Глава IX. Групиране и Агрегиране (GroupBy)

В тази глава ще се потопим в мощната функционалност на Pandas, която позволява **групиране и агрегиране на данни**. Ще разгледаме как да организираме сложни набори от данни в по-смислени подгрупи въз основа на стойностите в една или няколко колони (или дори нива на индекс, `Series` или функции).

Основната концепция, която ще ръководи нашето обучение, е **"Раздели-Приложи-Комбинирай" (Split-Apply-Combine)**. Ще научим как да:

- **Разделим (Split)** `DataFrame`-а на групи от редове въз основа на специфични критерии.
- **Приложим (Apply)** функция (агрегираща, трансформираща или филтрираща) към всяка от тези групи поотделно.
- **Комбинираме (Combine)** резултатите от тези приложения обратно в структуриран `DataFrame` или `Series`.

Ще започнем с детайлно разглеждане на метода `.groupby()`, който е в основата на процеса на групиране. Ще видим различните начини за създаване на `GroupBy` обекти, включително групиране по една или множество колони, по `Series` или NumPy array, по нива на индекс и дори по резултата от функции, приложени към данните.

След като създадем `GroupBy` обект, ще се научим как да го инспектираме, за да разберем структурата на създадените групи (чрез `.groups`), да извличаме конкретни групи (чрез `.get_group()`) и да получаваме основна информация за тях (като `.size()`, `.first()`, `.last()`). Ще разгледаме и как да итерираме през отделните групи.

Основна част от главата ще бъде посветена на **агрегацията** с методите `.aggregate()` (или краткия му вариант `.agg()`). Ще научим как да прилагаме различни вградени агрегиращи функции (като сума, средна стойност, стандартно отклонение, брой и други) към групираните данни. Ще видим как да прилагаме множество функции едновременно, както и различни функции към различни колони. Накрая, ще се научим да създаваме и прилагаме свои собствени, потребителски агрегиращи функции.

След агрегацията ще разгледаме **трансформацията** с метода `.transform()`, която позволява да се прилагат функции към всяка група, но за разлика от агрегацията, връща резултат със същата форма (същия брой редове) като оригиналния входен `DataFrame`.

Ще продължим с **филтрирането** на групи с метода `.filter()`, където можем да премахваме цели групи от данни въз основа на резултата от прилагане на функция, която връща булева стойност за всяка група.

Накрая, ще разгледаме изключително гъвкавия метод `.apply()`, който позволява да се прилага произволна функция към всяка група и да се контролира формата на върнатия резултат.

Чрез изучаването на тези концепции и методи, вие ще придобиете уменията да анализирате сложни набори от данни, да извличате обобщена информация, да извършвате специфични за групите изчисления и да трансформирате данните си по ефективен и мощен начин.

I. Концепцията "Раздели-Приложи-Комбинирай" (Split-Apply-Combine)

Нека разгледаме основната концепция, която стои в основата на груповата операция в Pandas: **"Раздели-Приложи-Комбинирай" (Split-Apply-Combine)**. Тази парадигма е изключително мощна и гъвкава за анализ на данни и може да бъде приложена в много ситуации, където искаме да извършим някакви изчисления или трансформации върху подмножества от нашите данни.

Нека си представим, че имаме `DataFrame` с данни, които могат да бъдат категоризирани по един или няколко критерия (например, по държава, по продукт, по дата и т.н.). Концепцията "Раздели-Приложи-Комбинирай" ни позволява да обработим тези категории по следния начин:

1. Раздели (Split):

- Първата стъпка е да **разделим** оригиналния `DataFrame` на множество по-малки `DataFrame`-и (или "групи") въз основа на стойностите в една или повече колони (които наричаме **ключове за групиране**).
- Всяка уникална стойност (или комбинация от стойности) в ключовите колони формира отделна група.
- Pandas `.groupby()` метод е основният инструмент за тази стъпка. Той не извършва никакво изчисление сам по себе си, а създава `GroupBy` обект, който представлява "плана" за това как данните ще бъдат разделени.

➢ Пример за разделяне:

Представете си `DataFrame` с продажби, съдържащ колони 'Продукт' и 'Продадено количество'. Ако искаме да анализираме продажбите по продукт, ще разделим `DataFrame`-а на групи, където всяка група съдържа всички редове за даден уникален продукт (напр., една група за "Ябълки", друга за "Банани" и т.н.).

2. Приложи (Apply):

- След като данните са разделени на групи, можем да **приложим** някаква операция към всяка от тези групи независимо.
- Операцията може да бъде една от следните (или комбинация от тях):
    - **Агрегация (Aggregation):** Изчисляване на обобщени статистики за всяка група (напр., сума, средна стойност, брой, минимум, максимум).
    - **Трансформация (Transformation):** Извършване на поелементни изчисления или други трансформации, които връщат `Series` със същия индекс като оригиналната група. Резултатът от трансформацията обикновено се комбинира обратно с оригиналния `DataFrame`.
    - **Филтрация (Filtration):** Премахване на цели групи от данни въз основа на определено условие, което се оценява върху групата като цяло.
    - **Прилагане на произволна функция (Application):** Използване на `.apply()` за прилагане на дефинирана от потребителя функция към всяка група, което може да доведе до резултати с произволна форма (скалар, `Series` или `DataFrame`).

➢ Пример за прилагане:

В нашия пример с продажбите по продукт, можем да приложим агрегираща функция като `.sum()` към колоната 'Продадено количество' във всяка група. Това ще ни даде общото продадено количество за всеки продукт.

3. Комбинирай (Combine):

- Последната стъпка е да **комбинираме** резултатите от приложената операция обратно в един `DataFrame` или `Series`.
- Как точно се комбинират резултатите зависи от вида на приложената операция:
  - При **агрегацията**, резултатът обикновено е `DataFrame` или `Series` с един ред за всяка група и колони, съдържащи агрегираните стойности.
  - При **трансформацията**, резултатът обикновено е `Series`, който се излъчва обратно към оригиналния `DataFrame` въз основа на индекса.
  - При **филтрацията**, резултатът е `DataFrame`, който съдържа само групите, които са изпълнили условието за филтриране.
  - При **прилагането на произволна функция**, формата на комбинирания резултат зависи от това какво връща функцията за всяка група.

а) Пример за комбиниране:

След като сме изчислили общото продадено количество за всеки продукт (чрез агрегация), резултатът ще бъде `Series` или `DataFrame`, където индексът са уникалните продукти, а стойността е съответното общо количество.

б) Визуализация на "Раздели-Приложи-Комбинирай":

```
Оригинален DataFrame
--------------------
| Категория | Стойност |
|-----------|----------|
| A         | 10       |
| B         | 15       |
| A         | 20       |
| B         | 25       |
| C         | 30       |
--------------------

Раздели (по 'Категория'):

Група 'A':          Група 'B':          Група 'C':
-------------         -------------         -------------
```

```
| Категория | Стойност |   | Категория | Стойност |   | Категория |
Стойност |
|-----------|----------|   |-----------|----------|   |-----------|-------
---|
| A         | 10       |   | B         | 15       |   | C         | 30
|
| A         | 20       |   | B         | 25       |   -------------
-------------         -------------

Приложи (например, sum() към 'Стойност'):

Резултат за 'A': 30     Резултат за 'B': 40     Резултат за 'C': 30

Комбинирай:

--------------------
| Категория | Сума    |
|-----------|---------|
| A         | 30      |
| B         | 40      |
| C         | 30      |
--------------------
```

Разбирането на тази концепция е ключово за ефективното използване на `.groupby()` в Pandas. В следващите теми ще разгледаме как да създаваме `GroupBy` обекти и как да прилагаме различните видове операции към тях.

II. Създаване на GroupBy обект (.groupby())

Методът `.groupby()` е **основният инструмент в Pandas за имплементиране на фазата "Раздели"** от концепцията "Раздели-Приложи-Комбинирай". Когато извикате `.groupby()` върху `DataFrame` или `Series`, вие не получавате нов `DataFrame` или `Series` веднага. Вместо това, `.groupby()` връща специален обект, наречен **`GroupBy` обект**.

1. Какво представлява `GroupBy` обектът?

`GroupBy` обектът е междинна структура, която съдържа информация за това как оригиналните данни трябва да бъдат групирани. Той по същество е **колекция от групи**, където всяка група съдържа всички редове (или стойности в `Series`), които споделят една и съща стойност (или комбинация от стойности) в ключа за групиране.

Ключове за групиране:

Ключовете за групиране определят как ще се извърши разделянето на данните. Те могат да бъдат:

- Една или няколко колони от `DataFrame`-а.
- `Series` със същата дължина като `DataFrame`-а.
- NumPy array със същата дължина като `DataFrame`-а.
- Функция, която се извиква върху всеки индекс или всяка стойност от колона.
- Ниво на MultiIndex (ако `DataFrame`-ът има такъв).

а) Как работи `.groupby()`?

1. Методът `.groupby()` идентифицира **уникалните стойности** (или комбинации от стойности) в предоставения ключ за групиране.
2. За всяка уникална стойност (или комбинация), той **събира всички редове** (или стойности от `Series`) от оригиналния обект, които съответстват на тази стойност, и ги формира в отделна група.
3. `GroupBy` обектът съдържа **картографиране** между всяка уникална стойност (или комбинация) и съответните редове (или стойности).

б) Какво можем да правим с `GroupBy` обект?

След като създадем `GroupBy` обект, той сам по себе си не е много полезен за директно разглеждане на данните. Истинската сила на `GroupBy` обекта се разгръща, когато **приложим една от следните операции върху него**:

- **Агрегация (`.aggregate()` или `.agg()`):** Изчисляване на обобщени статистики за всяка група.
- **Трансформация (`.transform()`):** Прилагане на функция към всяка група и връщане на резултат със същата форма като оригиналния `DataFrame`.
- **Филтрация (`.filter()`):** Премахване на цели групи въз основа на условие.
- **Прилагане (`.apply()`):** Прилагане на произволна функция към всяка група.
- **Инспекция:** Разглеждане на структурата на групите (`.groups`), извличане на конкретна група (`.get_group()`), определяне на размера на всяка група (`.size()`), показване на първия/последния елемент във всяка група (`.first()`, `.last()`).
- **Итериране:** Преминаване през всяка група и нейните данни.

2. Групиране по една или множество колони

Когато искаме да групираме `DataFrame` въз основа на уникалните стойности в една колона, просто подаваме **името на тази колона (като низ)** на метода `.groupby()`.

➢ Пример:

Представете си `DataFrame` с информация за продажби на различни продукти в различни градове:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df_sales)

# Групиране по колона 'Град'
grouped_by_city = df_sales.groupby('Град')

# 'grouped_by_city' сега е GroupBy обект, групиран по уникалните
стойности в колона 'Град'
print("\nGroupBy обект (групиран по 'Град'):\n", grouped_by_city)
```

В този пример, df_sales.groupby('Град') създава GroupBy обект. Вътрешно, Pandas ще идентифицира уникалните градове ('София', 'Пловдив', 'Варна') и ще събере всички редове, които съответстват на всеки град, в отделни групи.

3. Групиране по множество колони:

Можем да групираме `DataFrame` и по комбинация от уникални стойности от няколко колони. За целта подаваме **списък от имената на колоните (като низове)** на метода `.groupby()`.

Пример:

Продължавайки с горния пример, да предположим, че искаме да анализираме продажбите не само по град, но и по продукт във всеки град:

```
# Групиране по колони 'Град' и 'Продукт'
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# 'grouped_by_city_product' е GroupBy обект, групиран по уникалните
комбинации от ('Град', 'Продукт')
print("\nGroupBy обект (групиран по ['Град', 'Продукт']):\n",
grouped_by_city_product)
```

Сега, grouped_by_city_product ще съдържа групи за всяка уникална комбинация от град и продукт (например, ('София', 'A'), ('София', 'B'), ('Пловдив', 'A'), ('Пловдив', 'B'), ('Варна', 'A'), ('Варна', 'C')).

а) Какво се случва след групирането?

Както споменахме, `.groupby()` връща `GroupBy` обект. За да видим реалните групи и да извършим някакъв анализ, трябва да приложим операция върху този обект (например, агрегация).

б) Пример за прилагане на агрегация след групиране:

```
# Изчисляване на средните продажби за всяка група (по град)
average_sales_by_city = grouped_by_city['Продажби'].mean()
print("\nСредни продажби по град:\n", average_sales_by_city)

# Изчисляване на сумарните продажби за всяка група (по град и продукт)
total_sales_by_city_product = grouped_by_city_product['Продажби'].sum()
print("\nСумарни продажби по град и продукт:\n",
total_sales_by_city_product)
```

В тези примери, първо групираме данните, а след това избираме колоната 'Продажби' от GroupBy обекта и прилагаме агрегираща функция (.mean() и .sum()). Резултатът е Series, където индексът е ключът за групиране (град или комбинация от град и продукт), а стойността е резултатът от агрегацията.

Групирането по една или множество колони е основен и много гъвкав начин за структуриране на данните за последващ анализ с помощта на "Раздели-Приложи-Комбинирай".

4. Групиране по Series или NumPy array

Нека разгледаме как можем да създадем `GroupBy` обект, използвайки `Series` или NumPy array като ключ за групиране. Това е полезно, когато искаме да групираме данните въз основа на критерии, които не са пряко налични като колони в самия `DataFrame`, но могат да бъдат изчислени или предоставени външно.

а) Групиране по Series:

Можем да подадем `Series` обект на метода `.groupby()`, при условие че `Series`-ът има **същата дължина като `DataFrame`-а**, който искаме да групираме. Стойностите в `Series`-а ще бъдат използвани като ключове за групиране, а индексът на `Series`-а трябва да съвпада с индекса на `DataFrame`-а.

Сценарио от реалния живот:

Представете си, че имате `DataFrame` с информация за продажби на различни артикули и имате отделен `Series`, който класифицира всеки артикул в определена категория. Можете да използвате този `Series` за да групирате продажбите по категория.

Пример:

```
import pandas as pd

data = {'Артикул': ['Ябълка', 'Банан', 'Ябълка', 'Портокал', 'Банан',
'Ябълка'],
        'Продажби': [10, 15, 12, 20, 18, 9]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df_sales)

# Създаваме Series, който съдържа категорията на всеки артикул
category_map = pd.Series(['Плодове', 'Плодове', 'Плодове', 'Плодове',
'Плодове', 'Плодове'])
print("\nSeries с категории:\n", category_map)

# Уверяваме се, че дължината на Series съвпада с дължината на DataFrame
assert len(category_map) == len(df_sales)

# Групиране по Series
grouped_by_category = df_sales.groupby(category_map)
print("\nGroupBy обект (групиран по Series):\n", grouped_by_category)

# Прилагаме агрегация, за да видим общите продажби по категория
total_sales_by_category = grouped_by_category['Продажби'].sum()
print("\nОбщи продажби по категория:\n", total_sales_by_category)
```

В този пример, въпреки че df_sales няма колона за категория, ние използваме външен Series (category_map) със същата дължина, за да групираме продажбите. Всички редове, съответстващи на една и съща стойност в category_map (в случая само 'Плодове'), се събират в една група.

Важно: Индексът на `Series`-а, използван за групиране, трябва да съвпада с индекса на `DataFrame`-а. Ако индексите не съвпадат, Pandas ще подравни `Series`-а по индекса на `DataFrame`-а, което може да доведе до неочаквани резултати, ако не се внимава. В горния пример, тъй като не сме задавали

изрични индекси, и `DataFrame`-ът, и `Series`-ът имат подразбиращ се числов индекс (0, 1, 2, ...), така че подравняването е коректно.

б) Групиране по NumPy array:

Подобно на `Series`, можем да използваме и NumPy array като ключ за групиране, при условие че той има **същата дължина като `DataFrame`-а**. Стойностите в array-я ще определят групите.

Пример:

Да разширим предишния пример, като използваме NumPy array за категориите:

```
import pandas as pd
import numpy as np

data = {'Артикул': ['Ябълка', 'Банан', 'Ябълка', 'Портокал', 'Банан',
'Ябълка'],
        'Продажби': [10, 15, 12, 20, 18, 9]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df_sales)

# Създаваме NumPy array с категории
category_array = np.array(['Плодове', 'Плодове', 'Плодове', 'Плодове',
'Плодове', 'Плодове'])
print("\nNumPy array с категории:\n", category_array)

# Уверяваме се, че дължината на array съвпада с дължината на DataFrame
assert len(category_array) == len(df_sales)

# Групиране по NumPy array
grouped_by_array = df_sales.groupby(category_array)
print("\nGroupBy обект (групиран по NumPy array):\n", grouped_by_array)

# Прилагаме агрегация
total_sales_by_array = grouped_by_array['Продажби'].sum()
print("\nОбщи продажби по групи (от array):\n", total_sales_by_array)
```

Резултатът е същият като при използване на Series, тъй като стойностите и дължината са идентични.

Групирането по `Series` или NumPy array е мощен начин да се добавят външни критерии за анализ към съществуващи `DataFrame`-и. Важно е да се следи за съвпадението на дължината и индекса (когато се използва `Series`).

5. Групиране по нива на индекс

Нека разгледаме как да създадем `GroupBy` обект, когато нашият `DataFrame` има **MultiIndex** (йерархичен индекс). В такива случаи можем да групираме данните въз основа на едно или няколко **нива** от този MultiIndex.

Когато имаме `DataFrame` с MultiIndex, можем да подадем **името (като низ) или номера (като цяло число, започвайки от 0 за най-външното ниво)** на нивото на индекса, по което искаме да групираме, на метода `.groupby()`.

а) Сценарио от реалния живот:

Представете си, че имате `DataFrame`, който съдържа данни за продажби, индексирани по година и тримесечие. MultiIndex може да изглежда така: (Година, Тримесечие). Ако искате да анализирате общите продажби за всяка година, без значение от тримесечието, можете да групирате по нивото на индекса, което представлява годината.

б) Пример:

```
import pandas as pd

data = {'Продажби': [100, 150, 120, 200, 180, 90, 220, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
                                   ('2022', 'Q3'), ('2022', 'Q4'),
                                   ('2023', 'Q1'), ('2023', 'Q2'),
                                   ('2023', 'Q3'), ('2023', 'Q4')],
                                  names=['Година', 'Тримесечие'])
df_sales_multiindex = pd.DataFrame(data, index=index)
print("Оригинален DataFrame с MultiIndex:\n", df_sales_multiindex)

# Групиране по ниво 'Година' (използвайки името на нивото)
grouped_by_year = df_sales_multiindex.groupby(level='Година')
print("\nGroupBy обект (групиран по ниво 'Година'):\n", grouped_by_year)

# Изчисляване на общите продажби за всяка година
total_sales_by_year = grouped_by_year['Продажби'].sum()
print("\nОбщи продажби по година:\n", total_sales_by_year)
```

```
# Групиране по ниво 'Тримесечие' (използвайки номера на нивото - 1)
grouped_by_quarter = df_sales_multiindex.groupby(level=1)
print("\nGroupBy обект (групиран по ниво 'Тримесечие'):\n",
grouped_by_quarter)

# Изчисляване на средните продажби за всяко тримесечие (за всички
години)
average_sales_by_quarter = grouped_by_quarter['Продажби'].mean()
print("\nСредни продажби по тримесечие:\n", average_sales_by_quarter)

# Групиране по множество нива (използвайки списък от имена или номера)
grouped_by_year_quarter = df_sales_multiindex.groupby(level=['Година',
'Тримесечие'])
print("\nGroupBy обект (групиран по нива ['Година', 'Тримесечие']):\n",
grouped_by_year_quarter)

# Можем да приложим агрегация и тук (в този случай ще получим обратно
MultiIndex)
total_sales_by_year_quarter = grouped_by_year_quarter['Продажби'].sum()
print("\nОбщи продажби по година и тримесечие:\n",
total_sales_by_year_quarter)
```

В тези примери:

- `df_sales_multiindex.groupby(level='Година')` групира данните въз основа на уникалните стойности в нивото на индекса с име 'Година' ('2022' и '2023').
- `df_sales_multiindex.groupby(level=1)` групира данните въз основа на уникалните стойности в нивото на индекса с номер 1 (което е 'Тримесечие': 'Q1', 'Q2', 'Q3', 'Q4').
- `df_sales_multiindex.groupby(level=['Година', 'Тримесечие'])` (или `level=[0, 1]`) групира по уникалните комбинации от стойности в нивата 'Година' и 'Тримесечие'.

Групирането по нива на MultiIndex е изключително полезно за анализ на данни с йерархична структура, позволявайки лесно агрегиране и анализ на различни нива на обобщение.

6. Групиране по функции

Нека разгледаме как да създадем `GroupBy` обект, използвайки **функции** като ключ за групиране. Това е много гъвкав начин да групирате данни въз основа на логика, която може да бъде приложена към индекса или към стойностите в колоните.

Можем да подадем функция на метода `.groupby()`. Тази функция ще бъде **извикана върху всеки индекс** (ако групираме върху индекса) или върху всяка **стойност от колона** (ако групираме върху колона). Резултатът от функцията за всяка стойност ще определи към коя група ще принадлежи съответният ред.

а) Сценарио 1: Групиране по функция на индекса:

Представете си `DataFrame`, където индексът представлява дати. Можете да използвате функция, която извлича годината от всяка дата в индекса, за да групирате данните по година.

б) Пример 1:

```
import pandas as pd
import numpy as np

dates = pd.to_datetime(['2023-01-15', '2023-02-20', '2024-01-10', '2024-
03-25', '2024-04-01'])
data = {'Стойност': np.random.randint(1, 100, len(dates))}
df_time = pd.DataFrame(data, index=dates)
print("Оригинален DataFrame с DatetimeIndex:\n", df_time)

# Функция за извличане на годината от DatetimeIndex
def get_year(index):
    return index.year

# Групиране по резултата от функцията, приложена към индекса
grouped_by_year_func = df_time.groupby(get_year)
print("\nGroupBy обект (групиран по година от индекса):\n",
grouped_by_year_func)

# Изчисляване на средната стойност за всяка година
average_value_by_year_func = grouped_by_year_func['Стойност'].mean()
print("\nСредна стойност по година:\n", average_value_by_year_func)
```

В този пример, функцията `get_year` се прилага към всеки елемент от индекса (`DatetimeIndex`). Редовете с дати от 2023 се групират заедно, а тези от 2024 - също.

в) Сценарио 2: Групиране по функция на стойностите в колона:

Да предположим, че имате `DataFrame` с имена на хора и искате да ги групирате по първа буква на името.

г) Пример 2:

```
import pandas as pd

data = {'Име': ['Алиса', 'Борис', 'Бела', 'Виктор', 'Ваня', 'Георги']}
df_names = pd.DataFrame(data)
print("Оригинален DataFrame с имена:\n", df_names)

# Функция за получаване на първата буква от името
def get_first_letter(name):
    return name[0].upper()

# Групиране по резултата от функцията, приложена към колона 'Име'
grouped_by_first_letter =
df_names.groupby(df_names['Име'].apply(get_first_letter))
print("\nGroupBy обект (групиран по първа буква на името):\n",
grouped_by_first_letter)

# Изчисляване на броя на имената, започващи с всяка буква
count_by_first_letter = grouped_by_first_letter.size()
print("\nБрой имена по първа буква:\n", count_by_first_letter)
```

Тук, първо използваме .apply(get_first_letter) върху колоната 'Име', за да създадем Series от първите букви. След това подаваме този Series на .groupby().

д) По-елегантен начин за групиране по функция на колона:

Можем да предадем самата функция директно на `.groupby()` **ако функцията знае как да работи с `Series` (т.е., Pandas ще я приложи към всяка стойност в колоната)**. В случая с получаване на първата буква, можем да използваме lambda функция:

```
# Групиране по първа буква на името, използвайки lambda функция
```

```
grouped_by_first_letter_lambda =
df_names.groupby(df_names['Име'].apply(lambda x: x[0].upper()))
print("\nGroupBy обект (групиран по първа буква с lambda):\n",
grouped_by_first_letter_lambda)

count_by_first_letter_lambda = grouped_by_first_letter_lambda.size()
print("\nБрой имена по първа буква (с lambda):\n",
count_by_first_letter_lambda)
```

Или, ако функцията трябва да се приложи директно към всяка стойност на колоната по време на процеса на групиране:

```
# Групиране директно с функция, приложена към стойностите на колоната
grouped_by_first_letter_direct = df_names.groupby(lambda x:
df_names['Име'].iloc[x][0].upper())
print("\nGroupBy обект (групиран директно с функция):\n",
grouped_by_first_letter_direct)

count_by_first_letter_direct = grouped_by_first_letter_direct.size()
print("\nБрой имена по първа буква (директно с функция):\n",
count_by_first_letter_direct)
```

Важно: Когато използвате функция за групиране, е от решаващо значение функцията да връща консистентни резултати за едни и същи входни стойности, така че редовете да бъдат групирани правилно.

Групирането по функции отваря врати към много по-сложни и персонализирани начини за сегментиране на вашите данни за анализ.

III. Инспекция на GroupBy обект

1. Инспекция на GroupBy обект: .groups

Атрибутът `.groups` на `GroupBy` обекта е полезен за **получаване на информация за структурата на създадените групи**. Той връща **речник**, където:

- **Ключовете** на речника са **уникалните стойности (или комбинации от стойности)**, по които е извършено групирането. Те представляват имената на отделните групи.
- **Стойностите** на речника са **NumPy array от целочислени индекси**, които показват кои редове от оригиналния `DataFrame` принадлежат към съответната група.

а) Пример 1: Групиране по една колона

Нека използваме отново примера с продажбите по град:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Изглед на .groups атрибута
print("Структура на групите (grouped_by_city.groups):\n",
grouped_by_city.groups)
```

Резултатът ще бъде речник, подобен на този:

```
{'Пловдив': array([1, 4]), 'София': array([0, 2, 5]), 'Варна': array([3,
6])}
```

Това означава, че:

- Групата с ключ `'Пловдив'` съдържа редовете с индекси 1 и 4 от оригиналния `df_sales`.
- Групата с ключ `'София'` съдържа редовете с индекси 0, 2 и 5.
- Групата с ключ `'Варна'` съдържа редовете с индекси 3 и 6.

б) Пример 2: Групиране по множество колони

Сега да видим `.groups` когато групираме по 'Град' и 'Продукт':

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# Изглед на .groups атрибута
print("\nСтруктура на групите (grouped_by_city_product.groups):\n",
grouped_by_city_product.groups)
```

Резултатът ще бъде речник с tuple-и като ключове:

```
{('Пловдив', 'B'): array([1, 4]), ('София', 'A'): array([0, 2]),
('София', 'C'): array([5]), ('Варна', 'A'): array([6]), ('Варна', 'C'):
array([3])}
```

Тук ключовете са комбинации от стойностите в колоните 'Град' и 'Продукт', а стойностите са съответните индекси на редовете.

в) Пример 3: Групиране по `Series`

Нека използваме отново примера с категориите на артикулите:

```
category_map = pd.Series(['Плодове', 'Плодове', 'Плодове', 'Плодове',
'Плодове', 'Плодове'])
grouped_by_category = df_sales.groupby(category_map)

# Изглед на .groups атрибута
print("\nСтруктура на групите (grouped_by_category.groups):\n",
grouped_by_category.groups)
```

Резултатът:

```
{'Плодове': array([0, 1, 2, 3, 4, 5])}
```

Тъй като всички артикули бяха картографирани към една и съща категория ('Плодове'), всички редове попадат в една група.

г) Изводи за `.groups`:

- `.groups` дава директен поглед върху това как `.groupby()` е разделил оригиналния `DataFrame` въз основа на ключовете за групиране.
- Речникът, който се връща, съдържа информацията за принадлежността на всеки ред към определена група чрез неговия целочислен индекс.
- Този атрибут е полезен за разбиране на вътрешната структура на `GroupBy` обекта и за целите на дебъгване или по-задълбочен анализ на групирането.

2. Инспекция на GroupBy обект: `.get_group()`

Методът `.get_group()` позволява да **извлечем конкретна група** от `GroupBy` обекта като нов `DataFrame` (или `Series`, ако оригиналният обект е `Series`). За да използваме този метод, трябва да предоставим **стойността (или tuple от стойности, ако сме групирали по няколко колони)** на групата, която искаме да получим.

а) Пример 1: Групиране по една колона

Използвайки отново примера с продажбите по град:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Извличане на групата за град 'София'
sofia_group = grouped_by_city.get_group('София')
print("Група за 'София':\n", sofia_group)

# Извличане на групата за град 'Пловдив'
plovdiv_group = grouped_by_city.get_group('Пловдив')
print("\nГрупа за 'Пловдив':\n", plovdiv_group)
```

Резултатът ще покаже DataFrame-и, съдържащи само редовете, където колоната 'Град' има съответната стойност.

б) Пример 2: Групиране по множество колони

Когато сме групирали по няколко колони, трябва да предоставим tuple от стойностите на тези колони в реда, в който са били подадени на `.groupby()`.

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# Извличане на групата за ('София', 'A')
sofia_a_group = grouped_by_city_product.get_group(('София', 'A'))
print("\nГрупа за ('София', 'A'):\n", sofia_a_group)

# Извличане на групата за ('Пловдив', 'B')
plovdiv_b_group = grouped_by_city_product.get_group(('Пловдив', 'B'))
print("\nГрупа за ('Пловдив', 'B'):\n", plovdiv_b_group)
```

Тук, за да извлечем групата, съответстваща на град 'София' и продукт 'A', подаваме tuple ('София', 'A') на .get_group().

в) Пример 3: Групиране по ниво на MultiIndex

Ако сме групирали по ниво на MultiIndex, подаваме стойността на това ниво.

```
import pandas as pd

data = {'Продажби': [100, 150, 120, 200, 180, 90, 220, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
                                   ('2022', 'Q3'), ('2022', 'Q4'),
                                   ('2023', 'Q1'), ('2023', 'Q2'),
                                   ('2023', 'Q3'), ('2023', 'Q4')],
                                  names=['Година', 'Тримесечие'])
df_sales_multiindex = pd.DataFrame(data, index=index)
grouped_by_year = df_sales_multiindex.groupby(level='Година')

# Извличане на групата за година '2022'
year_2022_group = grouped_by_year.get_group('2022')
print("\nГрупа за година '2022':\n", year_2022_group)

grouped_by_quarter = df_sales_multiindex.groupby(level='Тримесечие')
```

```
# Извличане на групата за тримесечие 'Q1'
q1_group = grouped_by_quarter.get_group('Q1')
print("\nГрупа за тримесечие 'Q1':\n", q1_group)
```

г) Изводи за `.get_group()`:

- `.get_group()` е полезен метод, когато искаме да разгледаме данните в рамките на конкретна група, създадена от `.groupby()`.
- Той връща подмножество от оригиналния `DataFrame` (или `Series`), съдържащо само редовете (или стойностите), които съответстват на предоставения ключ на групата.
- Ключът трябва да съответства на типа и броя на ключовете, използвани при първоначалното групиране.

3. Инспекция на GroupBy обект: `.size()`

Методът `.size()` се използва, за да получим **броя на редовете (или елементите в `Series`) във всяка група** на `GroupBy` обекта. Той връща `Series`, където:

- **Индексът** на `Series`-а са **ключовете на групите** (уникалните стойности или комбинации от стойности, по които е извършено групирането).
- **Стойностите** на `Series`-а са **броят на елементите** във всяка съответна група.

`.size()` е много полезен за бързо получаване на представа за разпределението на данните по групи.

а) Пример 1: Групиране по една колона

Използвайки примера с продажбите по град:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Получаване на размера на всяка група
group_sizes_city = grouped_by_city.size()
print("Размер на групите (по град):\n", group_sizes_city)
```

Резултатът ще бъде `Series`:

```
Град
Пловдив    2
София      3
Варна      2
dtype: int64
```

Това показва, че има 2 записа за 'Пловдив', 3 за 'София' и 2 за 'Варна'.

б) Пример 2: Групиране по множество колони

Когато групираме по няколко колони, индексът на резултата от `.size()` ще бъде `MultiIndex`, съответстващ на комбинациите от стойности.

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# Получаване на размера на всяка група
group_sizes_city_product = grouped_by_city_product.size()
print("\nРазмер на групите (по град и продукт):\n",
group_sizes_city_product)
```

Резултатът:

```
Град      Продукт
Пловдив   B          2
София     A          2
          C          1
Варна     A          1
          C          1
dtype: int64
```

Това показва броя на записите за всяка уникална комбинация от град и продукт.

в) Пример 3: Групиране по ниво на MultiIndex

Когато групираме по ниво на MultiIndex, индексът на резултата ще съдържа уникалните стойности от това ниво.

```
import pandas as pd

data = {'Продажби': [100, 150, 120, 200, 180, 90, 220, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
                                   ('2022', 'Q3'), ('2022', 'Q4'),
                                   ('2023', 'Q1'), ('2023', 'Q2'),
                                   ('2023', 'Q3'), ('2023', 'Q4')],
                                  names=['Година', 'Тримесечие'])
df_sales_multiindex = pd.DataFrame(data, index=index)
grouped_by_year = df_sales_multiindex.groupby(level='Година')

# Получаване на размера на всяка група (по година)
group_sizes_year = grouped_by_year.size()
print("\nРазмер на групите (по година):\n", group_sizes_year)

grouped_by_quarter = df_sales_multiindex.groupby(level='Тримесечие')

# Получаване на размера на всяка група (по тримесечие)
group_sizes_quarter = grouped_by_quarter.size()
print("\nРазмер на групите (по тримесечие):\n", group_sizes_quarter)
```

Резултатите:

```
Размер на групите (по година):
Година
2022    4
2023    4
dtype: int64

Размер на групите (по тримесечие):
Тримесечие
Q1    2
Q2    2
Q3    2
Q4    2
```

```
dtype: int64
```

`.size()` е бърз и лесен начин да получите обобщена информация за броя на елементите във всяка група, което често е първа стъпка при анализа на групирани данни.

4. Инспекция на GroupBy обект: `.first()` и `.last()`

Методите `.first()` и `.last()` се използват, за да покажат **първия и последния ред (или елемент в `Series`) във всяка група** на `GroupBy` обекта, според реда, в който са се появили в оригиналния `DataFrame`. Те връщат нов `DataFrame` (или `Series`), където индексът са ключовете на групите, а стойностите са съответно първият и последният елемент от всяка група.

Тези методи са полезни за бърз преглед на представителни елементи от всяка група.

а) Пример 1: Групиране по една колона

Използвайки примера с продажбите по град:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Получаване на първия ред от всяка група
first_in_city = grouped_by_city.first()
print("Първи запис във всяка група (по град):\n", first_in_city)

# Получаване на последния ред от всяка група
last_in_city = grouped_by_city.last()
print("\nПоследен запис във всяка група (по град):\n", last_in_city)
```

Резултатите ще бъдат DataFrame-и, индексирани по 'Град', показващи първия и последния срещнат ред за всеки град в оригиналния df_sales.

б) Пример 2: Групиране по множество колони

При групиране по няколко колони, индексът на резултата ще бъде `MultiIndex`.

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# Получаване на първия ред от всяка група (по град и продукт)
first_in_city_product = grouped_by_city_product.first()
print("\nПърви запис във всяка група (по град и продукт):\n",
first_in_city_product)

# Получаване на последния ред от всяка група (по град и продукт)
last_in_city_product = grouped_by_city_product.last()
print("\nПоследен запис във всяка група (по град и продукт):\n",
last_in_city_product)
```

Резултатите ще бъдат DataFrame-и с MultiIndex ('Град', 'Продукт'), показващи първия и последния срещнат ред за всяка уникална комбинация.

в) Пример 3: Групиране по ниво на MultiIndex

```
import pandas as pd

data = {'Продажби': [100, 150, 120, 200, 180, 90, 220, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
                                   ('2022', 'Q3'), ('2022', 'Q4'),
                                   ('2023', 'Q1'), ('2023', 'Q2'),
                                   ('2023', 'Q3'), ('2023', 'Q4')],
                                  names=['Година', 'Тримесечие'])
df_sales_multiindex = pd.DataFrame(data, index=index)
grouped_by_year = df_sales_multiindex.groupby(level='Година')

# Получаване на първия запис за всяка година
first_by_year = grouped_by_year.first()
print("\nПърви запис за всяка година:\n", first_by_year)

# Получаване на последния запис за всяка година
```

```
last_by_year = grouped_by_year.last()
print("\nПоследен запис за всяка година:\n", last_by_year)
```

Резултатите ще покажат DataFrame-и, индексирани по 'Година', съдържащи първия и последния ред за всяка година в оригиналния df_sales_multiindex.

!!!Важно: Резултатите от `.first()` и `.last()` зависят от реда на данните в оригиналния `DataFrame`. Ако редът не е значим или ако данните са били сортирани, тези методи могат да дадат по-смислени резултати.

С тези методи (`.groups`, `.get_group()`, `.size()`, `.first()`, `.last()`) имаме основни инструменти за инспектиране на структурата и съдържанието на `GroupBy` обектите.

IV. Итериране през групи

Нека разгледаме как можем да **итерираме** през групите на `GroupBy` обект. Това е полезно, когато искаме да приложим някаква персонализирана логика към всяка група поотделно.

Когато итерираме през `GroupBy` обект (който е резултат от `.groupby()` върху `DataFrame`), получаваме **двойки (ключ, група)**:

- **Ключът (name):** Това е стойността (или tuple от стойности, ако сме групирали по няколко колони), която определя групата.
- **Групата (group):** Това е `DataFrame`, съдържащ всички редове, които съответстват на този ключ.

Ако сме групирали `Series`, итерирането ще даде двойки **(ключ, група)**, където групата ще бъде `Series`.

Пример 1: Итериране през групи, създадени от групиране по една колона

Използвайки примера с продажбите по град:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
```

```
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

for city, city_group in grouped_by_city:
    print(f"Група за град: {city}")
    print(city_group)
    print("-" * 20)
```

В този пример, при всяка итерация city ще бъде уникален град ('София', 'Пловдив', 'Варна'), а city_group ще бъде DataFrame, съдържащ само редовете за този град.

Пример 2: Итериране през групи, създадени от групиране по множество колони

Когато групираме по няколко колони, ключът ще бъде tuple от стойности.

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

for (city, product), group in grouped_by_city_product:
    print(f"Група за град: {city}, продукт: {product}")
    print(group)
    print("-" * 30)
```

Тук, при всяка итерация (city, product) ще бъде tuple, представляващ уникална комбинация от град и продукт, а group ще бъде съответният DataFrame.

Пример 3: Итериране през групи, създадени от групиране по ниво на MultiIndex

Когато групираме по ниво на MultiIndex, ключът ще бъде стойността на това ниво.

```
import pandas as pd

data = {'Продажби': [100, 150, 120, 200, 180, 90, 220, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
                                   ('2022', 'Q3'), ('2022', 'Q4'),
                                   ('2023', 'Q1'), ('2023', 'Q2'),
```

```
                                        ('2023', 'Q3'), ('2023', 'Q4')],
                                       names=['Година', 'Тримесечие'])
df_sales_multiindex = pd.DataFrame(data, index=index)
grouped_by_year = df_sales_multiindex.groupby(level='Година')

for year, year_group in grouped_by_year:
    print(f"Група за година: {year}")
    print(year_group)
    print("-" * 20)
```

В този случай, `year` ще бъде уникална година ('2022', '2023'), а `year_group` ще бъде `DataFrame`, съдържащ всички записи за тази година.

Изводи за итерирането:

- Итерирането през `GroupBy` обект е мощен начин за извършване на персонализирани операции върху всяка група.
- При всяка итерация получавате ключа на групата и самия `DataFrame` (или `Series`) на групата.
- Можете да използвате стандартни конструкции на Python (като `for` цикъл) за обработка на всяка група.
- Въпреки че итерирането е гъвкаво, за много често срещани операции (като агрегация, трансформация, филтрация) съществуват по-оптимизирани методи, които трябва да се предпочитат пред явната итерация за по-добра производителност, особено при големи набори от данни.

V. Агрегация (.aggregate(), .agg())

След като сме разделили нашия `DataFrame` на групи с помощта на `.groupby()`, често следващата стъпка е да **изчислим обобщени статистики** за всяка от тези групи. Този процес се нарича **агрегация**. Pandas предоставя мощните и гъвкави методи `.aggregate()` (който може да бъде съкратен като `.agg()`) за извършване на тези операции.

Агрегацията позволява да се **редуцира информацията във всяка група до по-компактна форма**, като се изчисляват мерки като сума, средна стойност, брой, минимална и максимална стойност и много други. Резултатът от агрегацията обикновено е `Series` или `DataFrame`, където индексът е ключът за групиране, а колоните съдържат изчислените агрегирани стойности.

Методите `.aggregate()` и `.agg()` могат да приемат различни видове аргументи, което ги прави изключително гъвкави:

- **Единична агрегираща функция (като низ или функция):** Можем да приложим една функция към всички (подходящи) колони във всяка група. Pandas разпознава много често използвани агрегиращи функции по техните имена (напр., `'sum'`, `'mean'`, `'count'`, `'min'`, `'max'`, `'std'`, `'var'`, `'nunique'`). Можем да подадем и самите функции (напр., `np.sum`, `np.mean`).

- **Списък от агрегиращи функции:** Можем да приложим множество агрегиращи функции едновременно към всички (подходящи) колони. Резултатът ще бъде `DataFrame` с MultiIndex за колоните, където вътрешното ниво съдържа имената на приложените функции.
- **Речник от агрегиращи функции, приложени към конкретни колони:** Можем да укажем кои агрегиращи функции да бъдат приложени към кои колони, като използваме речник. Ключовете на речника са имената на колоните, а стойностите са единични функции или списъци от функции, които да бъдат приложени към съответната колона.
- **Потребителски (дефинирани от потребителя) агрегиращи функции:** Можем да създадем свои собствени функции за агрегиране, които приемат `Series` (представляваща колона в групата) като вход и връщат скаларна стойност. Тези функции могат да бъдат подадени на `.aggregate()` или `.agg()`.

В следващите подтеми ще разгледаме всеки от тези начини за прилагане на агрегация с `.aggregate()` и `.agg()` с конкретни примери. Ще видим как да използваме вградени функции, как да прилагаме множество функции едновременно, как да насочваме различни функции към различни колони и как да създаваме и използваме свои собствени агрегиращи функции. Разбирането на тези техники ще ви даде пълен контрол върху обобщаването на вашите групирани данни.

1. Прилагане на единична агрегираща функция ('sum', 'mean', 'std', 'count', 'min', 'max', 'var', 'nunique', etc.)

Когато искаме да изчислим една обобщена статистика (като сума, средна стойност и т.н.) за всяка група, можем да предоставим името на вградена функция (като низ) или самата функция като аргумент на `.agg()` (или `.aggregate()`). Тази функция ще бъде приложена към **всички подходящи колони** във всяка група.

Използване на име на вградена функция (като низ):

Pandas разпознава много често използвани агрегиращи функции по техните имена. Ето някои от най-често срещаните:

- `'sum'`: Сума на стойностите.
- `'mean'`: Средна стойност.
- `'std'`: Стандартно отклонение.
- `'count'`: Брой на не-NaN стойностите.
- `'min'`: Минимална стойност.
- `'max'`: Максимална стойност.
- `'var'`: Дисперсия.
- `'median'`: Медиана.
- `'first'`: Първа стойност в групата.
- `'last'`: Последна стойност в групата.
- `'nunique'`: Брой на уникалните стойности.

Пример 1: Групиране по една колона и прилагане на `'sum'`:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220],
        'Разходи': [50, 70, 60, 100, 80, 45, 110]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Изчисляване на общите продажби и разходи за всеки град
total_by_city = grouped_by_city[['Продажби', 'Разходи']].agg('sum')
print("Общо продажби и разходи по град:\n", total_by_city)
```

В този пример, след като групирахме по 'Град', избрахме колоните 'Продажби' и 'Разходи' и приложихме функцията `'sum'` към всяка група. Резултатът е `DataFrame`, индексиран по 'Град', с колони 'Продажби' и 'Разходи', съдържащи сумите за всеки град.

Пример 2: Групиране по множество колони и прилагане на `'mean'`:

```
grouped_by_city_product = df_sales.groupby(['Град', 'Продукт'])

# Изчисляване на средните продажби за всяка комбинация от град и продукт
average_sales_by_city_product =
grouped_by_city_product['Продажби'].agg('mean')
print("\nСредни продажби по град и продукт:\n",
average_sales_by_city_product)
```

Тук, групирахме по 'Град' и 'Продукт', след което приложихме функцията 'mean' само към колоната 'Продажби'. Резултатът е Series с MultiIndex ('Град', 'Продукт'), съдържащ средната стойност на продажбите за всяка група.

- Използване на самата функция като аргумент:

Вместо да използваме низови имена на вградени функции, можем директно да подадем самите функции (от библиотеката NumPy или вградени функции на Python) като аргументи на `.agg()` (или `.aggregate()`).

Пример 3: Използване на `numpy.mean` и `numpy.std`:

```
import pandas as pd
import numpy as np

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Изчисляване на средните продажби, използвайки numpy.mean
average_sales_np = grouped_by_city['Продажби'].agg(np.mean)
print("Средни продажби по град (с numpy.mean):\n", average_sales_np)

# Изчисляване на стандартното отклонение на продажбите, използвайки
numpy.std
std_sales_np = grouped_by_city['Продажби'].agg(np.std)
print("\nСтандартно отклонение на продажбите по град (с numpy.std):\n",
std_sales_np)
```

Резултатите ще бъдат Series, индексирани по 'Град', съдържащи съответно средната стойност и стандартното отклонение на продажбите за всеки град.

Пример 4: Използване на вградена функция на Python `len` (в контекста на `count`) и `min`:

Въпреки че `'count'` е по-подходяща за броене на не-NaN стойности, можем да използваме `len` за да видим общия брой на редовете във всяка група.

```
# Изчисляване на броя на записите за всеки град, използвайки len
count_by_city_len = grouped_by_city['Продажби'].agg(len)
print("\nБрой записи по град (с len):\n", count_by_city_len)

# Изчисляване на минималните продажби за всеки град, използвайки min
min_sales_builtin = grouped_by_city['Продажби'].agg(min)
print("\nМинимални продажби по град (с вградена min):\n",
min_sales_builtin)
```

Резултатите ще покажат броя на редовете и минималните продажби за всеки град.

Важно при прилагане на единична функция:

- Функцията, която подавате на `.agg()`, трябва да бъде **агрегираща функция**, т.е., тя трябва да приема `Series` (представляваща колона от групата) като вход и да връща **скаларна стойност** като резултат за тази група.
- Когато прилагате функция към `GroupBy` обект, без да избирате конкретна колона (например, `grouped_by_city.agg('sum')`), функцията ще се опита да бъде приложена към всички колони, за които е смислена (например, числови колони за `'sum'`).

2. Прилагане на множество функции (списък)

За да приложим няколко агрегиращи функции наведнъж към групираните данни, можем да подадем **списък** от функции (като низови имена или самите функции) като аргумент на `.agg()` (или `.aggregate()`). Когато направим това, Pandas ще приложи всяка от предоставените функции към всички подходящи колони в рамките на всяка група.

а) Пример 1: Прилагане на списък от вградени функции към една колона:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Прилагане на sum, mean и std към колона 'Продажби' за всяка група
sales_summary = grouped_by_city['Продажби'].agg(['sum', 'mean', 'std'])
print("Обобщена информация за продажбите по град:\n", sales_summary)
```

Резултатът ще бъде DataFrame, индексиран по 'Град', с колони, съответстващи на приложените функции ('sum', 'mean', 'std'), показващи резултатите от всяка функция за всяка група.

б) Пример 2: Прилагане на списък от функции към множество колони:

Когато приложим списък от функции към целия `GroupBy` обект (или към селекция от няколко колони), всяка функция ще бъде приложена към всяка от избраните колони.

```
# Прилагане на sum и mean към колоните 'Продажби' и 'Разходи' (ако има такава)
sales_expenses_summary = grouped_by_city[['Продажби']].agg(['sum', 'mean'])
print("\nОбобщена информация за продажбите и разходите по град:\n", sales_expenses_summary)
```

Резултатът ще бъде DataFrame с MultiIndex за колоните. Външното ниво ще бъде името на колоната ('Продажби'), а вътрешното ниво ще съдържа имената на приложените функции ('sum', 'mean').

в) Използване на редица от функции:

Вместо списък, можем да подадем и NumPy array от функции. Резултатът ще бъде същият.

```
import numpy as np

sales_summary_numpy = grouped_by_city['Продажби'].agg([np.sum, np.mean, np.std])
print("\nОбобщена информация за продажбите по град (с numpy функции):\n", sales_summary_numpy)
```

г) Преименуване на колоните на резултата:

Когато използваме списък от функции, Pandas автоматично използва имената на функциите като имена на колони (или вътрешно ниво на MultiIndex). Ако искаме да дадем по-ясни имена на тези колони, можем да използваме **речник** като аргумент на `.agg()`, което ще разгледаме в следващата подтема.

Прилагането на множество функции едновременно е удобен начин за получаване на различни обобщени статистики за групите с един ред код. В следващата тема ще видим как да прилагаме различни функции към различни колони.

3. Прилагане на множество функции (речник)

Когато подадем речник на `.aggregate()` (или `.agg()`), **ключовете на речника са имената на колоните**, към които искаме да приложим агрегация, а **стойностите са единични функции (като низ или функция) или списъци от функции**, които да бъдат приложени към съответната колона.

Пример 1: Прилагане на различни функции към различни колони:

Да предположим, че искаме да намерим сумата на продажбите и средната стойност на разходите за всеки град. Можем да използваме речник, за да укажем това:

```
import pandas as pd

data = {'Град': ['София', 'Пловдив', 'София', 'Варна', 'Пловдив',
'София', 'Варна'],
        'Продукт': ['A', 'B', 'A', 'C', 'B', 'C', 'A'],
        'Продажби': [100, 150, 120, 200, 180, 90, 220],
        'Разходи': [50, 70, 60, 100, 80, 45, 110]}
df_sales = pd.DataFrame(data)
grouped_by_city = df_sales.groupby('Град')

# Прилагане на sum към 'Продажби' и mean към 'Разходи'
city_summary_dict = grouped_by_city.agg({'Продажби': 'sum',
                                         'Разходи': 'mean'})
print("Обобщена информация по град (с речник):\n", city_summary_dict)
```

В този случай, резултатът ще бъде DataFrame, индексиран по 'Град', с колони 'Продажби' (съдържаща сумата) и 'Разходи' (съдържаща средната стойност).

Пример 2: Прилагане на множество функции към една колона и различни функции към друга:

Можем също да приложим списък от функции към една колона и единична функция към друга:

```
city_detailed_summary = grouped_by_city.agg({'Продажби': ['sum',
'mean'],
                                             'Разходи': 'max'})
print("\nДетайлна обобщена информация по град (с речник и списък):\n",
city_detailed_summary)
```

Резултатът ще бъде DataFrame с MultiIndex за колоните, където 'Продажби' ще има нива 'sum' и 'mean', а 'Разходи' ще има ниво 'max'.

Преименуване на резултатните колони при използване на речник:

Един от начините за преименуване на колоните, когато използваме речник за прилагане на множество функции към една колона, е като използваме **tuple** като стойност в речника, където първият елемент е името на функцията (като низ), а вторият е желаното име на колоната:

```
city_renamed_summary = grouped_by_city.agg({'Продажби': [('Общо
продажби', 'sum'), ('Средни продажби', 'mean')],
                                            'Разходи': ('Максимални
разходи', 'max')})
print("\nОбобщена информация с преименувани колони:\n",
city_renamed_summary)
```

Резултатът ще има колони 'Продажби' с под-колони 'Общо продажби' и 'Средни продажби', и колона 'Разходи' с под-колона 'Максимални разходи'.

- Обобщение за прилагане на множество функции с речник:

  - Речникът предоставя **прецизен контрол** върху това кои агрегиращи функции се прилагат към кои колони.
  - Можем да прилагаме **различен набор от функции** към различни колони.
  - Използването на tuple-и в стойностите на речника позволява **преименуване на резултатните колони**, което прави резултата по-лесен за разбиране.
  - Когато прилагаме множество функции към една колона с речник (без преименуване), резултатът ще има MultiIndex за колоните.

Използването на речник за агрегация е много мощен и гъвкав подход, особено когато искаме да получим специфични обобщени статистики за различни аспекти на нашите групирани данни.

4. Прилагане на различни функции към различни колони

Основната сила на използването на речник като аргумент на `.agg()` се крие именно в тази гъвкавост. Ето няколко ключови момента и сценария, които подчертават тази възможност:

а) Целенасочена агрегация:

Вместо да прилагаме един и същ набор от агрегиращи функции към всички (числови) колони, можем да бъдем много **селективни**. Например:

- Искаме да намерим средната оценка (`'Оценка'`) и броя на участниците (`'Участници'`) за всяко събитие (`'Събитие'`).
- Искаме да изчислим минималната (`'Цена'`) и максималната (`'Цена'`) стойност, както и броя на уникалните клиенти (`'КлиентID'`) за всеки продукт (`'Продукт'`).

Речникът ни позволява да дефинираме точно тези връзки между колони и функции.

Пример:

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

```
data = {'Събитие': ['Конференция', 'Семинар', 'Конференция', 'Уебинар',
'Семинар'],
        'Оценка': [8.5, 9.2, 7.8, 8.9, 9.5],
        'Участници': [150, 80, 200, 120, 90],
        'Цена': [None, 50, None, None, 60]}
df_events = pd.DataFrame(data)
grouped_by_event = df_events.groupby('Събитие')

# Прилагане на различни функции към различни колони
event_summary = grouped_by_event.agg({'Оценка': 'mean',
                                      'Участници': 'sum',
                                      'Цена': ['min', 'max', 'count']})
print("Обобщена информация за събития:\n", event_summary)
```

Тук виждаме как към колона `'Оценка'` прилагаме `'mean'`, към `'Участници'` - `'sum'`, а към `'Цена'` - списък от функции `['min', 'max', 'count']`. Резултатът от прилагането на множество функции към една колона води до `MultiIndex` за колоните.

б) Избягване на безсмислени агрегации:

Понякога прилагането на определени агрегиращи функции към дадена колона няма смисъл. Например, изчисляването на средната стойност на колона с булеви стойности може да е по-интересно като процент на `True` стойности (което е еквивалентно на средната стойност, но с по-ясна интерпретация). Използването на речник ни позволява да пропуснем прилагането на `mean` към булеви колони, ако това не е желателно, и да се фокусираме върху по-подходящи агрегации като `sum` (за броя на `True` стойностите).

в) Комбиниране на вградени и потребителски функции:

Можем безпроблемно да комбинираме вградени агрегиращи функции с дефинирани от нас потребителски функции в речника за `.agg()`. Това ни дава голяма гъвкавост при анализа.

Пример с потребителска функция:

```
def range_val(series):
    return series.max() - series.min()

event_range_summary = grouped_by_event.agg({'Оценка': ['mean',
range_val],
                                            'Участници': 'sum'})
```

```
print("\nОбобщена информация за събития с потребителска функция:\n",
event_range_summary)
```

Тук, за колона 'Оценка', прилагаме както вградената 'mean', така и нашата потребителска функция range_val.

г) В заключение:

Използването на речник в `.agg()` е ключово, когато искаме:

- Да прилагаме **различен набор** от агрегиращи функции към **различни колони**.
- Да бъдем **селективни** кои агрегации да се извършат върху кои колони.
- Да **комбинираме** вградени и потребителски агрегиращи функции в единна операция.

5. Създаване на потребителски агрегиращи функции

Потребителските агрегиращи функции са изключително полезни, когато вградените функции на Pandas не покриват специфичните нужди на вашия анализ. Те ви позволяват да дефинирате своя собствена логика за обобщаване на данните в рамките на всяка група.

а) Как се създава потребителска агрегираща функция?

Потребителската агрегираща функция е просто **Python функция**, която приема като аргумент **`Series`** (представляваща една колона от текущата група) и **връща една скаларна стойност**. Тази скаларна стойност ще бъде агрегираният резултат за тази колона в тази група.

Пример 1: Функция за изчисляване на разликата между максимална и минимална стойност (range):

```
import pandas as pd

def calculate_range(series):
    return series.max() - series.min()

data = {'Група': ['A', 'A', 'B', 'B', 'A', 'B'],
        'Стойност': [10, 15, 20, 25, 12, 22]}
df = pd.DataFrame(data)
grouped = df.groupby('Група')

# Използване на потребителската функция с .agg()
range_by_group = grouped['Стойност'].agg(calculate_range)
print("Разлика между макс и мин стойност за всяка група:\n",
range_by_group)
```

В този пример, функцията calculate_range приема Series от стойности за всяка група и връща разликата между максималната и минималната стойност. След това използваме .agg() да приложим тази функция към колона 'Стойност' за всяка група, дефинирана от 'Група'.

б) Пример 2: Функция за изчисляване на коефициент на вариация (CV):

Тук, `coefficient_of_variation` изчислява относителната дисперсия на цените във всяка продуктова група.

Пример 3: Използване на `lambda` функция за кратка потребителска агрегация:

За по-кратки потребителски функции, можем да използваме `lambda` изрази директно в `.agg()`:

```
# Изчисляване на медианата, използвайки lambda функция
median_by_group = grouped['Стойност'].agg(lambda x: x.median())
print("\nМедиана на стойностите за всяка група (с lambda):\n",
median_by_group)
```

Прилагане на потребителски функции към различни колони:

Както видяхме и при вградените функции, можем да използваме речник с `.agg()` за да приложим различни потребителски функции към различни колони:

```
price_analysis = grouped_prices.agg({'Цена': [np.mean, calculate_range,
coefficient_of_variation]})
print("\nПодробен анализ на цените по продукт:\n", price_analysis)
```

в) Важно при създаване на потребителски агрегиращи функции:

- **Входен аргумент:** Функцията трябва да приема един аргумент, който ще бъде `Series` (една колона от групата).
- **Връщана стойност:** Функцията трябва да връща **една скаларна стойност** като резултат от агрегацията за тази група и колона.
- **Име на функцията:** Ако подадете функцията директно на `.agg()`, името на колоната в резултата ще бъде името на самата функция. Можете да контролирате имената, като използвате списък от tuple-и `(име, функция)` или речник, както беше показано по-рано.
- **Работа с `NaN` стойности:** Уверете се, че вашата функция обработва `NaN` стойностите по желания от вас начин (например, като ги игнорирате или ги третирате по специален начин).

Създаването на потребителски агрегиращи функции е мощен начин да разширите възможностите на Pandas за анализ на групирани данни и да извършвате специфични за вашия домейн обобщения.

VI. Трансформация (.transform()) - връщане на резултат със същата форма като входа

Нека разгледаме метода `.transform()`, който се използва за **трансформиране на стойностите във всяка група**, но с ключовото свойство, че **връща резултат със същата форма (същия индекс и същата дължина)** като оригиналния `DataFrame`.

Разликата между `.transform()` и `.aggregate()` е съществена:

- **.aggregate()**: Редуцира размера на всяка група до една скаларна стойност (или няколко скаларни стойности, ако се прилагат множество функции). Резултатът има различен индекс от оригиналния `DataFrame` (индексът е ключът за групиране).
- **.transform()**: Прилага функция към всяка група и връща `Series` със същия индекс като оригиналния `DataFrame`. Резултатът от трансформацията се "излъчва" обратно към съответните редове на оригиналния `DataFrame` въз основа на групата, към която принадлежи всеки ред.

1. Как работи `.transform()`?

1) **Групиране:** Първо, данните се групират по указания ключ (колона, `Series`, функция и т.н.), както при `.groupby()`.
2) **Прилагане:** След това, към всяка група се прилага предоставената функция. Тази функция трябва да приема `Series` (представляваща една колона от групата) като вход и да връща `Series` със същия размер като входната група.
3) **Комбиниране:** Накрая, резултатите от всяка група се комбинират обратно в `Series`, който има същия индекс като оригиналния `DataFrame`. Резултатът се подравнява по индекса, така че трансформираната стойност за всеки оригинален ред е на същата позиция.

а) Пример 1: Изчисляване на средната стойност в група:

```
import pandas as pd

data = {'Група': ['A', 'A', 'B', 'B', 'A', 'B'],
        'Стойност': [10, 15, 20, 25, 12, 22]}
df = pd.DataFrame(data)
grouped = df.groupby('Група')

# Трансформиране на колона 'Стойност', като се изчислява средната
стойност за всяка група
mean_value_by_group = grouped['Стойност'].transform('mean')
print("Средна стойност по група (трансформирана):\n",
mean_value_by_group)
```

```
# Сравнете с агрегацията:
mean_value_aggregated = grouped['Стойност'].mean()
print("\nСредна стойност по група (агрегирана):\n",
mean_value_aggregated)

# Забележете, че mean_value_by_group има същия индекс като df,
# докато mean_value_aggregated има индекс 'Група'.
```

В този пример, `.transform('mean')` изчислява средната стойност на колона `'Стойност'` за група 'A' (което е (10 + 15 + 12) / 3 = 12.33) и за група 'B' (което е (20 + 25 + 22) / 3 = 22.33). След това тези средни стойности се "излъчват" обратно към всеки ред, принадлежащ към съответната група.

б) Пример 2: Нормализиране на стойности в група:

Можем да използваме `.transform()` с потребителска функция за по-сложни трансформации:

```
def normalize_within_group(series):
    return (series - series.mean()) / series.std()

normalized_value = grouped['Стойност'].transform(normalize_within_group)
print("\nНормализирани стойности по група:\n", normalized_value)

# Можем да добавим тази трансформирана колона към оригиналния DataFrame
df['Нормализирана стойност'] = normalized_value
print("\nDataFrame с нормализирана стойност:\n", df)
```

Тук, `normalize_within_group` центрира и мащабира стойностите в рамките на всяка група, така че всяка група има средна стойност 0 и стандартно отклонение 1. Резултатът е `Series` със същия индекс като `df`.

в) Пример 3: Използване на `lambda` функция:

```
max_value_by_group = grouped['Стойност'].transform(lambda x: x.max())
print("\nМаксимална стойност по група (трансформирана):\n",
max_value_by_group)
```

```
df['Макс. стойност в група'] = max_value_by_group
print("\nDataFrame с максимална стойност в група:\n", df)
```

2. Кога да използваме `.transform()`?

`.transform()` е полезен в следните сценарии:

- Когато искате да извършите **поелементни операции**, които зависят от статистиката на групата, към която принадлежи елементът.
- Когато искате да **добавите нови колони** към вашия `DataFrame`, които съдържат групови статистики, без да променяте броя на редовете.
- За **центриране, мащабиране или попълване на липсващи стойности** в рамките на всяка група.

VII. Филтриране (.filter()) - премахване на цели групи по условие

Нека разгледаме метода **`.filter()`**, който се използва за **премахване на цели групи** от `GroupBy` обект въз основа на определено условие. За разлика от филтрирането на отделни редове с булеви маски, `.filter()` оценява условието върху всяка група като цяло.

1. Как работи `.filter()`?

1. **Групиране:** Първо, данните се групират по указания ключ (колона, `Series`, функция и т.н.), както при `.groupby()`.
2. **Прилагане на функция:** След това, към всяка група се прилага предоставената функция. Тази функция трябва да приема `DataFrame` (представляващ цялата група) като вход и да връща **една булева стойност** (`True` или `False`).
3. **Филтриране:** Ако функцията върне `True` за дадена група, **всички редове** от тази група се запазват в резултата. Ако функцията върне `False`, **всички редове** от тази група се изключват от резултата.
4. **Комбиниране:** Накрая, запазените редове от всички групи се комбинират в нов `DataFrame`, който има същия индекс като оригиналния (но може да съдържа по-малко редове).

Пример 1: Филтриране на групи въз основа на техния размер:

Да предположим, че искаме да запазим само групите, които съдържат поне 2 записа.

```
import pandas as pd
```

```
data = {'Група': ['A', 'A', 'B', 'B', 'A', 'C'],
        'Стойност': [10, 15, 20, 25, 12, 30]}
df = pd.DataFrame(data)
grouped = df.groupby('Група')

# Филтриране на групи, които имат размер >= 2
filtered_df = grouped.filter(lambda x: len(x) >= 2)
print("DataFrame след филтриране на групи с размер >= 2:\n",
filtered_df)
```

В този пример, lambda функцията lambda x: len(x) >= 2 се прилага към всяка група (x е DataFrame на групата). Групите 'A' (размер 3) и 'B' (размер 2) връщат True, докато група 'C' (размер 1) връща False. Следователно, само редовете от групи 'A' и 'B' се запазват в filtered_df.

Пример 2: Филтриране на групи въз основа на сумата на стойностите:

Да запазим само групите, където сумата на колона `'Стойност'` е по-голяма от 30.

```
filtered_df_sum = grouped.filter(lambda x: x['Стойност'].sum() > 30)
print("\nDataFrame след филтриране на групи със сума на 'Стойност' >
30:\n", filtered_df_sum)
```

Тук, функцията проверява дали сумата на колона 'Стойност' в дадена група е по-голяма от 30. Група 'A' (10 + 15 + 12 = 37) и група 'B' (20 + 25 = 45) отговарят на условието, докато група 'C' (30) не.

Пример 3: Филтриране на групи въз основа на средната стойност:

Да запазим само групите, където средната стойност на колона `'Стойност'` е по-голяма от 15.

```
filtered_df_mean = grouped.filter(lambda x: x['Стойност'].mean() > 15)
print("\nDataFrame след филтриране на групи със средна стойност на
'Стойност' > 15:\n", filtered_df_mean)
```

В този случай, средната стойност за група 'A' е 12.33, за група 'B' е 22.5, а за група 'C' е 30. Следователно, само редовете от групи 'B' и 'C' ще бъдат в резултата.

2. Важно при използване на `.filter()`:

- Функцията, подадена на `.filter()`, трябва да приема **цялата група като `DataFrame`** и да връща **една булева стойност**.
- Филтрирането се извършва на ниво **група**, а не на ниво отделен ред. Ако една група отговаря на условието, всички редове в нея се запазват.
- Резултатът от `.filter()` е `DataFrame` със същия индекс като оригиналния, но може да съдържа по-малко редове (ако някои групи са били филтрирани).

`.filter()` е мощен инструмент за селективно запазване на цели подмножества от данни въз основа на групови характеристики.

VIII. Гъвкаво прилагане (.apply()) - прилагане на произволна функция към всяка група

Метода **`.apply()`**, който е изключително гъвкав и позволява да се прилага **произволна функция** към всяка група на `GroupBy` обекта. `.apply()` е по-общ от `.aggregate()` и `.transform()` и може да се използва за много различни задачи, включително агрегация, трансформация и дори филтрация (макар че за филтрация има по-специализиран метод `.filter()`).

1. Как работи `.apply()`?

1. **Групиране:** Първо, данните се групират по указания ключ.
2. **Прилагане на функция:** Предоставената функция се прилага към **всяка група като цяло**. Функцията получава `DataFrame` (или `Series`, ако оригиналният обект е `Series`) на текущата група като аргумент.
3. **Комбиниране на резултати:** Резултатите от прилагането на функцията към всяка група се комбинират обратно в един `DataFrame` или `Series`. Формата на комбинирания резултат зависи от това какво връща функцията за всяка група.

2. Гъвкавостта на `.apply()`:

Основната сила на `.apply()` е, че функцията, която му се подава, може да върне:

- **Скаларна стойност:** В този случай `.apply()` ще действа подобно на агрегация, като за всяка група ще има един резултат.
- **`Series`:** Резултатът от `.apply()` ще бъде `DataFrame`, където индексът на `Series`-а, върнат от всяка група, ще стане индекс на редовете в съответната група в резултатния `DataFrame`.

Индексът на групиране ще стане външно ниво на MultiIndex, ако групирането е било по повече от една колона.

- **`DataFrame`:** Подобно на връщането на `Series`, резултатът ще бъде `DataFrame`, където индексът и колоните на `DataFrame`-а, върнат от всяка група, ще се комбинират.

Пример 1: Изчисляване на диапазона (разлика между макс и мин) с `.apply()` (подобно на потребителска агрегация):

```
import pandas as pd

def range_apply(group):
    return group['Стойност'].max() - group['Стойност'].min()

data = {'Група': ['A', 'A', 'B', 'B', 'A'],
        'Стойност': [10, 15, 20, 25, 12]}
df = pd.DataFrame(data)
grouped = df.groupby('Група')

range_by_group_apply = grouped.apply(range_apply)
print("Диапазон на стойностите за всяка група (с apply):\n",
range_by_group_apply)
```

В този случай, функцията range_apply приема DataFrame на всяка група и връща скаларна стойност (диапазона). Резултатът е Series, индексиран по името на групата.

Пример 2: Намиране на топ 2 стойности във всяка група:

```
def top_two(group):
    return group.nlargest(2, 'Стойност')

top_two_by_group = grouped.apply(top_two)
print("\nТоп 2 стойности във всяка група:\n", top_two_by_group)
```

Тук, функцията top_two приема DataFrame на всяка група и връща DataFrame, съдържащ двата най-големи реда според колона 'Стойност'. Резултатът е DataFrame с MultiIndex (група и оригинален индекс).

Пример 3: Прилагане на функция, която връща `Series`:

```
def summarize_group(group):
    return pd.Series({'средно': group['Стойност'].mean(),
                      'медиана': group['Стойност'].median(),
                      'брой': len(group)})

summary_by_group = grouped.apply(summarize_group)
print("\nОбобщена статистика за всяка група (с apply, връщащ
Series):\n", summary_by_group)
```

Функцията summarize_group връща Series с няколко обобщаващи статистики за всяка група. Резултатът е DataFrame, където индексът е името на групата, а колоните са имената от Series-а, върнат от функцията.

3. Кога да използваме `.apply()`?

`.apply()` е най-подходящ, когато:

- Искате да извършите **сложни операции** върху всяка група, които не могат лесно да бъдат изразени с вградените функции за агрегация или трансформация.
- Вашата функция **трябва да работи с целия `DataFrame` на групата**, а не само с отделни колони.
- Функцията ви може да **върне резултат с произволна форма** (скалар, `Series`, `DataFrame`).

4. Внимание при използване на `.apply()`:

Въпреки гъвкавостта си, `.apply()` може да бъде **по-бавен** от специализираните методи `.agg()` и `.transform()`, особено при големи набори от данни, тъй като може да включва повече overhead при извикването на функцията за всяка група. За често срещани операции е препоръчително да се използват по-оптимизираните методи, ако е възможно.

Въпреки това, `.apply()` е незаменим инструмент, когато се нуждаете от пълна гъвкавост при обработката на групирани данни.

Казус: Анализ на данни за онлайн магазин

Представете си, че работите за онлайн магазин и имате данни за продажбите, съхранявани в `DataFrame`, който съдържа следната информация:

- `CustomerID`: Уникален идентификатор на клиента.
- `OrderID`: Уникален идентификатор на поръчката.
- `OrderDate`: Дата на поръчката.

- `ProductCategory`: Категория на закупения продукт.
- `ProductName`: Име на закупения продукт.
- `Quantity`: Количество на закупения продукт в поръчката.
- `UnitPrice`: Единична цена на продукта.

Вашата задача е да анализирате тези данни, за да получите различни статистики и прозрения за продажбите на магазина.

Стъпки за анализ и решение:

1) **Зареждане на данните:** Първо, трябва да заредим данните в `DataFrame`. (За целите на примера, ще създадем `DataFrame` с примерни данни).

```
import pandas as pd
import numpy as np

# Създаване на примерни данни
data = {
    'CustomerID': [1, 1, 2, 2, 3, 3, 1, 2, 3, 4, 4, 4],
    'OrderID': [101, 102, 201, 202, 301, 302, 103, 203, 303, 401, 402,
403],
    'OrderDate': pd.to_datetime(['2023-01-05', '2023-01-10', '2023-01-
15', '2023-01-20', '2023-02-01', '2023-02-05', '2023-02-10', '2023-02-
15', '2023-02-20', '2023-03-01', '2023-03-05', '2023-03-10']),
    'ProductCategory': ['Electronics', 'Books', 'Electronics',
'Clothing', 'Books', 'Electronics', 'Clothing', 'Books', 'Electronics',
'Clothing', 'Electronics', 'Books'],
    'ProductName': ['Laptop', 'Fiction Book', 'Smartphone', 'T-Shirt',
'Science Book', 'Headphones', 'Jeans', 'Mystery Novel', 'Smartwatch',
'Jacket', 'Tablet', 'Cookbook'],
    'Quantity': [1, 2, 1, 3, 1, 2, 2, 1, 1, 1, 1, 3],
    'UnitPrice': [1200, 15, 800, 25, 20, 100, 40, 18, 300, 60, 400, 12]
}
df_sales = pd.DataFrame(data)

# Добавяне на колона за обща цена на артикул в поръчката
df_sales['TotalPrice'] = df_sales['Quantity'] * df_sales['UnitPrice']

print("Оригинални данни за продажби:\n", df_sales)
```

2) **Създаване на GroupBy обект (Групиране по една или множество колони):**

а) Да намерим общата стойност на покупките за всеки клиент. Групираме по `CustomerID`.

```
customer_grouped = df_sales.groupby('CustomerID')
print("\nGroupBy обект (групиран по CustomerID):\n", customer_grouped)
```

б) Да анализираме средното количество и средната цена на артикулите, закупени във всяка категория продукти. Групираме по `ProductCategory`.

```
category_grouped = df_sales.groupby('ProductCategory')
print("\nGroupBy обект (групиран по ProductCategory):\n",
category_grouped)
```

в) Да разгледаме общата стойност на продажбите за всяка комбинация от категория и продукт. Групираме по `ProductCategory` и `ProductName`.

```
product_grouped = df_sales.groupby(['ProductCategory', 'ProductName'])
print("\nGroupBy обект (групиран по ProductCategory и ProductName):\n",
product_grouped)
```

3) **Инспекция на GroupBy обект:**

а) Да видим кои са групите, създадени при групиране по `CustomerID`.

```
print("\nГрупи по CustomerID (.groups):\n", customer_grouped.groups)
```

б) Да извлечем данните за клиент с `CustomerID` 2.

```
customer_2_data = customer_grouped.get_group(2)
print("\nДанни за CustomerID 2 (.get_group(2)):\n", customer_2_data)
```

в) Да преброим броя на поръчките, направени от всеки клиент.

```
customer_order_count = customer_grouped['OrderID'].size()
print("\nБрой поръчки на клиент (.size()):\n", customer_order_count)
```

г) Да видим първата поръчка за всеки клиент.

```
first_order_by_customer = customer_grouped.first()
print("\nПърва поръчка на клиент (.first()):\n",
first_order_by_customer)
```

д) Да видим последната поръчка за всеки клиент.

```
last_order_by_customer = customer_grouped.last()
print("\nПоследна поръчка на клиент (.last()):\n",
last_order_by_customer)
```

4) Агрегация:

а) Да намерим общата сума, похарчена от всеки клиент.

```
total_spent_by_customer = customer_grouped['TotalPrice'].sum()
print("\nОбща сума, похарчена от клиент (единична функция 'sum'):\n",
total_spent_by_customer)
```

б) Да изчислим средното количество и средната единична цена за всяка категория продукти.

```
category_stats = category_grouped[['Quantity',
'UnitPrice']].agg(['mean'])
print("\nСредно количество и цена по категория (множество функции -
списък):\n", category_stats)
```

в) Да намерим общата стойност и средното количество за всяка категория.

```
category_summary = category_grouped.agg({'TotalPrice': 'sum',
'Quantity': 'mean'})
print("\nОбща стойност и средно количество по категория (различни
функции за различни колони - речник):\n", category_summary)
```

г) Да създадем потребителска функция за намиране на диапазона на цените (максимална - минимална) за всяка категория.

```
def price_range(series):
    return series.max() - series.min()

category_price_range = category_grouped['UnitPrice'].agg(price_range)
print("\nДиапазон на цените по категория (потребителска функция):\n",
category_price_range)
```

5) Трансформация:

а) Да изчислим какъв процент от общата сума, похарчена от клиента, представлява всяка негова поръчка.

```
df_sales['CustomerTotalSpend'] =
customer_grouped['TotalPrice'].transform('sum')
df_sales['PercentageOfCustomerSpend'] = (df_sales['TotalPrice'] /
df_sales['CustomerTotalSpend']) * 100
print("\nПроцент от общата сума, похарчена от клиента за всяка поръчка
(трансформация):\n", df_sales[['CustomerID', 'OrderID', 'TotalPrice',
'PercentageOfCustomerSpend']])
```

6) Филтриране:

а) Да оставим само клиентите, които са направили повече от 1 поръчка.

```
customers_with_multiple_orders = customer_grouped.filter(lambda x:
len(x) > 1)
print("\nКлиенти с повече от 1 поръчка (филтриране):\n",
customers_with_multiple_orders['CustomerID'].unique())
```

б) Да оставим само категориите продукти, където средната единична цена е над 50.

```
categories_expensive = category_grouped.filter(lambda x:
x['UnitPrice'].mean() > 50)
print("\nКатегории със средна цена над 50 (филтриране):\n",
categories_expensive['ProductCategory'].unique())
```

7) Гъвкаво прилагане (`.apply()`):

а) Да намерим най-скъпия продукт, закупен от всеки клиент.

```
def most_expensive_item(group):
    return group.nlargest(1, 'TotalPrice')

most_expensive_by_customer = customer_grouped.apply(most_expensive_item)
print("\nНай-скъпият продукт, закупен от всеки клиент (.apply()):\n",
most_expensive_by_customer)
```

б) Да създадем обобщена статистика за всяка категория, включваща средна цена и брой уникални клиенти, закупили продукти от тази категория.

```
def category_analysis(group):
    unique_customers = group['CustomerID'].nunique()
    average_price = group['UnitPrice'].mean()
    return pd.Series({'Средна цена': average_price, 'Уникални клиенти':
unique_customers})

category_analysis_summary = category_grouped.apply(category_analysis)
print("\nОбобщен анализ по категория (.apply()):\n",
category_analysis_summary)
```

Този казус и неговото решение демонстрират как можем да използваме различните аспекти на GroupBy в Pandas за анализ на данни от реален свят, като обхващат създаване на GroupBy обекти по различни начини, инспекция, агрегация с различни функции, трансформация, филтриране на групи и гъвкаво прилагане на персонализирани функции. Всяка стъпка е придружена от код и описание на нейната цел.

КАЗУС: Анализ на данни за представянето на ученици по различни предмети.

ЗАДАЧА:

Да се анализират резултатите на ученици от различни класове по няколко предмета, като се извлекат статистики за всеки клас и предмет, да се идентифицират класове с по-добро представяне и да се трансформират резултатите за сравнение.

ВХОДНИ ДАННИ:

```
import pandas as pd

data = {
    'Клас': ['A', 'A', 'B', 'B', 'A', 'B', 'A', 'B', 'C', 'C', 'C',
'A'],
    'Предмет': ['Математика', 'Български', 'Математика', 'Български',
'История', 'История', 'Физика', 'Физика', 'Математика', 'Български',
'История', 'Химия'],
```

```
    'Ученик': ['Иван', 'Петър', 'Мария', 'Анна', 'Георги', 'Стефан',
'Димитър', 'Елена', 'Николай', 'Виктория', 'Александър', 'София'],
    'Резултат': [85, 92, 78, 88, 90, 82, 75, 86, 65, 72, 79, 95]
}
df_резултати = pd.DataFrame(data)

print("Входни данни:\n", df_резултати)
```

СКРИПТ НА РЕШЕНИЕТО:

```
import pandas as pd

data = {
    'Клас': ['A', 'A', 'B', 'B', 'A', 'B', 'A', 'B', 'C', 'C', 'C',
'A'],
    'Предмет': ['Математика', 'Български', 'Математика', 'Български',
'История', 'История', 'Физика', 'Физика', 'Математика', 'Български',
'История', 'Химия'],
    'Ученик': ['Иван', 'Петър', 'Мария', 'Анна', 'Георги', 'Стефан',
'Димитър', 'Елена', 'Николай', 'Виктория', 'Александър', 'София'],
    'Резултат': [85, 92, 78, 88, 90, 82, 75, 86, 65, 72, 79, 95]
}
df_резултати = pd.DataFrame(data)

# 1. Групиране по клас и предмет
групирани_резултати = df_резултати.groupby(['Клас', 'Предмет'])

# 2. Агрегиране: Среден резултат, минимален резултат, максимален
резултат и брой ученици за всеки клас и предмет
статистики_клас_предмет = групирани_резултати['Резултат'].agg(['mean',
'min', 'max', 'count'])
print("\nСтатистики за всеки клас и предмет:\n",
статистики_клас_предмет)

# 3. Агрегиране: Среден резултат за всеки клас
```

```
среден_резултат_клас = df_резултати.groupby('Клас')['Резултат'].mean()
print("\nСреден резултат за всеки клас:\n", среден_резултат_клас)

# 4. Филтриране: Класове със среден резултат над 80
класове_добро_представяне = среден_резултат_клас[среден_резултат_клас > 80]
print("\nКласове със среден резултат над 80:\n", класове_добро_представяне)

# 5. Трансформация: Центриране на резултатите във всеки предмет (изваждане на средния резултат за предмета)
среден_резултат_предмет = df_резултати.groupby('Предмет')['Резултат'].transform('mean')
df_резултати['Центриран_резултат'] = df_резултати['Резултат'] - среден_резултат_предмет
print("\nДанни с центрирани резултати по предмет:\n", df_резултати[['Клас', 'Предмет', 'Ученик', 'Резултат', 'Центриран_резултат']])

# 6. Прилагане (.apply()): Намиране на ученика с най-висок резултат за всеки клас и предмет
def най_добър_ученик(група):
    return група.nlargest(1, 'Резултат')

най_добър_по_клас_предмет = групирани_резултати.apply(най_добър_ученик)
print("\nНай-добър ученик по клас и предмет:\n", най_добър_по_клас_предмет)
```

ОПИСАНИЕ НА РЕШЕНИЕТО (ПО СТЪПКИ):

1. **Групиране по клас и предмет:** Използва се `.groupby(['Клас', 'Предмет'])` за създаване на `GroupBy` обект, който групира резултатите по уникални комбинации от клас и предмет.
2. **Агрегиране на статистики за клас и предмет:**
   - `.agg(['mean', 'min', 'max', 'count'])` се прилага към колона `'Резултат'` на групирания обект.
   - Изчисляват се средният, минималният, максималният резултат и броят на учениците за всяка група (комбинация от клас и предмет).

- Резултатът е `DataFrame` с MultiIndex (Клас, Предмет) и колони за всяка от агрегиращите функции.

3. **Агрегиране на среден резултат по клас:**
   - `.groupby('Клас')['Резултат'].mean()` се използва за групиране само по клас и изчисляване на средния резултат за всеки клас.
   - Резултатът е `Series`, индексиран по клас, съдържащ средния резултат.
4. **Филтриране на класове с добро представяне:**
   - Използва се булева индексация върху `среден_резултат_клас` за филтриране на класовете, чийто среден резултат е по-висок от 80.
5. **Трансформация на резултатите:**
   - `.groupby('Предмет')['Резултат'].transform('mean')` се използва за изчисляване на средния резултат за всеки предмет и след това за "излъчване" на тази средна стойност обратно към всички редове, принадлежащи към същия предмет.
   - Създава се нова колона `'Центриран_резултат'`, която съдържа разликата между индивидуалния резултат на ученика и средния резултат за съответния предмет. Това позволява сравнение на представянето на учениците спрямо средното за предмета, елиминирайки разликата в трудността между предметите.
6. **Прилагане за намиране на най-добър ученик:**
   - Дефинира се потребителска функция `най_добър_ученик`, която приема `DataFrame` на група (клас и предмет) и връща реда с най-високия резултат, използвайки `.nlargest(1, 'Резултат')`.
   - `.apply(най_добър_ученик)` се прилага към групирания по клас и предмет обект, за да се намери най-добрият ученик във всяка комбинация. Резултатът е `DataFrame` с MultiIndex (Клас, Предмет) и съдържа информацията за най-добрия ученик.

Този казус демонстрира използването на всички основни концепции от главата "Групиране и Агрегиране" за анализ на образователни данни.

ВЪПРОСИ:

1. Обяснете с прости думи концепцията "Раздели-Приложи-Комбинирай" (Split-Apply-Combine) в контекста на Pandas. Дайте кратък пример.
2. Каква е разликата между `.aggregate()` и `.transform()`? В какви сценарии бихте използвали всеки от тях?
3. Какво връща методът `.groupby()`? Как можем да видим реалните групирани данни?
4. Опишете различните начини за създаване на `GroupBy` обект (по колони, `Series`, индекс, функция). Дайте по един кратък пример за всеки начин.
5. Каква е целта на метода `.filter()` при работа с `GroupBy` обекти? Как се определя кои групи се запазват?
6. В какви ситуации би било полезно да итерираме през групите на `GroupBy` обект? Какъв тип обекти получаваме при всяка итерация?
7. Обяснете как можем да приложим множество агрегиращи функции едновременно, използвайки списък и речник с метода `.agg()`. Как се различава резултатът в двата случая?
8. Как можем да създадем и използваме потребителски агрегиращи функции с `.agg()`? Какви са основните изисквания към тези функции?
9. Кога бихте използвали метода `.apply()` вместо `.agg()` или `.transform()`? Какви са възможните типове резултати от функция, приложена с `.apply()`?
10. Какви са методите за инспектиране на `GroupBy` обект, които научихте? Обяснете накратко какво връща всеки от тях.

ЗАДАЧИ:

Използвайте следния `DataFrame` за решаване на задачите по-долу:

```
import pandas as pd


data = {
    'Отдел': ['Продажби', 'Маркетинг', 'Продажби', 'Развитие',
'Маркетинг', 'Развитие', 'Продажби', 'Маркетинг'],
    'Служител': ['Алиса', 'Борис', 'Цветан', 'Диана', 'Елена', 'Георги',
'Иван', 'Жана'],
    'Пол': ['Ж', 'М', 'М', 'Ж', 'Ж', 'М', 'М', 'Ж'],
    'Години': [25, 32, 40, 28, 35, 45, 30, 26],
    'Заплащане': [5000, 6000, 7500, 8000, 6500, 9000, 7000, 5500],
    'Бонус': [500, None, 750, 1000, 650, 1200, 700, None]
}
df_employees = pd.DataFrame(data)
print("\nDataFrame за задачите:\n", df_employees)
```

1. Групирайте `df_employees` по колона `'Отдел'`.
   а) Изведете всички служители в отдел 'Продажби'.
   б) Намерете средната възраст и средното заплащане за всеки отдел.
   в) Пребройте броя на служителите във всеки отдел.
   г) Намерете служителя с най-високо заплащане във всеки отдел.

2. Групирайте `df_employees` по колона `'Пол'`.
   а) Намерете средното заплащане за мъжете и жените.
   б) Изведете броя на мъжете и жените във всеки отдел. *(Подсказка: може да се наложи да групирате по повече от една колона и след това да използвате `.size()` или `.count()`)*

3. Групирайте `df_employees` по колона `'Години'`, като използвате функция, която връща възрастова група ('Млади', 'Средна възраст', 'Възрастни'). Дефинирайте сами границите на тези групи. Намерете средното заплащане за всяка възрастова група.

4. Използвайте `.transform()` за да добавите колона към df_employees, която съдържа средното заплащане за отдела на всеки служител.

5. Използвайте `.filter()` за да запазите само отделите, в които има повече от 2 служители.

6. Използвайте `.apply()` за да намерите служителя с най-нисък бонус (ако има такъв, в противен случай върнете `None`) за всеки отдел.

7. Групирайте `df_employees` по `'Отдел'` и за всяка група приложете функция, която връща `DataFrame` само с жените от този отдел.