Programação para Ciência de Dados

Manipulação de Dados com Pandas

Arthur Casals

23 de Outubro de 2025

Agenda

- Transformações e Apply
- GroupBy e Agregações
- Combinação de DataFrames
- Reshape e Pivot

Recapitulação Aula 05

O que já sabemos:

- Series e DataFrames: estruturas fundamentais
- Criação: dicionários, listas, CSV
- Exploração: head(), info(), describe()
- Seleção: [ ], loc[], iloc[]
- Filtragem: boolean indexing, query()
- Ordenação: sort_values(), nlargest()

Hoje vamos aprofundar:

- Transformar dados de múltiplas formas
- Análise por grupos (split-apply-combine)
- Combinar múltiplos DataFrames
- Reestruturar dados (pivot, melt)

Bloco 1

Transformações e Apply

Transformações de Dados

Por que transformar dados?

- Criar novas features (feature engineering)
- Normalizar e padronizar valores
- Corrigir inconsistências
- Preparar dados para modelagem
- Extrair informações de colunas existentes

Métodos principais:

- **drop():** Remover linhas/colunas
- **Operações vetorizadas:** Mais rápidas e simples
- **apply():** Aplicar função a linhas/colunas
- **map():** Transformar Series elemento por elemento
- **applymap():** Aplicar função a cada célula (deprecated)
- **String methods:** Operações específicas para texto

DataFrame.drop()

Método para remover linhas ou colunas de um DataFrame

Python

```python
DataFrame.drop(
    labels=None,
    axis=0,
    index=None,
    columns=None,
    level=None,
    inplace=False,
    errors='raise'
)
# Retorna: DataFrame ou None
# Retorna um novo DataFrame sem os rótulos especificados (padrão)
# Retorna None quando \texttt{inplace=True}
```

DataFrame.drop() - Parâmetros Básicos

Parâmetro	Tipo/Valores	Descrição
`labels`	único/lista	Rótulo(s) a remover (nomes de linhas ou colunas)
`axis`	0/'index' ou 1/'columns'	Eixo para remover: 0=linhas, 1=colunas
`index`	único/lista	Alternativa: especificar rótulos de linhas diretamente
`columns`	único/lista	Alternativa: especificar rótulos de colunas diretamente

Nota Importante

Boa Prática: Use os parâmetros explícitos `index=` ou `columns=` para maior clareza ao invés de `labels` com `axis`.

DataFrame.drop() - Parâmetros Avançados

Parâmetro	Tipo/Valores	Descrição
`level`	int/nome	Para MultiIndex: especifica qual nível remover
`inplace`	bool	`True`: modifica o DataFrame original, retorna None `False` (padrão): retorna novo DataFrame
`errors`	'raise' / 'ignore'	`'raise'` (padrão): lança erro se rótulo não existe `'ignore'`: suprime erros

Nota Importante

Importante: Por padrão, `drop()` **não modifica** o DataFrame original. Sempre armazene o resultado ou use `inplace=True`.

Adicionar e Remover Colunas

Python

```python
df = pd.read_csv('titanic.csv')
# Adicionar nova coluna (constante)
df['Status'] = 'Passenger'

# Adicionar baseada em outra coluna
df['AgeDouble'] = df['Age'] * 2

# Adicionar com condicao
df['IsChild'] = df['Age'] < 18

# Remover coluna
df = df.drop('AgeDouble', axis=1)

# Remover multiplas colunas
df = df.drop(['Status', 'IsChild'], axis=1)
```

Adicionar Colunas: Formas Alternativas

Python

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

# Usando assign (nao modifica original)
df_new = df.assign(
    AgeGroup='Adult',
    FareDouble=df['Fare'] * 2
)

# Multiplas colunas de uma vez
df[['Col1', 'Col2', 'Col3']] = [[1, 2, 3]] * len(df)

# Insert em posicao especifica
df.insert(2, 'NewCol', 0)  # Posicao 2, nome, valor

```

Operações Vetorizadas: Preferência

Python

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

# Operacoes aritmeticas (vetorizadas - RAPIDO)
df['FareEuros'] = df['Fare'] * 0.85
df['AgePlus10'] = df['Age'] + 10
df['FareSquared'] = df['Fare'] ** 2

# Operacoes entre colunas
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

# Operacoes condicionais vetorizadas
df['IsExpensive'] = df['Fare'] > 50

```

Operações Vetorizadas: Preferência

Nota Importante

Sempre prefira operações vetorizadas - são muito mais rápidas!

apply(), map(), applymap(): Diferenças

Método	Aplica a	Retorna
`apply()`	DataFrame/Series (linhas ou colunas)	Series/DataFrame
`map()`	Series (elemento por elemento)	Series
`applymap()`	DataFrame (cada célula)	DataFrame (deprecated)

Quando usar:

- **apply():** Funções complexas em linhas/colunas
- **map():** Transformações elemento-a-elemento em Series
- **Vetorização:** Sempre que possível (mais rápido)
  - where(), select(), cut(), qcut(), acessores (.str, .dt), pd.Categorical...
  - Numba: decoradores @vectorize, @jit

apply(): Aplicar Função a Colunas

Python

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

# Aplicar funcao a cada coluna (axis=0, padrao)
df_numeric = df[['Age', 'Fare']]
medias = df_numeric.apply(np.mean)

print(medias)
# Age     29.69
# Fare    32.20
```

apply(): Aplicar Função a Colunas (cont.)

Python

```python
# Funcao customizada
def amplitude(col):
    return col.max() - col.min()

amplitudes = df_numeric.apply(amplitude)

print(amplitudes)
# Age      79.5800
# Fare    512.3292
```

apply(): Aplicar Função a Linhas

Python

```python
df = pd.read_csv('titanic.csv')
# Aplicar funcao a cada linha (axis=1)
def calcular_familia(row):
    return row['SibSp'] + row['Parch'] + 1

df['FamilySize'] = df.apply(calcular_familia, axis=1)

# Funcao inline (lambda)
df['FamilySize'] = df.apply(
    lambda row: row['SibSp'] + row['Parch'] + 1,
    axis=1
)

# MELHOR: vetorizado (evite apply se possivel)
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
```

apply(): Retornar Múltiplos Valores

</> Python

```python
df = pd.read_csv('titanic.csv')
# Funcao que retorna Series
def stats(col):
    return pd.Series({
        'min': col.min(),
        'max': col.max(),
        'mean': col.mean()
    })

resultado = df[['Age', 'Fare']].apply(stats)
print(resultado)
#           Age       Fare
# min      0.42       0.00
# max     80.00     512.33
# mean    29.69      32.20
```

apply(): Exemplo Complexo

Python

```python
df = pd.read_csv('titanic.csv')
# Extrair titulo do nome (Mr., Mrs., Miss., etc.)
def extrair_titulo(nome):
    titulo = nome.split(',')[1].split('.')[0].strip()
    return titulo

df['Titulo'] = df['Name'].apply(extrair_titulo)

print(df['Titulo'].value_counts().head())
# Mr          517
# Miss        182
# Mrs         125
# Master       40
# Dr            7
```

map(): Transformar Series

</> Python

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

# Map com dicionario
mapa_classe = {1: 'First', 2: 'Second', 3: 'Third'}
df['ClasseName'] = df['Pclass'].map(mapa_classe)

# Map com funcao
df['SexLong'] = df['Sex'].map(
    lambda x: 'Male' if x == 'male' else 'Female'
)

# Map com Series (como um dicionario)
medias_por_classe = df.groupby('Pclass')['Fare'].mean()
df['FareMedia'] = df['Pclass'].map(medias_por_classe)
```

map(): Transformar Series

</> Python

```python
print(medias_por_classe)
# Pclass
# 1    84.154687
# 2    20.662183
# 3    13.675550
```

map() vs apply() em Series

</> Python

```python
df = pd.read_csv('titanic.csv')
# map: apenas para Series
df['Sex'].map(lambda x: x.upper())
# apply: funciona para Series e DataFrame
df['Sex'].apply(lambda x: x.upper())

# Para Series simples, map e apply sao similares
# map é ligeiramente mais rapido para mapeamentos
# apply é mais flexivel (aceita funcoes complexas)
def processar(x):
    # Logica complexa
    return resultado

df['Sex'].apply(processar)
```

replace(): Substituição de Valores

</> Python

```python
df = pd.read_csv('titanic.csv')
# Substituir valores especificos
df['Embarked'] = df['Embarked'].replace('S', 'Southampton')

# Multiplas substituicoes
df['Embarked'] = df['Embarked'].replace({
    'S': 'Southampton',
    'C': 'Cherbourg',
    'Q': 'Queenstown'
})
# Substituir em todo o DataFrame
df = df.replace({'male': 'M', 'female': 'F'})

# Substituir valores faltantes
df = df.replace(np.nan, 0)
```

String Methods: str Accessor

Operações com strings no Pandas:

- Acessar via `.str`
- Métodos vetorizados (aplicados a toda Series)
- Similar a métodos de strings Python
- Lidam automaticamente com NaN

Principais categorias:

- **Case:** lower(), upper(), capitalize()
- **Limpeza:** strip(), replace()
- **Divisão:** split(), extract()
- **Busca:** contains(), startswith(), endswith()
- **Contagem:** len(), count()

String Methods: Case e Limpeza

</> Python

```python
df = pd.read_csv('titanic.csv')
# Minusculas/Maiusculas
df['NameLower'] = df['Name'].str.lower()
df['NameUpper'] = df['Name'].str.upper()
df['NameTitle'] = df['Name'].str.title()

# Remover espacos
df['NameClean'] = df['Name'].str.strip()

# Substituir substring
df['NameMod'] = df['Name'].str.replace('Mr.', 'Mister')

# Remover caracteres especiais
df['NameAlpha'] = df['Name'].str.replace('[^a-zA-Z ]', '',
                                         regex=True)
```

String Methods: Divisão e Extração

Python

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

# Dividir string
# Name: "Braund, Mr. Owen Harris"
sobrenome = df['Name'].str.split(',').str[0]
resto = df['Name'].str.split(',').str[1]

# Split com expand (retorna DataFrame)
nome_split = df['Name'].str.split(',', expand=True)
df['Sobrenome'] = nome_split[0]
df['Resto'] = nome_split[1]

# Extrair com regex
df['Titulo'] = df['Name'].str.extract(r', ([^.]+)\.')

```

String Methods: Busca e Verificação

Python

```python
df = pd.read_csv('titanic.csv')
# Verificar se contem substring
tem_mr = df['Name'].str.contains('Mr.')
print(tem_mr.sum())  # Quantos "Mr."
# Comeca com
comeca_b = df['Name'].str.startswith('B')
# Termina com
termina_s = df['Name'].str.endswith('s')
# Case insensitive
tem_john = df['Name'].str.contains('john', case=False)
# Contar ocorrencias
n_espacos = df['Name'].str.count(' ')
```

String Methods: Tamanho e Padding

</> Python

```python
df = pd.read_csv('titanic.csv')
# Tamanho da string
df['NameLength'] = df['Name'].str.len()

# Slice (fatiar)
primeiras_5 = df['Name'].str[:5]
ultimas_3 = df['Name'].str[-3:]

# Padding (preencher com caracteres)
df['IdPadded'] = df['PassengerId'].astype(str).str.zfill(5)
# 1 -> 00001, 42 -> 00042

# Concatenar strings
df['Info'] = df['Name'] + ' (' + df['Sex'] + ')'
```

Exemplo Prático: Extrair Título

Python

```python
df = pd.read_csv('titanic.csv')
# Extrair titulo do nome
# "Braund, Mr. Owen Harris" -> "Mr"
df['Titulo'] = df['Name'].str.extract(r', ([^.]+)\.')
# Limpar espacos
df['Titulo'] = df['Titulo'].str.strip()
# Ver distribuicao
print(df['Titulo'].value_counts())
# Agrupar titulos raros
titulos_raros = ['Don', 'Rev', 'Dr', 'Mme', 'Ms',
                 'Major', 'Lady', 'Sir', 'Mlle',
                 'Col', 'Capt', 'Countess', 'Jonkheer']
df['TituloGroup'] = df['Titulo'].replace(titulos_raros, 'Rare')
```

Conversão de Tipos

Por que converter tipos?

- Otimizar memória (int64 $\rightarrow$ int32)
- Corrigir leitura incorreta de CSV
- Preparar dados para operações específicas
- Converter categóricas para economia de memória

Métodos principais:

- **astype():** Conversão forçada
- **to_numeric():** Strings $\rightarrow$ números (com error handling)
- **to_datetime():** Strings $\rightarrow$ datas
- **to_timedelta():** Strings $\rightarrow$ intervalos de tempo
- **astype('category'):** Otimizar categóricas

astype(): Conversão Básica

</> Python

```python
df = pd.read_csv('titanic.csv')
# Converter para diferente tipo numerico
df['Pclass'] = df['Pclass'].astype('int32')
df['Age'] = df['Age'].astype('float32')
# Converter para string
df['PassengerId'] = df['PassengerId'].astype(str)
# Converter booleano
df['Survived'] = df['Survived'].astype(bool)
```

astype(): Conversão Básica (cont.)

Python

```python
# Verificar tipos
print(df.dtypes)
# PassengerId     object
# Survived          bool
# Pclass           int32
# ...

# Memoria usada
print(df.memory_usage(deep=True))
# Index            132
# PassengerId    46224
# Survived         891
# ...
```

to_numeric(): Conversão Segura

Python

```python
# Dados com valores invalidos
s = pd.Series(['1', '2', '3.5', 'four', '5'])
# astype vai dar erro: s.astype(float) => ValueError!
# to_numeric com error handling
s_numeric = pd.to_numeric(s, errors='coerce')
print(s_numeric)
# 0    1.0
# 1    2.0
# 2    3.5
# 3    NaN   <- invalido virou NaN
# 4    5.0
# dtype: float64
```

to_numeric(): Conversão Segura

Python

```python
# Ignorar erros (manter original)
s_numeric = pd.to_numeric(s, errors='ignore')
print(s_numeric)
# 0    1.0
# 1    2.0
# 2    3.5
# 3   four
# 4    5.0
# dtype: object  <- tipo agora é objeto!
```

Categorical: Otimização de Memória

Python

```python
df = pd.read_csv('titanic.csv')
# Memoria antes
print(f"Memoria Sex: {df['Sex'].memory_usage(deep=True)} bytes")
# Memoria Sex: 47983 bytes
# Converter para categorical
df['Sex'] = df['Sex'].astype('category')
df['Embarked'] = df['Embarked'].astype('category')
df['Pclass'] = df['Pclass'].astype('category')
# Memoria depois
print(f"Memoria Sex: {df['Sex'].memory_usage(deep=True)} bytes")
# Memoria Sex: 1239 bytes
print(df['Sex'].cat.categories)
# Index(['female', 'male'], dtype='object')
```

Binning: Discretização de Variáveis

O que é binning?

- Converter variável contínua em categórica
- Agrupar valores em ”bins” (faixas)
- Facilita análise e visualização
- Reduz ruído e outliers

Dois métodos principais:

- **pd.cut():** Bins de largura igual
- **pd.qcut():** Bins com frequência igual (quantis)

Nota Importante

Escolha baseada no objetivo: distribuição uniforme (qcut) ou intervalos fixos (cut)

pd.cut(): Bins de Largura Igual

Python

```python
df = pd.read_csv('titanic.csv')
# Criar faixas de idade com largura igual
df['AgeGroup'] = pd.cut(df['Age'], bins=4)  # 4 bins automáticos
# Criar faixas de idade pré-definidas
df['AgeGroup'] = pd.cut(
    df['Age'],
    bins=[0, 18, 40, 60, 100], # (0, 18], (18, 40], (40, 60], (60, 100]
    labels=['Crianca', 'Jovem', 'Adulto', 'Idoso']
)
print(df['AgeGroup'].value_counts())
# Jovem     357
# Adulto    241
# Crianca    83
# Idoso      33

```

pd.qcut(): Bins por Quantis

Python

```python
df = pd.read_csv('titanic.csv')
# Criar faixas com frequência similar (quartis)
df['FareQuartile'] = pd.qcut(
    df['Fare'],
    q=4,
    labels=['Q1-Baixo', 'Q2-Medio', 'Q3-Alto', 'Q4-Premium']
)
print(df['FareQuartile'].value_counts())
# Q1-Baixo      223
# Q2-Medio      223
# Q3-Alto       223
# Q4-Premium    222
# Ver os limites criados
print(df['FareQuartile'].cat.categories)
# Index(['Q1-Baixo', 'Q2-Medio', 'Q3-Alto', 'Q4-Premium'], dtype='object')
```

Binning: Comparação cut vs qcut

</> Python

```python
df = pd.read_csv('titanic.csv')
# cut: intervalos fixos
df['Fare_Cut'] = pd.cut(df['Fare'], bins=4)
print("Distribuição com cut():")
print(df['Fare_Cut'].value_counts().sort_index())
# (-0.513, 128.08]    871  <- DESBALANCEADO!
# (128.08, 256.16]     12
# (256.16, 384.24]      6
# (384.24, 512.33]      2
# qcut: frequências iguais
df['Fare_Qcut'] = pd.qcut(df['Fare'], q=4)
print("\nDistribuição com qcut():")
print(df['Fare_Qcut'].value_counts().sort_index())
# Todos com ~223 elementos (BALANCEADO!)
```

Binning: Opções Avançadas

Python

```python
df = pd.read_csv('titanic.csv')
# Incluir bins abertos à direita ou esquerda
df['Age_Right'] = pd.cut(df['Age'], bins=[0, 18, 60, 100],
                         right=True)   # (0, 18]
df['Age_Left'] = pd.cut(df['Age'], bins=[0, 18, 60, 100],
                        right=False)  # [0, 18)
# Retornar os bins como inteiros
df['Age_Bin'] = pd.cut(df['Age'], bins=5, labels=False)
# 0, 1, 2, 3, 4
# Incluir valores fora dos limites
df['Age_Extended'] = pd.cut(df['Age'], bins=[0, 18, 60],
                            labels=['Jovem', 'Adulto'])
# NaN para idade > 60
```

Quando Usar cut vs qcut

Use pd.cut() quando:

- Limites têm significado real
- Faixas fixas são importantes
- Exemplo: Idade (0-18, 18-65, 65+)
- Exemplo: Notas (0-49, 50-69, 70-100)
- Exemplo: IMC (abaixo, normal, sobrepeso)

Use pd.qcut() quando:

- Quer distribuição balanceada
- Valores extremos (outliers)
- Machine learning (features balanceadas)
- Exemplo: Segmentação de clientes
- Exemplo: Ranking percentual

Nota Importante

qcut é útil para evitar bins vazios com dados assimétricos

Exemplo Prático: Pipeline de Transformação

Python

```python
df = pd.read_csv('titanic.csv')
print(f"Features: {df.shape[1]} colunas")
# Features: 12 colunas
# 1. Criar FamilySize
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
# 2. Extrair titulo
df['Titulo'] = df['Name'].str.extract(r', ([^.]+)\.')
# 3. Criar IsAlone
df['IsAlone'] = (df['FamilySize'] == 1).astype(int)
# 4. Categorizar idade
df['AgeGroup'] = pd.cut(df['Age'],
                        bins=[0, 12, 18, 60, 100],
                        labels=['Child','Teen','Adult','Senior'])
```

Exemplo Prático: Pipeline (cont.)

Python

```python
# 5. Converter para categorical
for col in ['Sex', 'Embarked', 'Pclass', 'Titulo']:
    df[col] = df[col].astype('category')

# 6. Normalizar tarifa
df['FareNorm'] = (df['Fare'] - df['Fare'].min()) / \
                 (df['Fare'].max() - df['Fare'].min())

# 7. Preencher idade faltante com mediana
df['Age'].fillna(df['Age'].median(), inplace=True)

print(f"Features: {df.shape[1]} colunas")
# Features: 17 colunas
print(df.head())
# Check new columns
```

Window Functions: Operações em Janelas

O que são Window Functions?

- Operações que consideram valores ”vizinhos”
- Não agregam (mantêm número de linhas)
- Úteis para comparações temporais/ordenadas
- Similar ao SQL window functions

Principais funções:

- **shift():** Valores anteriores/posteriores
- **diff():** Diferença entre valores consecutivos
- **pct_change():** Variação percentual
- **rolling():** Médias móveis
- **rank():** Ranking dentro de grupos
- **cumsum()/cumcount():** Acumulados

shift(): Acessar Valores Anteriores

Python

```python
vendas = pd.DataFrame({  # Dados de vendas mensais
    'mes': ['Jan', 'Fev', 'Mar', 'Abr', 'Mai'],
    'valor': [100, 120, 115, 140, 150]
})
vendas['mes_anterior'] = vendas['valor'].shift(1)
vendas['mes_seguinte'] = vendas['valor'].shift(-1)
print(vendas)
#   mes  valor  mes_anterior  mes_seguinte
# 0  Jan    100           NaN         120.0
# 1  Fev    120         100.0         115.0
# 2  Mar    115         120.0         140.0
# 3  Abr    140         115.0         150.0
# 4  Mai    150         140.0           NaN
```

diff() e pct_change(): Variações

Python

```python
vendas = pd.DataFrame({
    'mes': ['Jan', 'Fev', 'Mar', 'Abr'],
    'valor': [100, 120, 115, 140]
})
# Diferença absoluta
vendas['variacao'] = vendas['valor'].diff()
# Variação percentual
vendas['pct_variacao'] = vendas['valor'].pct_change()
print(vendas)
#   mes  valor  variacao  pct_variacao
# 0  Jan    100       NaN           NaN
# 1  Fev    120      20.0      0.200000  # +20%
# 2  Mar    115      -5.0     -0.041667  # -4.17%
# 3  Abr    140      25.0      0.217391  # +21.74%
```

rank(): Ranking de Valores

Python

```python
df = pd.read_csv('titanic.csv')
# Ranking geral de tarifa (maior = 1)
df['FareRank'] = df['Fare'].rank(ascending=False, method='dense')
# Ranking dentro de cada classe
df['FareRankClass'] = df.groupby('Pclass')['Fare'].rank(
    ascending=False,
    method='dense'
)
# Métodos de ranking
# 'average': média das posições (padrão)
# 'min': menor posição
# 'max': maior posição
# 'first': ordem de aparição
# 'dense': sem gaps (1, 2, 3, ...)
```

rank(): Ranking de Valores (cont.)

Python

```python
print(df[['Name','Pclass','Fare','FareRankClass']].head(10))
#    [Name...] Pclass     Fare  FareRankClass
# 0                 3   7.2500          102.0
# 1                 1  71.2833           40.0
# 2                 3   7.9250           79.0
# 3                 1  53.1000           55.0
# ...
```

cumsum() e cumcount(): Acumulados

Python

```python
vendas = pd.DataFrame({
    'dia': range(1, 6),
    'valor': [100, 120, 115, 140, 150]
})
# Soma acumulada
vendas['acumulado'] = vendas['valor'].cumsum()
# Contagem acumulada (útil com groupby)
vendas['count'] = vendas['valor'].cumcount()
print(vendas)
#    dia  valor  acumulado  count
# 0    1    100        100      0
# 1    2    120        220      1
# 2    3    115        335      2
# 3    4    140        475      3
# 4    5    150        625      4
```

Window Functions com GroupBy

</> Python

```python
df = pd.read_csv('titanic.csv')
# Ranking de idade dentro de cada classe
df['AgeRank'] = df.groupby('Pclass')['Age'].rank(
    ascending=False
)
# Diferença da idade média da classe
df['AgeDiff'] = df.groupby('Pclass')['Age'].transform(
    lambda x: x - x.mean()
)
# Acumulado de tarifa por classe (ordem alfabética de nome)
df = df.sort_values(['Pclass', 'Name'])
df['FareCumsum'] = df.groupby('Pclass')['Fare'].cumsum()
```

Window Functions com GroupBy (cont.)

Python

```python
print(df[['Pclass','Age','AgeRank','AgeDiff']].head())
#       Pclass    Age  AgeRank    AgeDiff
# 730        1  29.00    133.0  -9.233441
# 305        1   0.92    186.0 -37.313441
# 297        1   2.00    185.0 -36.233441
# 498        1  25.00    146.0 -13.233441
# 460        1  48.00     52.0   9.766559
```

Exemplo Prático: Análise de Crescimento

Python

```python
# Vendas por mês
vendas_mensais = pd.DataFrame({
    'mes': pd.date_range('2024-01', periods=6, freq='MS'),
    'vendas': [1000, 1200, 1150, 1400, 1500, 1350]
})
# Pipeline completo de análise temporal
vendas_mensais['mes_anterior'] = vendas_mensais['vendas'].shift(1)
vendas_mensais['crescimento'] = vendas_mensais['vendas'].diff()
vendas_mensais['crescimento_pct'] = vendas_mensais['vendas'].pct_change()
vendas_mensais['acumulado'] = vendas_mensais['vendas'].cumsum()
vendas_mensais['media_movel_3'] = vendas_mensais['vendas'].rolling(
    window=3
).mean()
```

Exemplo Prático: Análise de Crescimento (cont.)

Python

```python
print(vendas_mensais)
#          mes  vendas  mes_anterior  crescimento  crescimento_pct  \
# 0 2024-01-01    1000           NaN          NaN              NaN
# 1 2024-02-01    1200        1000.0        200.0         0.200000
# 2 2024-03-01    1150        1200.0        -50.0        -0.041667
# ...
#   acumulado  media_movel_3
# 0      1000           NaN
# 1      2200           NaN
# 2      3350   1116.666667
# ...
```

Window Functions: Casos de Uso

Cenários comuns:

1. **Comparações temporais**
   - Vendas mês-a-mês
   - Crescimento ano-a-ano (YoY)
2. **Rankings**
   - Top N produtos por categoria
   - Classificação de clientes
3. **Métricas acumuladas**
   - Revenue acumulado
   - Contagem progressiva de eventos
4. **Médias móveis**
   - Suavização de tendências
   - Detecção de anomalias

Bloco 2

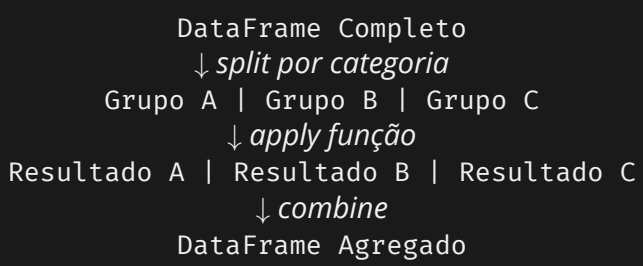
GroupBy e Agregações

GroupBy: Split-Apply-Combine

Conceito:

1. **Split:** Dividir dados em grupos
2. **Apply:** Aplicar função a cada grupo
3. **Combine:** Combinar resultados

Visual:



GroupBy: Quando Usar

Perguntas que GroupBy responde:

- Qual a **média** por categoria?
- Quantos elementos em cada grupo?
- Qual o **máximo/mínimo** por grupo?
- Como os grupos se **comparam**?

Exemplos práticos:

- Taxa de sobrevivência por classe (Titanic)
- Vendas totais por região
- Salário médio por departamento
- Nota média por turma
- Métricas por segmento de clientes

GroupBy: Sintaxe Básica

</> Python

```python
df = pd.read_csv('titanic.csv')
# Agrupar por uma coluna
grouped = df.groupby('Pclass')
print(type(grouped))  # DataFrameGroupBy object
# Ver grupos
print(grouped.groups.keys())
# dict_keys([1, 2, 3])
# Numero de elementos em cada grupo
print(grouped.size())
# Pclass
# 1    216
# 2    184
# 3    491
```

GroupBy: Agregações Simples

Python

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

# Agrupar e calcular media
media_por_classe = df.groupby('Pclass')['Age'].mean()
print(media_por_classe)
# Pclass
# 1    38.23
# 2    29.88
# 3    25.14

# Multiplas estatisticas
stats = df.groupby('Pclass')['Age'].agg(['mean', 'std',
                                         'min', 'max'])
print(stats)
# Colunas: mean, std, min, max
```

GroupBy: Múltiplas Colunas

</> Python

```python
df = pd.read_csv('titanic.csv')
# Agrupar por multiplas colunas
grouped = df.groupby(['Pclass', 'Sex'])

# Taxa de sobrevivencia por classe e sexo
sobrev = grouped['Survived'].mean()
print(sobrev)
# Pclass  Sex
# 1       female    0.968
#         male      0.369
# 2       female    0.921
#         male      0.157
# 3       female    0.500
#         male      0.135
```

GroupBy: Agregações em Múltiplas Colunas

</> Python

```python
df = pd.read_csv('titanic.csv')
# Agregar multiplas colunas
resultado = df.groupby('Pclass')[['Age', 'Fare']].mean()
print(resultado)
#               Age        Fare
# Pclass
# 1         38.23    84.15
# 2         29.88    20.66
# 3         25.14    13.68
# Diferentes funcoes para diferentes colunas
resultado = df.groupby('Pclass').agg({
    'Age': 'mean',
    'Fare': 'sum',
    'Survived': 'count'
})
```

agg(): Agregações Personalizadas

Python

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

# Multiplas funcoes para uma coluna
resultado = df.groupby('Pclass')['Age'].agg([
    'count',
    'mean',
    'std',
    'min',
    ('q25', lambda x: x.quantile(0.25)),
    ('q75', lambda x: x.quantile(0.75)),
    'max'
])
```

agg(): Agregações Personalizadas (cont.)

Python

```python
print(resultado)
#         count        mean         std   min   q25   q75   max
# Pclass
# 1         186  38.233441  14.802856  0.92  27.0  49.0  80.0
# 2         173  29.877630  14.001077  0.67  23.0  36.0  70.0
# 3         355  25.140620  12.495398  0.42  18.0  32.0  74.0


```

agg(): Funções Customizadas

Python

```python
df = pd.read_csv('titanic.csv')
# Funcao customizada
def amplitude(x):
    return x.max() - x.min()
# Usar funcao customizada
resultado = df.groupby('Pclass')['Age'].agg([
    'mean',
    amplitude,
    ('range', amplitude)  # Com nome customizado
])
# Funcoes lambda
resultado = df.groupby('Pclass')['Fare'].agg([
    ('soma', 'sum'),
    ('dobro_media', lambda x: x.mean() * 2)
])
```

agg(): Diferentes Funções por Coluna

Python

```python
df = pd.read_csv('titanic.csv')
# Dicionario: coluna -> funcoes
resultado = df.groupby('Pclass').agg({
    'Age': ['mean', 'std'],
    'Fare': ['sum', 'mean', 'max'],
    'Survived': ['sum', 'mean'],
    'PassengerId': 'count'
})
```

agg(): Diferentes Funções por Coluna (cont.)

Python

```python
print(resultado)
# MultiIndex em colunas - rótulos são tuplas: ('Age', 'mean')
#               Age                    Fare                        Survived
#              mean         std         sum       mean       max      sum
# Pclass
# 1       38.233441   14.802856  18177.4125  84.154687  512.3292      136
# 2       29.877630   14.001077   3801.8417  20.662183   73.5000       87
# 3       25.140620   12.495398   6714.6951  13.675550   69.5500      119
# \
#                PassengerId
#             mean       count
# Pclass
# 1       0.629630         216
# 2       0.472826         184
# 3       0.242363         491
```

agg(): Diferentes Funções por Coluna (cont.)

</> Python

```python
# Achatar colunas: ('Age', 'mean') => Age_mean
resultado.columns = ['_'.join(col) for col in resultado.columns]
print(resultado)
#           Age_mean    Age_std    Fare_sum  Fare_mean  Fare_max [...]
# Pclass
# 1       38.233441  14.802856  18177.4125  84.154687  512.3292
# 2       29.877630  14.001077   3801.8417  20.662183   73.5000
# 3       25.140620  12.495398   6714.6951  13.675550   69.5500
```

Named Aggregations: Sintaxe Moderna

Problema com agg() tradicional:

- Colunas com MultiIndex difíceis de usar
- Precisa renomear manualmente depois
- Código menos legível

Solução: Named Aggregations

- Nomear agregações diretamente
- Sintaxe clara: `nome=(coluna, função)`
- Sem MultiIndex nos resultados
- Mais fácil de documentar

Nota Importante

Recomendado para código de produção - muito mais legível!

Named Aggregations: Sintaxe

</> Python

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

# ANTES: agg tradicional (difícil de ler)
resultado_antigo = df.groupby('Pclass').agg({
    'Age': ['mean', 'std'],
    'Fare': ['sum', 'mean'],
    'Survived': 'sum'
})
# Colunas: ('Age', 'mean'), ('Age', 'std'), ...
resultado_antigo.columns = ['_'.join(col) for col in resultado_antigo.
    columns]
```

Named Aggregations: Sintaxe (cont.)

Python

```python
# AGORA: Named aggregations (limpo!)
resultado_novo = df.groupby('Pclass').agg(
    idade_media=('Age', 'mean'),
    idade_desvio=('Age', 'std'),
    receita_total=('Fare', 'sum'),
    ticket_medio=('Fare', 'mean'),
    sobreviventes=('Survived', 'sum')
)
```

Named Aggregations: Sintaxe (cont.)

Python

```python
print(resultado_novo)
#         idade_media  idade_desvio  receita_total  ticket_medio
    sobreviventes
# Pclass
# 1         38.233441     14.802856     18177.4125     84.154687
      136
# 2         29.877630     14.001077      3801.8417     20.662183
       87
# 3         25.140620     12.495398      6714.6951     13.675550
      119
```

Named Aggregations: Múltiplos Grupos

</> Python

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

# Agregações nomeadas com múltiplos grupos
analise = df.groupby(['Pclass', 'Sex']).agg(
    total_passageiros=('PassengerId', 'count'),
    total_sobreviventes=('Survived', 'sum'),
    taxa_sobrevivencia=('Survived', 'mean'),
    idade_media=('Age', 'mean'),
    tarifa_mediana=('Fare', 'median'),
    tarifa_maxima=('Fare', 'max')
).round(2)
```

Named Aggregations: Múltiplos Grupos (cont.)

Python

```python
print(analise) # Colunas com nomes claros, sem MultiIndex!
#                total_passageiros  total_sobreviventes  [...]
# Pclass Sex
# 1      female                 94                   91
#        male                  122                   45
# 2      female                 76                   70
#        male                  108                   17
# 3      female                144                   72
#        male                  347                   47
```

Named Aggregations: Funções Customizadas

</> Python

```python
df = pd.read_csv('titanic.csv')
# Funções customizadas com named agg
def amplitude(x):
    return x.max() - x.min()
def percentil_90(x):
    return x.quantile(0.9)
resultado = df.groupby('Pclass').agg(
    idade_min=('Age', 'min'),
    idade_max=('Age', 'max'),
    idade_amplitude=('Age', amplitude),
    tarifa_p90=('Fare', percentil_90),
    tarifa_total=('Fare', 'sum')
)
```

Named Aggregations: Com Lambda

</> Python

```python
df = pd.read_csv('titanic.csv')
# Lambda functions com named aggregations
resultado = df.groupby('Pclass').agg(
    # Funções built-in
    total=('PassengerId', 'count'),
    idade_media=('Age', 'mean'),
    # Lambda para cálculos customizados
    idade_cv=('Age', lambda x: x.std() / x.mean()),  # Coef. variação
    tarifa_iqr=('Fare', lambda x: x.quantile(0.75) - x.quantile(0.25)),
    sobrev_pct=('Survived', lambda x: x.mean() * 100)
).round(2)
```

Named Aggregations: Vantagens

Por que usar Named Aggregations:

1. **Legibilidade**
   - Código auto-documentado
   - Nomes significativos para cada métrica
2. **Sem pós-processamento**
   - Não precisa renomear colunas depois
   - Sem MultiIndex para ”achatar”
3. **Fácil manutenção**
   - Adicionar/remover métricas é simples
   - Fica claro o que cada agregação faz
4. **Compatível com pipelines**
   - Integra bem com method chaining
   - Pronto para usar em visualizações

Filtrar Grupos

Python

```python
df = pd.read_csv('titanic.csv')
# Criar grupos por idade
df['AgeGroup'] = pd.cut(df['Age'], bins=[0, 18, 60, 100],
                        labels=['Child', 'Adult', 'Senior'])
# Filtrar grupos com mais de 100 pessoas
grupos_grandes = df.groupby('AgeGroup').filter(
    lambda x: len(x) > 100
)
print(f"Original: {len(df)}")
print(f"Filtrado: {len(grupos_grandes)}")
# Apenas grupos onde sobrevivencia > 50%
sobrev_alta = df.groupby('Pclass').filter(
    lambda x: x['Survived'].mean() > 0.5
)
```

transform() vs aggregate()

aggregate():

- Reduz grupos a valores únicos
- Retorna resultado menor
- Uma linha por grupo
- Ex: média, soma, count

Resultado:

- Shape menor que original
- Índice = grupos

transform():

- Mantém shape original
- Retorna valor para cada linha
- Broadcasting de agregação
- Ex: normalização por grupo

Resultado:

- Shape igual ao original
- Pode adicionar como coluna

transform(): Exemplo

Python

```python
df = pd.read_csv('titanic.csv')
# Calcular media de idade por classe
media_classe = df.groupby('Pclass')['Age'].transform('mean')

# Agora media_classe tem o mesmo tamanho que df!
print(len(media_classe))  # 891
print(len(df))            # 891

# Adicionar como nova coluna
df['AgeMeanClass'] = media_classe

# Normalizar idade dentro de cada classe
df['AgeNormClass'] = df.groupby('Pclass')['Age'].transform(
    lambda x: (x - x.mean()) / x.std()
)
```

transform(): Casos de Uso

</> Python

```python
df = pd.read_csv('titanic.csv')
# 1. Preencher NaN com media do grupo
df['Age'] = df.groupby('Pclass')['Age'].transform(
    lambda x: x.fillna(x.mean())
)

# 2. Ranking dentro do grupo
df['FareRank'] = df.groupby('Pclass')['Fare'].transform(
    lambda x: x.rank(ascending=False)
)

# 3. Diferenca da media do grupo
df['FareDiff'] = df.groupby('Pclass')['Fare'].transform(
    lambda x: x - x.mean()
)
```

apply(): Mais Flexível que agg/transform

</> Python

```python
df = pd.read_csv('titanic.csv')
# apply pode retornar Series, DataFrame ou escalar
def top_fares(group):
    return group.nlargest(3, 'Fare')[['Name', 'Fare']]
# Top 3 tarifas por classe
resultado = df.groupby('Pclass').apply(top_fares)
print(resultado)
# Funcao customizada complexa
def stats_customizado(group):
    return pd.Series({
        'count': len(group),
        'survived_pct': group['Survived'].mean(),
        'avg_age': group['Age'].mean()
    })
resultado = df.groupby('Pclass').apply(stats_customizado)
```

Iteração sobre Grupos

Python

```python
df = pd.read_csv('titanic.csv')
# Iterar sobre grupos
for nome_grupo, dados_grupo in df.groupby('Pclass'):
    print(f"\n=== Classe {nome_grupo} ===")
    print(f"Passageiros: {len(dados_grupo)}")
    print(f"Sobreviventes: {dados_grupo['Survived'].sum()}")
    print(f"Taxa: {dados_grupo['Survived'].mean():.2%}")
# === Classe 1 ===
# Passageiros: 216
# Sobreviventes: 136
# Taxa: 62.96%
grouped = df.groupby('Pclass')
primeira_classe = grouped.get_group(1)
print(primeira_classe.head())
# Visualiza só primeira classe
```

Exemplo Prático: Análise por Grupo

</> Python

```python
df = pd.read_csv('titanic.csv')
# Analise completa por classe e sexo
analise = df.groupby(['Pclass', 'Sex']).agg({
    'PassengerId': 'count',
    'Survived': ['sum', 'mean'],
    'Age': ['mean', 'median'],
    'Fare': ['mean', 'max']
})
# Renomear colunas
analise.columns = ['_'.join(col) for col in analise.columns]
analise = analise.rename(columns={
    'PassengerId_count': 'Total',
    'Survived_sum': 'Sobreviventes',
    'Survived_mean': 'Taxa_Sobrev'
})
```

Exemplo Prático: Ranking por Grupo

Python

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

# Ranking de tarifa dentro de cada classe
df['FareRank'] = df.groupby('Pclass')['Fare'].rank(
    ascending=False,
    method='dense'
)

# Top 5 tarifas de cada classe
top5_por_classe = df[df['FareRank'] <= 5].sort_values(
    ['Pclass', 'FareRank']
)
```

Bloco 3

Combinação de DataFrames

Por que Combinar DataFrames?

Cenários comuns:

- Dados em múltiplos arquivos
- Informações complementares em tabelas separadas
- Junção de datasets de diferentes fontes
- Adicionar features de lookup tables
- Consolidar dados temporais

Métodos principais:

- **concat():** Empilhar DataFrames (vertical/horizontal)
- **merge():** JOIN estilo SQL (por chaves)
- **join():** Merge pelo índice
- **append():** Adicionar linhas (deprecated, use concat)

concat() vs merge()

concat():

- **Empilhamento** de dados
- Não requer chave comum
- Baseado em posição/índice
- Útil para dados similares

Casos de uso:

- Múltiplos CSVs com mesma estrutura
- Dados de períodos diferentes
- Adicionar colunas

merge():

- **JOIN** estilo SQL
- Requer chave(s) comum(ns)
- Baseado em valores
- Útil para relacionamentos

Casos de uso:

- Juntar tabelas relacionadas
- Adicionar info de lookup
- Relacionamentos N:M

concat(): Empilhamento Vertical

Python

```python
# Dois DataFrames com mesma estrutura
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

df2 = pd.DataFrame({
    'A': [7, 8, 9],
    'B': [10, 11, 12]
})

```

concat(): Empilhamento Vertical (cont.)

</> Python

```python
# Concatenar verticalmente (empilhar linhas)
resultado = pd.concat([df1, df2])
print(resultado)
#    A   B
# 0  1   4
# 1  2   5
# 2  3   6
# 0  7  10  <- indice repetido!
# 1  8  11
# 2  9  12

```

concat(): Resetar Índice

Python

```python
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Sem resetar indice
resultado = pd.concat([df1, df2])
print(resultado.index)  # [0, 1, 0, 1]

# Ignorar indice original
resultado = pd.concat([df1, df2], ignore_index=True)
print(resultado.index)  # [0, 1, 2, 3]

# Alternativa: resetar depois
resultado = pd.concat([df1, df2]).reset_index(drop=True)

```

concat(): Empilhamento Horizontal

Python

```python
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
df2 = pd.DataFrame({
    'C': [7, 8, 9],
    'D': [10, 11, 12]
})
# Concatenar horizontalmente (adicionar colunas)
resultado = pd.concat([df1, df2], axis=1)
print(resultado)
#    A  B   C   D
# 0  1  4   7  10
# 1  2  5   8  11
# 2  3  6   9  12
```

concat(): Lidando com Colunas Diferentes

Python

```python
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'B': [5, 6], 'C': [7, 8]})
# Por padrao: outer join (todas as colunas)
resultado = pd.concat([df1, df2])
print(resultado)
#      A    B    C
# 0  1.0  3.0  NaN
# 1  2.0  4.0  NaN
# 0  NaN  5.0  7.0
# 1  NaN  6.0  8.0

# Inner join (apenas colunas comuns)
resultado = pd.concat([df1, df2], join='inner')
print(resultado)  # Apenas coluna B
```

concat(): Múltiplos DataFrames

</> Python

```python
# Lista de DataFrames
dfs = []
for i in range(5): # Cria 5 DFs
    df = pd.DataFrame({
        'id': range(i*10, (i+1)*10),
        'value': np.random.randn(10)
    })
    dfs.append(df)
# Concatenar todos de uma vez
resultado = pd.concat(dfs, ignore_index=True) # Cria índice novo
print(f"Total de linhas: {len(resultado)}")  # 50
# Com identificador de origem
resultado = pd.concat(dfs, keys=['df1','df2','df3','df4','df5'])
print(resultado.index)  # MultiIndex
```

merge(): Tipos de JOIN

Tipos de JOIN (como SQL):

Tipo	Descrição
`inner`	Apenas registros com chave em ambos
`left`	Todos de left, matching de right
`right`	Todos de right, matching de left
`outer`	Todos de ambos (union)
`cross`	Produto cartesiano (todas combinações)

Padrão:

- Por padrão, `merge()` usa `how='inner'`
- Apenas linhas com chave comum em ambos os DataFrames

merge(): Inner Join

</> Python

```python
passageiros = pd.DataFrame({ # DataFrame de passageiros
    'PassengerId': [1, 2, 3, 4],
    'Name': ['John', 'Anna', 'Peter', 'Linda']
})
cabines = pd.DataFrame({ # DataFrame de cabines
    'PassengerId': [1, 2, 5, 6],
    'Cabin': ['A1', 'B2', 'C3', 'D4']
})
# Inner join (apenas IDs comuns: 1, 2)
resultado = pd.merge(passageiros, cabines,
                     on='PassengerId', how='inner')
print(resultado)
#    PassengerId   Name Cabin
# 0            1   John    A1
# 1            2   Anna    B2
```

merge(): Left Join

Python

```python
passageiros = pd.DataFrame({
    'PassengerId': [1, 2, 3, 4],
    'Name': ['John', 'Anna', 'Peter', 'Linda']
})

cabines = pd.DataFrame({
    'PassengerId': [1, 2, 5, 6],
    'Cabin': ['A1', 'B2', 'C3', 'D4']
})
```

merge(): Left Join (cont.)

</> Python

```python
# Left join (todos de passageiros)
resultado = pd.merge(passageiros, cabines,
                     on='PassengerId', how='left')
print(resultado)
#    PassengerId   Name Cabin
# 0            1   John    A1
# 1            2   Anna    B2
# 2            3  Peter   NaN
# 3            4  Linda   NaN

```

merge(): Right e Outer Join

</> Python

```python
# Right join (todos de cabines)
resultado = pd.merge(passageiros, cabines,
                     on='PassengerId', how='right')
# Resultado: PassengerId 1, 2, 5, 6
# Outer join (todos de ambos)
resultado = pd.merge(passageiros, cabines,
                     on='PassengerId', how='outer')
print(resultado)
#    PassengerId   Name Cabin
# 0            1   John    A1
# 1            2   Anna    B2
# 2            3  Peter   NaN
# 3            4  Linda   NaN
# 4            5    NaN    C3
# 5            6    NaN    D4
```

merge(): Múltiplas Chaves

Python

```python
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Date': ['2024-01-01', '2024-01-01', '2024-01-02'],
    'Value1': [10, 20, 30]
})
df2 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Date': ['2024-01-01', '2024-01-02', '2024-01-01'],
    'Value2': [100, 200, 300]
})
# Merge por multiplas colunas
resultado = pd.merge(df1, df2, on=['ID', 'Date'])
print(resultado)
# Apenas linhas com ID E Date iguais
```

merge(): Colunas com Nomes Diferentes

Python

```python
passageiros = pd.DataFrame({
    'PassengerId': [1, 2, 3],
    'Name': ['John', 'Anna', 'Peter']
})
tickets = pd.DataFrame({
    'TicketId': [1, 2, 4],
    'Price': [50, 100, 150]
})

```

merge(): Colunas com Nomes Diferentes (cont.)

Python

```python
# Especificar colunas diferentes
resultado = pd.merge(passageiros, tickets,
                     left_on='PassengerId',
                     right_on='TicketId',
                     how='left')
print(resultado)
#    PassengerId   Name  TicketId  Price
# 0            1   John       1.0   50.0
# 1            2   Anna       2.0  100.0
# 2            3  Peter       NaN    NaN
```

merge(): Sufixos para Colunas Duplicadas

</> Python

```python
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Value': [10, 20, 30],
    'Status': ['A', 'B', 'C']
})

df2 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Value': [100, 200, 300],
    'Status': ['X', 'Y', 'Z']
})

```

merge(): Sufixos para Colunas Duplicadas (cont.)

Python

```python
# Colunas 'Value' e 'Status' existem em ambos
resultado = pd.merge(df1, df2, on='ID',
                     suffixes=('_left', '_right'))
print(resultado.columns)
# ['ID','Value_left','Status_left','Value_right','Status_right']

```

merge(): Indicador de Origem

</> Python

```python
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Val': [10, 20, 30]})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Val': [200, 300, 400]})
# Adicionar indicador de origem
resultado = pd.merge(df1, df2, on='ID', how='outer',
                     indicator=True, suffixes=('_1', '_2'))
print(resultado)
#    ID  Val_1  Val_2      _merge
# 0   1   10.0    NaN   left_only
# 1   2   20.0  200.0        both
# 2   3   30.0  300.0        both
# 3   4    NaN  400.0  right_only
# Renomear indicador
resultado = pd.merge(df1, df2, on='ID', how='outer',
                     indicator='source')
# Val_1 => Val_x, Val_2 => Val_y, _merge => source
```

Merge Validation: Evitar Erros

Problema comum:

- Merge pode duplicar linhas sem avisar
- Chaves duplicadas causam explosão de dados
- Difícil detectar em datasets grandes

Solução: Parâmetro validate:

- **'1:1'** - Um para um (ambas únicas)
- **'1:m'** - Um para muitos (left única, right pode repetir)
- **'m:1'** - Muitos para um (left pode repetir, right única)
- **'m:m'** - Muitos para muitos (ambas podem repetir)

⚠ Atenção

Sempre use validate em produção - previne bugs silenciosos!

validate: Exemplo de Erro

Python

```python
# Clientes (deveria ter IDs únicos, mas tem duplicata!)
clientes = pd.DataFrame({
    'id': [1, 2, 2, 3],  # ID 2 duplicado!
    'nome': ['Ana', 'Bruno', 'Bruno Jr', 'Carlos']
})
# Pedidos
pedidos = pd.DataFrame({
    'id': [1, 2, 3],
    'valor': [100, 200, 150]
})
# SEM validação: merge aceita, mas duplica linhas!
resultado_ruim = pd.merge(clientes, pedidos, on='id')
print(len(resultado_ruim))  # 4 linhas (deveria ser 3!)
```

validate: Detectar Erro

</> Python

```python
clientes = pd.DataFrame({
    'id': [1, 2, 2, 3],
    'nome': ['Ana', 'Bruno', 'Bruno Jr', 'Carlos']
})
pedidos = pd.DataFrame({
    'id': [1, 2, 3],
    'valor': [100, 200, 150]
})
# COM validação: detecta problema!
try:
    resultado = pd.merge(clientes, pedidos, on='id',
                         validate='1:1')
except pd.errors.MergeError as e:
    print(f"ERRO DETECTADO: {e}")
    # MergeError: Merge keys are not unique in left dataset
```

validate: Casos de Uso

Python

```python
# 1:1 - Enriquecer cadastro (ambas únicas)
clientes = pd.DataFrame({'id': [1, 2, 3], 'nome': ['A', 'B', 'C']})
enderecos = pd.DataFrame({'id': [1, 2, 3], 'cidade': ['SP', 'RJ', 'MG']})
resultado = pd.merge(clientes, enderecos, on='id', validate='1:1')

# 1:m - Um cliente, múltiplos pedidos
clientes = pd.DataFrame({'id': [1, 2], 'nome': ['Ana', 'Bruno']})
pedidos = pd.DataFrame({'id': [1, 1, 2], 'valor': [100, 150, 200]})
resultado = pd.merge(clientes, pedidos, on='id', validate='1:m')

# m:1 - Múltiplos produtos, uma categoria cada
produtos = pd.DataFrame({'cat_id': [1, 1, 2], 'prod': ['A', 'B', 'C']})
categorias = pd.DataFrame({'cat_id': [1, 2], 'cat': ['Eletro', 'Livros']})
resultado = pd.merge(produtos, categorias, on='cat_id', validate='m:1')

```

validate: Best Practices

Python

```python
# SEMPRE especifique validate quando souber a relação
df1 = pd.read_csv('customers.csv')
df2 = pd.read_csv('orders.csv')
# BOM: Valida expectativa
resultado = pd.merge(
    df1, df2,
    on='customer_id',
    how='left',
    validate='1:m',  # Um cliente, múltiplos pedidos
    indicator=True   # Também adicionar indicador
)
# Verificar resultado
print(f"Total de clientes: {df1.shape[0]}")
print(f"Total após merge: {resultado.shape[0]}")
print(f"Distribuição: {resultado['_merge'].value_counts()}")
```

validate: Matriz de Decisão

Relação	validate	Exemplo
Cliente ↔ Endereço	`'1:1'`	Cadastros complementares
Categoria → Produtos	`'1:m'`	Um para muitos
Pedidos → Categoria	`'m:1'`	Muitos para um
Produtos ↔ Tags	`'m:m'`	Muitos para muitos

Quando não usar:

- Merge exploratório (ainda descobrindo relações)
- Dados sujos que precisam limpeza primeiro
- Relação é realmente m:m e você espera isso

Nota Importante

Em produção: **sempre** use validate + indicator

merge_asof(): Merge por Proximidade

O que é merge_asof()?

- ”Merge by nearest key” - chave mais próxima
- Útil para séries temporais
- Junta valores ”aproximados” em vez de exatos
- Similar ao SQL ASOF JOIN

Casos de uso:

- Juntar preços de ações com eventos de notícias
- Correlacionar logs com timestamps diferentes
- Associar medições com tempo mais próximo
- Dados financeiros (quotes + trades)

Nota Importante

Requer dados ordenados pela coluna de merge!

merge_asof(): Exemplo Básico

</> Python

```python
# Cotações de ações (atualizadas a cada minuto)
quotes = pd.DataFrame({
    'time': pd.to_datetime(['09:00', '09:01', '09:02', '09:03'], format='%H:%M'),
    'price': [100, 102, 101, 103]
})

# Trades (executadas em horários variados)
trades = pd.DataFrame({
    'time': pd.to_datetime(['09:00:30', '09:01:45', '09:02:10'], format='%H:%M:%S'),
    'volume': [50, 100, 75]
})
```

merge_asof(): Exemplo Básico

Python

```python
# Merge asof: pega preço mais recente para cada trade
resultado = pd.merge_asof(
    trades,      # left (ordenado por time)
    quotes,      # right (ordenado por time)
    on='time',   # coluna temporal
    direction='backward'  # backward, forward, ou nearest
)
```

merge_asof(): Exemplo Básico(cont.)

Python

```python
print(resultado)
#                  time  volume  price
# 0 2024-01-01 09:00:30      50    100  # Usou cotação 09:00
# 1 2024-01-01 09:01:45     100    102  # Usou cotação 09:01
# 2 2024-01-01 09:02:10      75    101  # Usou cotação 09:02

# direction='backward': pega valor anterior mais próximo
# direction='forward': pega valor posterior mais próximo
# direction='nearest': pega o mais próximo (qualquer direção)
# Com tolerância máxima
resultado_tol = pd.merge_asof(
    trades, quotes,
    on='time',
    tolerance=pd.Timedelta('1min')  # No máximo 1min de diferença
)
```

merge_asof(): Com Grupos

Python

```python
# Cotações de múltiplas ações
quotes = pd.DataFrame({
    'time': pd.to_datetime(['09:00', '09:01', '09:00', '09:01'], format='%H:%M'),
    'ticker': ['AAPL', 'AAPL', 'GOOGL', 'GOOGL'],
    'price': [150, 152, 2800, 2805]
})

# Trades
trades = pd.DataFrame({
    'time': pd.to_datetime(['09:00:30', '09:00:45'], format='%H:%M:%S'),
    'ticker': ['AAPL', 'GOOGL'],
    'volume': [100, 50]
})
```

merge_asof(): Com Grupos (cont.)

</> Python

```python
# Merge asof por ticker
resultado = pd.merge_asof(
    trades.sort_values('time'),
    quotes.sort_values('time'),
    on='time',
    by='ticker'  # Agrupar por ticker
)
print(resultado)
#                  time ticker  volume  price
# 0 1900-01-01 09:00:30   AAPL     100    150
# 1 1900-01-01 09:00:45  GOOGL      50   2800
```

explode(): Expandir Listas em Linhas

O que é explode()?

- Transforma valores tipo lista em linhas separadas
- ”Explode”uma linha em múltiplas
- Útil para dados aninhados
- Repete valores das outras colunas

Casos de uso:

- Dados de e-commerce (produtos por pedido)
- Tags de artigos/posts
- Dados JSON aninhados
- Relacionamentos muitos-para-muitos

Nota Importante

Oposto de groupby().agg(list)

explode(): Exemplo

Python

```python
# Pedidos com múltiplos produtos
pedidos = pd.DataFrame({
    'pedido_id': [1, 2, 3],
    'cliente': ['Ana', 'Bruno', 'Carlos'],
    'produtos': [
        ['Mouse', 'Teclado'],
        ['Monitor'],
        ['Webcam', 'Headset', 'Mouse']
    ]
})
print(pedidos)
#    pedido_id  cliente                  produtos
# 0          1      Ana           [Mouse, Teclado]
# 1          2    Bruno                  [Monitor]
# 2          3   Carlos  [Webcam, Headset, Mouse]
```

explode(): Exemplo (cont.)

Python

```python
# Explodir coluna de produtos
resultado = pedidos.explode('produtos')
print(resultado)
#    pedido_id  cliente produtos
# 0          1      Ana    Mouse
# 0          1      Ana  Teclado
# 1          2    Bruno  Monitor
# 2          3   Carlos   Webcam
# 2          3   Carlos  Headset
# 2          3   Carlos    Mouse
# Note: índice repete para linhas do mesmo pedido
# Resetar índice se necessário
resultado_limpo = resultado.reset_index(drop=True)
```

explode(): Múltiplas Colunas

Python

```python
# Explodir múltiplas colunas simultaneamente
pedidos_completo = pd.DataFrame({
    'pedido_id': [1, 2],
    'produtos': [['Mouse', 'Teclado'], ['Monitor']],
    'precos': [[45, 120], [890]]
})
resultado = pedidos_completo.explode(['produtos', 'precos'])
resultado_limpo = resultado.reset_index(drop=True)
print(resultado_limpo)
#    pedido_id produtos  precos
# 0          1    Mouse      45
# 1          1  Teclado     120
# 2          2  Monitor     890
```

explode(): Múltiplas Colunas

Python

```python
# Agora podemos calcular total por pedido
total = resultado.groupby('pedido_id')['precos'].sum()
print(total)
# pedido_id
# 1    165
# 2    890
Name: precos, dtype: object
```

explode(): Pipeline Completo

Python

```python
# Caso real: análise de tags de artigos
artigos = pd.DataFrame({
    'titulo': ['Post 1', 'Post 2', 'Post 3'],
    'autor': ['Ana', 'Bruno', 'Ana'],
    'tags': [
        ['python', 'data'],
        ['sql', 'data', 'database'],
        ['python', 'ml']
    ]
})
```

explode(): Pipeline Completo (cont.)

Python

```python
# Pipeline: explodir e contar
analise = (
    artigos
    .explode('tags')
    .groupby('tags')
    .agg(
        total_artigos=('titulo', 'count'),
        autores=('autor', lambda x: ', '.join(x.unique()))
    )
    .sort_values('total_artigos', ascending=False)
)
print(analise)
```

join(): Merge pelo Índice

</> Python

```python
df1 = pd.DataFrame({
    'A': [1, 2, 3]
}, index=['x', 'y', 'z'])
df2 = pd.DataFrame({
    'B': [4, 5, 6]
}, index=['x', 'y', 'w'])
resultado = df1.join(df2) # Join pelo indice (left por padrao)
print(resultado)
#    A    B
# x  1  4.0
# y  2  5.0
# z  3  NaN
# Equivalente a merge com left_index e right_index
resultado = pd.merge(df1, df2, left_index=True,
                     right_index=True, how='left')
```

Exemplo Prático: Enriquecer Dados

Python

```python
# Dataset principal
df = pd.read_csv('titanic.csv')
# Informacoes adicionais de portos
portos = pd.DataFrame({
    'Embarked': ['S', 'C', 'Q'],
    'PortoNome': ['Southampton', 'Cherbourg', 'Queenstown'],
    'Pais': ['Inglaterra', 'Franca', 'Irlanda']
})
# Adicionar informacoes dos portos
df_enriquecido = pd.merge(df, portos, on='Embarked',
                          how='left')

print(df_enriquecido[['Name','Embarked','PortoNome']].head())
```

Exemplo Prático: Combinar Dados Temporais

Python

```python
# Dados de diferentes periodos
jan = pd.DataFrame({
    'ID': [1, 2, 3],
    'VendasJan': [100, 200, 150]
})
fev = pd.DataFrame({
    'ID': [1, 2, 3],
    'VendasFev': [120, 180, 160]
})
# Combinar horizontalmente
vendas = pd.merge(jan, fev, on='ID')
# Calcular total
vendas['Total'] = vendas['VendasJan'] + vendas['VendasFev']
print(vendas)
```

Bloco 4

Reshape e Pivot

Reshape: Reorganizar Dados

Por que reestruturar dados?

- Formatos diferentes para análises diferentes
- Preparar dados para visualização
- Converter entre formato wide e long
- Criar tabelas de contingência
- Facilitar agregações específicas

Principais operações:

- **pivot_table():** Wide format (tabela de resumo)
- **melt():** Long format (unpivot)
- **stack()/unstack():** Reorganizar índices
- **crosstab():** Tabulação cruzada

Wide vs Long Format

Wide Format (formato largo):

- Uma linha por entidade
- Múltiplas colunas para valores diferentes
- Boa para leitura humana
- Exemplo: Notas de alunos (uma coluna por prova)

Long Format (formato longo):

- Múltiplas linhas por entidade
- Colunas: ID, variável, valor
- Melhor para análise e visualização
- Exemplo: Notas com colunas [Aluno, Prova, Nota]

Nota Importante

Muitas funções do Pandas preferem long format

pivot_table(): Criar Tabela de Resumo

</> Python

```python
df = pd.read_csv('titanic.csv')
# Pivot table basica: sobrevivencia por classe e sexo
pivot = pd.pivot_table(
    df,
    values='Survived',
    index='Pclass',
    columns='Sex',
    aggfunc='mean'
)
print(pivot)
# Sex       female      male
# Pclass
# 1       0.968085  0.368852
# 2       0.921053  0.157407
# 3       0.500000  0.135447
```

pivot_table(): Múltiplas Agregações

Python

```python
df = pd.read_csv('titanic.csv')
pivot = pd.pivot_table(  # Multiplas funcoes de agregacao
    df,
    values='Fare',
    index='Pclass',
    columns='Sex',
    aggfunc=['mean', 'median', 'count']
)
print(pivot)
# MultiIndex em colunas
# mean            median          count
# Sex  female male female male female male
# ...
# Achatar colunas
pivot.columns = ['_'.join(col) for col in pivot.columns]
```

pivot_table(): Múltiplos Valores

</> Python

```python
df = pd.read_csv('titanic.csv')
pivot = pd.pivot_table(   # Agregar multiplas colunas
    df,
    values=['Survived', 'Age'],
    index='Pclass',
    columns='Sex',
    aggfunc='mean'
)
print(pivot)
#               Age              Survived
# Sex        female      male    female      male
# Pclass
# 1          34.61    41.28       0.968     0.369
# 2          28.72    30.74       0.921     0.157
# 3          21.75    26.51       0.500     0.135
```

pivot_table(): Totais (Margins)

Python

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

# Adicionar linha/coluna de totais
pivot = pd.pivot_table(
    df,
    values='Survived',
    index='Pclass',
    columns='Sex',
    aggfunc='mean',
    margins=True,
    margins_name='Total'
)

```

pivot_table(): Totais (Margins) (cont.)

</> Python

```python
print(pivot)
# Sex        female      male     Total
# Pclass
# 1        0.968085  0.368852  0.629630
# 2        0.921053  0.157407  0.472826
# 3        0.500000  0.135447  0.242363
# Total    0.742038  0.188908  0.383838

```

pivot_table(): Preencher Valores Ausentes

Python

```python
df = pd.read_csv('titanic.csv')
pivot = pd.pivot_table( # Algumas combinacoes podem nao existir
    df,
    values='Fare',
    index='Pclass',
    columns='Embarked',
    aggfunc='mean',
    fill_value=0  # Preencher NaN com 0
)
print(pivot)
# Embarked           C          Q          S
# Pclass
# 1         104.718529  90.000000  70.364862
# 2          25.358335  12.350000  20.327439
# 3          11.214083  11.183393  14.644083
```

pivot_table(): Preencher Valores Ausentes (cont.)

Python

```python
# Alternativa: dropna
pivot = pd.pivot_table(
    df,
    values='Fare',
    index='Pclass',
    columns='Embarked',
    aggfunc='mean',
    dropna=False
)
print(pivot)
# Embarked           C          Q          S    NaN
# Pclass
# 1         104.718529  90.000000  70.364862  80.0
# 2          25.358335  12.350000  20.327439   NaN
# 3          11.214083  11.183393  14.644083   NaN
```

crosstab(): Tabulação Cruzada

</> Python

```python
df = pd.read_csv('titanic.csv')
# Tabela de frequencias
tabela = pd.crosstab(df['Pclass'], df['Sex'])
print(tabela)
# Sex     female  male
# Pclass
# 1           94   122
# 2           76   108
# 3          144   347
# Com percentuais
tabela_pct = pd.crosstab(df['Pclass'], df['Sex'],
                         normalize='index')
print(tabela_pct)
# Proporcoes por linha (soma = 1.0 em cada linha)
```

crosstab(): Com Valores

Python

```python
df = pd.read_csv('titanic.csv')
# Crosstab com valores agregados
tabela = pd.crosstab(
    df['Pclass'],
    df['Sex'],
    values=df['Fare'],
    aggfunc='mean'
)
print(tabela)
# Tarifa media por classe e sexo, single index
```

crosstab(): Com Valores (cont.)

Python

```python
# Com multiplas funcoes
tabela = pd.crosstab(
    df['Pclass'],
    df['Sex'],
    values=df['Fare'],
    aggfunc=['mean', 'median', 'count'] #MultiIndex
)
print(tabela)
#                mean                median            count
# Sex          female       male    female     male female male
# Pclass
# 1       106.125798  67.226127  82.66455  41.2625     94  122
# 2        21.970121  19.741782  22.00000  13.0000     76  108
# 3        16.118810  12.661633  12.47500   7.9250    144  347
```

crosstab(): Múltiplas Variáveis

</> Python

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

# Crosstab com multiplas variaveis em linhas/colunas
tabela = pd.crosstab(
    [df['Pclass'], df['Sex']],
    [df['Survived'], df['Embarked']]
)
print(tabela)
# MultiIndex em linhas e colunas

```

crosstab(): Múltiplas Variáveis (cont.)

Python

```python
# Com totais
tabela = pd.crosstab(
    df['Pclass'],
    df['Sex'],
    margins=True
)
print(tabela)
# Linha e coluna 'All' com totais

```

melt(): Wide to Long (Unpivot)

</> Python

```python
# Dados wide
df_wide = pd.DataFrame({
    'Name': ['Ana', 'Bruno', 'Carlos'],
    'Math': [85, 90, 78],
    'Physics': [88, 92, 80],
    'Chemistry': [90, 85, 82]
})
print(df_wide)
#       Name  Math  Physics  Chemistry
# 0      Ana    85       88         90
# 1    Bruno    90       92         85
# 2   Carlos    78       80         82
```

melt(): Wide to Long (Unpivot) (cont.)

Python

```python
# Converter para long
df_long = pd.melt(
    df_wide,
    id_vars=['Name'],
    var_name='Subject',
    value_name='Grade'
)
print(df_long)
#       Name    Subject  Grade
# 0      Ana       Math     85
# 1    Bruno       Math     90
# 2   Carlos       Math     78
# 3      Ana    Physics     88
# ...
```

melt(): Selecionar Colunas

</> Python

```python
df_wide = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Ana', 'Bruno', 'Carlos'],
    'Q1': [100, 150, 120],
    'Q2': [110, 160, 130],
    'Q3': [105, 155, 125],
    'Q4': [115, 165, 135]
})
df_long = pd.melt(  # Melt apenas colunas de trimestres
    df_wide,
    id_vars=['ID', 'Name'],
    value_vars=['Q1', 'Q2', 'Q3', 'Q4'],
    var_name='Quarter',
    value_name='Sales'
)
```

stack() e unstack(): Reorganizar Níveis

</> Python

```python
# DataFrame com MultiIndex
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]
}, index=pd.MultiIndex.from_tuples([
    ('X', 'a'), ('X', 'b'), ('Y', 'a'), ('Y', 'b')
]))

# Unstack: mover nivel do indice para colunas
df_unstacked = df.unstack()
print(df_unstacked)
#       A       B
#       a   b   a   b
# X     1   2   5   6
# Y     3   4   7   8
```

stack() e unstack(): Reorganizar Níveis (cont.)

Python

```python
# Stack: mover nivel das colunas para indice
df_stacked = df_unstacked.stack()
print(df_stacked)
#       A  B
# X a   1  5
#   b   2  6
# Y a   3  7
#   b   4  8
```

stack() e unstack(): Exemplo Prático

</> Python

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

# Criar pivot simples
pivot = df.pivot_table(
    values='Survived',
    index='Pclass',
    columns='Sex',
    aggfunc='mean'
)

```

stack() e unstack(): Exemplo Prático (cont.)

Python

```python
# Unstack: transformar colunas em linhas
unstacked = pivot.unstack()
print(unstacked)
# Sex      Pclass
# female   1         0.968085
#          2         0.921053
#          3         0.500000
# male     1         0.368852
#          2         0.157407
#          3         0.135447

```

Tratamento de Duplicatas

Python

```python
df = pd.read_csv('titanic.csv')
# Verificar duplicatas
duplicatas = df.duplicated()
print(f"Linhas duplicadas: {duplicatas.sum()}")
# Duplicatas em colunas especificas
dup_nome = df.duplicated(subset=['Name'])
print(f"Nomes duplicados: {dup_nome.sum()}")
# Ver as duplicatas
df_duplicados = df[df.duplicated(keep=False)]
# Manter apenas primeira ocorrencia
df_limpo = df.drop_duplicates()
# Manter ultima ocorrencia
df_limpo = df.drop_duplicates(keep='last')
```

drop_duplicates(): Colunas Específicas

</> Python

```python
df = pd.read_csv('titanic.csv')
# Remover duplicatas baseado em colunas especificas
# Manter primeiro passageiro de cada combinacao classe+sexo
df_unico = df.drop_duplicates(subset=['Pclass', 'Sex'])
print(f"Original: {len(df)}, Unico: {len(df_unico)}")
# Original: 891, Unico: 6
# Verificar combinacoes unicas
combinacoes = df[['Pclass', 'Sex']].drop_duplicates()
print(combinacoes)
# Indice, Pclass, Sex - indices nao-consecutivos
# Contar ocorrencias de cada combinacao
contagem = df.groupby(['Pclass', 'Sex']).size()
print(contagem)
# Numero de ocorrencias de cada combinacao
```

Exemplo Completo: Pipeline de Transformação

</> Python

```python
df = pd.read_csv('titanic.csv')
# 1. Remover duplicatas
df = df.drop_duplicates()
# 2. Criar novas features
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
df['IsAlone'] = (df['FamilySize'] == 1).astype(int)
df['Titulo'] = df['Name'].str.extract(r', ([^.]+)\.')
# 3. Agrupar titulos raros
titulos_comuns = ['Mr', 'Miss', 'Mrs', 'Master']
df['TituloGroup'] = df['Titulo'].apply(
    lambda x: x if x in titulos_comuns else 'Other'
)
```

Exemplo Completo: Pipeline (cont.)

Python

```python
# 4. Preencher valores faltantes por grupo
df['Age'] = df.groupby('TituloGroup')['Age'].transform(
    lambda x: x.fillna(x.median())
)
# 5. Criar tabela de resumo
resumo = pd.pivot_table(
    df,
    values=['Survived', 'Age', 'Fare'],
    index='Pclass',
    columns='Sex',
    aggfunc={'Survived': 'mean',
             'Age': 'median',
             'Fare': 'mean'}
)
```

Exemplo Completo: Pipeline (cont.)

Python

```python
print(resumo)
#            Age                Fare              Survived
# Sex     female   male       female        male    female      male
# Pclass
# 1         35.0   36.5   106.125798   67.226127  0.968085  0.368852
# 2         28.0   30.0    21.970121   19.741782  0.921053  0.157407
# 3         21.0   30.0    16.118810   12.661633  0.500000  0.135447
```

Exemplo Prático: Análise de Família

Python

```python
df = pd.read_csv('titanic.csv')
# Criar features de familia
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
df['FamilyType'] = pd.cut(
    df['FamilySize'],
    bins=[0, 1, 4, 20],
    labels=['Alone', 'Small', 'Large']
)
# Analise por tipo de familia
analise = df.groupby('FamilyType', observed=False).agg({
    'Survived': ['count', 'sum', 'mean'],
    'Age': 'mean',
    'Fare': 'mean'
})
print(analise)
```

Exemplo Prático: Enriquecer com Estatísticas

Python

```python
df = pd.read_csv('titanic.csv')
# Adicionar estatisticas do grupo como novas colunas
df['AgeMeanClass'] = df.groupby('Pclass')['Age'].transform('mean')
df['AgeStdClass'] = df.groupby('Pclass')['Age'].transform('std')
# Z-score normalizado por classe
df['AgeZScore'] = (df['Age'] - df['AgeMeanClass']) / \
                  df['AgeStdClass']
# Ranking de tarifa dentro da classe
df['FareRank'] = df.groupby('Pclass')['Fare'].rank(
    ascending=False
)
print(df[['Name','Pclass','Age','AgeMeanClass','AgeZScore']].head())
# Adiciona colunas com estatisticas (repetidas) para cada linha
```

Exemplo Prático: Relatório Multi-nível

</> Python

```python
df = pd.read_csv('titanic.csv')
# Criar relatorio complexo
relatorio = df.groupby(['Pclass', 'Sex', 'Embarked']).agg({
    'PassengerId': 'count',
    'Survived': ['sum', 'mean'],
    'Age': ['mean', 'median'],
    'Fare': ['mean', 'max']
})
# Achatar MultiIndex de colunas
relatorio.columns = ['_'.join(col) for col in relatorio.columns]
# Resetar indice para facilitar visualizacao
relatorio = relatorio.reset_index()
```

Exemplo Prático: Relatório Multi-nível (cont.)

Python

```python
print(relatorio.columns)
# Index(['Pclass', 'Sex', 'Embarked', 'PassengerId_count', 'Survived_sum',
#        'Survived_mean', 'Age_mean', 'Age_median', 'Fare_mean', 'Fare_max
    '],
#       dtype='object')
```

Method Chaining: Código Limpo e Legível

O que é Method Chaining?

- Encadear operações em sequência
- Cada método retorna DataFrame/Series
- Uma operação por linha
- Estilo ”fluent interface”

Vantagens:

- Código mais legível (lê-se como pipeline)
- Menos variáveis intermediárias
- Fácil adicionar/remover etapas
- Documentação clara do fluxo
- Evita modificações acidentais

Nota Importante

Estilo recomendado para código de produção

Sem *vs* Com Method Chaining

</> Python

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

# SEM method chaining (múltiplas variáveis)
df_limpo = df.dropna(subset=['Age'])
df_filtrado = df_limpo[df_limpo['Pclass'] == 1]
df_com_features = df_filtrado.copy()
df_com_features['FamilySize'] = df_com_features['SibSp'] + df_com_features
    ['Parch'] + 1
df_ordenado = df_com_features.sort_values('Fare', ascending=False)
resultado = df_ordenado.head(10)
```

Sem *vs* Com Method Chaining

</> Python

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

# COM method chaining (pipeline claro)
resultado = (
    df
    .dropna(subset=['Age'])
    .query('Pclass == 1')
    .assign(FamilySize=lambda x: x['SibSp'] + x['Parch'] + 1)
    .sort_values('Fare', ascending=False)
    .head(10)
)
```

assign(): Criar Colunas no Pipeline

Python

```python
df = pd.read_csv('titanic.csv')
# assign() retorna cópia com novas colunas
resultado = (
    df
    .assign(
        FamilySize=lambda x: x['SibSp'] + x['Parch'] + 1, # n colunas
        IsAlone=lambda x: x['FamilySize'] == 1,
        FamilyType=lambda x: pd.cut( # Usar coluna recém-criada
            x['FamilySize'],
            bins=[0, 1, 4, 20],
            labels=['Alone', 'Small', 'Large']
        )
    )
    .head()
)
```

assign(): Criar Colunas no Pipeline (cont.)

Python

```python
print(resultado[['Name', 'FamilySize', 'IsAlone', 'FamilyType']])
#                Name  FamilySize  IsAlone FamilyType
# 0 Braund, Mr. Owen           2    False      Small
# 1 Cumings, Mrs.[...]         2    False      Small
# 2 Heikkinen, [...]           1     True      Alone
# 3 Futrelle, [...]            2    False      Small
# 4 Allen, Mr. [...]           1     True      Alone
```

pipe(): Funções Customizadas no Pipeline

</> Python

```python
# Definir funções de transformação
def limpar_dados(df):
    return df.dropna(subset=['Age', 'Embarked'])

def criar_features(df):
    return df.assign(
        FamilySize=lambda x: x['SibSp'] + x['Parch'] + 1,
        AgeGroup=lambda x: pd.cut(x['Age'], bins=[0, 18, 60, 100],
                                  labels=['Child', 'Adult', 'Senior'])
    )

def filtrar_primeira_classe(df):
    return df.query('Pclass == 1')
```

pipe(): Funções Customizadas no Pipeline (cont.)

Python

```python
# Pipeline com pipe()
resultado = (
    pd.read_csv('titanic.csv')
    .pipe(limpar_dados)
    .pipe(criar_features)
    .pipe(filtrar_primeira_classe)
    .head()
)
```

Pipeline Completo: ETL

</> Python

```python
def processar_titanic(filepath):
    return (
        # EXTRACT
        pd.read_csv(filepath)
        # TRANSFORM - Limpeza
        .dropna(subset=['Embarked'])
        .drop_duplicates()
        .assign(Age=lambda x: x.groupby('Pclass')['Age']
                              .transform(lambda y: y.fillna(y.median())))
        # TRANSFORM - Features
        .assign(
            FamilySize=lambda x: x['SibSp'] + x['Parch'] + 1,
            Title=lambda x: x['Name'].str.extract(r', ([^.]+)\.')[0],
            AgeGroup=lambda x: pd.cut(x['Age'], bins=[0,18,60,100],
                                      labels=['Child','Adult','Senior'])
        )
```

Pipeline Completo: ETL (cont.)

Python

```python
        # TRANSFORM - Tipos
        .astype({
            'Pclass': 'category',
            'Sex': 'category',
            'Embarked': 'category',
            'AgeGroup': 'category'
        })
        # TRANSFORM - Ordenação
        .sort_values(['Pclass', 'Fare'], ascending=[True, False])
        .reset_index(drop=True)
    )
# Executar pipeline
df_processado = processar_titanic('titanic.csv')
# Pipeline é reutilizável e testável!
```

Debugging em Pipelines

</> Python

```python
# Técnica 1: Comentar etapas temporariamente
resultado = (
    df
    .dropna()
    .query('Age > 18')
    # .assign(NewCol=lambda x: x['A'] * 2)  # Comentar para debug
    .head()
)
```

Debugging em Pipelines

Python

```python
# Técnica 2: Quebrar pipeline com variável intermediária
temp = (
    df
    .dropna()
    .query('Age > 18')
)
print(temp.shape)  # Verificar até aqui
resultado = temp.assign(NewCol=lambda x: x['A'] * 2)
```

Debugging em Pipelines

Python

```python
# Técnica 3: Usar pipe() para debug
def debug_shape(df):
    print(f"Shape atual: {df.shape}")
    return df

resultado = df.pipe(debug_shape).dropna().pipe(debug_shape)
```

Method Chaining: Best Practices

Regras de ouro:

1. **Uma operação por linha**
   - Facilita leitura e debug
2. **Use parênteses externos**
   - Permite quebra de linha sem \
3. **Indentação consistente**
   - 4 espaços para cada nível
4. **assign() para novas colunas**
   - Em vez de df['col'] = valor
5. **query() para filtros simples**
   - Mais legível que boolean indexing
6. **pipe() para funções complexas**
   - Mantém pipeline fluido

Exemplo Real: Análise de Vendas

Python

```python
# Pipeline completo de análise
analise_vendas = (
    pd.read_csv('vendas.csv')
    # Limpeza
    .dropna(subset=['valor', 'data'])
    .drop_duplicates(subset=['pedido_id'])
    # Datas
    .assign(data=lambda x: pd.to_datetime(x['data']))
    .assign(
        mes=lambda x: x['data'].dt.to_period('M'),
        dia_semana=lambda x: x['data'].dt.day_name()
    )
```

Exemplo Real: Análise de Vendas (cont.)

Python

```python
    .assign(
        receita=lambda x: x['quantidade'] * x['preco'],
        margem=lambda x: x['receita'] - x['custo']
    )
    # Agregação
    .groupby(['mes', 'categoria'])
    .agg(
        receita_total=('receita', 'sum'),
        margem_media=('margem', 'mean'),
        num_pedidos=('pedido_id', 'count')
    )
    .round(2)
)
```

Performance: Otimização de Código Pandas

Hierarquia de performance (do mais rápido ao mais lento):

1. **Operações vetorizadas** (100x - 1000x mais rápido)
   - `df['col'] * 2`, `df['A'] + df['B']`
2. **Built-in Pandas functions** (10x - 100x)
   - `.groupby()`, `.merge()`, `.query()`
3. **apply() com funções NumPy** (5x - 10x)
   - `df.apply(np.mean)`
4. **apply() com Python puro** (2x - 5x)
   - `df['col'].apply(lambda x: x + 1)`
5. **Loops explícitos** (NUNCA USE!)
   - `for i in range(len(df)): ...`

Performance: Exemplo Comparativo

Python

```python
import time
df = pd.DataFrame({'valor': range(1000000)})
# LENTO: Loop explícito (NUNCA faça isso!)
start = time.time()
resultado = []
for i in range(len(df)):
    resultado.append(df.iloc[i]['valor'] * 2)
df['loop'] = resultado
print(f"Loop: {time.time() - start:.2f}s")  # ~5 segundos
```

Performance: Exemplo Comparativo (cont.)

Python

```python
# MÉDIO: apply()
start = time.time()
df['apply'] = df['valor'].apply(lambda x: x * 2)
print(f"Apply: {time.time() - start:.2f}s")  # ~0.5 segundos

# RÁPIDO: Vetorizado
start = time.time()
df['vetorizado'] = df['valor'] * 2
print(f"Vetorizado: {time.time() - start:.2f}s")  # ~0.005s
```

Performance: Uso de Memória

</> Python

```python
df = pd.read_csv('titanic.csv')
# Verificar uso de memória
print(df.memory_usage(deep=True))
# Index          128
# PassengerId   7128
# Name         52194  <- Strings pesam muito!
# Sex           5118
# ...
```

Performance: Uso de Memória (cont.)

</> Python

```python
# ANTES: object dtype
print(f"Memória Sex: {df['Sex'].memory_usage(deep=True)} bytes")
# 52500 bytes

# DEPOIS: category dtype
df['Sex'] = df['Sex'].astype('category')
print(f"Memória Sex: {df['Sex'].memory_usage(deep=True)} bytes")
# 1019 bytes (50x menor!)
```

Performance: Dtypes Apropriados

</> Python

```python
df = pd.read_csv('titanic.csv')
# Reduzir tamanho de inteiros/floats
print(df['PassengerId'].dtype)  # int64
df['PassengerId'] = df['PassengerId'].astype('int16')  # 0-32767
print(df['Age'].dtype)  # float64
df['Age'] = df['Age'].astype('float32')
# Converter colunas repetitivas para category
for col in ['Pclass', 'Sex', 'Embarked', 'Survived']:
    df[col] = df[col].astype('category')
# Comparar memória
print("\nEconomia de memória:")
original = pd.read_csv('titanic.csv')
print(f"Original: {original.memory_usage(deep=True).sum() / 1024:.1f} KB")
print(f"Otimizado: {df.memory_usage(deep=True).sum() / 1024:.1f} KB")
```

Performance: Filtrar Antes de Agrupar

Python

```python
df = pd.read_csv('huge_dataset.csv')  # 10 milhões de linhas
# LENTO: Agrupar tudo, depois filtrar
resultado_lento = (
    df
    .groupby('categoria')['valor'].sum()
    .loc[lambda x: x > 1000]  # Filtra depois
)
# RÁPIDO: Filtrar primeiro, depois agrupar
resultado_rapido = (
    df
    .query('valor > 10')  # Reduz dados primeiro
    .groupby('categoria')['valor'].sum()
    .loc[lambda x: x > 1000]
)
# Reduz de 10M para 1M linhas ANTES de agrupar
# Economia: ~10x mais rápido
```

Performance: Profiling com %%timeit

Python

```python
# No Jupyter/Colab: usar %%timeit para medir performance
%%timeit
df['resultado'] = df['A'] + df['B']
# 1.23 ms ± 45.2 µs per loop (mean ± std. dev. of 7 runs)
%%timeit
df['resultado'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
# 456 ms ± 12.3 ms per loop
# -> 370x mais lento!
# Comparar abordagens
%timeit resultado1 = df['A'] * 2  # Vetorizado
%timeit resultado2 = df['A'].apply(lambda x: x * 2)  # Apply
%timeit resultado3 = [x * 2 for x in df['A']]  # List comp
# Regra: Sempre teste alternativas em dados grandes!
```

Performance: Checklist de Otimização

Antes de otimizar:

1. **Meça primeiro!** Use `%%timeit` ou `time.time()`
2. Otimize apenas gargalos reais (regra 80/20)

Otimizações de alto impacto:

- ✓ Vetorizar em vez de apply() ou loops
- ✓ Usar dtypes apropriados (category, int32)
- ✓ Filtrar dados antes de operações pesadas
- ✓ Usar query() em vez de boolean indexing complexo
- ✓ Usar inplace=True com cuidado (economiza memória)

Trade-offs:

- Legibilidade vs Performance: Priorize legibilidade até medir gargalo
- Memória vs Velocidade: Categoricals economizam memória, podem ser mais lentos

Dicas Finais: Pandas em Produção

Boas práticas essenciais:

1. **Validação de dados**
   - Sempre verifique shape após operações
   - Use `validate` em merges
   - Adicione `indicator=True` para rastrear
2. **Tratamento de erros**
   - Try-except em leituras de arquivo
   - Validar dtypes esperados
   - Verificar valores faltantes críticos
3. **Documentação**
   - Docstrings em funções de pipeline
   - Comentar transformações complexas
   - Registrar decisões de limpeza
4. **Testes**
   - Testes unitários para funções de transformação
   - Verificar invariantes (ex: sum antes/depois)

Dicas Finais: Debugging Comum

Python

```python
# 1. KeyError: coluna não existe
try:
    df['coluna_inexistente']
except KeyError:
    print(f"Colunas disponíveis: {df.columns.tolist()}")

# 2. SettingWithCopyWarning
# Evite: df_subset['nova'] = valor
# Use: df.loc[mask, 'nova'] = valor

# 3. Merge duplicando linhas
resultado = pd.merge(df1, df2, on='id', validate='1:1')
```

Dicas Finais: Debugging Comum (cont.)

Python

```python
# 4. GroupBy perdendo dados
antes = len(df)
depois = len(df.groupby('cat').size().sum())
assert antes == depois, "GroupBy perdeu linhas!"

# 5. Tipos incorretos após merge
df['id'] = df['id'].astype(int)  # Forçar tipo correto

```

Recursos para Aprofundamento

Documentação oficial:

- User Guide: `https://pandas.pydata.org/docs/user_guide/`
- API Reference: Consulte sempre que precisar
- Performance: Enhancement proposals (PDEP)

Livros recomendados:

- ”Python for Data Analysis- Wes McKinney (criador do Pandas)
- ”Pandas 2.x Cookbook- Matt Harrison
- ”Effective Pandas- Matt Harrison

Prática:

- 100 Pandas Puzzles (GitHub)
- Kaggle: Datasets + Notebooks da comunidade
- Real Python: Tutoriais práticos

Demonstração Prática

Live Coding

Vamos praticar juntos:

1. Transformações com apply() e map()
2. Operações com strings (str accessor)
3. GroupBy com múltiplas agregações
4. Merge de DataFrames complementares
5. Criação de pivot tables
6. Pipeline completo de transformação

Revisão da Aula

Bloco 1 - Transformações:

- Adicionar/remover colunas
- apply(), map() para transformações customizadas
- String methods (str accessor)
- Conversão de tipos: astype(), to_numeric()
- Categorical para economia de memória

Bloco 2 - GroupBy:

- Split-apply-combine paradigm
- Agregações: agg(), sum(), mean(), count()
- transform() vs aggregate()
- Múltiplas chaves de agrupamento
- Funções customizadas

Revisão da Aula (cont.)

Bloco 3 - Combinação:

- concat(): empilhamento vertical/horizontal
- merge(): JOIN estilo SQL (inner, left, right, outer)
- Múltiplas chaves e colunas diferentes
- join(): merge pelo índice
- Sufixos e indicadores de origem

Bloco 4 - Reshape:

- pivot_table(): criar tabelas de resumo
- crosstab(): tabulação cruzada
- melt(): wide to long format
- stack()/unstack(): reorganizar níveis
- Tratamento de duplicatas

Conceitos-Chave para Lembrar

1. **Prefira vetorização** a apply() quando possível
2. **str accessor** para operações com strings
3. **GroupBy = Split-Apply-Combine**
4. **transform()** mantém shape, agg() reduz
5. **merge()** para relacionamentos, concat() para empilhamento
6. **Inner join** é padrão no merge()
7. **pivot_table()** cria resumos agregados
8. **melt()** converte wide $\rightarrow$ long
9. **Method chaining** torna pipelines legíveis

Performance: Dicas Importantes

Otimização de código Pandas:

1. **Vetorização } apply() > loops**
   - Use operações vetorizadas sempre que possível
   - apply() apenas quando necessário
   - NUNCA use loops explícitos
2. **Categorical para colunas repetitivas**
   - Economiza memória significativamente
   - Acelera operações de grupo
3. **Use dtypes apropriados**
   - int32 vs int64, float32 vs float64
   - Pode reduzir uso de memória pela metade
4. **Filter antes de groupby**
   - Reduza dados antes de agrupar
   - Menos dados = operações mais rápidas

Workflow Típico de Manipulação

Pipeline padrão:

1. **Carregar:** read_csv(), read_excel()
2. **Limpar:** drop_duplicates(), dropna() ou fillna()
3. **Transformar:** criar features, apply(), map()
4. **Enriquecer:** merge() com dados adicionais
5. **Agrupar:** groupby() para análises agregadas
6. **Pivotar:** pivot_table() para resumos
7. **Visualizar:** (próxima aula)
8. **Exportar:** to_csv(), to_excel()

Nota Importante

Use method chaining para pipelines mais legíveis

Boas Práticas

Ao manipular dados:

1. **Documente transformações**
   - Comente o ”porquê”, não apenas o ”o quê”
   - Use nomes descritivos de variáveis
2. **Verifique resultados intermediários**
   - Use head(), shape, info() frequentemente
   - Confirme que transformações funcionaram
3. **Mantenha dados originais**
   - Trabalhe com cópias: df_work = df.copy()
   - Não modifique dados brutos
4. **Valide agregações**
   - Verifique totais e contagens
   - Confirme que groupby não perdeu dados

Erros Comuns e Como Evitar

1. **GroupBy sem agregação**
   - `df.groupby('col')` apenas cria objeto
   - Precisa de .agg(), .mean(), etc.
2. **Merge sem especificar 'on'**
   - Pode usar colunas erradas automaticamente
   - Sempre especifique explicitamente
3. **Confundir concat com merge**
   - concat = empilhar, merge = JOIN
   - Use cada um para o propósito certo
4. **Esquecer de resetar índice após operações**
   - groupby, sort_values deixam índice bagunçado
   - Use reset_index() quando necessário

Próxima Aula: EDA - Parte 1

Aula 07 - O que vem:

- Estatística descritiva completa
- Correlação e relações entre variáveis
- Identificação de outliers
- Visualização com Matplotlib
- Múltiplos gráficos e layouts
- Dashboard simples

Preparação:

- Revisar conceitos de estatística básica
- Praticar transformações desta aula
- Explorar Iris dataset (próxima aula)
- Pensar em perguntas analíticas

Dicas de Estudo

Para dominar manipulação de dados:

1. **Pratique com datasets diversos**
   - Cada dataset tem desafios únicos
   - Kaggle, UCI ML Repository
2. **Crie seus próprios pipelines**
   - Do início ao fim: carregar $\rightarrow$ transformar $\rightarrow$ analisar
   - Documente cada etapa
3. **Estude exemplos reais**
   - Notebooks no Kaggle
   - Projetos open source
4. **Domine groupby**
   - É uma das operações mais poderosas
   - Pratique diferentes agregações

Recursos Adicionais

Documentação:

- Pandas User Guide: Group By
- Pandas Cookbook
- Stack Overflow: tag [pandas]

Tutoriais:

- Real Python: Pandas GroupBy
- Kaggle Learn: Data Cleaning
- DataCamp: Merging DataFrames

Prática:

- 100 pandas puzzles (GitHub)
- Exercícios do Kaggle
- Projetos pessoais

Preparação para Entrega Quinzenal

Segunda entrega (31/10) - Status:

- **Aula 5:** Pandas básico ✓
- **Aula 6:** Manipulação ✓(hoje)
- **Aula 7:** EDA Parte 1 (próxima)
- **Aula 8:** EDA Parte 2

Tempo de prática em aula:

- Semana 6: 1 hora (após Aula 7)
- Semana 8: 1 hora (após Aula 8) + entrega

Lembrete:

- Trabalho envolve Titanic E Wine Quality
- Feature engineering será importante
- Comece a pensar nas análises

Resumo: Poder do Pandas

O que conseguimos fazer agora:

`Dados Brutos`
↓ *transformar*
`Features Engenheiradas`
↓ *agrupar*
`Insights Agregados`
↓ *combinar*
`Datasets Enriquecidos`
↓ *pivotar*
`Tabelas de Resumo`
↓ *visualizar*
`Análises Completas` (próximas aulas)

A Seguir: Aula Prática

Exercício Prático

Tempo: 60 minutos
Entrega: via Moodle (notebook)
Dataset: titanic.csv
Tarefas:

1. (atualizado durante a aula)

Notebook: Disponível no Moodle

Obrigado!

Próxima aula: Análise Exploratória de Dados - Parte 1
Terça-feira, 28/10

Dúvidas: via Moodle ou atendimento