Глава III. Основно Разглеждане и Инспекция на Данни

Тази глава е фундаментална за всякакъв анализ на данни с Pandas, тъй като преди да можем да трансформираме, почистваме или моделираме данните си, е абсолютно необходимо да ги разберем. Ще се фокусираме върху техники и инструменти, които ни позволяват да получим бърз и ефективен поглед върху съдържанието, структурата и основните характеристики на нашите DataFrame-и и Series обекти.

Съдържание на главата:

В рамките на тази глава ще разгледаме следните основни аспекти:

- **Първи впечатления:** Как бързо да видим първите и последните няколко реда от нашия набор от данни, за да получим представа за съдържанието и формата му (`.head()`, `.tail()`).
- **Структурен преглед:** Как да получим обобщена информация за структурата на DataFrame-а, включително броя на редовете и колоните, типовете данни на всяка колона и използването на памет (`.info()`).
- **Статистически резюмета:** Как да генерираме описателни статистики за числовите колони, като средна стойност, медиана, стандартно отклонение, квартили и други. Ще разширим това и за да видим обобщена информация за категорийни и времеви данни (`.describe()`).
- **Размерност и обем:** Как да определим размера и формата на нашия DataFrame, включително броя на редовете и колоните (`.shape`) и общия брой на елементите (`.size`).
- **Типове данни в детайл:** Как да проверим типа на данните в отделна колона (`.dtype`) и типовете данни на всички колони едновременно (`.dtypes`).
- **Индекси и колони:** Как да извлечем информация за индекса (редовите етикети) и колоните (имената на колоните) на DataFrame-а (`.index`, `.columns`).
- **Уникални стойности и честота:** Как да идентифицираме уникалните стойности в колона, да преброим техния брой (`.unique()`, `.nunique()`) и да определим честотата на всяка стойност (`.value_counts()`).
- **Транспониране:** Как да разменим редовете и колоните на DataFrame-а (`.T`), което може да бъде полезно за определени видове анализ или визуализация.

Чрез тези методи ще придобием основни умения за първоначално изследване на данните, което е критична стъпка преди всякакви по-нататъшни операции. Разбирането на структурата, типовете данни и основните статистически характеристики ще ни помогне да вземаме информирани решения за това как да почистваме, трансформираме и анализираме нашите данни.

I. Преглед на първите и последните редове (.head(), .tail())

.head(), .tail() са два метода са изключително полезни за бързо придобиване на първоначална представа за съдържанието на DataFrame-а, веднага след като сме го прочели от някакъв източник. Те ни позволяват да видим примерни данни и да се ориентираме в структурата и типа на информацията, която съдържа.

1. `.head()` - Преглед на първите редове

Методът `.head(n=5)` връща първите `n` реда от DataFrame-а. По подразбиране, ако не укажем стойност за `n`, той връща първите 5 реда.

- *Синтаксис:*

```
dataframe.head(n=5)
```

където dataframe е вашият Pandas DataFrame обект, а n е броят на редовете, които искате да видите.

- *Пример:*

Представете си, че имаме DataFrame, съдържащ данни за продажби:

```
import pandas as pd

data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

# Преглед на първите 5 реда (по подразбиране)
print("Първите 5 реда:")
print(df_sales.head())

# Преглед на първите 3 реда
print("\nПървите 3 реда:")
```

```
print(df_sales.head(3))
```

Резултат:

```
Първите 5 реда:
  продукт  цена  количество
0       А  10.5           5
1       Б  20.0           2
2       А  10.5           3
3       В  15.0           1
4       Г  25.0           4

Първите 3 реда:
  продукт  цена  количество
0       А  10.5           5
1       Б  20.0           2
2       А  10.5           3
```

Както виждате, `.head()` ни дава бърз поглед върху структурата на DataFrame-а (имена на колони) и примерни стойности в първите няколко реда. Това може да ни помогне да се уверим, че данните са прочетени правилно и изглеждат както очакваме.

2. `.tail()` - Преглед на последните редове

Методът `.tail(n=5)` работи по аналогичен начин, но връща последните `n` реда от DataFrame-а. По подразбиране, ако не укажем стойност за `n`, той връща последните 5 реда.

- *Синтаксис:*

```
dataframe.tail(n=5)
```

където dataframe е вашият Pandas DataFrame обект, а n е броят на последните редове, които искате да видите.

- *Пример (използвайки същия `df_sales` DataFrame):*

```
# Преглед на последните 5 реда (по подразбиране)
print("\nПоследните 5 реда:")
print(df_sales.tail())

# Преглед на последните 2 реда
print("\nПоследните 2 реда:")
print(df_sales.tail(2))
```

Резултат:

```
Последните 5 реда:
  продукт  цена  количество
3       В  15.0           1
4       Г  25.0           4
5       Б  20.0           6
6       Г  25.0           2
7       А  10.5          10

Последните 2 реда:
  продукт  цена  количество
6       Г  25.0           2
7       А  10.5          10
```

.tail() е полезен, когато искаме да видим как завършва нашият набор от данни, особено при времеви серии (за да видим последните наблюдения) или когато искаме да проверим дали определени операции са приложени коректно в края на DataFrame-а.

3. Предимства на `.head()` и `.tail()`:

- **Бърз преглед:** Осигуряват бърз начин да се запознаем с данните.
- **Проверка на коректност:** Помагат да се уверим, че данните са прочетени правилно и изглеждат очаквано.
- **Разбиране на структурата:** Показват имената на колоните и типа на данните (приблизително).

II. Получаване на информация за структурата (.info())

Методът `.info()` е изключително полезен инструмент за получаване на сбита сводка за DataFrame-а. Той предоставя ключова информация за структурата на данните, която е важна за разбирането на техните основни характеристики и за идентифициране на потенциални проблеми.

1. Синтаксис:

```
dataframe.info(verbose=True, buf=None, max_info_columns=100,
memory_usage='deep', show_counts=True)
```

Въпреки че има няколко параметъра, най-често се използва просто върху DataFrame обекта без допълнителни аргументи:

```
dataframe.info()
```

*2. Информация, която **.info()** предоставя:*

Когато извикате `.info()`, ще видите изход, който включва следната информация:

- **Брой на редовете (entries):** Показва общия брой на редовете (индекси) в DataFrame-а.
- **Брой на колоните (columns):** Показва общия брой на колоните в DataFrame-а.
- **Имена на колоните:** Изброява имената на всички колони в DataFrame-а.
- **Брой не-Null стойностите (Non-Null Count) за всяка колона:** Това е изключително важна информация, която показва колко стойности във всяка колона не са липсващи (`NaN`). Сравнявайки броя на не-Null стойностите с общия брой на редовете, можем бързо да идентифицираме колони с липсващи данни.
- **Тип на данните (Dtype) за всяка колона:** Показва типа на данните, съхранявани във всяка колона (например, `int64`, `float64`, `object`, `datetime64[ns]`, `bool`, `category`). Разбирането на типовете данни е важно за последващи анализи и операции.
- **Използване на памет (Memory usage):** Показва приблизителното количество памет, което DataFrame-ът заема. Това може да бъде полезно при работа с големи набори от данни за оптимизиране на използването на паметта.

3. Пример (използвайки отново `df_sales`):

```
import pandas as pd
```

```
data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

print("Информация за df_sales:")
df_sales.info()
```

Резултат:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8 entries, 0 to 7
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   продукт     8 non-null      object
 1   цена        8 non-null      float64
 2   количество  8 non-null      int64 
dtypes: float64(1), int64(1), object(1)
memory usage: 320.0 bytes
```

4. Интерпретация на резултата:

- `<class 'pandas.core.frame.DataFrame'>`: Показва, че обектът е Pandas DataFrame.
- `RangeIndex: 8 entries, 0 to 7`: DataFrame-ът има 8 реда (индексирани от 0 до 7).
- `Data columns (total 3 columns)`: DataFrame-ът има 3 колони.
- `# Column Non-Null Count Dtype`: Следва списък на колоните:
  - `продукт`: 8 не-Null стойности, тип `object` (обикновено означава string или смесен тип).
  - `цена`: 8 не-Null стойности, тип `float64` (числа с плаваща запетая).
  - `количество`: 8 не-Null стойности, тип `int64` (цели числа).
- `dtypes: float64(1), int64(1), object(1)`: Обобщение на типовете данни - една колона е `float64`, една е `int64` и една е `object`.
- `memory usage: 320.0 bytes`: DataFrame-ът използва приблизително 320 байта памет.

5. *Пример с липсващи стойности:*

Нека създадем DataFrame с няколко липсващи стойности, за да видим как се отразява това в `.info()`:

```
import pandas as pd
import numpy as np

data_missing = {'колона1': [1, 2, np.nan, 4, 5],
                'колона2': ['a', np.nan, 'c', 'd', 'e'],
                'колона3': [True, False, True, np.nan, False]}
df_missing = pd.DataFrame(data_missing)

print("\nИнформация за df_missing:")
df_missing.info()
```

Резултат:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   колона1  4 non-null      float64
 1   колона2  4 non-null      object
 2   колона3  4 non-null      object
dtypes: float64(1), object(2)
memory usage: 248.0 bytes
```

6. *Интерпретация на резултата с липсващи стойности:*

Забележете, че за всяка колона броят на `Non-Null Count` е 4, докато общият брой на редовете е 5. Това ясно показва, че всяка колона съдържа една липсваща стойност (`NaN`). Типът на данните за `колона3` е `object`, тъй като Pandas не може автоматично да определи еднозначен булев тип, когато има липсващи стойности (в по-нови версии на Pandas може да се запази като `bool` с `pd.NA`).

Методът `.info()` е основен инструмент за първоначално разбиране на вашите данни и трябва да бъде една от първите команди, които изпълнявате след като прочетете данни в Pandas DataFrame.

III. Описателна статистика (.describe()) - включително за категорийни и времеви данни

Методът `.describe()` генерира обобщена статистика за числовите колони в DataFrame-а по подразбиране. Тази статистика включва централна тенденция, дисперсия и форма на разпределението на набора от данни, като изключва `NaN` стойностите.

1. Синтаксис:

```
dataframe.describe(percentiles=None, include=None, exclude=None,
datetime_is_numeric=False)
```

Повечето от параметрите са опционални и позволяват да се контролира кои колони и какви процентили да бъдат включени в резултата. Най-често се използва просто:

```
dataframe.describe()
```

2. Статистики, които `.describe()` предоставя за числови колони:

- **count:** Брой на не-Null стойностите.
- **mean:** Средна стойност.
- **std:** Стандартно отклонение.
- **min:** Минимална стойност.
- **25% (Q1):** Първи квартил (25-ти персентил).
- **50% (median или Q2):** Медиана (50-ти персентил).
- **75% (Q3):** Трети квартил (75-ти персентил).
- **max:** Максимална стойност.

3. Пример (използвайки `df_sales`):

```
import pandas as pd
```

```
data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

print("Описателна статистика за df_sales:")
print(df_sales.describe())
```

Резултат:

```
цена  количество
count  8.000000   8.000000
mean  17.062500   4.125000
std    6.480823   2.953586
min   10.500000   1.000000
25%   10.500000   2.000000
50%   17.500000   3.500000
75%   21.250000   4.500000
max   25.000000  10.000000
```

Резултатът показва описателна статистика само за числовите колони (`цена` и `количество`). Колоната `продукт` (тип `object`) е игнорирана по подразбиране.

4. Описателна статистика за категорийни данни

За да получим описателна статистика и за категорийни (и други не-числови) колони, можем да използваме параметъра `include` на `.describe()`.

```
dataframe.describe(include=['object'])  # За категорийни (string) колони
dataframe.describe(include=['category']) # Ако имаме колони с Pandas
Categorical dtype
dataframe.describe(include='all')       # За всички колони
```

- ***Статистики, които `.describe()` предоставя за категорийни колони:***

- **count:** Брой на не-Null стойностите.
- **unique:** Брой на уникалните стойности.
- **top:** Най-често срещаната стойност.
- **freq:** Честота на най-често срещаната стойност.

- ***Пример (използвайки `df_sales`):***

```
print("\nОписателна статистика за категорийни колони в df_sales:")
print(df_sales.describe(include=['object']))

print("\nОписателна статистика за всички колони в df_sales:")
print(df_sales.describe(include='all'))
```

Резултат:

```
    продукт
count         8
unique        4
top           A
freq          3

Описателна статистика за всички колони в df_sales:
       продукт       цена  количество
count        8   8.000000    8.000000
unique       4        NaN         NaN
top          A        NaN         NaN
freq         3        NaN         NaN
mean       NaN  17.062500    4.125000
std        NaN   6.480823    2.953586
min        NaN  10.500000    1.000000
25%        NaN  10.500000    2.000000
50%        NaN  17.500000    3.500000
```

```
75%          NaN   21.250000    4.500000
max          NaN   25.000000   10.000000
```

Резултатът показва описателна статистика само за числовите колони (цена и количество). Колоната продукт (тип object) е игнорирана по подразбиране.

5. Описателна статистика за категорийни данни

За да получим описателна статистика и за категорийни (и други не-числови) колони, можем да използваме параметъра `include` на `.describe()`.

- ***Синтаксис:***

```
dataframe.describe(include=['object'])  # За категорийни (string) колони
dataframe.describe(include=['category']) # Ако имаме колони с Pandas
Categorical dtype
dataframe.describe(include='all')       # За всички колони
```

- ***Статистики, които `.describe()` предоставя за категорийни колони:***

- **count:** Брой на не-Null стойностите.
- **unique:** Брой на уникалните стойности.
- **top:** Най-често срещаната стойност.
- **freq:** Честота на най-често срещаната стойност.

- ***Пример (използвайки `df_sales`):***

```
print("\nОписателна статистика за категорийни колони в df_sales:")
print(df_sales.describe(include=['object']))

print("\nОписателна статистика за всички колони в df_sales:")
print(df_sales.describe(include='all'))
```

Резултат:

```
       продукт
```

```
count          8
unique         4
top            A
freq           3

Описателна статистика за всички колони в df_sales:
       продукт        цена  количество
count        8    8.000000    8.000000
unique       4         NaN         NaN
top          A         NaN         NaN
freq         3         NaN         NaN
mean       NaN   17.062500    4.125000
std        NaN    6.480823    2.953586
min        NaN   10.500000    1.000000
25%        NaN   10.500000    2.000000
50%        NaN   17.500000    3.500000
75%        NaN   21.250000    4.500000
max        NaN   25.000000   10.000000
```

Когато използваме `include=['object']`, получаваме статистика само за колоната `продукт`. Когато използваме `include='all'`, получаваме както числови, така и категорийни статистики. За числовите колони се показват числовите статистики, а за категорийните - категорийните.

6. Описателна статистика за времеви данни

Ако DataFrame-ът съдържа колони с времеви данни (datetime), `.describe()` може да предостави полезна информация за тях, особено ако параметърът `datetime_is_numeric` е настроен на `True`. По подразбиране, той ще покаже статистика като брой, уникални стойности, най-често срещана стойност и нейната честота.

- *Пример с времеви данни (за по-стари версии до 1.6.0):*

```
import pandas as pd
import numpy as np

dates = pd.to_datetime(['2023-01-01', '2023-01-01', '2023-01-02', '2023-
01-03', np.nan, '2023-01-03'])
```

```
df_time = pd.DataFrame({'дата': dates, 'стойност': [10, 20, 15, 25, 30,
22]})

print("\nОписателна статистика за времеви данни:")
print(df_time['дата'].describe())

print("\nОписателна статистика за времеви данни като числови:")
print(df_time['дата'].describe(datetime_is_numeric=True))
```

Резултат:

```
Описателна статистика за времеви данни:
count                      5
unique                     3
top        2023-01-01 00:00:00
freq                       2
first      2023-01-01 00:00:00
last       2023-01-03 00:00:00
Name: дата, dtype: object

Описателна статистика за времеви данни като числови:
count                      5
mean     1672531200000000000
std      178885438199984640
min      1672531200000000000
25%      1672531200000000000
50%      1672617600000000000
75%      1672704000000000000
max      1672704000000000000
Name: дата, dtype: int64
```

ЗАБЕЛЕЖКА:

Ако горният код не работи, преработен скрипт (за по-нови версии pandas >= 2.0.0):

```
import pandas as pd
import numpy as np

dates = pd.to_datetime(['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-03', np.nan, '2023-01-03'])
df_time = pd.DataFrame({'дата': dates, 'стойност': [10, 20, 15, 25, 30, 22]})

print("\nОписателна статистика за времеви данни:")
print(df_time['дата'].describe())

# Конвертиране на времевите данни в Unix timestamps (брой секунди от 1970-01-01)
timestamp_data = df_time['дата'].astype('int64') // 10**9

print("\nОписателна статистика за времеви данни като числови (Unix timestamps):")
print(timestamp_data.describe())
```

Разяснение на корекцията:

1. Първо, извеждаме стандартната описателна статистика за колоната с дати, която включва брой, уникални стойности, най-често срещана дата и нейния брой, както и първа и последна дата.
2. След това, за да получим числова статистика, конвертираме колоната `дата` в `int64`, което представлява броя на наносекундите от епохата (1970-01-01). За да получим по-лесно интерпретируеми числа (секунди), извършваме целочислено деление (`//`) на 109.
3. Накрая, прилагаме `.describe()` върху тази серия от числови данни (Unix timestamps).

Когато datetime_is_numeric=True, времевите данни се третират като числови (Unix timestamps), което позволява изчисляване на средна стойност, стандартно отклонение и квартили. Имайте предвид, не се поддържа в pandas 2.0.0 и по-нови версии.

Методът `.describe()` е мощен инструмент за бързо получаване на обобщена информация за вашите данни и е следващата ключова стъпка след използването на `.info()` при първоначалното изследване на DataFrame-а.

IV. Получаване на размерности (.shape, .size)

Тези два атрибута са основни и често използвани за бързо разбиране на структурата и обема на нашите Pandas DataFrame и Series обекти.

1. *`.shape` - Размерност на DataFrame или Series*

Атрибутът `.shape` връща кортеж (tuple), който представлява размерността на DataFrame-а или Series обекта.

- **За DataFrame:** Кортежът съдържа два елемента: `(брой_редове, брой_колони)`.
- **За Series:** Кортежът съдържа един елемент: `(брой_елементи,)`. Забележете запетаята след броя на елементите, която указва, че това е кортеж с един елемент.

- *Пример с DataFrame (`df_sales`):*

```
import pandas as pd

data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

print("Размерност на df_sales (брой редове, брой колони):",
df_sales.shape)
```

Резултат:

```
Размерност на df_sales (брой редове, брой колони): (8, 3)
```

Този резултат показва, че `df_sales` има 8 реда и 3 колони.

- ***Пример със Series (една колона от `df_sales`):***

```
prices = df_sales['цена']
print("Размерност на Series 'цена' (брой елементи):", prices.shape)
```

Резултат:

```
Размерност на Series 'цена' (брой елементи): (8,)
```

Резултатът показва, че Series `prices` има 8 елемента.

Използването на `.shape` е много полезно, когато искаме програмно да получим броя на редовете или колоните, например за итериране, предварително алокиране на памет или за логически проверки в нашия код.

2. `.size` - Общ брой на елементите

Атрибутът `.size` връща общия брой на елементите в DataFrame-а или Series обекта.

- **За DataFrame:** Това е произведението на броя на редовете и броя на колоните.
- **За Series:** Това е просто броят на елементите в Series-а.

- *Пример с DataFrame (`df_sales`):*

```
print("Общ брой на елементите в df_sales:", df_sales.size)
```

Резултат:

```
Общ брой на елементите в df_sales: 24
```

Тъй като `df_sales` има 8 реда и 3 колони, общият брой на елементите е 8×3=24.

- ***Пример със Series (`prices`):***

```
print("Общ брой на елементите в Series 'цена':", prices.size)
```

Резултат:

```
Общ брой на елементите в Series 'цена': 8
```

3. Предимства на `.shape` и `.size`:

- **Бърз достъп до размерността:** Предоставят бърз и лесен начин за получаване на основна информация за размера на данните.
- **Програмно използване:** Могат лесно да бъдат използвани в код за контролиране на логиката и оптимизиране на процесите.
- **Разбиране на обема на данните:** Помагат да се ориентираме колко голям е нашият набор от данни.

4. Допълнителни аспекти

1) **Използване при условни проверки:** `.shape` може да бъде много полезен при условни проверки в кода. Например, може да искате да изпълните определена операция само ако DataFrame-ът има повече от 100 реда:

```
if df.shape[0] > 100:
    # Изпълни някаква операция
    print("DataFrame-ът има повече от 100 реда.")
```

2) **Създаване на празни DataFrame или Series:** Когато създавате празен DataFrame или Series, `.shape` ще върне съответно `(0, брой_колони)` или `(0,)`, а `.size` ще върне `0`.

```
empty_df = pd.DataFrame(columns=['A', 'B'])
print("Размерност на празен DataFrame:", empty_df.shape)  # (0, 2)
print("Размер на празен DataFrame:", empty_df.size)     # 0

empty_series = pd.Series()
print("Размерност на празен Series:", empty_series.shape)  # (0,)
print("Размер на празен Series:", empty_series.size)     # 0
```

3) **Влияние на индекса:** Важно е да се отбележи, че `.shape` и `.size` отчитат само данните в редовете и колоните. Индексът (редовите етикети) не се включва в тези размери.

5. Любопитно:

1) **Разлика между `.len()` и `.shape[0]` за DataFrame:** Функцията вградена в Python `len(df)` ще върне броя на редовете в DataFrame-а, което е еквивалентно на `df.shape[0]`.

```
print("Брой редове (използвайки len()):", len(df_sales))
print("Брой редове (използвайки .shape[0]):", df_sales.shape[0])
```

И двата подхода дават еднакъв резултат за броя на редовете. Изборът между тях често е въпрос на предпочитание или контекст. `.shape[0]` е по-явен за получаване на първия елемент от кортежа за размерност.

2) **`.size` като брой на клетките:** За DataFrame, `.size` може да се разглежда като общия брой на "клетките" в таблицата (брой редове по брой колони).

6. Възможни изключения и неочаквано поведение:

1) **Няма изключения от тип `Error`:** Самите атрибути `.shape` и `.size` обикновено не предизвикват изключения от тип `Error` (като `AttributeError`). Те винаги ще върнат кортеж (за `.shape`) или цяло число (за `.size`), стига обектът, върху който се прилагат, да е валиден Pandas DataFrame или Series.
2) **Неочаквани размери при грешно създаване на обект:** Ако по някакъв начин сте създали обект, който не е стандартен Pandas DataFrame или Series, е възможно `.shape` и `.size` да не работят по очаквания начин или да предизвикат грешка. Въпреки това, при правилна употреба на Pandas, това е рядкост.
3) **Разлики при други структури от данни:** Важно е да се помни, че `.shape` и `.size` са специфични за NumPy arrays и Pandas DataFrames и Series. Други структури от данни в Python (например, списъци, речници) имат свои начини за определяне на размера (например, `len()` за списъци и речници).
4) **Series с многомерен индекс:** Дори при Series с многомерен (MultiIndex), `.shape` ще върне кортеж с броя на елементите по всяко ниво на индекса (въпреки че `.size` все още ще бъде общият брой на елементите).

```
multi_index_series = pd.Series([1, 2, 3, 4],
                               index=pd.MultiIndex.from_tuples([('A',
1), ('A', 2), ('B', 1), ('B', 2)]))
print("Размерност на Series с MultiIndex:", multi_index_series.shape)  #
(4,)
print("Размер на Series с MultiIndex:", multi_index_series.size)    # 4
```

В общи линии, .shape и .size са много надеждни и директни начини за получаване на информация за размерността на данните ви в Pandas. Те рядко водят до неочаквано поведение, стига да се прилагат върху правилните обекти.

Тези два атрибута са фундаментални за разбиране на основните измерения на вашите данни в Pandas. В следващите теми ще продължим да разглеждаме други начини за инспектиране на данните, като например получаване на типовете данни.

V. Получаване на типове данни (.dtype, .dtypes)

Разбирането на типовете данни във вашите DataFrame-и и Series е критично, тъй като то влияе върху начина, по който данните могат да бъдат обработвани, анализирани и съхранявани.

1. `.dtype` - Тип на данните на Series

Атрибутът `.dtype` се използва за получаване на типа на данните, съдържащи се в **Series** обект. Един Series може да съдържа само един тип данни.

- *Синтаксис:*

```
series.dtype
```

- ***Пример (използвайки Series `prices` от `df_sales`):***

```
import pandas as pd

data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

prices = df_sales['цена']
print("Тип на данните в Series 'цена':", prices.dtype)

products = df_sales['продукт']
```

```
print("Тип на данните в Series 'продукт':", products.dtype)

quantities = df_sales['количество']
print("Тип на данните в Series 'количество':", quantities.dtype)
```

Резултат:

```
Тип на данните в Series 'цена': float64
Тип на данните в Series 'продукт': object
Тип на данните в Series 'количество': int64
```

Резултатът показва, че колоната 'цена' съдържа числа с плаваща запетая (64-битови), колоната 'продукт' съдържа обекти (най-често стрингове или смесени типове), а колоната 'количество' съдържа цели числа (64-битови).

2. `.dtypes` - Типове данни на DataFrame

Атрибутът `.dtypes` се използва за получаване на типа на данните на всяка колона в **DataFrame** обект. Той връща Series, където индексът са имената на колоните, а стойностите са съответните им типове данни.

- *Синтаксис:*

```
dataframe.dtypes
```

- *Пример (използвайки `df_sales`):*

```
print("\nТипове данни на колоните в df_sales:")
print(df_sales.dtypes)
```

Резултат:

```
продукт      object
цена        float64
количество    int64
dtype: object
```

Резултатът е Series, който показва типа на данните за всяка колона в DataFrame-а.

3. Често срещани типове данни в Pandas:

- **`int64`:** Цели числа.
- **`float64`:** Числа с плаваща запетая.
- **`object`:** Най-общ тип, може да съдържа стрингове или комбинация от различни типове. Често се използва за стрингове.
- **`bool`:** Булеви стойности (True или False).
- **`datetime64[ns]`:** Времеви данни (дати и часове).
- **`timedelta64[ns]`:** Разлики във времето.
- **`category`:** Категорийни данни, които могат да бъдат по-ефективни за памет и производителност при повтарящи се стойности.

4. Важност на типовете данни:

- **Използване на паметта:** Различните типове данни заемат различно количество памет. Изборът на подходящ тип може да помогне за оптимизиране на използването на паметта, особено при големи набори от данни.
- **Производителност:** Някои операции могат да бъдат по-бързи или по-бавни в зависимост от типа на данните. Например, числовите операции са по-бързи върху `int64` и `float64` отколкото върху `object`.
- **Съвместимост с библиотеки:** Някои библиотеки за анализ и визуализация могат да имат специфични изисквания към типовете данни.
- **Семантично значение:** Типът на данните трябва да отразява естеството на информацията в колоната (например, не трябва да съхраняваме дати като стрингове, ако искаме да извършваме времеви анализи).

5. Допълнителни аспекти:

1) **Проверка на конкретен тип данни:** Можете да проверите дали типът на данните на дадена Series е определен тип, като използвате атрибута `.dtype` и го сравните:

```
print(prices.dtype == 'float64')
print(products.dtype == object)  # Може да се използва и самият обект на типа
```

3) **DataFrame с един тип данни:** Въпреки че обикновено DataFrame-ите съдържат колони с различни типове данни, е възможно да имате DataFrame, където всички колони са от един и същ тип. В този случай `.dtypes` ще върне Series с една и съща стойност за всички колони.
2) **Nullable Integer, Boolean и String Types (Pandas >= 1.0):** По-новите версии на Pandas въведоха "nullable" типове за цели числа (`Int64`), булеви стойности (`Boolean`) и стрингове (`String`). Те позволяват представянето на липсващи стойности (`pd.NA`) по начин, който е по-съгласуван и може да избегне някои от проблемите с използването на `NaN` за цели числа и булеви стойности, които ги преобразуват във `float64` и `object` съответно.

```
nullable_int_series = pd.Series([1, 2, pd.NA, 4], dtype='Int64')
print(nullable_int_series.dtype)  # Int64

nullable_bool_series = pd.Series([True, False, pd.NA], dtype='Boolean')
print(nullable_bool_series.dtype) # Boolean

nullable_string_series = pd.Series(['a', pd.NA, 'c'], dtype='String')
print(nullable_string_series.dtype) # String
```

6. Любопитно:

1) **`object` dtype:** Когато видите `object` като dtype на колона, това често означава, че колоната съдържа стрингове. Въпреки това, тя може също да съдържа смесени типове (например, числа и стрингове) или други Python обекти (например, списъци, речници). Ако производителността е важна и знаете, че колоната съдържа само стрингове, може да има смисъл да я преобразувате в `String` dtype (ако използвате по-нова версия на Pandas).
2) **Автоматично определяне на типа данни:** Pandas автоматично определя типа на данните при четене на файлове или създаване на Series/DataFrame-и. Понякога може да се наложи ръчно да коригирате тези типове с `.astype()`, ако Pandas не е избрал най-подходящия тип. Например, колона, съдържаща само цели числа, може да бъде прочетена като `float64`, ако в данните има липсващи стойности (преди въвеждането на nullable integer types).

7. Възможни изключения и неочаквано поведение:

1. **`AttributeError` при неприложим обект:** Ако се опитате да приложите `.dtype` върху DataFrame (вместо `.dtypes`) или `.dtypes` върху Series (вместо `.dtype`), ще получите `AttributeError`, тъй като тези атрибути са специфични за съответния тип обект.

```
try:
    print(df_sales.dtype)
except AttributeError as e:
    print(f"Грешка: {e}")  # DataFrame не разполага с атрибут .dtype

try:
    print(prices.dtypes)
except AttributeError as e:
    print(f"Грешка: {e}")  # Series не разполага с атрибут .dtypes
```

2) **Неочаквани типове данни след операции:** Някои операции могат да променят типа на данните на колоната. Например, извършване на аритметична операция с колона от цели числа и плаваща запетая ще доведе до колона от плаващи запетаи. Също така, добавянето на `NaN` към целочислена колона (преди nullable types) обикновено я преобразува във `float64`.
3) **Разлики при различни версии на Pandas:** Както беше отбелязано с nullable типовете данни, поведението и наличните dtypes могат да се различават между различните версии на Pandas. Винаги е добре да сте наясно с версията, която използвате.
4) **MemoryError при много големи данни:** Въпреки че `.dtype` и `.dtypes` сами по себе си не консумират много памет, разбирането на типовете данни е важно за оптимизиране на използването на паметта при много големи набори от данни. Неподходящите типове данни могат да доведат до значително по-голям разход на памет и евентуално до `MemoryError` при последващи обработки.

В заключение, `.dtype` и `.dtypes` са основни и надеждни инструменти за получаване на информация за типовете данни във вашите Pandas обекти. Разбирането и проверката на типовете данни е важна стъпка при инспектирането на данни и често е последвана от преобразуване на типовете данни (`.astype()`), ако е необходимо.

VI. Получаване на информация за индекса и колоните (.index, .columns)

1. `.index` - Информация за индекса

Атрибутът `.index` връща обект, който представлява индекса на DataFrame-а или Series-а. Индексът е последователност от етикети, които идентифицират всеки ред. Той може да бъде от различни типове: `RangeIndex` (по подразбиране за цели числа), `Pandas.Index` (общ обект за индекс), `DatetimeIndex` (за времеви серии), `MultiIndex` (за многостепенни индекси) и други.

- ***Синтаксис:***

```
dataframe.index
series.index
```

- ***Пример с DataFrame (`df_sales`):***

```
import pandas as pd
```

```
data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

print("Индекс на df_sales:", df_sales.index)
```

Резултат:

```
Индекс на df_sales: RangeIndex(start=0, stop=8, step=1)
```

Сега индексът е Pandas.Index от стрингове, които ние сме задали.

- ***Пример със Series:***

```
prices = df_sales['цена']
print("\nИндекс на Series 'цена':", prices.index)
```

Резултат:

```
Индекс на Series 'цена': RangeIndex(start=0, stop=8, step=1)
```

Подобно на DataFrame-а по подразбиране, Series също има RangeIndex.

2. `.columns` - *Информация за колоните*

Атрибутът `.columns` връща обект `Pandas.Index`, съдържащ имената на колоните в DataFrame-а. Series обект няма атрибут `.columns`, тъй като е едномерна структура.

- ***Синтаксис:***

```
dataframe.columns
```

- ***Пример с DataFrame (`df_sales`):***

```
print("\nКолони на df_sales:", df_sales.columns)
```

Резултат:

```
Колони на df_sales: Index(['продукт', 'цена', 'количество'],
dtype='object')
```

Резултатът е Pandas.Index обект, съдържащ имената на колоните като стрингове.

3. Важност на индекса и колоните:

- **Идентификация и достъп до данни:** Индексът и колоните осигуряват етикети за идентифициране и достъпване на конкретни редове и колони в DataFrame-а.
- **Подравняване на данни при операции:** Pandas използва индексите и имената на колоните за автоматично подравняване на данните при извършване на операции между различни Series или DataFrame-и.
- **Структура и организация:** Те определят основната структура на табличните данни.
- **Времеви серии:** За времеви серии, `DatetimeIndex` предоставя специализирани функционалности за работа с времеви данни.
- **Многостепенни данни:** `MultiIndex` позволява представянето на данни с по-сложна йерархична структура.

4. Допълнителни аспекти:

1) **Преобразуване към списък:** Индексите и колоните могат лесно да бъдат преобразувани в Python списъци, което може да бъде полезно за итериране или други операции, които изискват стандартен списъчен формат:

```
column_list = df_sales.columns.tolist()
index_list = df_sales_indexed.index.tolist()
print("Списък с колони:", column_list)
print("Списък с индекс:", index_list)
```

2) **Проверка за съществуване на колона/индекс:** Можете лесно да проверите дали дадено име на колона или индекс съществува:

```
if 'цена' in df_sales.columns:
    print("Колоната 'цена' съществува.")

if 'първи' in df_sales_indexed.index:
    print("Индекс 'първи' съществува.")
```

3) **Множество нива на индекс (MultiIndex):** DataFrame-ите могат да имат сложни, многостепенни индекси, които позволяват представянето на данни с по-висока размерност в двумерна таблица. `.index` ще върне обект `MultiIndex` в такива случаи.

```
data_multi = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8]}
index_multi = pd.MultiIndex.from_tuples([('група1', 'под1'), ('група1',
'под2'), ('група2', 'под1'), ('група2', 'под2')])
df_multi = pd.DataFrame(data_multi, index=index_multi)
print("\nИндекс на DataFrame с MultiIndex:", df_multi.index)
```

4) **Именуване на индекси и колони:** Индексите и колоните могат да имат свои собствени имена, което е особено полезно при работа с MultiIndex или когато искате да документирате произхода на индекса/колоните.

```
df_multi.index.names = ['група', 'подгрупа']
df_multi.columns.name = 'характеристика'
print("\nИндекс и колони с имена:")
print(df_multi)
```

5. Любопитно:

1. **Индексът не е просто списък от стрингове/числа:** Въпреки че може да изглежда като такъв, `.index` и `.columns` връщат специални `Index` обекти на Pandas, които осигуряват допълнителна функционалност като бързо търсене, възможност за дублирани етикети (макар и често нежелателно) и други оптимизации.
2. **Влияние върху производителността:** Добре проектираният индекс може значително да подобри производителността на операции като търсене (`.loc[]`, `.iloc[]`) и джойниране на DataFrame-и.

6. Възможни изключения и неочаквано поведение:

1) **`AttributeError` при Series за `.columns`:** Както споменахме, Series обект няма атрибут `.columns`. Опитът за достъп до него ще доведе до `AttributeError`.

```
try:
    print(prices.columns)
except AttributeError as e:
    print(f"Грешка: {e}")
```

2) **Неизменяемост:** Индексът и колоните са **неизменяеми** (immutable). Това означава, че не можете директно да променяте елементите им след като DataFrame-ът е създаден. За да промените етикетите на индекса или колоните, трябва да присвоите нова `Index` последователност на `.index` или `.columns` атрибута или да използвате методи като `.rename()`.

```
try:
    df_sales.columns[0] = 'нов_продукт'
except TypeError as e:
    print(f"Грешка: {e}")  # Index does not support mutable operations
```

За да промените имената на колоните, трябва да направите нещо като:

```
df_sales.columns = ['нов_продукт', 'нова_цена', 'ново_количество']
print("\nDataFrame с нови имена на колони:")
print(df_sales.head())
```

3) **Дублирани етикети:** Въпреки че Pandas позволява DataFrame-и и Series-и с дублирани етикети в индекса или колоните, това може да доведе до неочаквано поведение при достъпване на данни и като цяло се счита за лоша практика.
4) **Тип на данните на индекса/колоните:** Типът на данните в `.index` (ако не е `RangeIndex`) обикновено е `object`, но може да бъде и друг тип (например, `datetime64[ns]` за `DatetimeIndex`). Типът на данните в `.columns` винаги е `object` (стрингове).

Разглеждането на `.index` и `.columns` е важна стъпка за разбиране на структурата на DataFrame-а и как са организирани данните в него.

VII. Проверка за уникални стойности (.unique(), .nunique(), .value_counts())

След като вече сме запознати с индекса и колоните, следващата важна стъпка в инспекцията на данните е да разберем какви уникални стойности се срещат в отделните колони и колко често се появяват те. За тази цел Pandas предоставя три много полезни метода: `.unique()`, `.nunique()` и `.value_counts()`.

1. `.unique()` - Връщане на уникалните стойности

Методът `.unique()` се прилага върху Series обект и връща NumPy array, съдържащ всички уникални стойности в този Series. Редът на уникалните стойности в резултата не е гарантиран да бъде същият като редът им на появяване в оригиналния Series. `NaN` стойностите също се включват (само веднъж) в резултата.

- *Синтаксис:*

```
series.unique()
```

- ***Пример (използвайки колоната 'продукт' от `df_sales`):***

```
import pandas as pd
import numpy as np

data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А', np.nan],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5,
np.nan],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10, np.nan]}
df_sales_nan = pd.DataFrame(data)

unique_products = df_sales_nan['продукт'].unique()
print("Уникални продукти:", unique_products)
```

Резултат:

```
Уникални продукти: ['А' 'Б' 'В' 'Г' nan]
```

Резултатът е NumPy array, съдържащ всички уникални стойности от колоната 'продукт', включително `NaN`.

2. .nunique() - Връщане на броя на уникалните стойности

Методът `.nunique()` се прилага върху Series или DataFrame обект и връща броя на уникалните стойности. По подразбиране, той не брои `NaN` стойностите, но това поведение може да бъде променено с параметъра `dropna=False`.

- *Синтаксис:*

```
series.nunique(dropna=True)
```

```
dataframe.nunique(axis=0, dropna=True) # axis=0 (по колони) е по
подразбиране
```

- *Пример (използвайки колоната 'продукт' от `df_sales_nan`):*

```
num_unique_products = df_sales_nan['продукт'].nunique()
print("Брой уникални продукти (без NaN):", num_unique_products)

num_unique_products_with_nan =
df_sales_nan['продукт'].nunique(dropna=False)
print("Брой уникални продукти (включително NaN):",
num_unique_products_with_nan)
```

Резултат:

```
Брой уникални продукти (без NaN): 4
Брой уникални продукти (включително NaN): 5
```

- *Пример за DataFrame (брой уникални стойности във всяка колона):*

```
num_unique_all = df_sales_nan.nunique()
print("\nБрой уникални стойности във всяка колона:\n", num_unique_all)
```

Резултат:

```
Брой уникални стойности във всяка колона:
продукт       5
цена          5
количество    6
dtype: int64
```

3. `.value_counts()` - Връщане на броя на срещанията на всяка уникална стойност

Методът `.value_counts()` се прилага върху Series обект и връща нов Series, съдържащ уникалните стойности като индекс и техните честоти (брой на срещанията) като стойности. Резултатът е сортиран по честота в низходящ ред (най-често срещаната стойност е първа). По подразбиране, `NaN` стойностите се изключват, но това може да бъде променено с параметъра `dropna=False`.

- *Синтаксис:*

```
series.value_counts(normalize=False, sort=True, ascending=False,
dropna=True)
```

- *Пример (използвайки колоната 'продукт' от `df_sales_nan`):*

```
product_counts = df_sales_nan['продукт'].value_counts()
print("\nЧестота на продуктите (без NaN):\n", product_counts)

product_counts_with_nan =
df_sales_nan['продукт'].value_counts(dropna=False)
print("\nЧестота на продуктите (включително NaN):\n",
product_counts_with_nan)

# Нормализирана честота (пропорции вместо брой)
normalized_counts = df_sales_nan['продукт'].value_counts(normalize=True)
print("\nНормализирана честота на продуктите (без NaN):\n",
normalized_counts)
```

Резултат:

```
Честота на продуктите (без NaN):
А    3
Б    2
Г    2
В    1
Name: продукт, dtype: int64
```

```
Честота на продуктите (включително NaN):
А      3
Б      2
Г      2
NaN    1
В      1
Name: продукт, dtype: int64

Нормализирана честота на продуктите (без NaN):
А    0.375
Б    0.250
Г    0.250
В    0.125
Name: продукт, dtype: float64
```

4. Предимства на .unique(), .nunique() и .value_counts():

- **Разбиране на разпределението на данните:** Тези методи помагат да се разбере какви стойности присъстват в колоните и колко често се срещат.
- **Идентифициране на грешки и аномалии:** Чрез преглед на уникалните стойности или честотите може да се открият неочаквани или грешни записи.
- **Категориен анализ:** `.value_counts()` е особено полезен за анализ на категорийни данни.
- **Предварителна обработка:** Информацията за уникалните стойности може да бъде важна за вземане на решения при почистване и трансформиране на данни (например, групиране на редки категории).

5. Допълнителни аспекти:

1) **Използване върху DataFrame:**
   - `.unique()` не може да се приложи директно върху цял DataFrame. Трябва да го приложите върху конкретна колона (Series).
   - `.nunique()` може да се приложи върху цял DataFrame, като по подразбиране връща Series с броя на уникалните стойности във всяка колона (`axis=0`). Можете също да го приложите по редове (`axis=1`), за да видите броя на уникалните стойности във всеки ред.
   - `.value_counts()` също не може да се приложи директно върху цял DataFrame.
2) **Сортиране на резултатите от `.value_counts()`:** Резултатът от `.value_counts()` е сортиран по честота в низходящ ред по подразбиране (`sort=True`, `ascending=False`). Можете да промените това поведение, като зададете `sort=False` или `ascending=True`.

3) **Работа с `NaN` стойности:** Както вече видяхме, параметърът `dropna` контролира дали `NaN` стойностите се включват в резултатите на `.nunique()` и `.value_counts()`. За `.unique()`, `NaN` винаги се включва (само веднъж).
4) **Използване с категорийни данни (`category` dtype):** Когато се прилагат върху Series с категорийни данни, тези методи работят по същия начин. `.value_counts()` може да бъде особено ефективен при такива данни.
5) **Брой на най-често срещаните стойности:** Методът `.value_counts()` поддържа параметър `n` (в по-нови версии на Pandas), който позволява да се върнат само `n`-те най-често срещани стойности (`.nlargest(n)`) или `.nsmallest(n)` за най-рядко срещаните.

```
top_3_products = df_sales_nan['продукт'].value_counts().nlargest(3)
print("Топ 3 продукта:\n", top_3_products)
```

6. Любопитно:

1) **Приблизителен брой уникални стойности за големи данни:** За много големи набори от данни, пресмятането на точния брой уникални стойности може да бъде ресурсоемко. Някои библиотеки предлагат методи за *приблизително* броене на уникални стойности (например, HyperLogLog), които могат да бъдат по-ефективни в такива случаи, но те не са вградени в Pandas.
2) **Уникалност на комбинации от колони:** За да намерите уникалните комбинации от стойности в няколко колони, можете да създадете нова колона, съдържаща кортежи от тези стойности, и след това да приложите `.unique()` върху тази нова колона.

```
unique_combinations = df_sales_nan[['продукт', 'цена']].apply(tuple,
axis=1).unique()
print("Уникални комбинации продукт-цена:\n", unique_combinations)
```

7. Възможни изключения и неочаквано поведение:

1. **Приложимост само към Series (за `.unique()` и `.value_counts()`):** Опитът да се приложат `.unique()` или `.value_counts()` директно върху DataFrame ще доведе до `AttributeError`.
2. **Ред на уникалните стойности в `.unique()`:** Както беше споменато, редът на елементите в NumPy array-я, върнат от `.unique()`, не е гарантиран и може да не съответства на реда на първото им появяване в оригиналния Series. Ако редът е важен, може да се наложи допълнителна обработка.
3. **Тип на данните в резултата от `.value_counts()`:** Резултатът от `.value_counts()` е Series, където индексът са уникалните стойности (със същия dtype като оригиналния Series), а стойностите са честотите (dtype `int64` или `float64` ако `normalize=True`).
4. **MemoryError при голям брой уникални стойности:** Ако Series съдържа изключително голям брой уникални стойности, `.unique()` и `.value_counts()` могат да консумират значително количество памет, тъй като трябва да съхранят всички уникални елементи.
5. **Неочаквано поведение при смесени типове данни:** Ако Series съдържа колона със смесени типове данни (`object` dtype), резултатите от тези методи ще включват всички тези уникални обекти. Сравнението на обекти може да има свои нюанси.

Тези три метода са основни инструменти за първоначално изследване на съдържанието на отделни колони във вашите Pandas DataFrame-и.

VIII. Транспониране на DataFrame (.T)

Транспонирането на DataFrame е операция, при която се разменят редовете и колоните на DataFrame-а. Редовете стават колони, а колоните стават редове. В Pandas това се осъществява много лесно с помощта на атрибута `.T`.

1. `.T` - Транспониране

Атрибутът `.T` връща транспонирана версия на DataFrame-а. Оригиналният DataFrame остава непроменен.

- *Синтаксис:*

```
dataframe.T
```

- ***Пример (използвайки `df_sales`):***

```
import pandas as pd

data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
        'количество': [5, 2, 3, 1, 4, 6, 2, 10]}
df_sales = pd.DataFrame(data)

print("Оригинален DataFrame:\n", df_sales)

transposed_df = df_sales.T
print("\nТранспониран DataFrame:\n", transposed_df)
```

Резултат:

```
Оригинален DataFrame:
```

```
   продукт  цена  количество
0       А  10.5           5
1       Б  20.0           2
2       А  10.5           3
3       В  15.0           1
4       Г  25.0           4
5       Б  20.0           6
6       Г  25.0           2
7       А  10.5          10


Транспониран DataFrame:
             0     1     2     3     4     5     6     7
продукт      А     Б     А     В     Г     Б     Г     А
цена      10.5  20.0  10.5  15.0  25.0  20.0  25.0  10.5
количество    5     2     3     1     4     6     2    10
```

Както виждате, колоните 'продукт', 'цена' и 'количество' сега са станали редове, а оригиналните редови индекси (0 до 7) са станали колони.

2. Важни аспекти при транспониране:

1) **Типове данни:** Типовете данни на колоните в оригиналния DataFrame могат да станат типове данни на редовете в транспонирания DataFrame. Ако оригиналните колони са от различни типове, транспонираните редове ще имат най-общия възможен тип (често `object`), за да могат да съхранят всички стойности.

```
print("\nТипове данни на оригиналния DataFrame:\n", df_sales.dtypes)
print("\nТипове данни на транспонирания DataFrame:\n",
transposed_df.dtypes)
```

Резултат:

```
Типове данни на оригиналния DataFrame:
продукт         object
цена           float64
```

```
количество      int64
dtype: object


Типове данни на транспонирания DataFrame:
0    object
1    object
2    object
3    object
4    object
5    object
6    object
7    object
dtype: object
```

Забележете, че всички колони в транспонирания DataFrame са от тип object, тъй като оригиналната колона 'продукт' беше от този тип.

2) **Индекс и колони:** Индексът на оригиналния DataFrame става колони в транспонирания, а имената на колоните на оригиналния стават индекс на транспонирания.
3) **Series не се транспонира по същия начин:** Ако транспонирате Series, той остава Series (едномерна структура), но може да промени ориентацията си (от "колона" към "ред" или обратно, но тъй като е само едно измерение, това не е толкова очевидно).

```
prices = df_sales['цена']
print("\nОригинален Series:\n", prices)
print("\nТранспониран Series:\n", prices.T)
```

Резултат:

```
Оригинален Series:
0    10.5
1    20.0
2    10.5
3    15.0
4    25.0
5    20.0
6    25.0
```

```
7    10.5
Name: цена, dtype: float64

Транспониран Series:
0    10.5
1    20.0
2    10.5
3    15.0
4    25.0
5    20.0
6    25.0
7    10.5
Name: цена, dtype: float64
```

Както виждате, транспонирането на Series не променя визуално структурата му.

3. Кога е полезно транспонирането?

- **Промяна на ориентацията на данните:** Понякога е по-лесно да се работи с данните, когато редовете са колони и обратно, особено за определени видове анализ или визуализация.
- **Съобразяване с изисквания на библиотеки или функции:** Някои библиотеки или функции може да очакват данните в определена ориентация.
- **Визуално представяне:** В някои случаи, транспонираният DataFrame може да бъде по-лесен за четене или представяне.

4. Любопитно:

1) **Вътрешно представяне на данните:** Вътрешно, Pandas съхранява данните в NumPy arrays, които са оптимизирани за операции с матрици. Транспонирането е сравнително бърза операция, тъй като не включва преместване на самите данни в паметта, а по-скоро промяна на "изгледа" върху тези данни.
2) **Използване при визуализация:** Понякога е по-удобно да се визуализират определени аспекти на данните, когато DataFrame-ът е транспониран. Например, може да е по-лесно да се сравняват стойностите на различни характеристики за един обект (който е бил ред в оригиналния DataFrame), когато тези характеристики са в един ред след транспонирането.

5. *Възможни изключения и неочаквано поведение:*

1) **`AttributeError` при Series:** Както вече отбелязахме, Series обект също има атрибут `.T`, но тъй като е едномерна структура, транспонирането му не променя формата му (връща същия Series). Няма грешка, но може да е неочаквано, ако очаквате промяна в размерността.
2) **Типове данни след транспониране (отново):** Бъдете внимателни за промените в типовете данни след транспониране, особено ако оригиналният DataFrame съдържа колони с различни типове. Резултатът може да бъде DataFrame с много колони от тип `object`, което може да повлияе на производителността на последващи анализи.
3) **Работа с MultiIndex след транспониране:** Ако DataFrame-ът има MultiIndex (както за редовете, така и за колоните), транспонирането ще размени нивата на тези MultiIndex-и. Това може да бъде полезно за преструктуриране на сложни данни, но изисква внимателно разбиране на структурата на MultiIndex-а преди и след транспонирането.

```
data_multi = {'A': [1, 2], 'B': [3, 4]}
index_multi = pd.MultiIndex.from_tuples([('група1', 'под1'), ('група2',
'под2')])
columns_multi = pd.MultiIndex.from_tuples([('характер1', 'стойност1'),
('характер2', 'стойност2')])
df_multi_complex = pd.DataFrame(data_multi, index=index_multi,
columns=columns_multi)
print("\nОригинален DataFrame с MultiIndex:\n", df_multi_complex)
print("\nТранспониран DataFrame с MultiIndex:\n", df_multi_complex.T)
print("\nИндекс на транспонирания:", df_multi_complex.T.index)
print("\nКолони на транспонирания:", df_multi_complex.T.columns)
```

4) **Размер на DataFrame:** Транспонирането на много голям DataFrame може да създаде нов DataFrame със същия размер, но с разменени измерения. Уверете се, че имате достатъчно памет за новия обект, ако работите с големи набори от данни.

В общи линии, .T е прост, но мощен инструмент за промяна на ориентацията на вашите данни в Pandas.

IX. Обобщение на главата и казуси от реалния живот

В тази глава разгледахме основните методи и атрибути на Pandas, които ни помагат да получим първоначална представа за нашите данни, да разберем тяхната структура и съдържание. Ето кратко обобщение на всеки метод и как той се прилага в практически сценарии:

1. .head() и .tail():

- **Цел:** Бърз преглед на първите или последните няколко реда от DataFrame-а.
- **Казус:**
  - **Анализ на продажби:** След като заредите данни за продажби, използвате `.head()` да видите първите няколко транзакции и да се уверите, че данните са заредени правилно и съдържат очакваните колони (например, дата, продукт, цена, количество).

Решение:

Използвайте метода `.head()` за да видите първите няколко реда от DataFrame-а.

```
import pandas as pd

# Симулираме зареждане на данни за продажби от CSV файл
data_sales = {'дата': ['2023-10-26', '2023-10-26', '2023-10-27', '2023-10-27', '2023-10-28'],
              'продукт': ['лаптоп', 'мишка', 'клавиатура', 'монитор', 'лаптоп'],
              'цена': [1200.50, 25.99, 75.00, 300.00, 1250.00],
              'количество': [1, 2, 1, 1, 1]}
df_sales = pd.DataFrame(data_sales)

print("Първите 5 реда от данните за продажби:\n", df_sales.head())

# Можете да укажете броя на редовете, които да се покажат (например, първите 3)
print("\nПървите 3 реда от данните за продажби:\n", df_sales.head(3))
```

```
# След като изпълните .head(), вие визуално проверявате:
# 1. Дали колоните 'дата', 'продукт', 'цена', 'количество' съществуват.
# 2. Дали данните под тези колони изглеждат смислени (например, цените
са числови стойности, датите са във валиден формат).
```

Обяснение:

Методът `.head()` по подразбиране показва първите 5 реда на DataFrame-а. Чрез подаване на аргумент (например, `3`), можете да укажете колко реда да се покажат. Този бърз преглед помага да се уверите, че процесът на зареждане на данни е успешен и че данните изглеждат както се очаква.

- **Логиране на събития:** При анализ на лог файлове, `.tail()` може да покаже последните събития, което е полезно за наблюдение на текуща активност или за откриване на скорошни грешки.

Решение:

- Използвайте метода `.tail()` за да видите последните няколко реда от DataFrame-а.

```
import pandas as pd

# Симулираме зареждане на лог данни
data_log = {'време': ['2023-10-28 10:00:00', '2023-10-28 10:01:15',
'2023-10-28 10:02:30',
                    '2023-10-28 10:03:45', '2023-10-28 10:05:00', '2023-
10-28 10:06:15'],
            'ниво': ['INFO', 'DEBUG', 'WARNING', 'ERROR', 'INFO',
'DEBUG'],
            'съобщение': ['Потребител X влезе в системата', 'Проверка на
база данни завършена',
                          'Възможен проблем с връзката', 'Неуспешен опит
за запис',
                          'Изпращане на отчет завършено', 'Изчисляване на
статистики']}
df_log = pd.DataFrame(data_log)

print("Последните 5 реда от лог данните:\n", df_log.tail())
```

```
# Можете да укажете броя на редовете от края, които да се покажат
(например, последните 2)
print("\nПоследните 2 реда от лог данните:\n", df_log.tail(2))

# Чрез преглеждане на последните записи, можете бързо да видите:
# 1. Последните действия или събития в системата.
# 2. Дали има скорошни грешки или предупреждения, които изискват
внимание.
```

Обяснение:

Методът `.tail()` по подразбиране показва последните 5 реда на DataFrame-а. Подобно на `.head()`, можете да укажете броя на редовете, които да се покажат от края на DataFrame-а. Това е особено полезно при анализ на времево-зависими данни като лог файлове, където най-скорошната информация често е най-важна.

Тези прости примери илюстрират как `.head()` и `.tail()` могат да бъдат бързи и ефективни инструменти за първоначална инспекция на данни в реални сценарии.

2. *.info():*

- **Цел:** Получаване на обобщена информация за DataFrame-а, включително брой редове, брой колони, типове данни на колоните и информация за липсващи стойности.
- **Казус:**
  - **Подготовка на данни за машино обучение:** Преди да тренирате модел, използвате `.info()` да проверите за липсващи стойности, които трябва да бъдат обработени, и да се уверите, че типовете данни са подходящи за модела (например, категорийните променливи трябва да бъдат кодирани).

Решение:

Използвайте метода `.info()` за да получите тази обобщена информация.

```
import pandas as pd
import numpy as np

# Симулираме DataFrame с данни за машинно обучение
data_ml = {'feature1': [1.0, 2.5, 3.0, np.nan, 5.1],
```

```
            'feature2': ['A', 'B', 'A', 'C', 'B'],
            'feature3': [10, 20, np.nan, 40, 50],
            'target': [0, 1, 0, 1, 0]}
df_ml = pd.DataFrame(data_ml)

print("Информация за DataFrame-а за машинно обучение:\n")
df_ml.info()

# От резултата на .info() можете да видите:
# 1. Общия брой на редовете и колоните.
# 2. Типа на данните за всяка колона (float64, object, int64).
# 3. Броя на non-null стойностите във всяка колона, което индикира наличието на липсващи стойности (NaN).
#    Например, 'feature1' и 'feature3' имат по 4 non-null стойности от общо 5, което означава, че има по една липсваща стойност във всяка от тях.
# 4. Използваната памет от DataFrame-а.

# Въз основа на тази информация, може да се наложи да предприемете стъпки като:
# - Обработка на липсващите стойности (например, чрез попълване или премахване).
# - Кодиране на категорийните променливи ('feature2') в числов формат, подходящ за повечето ML модели.
# - Проверка дали типовете данни са подходящи за очакваните операции.
```

Обяснение:

Методът `.info()` предоставя ключова информация, необходима за предварителна обработка на данни за машинно обучение. Идентифицирането на липсващи стойности и неправилни типове данни е критична стъпка преди обучението на модела, за да се гарантира неговата ефективност и да се избегнат грешки по време на процеса.

- **Интеграция на данни от различни източници:** Когато комбинирате данни от няколко файла или бази данни, `.info()` помага да се идентифицират несъответствия в типовете данни или липсващи колони.

Решение:

Използвайте `.info()` след обединяването на данните, за да проверите структурата на получения DataFrame.

```
import pandas as pd
import numpy as np

# Симулираме зареждане на данни от два различни файла
data_clients1 = {'clientID': [1, 2, 3, 4],
                 'age': [25, 30, 45, 35],
                 'city': ['Sofia', 'Plovdiv', 'Sofia', 'Varna']}
df_clients1 = pd.DataFrame(data_clients1)

data_clients2 = {'clientID': [3, 4, 5, 6],
                 'income': ['50000', '60000', 75000, 40000], #
Забележете, че една стойност е стринг, а друга - число
                 'occupation': ['engineer', 'manager', 'sales',
'developer']}
df_clients2 = pd.DataFrame(data_clients2)

# Обединяваме DataFrame-ите по 'clientID'
df_clients_merged = pd.merge(df_clients1, df_clients2, on='clientID',
how='outer')

print("Информация за обединения DataFrame с данни за клиенти:\n")
df_clients_merged.info()

# От резултата на .info() може да забележите:
```

```
# 1. Колоната 'income' може да има тип 'object', поради смесването на стринг и числова стойност във входните данни. Това може да доведе до проблеми при последващ числен анализ.
# 2. Възможно е да има липсващи стойности (NaN) в колони, които не са присъствали в един от оригиналните DataFrame-и за определени 'clientID'-та (в зависимост от типа на merge-а).

# В този случай, след като видим информацията от .info(), ще трябва да:
# - Преобразуваме колоната 'income' в числов тип (например, float), като предварително обработим стринговите стойности.
# - Обработим липсващите стойности в други колони, ако има такива.
```

Обяснение:

При интегриране на данни от различни източници, често се срещат несъответствия в типовете данни или липсващи стойности. Използването на `.info()` след обединяването на данните помага бързо да се идентифицират тези проблеми, които могат да възникнат поради различни формати на данните в източниците или поради начина на обединяване (например, `outer join` въвежда `NaN` стойности).

Тези два казуса показват как `.info()` е ценен инструмент за получаване на обща представа за вашите данни и за идентифициране на потенциални проблеми, които трябва да бъдат решени преди по-нататъшен анализ.

3. .describe():

- **Цел:** Генериране на описателна статистика за числовите колони (средна стойност, стандартно отклонение, минимум, максимум, квартили) и за категорийните и времеви данни (брой, уникални стойности, най-често срещана стойност, честота).
- **Казус:**
  - **Анализ на клиентско поведение:** Използвате `.describe()` върху колони като възраст, приходи или време, прекарано на уебсайт, за да получите обща представа за разпределението на тези променливи и да идентифицирате потенциални аномалии.

Решение:

Използвайте метода `.describe()` върху DataFrame-а, за да получите описателна статистика за числовите колони.

```
import pandas as pd
import numpy as np

# Симулираме DataFrame с данни за поведението на клиенти
data_behavior = {'clientID': [1, 2, 3, 4, 5, 6, 7, 8],
                 'age': [25, 30, 45, 35, 28, 60, 22, 40],
                 'total_spent': [150.75, 300.20, 550.90, 200.00, 180.50,
700.10, 120.30, np.nan],
                 'time_on_site': [15.5, 22.1, 30.0, 18.7, 25.3, 45.2,
10.1, 20.5]}
df_behavior = pd.DataFrame(data_behavior)

print("Описателна статистика за числовите колони:\n",
df_behavior.describe())

# От резултата на .describe() можете да видите:
# - count: Броя на валидните (не-NaN) стойности за всяка колона. За
'total_spent' е по-малък от общия брой редове, което показва липсваща
стойност.
# - mean: Средната стойност.
# - std: Стандартното отклонение, което показва разсейването на данните
около средната стойност.
# - min: Минималната стойност.
# - 25%: Първият квартил (25% от данните са под тази стойност).
# - 50%: Медианата (средната стойност, 50% от данните са под тази
стойност).
# - 75%: Третият квартил (75% от данните са под тази стойност).
# - max: Максималната стойност.

# Анализирайки тези статистики, можете да получите представа за:
# - Типичната възраст на клиентите (средна и медиана).
```

```
# - Разпределението на сумите, които клиентите харчат (средна, медиана, размах между квартилите).
# - Средното време, прекарано на сайта.
# - Потенциални аномалии (например, необичайно висока или ниска възраст или сума на поръчка).
```

- **Контрол на качеството в производството:** При анализ на данни от сензори, `.describe()` може да помогне за откриване на необичайни вариации в измерванията (например, температура, налягане), които могат да сигнализират за проблеми в процеса.

Решение:

Използвайте `.describe()` върху DataFrame-а, съдържащ данните от сензорите.

```
import pandas as pd
import numpy as np

# Симулираме DataFrame с данни от сензори
data_production = {'артикул_ID': [1, 2, 3, 4, 5, 6],
                   'температура': [25.1, 25.3, 24.9, 25.0, 25.2, 30.0],
# Една стойност може да е аномална
                   'налягане': [10.2, 10.1, 10.3, 10.2, 10.1, 10.2],
                   'тегло': [100.5, 100.3, 100.4, np.nan, 100.2, 100.6]}
df_production = pd.DataFrame(data_production)

print("Описателна статистика за производствените параметри:\n", df_production[['температура', 'налягане', 'тегло']].describe())

# От резултата на .describe() за числовите колони ('температура', 'налягане', 'тегло'):
# - Проверете средната стойност и стандартното отклонение за всяка характеристика. Голямото стандартно отклонение може да индикира по-голяма вариабилност.
```

```
# - Сравнете минималните и максималните стойности с очакваните граници. Необичайно ниски или високи стойности могат да бъдат индикатор за проблем.
# - Разгледайте квартилите, за да видите разпределението на данните. Голяма разлика между квартилите може да показва изкривяване на разпределението.
# - За 'тегло', count е по-малък от общия брой редове, което показва липсваща стойност, която трябва да бъде разгледана.
# - В колоната 'температура', максималната стойност (30.0) е значително по-висока от останалите и може да е аномалия, която да изисква проверка на производствения процес.
```

Обяснение:

`.describe()` е мощен инструмент за бързо получаване на статистически обобщения на числови данни. В контекста на контрол на качеството, тези статистики могат да помогнат за идентифициране на необичайни измервания или голяма вариабилност, които могат да сигнализират за проблеми в производствения процес и да насочат към по-нататъшно разследване.

4. .shape и .size:

- **Цел:** Получаване на размерността (брой редове и колони) и общия брой на елементите в DataFrame-а.
- **Казус:**
  - **Мащабируемост на алгоритми:** Преди да приложите алгоритъм за анализ, проверявате `.shape`, за да прецените дали размерът на данните е управляем за наличните ресурси.

Решение:

Използвайте `.shape` и `.size()` на примерни или очаквани по размер DataFrame-и, за да получите информация за техните размери.

```
import pandas as pd
import numpy as np

# Симулираме голям DataFrame с данни
num_rows = 1000000
num_cols = 10
```

```
data_large = np.random.rand(num_rows, num_cols)
columns_large = [f'колона_{i}' for i in range(num_cols)]
df_large = pd.DataFrame(data_large, columns=columns_large)

print("Размерност на големия DataFrame (брой редове, брой колони):", df_large.shape)
print("Общ брой на елементите в големия DataFrame:", df_large.size)

# В този случай, df_large.shape ще върне (1000000, 10), което означава 1 милион реда и 10 колони.
# df_large.size ще върне 10000000 (1 милион * 10).

# Въз основа на тези стойности, можете да прецените:
# - Времето, необходимо за обработка на данните от вашия алгоритъм (алгоритми с по-висока сложност могат да отнемат значително повече време при голям брой редове).
# - Паметта, която ще е необходима за съхранение и обработка на DataFrame-а.
# - Необходимостта от оптимизация на алгоритъма или използване на по-ефективни структури от данни за големи набори от данни.

# Можете също да използвате .shape след прилагане на филтри или други операции за намаляване на размера на данните:
df_filtered = df_large[df_large['колона_0'] > 0.5]
print("\nРазмерност на филтрирания DataFrame:", df_filtered.shape)
print("Общ брой на елементите във филтрирания DataFrame:", df_filtered.size)
```

Обяснение:

`.shape` ви дава директна информация за броя на редовете и колоните, което е ключово за разбиране на "мащаба" на вашите данни. `.size` предоставя общия брой на елементите, което също може да бъде полезно за оценка на паметта и потенциалната продължителност на изчисленията. При разработване на алгоритми, особено за машинно обучение или сложен анализ, е важно да се вземе предвид как размерът на входните данни ще повлияе на производителността.

- **Валидация на данни след трансформация:** След филтриране или агрегиране на данни, използвате `.shape`, за да се уверите, че броят на редовете или колоните е такъв, какъвто се очаква.

Решение:

Използвайте `.shape` след всяка ключова стъпка на трансформация, за да валидирате размера на DataFrame-а.

```
import pandas as pd

# Симулираме DataFrame с данни за продажби
data_sales = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б', 'Г', 'А'],
              'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0, 25.0, 10.5],
              'количество': [5, 2, 3, 1, 4, 6, 2, 10],
              'регион': ['Север', 'Юг', 'Север', 'Изток', 'Запад', 'Юг',
'Запад', 'Север']}
df_sales = pd.DataFrame(data_sales)

print("Размерност на оригиналния DataFrame:", df_sales.shape)

# Филтрираме продажбите само за регион 'Север'
df_north = df_sales[df_sales['регион'] == 'Север'].copy()
print("\nРазмерност след филтриране за 'Север':", df_north.shape)

# Добавяме нова колона с обща стойност на продажбата
df_north['обща_стойност'] = df_north['цена'] * df_north['количество']
print("\nРазмерност след добавяне на колона:", df_north.shape)

# Групираме по продукт и изчисляваме средна цена
df_grouped = df_north.groupby('продукт')['цена'].mean().reset_index()
print("\nРазмерност след групиране:", df_grouped.shape)

# Чрез проверка на .shape след всяка операция, можете да се уверите, че:
# - Филтрирането е премахнало очаквания брой редове.
# - Добавянето на колона не е променило броя на редовете, а е увеличило
броя на колоните с 1.
```

```
# - Групирането е намалило броя на редовете до броя на уникалните стойности в колоната за групиране ('продукт').

# Ако .shape след някоя стъпка не е това, което очаквате, това може да сигнализира за грешка в логиката на трансформацията.
```

Обяснение:

Използването на `.shape` като част от процеса на анализ и трансформация на данни помага за валидиране на резултатите от всяка стъпка. Уверяването, че броят на редовете и колоните се променя по очаквания начин след всяка операция, е важна практика за поддържане на коректността на анализа.

5. .dtype и .dtypes:

- **Цел:** Получаване на типа на данните в Series (`.dtype`) или за всяка колона в DataFrame-а (`.dtypes`).
- **Казус:**
  - **Преобразуване на данни:** Идентифицирате колони с неправилни типове данни (например, числа, съхранявани като стрингове) с `.dtypes` и след това ги конвертирате в подходящия тип с `.astype()`.

Решение:

Използвайте `.dtypes` за да идентифицирате колоните с неправилен тип данни и след това използвайте `.astype()` за да ги преобразувате в подходящия числен тип.

```
import pandas as pd

# Симулираме DataFrame, в който числови данни са прочетени като стрингове
data_strings = {'ID': [1, 2, 3, 4],
                'цена': ['10.50', '20.00', '15.75', '22.99'],
                'количество': ['5', '10', '3', '7']}
df_strings = pd.DataFrame(data_strings)

print("Типове данни преди преобразуване:\n", df_strings.dtypes)

# Забелязваме, че 'цена' и 'количество' са от тип 'object' (стрингове)

# Преобразуваме колона 'цена' в тип float
```

```
df_strings['цена'] = df_strings['цена'].astype(float)

# Преобразуваме колона 'количество' в тип integer
df_strings['количество'] = df_strings['количество'].astype(int)

print("\nТипове данни след преобразуване:\n", df_strings.dtypes)

# Сега колоните 'цена' и 'количество' са с правилните числови типове
(float64 и int64 съответно)
# и можем да извършваме математически операции с тях:
df_strings['обща_стойност'] = df_strings['цена'] *
df_strings['количество']
print("\nDataFrame с добавена колона 'обща_стойност':\n", df_strings)
```

Обяснение:

Често при четене на данни от външни източници (като CSV файлове), Pandas може да не успее автоматично да определи правилния тип данни за всяка колона. Колони, съдържащи числа, понякога могат да бъдат интерпретирани като стрингове, особено ако има нечисти данни (например, нечислови символи) или ако всички стойности са заобиколени в кавички. Използването на `.dtypes` е първата стъпка за идентифициране на такива проблеми, а `.astype()` е методът за коригиране на типовете данни, за да се позволи правилното обработване на данните.

- **Съвместимост с други библиотеки:** Проверявате `.dtypes`, за да се уверите, че типовете данни са съвместими с изискванията на библиотеки за визуализация (като Matplotlib или Seaborn) или за статистически анализ (като SciPy).

Решение:

Използвайте `.dtypes` за да се уверите, че типовете данни във вашия DataFrame са съвместими с изискванията на целевата библиотека. Ако е необходимо, използвайте `.astype()` за да ги преобразувате.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Симулираме DataFrame с данни за продажби по месеци
```

```
data_time_series = {'месец': ['Януари', 'Февруари', 'Март', 'Април',
'Май'],
                    'продажби': [150, 180, 220, 190, 250]}
df_time = pd.DataFrame(data_time_series)

print("Типове данни преди визуализация:\n", df_time.dtypes)

# Ако искаме да създадем линия на графика на продажбите по месеци,
# колоната 'продажби' трябва да е числов тип (int64 в този случай), а
'месец' може да е object (string).

# Да предположим, че 'продажби' по някаква причина е прочетен като
стринг:
df_time['продажби'] = df_time['продажби'].astype(str)
print("\nТипове данни след (грешно) преобразуване в стринг:\n",
df_time.dtypes)

# Опит за създаване на графика (ще доведе до грешка или неочаквана
графика)
# plt.plot(df_time['месец'], df_time['продажби'])
# plt.show()

# Коригираме типа на 'продажби' обратно към числов
df_time['продажби'] = df_time['продажби'].astype(int)
print("\nТипове данни след корекция:\n", df_time.dtypes)

# Сега можем да създадем коректна графика
plt.plot(df_time['месец'], df_time['продажби'])
plt.xlabel("Месец")
plt.ylabel("Продажби")
plt.title("Продажби по месеци")
plt.show()
```

```
# Подобно, някои статистически тестове в SciPy може да изискват
определени типове данни (например, числови за корелация).
```

6. .index и .columns:

- **Цел:** Получаване на информация за индекса (етикетите на редовете) и колоните (имената на колоните).
- **Казус:**
  - **Времеви серии:** При анализ на времеви данни, проверявате дали индексът е `DatetimeIndex`, което предоставя специализирани методи за работа с времеви редове.

Решение:

Използвайте `.index` за да проверите типа на индекса и, ако е необходимо, го преобразувайте в `DatetimeIndex` с помощта на `pd.to_datetime()`.

```
import pandas as pd

# Симулираме DataFrame с данни за продажби с колона за дата
data_time_series = {'дата': ['2023-01-01', '2023-01-08', '2023-01-15',
'2023-01-22', '2023-01-29'],
                    'продажби': [150, 180, 220, 190, 250]}
df_sales_date_col = pd.DataFrame(data_time_series)

print("Тип на индекса преди преобразуване:",
type(df_sales_date_col.index))
print("Индекс преди преобразуване:\n", df_sales_date_col.index)

# Задаваме колоната 'дата' като индекс
df_sales_indexed = df_sales_date_col.set_index('дата')
print("\nТип на индекса след задаване на колона 'дата' като индекс:",
type(df_sales_indexed.index))
print("Индекс след задаване на колона 'дата' като индекс:\n",
df_sales_indexed.index)

# Забелязваме, че индексът е от тип 'object' (стрингове), а не
DatetimeIndex
```

```
# Преобразуваме индекса в DatetimeIndex
df_sales_indexed.index = pd.to_datetime(df_sales_indexed.index)
print("\nТип на индекса след преобразуване в DatetimeIndex:",
type(df_sales_indexed.index))
print("Индекс след преобразуване в DatetimeIndex:\n",
df_sales_indexed.index)

# Сега можем да използваме специфични методи за времеви серии, например:
monthly_mean = df_sales_indexed['продажби'].resample('ME').mean()
print("\nСредни месечни продажби:\n", monthly_mean)
```

Обяснение:

Когато работите с времеви данни, е изключително важно индексът на DataFrame-а да бъде от тип `DatetimeIndex`. Това позволява използването на мощни вградени функции за анализ на времеви серии, като пресемплиране (`resample`), плъзгащи средни (`rolling`), и други. Чрез `.index` можете да проверите типа на текущия индекс и да определите дали е необходимо преобразуване с `pd.to_datetime()`.

- **Джойнване на данни:** Уверявате се, че имената на колоните, които ще използвате за свързване на два DataFrame-а, са правилни и съществуват.

Решение:

Използвайте `.columns` за да получите списък с имената на колоните във всеки DataFrame и да ги сравните.

```
import pandas as pd

# Симулираме два DataFrame-а с данни за клиенти
data_clients1 = {'CustomerID': [1, 2, 3, 4],
                 'Name': ['Alice', 'Bob', 'Charlie', 'David'],
                 'Age': [25, 30, 45, 35]}
df_clients1 = pd.DataFrame(data_clients1)
```

```
data_orders = {'ClientID': [2, 4, 1, 3], # Забележете различното име на
колоната за ID
               'Product': ['Laptop', 'Keyboard', 'Mouse', 'Monitor'],
               'OrderDate': ['2023-10-26', '2023-10-27', '2023-10-26',
'2023-10-28']}
df_orders = pd.DataFrame(data_orders)

print("Колони на df_clients1:", df_clients1.columns)
print("Колони на df_orders:", df_orders.columns)

# За да обединим тези DataFrame-и по клиентски ID, трябва имената на
колоните да съвпадат.
# В df_clients1 колоната е 'CustomerID', а в df_orders е 'ClientID'.

# Преименуваме колоната в df_orders, за да съвпадне с df_clients1
df_orders = df_orders.rename(columns={'ClientID': 'CustomerID'})
print("\nКолони на df_orders след преименуване:", df_orders.columns)

# Сега можем да обединим DataFrame-ите
df_merged = pd.merge(df_clients1, df_orders, on='CustomerID',
how='inner')
print("\nОбединен DataFrame:\n", df_merged)
```

Обяснение:

При обединяване на данни от различни източници, е критично да се уверяте, че колоните, по които ще се извършва обединяването, имат еднакви имена в участващите DataFrame-и. Използването на `.columns` ви позволява да инспектирате имената на колоните и да идентифицирате несъответствия. След това можете да използвате метода `.rename()` за да приведете имената в съответствие преди да извършите операцията за обединяване.

7. .unique(), .nunique(), .value_counts():

- **Цел:** Проверка за уникални стойности и тяхната честота в Series.
- **Казус:**
  - **Анализ на категорийни променливи:** Използвате `.value_counts()` за да видите разпределението на категориите (например, различни видове продукти, географски региони) и да идентифицирате най-често срещаните.

Решение:

Използвайте `.unique()` за да видите всички уникални стойности в дадена категорийна колона, `.nunique()` за да получите броя на уникалните стойности, и `.value_counts()` за да видите честотата на всяка уникална стойност.

```
import pandas as pd

# Симулираме DataFrame с данни за клиенти
data_customers = {'CustomerID': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                  'Пол': ['м', 'ж', 'м', 'ж', 'м', 'м', 'ж', 'м', 'ж',
'м'],
                  'Град': ['София', 'Пловдив', 'София', 'Варна',
'София', 'Бургас', 'Пловдив', 'София', 'Варна', 'София'],
                  'Плащане': ['карта', 'наложен', 'карта', 'карта',
'банков', 'наложен', 'карта', 'карта', 'банков', 'карта']}
df_customers = pd.DataFrame(data_customers)

# Анализ на колона 'Пол'
unique_genders = df_customers['Пол'].unique()
num_unique_genders = df_customers['Пол'].nunique()
gender_counts = df_customers['Пол'].value_counts()

print("Уникални стойности в колона 'Пол':", unique_genders)
print("Брой уникални стойности в колона 'Пол':", num_unique_genders)
print("Честота на стойностите в колона 'Пол':\n", gender_counts)

print("\n---")

# Анализ на колона 'Град'
unique_cities = df_customers['Град'].unique()
num_unique_cities = df_customers['Град'].nunique()
city_counts = df_customers['Град'].value_counts()

print("Уникални стойности в колона 'Град':", unique_cities)
print("Брой уникални стойности в колона 'Град':", num_unique_cities)
```

```
print("Честота на стойностите в колона 'Град':\n", city_counts)

print("\n---")

# Анализ на колона 'Плащане'
unique_payments = df_customers['Плащане'].unique()
num_unique_payments = df_customers['Плащане'].nunique()
payment_counts = df_customers['Плащане'].value_counts()

print("Уникални стойности в колона 'Плащане':", unique_payments)
print("Брой уникални стойности в колона 'Плащане':",
num_unique_payments)
print("Честота на стойностите в колона 'Плащане':\n", payment_counts)

# От тези резултати можем да разберем:
# - Кои са всички възможни стойности за пол, град и метод на плащане.
# - Колко различни града има в набора от данни.
# - Кой е най-често срещаният метод на плащане и как е разпределен полът
на клиентите.
```

- **Откриване на грешни или неконсистентни данни:** `.unique()` може да помогне за бързо идентифициране на неочаквани стойности в колона (например, различни начини за изписване на една и съща категория).

Решение:

Използвайте `.unique()` върху колоната с категории, за да видите всички уникални стойности и да идентифицирате потенциални грешки (например, различни начини за изписване на една и съща категория).

```
import pandas as pd

# Симулираме DataFrame с данни за продукти с потенциално неконсистентни
категории
data_products = {'ProductID': [1, 2, 3, 4, 5, 6, 7, 8],
```

```
                    'Name': ['Laptop', 'Mouse', 'Keyboard', 'Monitor',
'Tablet', 'Smartphone', 'Smartwatch', 'Headphones'],
                    'Category': ['Електроника', 'Аксесоари', 'Електроника',
'Монитори', 'Таблети', 'Смартфони', 'Аксесоари ', 'Слушалки']}
df_products = pd.DataFrame(data_products)

unique_categories = df_products['Category'].unique()
print("Уникални категории преди почистване:\n", unique_categories)

# Забелязваме, че 'Аксесоари' и 'Аксесоари ' (с интервал накрая) се
считат за различни категории.
# Също така 'Монитори', 'Таблети', 'Смартфони', 'Слушалки' може да е по-
добре да бъдат под обща категория 'Електроника'.

# Можем да почистим тези неконсистентности (примерно):
df_products['Category'] = df_products['Category'].str.strip() #
Премахване на водещи и крайни интервали
df_products['Category'] = df_products['Category'].replace(['Монитори',
'Таблети', 'Смартфони', 'Слушалки'], 'Електроника')

unique_categories_cleaned = df_products['Category'].unique()
print("\nУникални категории след почистване:\n",
unique_categories_cleaned)

category_counts_cleaned = df_products['Category'].value_counts()
print("\nЧестота на категориите след почистване:\n",
category_counts_cleaned)
```

Обяснение:

`.unique()` е много полезен за бързо идентифициране на всички различни стойности в категорийни колони, което може да помогне за откриване на грешки при въвеждане на данни, неконсистентно форматиране или различни начини за представяне на една и съща категория. След като тези проблеми бъдат идентифицирани, могат да се предприемат стъпки за почистване и стандартизиране на данните. `.value_counts()` след почистването дава ясна представа за разпределението на коректните категории.

8. .T (транспониране):

- **Цел:** Размяна на редовете и колоните на DataFrame-а.
- **Казус:**
  - **Визуализация на данни:** Понякога е по-лесно да се създадат определени видове графики (например, bar plots), когато характеристиките на обектите са представени като редове, а самите обекти като колони.

Решение:

Използвайте `.T` за да транспонирате DataFrame-а и да промените ориентацията на данните.

```
import pandas as pd
import matplotlib.pyplot as plt

# Симулираме DataFrame, където характеристиките са редове, а продуктите
- колони
data_product_features = {'Лаптоп': [1200.50, 2.5, 15.6, 1],
                         'Мишка': [25.99, 1.0, None, 2],
                         'Клавиатура': [75.00, 1.5, None, 1],
                         'Монитор': [300.00, 3.0, 27.0, 1]}
index_features = ['Цена', 'Тегло (кг)', 'Размер на екрана (инча)',
'Наличност']
df_features_by_product = pd.DataFrame(data_product_features,
index=index_features)

print("Оригинален DataFrame (характеристики по редове, продукти по
колони):\n", df_features_by_product)

# Транспонираме DataFrame-а
df_products_by_feature = df_features_by_product.T
print("\nТранспониран DataFrame (продукти по редове, характеристики по
колони):\n", df_products_by_feature)

# Сега е по-лесно да създадем бар графика, показваща цените на
различните продукти
df_products_by_feature['Цена'].plot(kind='bar')
plt.ylabel("Цена (в лв.)")
```

```
plt.title("Цена на различни продукти")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# По същия начин, може да се визуализират и други характеристики.
```

Обяснение:

В някои случаи, структурата на DataFrame-а може да не е оптимална за определен вид анализ или визуализация. Транспонирането с `.T` позволява бързо да се промени ориентацията на данните, което може да улесни последващата обработка или представяне. В този пример, транспонирането прави по-лесно директното използване на колоната 'Цена' за създаване на бар графика, сравняваща цените на различните продукти.

- **Подготовка на данни за специфични модели:** Някои алгоритми за машино обучение може да очакват входните данни в транспониран вид.

Решение:

Използвайте `.T` за да транспонирате DataFrame-а и да го приведете във формата, очакван от външната библиотека или функция.

```
import pandas as pd
import numpy as np

# Симулираме DataFrame, където редовете са характеристики, а колоните са проби
data_features_samples = pd.DataFrame(np.random.rand(3, 5),
                                     index=['feature_A', 'feature_B', 'feature_C'],
                                     columns=['sample_1', 'sample_2', 'sample_3', 'sample_4', 'sample_5'])

print("Оригинален DataFrame (характеристики по редове, проби по колони):\n", data_features_samples)

# Да предположим, че външна функция очаква пробите да бъдат редове, а характеристиките - колони
```

```
# Транспонираме DataFrame-а
data_samples_features = data_features_samples.T
print("\nТранспониран DataFrame (проби по редове, характеристики по колони):\n", data_samples_features)

# Сега data_samples_features е в ориентация, където всяка проба е ред, а всяка характеристика е колона,
# което може да е форматът, очакван от външната функция.

# (Тук би следвало извикване на външната функция с транспонирания DataFrame)
# external_function(data_samples_features)
```

Обяснение:

Различните инструменти и библиотеки за анализ на данни могат да имат различни конвенции за структурата на входните данни. Транспонирането с `.T` е бърз начин да се адаптира формата на вашия DataFrame към изискванията на тези външни инструменти, без да се налага ръчно преструктуриране на данните.

Тези методи представляват основния инструментариум за първоначално опознаване и разбиране на вашите данни в Pandas. В реалния свят, те често се използват в комбинация, за да се получи цялостна картина на набора от данни преди извършване на по-сложен анализ или моделиране.

X. Въпроси

1. Каква е разликата между `.head()` и `.tail()`? В какви ситуации бихте използвали всеки от тях?
2. Каква информация предоставя методът `.info()` за един DataFrame? Защо е полезно да го използвате в началото на анализа на данни?
3. Какви основни статистически мерки се изчисляват от метода `.describe()` за числови колони? Каква допълнителна информация може да предостави за категорийни и времеви колони?
4. Как `.shape` и `.size` се различават? Каква информация дават за DataFrame и Series обекти?
5. Каква е разликата между `.dtype` и `.dtypes()`? Кога се използва всеки от тях?
6. Какво представляват `.index` и `.columns` атрибутите на DataFrame? Защо са важни при работа с данни?
7. Обяснете разликата между `.unique()`, `.nunique()` и `.value_counts()`. В какви сценарии е полезен всеки от тези методи?
8. Какво прави атрибутът `.T`? В какви ситуации може да е полезно транспонирането на DataFrame?

XI. Задачи

1. **Зареждане и първоначален преглед:**
   - Заредете CSV файл с данни (можете да използвате произволен публичен dataset или да създадете свой собствен примерен файл).
   - Използвайте `.head()` и `.tail()` за да разгледате първите и последните няколко реда.
   - Използвайте `.info()` за да получите обща информация за DataFrame-а.
2. **Размерност и типове данни:**
   - Изведете броя на редовете и колоните на заредения DataFrame, използвайки `.shape`.
   - Изведете общия брой на елементите, използвайки `.size`.
   - Разгледайте типовете данни на всяка колона с `.dtypes`. Има ли колони с неочаквани типове данни?
3. **Описателна статистика:**
   - Използвайте `.describe()` за да получите описателна статистика за всички числови колони. Какви заключения можете да направите за разпределението на данните?
   - Използвайте `.describe(include='object')` за да видите статистиката за категорийните колони. Какви са най-често срещаните стойности?
4. **Уникални стойности и честоти:**
   - Изберете една или няколко категорийни колони от вашия DataFrame.
   - Използвайте `.unique()` за да видите всички уникални стойности в тези колони.
   - Използвайте `.nunique()` за да преброите броя на уникалните стойности.
   - Използвайте `.value_counts()` за да видите честотата на всяка уникална стойност. Има ли неочаквани или неконсистентни записи?
5. **Транспониране:**
   - Изберете DataFrame с няколко числови колони и няколко реда.
   - Транспонирайте DataFrame-а с `.T`.

- Разгледайте `.head()` и `.dtypes()` на транспонирания DataFrame. Как са се променили данните и типовете им?

Допълнителни задачи (по-напреднали):

6. **Работа с индекс:**
   - Ако вашият DataFrame има колона, която може да служи като уникален идентификатор или времева серия, задайте я като индекс с `.set_index()`.
   - Проверете типа на новия индекс с `.index`.
   - Ако е времеви индекс, опитайте да извлечете данни за определен период.
7. **Комбиниране на методи:**
   - Използвайте комбинация от `.info()`, `.describe()` и `.value_counts()` за да получите цялостна представа за разпределението на липсващите стойности във вашия DataFrame.