# Aula 10 – Agrupamento e Agregação de Dados com groupby

No vasto universo da análise de dados, raramente nos deparamos com informações que já vêm prontas para serem interpretadas. Pelo contrário, a realidade é que a maioria dos conjuntos de dados são brutos, desorganizados e, à primeira vista, pouco reveladores. Imagine ter uma planilha com milhares de vendas individuais de uma empresa: olhar para cada linha isoladamente seria como tentar entender uma floresta observando apenas uma folha. Para realmente compreender o cenário, precisamos de uma forma de organizar e resumir essas informações, revelando padrões e tendências que, de outra forma, permaneceriam ocultos.

É exatamente nesse ponto que o agrupamento e a agregação de dados se tornam ferramentas indispensáveis. Eles nos permitem transformar um mar de detalhes em um oceano de insights, consolidando informações de maneira significativa. Ao final desta aula, você não apenas entenderá os conceitos fundamentais por trás dessas operações, mas também será capaz de aplicá-los com maestria usando a biblioteca Pandas em Python, uma habilidade crucial para qualquer profissional de dados.

Objetivos de Aprendizagem

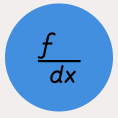
# O Que Você Vai Dominar

## Paradigma Split-Apply-Combine

Compreender a filosofia fundamental por trás do agrupamento de dados e como ela estrutura a análise categórica.

## Método .groupby()

Dominar o uso do poderoso método do Pandas para dividir dados em categorias lógicas e preparar agregações.



## Funções de Agregação

Aplicar sum, mean, count, max e min para extrair métricas valiosas de cada grupo de dados.

## Agregações Múltiplas

Combinar múltiplas funções e colunas para análises ricas e detalhadas que revelam insights profundos.

# A Filosofia "Split-Apply-Combine": O Coração do Agrupamento

Quando nos deparamos com um grande volume de dados, a tarefa de extrair informações úteis pode parecer esmagadora. Pense, por exemplo, em uma empresa de e-commerce que registra cada venda, cada cliente, cada produto. Se quisermos saber qual produto gerou mais receita em cada região, ou qual a média de idade dos clientes que compram um determinado tipo de item, olhar para a tabela inteira de vendas não nos dará a resposta de forma imediata. Precisamos de uma estratégia para organizar essa massa de informações.

É aqui que entra o paradigma **"Split-Apply-Combine"** (Dividir-Aplicar-Combinar), um conceito elegante e extremamente poderoso que forma a base de muitas operações de agrupamento em análise de dados, especialmente no Pandas. Ele descreve um processo intuitivo de três etapas que nos permite resolver problemas complexos de agregação de forma estruturada. Compreender essa filosofia é o primeiro passo para dominar o groupby.

### Analogia Prática

Imagine que você tem uma cesta de frutas variadas e precisa saber o peso total de cada tipo de fruta. Você não pesaria todas as frutas juntas e depois tentaria adivinhar. Em vez disso, você primeiro **dividiria** as frutas por tipo (maçãs, bananas, laranjas). Em seguida, para cada pilha de frutas, você **aplicaria** a operação de pesagem (somando o peso de cada fruta na pilha). Finalmente, você **combinaria** os resultados, apresentando o peso total de maçãs, o peso total de bananas, e assim por diante. Essa é a essência do Split-Apply-Combine.

# Entendendo o "Split": Dividindo para Conquistar

A primeira etapa do nosso processo, o "Split" ou "Dividir", é fundamental para qualquer análise categórica. Antes de podermos calcular médias, somas ou contagens para diferentes grupos, precisamos, logicamente, definir e separar esses grupos. No contexto do Pandas, isso significa pegar um DataFrame completo e, com base em uma ou mais colunas, particioná-lo em sub-DataFrames menores, onde cada um desses sub-DataFrames representa uma categoria única.

01

### Identificação de Categorias

O Pandas percorre a coluna especificada e identifica todos os valores únicos que existem nela.

02

### Criação de Grupos Virtuais

Para cada valor único encontrado, o Pandas cria um "grupo" virtual que conterá todas as linhas correspondentes.

03

### Organização Interna

Os dados originais não são fisicamente separados, mas organizados internamente para operações subsequentes.

> *"Pense em um professor que tem as notas de todos os seus alunos em uma única planilha. Se ele quiser calcular a média de notas por turma, ele não pode simplesmente calcular a média geral de todos os alunos. Ele precisa primeiro 'dividir' a planilha em grupos, um para cada turma. Cada grupo de alunos de uma mesma turma será tratado de forma independente nas próximas etapas."*

No Pandas, essa divisão é orquestrada principalmente pelo método `.groupby()`. Quando você especifica uma coluna (ou uma lista de colunas) para o .groupby(), o Pandas percorre essa coluna e identifica todos os valores únicos. Para cada valor único, ele cria um "grupo" virtual que contém todas as linhas do DataFrame original que possuem aquele valor na coluna especificada. É como se ele criasse várias "pastas" invisíveis, e em cada pasta, ele colocasse as linhas que pertencem àquele grupo específico.

# O Método .groupby() em Ação: Sintaxe e Primeiros Passos

## Sintaxe Básica

Agora que entendemos a lógica por trás da divisão, é hora de colocar as mãos na massa e ver como o .groupby() funciona na prática. O método é incrivelmente flexível e é a porta de entrada para a análise de dados por categoria no Pandas. Sua sintaxe básica é bastante direta, mas as possibilidades que ele abre são vastas.

> **Importante**
>
> O resultado de `.groupby()` não é um DataFrame diretamente, mas sim um objeto **DataFrameGroupBy**. Este objeto representa como os dados foram divididos e está pronto para a próxima fase: aplicar funções de agregação.

```
import pandas as pd

# Criando um DataFrame de exemplo
dados = {
    'Regiao': ['Norte', 'Sul', 'Norte',
               'Leste', 'Sul', 'Oeste',
               'Norte', 'Leste'],
    'Produto': ['A', 'B', 'A', 'C',
                'B', 'A', 'C', 'B'],
    'Valor_Venda': [100, 150, 200, 50,
                    120, 300, 80, 180]
}
vendas_df = pd.DataFrame(dados)

# Agrupando por 'Regiao'
grupos_por_regiao = vendas_df.groupby('Regiao')
print(grupos_por_regiao)
```

Para começar, vamos imaginar que temos um DataFrame chamado `vendas_df` com informações sobre transações, incluindo colunas como 'Região', 'Produto' e 'Valor_Venda'. Se quisermos agrupar essas vendas por 'Região', a operação é tão simples quanto o código acima.

Ao executar `vendas_df.groupby('Regiao')`, o que obtemos não é um DataFrame diretamente, mas sim um objeto DataFrameGroupBy. Este objeto é uma representação interna do Pandas de como os dados foram divididos. Ele "sabe" quais linhas pertencem a cada grupo ('Norte', 'Sul', 'Leste', 'Oeste'), mas ainda não aplicou nenhuma operação de agregação. Ele está pronto para a próxima fase do Split-Apply-Combine, onde você aplicará uma função para resumir cada um desses grupos.

# Explorando os Objetos GroupBy: O Que Acontece Após o Split?

Após aplicar o .groupby(), como vimos, o Pandas retorna um objeto DataFrameGroupBy. Este objeto é mais do que apenas um placeholder; ele é uma estrutura inteligente que contém todas as informações necessárias sobre como os dados foram divididos. Ele não exibe os grupos individualmente de imediato, mas está pronto para que você interaja com eles e aplique as operações desejadas.

### Catálogo Organizado

Pense nesse objeto DataFrameGroupBy como um catálogo bem organizado de pastas. Cada pasta representa um grupo (por exemplo, uma região, um tipo de produto), e dentro de cada pasta estão os itens (as linhas do DataFrame) que pertencem àquele grupo.

### Inspeção de Grupos

Você pode inspecionar esse catálogo, mas para ver o conteúdo de cada pasta ou para realizar alguma ação sobre ele, você precisa de um comando específico.

## Iterando sobre os Grupos

Podemos, por exemplo, iterar sobre esse objeto para ver cada grupo individualmente. Isso nos permite visualizar como o Pandas dividiu o DataFrame original. Embora raramente façamos isso em análises de larga escala, é uma excelente forma de entender a estrutura interna:

```
# Continuando do exemplo anterior
# Iterando sobre os grupos
print("Iterando sobre os grupos por Região:")
for nome_grupo, grupo_df in grupos_por_regiao:
    print(f"\nGrupo: {nome_grupo}")
    print(grupo_df)
```

Este loop nos mostra que o DataFrameGroupBy é, de fato, uma coleção de DataFrames menores, cada um correspondendo a um valor único na coluna 'Regiao'. Cada `grupo_df` dentro do loop é um DataFrame Pandas completo, contendo apenas as linhas que pertencem àquele `nome_grupo`. Essa capacidade de acessar e manipular cada grupo individualmente é o que torna o groupby tão poderoso e flexível para a fase de "Apply".

# O "Apply" e "Combine": Agregando Valor aos Grupos

Com os dados devidamente divididos em grupos pelo .groupby(), chegamos à fase crucial de **"Apply"** (Aplicar) e **"Combine"** (Combinar). É aqui que a mágica acontece, transformando os grupos de dados brutos em informações sumarizadas e prontas para análise. A fase "Apply" envolve a aplicação de uma função de agregação a cada um dos grupos que foram formados.

Imagine que você é o gerente de uma rede de lojas e dividiu suas vendas por filial. Agora, para cada filial, você quer saber o total de vendas. Você aplicaria a função "soma" (sum) a cada grupo de vendas de cada filial. O resultado de cada soma seria então "combinado" em uma nova estrutura, geralmente um Series ou um DataFrame, onde cada linha representa um grupo e a coluna exibe o resultado da agregação.

### Eficiência do Pandas

No Pandas, essa aplicação e combinação são frequentemente realizadas em uma única linha de código, tornando o processo muito eficiente. Após criar o objeto DataFrameGroupBy, você simplesmente chama uma função de agregação diretamente sobre ele. O Pandas se encarrega de aplicar essa função a cada sub-DataFrame (grupo) e, em seguida, de juntar todos os resultados em uma única estrutura de saída.

## Exemplo Prático

```
# Continuando do exemplo anterior
# Aplicando a função de agregação 'sum'
# para obter o total de vendas por região
total_vendas_por_regiao = vendas_df.groupby('Regiao')['Valor_Venda'].sum()
print("\nTotal de Vendas por Região:")
print(total_vendas_por_regiao)
```

Neste exemplo, `['Valor_Venda']` seleciona a coluna à qual a função `sum()` será aplicada, e o `.sum()` é a função de agregação. O resultado é um Series onde o índice são as regiões e os valores são os totais de vendas correspondentes, demonstrando perfeitamente a fase de "Apply" (somar em cada grupo) e "Combine" (juntar os resultados em um Series).

# Funções de Agregação Comuns: Sum, Mean e Count

As funções de agregação são o coração da fase "Apply" do paradigma Split-Apply-Combine. Elas nos permitem resumir os dados dentro de cada grupo de maneiras significativas. As mais comuns e frequentemente utilizadas são sum(), mean() e count(), cada uma oferecendo uma perspectiva diferente sobre os dados agrupados.

### sum()

A função `sum()` é ideal quando queremos obter o **total acumulado** de uma variável numérica dentro de cada grupo. Por exemplo, em um conjunto de dados de vendas, podemos querer saber o total de receita gerada por cada produto ou por cada vendedor. Ela nos dá uma visão da magnitude total.

### mean()

A função `mean()` calcula a **média aritmética** dos valores em cada grupo. É extremamente útil para entender o "valor típico" ou o "comportamento médio" de uma variável. Se quisermos saber a média de gastos por cliente ou a média de tempo de resposta por equipe de suporte, mean() é a escolha certa. Ela ajuda a suavizar as flutuações e a identificar tendências centrais.

### count()

Por fim, a função `count()` simplesmente **conta o número de itens** (linhas não nulas) em cada grupo. Isso é valioso para entender a frequência ou o volume de ocorrências. Por exemplo, quantos clientes cada vendedor atendeu, ou quantos produtos foram vendidos em cada categoria. Ela nos dá uma medida da quantidade de observações em cada grupo.

## Exemplos de Código

```
# Continuando com o DataFrame vendas_df
print("\nTotal de Vendas por Produto:")
print(vendas_df.groupby('Produto')['Valor_Venda'].sum())

print("\nMédia de Vendas por Região:")
print(vendas_df.groupby('Regiao')['Valor_Venda'].mean())

print("\nContagem de Vendas por Produto:")
print(vendas_df.groupby('Produto')['Valor_Venda'].count())
```

Essas três funções são a base para a maioria das análises exploratórias, permitindo-nos rapidamente obter um panorama quantitativo dos nossos dados agrupados.

# Funções de Agregação Comuns: Max e Min

Além de somar, calcular médias e contar, muitas vezes precisamos identificar os valores extremos dentro de nossos grupos de dados. É aí que as funções de agregação `max()` e `min()` se tornam indispensáveis. Elas nos permitem encontrar, respectivamente, o maior e o menor valor de uma variável numérica para cada categoria, revelando os limites e as variações dentro dos nossos dados.

### max()

A função `max()` é utilizada para determinar o **valor máximo** de uma coluna dentro de cada grupo. Imagine que você está analisando o desempenho de diferentes equipes de vendas e quer saber qual foi a maior venda realizada por cada equipe. Ou, em um contexto de monitoramento ambiental, qual foi a temperatura máxima registrada em cada cidade. max() nos ajuda a identificar os picos, os recordes ou os limites superiores.

### min()

Por outro lado, a função `min()` faz exatamente o oposto: ela encontra o **valor mínimo** de uma coluna para cada grupo. Seguindo os exemplos anteriores, você poderia usar min() para descobrir qual foi a menor venda de cada equipe, ou a temperatura mínima em cada cidade. Identificar os valores mínimos é crucial para entender os pisos, os pontos de menor desempenho ou os limites inferiores.


Ambas as funções são poderosas para identificar anomalias, estabelecer faixas de valores ou simplesmente entender a amplitude dos dados dentro de cada categoria. Elas complementam sum(), mean() e count() ao fornecer uma visão mais completa da distribuição dos valores.

```
# Continuando com o DataFrame vendas_df
print("\nMaior Venda por Região:")
print(vendas_df.groupby('Regiao')['Valor_Venda'].max())

print("\nMenor Venda por Produto:")
print(vendas_df.groupby('Produto')['Valor_Venda'].min())
```

Ao combinar max() e min() com as outras funções de agregação, você constrói um conjunto robusto de ferramentas para explorar e resumir seus dados de forma abrangente, extraindo insights valiosos sobre os extremos e a variabilidade em cada grupo.

# Agregação com Múltiplas Colunas e Funções: O Poder do .agg()

Até agora, vimos como aplicar uma única função de agregação a uma única coluna após o agrupamento. No entanto, na análise de dados do mundo real, raramente nos contentamos com uma única métrica. Muitas vezes, precisamos calcular várias estatísticas (como soma, média e contagem) para diferentes colunas ou até mesmo para a mesma coluna, tudo de uma vez. É aqui que o método `.agg()` (de "aggregate") do Pandas brilha, oferecendo uma flexibilidade incomparável.

### Múltiplas Funções

Especifique várias funções de agregação para uma mesma coluna usando uma lista.

### Múltiplas Colunas

Aplique diferentes funções a diferentes colunas usando um dicionário.

### Eficiência

Consolide resultados em um formato de fácil leitura em uma única operação.

## Exemplo: Múltiplas Funções em Uma Coluna

```
# Continuando com o DataFrame vendas_df
# Agregando múltiplas funções a uma única coluna
print("\nTotal, Média e Contagem de Vendas por Região:")
print(vendas_df.groupby('Regiao')['Valor_Venda'].agg(['sum', 'mean', 'count']))
```

## Exemplo: Diferentes Funções em Diferentes Colunas

```
# Agregando diferentes funções a diferentes colunas
dados_mais_completos = {
    'Regiao': ['Norte', 'Sul', 'Norte', 'Leste',
               'Sul', 'Oeste', 'Norte', 'Leste'],
    'Produto': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B'],
    'Valor_Venda': [100, 150, 200, 50, 120, 300, 80, 180],
    'Quantidade': [2, 1, 3, 1, 2, 4, 1, 2]
}
vendas_completo_df = pd.DataFrame(dados_mais_completos)

print("\nAgregação de Múltiplas Colunas com Múltiplas Funções por Região:")
print(vendas_completo_df.groupby('Regiao').agg(
    total_vendas=('Valor_Venda', 'sum'),
    media_quantidade=('Quantidade', 'mean'),
    max_venda=('Valor_Venda', 'max')
))
```

O .agg() permite que você especifique múltiplas funções de agregação para múltiplas colunas em uma única operação. Isso economiza tempo, torna o código mais limpo e, o mais importante, consolida os resultados em um formato de fácil leitura.

# Personalizando Agregações com Dicionários no .agg()

## Nomes Personalizados

A flexibilidade do .agg() vai além de simplesmente aplicar múltiplas funções. Podemos ir um passo adiante e **personalizar os nomes das colunas** resultantes da agregação, o que é extremamente útil para criar relatórios claros e autoexplicativos. Em vez de ter colunas com nomes genéricos como 'sum' ou 'mean', podemos nomeá-las de acordo com o contexto da nossa análise, como 'Total_Receita' ou 'Media_Preco_Unitario'.

Essa personalização é feita passando um dicionário para o método .agg(). As chaves do dicionário serão os novos nomes das colunas no DataFrame resultante, e os valores serão tuplas ou listas que especificam a coluna original a ser agregada e a função de agregação a ser aplicada. Isso nos dá um controle granular sobre a saída, tornando nossos resultados mais legíveis e profissionais.

### Comunicação Eficaz

Imagine que você está preparando um relatório para a diretoria, e eles precisam de métricas específicas com nomes claros. Em vez de apresentar uma tabela com 'Valor_Venda_sum' e 'Valor_Venda_mean', você pode renomeá-las para **'Receita Total'** e **'Ticket Médio'**, respectivamente. Essa pequena mudança faz uma grande diferença na clareza e na comunicação dos seus insights.

## Exemplo de Código

```
# Continuando com o DataFrame vendas_completo_df
print("\nAgregação Personalizada com Dicionário no .agg() por Produto:")
print(vendas_completo_df.groupby('Produto').agg(
    Receita_Total=('Valor_Venda', 'sum'),
    Ticket_Medio=('Valor_Venda', 'mean'),
    Unidades_Vendidas=('Quantidade', 'sum'),
    Maior_Venda_Unica=('Valor_Venda', 'max')
))
```

Com essa abordagem, cada métrica agregada recebe um nome descritivo, facilitando a interpretação dos resultados e a integração em relatórios ou dashboards. É uma prática recomendada para garantir que suas análises sejam não apenas precisas, mas também facilmente compreensíveis por qualquer pessoa que as consulte.

# Agrupamento com Múltiplas Colunas: Análise Multidimensional

Até agora, nossos exemplos de agrupamento focaram em uma única coluna, como 'Regiao' ou 'Produto'. No entanto, a realidade dos dados é frequentemente mais complexa, exigindo que analisemos informações sob múltiplas perspectivas simultaneamente. E se quisermos entender o total de vendas por 'Regiao' *e* por 'Produto' ao mesmo tempo? Ou a média de idade dos clientes por 'Gênero' *e* por 'Faixa_Etaria'?

### Visão Multidimensional

O Pandas permite que você agrupe seus dados por múltiplas colunas, criando uma análise multidimensional.

### Combinações Únicas

Em vez de dividir o DataFrame em grupos baseados em um único critério, ele cria grupos baseados em combinações únicas dos valores de todas as colunas especificadas.

### Insights Granulares

Isso nos permite extrair insights mais granulares e identificar padrões que seriam invisíveis em uma análise unidimensional.

> *"Pense em um mapa onde você não apenas vê as fronteiras dos países, mas também as cidades dentro de cada país. Agrupar por múltiplas colunas é como adicionar essa camada extra de detalhe, permitindo-lhe navegar por hierarquias e interações entre diferentes categorias."*

## Sintaxe e Exemplo

Para agrupar por múltiplas colunas, basta passar uma lista de nomes de colunas para o método .groupby(). O Pandas então criará grupos para cada combinação única de valores nessas colunas.

```
# Continuando com o DataFrame vendas_completo_df
# Agrupando por 'Regiao' e 'Produto'
print("\nTotal de Vendas por Região e Produto:")
print(vendas_completo_df.groupby(['Regiao', 'Produto'])['Valor_Venda'].sum())

# Agrupando e aplicando múltiplas agregações
print("\nAnálise Detalhada por Região e Produto:")
print(vendas_completo_df.groupby(['Regiao', 'Produto']).agg(
    TotalVendas=('Valor_Venda', 'sum'),
    MediaVendas=('Valor_Venda', 'mean'),
    ContagemTransacoes=('Valor_Venda', 'count')
))
```

O resultado é um Series ou DataFrame com um **índice multinível**, onde cada nível corresponde a uma das colunas de agrupamento. Essa estrutura é extremamente poderosa para análises detalhadas, permitindo-nos explorar as interações entre diferentes variáveis categóricas e descobrir insights mais profundos sobre o comportamento dos nossos dados.

# Análise de Dados por Categoria: Extraindo Insights Reais

O verdadeiro poder do agrupamento e agregação não reside apenas na execução técnica dos comandos, mas na capacidade de transformar os resultados em insights acionáveis. Depois de aplicar o groupby() e as funções de agregação, você terá tabelas resumidas que contam uma história. O desafio é ler essa história e extrair conclusões que possam guiar decisões estratégicas.

### Otimização de Marketing

Ao agrupar os resultados por 'Canal de Marketing' (e-mail, redes sociais, busca paga) e calcular a 'Receita Total' e o 'Custo por Aquisição' para cada um, você pode identificar qual canal está gerando o maior retorno sobre o investimento. Essa informação é vital para otimizar o orçamento da próxima campanha.

### Identificação de Padrões

A análise por categoria nos permite ir além das médias gerais e entender as nuances. Um produto pode ter uma média de vendas baixa no geral, mas ser um campeão de vendas em uma região específica. Um serviço pode ter uma alta taxa de satisfação geral, mas apresentar problemas graves em um segmento de clientes.

### Decisões Estratégicas

O agrupamento revela essas disparidades e oportunidades. Conecte os resultados agregados com o contexto do problema original para transformar dados em conhecimento e, finalmente, em ações.

## Exemplo de Extração de Insights

```
# Exemplo de extração de insights
# Suponha que o resultado do agrupamento por Regiao e Produto seja:
# Regiao  Produto
# Leste   B         180
#         C          50
# Norte   A         300
#         C          80
# Oeste   A         300
# Sul     B         270
# Name: Valor_Venda, dtype: int64


# A partir deste resultado, podemos observar:
# - O produto 'A' é forte nas regiões Norte e Oeste.
# - O produto 'B' é forte nas regiões Sul e Leste.
# - O produto 'C' tem vendas mais modestas em ambas as regiões onde aparece.
```

### Insight 1: Estratégia de Marketing

Campanhas para o Produto 'A' podem ser focadas no Norte e Oeste.

### Insight 2: Otimização de Estoque

Garantir que o estoque do Produto 'B' seja adequado no Sul e Leste.

### Insight 3: Análise de Desempenho

Investigar por que o Produto 'C' tem vendas mais baixas e se há potencial de crescimento.

A chave é sempre perguntar: **"O que esses números me dizem sobre o negócio ou o fenômeno que estou estudando?"**. Conecte os resultados agregados com o contexto do problema original para transformar dados em conhecimento e, finalmente, em ações.

# Desafios Comuns e Dicas de Otimização

Embora o groupby() seja uma ferramenta poderosa, a sua aplicação no mundo real pode apresentar alguns desafios. Estar ciente deles e saber como contorná-los é parte integrante de se tornar um analista de dados proficiente. Dois dos desafios mais comuns envolvem o tratamento de dados ausentes (valores NaN) e a otimização de performance em DataFrames muito grandes.

## Tratamento de Dados Ausentes (NaN)

Dados ausentes são uma realidade em quase todo conjunto de dados. Quando você agrupa e agrega, o Pandas geralmente **ignora os valores NaN** por padrão em muitas funções de agregação (como sum, mean, count). No entanto, é crucial entender esse comportamento. Se você precisa que os NaNs sejam tratados de forma diferente (por exemplo, preenchidos com zero antes da agregação), você deve realizar o pré-processamento (usando `fillna()`) antes de aplicar o groupby(). Ignorar NaNs pode levar a resultados agregados que não representam a totalidade dos dados se você esperava que eles fossem incluídos de alguma forma.

```
import numpy as np

dados_com_nan = {
    'Categoria': ['A', 'B', 'A', 'C',
                  'B', 'A', 'C', 'B'],
    'Valor': [100, 150, np.nan, 50,
              120, 300, 80, np.nan]
}
df_nan = pd.DataFrame(dados_com_nan)

# Agrupamento sem tratamento de NaN
print("Soma por Categoria (NaNs ignorados):")
print(df_nan.groupby('Categoria')['Valor'].sum())

# Tratando NaN antes do agrupamento
df_nan_preenchido = df_nan.fillna(0)
print("\nSoma por Categoria (NaNs preenchidos
com 0):")
print(df_nan_preenchido.groupby('Categoria')
['Valor'].sum())
```

## Otimização de Performance

Outro ponto importante é a performance, especialmente ao trabalhar com DataFrames que contêm milhões ou bilhões de linhas. Agrupar e agregar esses volumes de dados pode ser custoso computacionalmente. Uma dica de otimização é garantir que as colunas usadas para agrupamento sejam do tipo de dado mais eficiente. Para colunas categóricas com um número limitado de valores únicos, **converter o tipo de dado para** `category` pode acelerar significativamente as operações de groupby(), pois o Pandas pode otimizar o armazenamento e o processamento desses dados.

```
# Otimização de tipo para colunas categóricas
df_nan['Categoria'] =
df_nan['Categoria'].astype('category')
print(f"\nTipo da coluna 'Categoria' após
otimização: {df_nan['Categoria'].dtype}")
```

### Boas Práticas

Ao adotar essas práticas, você garante que suas análises sejam não apenas corretas, mas também eficientes, mesmo com grandes volumes de dados.

# O Ecossistema Pandas e o Fluxo de Trabalho End-to-End

O método .groupby() e as funções de agregação do Pandas não são ferramentas isoladas; eles se encaixam perfeitamente em um fluxo de trabalho de análise de dados mais amplo e completo, o famoso "end-to-end". Desde a coleta e limpeza dos dados até a visualização e comunicação dos resultados, o agrupamento desempenha um papel central na fase de exploração e transformação.

### Coleta e Limpeza

Carregar dados de CSV, bancos de dados ou APIs e realizar pré-processamento inicial.

### Agrupamento

Aplicar groupby() para análise exploratória e transformação de dados.

### Visualização

Criar gráficos com Matplotlib/Seaborn para comunicar insights.

Após carregar seus dados (talvez de um CSV, banco de dados ou API) e realizar as etapas iniciais de limpeza e pré-processamento (lidando com NaNs, formatando colunas, etc.), o groupby() entra em cena para a análise exploratória. É a ponte entre os dados brutos e os insights. Uma vez que você agrupou e agregou seus dados para obter as métricas desejadas, o próximo passo lógico é visualizar esses resultados.

É aqui que bibliotecas como **Matplotlib** e **Seaborn**, que fazem parte do ecossistema Python para ciência de dados, se conectam de forma poderosa. Os DataFrames ou Series resultantes das operações de groupby() e agg() são a entrada perfeita para criar gráficos de barras, gráficos de pizza, heatmaps ou outros tipos de visualizações que comunicam seus insights de forma clara e impactante.

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

# Agrupando e agregando para visualização
receita_por_regiao =
vendas_completo_df.groupby('Regiao')
['Valor_Venda'].sum().reset_index()

# Visualizando os resultados
plt.figure(figsize=(8, 5))
sns.barplot(x='Regiao', y='Valor_Venda',
        data=receita_por_regiao,
        palette='viridis')
plt.title('Receita Total por Região')
plt.xlabel('Região')
plt.ylabel('Receita Total')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Essa integração entre manipulação de dados (Pandas) e visualização (Matplotlib/Seaborn) dentro de ambientes interativos como Jupyter Notebooks ou Google Colab é o que define o fluxo de trabalho moderno de análise de dados. Permite que você explore, transforme, visualize e documente todo o seu processo de forma fluida e iterativa, transformando dados brutos em histórias convincentes e decisões estratégicas.

# Consolidação: Agrupando para Decidir

Chegamos ao fim de nossa jornada sobre agrupamento e agregação de dados com o método groupby() do Pandas. Vimos que a capacidade de dividir, aplicar e combinar informações é uma das pedras angulares da análise de dados eficaz. Desde a compreensão do conceito "Split-Apply-Combine" até a aplicação prática de funções de agregação como sum, mean, count, max e min, você agora possui as ferramentas para transformar dados brutos em insights valiosos. Exploramos como o .agg() oferece flexibilidade para múltiplas agregações e como agrupar por múltiplas colunas revela padrões multidimensionais.

### Em Prática

Use `groupby()` para resumir vendas por categoria de produto e identificar os mais lucrativos. Calcule a média de tempo de resposta de suporte por equipe para otimizar o atendimento. Conte o número de ocorrências de eventos por tipo para monitorar tendências. Essas operações são fundamentais para qualquer análise que busque entender o comportamento de grupos dentro de um conjunto de dados.

## Autoavaliação

1. Qual das seguintes opções melhor descreve o conceito de "Split-Apply-Combine"?
   a) Um método para dividir um DataFrame em vários arquivos separados.
   b) Um paradigma para organizar dados, aplicar funções a grupos e combinar os resultados.
   c) Uma técnica para combinar múltiplos DataFrames em um único.
   d) Um processo para limpar dados ausentes e inconsistentes.
2. No Pandas, qual método é utilizado para iniciar o processo de agrupamento de dados?
   a) .aggregate()
   b) .divide()
   c) .group_by()
   d) .groupby()
3. Se você deseja calcular o total de vendas e a média de preço por categoria de produto em um único passo, qual método do Pandas você utilizaria após o .groupby()?
   a) .sum()
   b) .mean()
   c) .agg()
   d) .count()
4. Qual das seguintes funções de agregação seria mais adequada para descobrir o maior valor de uma coluna numérica dentro de cada grupo?
   a) sum()
   b) mean()
   c) min()
   d) max()
5. Explique como o agrupamento por múltiplas colunas pode fornecer insights mais profundos do que o agrupamento por uma única coluna, utilizando um exemplo prático.

### Gabarito

1. b) | 2. d) | 3. c) | 4. d)

---

## Próxima Aula

**Aula 11**: Na próxima aula, daremos um passo adiante na manipulação de dados, aprendendo a combinar DataFrames de diversas formas. Exploraremos os métodos merge, join e concat, essenciais para integrar informações de diferentes fontes e construir conjuntos de dados mais completos para suas análises.

## Recursos Adicionais

- **Documentação oficial do Pandas**: Para aprofundar nos detalhes de groupby() e agg().
- **Livros sobre Python para Data Science**: Oferecem exemplos práticos e estudos de caso.
- **Cursos online de manipulação de dados com Pandas**: Para prática interativa e resolução de problemas.

**NOTA IMPORTANTE**: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre a documentação oficial das bibliotecas para verificar alterações e novas funcionalidades.