

value_counts(): Distribuição de Valores

</> Python

```python
df = pd.read_csv('titanic.csv')
# Contar valores unicos
print(df['Survived'].value_counts())
# 0    549
# 1    342
# Name: Survived, dtype: int64

# Com proporcoes (percentuais)
print(df['Survived'].value_counts(normalize=True))
# 0    0.616162
# 1    0.383838

# Incluir valores faltantes
print(df['Age'].value_counts(dropna=False))
```

value_counts(): Aplicações

Python

```python
# Distribuicao por classe
print(df['Pclass'].value_counts().sort_index())
# 1    216
# 2    184
# 3    491

# Distribuicao por sexo
print(df['Sex'].value_counts())
# male      577
# female    314

# Top 5 nomes mais comuns (se houver repetidos)
print(df['Name'].value_counts().head(5))

```

unique() e nunique()

</> Python

```python
df = pd.read_csv('titanic.csv')
# Valores unicos (array)
print(df['Pclass'].unique())
# array([3, 1, 2])

# Numero de valores unicos
print(df['Pclass'].nunique())
# 3

# Quantos nomes unicos?
print(df['Name'].nunique())
# 891 (todos diferentes)

# Quantas idades unicas?
print(df['Age'].nunique())
```

Atributos: shape, columns, index

</> Python

```python
df = pd.read_csv('titanic.csv')

# Shape (linhas, colunas)
print(df.shape)  # (891, 12)
print(f"Linhas: {df.shape[0]}")
print(f"Colunas: {df.shape[1]}")

# Nomes das colunas
print(df.columns)
# Index(['PassengerId', 'Survived', 'Pclass', ...])

# Indice (linhas)
print(df.index)
# RangeIndex(start=0, stop=891, step=1)

```

dtypes: Tipos de Dados

Python

```python
df = pd.read_csv('titanic.csv')
# Tipos de cada coluna
print(df.dtypes)
# PassengerId      int64
# Survived         int64
# Pclass           int64
# Name            object
# Sex             object
# Age            float64
# ...
# Resumo dos tipos
print(df.dtypes.value_counts())
# int64      5
# float64    2
# object     5
```

Tipos de Dados no Pandas

Principais dtypes:

Dtype	Descrição
`int64`	Inteiros
`float64`	Ponto flutuante
`object`	Strings, tipos mistos
`bool`	Booleanos (True/False)
`datetime64`	Datas e horários
`timedelta64`	Diferenças de tempo
`category`	Categóricas (economia de memória)

Nota Importante

`object` geralmente significa strings, mas pode conter qualquer tipo Python

Verificar Valores Faltantes

Python

```python
df = pd.read_csv('titanic.csv')
# Valores faltantes por coluna
print(df.isnull().sum())
# PassengerId      0
# Survived         0
# Pclass           0
# Name             0
# Sex              0
# Age            177
# Cabin          687
# Embarked         2
# ...

# Percentual de valores faltantes
print((df.isnull().sum() / len(df)) * 100)
```

Verificar Valores Faltantes: Total

Python

```python
# Total de valores faltantes no DataFrame
print(df.isnull().sum().sum())

# Linhas com pelo menos um valor faltante
print(df.isnull().any(axis=1).sum())

# Linhas sem valores faltantes
print(df.notnull().all(axis=1).sum())

# Apenas linhas completas (sem NaN)
df_completo = df.dropna()
print(f"Linhas completas: {len(df_completo)}")

```

Renomear Colunas: rename()

</> Python

```python
df = pd.read_csv('titanic.csv')
# Renomear colunas especificas
df_renamed = df.rename(columns={
    'PassengerId': 'ID',
    'Pclass': 'Classe',
    'Name': 'Nome'
})

# Renomear in-place (modifica o original)
df.rename(columns={'Sex': 'Sexo'}, inplace=True)

# Aplicar funcao a todos os nomes
df.columns = df.columns.str.lower()  # Minusculas
df.columns = df.columns.str.replace(' ', '_')
```

Renomear Índice: set_index()

Python

```python
df = pd.read_csv('titanic.csv')
# Definir coluna como indice
df_indexed = df.set_index('PassengerId')
print(df_indexed.head())
#              Survived  Pclass  Name  ...
# PassengerId                            ...
# 1                   0       3  ...   ...
# 2                   1       1  ...   ...

# Resetar indice
df_reset = df_indexed.reset_index()

# set_index in-place
df.set_index('Name', inplace=True)
```

Ordenar Colunas

</> Python

```python
df = pd.read_csv('titanic.csv')

# Reordenar colunas manualmente
colunas_ordem = ['Name', 'Age', 'Sex', 'Survived']
df_reord = df[colunas_ordem]

# Ordenar alfabeticamente
df_sorted = df[sorted(df.columns)]

# Mover coluna para o inicio
cols = df.columns.tolist()
cols = ['Survived'] + [c for c in cols if c != 'Survived']
df = df[cols]

```

Exemplo Prático: Exploração Inicial

Python

```python
import pandas as pd

# Carregar dados
df = pd.read_csv('titanic.csv')

# Pipeline de exploracao inicial
print("=== DIMENSOES ===")
print(f"Shape: {df.shape}")

print("\n=== PRIMEIRAS LINHAS ===")
print(df.head(3))

print("\n=== INFORMACOES ===")
df.info()

```

Exemplo Prático: Exploração (cont.)

Python

```python
print("\n=== ESTATISTICAS ===")
print(df.describe())

print("\n=== VALORES FALTANTES ===")
print(df.isnull().sum())

print("\n=== DISTRIBUICAO DE SOBREVIVENTES ===")
print(df['Survived'].value_counts())

print("\n=== DISTRIBUICAO POR CLASSE ===")
print(df['Pclass'].value_counts().sort_index())

```

Bloco 3

Seleção e Indexação

Formas de Selecionar Dados

Pandas oferece múltiplas formas:

- **Colchetes [ ]:** Seleção de colunas
- **.loc[ ]:** Indexação por **rótulo**
- **.iloc[ ]:** Indexação por **posição**
- **.at[ ]:** Acesso rápido a valor único (rótulo)
- **.iat[ ]:** Acesso rápido a valor único (posição)

Quando usar cada um?

para colunas

- .loc para seleção por nomes
- .iloc para seleção por posição numérica
- .at/.iat para acesso a elemento único (mais rápido)

Seleção de Colunas: Uma Coluna

Python

```python
df = pd.read_csv('titanic.csv')

# Selecionar uma coluna (retorna Series)
nomes = df['Name']
print(type(nomes))  # <class 'pandas.core.series.Series'>

# Alternativa (notacao de atributo)
# Funciona se nome nao tem espacos/caracteres especiais
idades = df.Age  # Mesmo que df['Age']

# Ver alguns valores
print(nomes.head())

```

Seleção de Colunas: Múltiplas Colunas

</> Python

```python
df = pd.read_csv('titanic.csv')

# Selecionar multiplas colunas (retorna DataFrame)
subset = df[['Name', 'Age', 'Sex']]
print(type(subset))  # <class 'pandas...DataFrame'>
print(subset.head())

# Ordem importa
subset2 = df[['Sex', 'Name', 'Age']]

# Todas exceto algumas
colunas_manter = [c for c in df.columns
                  if c not in ['PassengerId', 'Ticket']]
df_filtrado = df[colunas_manter]

```

Diferença: [ ] vs [[ ]]

</> Python

```python
df = pd.read_csv('titanic.csv')
# Uma coluna: retorna Series
series = df['Age']
print(type(series))  # Series
print(series.shape)  # (891,)

# Lista com uma coluna: retorna DataFrame
dataframe = df[['Age']]
print(type(dataframe))  # DataFrame
print(dataframe.shape)  # (891, 1)

# Diferenca visual
print(series.head(2))     # 0    22.0
                          # 1    38.0
print(dataframe.head(2)) # DataFrame formatado
```

loc vs iloc: Conceito

Diferença fundamental:

.loc[ ]:

- Indexação por **rótulo**
- Usa nomes de índice/colunas
- Inclusivo nos extremos
- `df.loc[0:5]` inclui linha 5

.iloc[ ]:

- Indexação por **posição**
- Usa números inteiros
- Exclusivo no extremo direito
- `df.iloc[0:5]` exclui linha 5

Nota Importante

loc = location (rótulo), **iloc** = integer location (posição)

loc[ ]: Seleção por Rótulo

Python

```python
df = pd.read_csv('titanic.csv')

# Uma linha por indice
print(df.loc[0])  # Series com dados da linha 0

# Multiplas linhas
print(df.loc[0:5])  # Linhas 0, 1, 2, 3, 4, 5 (inclusivo!)

# Uma celula especifica
print(df.loc[0, 'Name'])

# Subconjunto (linhas e colunas)
subset = df.loc[0:5, ['Name', 'Age', 'Sex']]
print(subset)

```

loc[ ]: Slicing de Colunas

</> Python

```python
df = pd.read_csv('titanic.csv')

# Todas as linhas, algumas colunas
subset = df.loc[:, ['Name', 'Age', 'Sex']]

# Slice de colunas (por nome)
subset = df.loc[:, 'Name':'Age']  # Inclusivo!

# Primeiras 10 linhas, colunas especificas
subset = df.loc[:9, ['Survived', 'Pclass', 'Name']]

# Uma linha, todas as colunas
linha = df.loc[5, :]

```

iloc[ ]: Seleção por Posição

Python

```python
df = pd.read_csv('titanic.csv')

# Uma linha por posicao
print(df.iloc[0])  # Primeira linha

# Multiplas linhas
print(df.iloc[0:5])  # Linhas 0, 1, 2, 3, 4 (exclusivo!)

# Uma celula especifica (linha 0, coluna 3)
print(df.iloc[0, 3])

# Subconjunto (primeiras 5 linhas, primeiras 3 colunas)
subset = df.iloc[0:5, 0:3]
print(subset)

```

iloc[ ]: Indexação Negativa

Python

```python
df = pd.read_csv('titanic.csv')
# Ultima linha
print(df.iloc[-1])

# Ultimas 5 linhas
print(df.iloc[-5:])

# Penultima linha, ultimas 3 colunas
print(df.iloc[-2, -3:])

# Todas as linhas exceto a primeira
print(df.iloc[1:])

# Primeiras 100 linhas, exceto primeiras 2 colunas
subset = df.iloc[:100, 2:]
```

iloc[ ]: Listas de Posições

Python

```python
df = pd.read_csv('titanic.csv')

# Linhas especificas
linhas = df.iloc[[0, 10, 20, 30]]

# Colunas especificas (posicoes 1, 3, 5)
colunas = df.iloc[:, [1, 3, 5]]

# Combinacao
subset = df.iloc[[0, 5, 10], [1, 3, 5]]

# Linhas pares
pares = df.iloc[::2]  # step=2

```

loc vs iloc: Comparação Direta

Python

```python
df = pd.read_csv('titanic.csv')

# loc: por rotulo (inclusivo)
a = df.loc[0:5, 'Name']
print(len(a))  # 6 elementos (0, 1, 2, 3, 4, 5)

# iloc: por posicao (exclusivo)
b = df.iloc[0:5, 3]  # coluna na posicao 3
print(len(b))  # 5 elementos (0, 1, 2, 3, 4)

# loc com indice customizado
df_custom = df.set_index('PassengerId')
print(df_custom.loc[1])  # Passageiro ID 1
print(df_custom.iloc[0])  # Primeira linha (pode ser != 1)

```

at[ ] e iat[ ]: Acesso Rápido

Python

```python
df = pd.read_csv('titanic.csv')

# at: acesso por rotulo (mais rapido que loc)
valor = df.at[0, 'Name']
print(valor)  # Nome do passageiro 0

# iat: acesso por posicao (mais rapido que iloc)
valor = df.iat[0, 3]
print(valor)  # Valor na linha 0, coluna 3

# Modificar valor
df.at[0, 'Age'] = 25
df.iat[0, 5] = 30  # Se Age estiver na posicao 5

```

Quando Usar Cada Método

Método	Uso	Retorna
`df['col']`	Uma coluna	Series
`df[['c1','c2']]`	Múltiplas colunas	DataFrame
`df.loc[]`	Rótulos (nomes)	Series/DataFrame
`df.iloc[]`	Posições (int)	Series/DataFrame
`df.at[]`	Um valor (rótulo)	Escalar
`df.iat[]`	Um valor (posição)	Escalar

Regra geral:

- Colunas: `df['col']` ou `df[['col1', 'col2']]`
- Linhas e colunas: `.loc` ou `.iloc`
- Um valor: `.at` ou `.iat`

Modificando Valores

Python

```python
df = pd.read_csv('titanic.csv')

# Modificar uma coluna inteira
df['Age'] = df['Age'] + 1

# Modificar uma celula especifica
df.loc[0, 'Age'] = 25
df.iloc[0, 5] = 30

# Modificar multiplas celulas
df.loc[0:5, 'Age'] = 30

# Criar nova coluna
df['AgeGroup'] = 'Adult'

```

Localizando Valores Aninhados

Python

```python
df = pd.read_csv('titanic.csv')

# Potencial problema (chained indexing)
# df['Age'][0] = 25  # Pode dar warning!

# Forma correta
df.loc[0, 'Age'] = 25

# Criar copia explicita se necessario
subset = df[['Name', 'Age']].copy()
subset.loc[0, 'Age'] = 25  # OK, é uma copia

```

! Chained Indexing

⚠ Atenção

Evite **chained indexing**: `df['col'][row]`. Use `df.loc[row, 'col']`

Exemplo Prático: Extrair Subset

Python

```python
df = pd.read_csv('titanic.csv')

# Extrair informacoes basicas dos primeiros 100
subset = df.loc[:99, ['Name', 'Age', 'Sex', 'Survived']]
print(subset.head())

# Extrair ultimos 50 passageiros, todas as colunas
ultimos = df.iloc[-50:]

# Extrair colunas especificas por posicao
# Colunas 1, 2, 3, 5 (Survived, Pclass, Name, Sex)
especificas = df.iloc[:, [1, 2, 3, 5]]

```

Bloco 4

Filtragem e Ordenação

Filtragem de Dados

Por que filtrar?

- Análise de subconjuntos específicos
- Remover outliers ou dados inválidos
- Focar em categorias de interesse
- Preparar dados para modelagem

Métodos principais:

- **Boolean indexing:** Máscaras booleanas
- **Operadores lógicos:** `&`, `|`, `~`
- **query():** Sintaxe SQL-like
- **isin():** Verificar pertencimento

Boolean Indexing: Conceito

Python

```python
df = pd.read_csv('titanic.csv')

# Criar mascara booleana
mascara = df['Age'] > 30
print(type(mascara))  # Series de booleanos
print(mascara.head())
# 0    False
# 1     True
# 2    False
# 3     True
# 4    False
# Aplicar mascara
maiores_30 = df[mascara]
print(len(maiores_30))
```

Boolean Indexing: Forma Compacta

Python

```python
df = pd.read_csv('titanic.csv')

# Forma compacta (mais comum)
maiores_30 = df[df['Age'] > 30]

# Outras comparacoes
primeira_classe = df[df['Pclass'] == 1]
mulheres = df[df['Sex'] == 'female']
sobreviventes = df[df['Survived'] == 1]

# Combinacao comum
mulheres_sobreviventes = df[(df['Sex'] == 'female') &
                            (df['Survived'] == 1)]

```

Operadores de Comparação

Python

```python
df = pd.read_csv('titanic.csv')
# Maior que
df[df['Age'] > 30]

# Menor ou igual
df[df['Fare'] <= 20]

# Igual
df[df['Pclass'] == 1]

# Diferente
df[df['Sex'] != 'male']

# Entre valores
df[(df['Age'] >= 18) & (df['Age'] <= 35)]
```

Operadores Lógicos: AND (&)

Python

```python
df = pd.read_csv('titanic.csv')

# AND: ambas condicoes devem ser True
mulheres_1classe = df[(df['Sex'] == 'female') &
                      (df['Pclass'] == 1)]
print(len(mulheres_1classe))

# Multiplas condicoes
filtro = df[(df['Age'] > 30) &
            (df['Pclass'] == 1) &
            (df['Survived'] == 1)]

```

Operadores Lógicos: AND (&) (cont.)

Python

```python
# Equivalente (mais legivel para muitas condicoes)
cond1 = df['Age'] > 30
cond2 = df['Pclass'] == 1
cond3 = df['Survived'] == 1
filtro = df[cond1 & cond2 & cond3]

```

Operadores Lógicos: OR (|)

Python

```python
df = pd.read_csv('titanic.csv')

# OR: pelo menos uma condicao deve ser True
primeira_ou_segunda = df[(df['Pclass'] == 1) |
                         (df['Pclass'] == 2)]
print(len(primeira_ou_segunda))

# Criancas ou idosos
extremos = df[(df['Age'] < 12) | (df['Age'] > 60)]

# Combinacao AND e OR
filtro = df[((df['Sex'] == 'female') | (df['Age'] < 18)) &
            (df['Survived'] == 1)]

```

Operadores Lógicos: OR (|)

⚠ Atenção

Use `&` e `|`, não `and` e `or`! Sempre use parênteses!

Operadores Lógicos: NOT (~)

Python

```python
df = pd.read_csv('titanic.csv')
# NOT: inverte a condicao
nao_sobreviventes = df[~(df['Survived'] == 1)]
# Equivalente a:
nao_sobreviventes = df[df['Survived'] == 0]

# Nao-criancas (idade >= 18)
adultos = df[~(df['Age'] < 18)]

# Excluir primeira classe
nao_primeira = df[~(df['Pclass'] == 1)]

# Combinacao complexa
filtro = df[~((df['Age'] < 18) | (df['Age'] > 60))]
# Pessoas entre 18 e 60 anos
```

isin(): Pertencimento à Lista

Python

```python
df = pd.read_csv('titanic.csv')
# Verificar se valor esta em lista
portos = ['S', 'C']
embarque_filtrado = df[df['Embarked'].isin(portos)]

# Classes 1 ou 2
classes_altas = df[df['Pclass'].isin([1, 2])]

# NOT isin (nao esta na lista)
nao_Southampton = df[~df['Embarked'].isin(['S'])]

# Nomes especificos
nomes = ['Smith', 'Johnson', 'Williams']
pessoas = df[df['Name'].str.contains('|'.join(nomes))]
```

query(): Sintaxe SQL-like

</> Python

```python
df = pd.read_csv('titanic.csv')
# Sintaxe query (string)
resultado = df.query('Age > 30')

# Multiplas condicoes
resultado = df.query('Age > 30 and Pclass == 1')
# OR
resultado = df.query('Pclass == 1 or Pclass == 2')
# Variavel externa
idade_minima = 30
resultado = df.query('Age > @idade_minima')

# Nomes com espacos (usar backticks)
# df.query('`Passenger Class` == 1')

```

query(): Vantagens

Vantagens do query():

- Sintaxe mais legível para queries complexas
- Não precisa repetir `df['coluna']`
- Usa `and`/`or` em vez de `&`/`|`
- Não precisa de parênteses extras

query(): Vantagens

Python

```python
# Comparacao: boolean indexing vs query
# Boolean indexing (verboso)
filtro1 = df[(df['Age'] > 30) & (df['Pclass'] == 1)]

# Query (conciso)
filtro2 = df.query('Age > 30 and Pclass == 1')

# Resultados identicos
print(filtro1.equals(filtro2))  # True

```

Filtragem com Strings

Python

```python
df = pd.read_csv('titanic.csv')
# Contem substring
mrs = df[df['Name'].str.contains('Mrs')]
# Comeca com
comeca_a = df[df['Name'].str.startswith('A')]
# Termina com
termina_son = df[df['Name'].str.endswith('son')]
# Case insensitive
braund = df[df['Name'].str.contains('braund',
                                    case=False)]
# Regex
import re
tem_numero = df[df['Name'].str.contains(r'\d')]
```

Filtragem de Valores Faltantes

</> Python

```python
df = pd.read_csv('titanic.csv')
# Linhas com idade faltante
sem_idade = df[df['Age'].isnull()]
print(len(sem_idade))

# Linhas com idade presente
com_idade = df[df['Age'].notnull()]
# ou: df[~df['Age'].isnull()]

# Linhas sem nenhum valor faltante
completas = df[df.notnull().all(axis=1)]

# Linhas com pelo menos um valor faltante
com_nan = df[df.isnull().any(axis=1)]
```

between(): Valores em Range

Python

```python
df = pd.read_csv('titanic.csv')

# Idade entre 20 e 40 (inclusivo)
faixa = df[df['Age'].between(20, 40)]

# Equivalente a:
faixa = df[(df['Age'] >= 20) & (df['Age'] <= 40)]

# Excluir extremos (exclusive)
faixa = df[df['Age'].between(20, 40, inclusive='neither')]

# Tarifa entre valores
tarifas = df[df['Fare'].between(10, 50)]

```

Ordenação de Dados

Por que ordenar?

- Identificar valores extremos
- Preparar dados para análise sequencial
- Facilitar visualização
- Ranking de elementos

Métodos principais:

- `sort_values()`: Ordenar por valores de coluna(s)
- `sort_index()`: Ordenar pelo índice
- `nlargest()/nsmallest()`: Top N valores

sort_values(): Ordenar por Coluna

</> Python

```python
df = pd.read_csv('titanic.csv')

# Ordenar por idade (crescente)
df_ordenado = df.sort_values('Age')
print(df_ordenado.head())

# Ordenar por idade (decrescente)
df_ordenado = df.sort_values('Age', ascending=False)

# Ordenar in-place (modifica original)
df.sort_values('Age', inplace=True)

# Resetar indice apos ordenar
df_ordenado = df.sort_values('Age').reset_index(drop=True)
```

sort_values(): Múltiplas Colunas

Python

```python
df = pd.read_csv('titanic.csv')
# Ordenar por classe, depois por tarifa
df_ordenado = df.sort_values(['Pclass', 'Fare'])

# Direcoes diferentes para cada coluna
df_ordenado = df.sort_values(
    ['Pclass', 'Age'],
    ascending=[True, False]  # Classe crescente, Idade decrescente
)
# Exemplo: classe crescente, dentro de cada classe
# ordenar por tarifa decrescente
df_ordenado = df.sort_values(
    ['Pclass', 'Fare'],
    ascending=[True, False]
)
```

sort_values(): Tratando NaN

Python

```python
df = pd.read_csv('titanic.csv')

# Por padrao, NaN vai para o final
df_ordenado = df.sort_values('Age')

# Colocar NaN no inicio
df_ordenado = df.sort_values('Age', na_position='first')

# Remover NaN antes de ordenar
df_sem_nan = df.dropna(subset=['Age'])
df_ordenado = df_sem_nan.sort_values('Age')

# Ordenar apenas valores nao-nulos
df_ordenado = df[df['Age'].notnull()].sort_values('Age')

```

sort_index(): Ordenar por Índice

</> Python

```python
df = pd.read_csv('titanic.csv')

# Depois de filtros, indice pode estar desordenado
filtrado = df[df['Age'] > 30]
print(filtrado.index[:5])  # [1, 3, 6, 11, 15, ...]

# Ordenar pelo indice
df_ordenado = filtrado.sort_index()
print(df_ordenado.index[:5])  # [1, 3, 6, 11, 15, ...]

# Decrescente
df_ordenado = filtrado.sort_index(ascending=False)

# Para indice customizado (strings)
df_custom = df.set_index('Name').sort_index()

```

nlargest() e nsmallest(): Top N

Python

```python
df = pd.read_csv('titanic.csv')

# Top 10 tarifas mais caras
top_tarifas = df.nlargest(10, 'Fare')
print(top_tarifas[['Name', 'Fare']])

# 10 passageiros mais jovens
mais_jovens = df.nsmallest(10, 'Age')

# Top 5 por multiplas colunas
# (primeiro por Fare, depois por Age)
top_5 = df.nlargest(5, ['Fare', 'Age'])

```

nlargest() e nsmallest(): Top N

Nota Importante

Mais eficiente que `sort_values().head(n)` para Top N

Exemplo Prático: Análise de Sobreviventes

</> Python

```python
df = pd.read_csv('titanic.csv')

# Mulheres que sobreviveram, primeira classe
filtro = df[(df['Sex'] == 'female') &
            (df['Survived'] == 1) &
            (df['Pclass'] == 1)]

# Ordenar por idade
filtro_ordenado = filtro.sort_values('Age')

# Ver informacoes basicas
resultado = filtro_ordenado[['Name', 'Age', 'Fare']]
print(f"Total: {len(resultado)}")
print(resultado.head(10))

```

Exemplo Prático: Top Passageiros

</> Python

```python
df = pd.read_csv('titanic.csv')

# 10 passageiros que pagaram tarifas mais altas
top_10_tarifas = df.nlargest(10, 'Fare')

# Dessas, quantas sobreviveram?
sobreviventes = top_10_tarifas['Survived'].sum()
taxa = (sobreviventes / 10) * 100

print(f"Sobreviventes entre top 10 tarifas: {sobreviventes}")
print(f"Taxa de sobrevivencia: {taxa}%")

# Ver detalhes
print(top_10_tarifas[['Name', 'Fare', 'Survived']])

```

Exemplo Prático: Crianças por Classe

Python

```python
df = pd.read_csv('titanic.csv')
# Definir criancas (idade < 18)
criancas = df[df['Age'] < 18]

print(f"Total de criancas: {len(criancas)}")

# Distribuicao por classe
print("\nCriancas por classe:")
print(criancas['Pclass'].value_counts().sort_index())

# Taxa de sobrevivencia de criancas
taxa = criancas['Survived'].mean() * 100
print(f"\nTaxa sobrevivencia criancas: {taxa:.1f}%")
```

Exemplo Prático: Análise Complexa

</> Python

```python
df = pd.read_csv('titanic.csv')
# Adultos (>= 18), primeira ou segunda classe
# que sobreviveram, ordenados por idade
resultado = df.query(
    'Age >= 18 and (Pclass == 1 or Pclass == 2) '
    'and Survived == 1'
).sort_values('Age')
# Estatisticas
print(f"Total: {len(resultado)}")
print(f"Idade media: {resultado['Age'].mean():.1f}")
print(f"Tarifa media: {resultado['Fare'].mean():.2f}")
# Top 5 mais jovens
print("\n5 mais jovens:")
print(resultado.head(5)[['Name', 'Age', 'Pclass']])
```

Pipeline de Análise Completo

</> Python

```python
import pandas as pd
# Carregar
df = pd.read_csv('titanic.csv')
# Filtrar
adultos_1classe = df[
    (df['Age'] >= 18) &
    (df['Pclass'] == 1)
]
# Ordenar
ordenado = adultos_1classe.sort_values(
    ['Survived', 'Age'],
    ascending=[False, True]
)
# Selecionar colunas
resultado = ordenado[['Name', 'Age', 'Fare', 'Survived']]
```

Pipeline: Forma Encadeada (Method Chaining)

</> Python

```python
import pandas as pd

# Pipeline completo em uma linha
resultado = (
    pd.read_csv('titanic.csv')
    .query('Age >= 18 and Pclass == 1')
    .sort_values(['Survived', 'Age'],
                 ascending=[False, True])
    [['Name', 'Age', 'Fare', 'Survived']]
)

print(resultado.head())

```

Pipeline: Forma Encadeada (Method Chaining)

Nota Importante

Method chaining torna o código mais legível e conciso

Estatísticas de Subgrupos

</> Python

```python
df = pd.read_csv('titanic.csv')

# Estatisticas por grupo
print("=== PRIMEIRA CLASSE ===")
primeira = df[df['Pclass'] == 1]
print(f"Passageiros: {len(primeira)}")
print(f"Taxa sobrevivencia: {primeira['Survived'].mean():.2%}")
print(f"Idade media: {primeira['Age'].mean():.1f}")

print("\n=== TERCEIRA CLASSE ===")
terceira = df[df['Pclass'] == 3]
print(f"Passageiros: {len(terceira)}")
print(f"Taxa sobrevivencia: {terceira['Survived'].mean():.2%}")
print(f"Idade media: {terceira['Age'].mean():.1f}")

```

Criando Categorias (Binning Manual)

</> Python

```python
df = pd.read_csv('titanic.csv')
# Criar coluna de faixa etaria
df['FaixaEtaria'] = 'Adulto'
df.loc[df['Age'] < 18, 'FaixaEtaria'] = 'Crianca'
df.loc[df['Age'] >= 60, 'FaixaEtaria'] = 'Idoso'

# Ver distribuicao
print(df['FaixaEtaria'].value_counts())

# Analise por faixa
for faixa in ['Crianca', 'Adulto', 'Idoso']:
    subset = df[df['FaixaEtaria'] == faixa]
    taxa = subset['Survived'].mean()
    print(f"{faixa}: {taxa:.2%}")
```

Salvar Resultados Filtrados

Python

```python
df = pd.read_csv('titanic.csv')
# Filtrar dados de interesse
sobreviventes = df[df['Survived'] == 1]
# Salvar em novo CSV
sobreviventes.to_csv('sobreviventes.csv', index=False)
# Salvar apenas colunas especificas
colunas = ['Name', 'Age', 'Sex', 'Pclass']
sobreviventes[colunas].to_csv(
    'sobreviventes_basico.csv',
    index=False
)
# Salvar em Excel
sobreviventes.to_excel('sobreviventes.xlsx',
                       index=False,
                       sheet_name='Sobreviventes')
```

Demonstração Prática

Live Coding

Vamos praticar juntos:

1. Carregar dataset Titanic
2. Exploração inicial (head, info, describe)
3. Seleção de colunas (loc, iloc)
4. Filtragem com boolean indexing
5. Ordenação por múltiplas colunas
6. Análise de subgrupos

Revisão da Aula

Bloco 1 - Estruturas Pandas:

- Series: array 1D com índice
- DataFrame: tabela 2D com rótulos
- Criação: dicionários, listas, arrays, CSV
- Relação com NumPy
- Atributos: shape, columns, index, dtypes

Bloco 2 - Visualização e Inspeção:

- head(), tail(), sample()
- info(), describe(), value_counts()
- Verificação de valores faltantes
- Renomeação: rename(), set_index()

Revisão da Aula (cont.)

Bloco 3 - Seleção e Indexação:

- Seleção de colunas: [ ] e [[ ]]
- loc[ ]: indexação por rótulo (inclusivo)
- iloc[ ]: indexação por posição (exclusivo)
- at[ ]/iat[ ]: acesso rápido a valor único
- SettingWithCopyWarning: evitar chained indexing

Bloco 4 - Filtragem e Ordenação:

- Boolean indexing: máscaras booleanas
- Operadores lógicos: &, |, ~
- query(): sintaxe SQL-like
- sort_values(), sort_index()
- nlargest(), nsmallest()

Conceitos-Chave para Lembrar

1. **Series = Array + Index**, DataFrame = Coleção de Series
2. **loc usa rótulos** (inclusivo), iloc usa posições (exclusivo)
3. **head()/info()/describe()** são essenciais para exploração inicial
4. **Boolean indexing**: df[df['col'] > valor]
5. **Use &, |, ~**, não and, or, not
6. **Sempre use parênteses** em condições múltiplas
7. **query()** é mais legível para filtros complexos
8. **sort_values()** para ordenar, nlargest() para Top N
9. **Method chaining** torna código mais limpo

Pandas vs NumPy: Quando Usar

Use NumPy quando:

- Performance crítica
- Arrays numéricos puros
- Álgebra linear
- Processamento de imagens
- Algoritmos matemáticos

Use Pandas quando:

- Dados tabulares (CSV, Excel)
- Colunas com tipos diferentes
- Análise exploratória
- Limpeza de dados
- Agregações e groupby
- Séries temporais

Nota Importante

Pandas é construído sobre NumPy - use o melhor de cada um!

Boas Práticas

Ao trabalhar com Pandas:

1. **Sempre explore primeiro**: head(), info(), describe()
2. **Verifique valores faltantes**: isnull().sum()
3. **Use nomes descritivos**: df_filtrado, sobreviventes
4. **Comente filtros complexos**: o que você está filtrando?
5. **Use method chaining**: código mais limpo
6. **Evite loops**: Pandas já é vetorizado
7. **Use copy() quando necessário**: evitar modificações acidentais
8. **Documente transformações**: rastreabilidade
9. **Salve checkpoints**: to_csv() para dados intermediários

Erros Comuns e Como Evitar

1. **SettingWithCopyWarning**
   - Evite: `df['col'][row] = value`
   - Use: `df.loc[row, 'col'] = value`
2. **KeyError: coluna não existe**
   - Verifique: `df.columns`
   - Cuidado com espaços e case-sensitive
3. **Usar and/or em vez de &/|**
   - Sempre use: `&`, `|`, `~`
   - Sempre use parênteses: `(cond1) & (cond2)`
4. **Confundir loc com iloc**
   - loc = labels (nomes)
   - iloc = integer location (posições)

Próxima Aula: Manipulação de Dados

Aula 06 - O que vem:

- Transformações: apply(), map(), applymap()
- Operações com strings
- GroupBy e agregações avançadas
- Combinação de DataFrames: merge(), concat(), join()
- Pivot tables e reshape
- Tratamento de duplicatas

Preparação:

- Revisar conceitos desta aula
- Praticar filtragem e seleção
- Experimentar com Titanic dataset
- Pensar em transformações que você gostaria de fazer

Dicas de Estudo

Para dominar Pandas:

1. **Pratique com dados reais**
   - Kaggle datasets
   - Dados públicos (IBGE, data.gov)
   - Seus próprios dados
2. **Documentação oficial**
   - `https://pandas.pydata.org/docs/`
   - Guia do usuário muito completo
   - Exemplos práticos
3. **Cheat sheets**
   - DataCamp Pandas Cheat Sheet
   - Pandas oficial cheat sheet
4. **Explore gradualmente**
   - Comece simples (filtros básicos)
   - Aumente complexidade aos poucos

Recursos Adicionais

Livros:

- ”Python for Data Analysis- Wes McKinney (criador do Pandas)
- ”Pandas Cookbook- Theodore Petrou

Tutoriais Online:

- Pandas oficial: 10 minutes to Pandas
- Real Python: Pandas tutorials
- Kaggle Learn: Pandas micro-course

Vídeos:

- Corey Schafer: Pandas Tutorial Series
- Keith Galli: Complete Python Pandas Data Science Tutorial

Prática:

- Kaggle: Titanic, House Prices, etc.
- DataCamp: Interactive exercises

Resumo: Do NumPy ao Pandas

Progressão do curso:

`Python Básico (Aulas 1-2)`

↓

`NumPy (Aulas 3-4)`

Arrays, vetorização, álgebra linear

↓

`Pandas (Aula 5)`

DataFrames, indexação, filtragem

↓

`Análise de Dados (Próximas aulas)`

Transformações, agregações, visualização

A Seguir: Aula Prática

Exercício Prático

Tempo: 60 minutos
Entrega: via Moodle (notebook)

Tarefas:

1. (atualizado durante a aula)

Notebook: Disponível no Moodle

Obrigado!

Próxima aula: Manipulação de Dados com Pandas
Quinta-feira, 23/10

Dúvidas: via Moodle ou atendimento