

## Глава VII. Операции и Манипуляции на Данни.

# Операции и Манипуляции на Данни



След като вече сме научили как да зареждаме, разглеждаме и почистваме данни (включително справянето с липсващи стойности), е време да се потопим в същинската обработка и трансформация на данните с помощта на Pandas. Тази глава ще ви предостави арсенал от инструменти за извършване на сложни операции и манипулации върху вашите Series и DataFrame обекти.

В тази глава ще разгледаме следните ключови теми:

- **Векторизирани операции:** Ще се научим как да извършваме бързи и ефективни аритметични, сравнителни и логически операции върху цели Series и колони на DataFrame без използването на явни цикли. Това е една от основните сили на Pandas.
- **Прилагане на функции:** Ще разгледаме три мощни метода за прилагане на функции към данните:
  - `.map()` за елемент по елемент прилагане върху Series.
  - `.applymap()` за елемент по елемент прилагане върху DataFrame.
  - `.apply()` за прилагане на функции по редове или колони на DataFrame или върху цели Series. Ще видим и как да използваме `lambda` функции за кратки и гъвкави операции.
- **Създаване на нови колони:** Ще научим различни начини за добавяне на нови колони към DataFrame: с константни стойности, на базата на съществуващи колони (чрез векторизирани операции и функции), с помощта на метода `.assign()` и условно създаване на колони с `np.where` и `.loc`.
- **Преименуване на колони и индекси:** Ще разгледаме методите `.rename()`, `.rename_axis()` и `.set_axis()` за промяна на имената на колоните и етикетите на индексите.
- **Задаване и нулиране на индекса:** Ще научим как да превърнем съществуваща колона в индекс с `.set_index()` и как да върнем индекса обратно в колона с `.reset_index()`.
- **Сортиране на данни:** Ще разгледаме методите `.sort_values()` за сортиране по стойности в една или няколко колони (с контрол на посоката и обработката на липсващи стойности) и `.sort_index()` за сортиране по етикетите на индекса.
- **Промяна на типа данни на колони:** Ще научим как да конвертираме типа данни на колоните с `.astype()` и специализираните функции `pd.to_numeric()`, `pd.to_datetime()` и `pd.to_timedelta()`.
- **Работа с текстови данни:** Ще се запознаем с аксесора `.str`, който предоставя множество методи за ефективна обработка на текстови данни в Series.
- **Работа с данни за дата и час:** Ще разгледаме аксесора `.dt`, който дава достъп до различни компоненти на `datetime` и `timedelta` обекти в Series.
- **Работа с категорийни данни:** Ще направим кратко въведение в аксесора `.cat` за работа с категорийни данни (по-подробно ще бъде разгледано в част Б).
- **Използване на `.pipe()` за верижни операции:** Ще научим как да използваме метода `.pipe()` за създаване на по-четим и организиран код при последователно прилагане на множество операции върху DataFrame.

**Усвояването на тези техники ще ви даде гъвкавостта и мощта да трансформирате и анализирате вашите данни по ефективен и елегантен начин с Pandas.**

# I. Векторизирани операции (аритметични, сравнителни, логически)

Една от ключовите характеристики и предимства на Pandas (наследена от NumPy, върху която е изградена) е способността за извършване на **векторизирани операции**. Това означава, че можете да прилагате операции върху цели Series или колони на DataFrame наведнъж, без да е необходимо да използвате явни цикли (като for loop). Векторизираните операции са значително по-бързи и по-ефективни от итерирането през елементите, особено при големи набори от данни.

Нека разгледаме основните видове векторизирани операции:

## 1. Аритметични операции:

Можете да извършвате стандартни аритметични операции (+, -, \*, /, // - целочислено деление, % - остатък, \*\*\* - степен) между:

- Series и скаларна стойност.
- Два Series обекта (с автоматично подравняване по индекс).
- Колони на DataFrame и скаларна стойност.
- Две или повече колони на DataFrame.
- DataFrame и скаларна стойност (операцията се прилага към всеки елемент).
- Два DataFrame обекта (с автоматично подравняване по индекс и колони).

```
import pandas as pd

# Series и скалар
s = pd.Series([1, 2, 3, 4])
print("Оригинален Series:\n", s)
print("\nSeries + 5:\n", s + 5)
print("\nSeries * 2:\n", s * 2)

# Два Series
s2 = pd.Series([10, 20, 30, 40], index=s.index)
print("\nВтори Series:\n", s2)
print("\nSeries + Series2:\n", s + s2)

# DataFrame и скалар
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
print("\nОригинален DataFrame:\n", df)
print("\nDataFrame * 10:\n", df * 10)
```

```
# Две колони на DataFrame
print("\nКолона 'A' + колона 'B':\n", df['A'] + df['B'])

# Два DataFrame
data2 = {'A': [10, 20, 30], 'C': [7, 8, 9]}
df2 = pd.DataFrame(data2, index=df.index)
print("\nВтори DataFrame:\n", df2)
print("\nDataFrame + DataFrame2 (забележете NaN където колони/индекси липсват):\n", df + df2)
```

#### 1) Series и скалар:

- Създава се Pandas Series *s* с целочислени стойности.
- Извежда се оригиналният Series.
- Извършва се векторизирано събиране на скаларната стойност 5 с всеки елемент на *s*. Резултатът се извежда.
- Извършва се векторизирано умножение на скаларната стойност 2 с всеки елемент на *s*. Резултатът се извежда.

#### 2) Два Series:

- Създава се втори Pandas Series *s2* със същия индекс като *s*.
- Извежда се вторият Series.
- Извършва се векторизирано събиране на *s* и *s2*. Операцията се извършва поелементно въз основа на съвпадащия индекс. Резултатът се извежда.

#### 3) DataFrame и скалар:

- Създава се Pandas DataFrame *df* с две числови колони ('A' и 'B').
- Извежда се оригиналният DataFrame.
- Извършва се векторизирано умножение на скаларната стойност 10 с всеки елемент на *df*. Резултатът (нов DataFrame с умножените стойности) се извежда.

#### 4) Две колони на DataFrame:

- Извършва се векторизирано събиране на колоните 'A' и 'B' на *df*. Резултатът е нов Series, чийто индекс е същият като на *df*, а стойностите са сумата на съответните елементи от двете колони. Резултатът се извежда.

#### 5) Два DataFrame:

- Създава се втори Pandas DataFrame *df2* с колони 'A' и 'C', и същия индекс като *df*. Забележете, че *df* има колони 'A' и 'B', а *df2* има колони 'A' и 'C'.
- Извежда се вторият DataFrame.
- Извършва се векторизирано събиране на *df* и *df2*. Операцията се извършва поелементно въз основа на съвпадащия индекс и име на колона. Където има съвпадение на индекс и колона, стойностите се събират. Където има липса на съвпадение (например колона 'B' в *df* или колона 'C' в *df2*), резултатната стойност е NaN (Not a Number), което показва липсваща стойност след операцията. Резултатът се извежда.

## Накратко:

Примерът демонстрира как основните аритметични операции (+, \*) могат да се прилагат директно върху цели Series и DataFrame обекти, както и между тях. Той също така илюстрира автоматичното подравняване по индекс при операции между Series и по индекс и имена на колони при операции между DataFrame, като при липса на съвпадение се получават NaN стойности. Това е ключова концепция във векторизираните операции на Pandas, която осигурява ефективна и елегантна обработка на данни.

**!!!Важно за подравняването:** Когато извършвате операции между два Series или два DataFrame, Pandas автоматично подравнява данните по техните индекси (за Series) и по техните индекси и имена на колони (за DataFrame). Ако етикет не съществува в един от обектите, резултатът за тази комбинация ще бъде NaN.

## 2. Сравнителни операции:

Можете да използвате стандартни оператори за сравнение (==, !=, >, <, >=, <=) между:

- Series и скаларна стойност.
- Два Series обекта (с автоматично подравняване).
- Колони на DataFrame и скаларна стойност.
- Две или повече колони на DataFrame.
- DataFrame и скаларна стойност.
- Два DataFrame обекта (с автоматично подравняване).

Резултатът от сравнителните операции е булев Series или DataFrame, където True означава, че условието е изпълнено, а False - че не е.

```
print("\nSeries > 2:\n", s > 2)
print("\nDataFrame['A'] == DataFrame['B']:\n", df['A'] == df['B'])
print("\nDataFrame > 2:\n", df > 2)
```

*Булевите резултати от сравнителните операции често се използват за булево индексване (което ще разгледаме по-късно), за да филтрирате данни въз основа на определени условия.*

## 3. Логически операции:

Логическите операции (& - and, | - or, ~ - not, ^ - xor) могат да се прилагат върху булеви Series или булеви колони на DataFrame. Важно е да използвате **побитовите оператори** (&, |, ~, ^) вместо стандартните and, or, not, когато работите с цели Series или колони.

```
bool_s = pd.Series([True, False, True, False])
bool_s2 = pd.Series([False, True, True, False])
print("\nБулев Series 1:\n", bool_s)
print("\nБулев Series 2:\n", bool_s2)
print("\nbool_s & bool_s2:\n", bool_s & bool_s2)
```

```
print("\nbool_s | bool_s2:\n", bool_s | bool_s2)
print("\n~bool_s:\n", ~bool_s)

bool_df = pd.DataFrame({'P': [True, False, True], 'Q': [False, True, False]})
print("\nБулев DataFrame:\n", bool_df)
print("\nbool_df['P'] & bool_df['Q']:\n", bool_df['P'] & bool_df['Q'])
```

#### 1) Булеви Series:

- Създават се два Pandas Series `bool_s` и `bool_s2`, съдържащи булеви стойности (True и False).
- Извеждат се съдържанието на двата булеви Series.
- Извършва се векторизирана логическа операция AND (&) между `bool_s` и `bool_s2`. Резултатът е нов булев Series, в който стойността на всяка позиция е True само ако и в двата оригинални Series на същата позиция стойността е True. Резултатът се извежда.
- Извършва се векторизирана логическа операция OR (|) между `bool_s` и `bool_s2`. Резултатът е нов булев Series, в който стойността на всяка позиция е True, ако поне в един от двата оригинални Series на същата позиция стойността е True. Резултатът се извежда.
- Извършва се векторизирана логическа операция NOT (~) върху `bool_s`. Резултатът е нов булев Series, в който всяка стойност е инвертирана (True става False, а False става True). Резултатът се извежда.

#### 2) Булев DataFrame:

- Създава се Pandas DataFrame `bool_df` с две булеви колони ('P' и 'Q').
- Извежда се съдържанието на булевия DataFrame.
- Извършва се векторизирана логическа операция AND (&) между колоните 'P' и 'Q' на `bool_df`. Резултатът е булев Series, в който стойността на всеки ред е True само ако и в двете колони ('P' и 'Q') на същия ред стойността е True. Резултатът се извежда.

## Накратко:

Примерът демонстрира как побитовите логически оператори (& за AND, | за OR, ~ за NOT) могат да се прилагат директно върху булеви Series и булеви колони на DataFrame в Pandas. Тези операции се извършват векторизирано, което е ефективно за обработка на булеви маски и за комбиниране на условия при филтриране и анализ на данни. Важно е да се използват побитовите оператори вместо стандартните `and`, `or`, `not` при работа с цели Series или колони.

## 4. Предимства на векторизираните операции:

- **Производителност:** Векторизираните операции се изпълняват много по-бързо от еквивалентните операции с цикли, тъй като те се имплементират на ниско ниво (често на C или Fortran) и могат да се възползват от SIMD (Single Instruction, Multiple Data) инструкциите на процесора.



- **Четимост на кода:** Кодът, използващ векторизирани операции, е по-кратък и по-лесен за четене и разбиране. Той по-директно изразява операцията, която искате да извършите върху целия набор от данни.

## 5. Примери:

### а) Аритметични операции със Series

```
import pandas as pd

# Създайте Series от числа
temperatures_celsius = pd.Series([25.0, 28.5, 22.0, 30.1, 26.7])
print("Температури в Целзий:\n", temperatures_celsius)

# Преобразувайте ги във Фаренхайт, като използвате векторизирана операция
temperatures_fahrenheit = (temperatures_celsius * 9/5) + 32
print("\nТемператури във Фаренхайт:\n", temperatures_fahrenheit)

# Изчислете разликата между всяка температура и средната температура в Целзий
average_celsius = temperatures_celsius.mean()
difference_from_mean = temperatures_celsius - average_celsius
print("\nРазлика от средната температура (Целзий):\n",
      difference_from_mean)
```

Примерът демонстрира основни векторизирани аритметични операции върху Pandas Series.

- **Създава се Series от температури в Целзий:** Инициализира се Pandas Series с пет стойности, представляващи температури в градуси Целзий. Оригиналният Series се извежда на конзолата.
- **Преобразуване в градуси Фаренхайт (векторизирана операция):** Извършва се математическо преобразуване на всички температури от Целзий във Фаренхайт. Това се постига чрез прилагане на формулата  $(C * 9/5) + 32$  директно върху целия `temperatures_celsius` Series. Pandas извършва тази операция елемент по елемент, но по оптимизиран, векторизиран начин, без нужда от изричен цикъл. Резултатът е нов Series `temperatures_fahrenheit`, съдържащ еквивалентните температури в градуси Фаренхайт, който се извежда на конзолата.
- **Изчисляване на разликата от средната стойност (векторизирана операция):**
  1. Първо, се изчислява средната стойност на температурите в Целзий с помощта на метода `.mean()` върху `temperatures_celsius`. Резултатът се съхранява в променливата `average_celsius`.

2. След това се изчислява разликата между всяка индивидуална температура в `temperatures_celsius` и изчислената средна стойност. Това също е векторизирана операция, при която `average_celsius` се изважда от всеки елемент на `temperatures_celsius`. Резултатът е нов Series `difference_from_mean`, показващ отклонението на всяка температура от средната, който се извежда на конзолата.

### б) Сравнителни операции с DataFrame

```
import pandas as pd

# Създайте DataFrame с данни за продажби
sales_data = pd.DataFrame({
    'Продукт': ['A', 'B', 'A', 'C', 'B'],
    'Продажби_Q1': [100, 150, 120, 90, 160],
    'Продажби_Q2': [110, 140, 130, 100, 155]
})

print("Данни за продажби:\n", sales_data)

# Проверете кои продажби през Q1 са били над 100
q1_above_100 = sales_data['Продажби_Q1'] > 100
print("\nПродажби през Q1 над 100:\n", q1_above_100)

# Проверете кои продукти са имали по-високи продажби през Q2 спрямо Q1
q2_better_than_q1 = sales_data['Продажби_Q2'] >
sales_data['Продажби_Q1']
print("\nПродукти с по-добри продажби през Q2:\n", q2_better_than_q1)

# Създайте булев DataFrame, показващ къде продажбите са били над 120
# през който и да е от двата квартала
above_120 = (sales_data['Продажби_Q1'] > 120) |
(sales_data['Продажби_Q2'] > 120)
print("\nПродажби над 120 през Q1 или Q2:\n", above_120)
```

Примерът демонстрира основни векторизирани сравнителни и логически операции върху колони на Pandas DataFrame.

- 1) **Създава се DataFrame с данни за продажби:** Инициализира се Pandas DataFrame с три колони: 'Продукт' (съдържаща продуктови кодове), 'Продажби\_Q1' (съдържаща продажби за първо тримесечие) и 'Продажби\_Q2' (съдържаща продажби за второ тримесечие). Оригиналният DataFrame се извежда на конзолата.



- 2) **Проверка за продажби над 100 през Q1 (векторизирана сравнителна операция):** Извършва се сравнение на всяка стойност в колоната 'Продажби\_Q1' с числото 100, като се използва операторът за по-голямо (>). Резултатът е булев Series `q1_above_100`, където `True` означава, че продажбата за съответния продукт през Q1 е била над 100, а `False` - че не е. Този булев Series се извежда на конзолата.
- 3) **Проверка за по-високи продажби през Q2 спрямо Q1 (векторизирана сравнителна операция):** Извършва се сравнение между колоните 'Продажби\_Q2' и 'Продажби\_Q1'. За всеки ред се проверява дали стойността в 'Продажби\_Q2' е по-голяма от стойността в 'Продажби\_Q1'. Резултатът е булев Series `q2_better_than_q1`, където `True` показва продуктите с по-добри продажби през второто тримесечие. Този булев Series се извежда на конзолата.
- 4) **Създаване на булев Series за продажби над 120 през Q1 ИЛИ Q2 (векторизирани сравнителни и логическа операция):**
  - Първо се извършват две сравнителни операции: `sales_data['Продажби_Q1'] > 120` (проверка за продажби над 120 през Q1) и `sales_data['Продажби_Q2'] > 120` (проверка за продажби над 120 през Q2). Тези операции връщат два булеви Series.
  - След това се прилага логическата операция ИЛИ (`|`) между тези два булеви Series. За всеки ред резултатът ще бъде `True`, ако поне едно от условията (продажби над 120 през Q1 или продажби над 120 през Q2) е изпълнено. Резултатът е булев Series `above_120`, който се извежда на конзолата.

## Накратко:

Примерът илюстрира как Pandas позволява извършването на сравнения между колони и скаларни стойности, както и комбинирането на тези сравнения с логически оператори, за да се получат булеви резултати, които могат да се използват за филтриране или анализ на данните. Всички тези операции се извършват

### в) Логически операции с булев Series

```
import pandas as pd

# Създайте булев Series, показващ дали дадена транзакция е била голяма
или спешна
is_large = pd.Series([True, False, True, False, True])
is_urgent = pd.Series([False, True, True, False, False])
print("Големи транзакции:\n", is_large)
print("\nСпешни транзакции:\n", is_urgent)

# Намерете транзакциите, които са едновременно големи и спешни
large_and_urgent = is_large & is_urgent
print("\nГолеми И спешни транзакции:\n", large_and_urgent)

# Намерете транзакциите, които са или големи, или спешни (или и двете)
large_or_urgent = is_large | is_urgent
```

```
print("\nГолеми ИЛИ спешни транзакции:\n", large_or_urgent)

# Намерете транзакциите, които НЕ са големи
not_large = ~is_large
print("\nНе големи транзакции:\n", not_large)
```

Примерът демонстрира основни векторизирани логически операции върху Pandas Series от булеви стойности.

- 1) **Създават се два булеви series:** Инициализират се два Pandas Series: `is_large`, който показва дали дадена транзакция е голяма (True) или не (False), и `is_urgent`, който показва дали транзакцията е спешна (True) или не (False). И двата Series се извеждат на конзолата.
- 2) **Намиране на транзакции, които са едновременно големи И спешни (логическа операция AND):** Използва се побитовият оператор AND (&) между двата булеви Series. За всяка позиция, резултатът ще бъде True само ако и в двата Series на същата позиция има True. Резултатът се съхранява в `large_and_urgent` и се извежда на конзолата, показвайки кои транзакции отговарят и на двете условия.
- 3) **Намиране на транзакции, които са или големи, или спешни (или и двете) (логическа операция OR):** Използва се побитовият оператор OR (|) между двата булеви Series. За всяка позиция, резултатът ще бъде True, ако поне в един от двата Series на същата позиция има True. Резултатът се съхранява в `large_or_urgent` и се извежда на конзолата, показвайки транзакциите, които са или големи, или спешни, или и двете.
- 4) **Намиране на транзакции, които НЕ са големи (логическа операция NOT):** Използва се побитовият оператор NOT (~) пред булевия Series `is_large`. Той инвертира всяка булева стойност в Series-a (True става False, а False става True). Резултатът се съхранява в `not_large` и се извежда на конзолата, показвайки транзакциите, които не са били класифицирани като големи.

## Накратко:

Примерът демонстрира как да се извършват основни логически операции (AND, OR, NOT) върху булеви Series в Pandas по векторизиран начин. Тези операции са фундаментални за филтриране и анализ на данни въз основа на множество условия. Важно е да се използват побитовите оператори (&, |, ~) вместо стандартните `and`, `or`, `not` при работа с цели Series.

## Допълнителни Указания:

- Наблюдавайте резултата и се опитайте да разберете как векторизираната операция се прилага върху целия Series или колона на DataFrame едновременно.
- Експериментирайте, като променяте данните или самите операции, за да видите как се променя резултатът. Например, опитайте да умножите две колони на `sales_data` или да сравните `temperatures_celsius` с друга Series от температури.

## Казус 1: Анализ на продажби

Представете си, че имате `DataFrame`, съдържащ данни за продажби на различни продукти за определен период. Искате да анализирате тези данни, като изчислите общата стойност на всяка продажба, да определите кои продажби са над определена сума и да филтрирате продажбите на конкретен продукт.

```
import pandas as pd

# Създаваме DataFrame с данни за продажби
sales_data = {
    'Продукт': ['Телевизор', 'Лаптоп', 'Мишка', 'Клавиатура',
                'Телевизор', 'Слушалки'],
    'Количество': [2, 1, 5, 3, 1, 10],
    'Единична_цена': [500.00, 1200.00, 25.00, 40.00, 550.00, 80.00]
}
df_sales = pd.DataFrame(sales_data)
```

- Казус 1.1: Изчислете общата стойност на всяка продажба (`количество * единична_цена`).
- Казус 1.2: Определете кои продажби имат обща стойност над 1000 лв.
- Казус 1.3: Филтрирайте всички продажби на продукт 'Телевизор'.

## Решение на Казус 1:

```
# 1.1: Изчисляване на общата стойност (векторизирана аритметична
операция)
df_sales['обща_стойност'] = df_sales['количество'] *
df_sales['единична_цена']
print("DataFrame с обща стойност:\n", df_sales)

# 1.2: Определяне на продажби с обща стойност над 1000 лв.
(векторизирана операция за сравнение)
high_value_sales = df_sales['обща_стойност'] > 1000
print("\nПродажби с обща стойност над 1000 лв.: \n",
df_sales[high_value_sales])

# 1.3: Филтриране на продажби на 'Телевизор' (векторизирана операция за
сравнение и логическо индексване)
tv_sales = df_sales['продукт'] == 'Телевизор'
```

```
print("\nПродажби на 'Телевизор':\n", df_sales[tv_sales])
```

## Казус 2: Анализ на резултати от тест

Имате DataFrame, съдържащ резултати от тест на ученици по два предмета. Искате да определите средния резултат на всеки ученик, да проверите кои ученици са издържали и двата теста (при минимален резултат 60) и да класирате учениците според техния среден резултат.

```
import pandas as pd

# Създаваме DataFrame с резултати от тест
grades_data = {
    'ученик': ['Алекс', 'Борис', 'Вера', 'Георги', 'Диана'],
    'математика': [75, 55, 88, 62, 95],
    'физика': [80, 65, 78, 58, 90]
}
df_grades = pd.DataFrame(grades_data)
```

- Казус 2.1: Изчислете средния резултат на всеки ученик.
- Казус 2.2: Определете кои ученици са издържали и двата теста (резултат  $\geq 60$ ).
- Казус 2.3: Класирайте учениците според техния среден резултат (в низходящ ред).

## Решение на Казус 2:

```
# 2.1: Изчисляване на средния резултат (векторизирани аритметични
операции и метод)
df_grades['среден_резултат'] = (df_grades['математика'] +
df_grades['физика']) / 2
print("DataFrame със среден резултат:\n", df_grades)

# 2.2: Определяне на издържалите и двата теста (векторизирани операции
за сравнение и логически оператор)
passed_math = df_grades['математика'] >= 60
passed_physics = df_grades['физика'] >= 60
passed_both = passed_math & passed_physics
print("\nУченици, издържали и двата теста:\n", df_grades[passed_both])

# 2.3: Класиране според средния резултат (векторизиран метод за
сортиране)
```

```
ranked_grades = df_grades.sort_values(by='среден_резултат',
ascending=False)
print("\nКласиране на учениците по среден резултат:\n", ranked_grades)
```

## II. Прилагане на функции: Елемент по елемент (`.map()` за `Series`, `.applymap()` за `DataFrame`).

Понякога векторизираните операции не са достатъчни за сложни трансформации, или искате да приложите потребителска функция към данните си. Pandas предоставя методите `.map()` и `.applymap()` за прилагане на функции към всеки отделен елемент на `Series` или `DataFrame` съответно.

### 1. `.map()` за `Series`:

Методът `.map()` се използва за прилагане на функция към всеки елемент в `Series`. Той може да приема като аргумент:

- **Функция (callable):** Тази функция ще бъде извикана за всеки елемент от `Series`.
- **Речник (dictionary) или `Series`:** В този случай, стойностите в `Series` ще бъдат заменени със стойностите от речника или другия `Series`, където ключовете/индексите съвпадат.

#### а) Прилагане на функция към `Series`

```
import pandas as pd

# Създаваме Series от имена
names = pd.Series(['alice', 'bob', 'charlie'])
print("Оригинален Series от имена:\n", names)

# Дефинираме функция за капитализиране на име
def capitalize_name(name):
    return name.capitalize()

# Прилагаме функцията към всеки елемент на Series с .map()
capitalized_names = names.map(capitalize_name)
print("\nКапитализиран Series от имена:\n", capitalized_names)

# Използване на lambda функция за същата цел
capitalized_names_lambda = names.map(lambda x: x.upper())
```

```
print("\nИмена с главни букви (lambda):\n", capitalized_names_lambda)
```

Примерът демонстрира как се използва методът `.map()` в Pandas за прилагане на функция към всеки елемент на Series.

- **Създава се Series от имена:** Инициализира се Pandas Series с три имена: 'alice', 'bob' и 'charlie'. Оригиналният Series се извежда на конзолата.
- **Дефинира се обикновена функция:** Създава се функция с име `capitalize_name`, която приема един аргумент (име) и връща същото име, но с първа буква главна (капитализирано).
- **Прилага се обикновена функция с `.map()`:** Методът `.map()` се използва върху Series-a `names`, като му се подава функцията `capitalize_name` като аргумент. Резултатът е нов Series, в който всяко име от оригиналния Series е капитализирано. Този капитализиран Series се извежда на конзолата.
- **Прилага се lambda функция с `.map()`:** Същата операция (преобразуване на имената в главни букви) се извършва отново, но този път вместо дефинирана функция се използва анонимна (lambda) функция. Lambda функцията `lambda x: x.upper()` приема всеки елемент (`x`) от Series-a и го преобразува в главни букви. Резултатът е Series с имената, написани с главни букви, който също се извежда на конзолата.

#### *б) Използване на речник за замяна на стойности в Series*

```
# Създаваме Series от кодове на продукти
product_codes = pd.Series(['A1', 'B2', 'A1', 'C3', 'B2'])
print("\nSeries от кодове на продукти:\n", product_codes)

# Създаваме речник за съответствие между кодове и пълни имена
product_mapping = {'A1': 'Ябълка', 'B2': 'Банан', 'C3': 'Портокал'}

# Заменяме кодовете с пълните имена, използвайки .map() и речника
product_names = product_codes.map(product_mapping)
print("\nSeries от имена на продукти:\n", product_names)

# За стойности, които не съществуват в речника, резултатът ще бъде NaN
product_codes_with_unknown = pd.Series(['A1', 'D4', 'B2'])
product_names_with_nan = product_codes_with_unknown.map(product_mapping)
print("\nSeries с неизвестен код (резултатът е NaN):\n",
product_names_with_nan)
```



```
# Можем да предоставим стойност по подразбиране с .get() в lambda
функцията
product_names_default = product_codes_with_unknown.map(lambda x:
product_mapping.get(x, 'Неизвестен продукт'))
print("\nSeries с неизвестен код (със стойност по подразбиране): \n",
product_names_default)
```

*Примерът демонстрира използването на метода .map() върху Pandas Series за замяна на стойности въз основа на речник.*

- 1) **Създава се series от кодове на продукти:** Инициализира се Pandas Series с няколко кратки кодове на продукти ('A1', 'B2', 'A1', 'C3', 'B2'). Оригиналният Series се извежда на конзолата.
- 2) **Създава се речник за съответствие:** Дефинира се Python речник product\_mapping, който съдържа съответствие между кодовете на продуктите (като ключове) и техните пълни имена (като стойности).
- 3) **Замяна на кодове с имена с помощта на .map() и речник:** Методът .map() се прилага върху product\_codes Series, като му се подава речникът product\_mapping. За всеки код в product\_codes, .map() търси съответстващ ключ в product\_mapping и заменя кода със съответната стойност (пълното име на продукта). Резултатът е нов Series product\_names, съдържащ пълните имена на продуктите, който се извежда на конзолата.
- 4) **Обработка на неизвестни кодове (резултат nan):** Създава се нов Series product\_codes\_with\_unknown, който съдържа един код ('D4'), който не съществува като ключ в product\_mapping. Когато .map() се приложи с този речник, за неизвестния код 'D4' не се намира съответствие и стойността в резултатиращия Series product\_names\_with\_nan става NaN (Not a Number), което показва липсваща стойност. Този Series се извежда на конзолата.
- 5) **Предоставяне на стойност по подразбиране с lambda функция и .get():** Отново се прилага .map() върху product\_codes\_with\_unknown, но този път се използва lambda функция. Тази lambda функция приема всеки код (x) и използва метода .get() на речника product\_mapping. Методът .get(x, 'Неизвестен продукт') търси ключа x в речника. Ако ключът съществува, връща съответната стойност; ако не съществува, връща стойността по подразбиране 'Неизвестен продукт'. Резултатиращият Series product\_names\_default показва как да се обработят неизвестни кодове, като им се присвои конкретна стойност вместо NaN. Този Series се извежда на конзолата.

## Накратко:

Примерът илюстрира как .map() може да се използва за ефективно преобразуване на стойности в Series въз основа на съществуващо съответствие, дефинирано в речник. Също така показва как се обработват липсващи ключове в речника (водят до NaN) и как може да се предостави стойност по подразбиране за такива случаи с помощта на lambda функция и метода .get() на речника.

## 2. .applymap() За DataFrame:

Методът .applymap() е подобен на .map(), но се прилага към всеки елемент в целия DataFrame. Той приема само една функция като аргумент.

Прилагане на функция към всеки елемент на DataFrame

```

import pandas as pd

# Създаваме DataFrame с числови данни
data = pd.DataFrame({'col1': [1, 2, 3], 'col2': [4, 5, 6]})
print("Оригинален DataFrame:\n", data)

# Дефинираме функция за удвояване на число
def double_value(x):
    return x * 2

# Прилагаме функцията към всеки елемент на DataFrame с .applymap()
doubled_df = data.applymap(double_value)
print("\nDataFrame с удвоени стойности:\n", doubled_df)

# Използване на lambda функция за същата цел
squared_df = data.applymap(lambda x: x ** 2)
print("\nDataFrame с квадратни стойности (lambda):\n", squared_df)

# Можем да прилагаме и функции, които връщат различен тип данни (в този
случай, низ)
def format_value(x):
    return f"Value: {x}"

formatted_df = data.applymap(format_value)
print("\nDataFrame с форматиранни стойности:\n", formatted_df)

```

Примерът демонстрира използването на метода `.applymap()` в Pandas за прилагане на функция към всеки отделен елемент на `DataFrame`.

- 1) **Създава се DataFrame с числови данни:** Инициализира се Pandas `DataFrame` с две колони ('col1' и 'col2') и три реда, съдържащи числови стойности. Оригиналният `DataFrame` се извежда на конзолата.
- 2) **Дефинира се обикновена функция:** Създава се функция с име `double_value`, която приема едно число като аргумент и връща неговата удвоена стойност.
- 3) **Прилагане на обикновена функция с `.applymap()`:** Методът `.applymap()` се използва върху `DataFrame`-а `data`, като му се подава функцията `double_value`. Тази функция се прилага към всеки елемент (всяка клетка) в `DataFrame`-а. Резултатът е нов `DataFrame` `doubled_df`, в който всяка стойност от оригиналния `DataFrame` е умножена по 2. Този `DataFrame` се извежда на конзолата.

- 4) **Използване на lambda функция за същата цел:** Същата операция (повдигане на квадрат) се извършва отново, но този път вместо дефинирана функция се използва анонимна (lambda) функция. Lambda функцията `lambda x: x ** 2` приема всеки елемент (`x`) от `DataFrame`-а и връща неговия квадрат. Резултатът е нов `DataFrame` `squared_df`, съдържащ квадратите на оригиналните стойности, който се извежда на конзолата.
- 5) **Прилагане на функция, връщаща различен тип данни:** Дефинира се функция `format_value`, която приема число като аргумент и връща низ, съдържащ текста "Value: " и оригиналното число. Методът `.applymap()` се използва отново с тази функция. Резултатът е нов `DataFrame` `formatted_df`, в който всяка оригинална числова стойност е преобразувана в низ по зададения формат. Този `DataFrame` се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.applymap()` позволява прилагането на функция към всеки отделен елемент на `DataFrame`, независимо от неговия тип. Това е полезно за извършване на трансформации на ниво клетка, като математически операции, форматиране или преобразуване на типа данни на елементите. Както се вижда, `.applymap()` може да работи както с предварително дефинирани функции, така и с кратки lambda функции.

## 3. Важно:

- `.map()` се използва само за `Series`.
- `.applymap()` се използва само за `DataFrame`.
- И двата метода връщат нов `Series` или `DataFrame` и не променят оригиналния обект (освен ако не използвате присвояване).
- Тези методи са по-гъвкави от векторизираните операции, но могат да бъдат по-бавни за много големи набори от данни, тъй като функцията се извиква за всеки елемент поотделно (въпреки че `Pandas` вътрешно оптимизира процеса).

## Казус 1: Стандартизиране на продуктови наименования (Series)

Представете си, че имате `Series`, съдържащ наименования на продукти, които са въведени по различен начин (с различни главни и малки букви, допълнителни интервали и т.н.). Искате да ги стандартизирате, като ги превърнете в малки букви и премахнете водещите и завършващи интервали.

```
import pandas as pd

# Създаваме Series с нестандартизирани наименования на продукти
product_names = pd.Series([' Apple iPhone X ', 'Samsung Galaxy S10+',
' xiaomi mi 9 ', 'Google Pixel 3a'])
```

*# Казус: Стандартизирайте наименованията на продуктите, като ги превърнете в малки букви и премахнете излишните интервали.*

## Решение на Казус 1:

```
# Дефинираме функция за стандартизиране на наименование
def standardize_name(name):
    return name.strip().lower()

# Прилагаме функцията елемент по елемент с .map()
standardized_names = product_names.map(standardize_name)

print("Оригинални наименования:\n", product_names)
print("\nСтандартизирани наименования:\n", standardized_names)
```

*В този случай, функцията `standardize_name` се прилага към всеки елемент от `product_names` Series-a, като връща нов Series със стандартизирани наименования.*

## Казус 2: Форматиране на числови данни в DataFrame (DataFrame)

Представете си, че имате DataFrame, съдържащ финансови данни, където числата са представени като десетични дробни. Искате да ги форматирате като низове с две десетични места и добавен символ за валута (€).

```
import pandas as pd

# Създаваме DataFrame с финансови данни
financial_data = {
    'продукт': ['А', 'Б', 'В'],
    'цена': [12.345, 98.7654, 5.6789],
    'количество': [10, 5, 20]
}
df_financial = pd.DataFrame(financial_data)

# Казус: Форматирайте колоната 'цена' като низове с две десетични места
и символ '€'.
```

## Решение на Казус 2:

```
# Дефинираме функция за форматиране на число като валута
def format_currency(value):
    return f'€{value:.2f}'

# Прилагаме функцията елемент по елемент само върху колоната 'цена'
df_financial['цена_форматирана'] =
df_financial['цена'].map(format_currency)
print("DataFrame преди форматиране:\n", df_financial)
print("\nDataFrame след форматиране на цената:\n",
df_financial[['продукт', 'цена_форматирана', 'количество']])

# Ако искаме да приложим форматиране към всички числови колони (в този
случай само 'цена'),
# можем да използваме .applymap() (въпреки че тук е по-специфична
задача)
def format_if_float(value):
    if isinstance(value, float):
        return f'€{value:.2f}'
    return value

# df_financial_formatted = df_financial.applymap(format_if_float)
# print("\nDataFrame след .applymap() (само цената е форматирана):\n",
df_financial_formatted)
```

В този казус, използваме `.map()` директно върху `Series`-а 'цена', за да приложим функцията за форматиране само към тази колона. Въпреки че `.applymap()` може да се използва върху целия `DataFrame`, в този случай е по-ефективно да се използва `.map()` върху конкретната колона. В коментарите е показан и пример как би се използвал `.applymap()` с условна логика за прилагане само към числови стойности.

Тези казуси илюстрират как `.map()` и `.applymap()` позволяват гъвкаво прилагане на персонализирани функции към отделни елементи на `Series` и `DataFrame`, което е полезно за почистване, трансформиране и форматиране на данни.

### III. Прилагане на функции: по редове/колони (`.apply()`).

Методът `.apply()` е изключително гъвкав и може да се използва както за `Series`, така и за `DataFrame`. За разлика от `.map()` и `.applymap()`, които работят елемент по елемент, `.apply()` може да прилага функцията:

- **Към цял `Series`:** В този случай функцията получава `Series` като аргумент.
- **По редове на `DataFrame` (`axis=1`):** Функцията се прилага към всеки ред като `Series` обект (индексиран по имената на колоните).
- **По колони на `DataFrame` (`axis=0` - по подразбиране):** Функцията се прилага към всяка колона като `Series` обект (индексиран по индекса на `DataFrame`).

#### Синтаксис:

```
series.apply(func, convert_dtype=True, args=(), **kwargs)
dataframe.apply(func, axis=0, raw=False, result_type=None, args=(),
**kwargs)
```

Нека разгледаме примери за различните начини на използване на `.apply()`:

#### 1. Прилагане на функцията към цял `Series`:

```
import pandas as pd

# Създаваме Series от числа
numbers = pd.Series([1, 2, 3, 4, 5])
print("Оригинален Series:\n", numbers)

# Дефинираме функция за изчисляване на сумата и средната стойност
def sum_and_mean(s):
    return pd.Series({'Сума': s.sum(), 'Средно': s.mean()})

# Прилагаме функцията към Series
result = numbers.apply(sum_and_mean)
print("\nРезултат от прилагане на функцията към Series:\n", result)
```

*Примерът демонстрира използването на метода `.apply()` върху `Pandas Series`, където функцията, подадена на `.apply()`, обработва целия `Series` и връща друг `Series` като резултат.*



- 1) **Създава се Series от числа:** Инициализира се Pandas Series с пет числови стойности (1, 2, 3, 4, 5). Оригиналният Series се извежда на конзолата.
- 2) **Дефинира се функция за сума и средна стойност:** Създава се функция с име `sum_and_mean`, която приема Pandas Series като аргумент (`s`). Вътре в функцията се изчисляват сумата (`s.sum()`) и средната стойност (`s.mean()`) на елементите на този Series. Функцията връща нов Pandas Series с два елемента: 'Сума', съдържащ изчислената сума, и 'Средно', съдържащ изчислената средна стойност.
- 3) **Прилагане на функцията към Series с `.apply()`:** Методът `.apply()` се използва върху `numbers` Series, като му се подава функцията `sum_and_mean`. В този контекст, функцията `sum_and_mean` се извиква веднъж, като целият `numbers` Series се подава като аргумент `s`. Резултатът от това извикване е Series-ът, върнат от `sum_and_mean`, който съдържа сумата и средната стойност на оригиналния `numbers` Series. Този резултатен Series се съхранява в променливата `result` и се извежда на конзолата.

## Накратко:

Примерът показва, че когато `.apply()` се използва върху Series и функцията, която му се подава, е предназначена да работи с целия Series, резултатът може да бъде друг Series (или скаларна стойност, в зависимост от това какво връща функцията). В този случай, `.apply()` обобщава информация от целия `numbers` Series в нов Series, съдържащ сумата и средната му стойност.

## 2. Прилагане на функция по колони на DataFrame (`axis=0`):

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с числови данни
data = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, np.nan]})
print("\nОригинален DataFrame:\n", data)

# Дефинираме функция за намиране на минимума и максимума на колона
def min_max(col):
    return pd.Series({'Минимум': col.min(), 'Максимум': col.max()})

# Прилагаме функцията към всяка колона (axis=0 е по подразбиране)
result_by_column = data.apply(min_max)
print("\nРезултат от прилагане на функция по колони:\n",
      result_by_column)

# Използване на lambda функция за изчисляване на средната стойност на
всяка колона
mean_by_column = data.apply(lambda col: col.mean())
```

```
print("\nСредна стойност по колони (lambda):\n", mean_by_column)
```

Примерът демонстрира използването на метода `.apply()` върху Pandas DataFrame за прилагане на функция към всяка колона (`axis=0`).

- 1) **Създава се DataFrame с числови данни:** Инициализира се Pandas DataFrame с две колони ('A' и 'B') и три реда. Забележете, че в колона 'B' има една липсваща стойност (`np.nan`). Оригиналният DataFrame се извежда на конзолата.
- 2) **Дефинира се функция за намиране на минимум и максимум на колона:** Създава се функция с име `min_max`, която приема Pandas Series (представляваща колона) като аргумент (`col`). Вътре в функцията се изчисляват минималната стойност (`col.min()`) и максималната стойност (`col.max()`) на елементите в тази колона. Функцията връща нов Pandas Series с два елемента: 'Минимум' и 'Максимум', съдържащи съответно минималната и максималната стойност от колоната.
- 3) **Прилагане на функцията по колони с `.apply()`:** Методът `.apply()` се използва върху data DataFrame без изрично задаване на `axis` (по подразбиране `axis=0`, което означава прилагане към всяка колона). За всяка колона на DataFrame-a, функцията `min_max` се извиква, като съответната колона се подава като аргумент. Резултатът е нов DataFrame `result_by_column`, където всеки ред представлява колона от оригиналния DataFrame ('A' и 'B'), а колоните съдържат резултатите от функцията `min_max` ('Минимум' и 'Максимум'). Този резултатен DataFrame се извежда на конзолата. За колона 'B', минимумът ще бъде 4, а максимумът 5 (липсващата стойност `NaN` се игнорира от `.min()` и `.max()`).
- 4) **Използване на lambda функция за изчисляване на средната стойност на всяка колона:** Методът `.apply()` се използва отново върху data DataFrame, този път с анонимна (lambda) функция. Lambda функцията `lambda col: col.mean()` приема Pandas Series (колона) като аргумент (`col`) и връща средната стойност на елементите в тази колона (`col.mean()`). Резултатът е Pandas Series `mean_by_column`, където индексът съответства на имената на колоните ('A' и 'B'), а стойностите са средните стойности на всяка колона. Този Series се извежда на конзолата. За колона 'B', средната стойност ще бъде  $(4 + 5) / 2 = 4.5$  (отново, `NaN` се игнорира от `.mean()`).

## Накратко:

Примерът илюстрира как `.apply()` може да се използва за прилагане на функция, която обобщава информация за всяка колона на DataFrame. Той показва както използването на предварително дефинирана функция, която връща Series, така и използването на кратка lambda функция, която връща скаларна стойност. Резултатът от `.apply()` зависи от това какво връща подадената функция. Когато се прилага по колони, резултатът обикновено има индекс, съответстващ на имената на колоните.

### 3. Прилагане на функция по редове на DataFrame (`axis=1`):

```
# Създаваме DataFrame с данни за продукти
products = pd.DataFrame({
    'Име': ['Ябълка', 'Банан', 'Портокал'],
    'Цена': [1.20, 0.80, 1.00],
```

```

        'Количество': [100, 150, 80]
    })
print("\nОригинален DataFrame:\n", products)

# Дефинираме функция за изчисляване на общата стойност на ред
def calculate_total(row):
    return row['Цена'] * row['Количество']

# Прилагаме функцията към всеки ред (axis=1)
total_value = products.apply(calculate_total, axis=1)
print("\nОбща стойност на всеки продукт:\n", total_value)

# Добавяме резултата като нова колона
products['Обща_стойност'] = total_value
print("\nDataFrame с добавена колона за обща стойност:\n", products)

# Използване на lambda функция за същата цел
products['Отстъпка'] = products.apply(lambda row: row['Обща_стойност'] *
0.10 if row['Количество'] > 90 else 0, axis=1)
print("\nDataFrame с добавена колона за отстъпка (lambda):\n", products)

```

Примерът демонстрира използването на метода `.apply()` върху Pandas DataFrame за прилагане на функция към всеки ред (`axis=1`) и създаване на нови колони въз основа на резултатите.

- 1) **Създава се DataFrame с данни за продукти:** Инициализира се Pandas DataFrame с три колони: 'Име', 'Цена' и 'Количество', съдържащи информация за различни продукти. Оригиналният DataFrame се извежда на конзолата.
- 2) **Дефинира се функция за изчисляване на общата стойност на ред:** Създава се функция с име `calculate_total`, която приема Pandas Series (представляващ ред) като аргумент (`row`). Вътре в функцията се изчислява общата стойност за продукта, като се умножават стойностите от колоните 'Цена' и 'Количество' на този ред. Функцията връща резултата от това умножение.
- 3) **Прилагане на функцията по редове с `.apply()`:** Методът `.apply()` се използва върху `products` DataFrame с аргумент `axis=1`, което указва, че функцията `calculate_total` трябва да бъде приложена към всеки ред. Резултатът от прилагането на функцията към всеки ред е Pandas Series `total_value`, където индексът съответства на индекса на оригиналния DataFrame, а стойностите са изчислените общи стойности за всеки продукт. Този Series се извежда на конзолата.
- 4) **Добавяне на резултата като нова колона:** Създава се нова колона с име 'Обща\_стойност' в `products` DataFrame и ѝ се присвоява `total_value` Series. Това добавя колона с изчислените общи стойности към DataFrame-а, който след това се извежда на конзолата.

- 5) **Използване на lambda функция за създаване на условна колона:** Създава се още една нова колона с име 'Отстъпка'. За изчисляване на стойностите в тази колона се използва `.apply()` с `axis=1` и `lambda` функция. Lambda функцията приема всеки ред (`row`) и проверява дали стойността в колоната 'Количество' е по-голяма от 90. Ако е така, връща 10% от стойността в колоната 'Обща\_стойност'; в противен случай връща 0. Резултатът от прилагането на тази `lambda` функция към всеки ред се присвоява на новата колона 'Отстъпка', а обновеният `DataFrame` се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.apply()` с `axis=1` позволява да се прилага функция, която работи с данните от няколко колони на всеки ред, за да се изчисли нова стойност. Резултатът от това може да бъде използван за създаване на нови колони в `DataFrame`-а, като се използва както дефинирана функция, така и кратка `lambda` функция за по-прости условни изчисления.

## 4. Важно:

- `.apply()` е по-гъвкав от `.map()` и `.applymap()`, но може да бъде по-бавен за прости елементни операции, тъй като функцията се извиква за цели `Series` (колона или ред). За елементни трансформации `.map()` (за `Series`) и `.applymap()` (за `DataFrame`) често са по-ефективни.
- Вътре във функцията, подадена на `.apply()`, ще работите със `Series` обекти (представляващи колона или ред).
- Параметърът `raw=True` (само за `DataFrame`) може да подобри производителността за числови данни, като подава `NumPy array` вместо `Series` към функцията. Въпреки това, трябва да внимавате, тъй като губите достъп до етикетите на колоните/индекса.
- Параметърът `result_type` може да се използва за контролиране на типа на резултата (например, `'expand'` за разширяване на резултати, подобни на `Series`, в колони на `DataFrame`).

## Казус 1: Изчисляване на разликата между най-висока и най-ниска цена за продукт (по редове)

Представете си, че имате `DataFrame`, съдържащ дневни цени на различни продукти от няколко магазина. Искате да намерите разликата между най-високата и най-ниската цена, отчетена за всеки продукт в рамките на деня.

```
import pandas as pd

# Създаваме DataFrame с дневни цени на продукти
price_data = {
    'продукт': ['A', 'A', 'B', 'B', 'B', 'B'],
    'магазин': ['M1', 'M2', 'M1', 'M2', 'M1', 'M2'],
```

```

    'цена': [10.50, 11.20, 25.00, 24.50, 5.60, 5.80]
}
df_prices = pd.DataFrame(price_data)

# За да илюстрираме .apply() по редове, ще групираме по продукт и ще
намерим
# разликата между макс. и мин. цена за всеки продукт (въпреки че
.groupby().agg()
# би бил по-ефективен за този конкретен случай).
# Нека създадем по-подходящ казус за .apply() по редове.

# Нов DataFrame, където искаме да приложим логика на базата на стойности
от няколко колони в ред.
exam_results = {
    'ученик': ['Алекс', 'Борис', 'Вера'],
    'математика': [70, 55, 88],
    'физика': [80, 65, 78],
    'химия': [75, 60, 92]
}
df_exams = pd.DataFrame(exam_results)

# Казус 1: Определете дали всеки ученик е издържал успешно, ако трябва
да има среден резултат
# поне 65 от всички предмети.

```

## Решение на Казус 1:

```

# Дефинираме функция за проверка дали ученик е издържал
def is_passing(row):
    average_score = row[['математика', 'физика', 'химия']].mean()
    return "Издържал" if average_score >= 65 else "Не е издържал"

# Прилагаме функцията по редове (axis=1)
df_exams['статус'] = df_exams.apply(is_passing, axis=1)

print("Резултати от изпити:\n", df_exams)

```

## Казус 2: Намиране на продукта с най-висока средна цена (по колони)

Представете си, че имате DataFrame, съдържащ средни цени на различни продукти за няколко различни години. Искате да намерите продукта, който има най-висока средна цена през всички години.

```
import pandas as pd

# Създаваме DataFrame със средни цени на продукти по години
average_prices = {
    'продукт': ['A', 'B', 'B'],
    '2020': [10.20, 25.50, 5.80],
    '2021': [10.80, 26.00, 6.20],
    '2022': [11.00, 27.00, 6.50]
}
df_avg_prices = pd.DataFrame(average_prices).set_index('продукт')

# Казус 2: Намерете продукта с най-висока средна цена за всички години.
```

### Решение на Казус 2:

```
# Дефинираме функция за изчисляване на средната цена на продукт
def get_average_price(column):
    return column.mean()

# Прилагаме функцията по колони (axis=0) към числовите колони и намираме
средната цена за всеки продукт
df_avg_prices['средна_цена'] = df_avg_prices[['2020', '2021',
'2022']].apply(get_average_price, axis=1)

# Намираме продукта с най-висока средна цена
product_with_highest_average = df_avg_prices['средна_цена'].idxmax()
highest_average_price = df_avg_prices['средна_цена'].max()

print("Средни цени по продукти:\n", df_avg_prices)
```



```
print(f"\nПродуктът с най-висока средна цена е:  
{product_with_highest_average} ({highest_average_price:.2f})")
```

В този казус, функцията `get_average_price` се прилага по редове (`axis=1`) към колоните с цени за всяка година, за да се изчисли средната цена за всеки продукт. След това, използваме `.idxmax()` и `.max()` върху `Series`-а 'средна\_цена', за да намерим продукта с най-висока средна стойност.

Тези казуси илюстрират как `.apply()` може да се използва за прилагане на сложна логика както по редове, така и по колони на `DataFrame`, особено когато в изчислението участват стойности от няколко колони (за `.apply(axis=1)`) или когато искаме да обобщим информация за всяка колона (`.apply(axis=0)`).

## IV. Използване на `lambda` функции

`Lambda` функциите в Python са малки, анонимни (без име) функции, които могат да бъдат дефинирани на един ред. Те са особено полезни, когато искате да приложите проста функция към данни в `Pandas` без да е необходимо да дефинирате отделна, пълноценна функция с `def`.

`Lambda` функциите имат следния синтаксис:

```
lambda arguments: expression
```

- `lambda`: Ключовата дума, която указва, че се дефинира анонимна функция.
- `arguments`: Списък от аргументи, които функцията приема (може да бъде празен).
- `expression`: Единичен израз, който се оценява и връща като резултат от функцията. `Lambda` функциите могат да съдържат само един израз.

`Lambda` функциите често се използват в комбинация с методите `.map()`, `.applymap()` и `.apply()` в `Pandas`, тъй като те позволяват бързо и елегантно дефиниране на функции за трансформация на данни.

### 1. Примери с `.map()` и `lambda` функции (за `Series`):

```
import pandas as pd  
  
# Създаваме Series от числа  
numbers = pd.Series([1, 2, 3, 4, 5])  
print("Оригинален Series:\n", numbers)  
  
# Удвояване на всеки елемент с lambda функция
```

```

doubled_numbers = numbers.map(lambda x: x * 2)
print("\nУдвоени числа (lambda):\n", doubled_numbers)

# Проверка дали всяко число е четно с lambda функция
is_even = numbers.map(lambda x: x % 2 == 0)
print("\nЧетни числа (lambda):\n", is_even)

# Прилагане на условна логика с lambda функция
transformed_numbers = numbers.map(lambda x: x + 10 if x > 3 else x - 5)
print("\nТрансформирани числа (lambda с условие):\n",
transformed_numbers)

# Използване на lambda функция за работа с низове в Series
names = pd.Series(['alice', 'bob', 'charlie'])
upper_case_names = names.map(lambda name: name.upper())
print("\nИмена с главни букви (lambda):\n", upper_case_names)

```

Примерът демонстрира гъвкавостта на метода `.map()` при работа с Pandas Series и използването на lambda функции за различни елементни трансформации.

- 1) **Създава се series от числа:** Инициализира се Pandas Series с пет числови стойности (1, 2, 3, 4, 5). Оригиналният Series се извежда на конзолата.
- 2) **Удвояване на елементите (lambda функция):** Методът `.map()` се използва с lambda функция `lambda x: x * 2`, която умножава всеки елемент на numbers Series по 2. Резултатът `doubled_numbers` е нов Series с удвоените стойности, който се извежда на конзолата.
- 3) **Проверка за четни числа (lambda функция):** `.map()` се използва с lambda функция `lambda x: x % 2 == 0`, която проверява дали всеки елемент на numbers Series е четен (връща True ако е четен, False ако е нечетен). Резултатът `is_even` е булев Series, който се извежда на конзолата.
- 4) **Условна трансформация (lambda функция):** `.map()` се използва с lambda функция `lambda x: x + 10 if x > 3 else x - 5`. Тази функция прилага условна логика към всеки елемент на numbers Series: ако елементът е по-голям от 3, към него се добавя 10; в противен случай от него се изважда 5. Резултатът `transformed_numbers` е нов Series с трансформираните стойности, който се извежда на конзолата.
- 5) **Работа с низове (lambda функция):** Създава се нов Series `names` от низове ('alice', 'bob', 'charlie'). След това `.map()` се използва с lambda функция `lambda name: name.upper()`, която преобразува всеки низ в names Series в главни букви. Резултатът `upper_case_names` е нов Series с имената, написани с главни букви, който се извежда на конзолата.

## Накратко:

Примерът демонстрира как `.map()` в комбинация с `lambda` функции може да се използва за извършване на различни видове елементни операции върху `Series`, включително аритметични операции, логически проверки и манипулации с низове. `Lambda` функциите предоставят кратък и удобен начин за дефиниране на тези трансформации директно в извикването на `.map()`.

## 2. Примери с `.applymap()` и `lambda` функции (за `DataFrame`):

```
import pandas as pd

# Създаваме DataFrame с числови данни
data = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print("Оригинален DataFrame:\n", data)

# Увеличаване на всеки елемент с 1 с lambda функция
incremented_df = data.applymap(lambda x: x + 1)
print("\nУвеличени стойности (lambda):\n", incremented_df)

# Форматиране на всеки елемент като низ с lambda функция
formatted_df = data.applymap(lambda x: f"Value: {x}")
print("\nФорматирани стойности (lambda):\n", formatted_df)
```

Примерът демонстрира използването на метода `.applymap()` върху `Pandas DataFrame` в комбинация с `lambda` функции за прилагане на функции към всеки отделен елемент.

- 1) **Създава се `DataFrame` с числови данни:** Инициализира се `Pandas DataFrame` с две колони ('A' и 'B') и три реда, съдържащи числови стойности. Оригиналният `DataFrame` се извежда на конзолата.
- 2) **Увеличаване на всеки елемент с 1 (`lambda` функция):** Методът `.applymap()` се използва с `lambda` функция `lambda x: x + 1`. Тази функция се прилага към всеки елемент (всяка клетка) в `DataFrame`-а, като към всяка стойност се добавя 1. Резултатът `incremented_df` е нов `DataFrame` с увеличените стойности, който се извежда на конзолата.
- 3) **Форматиране на всеки елемент като низ (`lambda` функция):** Методът `.applymap()` се използва отново, този път с `lambda` функция `lambda x: f"Value: {x}"`. Тази функция приема всеки елемент (`x`) и го преобразува в низ, като го вмъква във форматиращ низ `"Value: {x}"`. Резултатът `formatted_df` е нов `DataFrame`, в който всяка оригинална числова стойност е представена като низ с префикса `"Value: "`. Този `DataFrame` се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.applymap()` позволява прилагането на кратки `lambda` функции към всеки елемент на `DataFrame` за извършване на елементни трансформации, като аритметични операции

или промяна на типа данни (в случая от числов в низ). `.applymap()` е удобен начин за бързо прилагане на една и съща функция към всички стойности в `DataFrame-a`.

### 3. Примери с `.apply()` и `lambda` функции (за `Series` и `DataFrame`):

```
import pandas as pd
import numpy as np

# Series: Изчисляване на дължината на всеки низ
cities = pd.Series(['София', 'Пловдив', 'Варна'])
city_lengths = cities.apply(lambda city: len(city))
print("\nДължина на градовете (lambda върху Series):\n", city_lengths)

# DataFrame (по колони): Изчисляване на средната стойност на всяка
колона
df = pd.DataFrame({'col1': [1, 2, 3], 'col2': [4, 5, np.nan]})
mean_cols = df.apply(lambda col: col.mean())
print("\nСредна стойност по колони (lambda върху DataFrame):\n",
mean_cols)

# DataFrame (по редове): Създаване на нов Series, показващ дали сумата
на реда е по-голяма от 5
row_sum_greater_than_5 = df.apply(lambda row: row.sum() > 5, axis=1)
print("\nСума на реда по-голяма от 5 (lambda върху DataFrame):\n",
row_sum_greater_than_5)

# DataFrame (по редове): Комбиниране на стойности от няколко колони в
низ
data_products = pd.DataFrame({
    'Име': ['Ябълка', 'Банан'],
    'Цена': [1.20, 0.80]
})
product_info = data_products.apply(lambda row: f"{row['Име']} - Цена:
{row['Цена']}", axis=1)
print("\nИнформация за продуктите (lambda върху DataFrame):\n",
product_info)
```

Примерът демонстрира гъвкавостта на метода `.apply()` при работа както със `Series`, така и с `DataFrame`, в комбинация с `lambda` функции за извършване на различни операции.

- 1) **Прилагане на `.apply()` към `Series` (изчисляване на дължина на низ):** Създава се `Series` `cities` от низове (имена на градове). Методът `.apply()` се използва с `lambda` функция `lambda city: len(city)`, която приема всеки низ от `Series`-а и връща неговата дължина. Резултатът `city_lengths` е нов `Series`, съдържащ дължината на всеки град, който се извежда на конзолата.
- 2) **Прилагане на `.apply()` към `DataFrame` по колони (изчисляване на средна стойност):** Създава се `DataFrame` `df` с две числови колони ('col1' и 'col2', като 'col2' съдържа една липсваща стойност). Методът `.apply()` се използва с `lambda` функция `lambda col: col.mean()`. По подразбиране (`axis=0`), функцията се прилага към всяка колона, като изчислява средната стойност на елементите в нея (липсващите стойности се игнорират). Резултатът `mean_cols` е `Series`, където индексът са имената на колоните, а стойностите са техните средни стойности, който се извежда на конзолата.
- 3) **Прилагане на `.apply()` към `DataFrame` по редове (проверка на сума):** Същият `DataFrame` `df` се използва отново. Методът `.apply()` се използва с `lambda` функция `lambda row: row.sum() > 5` и `axis=1`, което означава, че функцията се прилага към всеки ред. `Lambda` функцията изчислява сумата на елементите във всеки ред и връща `True`, ако сумата е по-голяма от 5, и `False` в противен случай. Резултатът `row_sum_greater_than_5` е `Series` от булеви стойности, който се извежда на конзолата.
- 4) **Прилагане на `.apply()` към `DataFrame` по редове (комбиниране на стойности в низ):** Създава се нов `DataFrame` `data_products` с колони 'Име' и 'Цена'. Методът `.apply()` се използва с `lambda` функция `lambda row: f"{row['Име']} - Цена: {row['Цена']}"` и `axis=1`. `Lambda` функцията приема всеки ред и форматира стойностите от колоните 'Име' и 'Цена' в един низ. Резултатът `product_info` е `Series` от низове, съдържащи информация за всеки продукт, който се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.apply()` в комбинация с `lambda` функции може да се използва за гъвкава обработка на данни както в `Series`, така и в `DataFrame`. Той показва как да се прилагат функции към цели `Series`, към колони на `DataFrame` (за обобщаващи статистики) и към редове на `DataFrame` (за създаване на нови стойности въз основа на няколко колони). `Lambda` функциите осигуряват кратък и удобен начин за дефиниране на тези операции.

## 4. Предимства на използването на `lambda` функции с `Pandas`:

- **Кратък и елегантен код:** `Lambda` функциите позволяват дефинирането на прости операции директно в извикването на `.map()`, `.applymap()` или `.apply()`, което прави кода по-компактен и по-лесен за четене за прости трансформации.
- **Удобство за еднократни операции:** Когато имате нужда от функция, която ще се използва само веднъж, `lambda` функцията е удобен начин да я дефинирате без да clutter-вате кода с излишни дефиниции на функции.
- **Гъвкавост:** `Lambda` функциите могат да приемат множество аргументи (в зависимост от това какво им подава `.map()`, `.applymap()` или `.apply()`) и да извършват различни операции.

Въпреки че `lambda` функциите са мощни за кратки операции, за по-сложна логика е препоръчително да дефинирате отделни, именувани функции, за да подобрите четимостта и поддръжката на кода.

## Казус 1: Категоризиране на клиенти според общата сума на поръчките (с `.map()` и `lambda`)

Представете си, че имате `DataFrame`, съдържащ информация за клиенти и общата сума на техните поръчки. Искате да категоризирате клиентите като "VIP" (ако общата сума е над 5000 лв.) и "Редовен" (в противен случай).

```
import pandas as pd

# Създаваме DataFrame с данни за клиенти и обща сума на поръчките
customer_data = {
    'клиент': ['Иван', 'Петър', 'Мария', 'Георги', 'Анна'],
    'обща_сума': [1200.50, 6500.80, 3400.20, 7800.90, 4900.10]
}
df_customers = pd.DataFrame(customer_data)

# Казус: Създайте нова колона 'категория', която съдържа 'VIP' за
клиенти с обща сума > 5000 и 'Редовен' за останалите.
```

### Решение на Казус 1:

```
# Използваме .map() с lambda функция за категоризиране
df_customers['категория'] = df_customers['обща_сума'].map(lambda x:
    'VIP' if x > 5000 else 'Редовен')

print("Данни за клиенти с категория:\n", df_customers)
```

В този казус, `lambda` функцията `lambda x: 'VIP' if x > 5000 else 'Редовен'` се прилага към всеки елемент от колоната 'обща\_сума'. Ако сумата е по-голяма от 5000, се връща 'VIP', в противен случай се връща 'Редовен'. Резултатът се присвоява на новата колона 'категория'.



## Казус 2: Изчисляване на процента на промяна на цените по групи продукти (с `.groupby()` и `.apply()` с `lambda`)

Представете си, че имате `DataFrame`, съдържащ месечни цени на различни продукти, които са групирани по категория. Искате да изчислите процента на промяна на цената за всеки продукт спрямо предходния месец в рамките на всяка категория.

```
import pandas as pd

# Създаваме DataFrame с месечни цени на продукти
price_history = {
    'категория': ['Електроника', 'Електроника', 'Храни', 'Храни',
                  'Дрехи', 'Дрехи'],
    'продукт': ['Телевизор', 'Лаптоп', 'Ябълки', 'Мляко', 'Риза',
                'Панталон'],
    'месец': ['Януари', 'Февруари', 'Януари', 'Февруари', 'Януари',
              'Февруари'],
    'цена': [500, 520, 2, 2.10, 30, 31.50]
}
df_prices = pd.DataFrame(price_history)

# Казус: Изчислете процента на промяна на цената за всеки продукт спрямо
предходния месец в рамките на всяка категория.
```

### Решение на Казус 2:

```
# Сортираме данните по категория и месец, за да осигурим правилния ред
за изчисление
df_prices = df_prices.sort_values(['категория', 'продукт', 'месец'])

# Групираме по категория и продукт и прилагаме lambda функция за
изчисляване на процента на промяна
df_prices['процент_промяна'] = df_prices.groupby(['категория',
                                                    'продукт'])['цена'].apply(
    lambda x: x.pct_change() * 100
)
```

```
print("История на цените с процент на промяна:\n", df_prices)
```

В този казус, първо сортираме данните, за да гарантираме, че цените са подредени по категория, продукт и месец. След това, групираме DataFrame-a по 'категория' и 'продукт'. В рамките на всяка група, *lambda* функцията *lambda x: x.pct\_change() \* 100* се прилага към Series-a от цени. *pct\_change()* изчислява процента на промяна между текущия и предходния елемент, а умножаването по 100 го превръща в процент. Резултатът се присвоява на новата колона 'процент\_промяна'.

Тези казуси илюстрират гъвкавостта и краткостта на *lambda* функциите при извършване на елементни трансформации или по-сложни изчисления в контекста на Pandas методи. Те са особено полезни за дефиниране на бързи, еднократни функции, които се използват директно в рамките на други операции.

## V. Създаване на нови колони с константни стойности

Един от най-простите начини за добавяне на нова колона към DataFrame е като ѝ присвоите константна стойност. Това ще създаде нова колона, в която всички редове ще имат тази една и съща стойност.

### Синтаксис:

```
df['Име_на_нова_колона'] = константна_стойност
```

Нека разгледаме няколко примера:

#### 1. Добавяне на колона с една и съща стойност за всички редове

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'Име': ['Алиса', 'Боб', 'Чарли'],
        'Възраст': [25, 30, 22]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Добавяме нова колона 'Държава' с константна стойност 'България'
df['Държава'] = 'България'
print("\nDataFrame с добавена колона 'Държава':\n", df)
```

```
# Добавяме друга колона 'Статус' с константна стойност 'Активен'
df['Статус'] = 'Активен'
print("\nDataFrame с добавена колона 'Статус':\n", df)
```

Примерът демонстрира най-основния начин за създаване на нови колони в Pandas DataFrame - чрез присвояване на константна стойност.

- 1) **Създава се примерен DataFrame:** Инициализира се Pandas DataFrame df с две колони: 'Име' (съдържаща имена) и 'Възраст' (съдържаща възрасти). Оригиналният DataFrame се извежда на конзолата.
- 2) **Добавяне на колона 'Държава' с константна стойност:** Създава се нова колона с име 'Държава' в DataFrame-a df. На тази нова колона се присвоява константната низова стойност 'България'. В резултат на това, всеки ред в DataFrame-a получава стойност 'България' в колоната 'Държава'. Обновеният DataFrame се извежда на конзолата.
- 3) **Добавяне на колона 'Статус' с константна стойност:** Създава се още една нова колона с име 'Статус' в DataFrame-a df. На тази колона се присвоява константната низова стойност 'Активен'. Подобно на предходната стъпка, всеки ред в DataFrame-a получава стойност 'Активен' в колоната 'Статус'. Окончателният DataFrame с двете добавени колони се извежда на конзолата.

## Накратко:

Примерът показва как лесно може да се добави нова колона към съществуващ DataFrame, като се използва синтаксисът за присвояване (`df['Име_на_нова_колона'] = стойност`). Когато се присвоява константна стойност, тя се разпространява във всички редове на новата колона. Този метод е полезен за добавяне на статична информация или етикети към целия набор от данни.

## 2. Добавяне на колона с булева стойност

```
# Добавяме колона 'Член' с константна стойност True
df['Член'] = True
print("\nDataFrame с добавена булева колона 'Член':\n", df)
```

Примерът демонстрира как да се добави нова колона с булева (логическа) константна стойност към Pandas DataFrame.

1. **Използва се DataFrame df от предходни примери:** Предполага се, че променливата df съдържа DataFrame с колони 'Име', 'Възраст' и 'Държава', както беше създаден в предишни примери.
2. **Добавяне на булева колона 'Член':** Създава се нова колона с име 'Член' в DataFrame-a df. На тази нова колона се присвоява константната булева стойност True. В резултат на това, всеки ред в DataFrame-a получава стойност True в колоната 'Член'.
3. **Извежда се обновеният DataFrame:** DataFrame df с добавената нова булева колона 'Член' се извежда на конзолата.

## Накратко:

Примерът показва, че добавянето на колона с константна стойност може да се използва и за булеви стойности. Това е полезно за добавяне на флагове или индикатори, които са еднакви за всички записи в даден момент.

### 3. Добавяне на колона с числова стойност

```
# Добавяме колона 'Идентификатор' с константна стойност 1
df['Идентификатор'] = 1
print("\nDataFrame с добавена числова колона 'Идентификатор':\n", df)
```

Примерът демонстрира как да се добави нова колона с числова константна стойност към Pandas DataFrame.

1. **Използва се DataFrame df от предходни примери:** Предполага се, че променливата df съдържа DataFrame с колони 'Име', 'Възраст', 'Държава' и 'Член', както беше създаден в предишни примери.
2. **Добавяне на числова колона 'Идентификатор':** Създава се нова колона с име 'Идентификатор' в DataFrame-a df. На тази нова колона се присвоява константната числова стойност 1. В резултат на това, всеки ред в DataFrame-a получава стойност 1 в колоната 'Идентификатор'.
3. **Извежда се обновеният DataFrame:** DataFrame df с добавената нова числова колона 'Идентификатор' се извежда на конзолата.

## Накратко:

Примерът показва, че добавянето на колона с константна стойност може да се използва и за числови стойности. Това е често срещано при добавяне на поредици, версии или други статични числови идентификатори към набора от данни.

### 4. Важно:

- Ако колона с това име вече съществува, тази операция ще **презапише** съществуващите стойности.
- Типът на данните на новата колона ще бъде определен от типа на константната стойност (например, низ за низове, булев за True/False, числов за числа).
- Можете да добавяте колони по този начин по всяко време след създаването на DataFrame.

Добавянето на колони с константни стойности е често използвано, например, за добавяне на времеви маркери, етикети или статична информация към набора от данни.

## VI. Създаване на колони на базата на съществуващи колони (чрез операции и функции)

Една от най-мощните възможности на Pandas е създаването на нови колони, чиито стойности се изчисляват или трансформират въз основа на данните в други колони. Това може да се постигне чрез:

### 1. Векторизирани операции:

Както вече разгледахме, можем да прилагаме аритметични, сравнителни и логически операции директно върху колони (като Series обекти) на DataFrame. Резултатът от тези операции може да бъде присвоен на нова колона.

#### а) Аритметични операции между колони

```
import pandas as pd

# Създаваме DataFrame с данни за продажби
data = {'Цена': [1.20, 0.80, 1.00],
        'Количество': [100, 150, 80]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame за продажби:\n", df_sales)

# Създаваме нова колона 'Обща_стойност', като умножаваме 'Цена' и
'Количество'
df_sales['Обща_стойност'] = df_sales['Цена'] * df_sales['Количество']
print("\nDataFrame с добавена колона 'Обща_стойност':\n", df_sales)

# Създаваме колона 'Средна_цена_на_единица' (в този случай е същата като
'Цена')
df_sales['Средна_цена_на_единица'] = df_sales['Обща_стойност'] /
df_sales['Количество']
print("\nDataFrame с добавена колона 'Средна_цена_на_единица':\n",
df_sales)
```

Примерът демонстрира как да се създават нови колони в Pandas DataFrame, като се използват векторизирани аритметични операции върху съществуващи колони.

1. **Създава се DataFrame с данни за продажби:** Инициализира се Pandas DataFrame df\_sales с две колони: 'Цена' (съдържаща цени на продукти) и 'Количество' (съдържаща продадени количества). Оригиналният DataFrame се извежда на конзолата.

2. **Създаване на колона 'Обща\_стойност':** Създава се нова колона с име 'Обща\_стойност'. Стойностите в тази колона се изчисляват чрез векторизирано умножение на стойностите от колона 'Цена' и колона 'Количество' за всеки съответен ред. Резултатът от това умножение се присвоява на новата колона. Обновеният DataFrame с добавената колона 'Обща\_стойност' се извежда на конзолата.
3. **Създаване на колона 'Средна\_цена\_на\_единица':** Създава се още една нова колона с име 'Средна\_цена\_на\_единица'. Стойностите в тази колона се изчисляват чрез векторизирано деление на стойностите от колона 'Обща\_стойност' на стойностите от колона 'Количество' за всеки съответен ред. Резултатът от това деление се присвоява на новата колона. В този конкретен случай, тъй като 'Обща\_стойност' е 'Цена' \* 'Количество', а след това се дели на 'Количество', стойностите в 'Средна\_цена\_на\_единица' ще бъдат същите като в колона 'Цена' (ако 'Количество' не е нула). Окончателният DataFrame с двете добавени колони се извежда на конзолата.

## Накратко:

Примерът илюстрира как лесно може да се създават нови колони в DataFrame, като се извършват аритметични операции между съществуващи колони. Pandas извършва тези операции елемент по елемент по ефективен, векторизиран начин. Резултатите от тези операции могат директно да бъдат присвоени на нови колони.

### б) Сравнителни операции за създаване на булеви колони

```
# Създаваме колона 'Голяма_поръчка', която е True, ако 'Количество' е над 90
df_sales['Голяма_поръчка'] = df_sales['Количество'] > 90
print("\nDataFrame с добавена булева колона 'Голяма_поръчка':\n",
df_sales)
```

Примерът демонстрира как да се създаде нова булева колона ('Голяма\_поръчка') в Pandas DataFrame въз основа на сравнение на стойностите в съществуваща колона ('Количество') с определена стойност (90).

1. **Използва се DataFrame df\_sales от предходни примери:** Предполага се, че df\_sales съдържа колона 'Количество', както беше създаден в предишни примери (със данни за продажби).
2. **Създаване на булева колона с векторизирана сравнителна операция:** Създава се нова колона с име 'Голяма\_поръчка'. Стойностите в тази колона се определят от резултата на векторизирана операция за сравнение: df\_sales['Количество'] > 90. Тази операция се прилага към всеки елемент в колоната 'Количество' и връща True, ако стойността е по-голяма от 90, и False в противен случай. Резултатът от тази сравнителна операция (булев Series) се присвоява като стойности на новата колона 'Голяма\_поръчка' в df\_sales.
3. **Извежда се обновеният DataFrame:** DataFrame df\_sales с добавената нова булева колона 'Голяма\_поръчка' се извежда на конзолата, показвайки за всяка транзакция дали количеството е било над 90.

## Накратко:

Примерът илюстрира как векторизираните сравнителни операции могат да бъдат използвани за лесно създаване на нови булеви колони в `DataFrame`. Тези булеви колони могат да бъдат много полезни за филтриране на данни, условно присвояване на стойности или за други логически анализи.

## 2. Прилагане на функции (`.map()`, `.apply()`, `.applymap()`):

Можем да използваме тези методи, за да приложим по-сложна логика или външни функции към една или няколко колони и да присвоим резултата на нова колона.

### а) Използване на `.map()` за трансформация на стойности в колона

```
# Създаваме DataFrame с колона от имена
df_names = pd.DataFrame({'Име': ['алиса', 'боб', 'чарли']})
print("\nОригинален DataFrame с имена:\n", df_names)

# Създаваме нова колона 'Име_Капитализирано', като капитализираме всяко
име
df_names['Име_Капитализирано'] = df_names['Име'].map(str.capitalize)
print("\nDataFrame с добавена колона 'Име_Капитализирано':\n", df_names)
```

Примерът демонстрира как да се създаде нова колона ('Име\_Капитализирано') в `Pandas DataFrame`, като се приложи функцията (в случая `str.capitalize()`) към съществуваща колона ('Име') с помощта на метода `.map()`.

1. **Създава се `DataFrame` с колона от имена:** Инициализира се `Pandas DataFrame` `df_names` с една колона 'Име', съдържаща имена, написани с малки букви. Оригиналният `DataFrame` се извежда на конзолата.
2. **Създаване на нова колона с `.map()`:** Създава се нова колона с име 'Име\_Капитализирано'. Стойностите в тази колона се получават, като методът `.map()` се прилага към колоната `df_names['Име']`. Като аргумент на `.map()` се подава вградената Python функция за низове `str.capitalize`. Тази функция се прилага към всеки елемент (всяко име) в колоната 'Име', като капитализира първата буква на всяко име. Резултатът от `.map()` е нов `Series` с капитализираните имена, който се присвоява като стойности на новата колона 'Име\_Капитализирано' в `df_names`.
3. **Извежда се обновеният `DataFrame`:** `DataFrame` `df_names` с добавената нова колона 'Име\_Капитализирано' се извежда на конзолата, показвайки оригиналните имена и техните капитализирани версии.

## Накратко:

Примерът показва как `.map()` може да се използва за прилагане на функция, която трансформира всяка стойност в дадена колона на `DataFrame`. В този случай, той се използва за капитализиране на низови стойности и добавянето им като нова колона към `DataFrame`-а. Този метод е полезен за прилагане на елементни трансформации, които не могат лесно да бъдат извършени с прости векторизирани операции.



б) Използване на `.apply()` за създаване на колона на базата на няколко колони (по редове)

```
# Използваме DataFrame df_sales от предходния пример
print("\nDataFrame за продажби:\n", df_sales)

# Създаваме колона 'Статус_на_поръчка', която зависи от 'Количество' и
'Обща_стойност'
def categorize_order(row):
    if row['Количество'] > 100 and row['Обща_стойност'] > 100:
        return 'Голяма и скъпа'
    elif row['Количество'] > 100:
        return 'Голяма'
    elif row['Обща_стойност'] > 100:
        return 'Скъпа'
    else:
        return 'Нормална'

df_sales['Статус_на_поръчка'] = df_sales.apply(categorize_order, axis=1)
print("\nDataFrame с добавена колона 'Статус_на_поръчка':\n", df_sales)
```

Примерът демонстрира как да се създаде нова колона в Pandas DataFrame ('Статус\_на\_поръчка'), чиито стойности се определят въз основа на условия, включващи стойности от други колони ('Количество' и 'Обща\_стойност'), като се използва методът `.apply()` с `axis=1` (прилагане по редове).

1. **Извежда се съществуващият DataFrame df\_sales:** Първо, се извежда DataFrame `df_sales` от предходния пример, който съдържа колони 'Цена', 'Количество' и 'Обща\_стойност'.
2. **Дефинира се функция за категоризиране на поръчка:** Създава се функция с име `categorize_order`, която приема един аргумент `row`, представляващ всеки ред от DataFrame-a като Series обект. Вътре във функцията се извършват поредица от условни проверки (`if`, `elif`, `else`) въз основа на стойностите в колоните 'Количество' и 'Обща\_стойност' на текущия ред. В зависимост от изпълненото условие, функцията връща един от четирите низови статуса: 'Голяма и скъпа', 'Голяма', 'Скъпа' или 'Нормална'.
3. **Прилагане на функцията по редове и създаване на нова колона:** Методът `.apply()` се използва върху `df_sales` с аргумент `axis=1`, което указва, че функцията `categorize_order` трябва да бъде приложена към всеки ред на DataFrame-a. Резултатът от прилагането на функцията към всеки ред (върнатият статус) се присвоява на нова колона с име 'Статус\_на\_поръчка'.

4. **Извежда се обновеният DataFrame:** Накрая, се извежда DataFrame `df_sales` с добавената нова колона 'Статус\_на\_поръчка', която съдържа статуса на всяка поръчка, определен от логиката във функцията `categorize_order`.

## Накратко:

Примерът илюстрира как `.apply(func, axis=1)` позволява да се извършват сложни, условни преобразувания, които зависят от стойностите в няколко колони на всеки ред. Резултатът от тези преобразувания може да бъде използван за създаване на нови, информативни колони в DataFrame-a.

### 3. Важно:

- Когато създавате нова колона, като използвате съществуващи, Pandas автоматично подравнява данните по индекса.
- Можете да използвате верижно присвояване за създаване на множество нови колони наведнъж (въпреки че `.assign()` е по-препоръчителен за това, както ще видим по-късно).

**Създаването на нови колони на базата на съществуващи е основна операция при подготовката и трансформацията на данни за анализ. Тя позволява да извличате нова информация, да извършвате сложни преобразувания и да подготвяте данните в желания формат.**

## VII. Създаване на нови колони чрез метода `.assign()`.

Методът `.assign()` е удобен и често предпочитан начин за създаване на нови колони в DataFrame. Той връща **нов** DataFrame с добавените или модифицираните колони, като оригиналният DataFrame остава непроменен (освен ако не презапишете променливата). `.assign()` е особено полезен при верижни операции (`.pipe()`), тъй като позволява създаването на множество нови колони в един последователен стил.

### Синтаксис:

```
df.assign(име_на_нова_колона=стойност_или_функция,  
          друго_име=друга_стойност_или_функция,  
          ...)
```

- **име\_на\_нова\_колона:** Името на колоната, която искате да създадете (като низ).
- **стойност\_или\_функция:** Това може да бъде:
  - Скаларна стойност (ще бъде присвоена на всички редове в новата колона).
  - Series (индексът ѝ ще бъде подравнен с индекса на DataFrame).

- Функция, която приема DataFrame като единствен аргумент. Резултатът от тази функция трябва да бъде Series или NumPy array, чиято дължина съвпада с броя на редовете на DataFrame.

Нека разгледаме няколко примера:

## 1. Добавяне на колона с константна стойност

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'Име': ['Алиса', 'Боб', 'Чарли'],
        'Възраст': [25, 30, 22]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Добавяме нова колона 'Държава' с константна стойност
df_assigned = df.assign(Държава='България')
print("\nDataFrame след използване на .assign() за добавяне на
'Държава':\n", df_assigned)
```

Примерът демонстрира основното използване на метода `.assign()` за добавяне на нова колона с константна стойност към Pandas DataFrame.

1. **Създава се примерен DataFrame:** Инициализира се Pandas DataFrame `df` с две колони: 'Име' (съдържаща имена) и 'Възраст' (съдържаща възрасти). Оригиналният DataFrame се извежда на конзолата.
2. **Добавяне на колона 'Държава' с `.assign()`:** Методът `.assign()` се използва върху DataFrame-a `df`. Като аргумент на `.assign()` се подава ключова дума Държава, на която се присвоява константната стойност 'България'. Методът `.assign()` връща **нов** DataFrame (в този случай присвоен на променливата `df_assigned`) с добавената колона 'Държава', където всички редове имат стойност 'България'. Оригиналният DataFrame `df` остава непроменен. Новият DataFrame `df_assigned` се извежда на конзолата.

## Накратко:

Примерът показва как `.assign()` предоставя елегантен начин за добавяне на нови колони към DataFrame. Той връща нов DataFrame с добавените колони, което е полезно за верижни операции и избягване на нежелани модификации на оригиналния DataFrame. В този случай, добавената колона 'Държава' има една и съща константна стойност за всички редове.

## 2. Добавяне на колона на базата на съществуваща колона

```
# Използваме DataFrame df от предходния пример
df_assigned = df.assign(Години_след_18=df['Възраст'] - 18)
print("\nDataFrame след използване на .assign() за добавяне на
'Години_след_18':\n", df_assigned)
```

Примерът демонстрира как да се създаде нова колона в Pandas DataFrame ('Години\_след\_18') с помощта на метода `.assign()`, като стойностите в новата колона се изчисляват въз основа на съществуваща колона ('Възраст').

- 1) **Използва се DataFrame df от предходния пример:** Предполага се, че променливата df съдържа DataFrame с колони 'Име' и 'Възраст', както беше създаден в предишния пример.
- 2) **Добавяне на колона 'Години\_след\_18' с `.assign()`:** Методът `.assign()` се използва върху DataFrame-a df. Като аргумент се подава ключова дума `Години_след_18`, а стойността, която се присвоява на тази нова колона, е резултатът от векторизираната операция `df['Възраст'] - 18`. Това означава, че за всеки ред от DataFrame-a се изчислява разликата между стойността в колона 'Възраст' и числото 18. `.assign()` връща **нов** DataFrame (в този случай присвоен на променливата `df_assigned`) с добавената колона 'Години\_след\_18', съдържаща тези изчислени разлики. Оригиналният DataFrame df остава непроменен. Новият DataFrame `df_assigned` се извежда на конзолата.

### Накратко:

Примерът показва как `.assign()` може да се използва за създаване на нови колони, чиито стойности се базират на изчисления, включващи стойности от други колони. Това е чист и четим начин за добавяне на производни характеристики към DataFrame-a.

## 3. Добавяне на множество колони наведнъж

```
df_assigned_multiple = df.assign(
    Държава='България',
    Статус='Активен',
    Години_до_пенсия=lambda x: 65 - x['Възраст']
)
print("\nDataFrame след използване на .assign() за добавяне на няколко
колони:\n", df_assigned_multiple)
```

Примерът демонстрира как методът `.assign()` може да бъде използван за добавяне на **няколко** нови колони към Pandas DataFrame едновременно.

1. **Използва се DataFrame df от предходните примери:** Предполага се, че променливата df съдържа DataFrame с колони 'Име' и 'Възраст'.
2. **Добавяне на множество колони с .assign():** Методът .assign() се използва върху DataFrame-a df. Като аргументи се подават няколко ключови думи, всяка от които представлява име на нова колона и стойността, която ще бъде присвоена на тази колона:
  - Държава='България': Създава нова колона 'Държава' с константна стойност 'България' за всички редове.
  - Статус='Активен': Създава нова колона 'Статус' с константна стойност 'Активен' за всички редове.
  - Години\_до\_пенсия=lambda x: 65 - x['Възраст']: Създава нова колона 'Години\_до\_пенсия'. Стойностите в тази колона се изчисляват с помощта на lambda функция, която приема целия DataFrame (x) като аргумент и за всеки ред изчислява разликата между 65 и стойността в колона 'Възраст' (x['Възраст']).
3. **Извежда се обновеният DataFrame:** Методът .assign() връща нов DataFrame (в този случай присвоен на променливата df\_assigned\_multiple) с всички добавени колони. Оригиналният DataFrame df остава непроменен. Новият DataFrame с добавените колони 'Държава', 'Статус' и 'Години\_до\_пенсия' се извежда на конзолата.

## Накратко:

Примерът показва ефективността на .assign() за добавяне на множество нови колони в една операция. Той също така илюстрира как .assign() може да приема както константни стойности, така и функции (включително lambda функции, които могат да се базират на други колони в DataFrame-a) за определяне на стойностите в новите колони. Това прави .assign() мощен инструмент за трансформация и обогатяване на данни.

## 4. Забележки за използването на функция в .assign():

- Когато използвате функция, тя получава целия DataFrame като аргумент (обикновено се обозначава с df или x).
- В рамките на функцията можете да достъпвате колони на DataFrame-a като атрибути (напр., x.Възраст) или като елементи на речник (x['Възраст']).
- Резултатът от функцията трябва да бъде Series или NumPy array с дължина, съответстваща на броя на редовете на DataFrame.

## 5. Предимства на използването на .assign():

- **Верижност:** .assign() връща нов DataFrame, което го прави идеален за използване във верижни операции с .pipe(). Това позволява по-четим и организиран код при извършване на множество трансформации.
- **Яснота:** Синтаксисът на .assign() е ясен и лесен за разбиране при създаване на нови колони.
- **Избягване на странични ефекти:** Тъй като .assign() връща нов DataFrame, оригиналният обект остава непроменен, което може да помогне за избягване на нежелани странични ефекти в по-сложен код.

Въпреки че директното присвояване на нова колона (напр., df['Нова\_колона'] = ...) е по-кратко за единични операции, .assign() е често предпочитан, когато се създават множество нови колони или когато се работи във верига от операции.

## VIII. Условно създаване на колони (np.where, .loc)

Често се налага да създаваме нови колони, чиито стойности зависят от определени условия, базирани на данните в други колони. Pandas предлага няколко мощни начина за постигане на това:

### 1. Използване на `np.where`:

Функцията `np.where()` от библиотеката NumPy е много полезна за условно присвояване на стойности в Series или колони на DataFrame. Тя има следния основен синтаксис:

```
numpy.where(condition, x, y)
```

- `condition`: Булев масив (или Series), където `True` указва къде да се вземе стойност от `x`, а `False` - къде да се вземе стойност от `y`.
- `x`: Стойност (скаларна или масив/Series) за присвояване, когато условието е `True`.
- `y`: Стойност (скаларна или масив/Series) за присвояване, когато условието е `False`.

Резултатът от `np.where()` е NumPy array, който след това може да бъде присвоен на нова колона в DataFrame.

*Пример: Условно създаване на колона с `np.where`*

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с данни за продажби
data = {'Продукт': ['A', 'B', 'A', 'C', 'B'],
        'Продажби': [100, 150, 120, 90, 160]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame за продажби:\n", df_sales)

# Създаваме колона 'Категория_продажби' въз основа на 'Продажби'
df_sales['Категория_продажби'] = np.where(df_sales['Продажби'] >= 120,
    'Високи', 'Ниски')
print("\nDataFrame с добавена колона 'Категория_продажби':\n", df_sales)

# Условно присвояване на числови стойности
df_sales['Бонус'] = np.where(df_sales['Продажби'] >= 120, 50, 0)
```

```
print("\nDataFrame с добавена колона 'Бонус':\n", df_sales)
```

- 1) **Създава се DataFrame с данни за продажби:** Инициализира се Pandas DataFrame `df_sales` с две колони: 'Продукт' (съдържаща продуктови кодове) и 'Продажби' (съдържаща стойности на продажби). Оригиналният DataFrame се извежда на конзолата.
- 2) **Създаване на колона 'Категория\_продажби' с `np.where`:** Създава се нова колона с име 'Категория\_продажби'. Стойностите в тази колона се определят условно въз основа на стойностите в колона 'Продажби'. Функцията `np.where()` проверява за всеки ред дали стойността в колона 'Продажби' е по-голяма или равна на 120. Ако условието е `True`, на съответния ред в колона 'Категория\_продажби' се присвоява стойност 'Високи'; в противен случай се присвоява стойност 'Ниски'. Обновеният DataFrame с добавената колона се извежда на конзолата.
- 3) **Създаване на колона 'Бонус' с `np.where` (числови стойности):** Създава се още една нова колона с име 'Бонус'. По същия начин, функцията `np.where()` се използва за условно присвояване на числови стойности. Ако стойността в колона 'Продажби' е по-голяма или равна на 120, на съответния ред в колона 'Бонус' се присвоява стойност 50; в противен случай се присвоява стойност 0. Обновеният DataFrame с добавената колона 'Бонус' се извежда на конзолата.

## Накратко:

Примерът илюстрира как `np.where()` позволява създаването на нови колони в DataFrame, като стойностите в тях се определят въз основа на булево условие, приложено към друга колона. Той показва как условно могат да се присвояват както низови, така и числови стойности. `np.where()` е мощен инструмент за бързо и ефективно условно модифициране или създаване на колони в Pandas.

Можем да използваме и вложени `np.where` за по-сложни условия:

```
# Създаваме по-детайлна категоризация
df_sales['Детайлна_категория'] = np.where(df_sales['Продажби'] >= 150,
    'Отлични',

    np.where(df_sales['Продажби'] >= 120, 'Добри', 'Средни'))
print("\nDataFrame с добавена колона 'Детайлна_категория':\n", df_sales)
```

- 1) **Използва се DataFrame `df_sales` от предходния пример:** Предполага се, че `df_sales` съдържа колона 'Продажби'.
- 2) **Създаване на колона 'Детайлна\_категория' с вложени `np.where()`:** Създава се нова колона с име 'Детайлна\_категория'. Стойностите в тази колона се определят въз основа на две нива на условна проверка, използвайки вложени извиквания на `np.where()`:
  - Първото ниво `np.where(df_sales['Продажби'] >= 150, 'Отлични', ...)` проверява дали стойността в колона 'Продажби' е по-голяма или равна на 150. Ако е `True`, на съответния ред в колона 'Детайлна\_категория' се присвоява стойност 'Отлични'. Ако е `False`, се изпълнява второто (вътрешно) `np.where()` условие.



- Второто ниво `np.where(df_sales['Продажби'] >= 120, 'Добри', 'Средни')` проверява дали стойността в колона 'Продажби' е по-голяма или равна на 120 (но по-малка от 150, тъй като първото условие е било `False`). Ако е `True`, се присвоява стойност 'Добри'; в противен случай (ако е по-малка от 120), се присвоява стойност 'Средни'.
- 3) **Извежда се обновеният DataFrame:** `DataFrame df_sales` с добавената нова колона 'Детайлна\_категория', съдържаща по-детайлната категоризация на продажбите, се извежда на конзолата.

## Накратко:

Примерът илюстрира как могат да се използват вложени `np.where()` функции за реализиране на по-сложна условна логика при създаване на нови колонии в `DataFrame`. Това позволява категоризиране на данни в повече от две групи въз основа на стойностите в други колонии. Въпреки че е ефективно, при много сложни условия може да стане трудно за четене и поддръжка, като в такива случаи `.loc` с множество условия може да бъде по-предпочитан.

## 2. Използване на `.loc` за условно присвояване:

Аксесорът `.loc` се използва главно за селекция на данни по етикети на редове и колонии. Той също така може да се използва за условно присвояване на стойности на подмножество от редове в дадена колона. Синтаксисът за условно присвояване с `.loc` е:

```
df.loc[condition, 'име_на_нова_колона'] = стойност
```

- `condition`: Булев `Series` или масив, който указва кои редове да бъдат засегнати.
- `'име_на_нова_колона'`: Името на колоната, на която присвояваме стойност (ако колоната не съществува, тя ще бъде създадена).
- `стойност`: Стойността, която ще бъде присвоена на редовете, където условието е `True`.

Можем да използваме `.loc` няколко пъти за различни условия и стойности.

### Пример: Условно създаване на колона с `.loc`

```
import pandas as pd

# Създаваме DataFrame с данни за клиенти
data_clients = {'Име': ['Иван', 'Петър', 'Мария', 'Анна'],
                 'Пол': ['м', 'м', 'ж', 'ж'],
                 'Възраст': [28, 35, 22, 40]}
df_clients = pd.DataFrame(data_clients)
print("Оригинален DataFrame за клиенти:\n", df_clients)

# Създаваме колона 'Поздрав' и присвояваме стойности условно
```

```

df_clients['Поздрав'] = '' # Първо създаваме колоната с празни низове
(или np.nan)
df_clients.loc[df_clients['Пол'] == 'м', 'Поздрав'] = 'Г-н'
df_clients.loc[df_clients['Пол'] == 'ж', 'Поздрав'] = 'Г-жа'
print("\nDataFrame с добавена колона 'Поздрав':\n", df_clients)

# Условно присвояване въз основа на множество условия
df_clients['Възрастова_група'] = 'Млад' # Задаваме стойност по
подразбиране
df_clients.loc[df_clients['Възраст'] >= 30, 'Възрастова_група'] = 'Зрял'
df_clients.loc[df_clients['Възраст'] >= 40, 'Възрастова_група'] =
'Възрастен'
print("\nDataFrame с добавена колона 'Възрастова_група':\n", df_clients)

```

- 1) **Създава се DataFrame с данни за клиенти:** Инициализира се Pandas DataFrame `df_clients` с три колони: 'Име', 'Пол' и 'Възраст', съдържащи информация за клиенти. Оригиналният DataFrame се извежда на конзолата.
- 2) **Създаване на колона 'Поздрав' с условно присвояване:**
  - Първо, към DataFrame-а се добавя нова колона с име 'Поздрав' и ѝ се присвоява празен низ (") като стойност по подразбиране за всички редове.
  - След това се използва `.loc` за условно присвояване на стойности в колона 'Поздрав' въз основа на стойностите в колона 'Пол'.
    - `df_clients.loc[df_clients['Пол'] == 'м', 'Поздрав'] = 'Г-н'`: За всички редове, където стойността в колона 'Пол' е 'м', стойността в колона 'Поздрав' се променя на 'Г-н'.
    - `df_clients.loc[df_clients['Пол'] == 'ж', 'Поздрав'] = 'Г-жа'`: За всички редове, където стойността в колона 'Пол' е 'ж', стойността в колона 'Поздрав' се променя на 'Г-жа'.
  - Обновеният DataFrame с добавената колона 'Поздрав' се извежда на конзолата.
- 3) **Създаване на колона 'Възрастова\_група' с условно присвояване (множество условия):**
  - Към DataFrame-а се добавя нова колона с име 'Възрастова\_група' и ѝ се присвоява стойност 'Млад' като стойност по подразбиране за всички редове.
  - След това `.loc` се използва многократно за условно присвояване на различни възрастови групи въз основа на стойността в колона 'Възраст':
    - `df_clients.loc[df_clients['Възраст'] >= 30, 'Възрастова_група'] = 'Зрял'`: За всички редове, където стойността в колона 'Възраст' е по-голяма или равна на 30, стойността в колона 'Възрастова\_група' се променя на 'Зрял'.
    - `df_clients.loc[df_clients['Възраст'] >= 40, 'Възрастова_група'] = 'Възрастен'`: За всички редове, където стойността в колона 'Възраст' е по-голяма или равна на 40, стойността в колона 'Възрастова\_група' се променя на 'Възрастен'. (Забележете, че ред, който вече е 'Зрял' и отговаря на това условие, ще бъде презаписан като 'Възрастен').
  - Обновеният DataFrame с добавената колона 'Възрастова\_група' се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.loc` може да се използва за условно присвояване на стойности в нови или съществуващи колони на `DataFrame` въз основа на булеви условия, приложени към други колони. Той показва как да се обработват прости и по-сложни сценарии с множество условия за категоризиране на данни. `.loc` е мощен и често използван инструмент за селективно модифициране на данни в `Pandas`.

### 3. Сравнение между `np.where` и `.loc` за условно създаване на колони:

- `np.where` е по-компактен за прости условни присвоявания с две възможни стойности. Той връща NumPy array, който трябва да бъде присвоен на колона.
- `.loc` е по-гъвкав, когато имате множество различни условия и стойности за присвояване, особено когато искате да модифицирате стойности в съществуваща или нова колона въз основа на сложни булеви условия. Той позволява последователно прилагане на различни условия.
- За по-сложни условни логики, включващи множество клонове, често `.loc` е по-четим и по-лесен за поддръжка.
- `np.select()` е друга NumPy функция, която е полезна за условно присвояване с множество условия и съответстващи стойности.

Изборът между `np.where` и `.loc` зависи от конкретния случай и сложността на условната логика, която искате да приложите. И двата метода са мощни инструменти за условно манипулиране на данни в `Pandas`.

### Казус 1: Добавяне на информация за валута и статус на обработка (константна стойност и условно)

Представете си, че имате `DataFrame`, съдържащ данни за поръчки, включващи сума. Искате да добавите колона, указваща валутата (например, 'лв.') за всички поръчки, и колона, указваща дали поръчката е "Обработена" или "Необработена" въз основа на стойността на сумата (например, ако сумата е над 100 лв., е "Обработена").

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с данни за поръчки
orders_data = {
    'номер_поръчка': [1, 2, 3, 4, 5],
    'сума': [50.00, 120.00, 85.50, 200.00, 30.00]
}
df_orders = pd.DataFrame(orders_data)
```

```
# Казус 1.1: Добавете колона 'валута' с константна стойност 'лв.'.
# Казус 1.2: Създайте колона 'статус_обработка', която е 'Обработена',
ако сумата е > 100, и 'Необработена' в противен случай (използвайте
np.where).
```

## Решение на Казус 1:

```
# 1.1: Добавяне на колона с константна стойност
df_orders['валута'] = 'лв.'
print("DataFrame с добавена валута:\n", df_orders)

# 1.2: Условно създаване на колона с np.where
df_orders['статус_обработка'] = np.where(df_orders['сума'] > 100,
'Обработена', 'Необработена')
print("\nDataFrame със статус на обработка:\n", df_orders)
```

## Казус 2: Изчисляване на обща цена с отстъпка и създаване на флаг за голяма поръчка (на база съществуващи колони и .loc)

Представете си, че имате DataFrame с информация за поръчани продукти, включваща количество и единична цена. Искате да изчислите общата цена на всяка поръчка и да приложите отстъпка от 10% за поръчки с обща стойност над 200 лв. Също така искате да създадете булева колона, указваща дали поръчката е "голяма" (количество > 5).

```
import pandas as pd

# Създаваме DataFrame с данни за поръчани продукти
items_data = {
    'продукт': ['А', 'Б', 'В', 'Г', 'Д'],
    'количество': [2, 7, 3, 10, 5],
    'единична_цена': [25.00, 15.50, 50.00, 10.00, 30.00]
}
df_items = pd.DataFrame(items_data)

# Казус 2.1: Създайте колона 'обща_цена' като произведение на
'количество' и 'единична_цена'.
```

```
# Казус 2.2: Създайте колона 'цена_след_отстъпка', която е 'обща_цена'
намалена с 10%, ако 'обща_цена' > 200, и 'обща_цена' в противен случай
(.loc).

# Казус 2.3: Създайте булева колона 'голяма_поръчка', която е True, ако
'количество' > 5, и False в противен случай.
```

## Решение на Казус 2:

```
# 2.1: Създаване на колона на база съществуващи колони (векторизирана
операция)
df_items['обща_цена'] = df_items['количество'] *
df_items['единична_цена']
print("DataFrame с обща цена:\n", df_items)

# 2.2: Условно създаване на колона с .loc
df_items['цена_след_отстъпка'] = df_items['обща_цена'].copy() #
Създаваме копие, за да избегнем SettingWithCopyWarning
discount_condition = df_items['обща_цена'] > 200
df_items.loc[discount_condition, 'цена_след_отстъпка'] =
df_items.loc[discount_condition, 'обща_цена'] * 0.9
print("\nDataFrame с цена след отстъпка:\n", df_items)

# 2.3: Създаване на булева колона на база съществуваща колона
(векторизирана операция за сравнение)
df_items['голяма_поръчка'] = df_items['количество'] > 5
print("\nDataFrame с флаг за голяма поръчка:\n", df_items)
```

## Казус 3: Създаване на няколко нови колони с .assign()

Представете си, че имате DataFrame с информация за служители, включваща месечна заплата и брой отработени часове. Искате да изчислите годишната заплата и часовата ставка на всеки служител, използвайки метода .assign().

```
import pandas as pd

# Създаваме DataFrame с данни за служители
```

```
employees_data = {
    'служител': ['Иван', 'Петър', 'Мария'],
    'месечна_заплата': [1500, 2200, 1800],
    'отработени_часове': [160, 170, 150]
}
df_employees = pd.DataFrame(employees_data)

# Казус: Използвайте .assign() за да добавите колони 'годишна_заплата'
(месечна_заплата * 12) и 'часова_ставка' (месечна_заплата / отработени
часове).
```

### Решение на Казус 3:

```
# Използване на .assign() за създаване на няколко нови колони
df_employees = df_employees.assign(
    годишна_заплата=df_employees['месечна_заплата'] * 12,
    часова_ставка=df_employees['месечна_заплата'] /
df_employees['отработени_часове']
)

print("DataFrame с годишна заплата и часова ставка:\n", df_employees)
```

Тези казуси илюстрират различните и гъвкави начини за създаване на нови колони в Pandas DataFrame в зависимост от изискванията: с константни стойности, на базата на съществуващи колони чрез векторизирани операции и функции, използвайки елегантния метод .assign() за създаване на множество колони едновременно, и условно създаване на колони с помощта на np.where и .loc за по-сложни логически условия.

## IX. Преименуване на колони и индекси (.rename(), .rename\_axis(), .set\_axis())

Често при работа с данни се налага да преименуваме колони или етикети на индексите, за да ги направим по-ясни, по-лесни за използване или съвместими с други системи. Pandas предлага няколко метода за тази цел.

## 1. Преименуване на колони и индекси с `.rename()`:

Методът `.rename()` е гъвкав и позволява преименуване както на колони, така и на етикети на индекси. Той приема речник, в който ключовете са старите имена, а стойностите са новите имена.

### Синтаксис:

```
df.rename(columns={'старо_име_кол1': 'ново_име_кол1', 'старо_име_кол2':  
'ново_име_кол2', ...},  
          index={'стар_етикет_ред1': 'нов_етикет_ред1',  
'стар_етикет_ред2': 'нов_етикет_ред2', ...},  
          inplace=False)
```

- `columns`: Речник за преименуване на колони.
- `index`: Речник за преименуване на етикети на индекси.
- `inplace`: Булев параметър. Ако е `True`, промените се извършват директно върху оригиналния `DataFrame`. По подразбиране е `False`, което означава, че методът връща нов `DataFrame` с преименуваните етикети.

### Пример: Преименуване на колони:

```
import pandas as pd  
  
# Създаваме примерен DataFrame  
data = {'col_A': [1, 2, 3], 'col_B': [4, 5, 6], 'col_C': [7, 8, 9]}  
df = pd.DataFrame(data)  
print("Оригинален DataFrame:\n", df)  
  
# Преименуване на колони с .rename()  
df_renamed_cols = df.rename(columns={'col_A': 'Първа_колона', 'col_B':  
'Втора_колона'})  
print("\nDataFrame с преименувани колони:\n", df_renamed_cols)  
  
# Преименуване на колони на място  
df.rename(columns={'col_C': 'Трета_колона'}, inplace=True)
```



```
print("\nDataFrame след преименуване на колона на място:\n", df)
```

- 1) **Създава се примерен DataFrame:** Инициализира се Pandas DataFrame df с три колони: 'col\_A', 'col\_B' и 'col\_C', съдържащи числови данни. Оригиналният DataFrame се извежда на конзолата.
- 2) **Преименуване на колони (създаване на нов DataFrame):** Методът `.rename()` се използва с аргумент `columns`, който представлява речник. Ключовете на речника са старите имена на колоните ('col\_A', 'col\_B'), а стойностите са новите имена ('Първа\_колона', 'Втора\_колона'). Параметърът `inplace` е оставен на стойността си по подразбиране `False`, което означава, че `.rename()` връща нов DataFrame `df_renamed_cols` с преименуваните колони. Оригиналният DataFrame df остава непроменен. Новият DataFrame с преименуваните колони се извежда на конзолата.
- 3) **Преименуване на колона на място (модифициране на оригиналния DataFrame):** Методът `.rename()` се използва отново върху оригиналния DataFrame df. Този път, в речника за `columns` се преименува колоната 'col\_C' на 'Трета\_колона'. Важно е, че е зададен параметърът `inplace=True`. Това означава, че промените се извършват директно върху DataFrame df, без да се създава нов обект. След изпълнението на тази команда, df вече ще има преименуваната колона. Обновеният DataFrame df се извежда на конзолата.

## Накратко:

Примерът показва два начина за преименуване на колони в Pandas DataFrame с помощта на `.rename()`: създаване на нов DataFrame с преименуваните колони (когато `inplace=False`) и модифициране на самия DataFrame на място (когато `inplace=True`). Използването на речник позволява преименуването на няколко колони едновременно.

### Пример: преименуване на етикети на индекси:

```
# Създаваме DataFrame с потребителски индекс
data_indexed = {'col1': [10, 20], 'col2': [30, 40]}
df_indexed = pd.DataFrame(data_indexed, index=['ред_1', 'ред_2'])
print("\nОригинален DataFrame с потребителски индекс:\n", df_indexed)

# преименуване на етикети на индекси
df_renamed_index = df_indexed.rename(index={'ред_1': 'Първи_ред',
'ред_2': 'Втори_ред'})
print("\nDataFrame с преименувани етикети на индекси:\n",
df_renamed_index)
```

- 1) **Създава се DataFrame с потребителски индекс:** Инициализира се Pandas DataFrame `df_indexed` с две колони ('col1' и 'col2') и потребителски индекс, състоящ се от етикетите 'ред\_1' и 'ред\_2'. Оригиналният DataFrame с този потребителски индекс се извежда на конзолата.
- 2) **Преименуване на етикети на индекси:** Методът `.rename()` се използва с аргумент `index`, който също представлява речник. Ключовете на речника са старите етикети на индексите ('ред\_1', 'ред\_2'), а стойностите са новите етикети ('Първи\_ред', 'Втори\_ред'). Параметърът `inplace` е оставен на стойността си по подразбиране `False`, което означава, че `.rename()` връща нов DataFrame `df_renamed_index` с преименуваните етикети на индексите. Оригиналният DataFrame `df_indexed` остава непроменен. Новият DataFrame с преименуваните етикети на индексите се извежда на конзолата.

## Накратко:

Примерът показва как `.rename()` може да се използва не само за преименуване на колони, но и за преименуване на етикетите на индексите (редовете) на DataFrame. Отново, използва се речник, където ключовете са старите етикети, а стойностите са новите. По подразбиране, операцията връща нов DataFrame. За да се промени оригиналният DataFrame на място, трябва да се използва `inplace=True`.

## 2. Преименуване на имената на нивата на MultiIndex с

`.rename_axis():`

Ако DataFrame или Series има MultiIndex (многостепенен индекс) за редовете или колоните, `.rename_axis()` се използва за преименуване на имената на тези нива.

## Синтаксис:

```
df.rename_axis(index='ново_име_ниво_ред', columns='ново_име_ниво_кол',
inplace=False)
```

- `index`: Стринг или списък от стрингове за новите имена на нивата на индекса на редовете.
- `columns`: Стринг или списък от стрингове за новите имена на нивата на индекса на колоните.
- `inplace`: Булев параметър за извършване на промени на място.

### Пример: Преименуване на имена на нива на MultiIndex:

```
# Създаваме DataFrame с MultiIndex за колоните
data_multi_col = {'Продукт': ['А', 'А', 'В', 'В'],
                  'Показател': ['Цена', 'Количество', 'Цена',
                                'Количество'],
                  'Стойност': [10, 5, 20, 8]}
df_multi_col = pd.DataFrame(data_multi_col)
```

```
df_multi_col = df_multi_col.pivot_table(index=df_multi_col.index,
columns=['Продукт', 'Показател'], values='Стойност')
print("\nОригинален DataFrame с MultiIndex за колоните:\n",
df_multi_col)

# Преименуване на имената на нивата на индекса на колоните
df_multi_col_renamed_axis = df_multi_col.rename_axis(columns=['Продукт',
'Характеристика'])
print("\nDataFrame с преименувани имена на нива на MultiIndex
(колони):\n", df_multi_col_renamed_axis)
```

- 1) **Създава се DataFrame с MultiIndex за колоните:** Първо, създава се обикновен DataFrame `df_multi_col` с колони 'Продукт', 'Показател' и 'Стойност'. След това се използва методът `.pivot_table()` за трансформиране на този DataFrame така, че колоните да образуват MultiIndex, състоящ се от нивата 'Продукт' и 'Показател'. Оригиналният DataFrame с този MultiIndex за колоните се извежда на конзолата. Забележете, че първоначално нивата на този MultiIndex нямат изрични имена (ще се показват като None или празни).
- 2) **Преименуване на имената на нивата на MultiIndex (колони):** Методът `.rename_axis()` се използва върху `df_multi_col` с аргумент `columns`. На този аргумент се присвоява списък от низове ['Продукт', 'Характеристика']. Този списък съдържа новите имена за нивата на MultiIndex на колоните. Поредността на имената в списъка съответства на поредността на нивата в MultiIndex-a. Методът `.rename_axis()` връща **нов** DataFrame `df_multi_col_renamed_axis` с преименуваните имена на нивата на MultiIndex на колоните. Оригиналният DataFrame `df_multi_col` остава непроменен. Новият DataFrame с преименуваните имена на нивата на колоните се извежда на конзолата.

## Накратко:

Примерът показва как `.rename_axis()` се използва специално за преименуване на имената на нивата в MultiIndex (както за колони, така и за редове). Това е полезно за подобряване на четимостта и разбирането на DataFrame-и с многостепенни индекси. Аргументът `columns` се използва за преименуване на имената на нивата на индекса на колоните, а аналогично се използва аргументът `index` за преименуване на имената на нивата на индекса на редовете.

### 3. Задаване на нови имена на колони или индекс с `.set_axis()`:

Методът `.set_axis()` позволява да зададете нови етикети на колоните или индекса (или и двете) директно, като приеме списък от нови имена. Броят на новите етикети трябва да съвпада с броя на съществуващите.

## Синтаксис:

```
df.set_axis(labels, axis=0, inplace=False) # За индекс (ред)
```

```
df.set_axis(labels, axis=1, inplace=False) # За колони
```

- labels: Списък от нови етикети.
- axis: 0 за индекс (ред), 1 за колони.
- inplace: Булев параметър за извършване на промени на място.

*Пример: Задаване на нови имена на колони с .set\_axis()*

```
# Използваме DataFrame df от Пример 1
print("\nDataFrame преди .set_axis():\n", df)

new_column_names = ['Първа', 'Втора', 'Трета']
df_set_axis_cols = df.set_axis(new_column_names, axis=1)
print("\nDataFrame след .set_axis() за колони:\n", df_set_axis_cols)
```

- 1) **Извежда се DataFrame df от Пример 1:** Първо, се извежда DataFrame df, който от предходните примери би трябвало да има колони 'Първа\_колона', 'Втора\_колона' и 'Трета\_колона' (ако е бил изпълнен примерът с inplace=True). Ако не е бил изпълнен с inplace=True, ще има колони 'col\_A', 'col\_B' и 'Трета\_колона'. Във всеки случай, df е DataFrame с три колони.
- 2) **Дефинира се списък с нови имена на колони:** Създава се списък new\_column\_names, съдържащ три низа: 'Първа', 'Втора' и 'Трета'. Броят на елементите в този списък трябва да съвпада с броя на колоните в DataFrame-a.
- 3) **Задаване на нови имена на колони с .set\_axis():** Методът .set\_axis() се използва върху DataFrame-a df. Първият аргумент е списъкът с новите имена на колони new\_column\_names. Вторият аргумент е axis=1, който указва, че новите етикети се прилагат към колоните. Параметърът inplace по подразбиране е False, така че .set\_axis() връща **нов** DataFrame df\_set\_axis\_cols с преименуваните колони. Оригиналният DataFrame df остава непроменен. Новият DataFrame с обновените имена на колоните се извежда на конзолата.

## Накратко:

Примерът показва как .set\_axis() позволява директно да се зададе нов списък от имена за всички колони на DataFrame-a. Важно е броят на новите имена да съвпада с броя на съществуващите колони. Методът връща нов DataFrame, освен ако не се използва inplace=True за модифициране на оригиналния.

*Пример: Задаване на нови етикети на индекс с .set\_axis()*

```
# Използваме DataFrame df_indexed от Пример 2
print("\nDataFrame с потребителски индекс преди .set_axis():\n",
df_indexed)
```

```
new_index_labels = ['P1', 'P2']
df_set_axis_index = df_indexed.set_axis(new_index_labels, axis=0)
print("\nDataFrame след .set_axis() за индекс:\n", df_set_axis_index)
```

- 1) **Извежда се DataFrame df\_indexed от Пример 2:** Първо, се извежда DataFrame df\_indexed, който има потребителски индекс с етикети 'ред\_1' и 'ред\_2', и две колони ('col1' и 'col2').
- 2) **Дефинира се списък с нови етикети на индекс:** Създава се списък new\_index\_labels, съдържащ два низа: 'P1' и 'P2'. Броят на елементите в този списък трябва да съвпада с броя на редовете (етикетите на индекса) в DataFrame-a.
- 3) **Задаване на нови етикети на индекс с .set\_axis():** Методът .set\_axis() се използва върху DataFrame-a df\_indexed. Първият аргумент е списъкът с новите етикети на индекс new\_index\_labels. Вторият аргумент е axis=0, който указва, че новите етикети се прилагат към индекса (редовете). Параметърът inplace по подразбиране е False, така че .set\_axis() връща **нов** DataFrame df\_set\_axis\_index с обновените етикети на индекса. Оригиналният DataFrame df\_indexed остава непроменен. Новият DataFrame с обновените етикети на индекса се извежда на конзолата.

## Накратко:

Примерът показва как .set\_axis() позволява директно да се зададе нов списък от етикети за индекса (редовете) на DataFrame-a. Важно е броят на новите етикети да съвпада с броя на съществуващите редове. Методът връща нов DataFrame, освен ако не се използва inplace=True за модифициране на оригиналния.

## 4. Обобщение:

- Използвайте .rename(), когато искате да преименувате конкретни колони или етикети на индекси по техните стари имена.
- Използвайте .rename\_axis(), когато работите с MultiIndex и искате да преименувате имената на нивата на индекса (както за редове, така и за колони).
- Използвайте .set\_axis(), когато искате да зададете изцяло нов списък от имена за колоните или етикети за индекса, като знаете точния брой и последователност на новите имена.

**Разбирането на тези методи е важно за привеждане на имената на колоните и индексите във вид, удобен за по-нататъшна обработка и анализ.**

## Казус 1: Стандартизиране на имена на колони (`.rename()` за колони)

Представете си, че сте получили `DataFrame` от външен източник, където имената на колоните не следват вашите стандарти (например, съдържат интервали, главни букви или са на чужд език). Искате да ги преименувате, за да бъдат по-лесни за използване и анализ.

```
import pandas as pd

# Създаваме DataFrame с нестандартни имена на колони
data = {
    'Име на Продукт': ['А', 'Б', 'В'],
    'Единична Цена (EUR)': [10.50, 20.30, 5.75],
    'Количество На Склад': [100, 50, 200]
}
df_products = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df_products)

# Казус: Преименувайте колоните на 'продукт', 'цена_евро' и 'наличност'.
```

### Решение на Казус 1:

```
# Използване на .rename() с речник за преименуване на колони
df_products_renamed = df_products.rename(columns={
    'Име на Продукт': 'продукт',
    'Единична Цена (EUR)': 'цена_евро',
    'Количество На Склад': 'наличност'
})
print("\nDataFrame след преименуване на колоните:\n",
df_products_renamed)
```

## Казус 2: Добавяне на име на индекс (`.rename_axis()` за индекс)

Представете си, че имате `DataFrame` с продажби по месеци, където индексът представлява месеците. Искате да дадете име на този индекс, за да бъде по-ясно какво представлява.

```
import pandas as pd
```

```
# Създаваме DataFrame с продажби по месеци
sales_data = {
    'продукт_А': [10, 15, 12],
    'продукт_Б': [20, 25, 18]
}
df_sales = pd.DataFrame(sales_data, index=['Януари', 'Февруари',
'Mарт'])
print("Оригинален DataFrame с индекс:\n", df_sales)

# Казус: Дайте име 'месец' на индекса на DataFrame-а.
```

## Решение на Казус 2:

```
# Използване на .rename_axis() за даване на име на индекса
df_sales_with_named_index = df_sales.rename_axis('месец')
print("\nDataFrame с име на индекс:\n", df_sales_with_named_index)
```

## Казус 3: Преименуване на нива на мултииндекс (.rename\_axis() за мултииндекс)

Представете си, че имате DataFrame с мултииндекс, представляващ година и тримесечие. Искате да дадете по-описателни имена на тези нива на индекса.

```
import pandas as pd

# Създаваме DataFrame с мултииндекс
data = {'продажби': [100, 120, 150, 130]}
index = pd.MultiIndex.from_tuples([('2022', 'Q1'), ('2022', 'Q2'),
('2023', 'Q1'), ('2023', 'Q2')], names=['година', 'тримесечие'])
df_multi_index = pd.DataFrame(data, index=index)
print("Оригинален DataFrame с мултииндекс:\n", df_multi_index)

# Казус: Преименувайте нивата на индекса от 'година' и 'тримесечие' на
'Година на отчитане' и 'Квартал'.
```



## Решение на Казус 3:

```
# Използване на .rename_axis() с речник за преименуване на нива на мултииндекс
df_multi_index_renamed_index =
df_multi_index.rename_axis(index={'година': 'Година на отчитане',
'тримесечие': 'Квартал'})
print("\nDataFrame с преименувани нива на мултииндекс:\n",
df_multi_index_renamed_index)
```

## Казус 4: Задаване на нови имена на колони за съществуващ DataFrame (.set\_axis() за колони)

Представете си, че имате DataFrame без описателни имена на колони (например, само числови индекси) и искате да ги замените с нови имена.

```
import pandas as pd

# Създаваме DataFrame без описателни имена на колони
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
df_no_column_names = pd.DataFrame(data)
print("Оригинален DataFrame без имена на колони:\n", df_no_column_names)

# Казус: Задайте новите имена на колони 'A', 'B' и 'C'.
```

## Решение на Казус 4:

```
# Използване на .set_axis() за задаване на нови имена на колони
new_column_names = ['A', 'B', 'C']
df_with_column_names = df_no_column_names.set_axis(new_column_names,
axis=1)
print("\nDataFrame с нови имена на колони:\n", df_with_column_names)
```

## Казус 5: Задаване на нови имена на индекс за Series (.set\_axis() за индекс)

Представете си, че имате Series с числови индекси и искате да ги замените с по-описателни имена.

```
import pandas as pd

# Създаваме Series с числови индекси
data = pd.Series([10, 20, 30])
print("Оригинален Series с числов индекс:\n", data)

# Казус: Задайте новите имена на индекс 'първи', 'втори' и 'трети'.
```

### Решение на Казус 5:

```
# Използване на .set_axis() за задаване на нови имена на индекс
new_index_names = ['първи', 'втори', 'трети']
data_with_named_index = data.set_axis(new_index_names, axis=0)
print("\nSeries с нови имена на индекс:\n", data_with_named_index)
```

Тези казуси илюстрират как `.rename()`, `.rename_axis()` и `.set_axis()` предоставят гъвкави начини за преименуване и задаване на имена на колони и индекси в Pandas, което е важно за почистване на данни, подобряване на четимостта и подготовка за анализ.

## Х. Задаване и нулиране на индекса (.set\_index(), .reset\_index())

Индексът в Pandas DataFrame и Series е мощен инструмент за етикетиране и достъп до данни. Понякога се налага да превърнем съществуваща колона в индекс или да върнем индекса обратно като обикновена колона. Методите `.set_index()` и `.reset_index()` са предназначени за тези цели.

### 1. Задаване на индекс с .set\_index():

Методът `.set_index()` позволява да превърнете една или повече съществуващи колони в индекс на DataFrame.

### Синтаксис:

```
df.set_index(keys, drop=True, append=False, inplace=False,
verify_integrity=False)
```

- `keys`: Стринг (име на една колона) или списък от стрингове (имена на няколко колони), които ще станат индекс.
- `drop`: Булев параметър. Ако е `True` (по подразбиране), колоните, които стават индекс, се премахват от `DataFrame`-а. Ако е `False`, те остават като обикновени колони.
- `append`: Булев параметър. Ако е `True`, колоните, посочени в `keys`, се добавят към съществуващия индекс (което води до `MultiIndex`). Ако е `False` (по подразбиране), съществуващият индекс се заменя.
- `inplace`: Булев параметър за извършване на промени на място.
- `verify_integrity`: Булев параметър. Ако е `True`, проверява се за дублирани стойности в новия индекс.

#### *а) Пример 1: Задаване на една колона като индекс*

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'Име': ['Алиса', 'Боб', 'Чарли'],
        'Възраст': [25, 30, 22],
        'Държава': ['България', 'САЩ', 'България']}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Задаваме колона 'Име' като индекс
df_indexed_name = df.set_index('Име')
print("\nDataFrame с 'Име' като индекс:\n", df_indexed_name)

# Задаваме колона 'Държава' като индекс и запазваме я като колона
df_indexed_country = df.set_index('Държава', drop=False)
print("\nDataFrame с 'Държава' като индекс (колоната остава):\n",
df_indexed_country)
```

- 1) **Създава се примерен `DataFrame`:** Инициализира се `Pandas DataFrame` `df` с три колони: 'Име', 'Възраст' и 'Държава', съдържащи данни за хора. Оригиналният `DataFrame` с автоматично генериран числов индекс се извежда на конзолата.
- 2) **Задаване на колона 'Име' като индекс:** Методът `.set_index('Име')` се използва, за да превърне съществуващата колона 'Име' в индекс на `DataFrame`-а. По подразбиране (`drop=True`), оригиналната колона 'Име' се премахва от `DataFrame`-а. Резултатът е нов `DataFrame`

`df_indexed_name`, където имената на хората вече са етикети на редовете (индексът). Този `DataFrame` се извежда на конзолата.

- 3) **Задаване на колона 'Държава' като индекс (колоната остава):** Методът `.set_index('Държава', drop=False)` се използва, за да превърне колоната 'Държава' в индекс. В този случай, параметърът `drop` е зададен на `False`. Това означава, че оригиналната колона 'Държава' не се премахва и остава като обикновена колона в `DataFrame`-а. Резултатът е нов `DataFrame` `df_indexed_country`, където държавите вече са етикети на редовете, а колоната 'Държава' все още съществува. Този `DataFrame` се извежда на конзолата.

## Накратко:

Примерът илюстрира как `.set_index()` позволява лесно да се превърне една съществуваща колона в индекс на `DataFrame`. Параметърът `drop` контролира дали оригиналната колона ще бъде премахната или ще остане като част от данните. Това е полезно за организиране на данните по определен атрибут и за улесняване на последващи операции, базирани на този индекс.

### б) Пример 2: Задаване на няколко колони като *MultiIndex*

Методът `.reset_index()` преобразува индекса (или *MultiIndex*) обратно в обикновени колони на `DataFrame`.

## Синтаксис:

```
df.reset_index(level=None, drop=False, inplace=False, names=None)
```

- `level`: Определя кое ниво на *MultiIndex* да бъде превърнато в колона. Може да бъде име на ниво (стринг) или пореден номер (int). По подразбиране (`None`) всички нива на индекса се нулират.
- `drop`: Булев параметър. Ако е `True`, индексът се премахва напълно, вместо да се добавя като нова колона.
- `inplace`: Булев параметър за извършване на промени на място.
- `names`: Списък от имена за колоните, създадени от нивата на индекса. Ако е `None`, се използват имената на нивата на индекса (или поредица от числа, ако индексът е без име).

### в) Пример 3: Нулиране на обикновен индекс

```
# Нулираме индекса на df_indexed_name от Пример 1
df_reset_name = df_indexed_name.reset_index()
print("\nDataFrame след нулиране на индекса 'Име':\n", df_reset_name)

# Нулираме индекса и го премахваме
df_reset_drop = df_indexed_name.reset_index(drop=True)
print("\nDataFrame след нулиране и премахване на индекса:\n",
df_reset_drop)
```

- 1) **Използва се DataFrame df\_indexed\_name от Пример 1:** Предполага се, че df\_indexed\_name е DataFrame, където колоната 'Име' е зададена като индекс (както беше показано в предходния пример).
- 2) **Нулиране на индекса 'Име' (превръщане в колона):** Методът .reset\_index() се използва върху df\_indexed\_name без изрични аргументи (или с drop=False, което е стойността по подразбиране). Това преобразува индекса ('Име') обратно в обикновена колона с име 'Име'. Оригиналният индекс се заменя с нов, автоматично генериран числов индекс (0, 1, 2, ...). Резултатът е нов DataFrame df\_reset\_name, който се извежда на конзолата.
- 3) **Нулиране на индекса и премахването му:** Методът .reset\_index(drop=True) се използва отново върху df\_indexed\_name. В този случай, параметърът drop е зададен на True. Това означава, че индексът ('Име') се нулира, но вместо да бъде добавен като нова колона, той се **премахва** напълно от DataFrame-а. DataFrame-ът получава нов, автоматично генериран числов индекс. Резултатът е нов DataFrame df\_reset\_drop (който вече няма колона 'Име' освен ако не е съществувала и преди задаването като индекс), който се извежда на конзолата.

## Накратко:

Примерът илюстрира как .reset\_index() позволява да се върне индексът на DataFrame обратно като обикновена колона, като се създава нов числов индекс. Използването на drop=True води до пълното премахване на стария индекс, без да се запазва като колона. Това е полезно, когато индексът вече не е необходим като етикет и искате да го третирате като част от данните или да го премахнете.

### г) Пример 4: Нулиране на MultiIndex

```
# Нулираме MultiIndex-а на df_multi_indexed от Пример 2
df_reset_multi = df_multi_indexed.reset_index()
print("\nDataFrame след нулиране на MultiIndex:\n", df_reset_multi)

# Нулираме само едно ниво от MultiIndex-а
df_reset_level = df_multi_indexed.reset_index(level='Държава')
print("\nDataFrame след нулиране само на ниво 'Държава':\n",
df_reset_level)
```

- 1) **Използва се DataFrame df\_multi\_indexed от Пример 2:** Предполага се, че df\_multi\_indexed е DataFrame, където колоните 'Държава' и 'Име' са зададени като MultiIndex (както беше показано в предходен пример). Индексът ще има две нива: 'Държава' (външно ниво) и 'Име' (вътрешно ниво).
- 2) **Нулиране на целия MultiIndex:** Методът .reset\_index() се използва върху df\_multi\_indexed без изрични аргументи (или с drop=False, което е стойността по подразбиране). Това преобразува **всички нива** на MultiIndex-а ('Държава' и 'Име') обратно в обикновени колони с имената на нивата ('Държава' и 'Име'). Оригиналният MultiIndex се заменя с нов, автоматично генериран числов индекс (0, 1, 2, ...). Резултатът е нов DataFrame df\_reset\_multi, който се извежда на конзолата.

- 3) **Нулиране само на едно ниво от MultiIndex-a:** Методът `.reset_index(level='Държава')` се използва върху `df_multi_indexed`. Аргументът `level='Държава'` указва, че само нивото на индекса с име 'Държава' трябва да бъде превърнато в обикновена колона. Нивото на индекса 'Име' остава като част от индекса. Резултатът е нов `DataFrame` `df_reset_level`, който се извежда на конзолата. Индексът вече ще бъде само ниво 'Име', а 'Държава' ще бъде обикновена колона.

## Накратко:

Примерът илюстрира как `.reset_index()` може да се използва за превръщане на `MultiIndex` обратно в обикновени колони. Когато не е посочен аргумент `level`, всички нива на `MultiIndex`-а стават колони. Чрез използване на аргумента `level`, може да се избере конкретно ниво от `MultiIndex`-а, което да се превърне в колона, докато останалите нива остават като индекс. Това дава гъвкавост при реструктуриране на `DataFrame`-и с многостепенни индекси.

Разбирането на `.set_index()` и `.reset_index()` е ключово за гъвкаво манипулиране на структурата на вашите `DataFrame`-и и за подготовка на данните за различни видове анализ и визуализация.

## 2. Обобщение на `.set_index()` и `.reset_index()`:

- **`.set_index()`:** Преобразува една или повече съществуващи колони в индекс на `DataFrame`. Това е полезно, когато искате да използвате стойности от колона за етиктиране на редовете, което може да улесни търсенето, подравняването и други операции.
- **`.reset_index()`:** Преобразува индекса (който може да е обикновен или `MultiIndex`) обратно в една или повече обикновени колони. Това е полезно, когато искате да третирате индекса като част от данните или когато индексът пречи на определени операции.

## 3. Насоки при работа

### а) със задаване на индекс (`.set_index()`):

- **Изберете подходящи колони за индекс:** Колоните, които избирате за индекс, трябва да съдържат уникални или почти уникални стойности, особено ако планирате да използвате индекса за бързо търсене на редове. Ако изберете колона с повтарящи се стойности, ще получите индекс с повтарящи се етикети, което е валидно, но може да повлияе на производителността на някои операции.
- **Обмислете `MultiIndex`:** Ако имате няколко колони, които заедно идентифицират уникално редовете или представляват йерархична структура, задаването им като `MultiIndex` може да бъде много полезно за по-сложно индексване и анализ.
- **Контролирайте премахването на колоните:** По подразбиране, колоните, които стават индекс, се премахват от `DataFrame`-а (`drop=True`). Ако искате да ги запазите като обикновени колони, използвайте `drop=False`.
- **Внимавайте с `append=True`:** Използвайте `append=True` само когато съзнателно искате да добавите нови колони към съществуващия индекс, създавайки `MultiIndex`. В противен случай, оставете тази опция на `False`, за да замените съществуващия индекс.

- **Проверявайте за дубликати:** Ако е важно индексът да бъде уникален, използвайте `verify_integrity=True` за да хвърлите грешка, ако бъдат открити дублирани стойности в новия индекс.

#### *б) с нулиране на индекс (`.reset_index()`):*

- **Определете нивото за нулиране:** При `MultiIndex`, използвайте параметъра `level`, за да изберете кои нива на индекса да станат колони. Можете да използвате името на нивото или неговия пореден номер.
- **Решете дали да запазите или премахнете стария индекс:** По подразбиране, `.reset_index()` добавя стария индекс като нова колона (`drop=False`). Ако не ви е нужен, използвайте `drop=True`, за да го премахнете.
- **Наименувайте новите колони:** Когато нулирате `MultiIndex`, можете да предоставите списък от имена за новите колони чрез параметъра `names`. Ако не предоставите имена, ще се използват имената на нивата на индекса (ако има такива).
- **Последователност на колоните:** Когато нулирате индекс, новите колони, произлизащи от индекса, обикновено се вмъкват в началото на `DataFrame`-а. Имайте това предвид при последваща обработка.
- **Възстановяване на първоначален числов индекс:** Ако искате да върнете `DataFrame`-а към неговия първоначален числов индекс (0, 1, 2, ...), често е препоръчително да използвате `.reset_index(drop=True)`.

## *4. Кога да използвате `.set_index()` и `.reset_index()`:*

- Използвайте `.set_index()`, когато искате да използвате една или повече колони за по-ефективно търсене на редове, за извършване на операции, базирани на етикети, или за привеждане на данните в определен формат, очакван от други функции или библиотеки.
- Използвайте `.reset_index()`, когато индексът пречи на определени операции (например, когато искате да третирате всички данни еднакво при агрегиране или присъединяване), когато искате да запазите стойностите от индекса като част от данните за по-нататъшен анализ, или когато искате да върнете `DataFrame`-а към стандартен числов индекс.

### *Казус 1: Задаване на колона като индекс (`.set_index()`)*

Представете си, че имате `DataFrame` с данни за продажби, който включва колона с уникални идентификатори на продукти. Искате да използвате тази колона като индекс на `DataFrame`-а за по-лесно търсене и анализ на данни по продукт.

```
import pandas as pd

# Създаваме DataFrame с данни за продажби
sales_data = {
    'product_id': [101, 102, 103, 104, 105],
    'product_name': ['Телевизор', 'Лаптоп', 'Мишка', 'Клавиатура',
                    'Монитор'],
```



```

    'sales': [150, 220, 30, 45, 180]
}
df_sales = pd.DataFrame(sales_data)
print("Оригинален DataFrame:\n", df_sales)

# Казус: Задайте колоната 'product_id' като индекс на DataFrame-а.

```

## Решение на Казус 1:

```

# Използване на .set_index() за задаване на колона като индекс
df_sales_indexed = df_sales.set_index('product_id')
print("\nDataFrame с 'product_id' като индекс:\n", df_sales_indexed)

```

## Казус 2: Задаване на мултииндекс от няколко колони (.set\_index() с множество колони)

Представете си, че имате DataFrame с данни за продажби, категоризирани по регион и месец. Искате да създадете мултииндекс от колоните 'регион' и 'месец' за по-детайлен анализ.

```

import pandas as pd

# Създаваме DataFrame с данни за продажби по регион и месец
sales_region_month = {
    'регион': ['Север', 'Север', 'Юг', 'Юг'],
    'месец': ['Януари', 'Февруари', 'Януари', 'Февруари'],
    'продажби': [100, 120, 150, 130]
}
df_sales_multi = pd.DataFrame(sales_region_month)
print("Оригинален DataFrame:\n", df_sales_multi)

# Казус: Задайте колоните 'регион' и 'месец' като мултииндекс.

```

## Решение на Казус 2:

```
# Използване на .set_index() с списък от колони за създаване на
мултииндекс

df_sales_multi_indexed = df_sales_multi.set_index(['регион', 'месец'])
print("\nDataFrame с мултииндекс ('регион', 'месец'):\n",
df_sales_multi_indexed)
```

### Казус 3: Нулиране на индекса (.reset\_index())

Представете си, че имате DataFrame с индекс, който е бил зададен от колона, но сега искате да го превърнете обратно в обикновени колони, като добавите нов числов индекс по подразбиране.

```
import pandas as pd

# Създаваме DataFrame с 'product_id' като индекс (от предходния казус)
sales_data = {
    'product_id': [101, 102, 103, 104, 105],
    'product_name': ['Телевизор', 'Лаптоп', 'Мишка', 'Клавиатура',
'Mонитор'],
    'sales': [150, 220, 30, 45, 180]
}
df_sales_indexed = pd.DataFrame(sales_data).set_index('product_id')
print("DataFrame с 'product_id' като индекс:\n", df_sales_indexed)

# Казус: Нулирайте индекса, превръщайки 'product_id' обратно в колона.
```

### Решение на Казус 3:

```
# Използване на .reset_index() за нулиране на индекса
df_sales_reset = df_sales_indexed.reset_index()
print("\nDataFrame след нулиране на индекса:\n", df_sales_reset)
```

## Казус 4: Нулиране на мултииндекс (.reset\_index())

Представете си, че имате DataFrame с мултииндекс и искате да го превърнете обратно в DataFrame с обикновен числов индекс и нива на мултииндекса като отделни колони.

```
import pandas as pd

# Създаваме DataFrame с мултииндекс ('регион', 'месец') (от предходния
казус)
sales_region_month = {
    'регион': ['Север', 'Север', 'Юг', 'Юг'],
    'месец': ['Януари', 'Февруари', 'Януари', 'Февруари'],
    'продажби': [100, 120, 150, 130]
}
df_sales_multi_indexed =
pd.DataFrame(sales_region_month).set_index(['регион', 'месец'])
print("DataFrame с мултииндекс ('регион', 'месец'):\n",
df_sales_multi_indexed)

# Казус: Нулирайте мултииндекса.
```

## Решение на Казус 4:

```
# Използване на .reset_index() за нулиране на мултииндекса
df_sales_multi_reset = df_sales_multi_indexed.reset_index()
print("\nDataFrame след нулиране на мултииндекса:\n",
df_sales_multi_reset)
```

## Казус 5: Запазване на стария индекс като колона при нулиране (.reset\_index(drop=False))

В предишния пример, .reset\_index() превърна индекса в колони. По подразбиране, оригиналният индекс се запазва като нова колона.

```
import pandas as pd

# Създаваме DataFrame с 'product_id' като индекс
```

```

sales_data = {
    'product_id': [101, 102, 103],
    'sales': [150, 220, 30]
}

df_sales_indexed = pd.DataFrame(sales_data).set_index('product_id')
print("DataFrame с 'product_id' като индекс:\n", df_sales_indexed)

# Казус: Нулирайте индекса, запазвайки 'product_id' като колона (което е
поведението по подразбиране).
df_sales_reset_default = df_sales_indexed.reset_index()
print("\nDataFrame след reset_index() (по подразбиране):\n",
df_sales_reset_default)

# Казус: Нулирайте индекса, премахвайки го напълно (`drop=True`).
df_sales_reset_dropped = df_sales_indexed.reset_index(drop=True)
print("\nDataFrame след reset_index(drop=True):\n",
df_sales_reset_dropped)

```

Тези казуси илюстрират как `.set_index()` и `.reset_index()` са основни инструменти за манипулиране на структурата на `DataFrame`-а чрез задаване на една или няколко колони като индекс и превръщането на индекса обратно в колони, което е важно за различни видове анализ и обработка на данни. Параметърът `drop` в `.reset_index()` предлага контрол върху това дали оригиналният индекс да бъде запазен като нова колона или да бъде премахнат.

## XI. Сортиране на данни:

Сортирането е фундаментална операция при анализа на данни, която ни позволява да подредим редовете на `DataFrame` или елементите на `Series` въз основа на определени критерии. `Pandas` предлага два основни метода за сортиране:

- `.sort_values()`: Сортира данните по стойностите в една или няколко колони.
- `.sort_index()`: Сортира данните по етикетите на индекса (редовете).

Разбирането на тези методи и техните параметри е ключово за организиране на данните по желания от нас начин за по-лесен анализ и интерпретация. Нека разгледаме всеки от тях по-подробно.

## 1. Сортиране по стойности (.sort\_values()):

Методът `.sort_values()` се използва за сортиране на редовете на `DataFrame` или елементите на `Series` въз основа на стойностите в една или повече колони.

**Синтаксис за DataFrame:**

```
df.sort_values(by, axis=0, ascending=True, inplace=False,
kind='quicksort', na_position='last', ignore_index=False, key=None)
```

- `by`: Стринг (за една колона) или списък от стрингове (за множество колони), указващи по кои колони да се сортира.
- `axis`: Ос за сортиране. 0 (по подразбиране) за сортиране по редове (в рамките на колони). 1 за сортиране по колони (в рамките на редове), което е по-рядко използвано.
- `ascending`: Булев или списък от булеви стойности. Определя посоката на сортиране. `True` (по подразбиране) за възходящ ред (от най-малката към най-голямата стойност). Ако е списък, трябва да съответства на броя на колоните в `by` и указва посоката на сортиране за всяка колона.
- `inplace`: Булев параметър. Ако е `True`, сортирането се извършва на място (върху оригиналния `DataFrame`). По подразбиране е `False`, което връща нов, сортиран `DataFrame`.
- `kind`: Алгоритъм за сортиране. По подразбиране е 'quicksort', но могат да се използват и други ('mergesort', 'heapsort').
- `na_position`: Стринг, указващ къде да се поставят липсващите стойности (`NaN`). Възможни стойности са 'first' (в началото) и 'last' (по подразбиране, в края).
- `ignore_index`: Булев параметър. Ако е `True`, новият индекс ще бъде 0, 1, ..., n-1.
- `key`: Функция, която се прилага към стойностите преди сортиране.

**Синтаксис за Series:**

```
s.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort',
na_position='last', ignore_index=False, key=None)
```

*Параметрите са сходни с тези на DataFrame, но `by` не е необходим, тъй като Series има само една "колона" от стойности.*

### а) Пример 1: Сортиране по една колона

```
import pandas as pd

# Създаваме DataFrame с данни за продажби
data = {'Продукт': ['A', 'B', 'A', 'C', 'B'],
        'Продажби': [100, 150, 120, 90, 160]}
df_sales = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df_sales)
```

```
# Сортиране по колона 'Продажби' във възходящ ред (по подразбиране)
df_sorted_sales_asc = df_sales.sort_values(by='Продажби')
print("\nDataFrame, сортиран по 'Продажби' (възходящо):\n",
df_sorted_sales_asc)

# Сортиране по колона 'Продукт' в низходящ ред
df_sorted_product_desc = df_sales.sort_values(by='Продукт',
ascending=False)
print("\nDataFrame, сортиран по 'Продукт' (низходящо):\n",
df_sorted_product_desc)

# Сортиране на място
df_sales.sort_values(by='Продажби', ascending=False, inplace=True)
print("\nDataFrame, сортиран по 'Продажби' (низходящо, на място):\n",
df_sales)
```

*В този пример виждаме как да сортираме DataFrame по една колона, като контролираме посоката на сортиране (възходяща или низходяща) и дали операцията да се извърши на място.*

- 1) **Създава се DataFrame с данни за продажби:** Инициализира се Pandas DataFrame `df_sales` с две колони: 'Продукт' и 'Продажби'. Оригиналният DataFrame се извежда на конзолата, показвайки данните в първоначалния им ред.
- 2) **Сортиране по колона 'Продажби' (възходящо):** Методът `.sort_values(by='Продажби')` се използва за сортиране на редовете на DataFrame-а въз основа на стойностите в колона 'Продажби'. Тъй като параметърът `ascending` не е изрично зададен, се използва стойността му по подразбиране `True`, което води до сортиране във възходящ ред (от най-малката към най-голямата стойност на продажбите). Резултатът е нов DataFrame `df_sorted_sales_asc`, който се извежда на конзолата, показващ редовете, подредени според продажбите във възходящ ред.
- 3) **Сортиране по колона 'Продукт' (низходящо):** Методът `.sort_values(by='Продукт', ascending=False)` се използва за сортиране на редовете въз основа на стойностите в колона 'Продукт'. Параметърът `ascending=False` указва, че сортирането трябва да бъде в низходящ ред (от 'Z' към 'A' за низове). Резултатът е нов DataFrame `df_sorted_product_desc`, който се извежда на конзолата, показващ редовете, подредени според името на продукта в низходящ азбучен ред.
- 4) **Сортиране по колона 'Продажби' (низходящо, на място):** Методът `.sort_values(by='Продажби', ascending=False, inplace=True)` се използва отново за сортиране по колона 'Продажби' в низходящ ред. Важно е, че е зададен параметърът `inplace=True`. Това означава, че сортирането се извършва директно върху оригиналния DataFrame `df_sales`, без да се създава нов обект. След изпълнението на тази команда, `df_sales` вече ще бъде сортиран по продажби в низходящ ред. Обновеният DataFrame `df_sales` се извежда на конзолата.

## Накратко:

Примерът демонстрира как `.sort_values()` позволява сортиране на `DataFrame` по една колона, като се контролира посоката на сортиране и дали промените да се приложат на място. Това е основна операция за подреждане на данни с цел анализ и визуализация.

## Пример 2: Сортиране по няколко колони

Когато сортираме по няколко колони, Pandas първо сортира по първата посочена колона, а след това, в рамките на всяка група от еднакви стойности в първата колона, сортира по втората колона и така нататък.

```
import pandas as pd

# Създаваме DataFrame с данни за поръчки
data_orders = {'Категория': ['Електроника', 'Дрехи', 'Електроника',
                              'Дрехи', 'Книги'],
               'Цена': [100, 50, 120, 60, 20],
               'Количество': [2, 5, 1, 3, 10]}

df_orders = pd.DataFrame(data_orders)
print("Оригинален DataFrame за поръчки:\n", df_orders)

# Сортиране първо по 'Категория' (възходящо), след това по 'Цена'
(низходящо)
df_sorted_multi = df_orders.sort_values(by=['Категория', 'Цена'],
                                         ascending=[True, False])
print("\nDataFrame, сортиран по 'Категория' (възходящо) и 'Цена'
(низходящо):\n", df_sorted_multi)
```

*Тук виждаме как да сортираме по две колони, като задаваме различна посока на сортиране за всяка от тях чрез списък в параметъра `ascending`.*

- 1) **Създава се DataFrame с данни за поръчки:** Инициализира се Pandas `DataFrame` `df_orders` с три колони: 'Категория', 'Цена' и 'Количество', съдържащи информация за поръчки. Оригиналният `DataFrame` се извежда на конзолата.
- 2) **Сортиране по множество колони:** Методът `.sort_values()` се използва с аргумента `by`, който приема списък от имена на колони `['Категория', 'Цена']`. Това указва, че сортирането ще се извърши първо по колона 'Категория', а след това, в рамките на всяка група от еднакви стойности в колона 'Категория', ще се извърши сортиране по колона 'Цена'.

Аргументът `ascending` също приема списък от булеви стойности `[True, False]`. Броят на елементите в този списък трябва да съответства на броя на колоните в списъка `by`. `True` указва възходящо сортиране за съответната колона, а `False` указва низходящо сортиране. В този



случай, `True` е за 'Категория' (сортиране по категории в азбучен ред), а `False` е за 'Цена' (сортиране по цена в низходящ ред в рамките на всяка категория).

- 3) **Извежда се сортираният DataFrame:** Резултатът е нов DataFrame `df_sorted_multi`, който се извежда на конзолата. Редовете са подредени първо по категория (Електроника, Дрехи, Книги), а след това, за всяка категория, по цена от най-високата към най-ниската.

## Накратко:

Примерът показва как `.sort_values()` може да се използва за сортиране на DataFrame по повече от една колона. Редът на колоните в списъка, подаден на `by`, определя приоритета на сортиране, а списъкът, подаден на `ascending`, определя посоката на сортиране за всяка съответна колона. Това е полезно за получаване на подреждане на данните, което отговаря на множество критерии.

### б) Пример 3: Управление на позицията на липсващи стойности (`na_position`)

Когато данните съдържат липсващи стойности (`NaN`), можем да контролираме къде да бъдат поставени те при сортиране.

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с липсващи стойности
data_na = {'Име': ['Алиса', 'Боб', 'Чарли', 'Деси'],
            'Оценка': [85, np.nan, 92, 78]}
df_na = pd.DataFrame(data_na)
print("Оригинален DataFrame с липсващи стойности:\n", df_na)

# Сортиране по 'Оценка', като NaN са поставени в началото
df_sorted_na_first = df_na.sort_values(by='Оценка', na_position='first')
print("\nDataFrame, сортиран по 'Оценка' (NaN в началото):\n",
      df_sorted_na_first)

# Сортиране по 'Оценка', като NaN са поставени в края (по подразбиране)
df_sorted_na_last = df_na.sort_values(by='Оценка', na_position='last')
print("\nDataFrame, сортиран по 'Оценка' (NaN в края):\n",
      df_sorted_na_last)
```

*Този пример показва как параметърът `na_position` ни позволява да контролираме местоположението на редовете с липсващи стойности при сортиране.*

- 1) **Създава се DataFrame с липсващи стойности:** Инициализира се Pandas DataFrame `df_na` с две колони: 'Име' и 'Оценка'. Колоната 'Оценка' съдържа една липсваща стойност (`np.nan`). Оригиналният DataFrame се извежда на конзолата.
- 2) **Сортиране с `na_position='first'`:** Методът `.sort_values(by='Оценка', na_position='first')` се използва за сортиране на редовете на DataFrame-а въз основа на стойностите в колоната 'Оценка'. Параметърът `na_position='first'` указва, че всички редове, съдържащи NaN в колоната за сортиране ('Оценка'), трябва да бъдат поставени в началото на сортирания DataFrame. Резултатът е нов DataFrame `df_sorted_na_first`, който се извежда на конзолата, показващ реда с NaN стойност в 'Оценка' като първи.
- 3) **Сортиране с `na_position='last'` (по подразбиране):** Методът `.sort_values(by='Оценка', na_position='last')` се използва отново за сортиране по колоната 'Оценка'. Параметърът `na_position='last'` указва, че всички редове с NaN стойности трябва да бъдат поставени в края на сортирания DataFrame. Това е стойността по подразбиране на параметъра `na_position`, така че ако не бъде изрично зададен, NaN стойностите ще бъдат поставени в края. Резултатът е нов DataFrame `df_sorted_na_last`, който се извежда на конзолата, показващ реда с NaN стойност в 'Оценка' като последен.

## Накратко:

Примерът илюстрира как параметърът `na_position` в метода `.sort_values()` позволява да се контролира местоположението на редовете с липсващи стойности (NaN) при сортиране. Можете да изберете дали NaN стойностите да се появяват в началото ('first') или в края ('last') на сортирания резултат, което може да бъде важно в зависимост от нуждите на анализа.

## 2. Сортиране по индекс (`.sort_index()`):

Методът `.sort_index()` се използва за сортиране на DataFrame или Series въз основа на етикетите на техния индекс (редовете). Той е особено полезен, когато индексът не е просто поредица от числа, а съдържа значима информация или е MultiIndex.

**Синтаксис за DataFrame:**

```
df.sort_index(axis=0, level=None, ascending=True, inplace=False,
sort_remaining=True, key=None)
```

- `axis`: Ос за сортиране. 0 (по подразбиране) за сортиране по индекс на редовете. 1 за сортиране по индекс на колоните (ако има такъв).
- `level`: Ако DataFrame има MultiIndex, този параметър указва кое ниво (или нива) на индекса да се използва за сортиране. Може да бъде име на ниво (string) или пореден номер (int) или списък от тях. Ако е None, се сортира по всички нива.

- `ascending`: Булев или списък от булеви стойности. Определя посоката на сортиране за всяко ниво (ако е приложимо). `True` (по подразбиране) за възходящ ред.
- `inplace`: Булев параметър за извършване на промени на място.
- `sort_remaining`: Ако се сортира по ниво на `MultiIndex`, този параметър определя дали останалите нива също да бъдат сортирани (в рамките на групите, определени от сортираното ниво). По подразбиране е `True`.
- `key`: Функция, която се прилага към етикетите на индекса преди сортиране.

## Синтаксис за `Series`:

```
s.sort_index(axis=0, level=None, ascending=True, inplace=False,
sort_remaining=False, key=None)
```

*Параметрите са сходни с тези на `DataFrame`.*

Нека разгледаме примери, които илюстрират сортиране по ос (индекс на редове и индекс на колони), по ниво на `MultiIndex` и в различна посока.

### *а) Пример 1: Сортиране по индекс на редове*

```
import pandas as pd

# Създаваме DataFrame с не сортиран индекс
data = {'Колона1': [10, 20, 30], 'Колона2': [40, 50, 60]}
df_unsorted_index = pd.DataFrame(data, index=['C', 'A', 'B'])
print("Оригинален DataFrame с не сортиран индекс:\n", df_unsorted_index)

# Сортиране по индекс във възходящ ред (по подразбиране)
df_sorted_index_asc = df_unsorted_index.sort_index()
print("\nDataFrame, сортиран по индекс (възходящо):\n",
df_sorted_index_asc)

# Сортиране по индекс в низходящ ред
df_sorted_index_desc = df_unsorted_index.sort_index(ascending=False)
print("\nDataFrame, сортиран по индекс (низходящо):\n",
df_sorted_index_desc)

# Сортиране на място
df_unsorted_index.sort_index(inplace=True)
print("\nDataFrame, сортиран по индекс (възходящо, на място):\n",
df_unsorted_index)
```

*Този пример показва как да сортираме DataFrame по етикетите на неговия индекс, като контролираме посоката и прилагаме промените на място.*

- 1) **Създава се DataFrame с не сортиран индекс:** Инициализира се Pandas DataFrame `df_unsorted_index` с две колони ('Колона1' и 'Колона2') и потребителски индекс, състоящ се от етикетите 'C', 'A' и 'B', които не са в азбучен ред. Оригиналният DataFrame с този не сортиран индекс се извежда на конзолата.
- 2) **Сортиране по индекс (възходящо):** Методът `.sort_index()` се използва без изрични аргументи (или с `ascending=True`, което е стойността по подразбиране). Това сортира DataFrame-а въз основа на етикетите на индекса във възходящ (азбучен) ред. Резултатът е нов DataFrame `df_sorted_index_asc`, който се извежда на конзолата, показващ редовете, подредени по индекса 'A', 'B', 'C'.
- 3) **Сортиране по индекс (низходящо):** Методът `.sort_index(ascending=False)` се използва с параметъра `ascending=False`. Това сортира DataFrame-а въз основа на етикетите на индекса в низходящ (обратен азбучен) ред. Резултатът е нов DataFrame `df_sorted_index_desc`, който се извежда на конзолата, показващ редовете, подредени по индекса 'C', 'B', 'A'.
- 4) **Сортиране по индекс на място (възходящо):** Методът `.sort_index(inplace=True)` се използва с параметъра `inplace=True`. Това сортира директно оригиналния DataFrame `df_unsorted_index` въз основа на неговия индекс във възходящ ред. След изпълнението на тази команда, `df_unsorted_index` вече ще бъде сортиран по индекса 'A', 'B', 'C'. Обновеният DataFrame `df_unsorted_index` се извежда на конзолата.

## Накратко:

Примерът демонстрира как `.sort_index()` позволява сортиране на DataFrame по неговите етикети на индекс. Може да се контролира посоката на сортиране (възходяща или низходяща), а с помощта на `inplace=True` промените могат да бъдат приложени директно върху оригиналния DataFrame. Този метод е полезен, когато искате да подредите данните според логическия ред на индекса.

### б) Пример 2: Сортиране по индекс на колони (`axis=1`)

```
# Създаваме DataFrame с не сортирани имена на колони
data_cols = {'C': [1, 2], 'A': [3, 4], 'B': [5, 6]}
df_unsorted_cols = pd.DataFrame(data_cols)
print("\nОригинален DataFrame с не сортирани колони:\n",
df_unsorted_cols)

# Сортиране по индекс на колоните във възходящ ред
df_sorted_cols_asc = df_unsorted_cols.sort_index(axis=1)
print("\nDataFrame, сортиран по индекс на колоните (възходящо):\n",
df_sorted_cols_asc)
```

*Тук виждаме как да сортираме DataFrame по азбучния ред на имената на колоните, като използваме `axis=1`.*

- 1) **Създава се DataFrame с не сортирани имена на колони:** Инициализира се Pandas DataFrame `df_unsorted_cols` с три колони, чиито имена ('C', 'A', 'B') не са в азбучен ред. Индексът на редовете е автоматично генериран числов индекс (0, 1). Оригиналният DataFrame с тези не сортирани имена на колони се извежда на конзолата.
- 2) **Сортиране по индекс на колоните (възходящо):** Методът `.sort_index(axis=1)` се използва с параметъра `axis=1`. Това указва, че сортирането трябва да се извърши по индекса на колоните (т.е., по имената на колоните). По подразбиране, `ascending` е `True`, така че колоните ще бъдат сортирани във възходящ (азбучен) ред. Резултатът е нов DataFrame `df_sorted_cols_asc`, който се извежда на конзолата, показващ колоните, подредени по име 'A', 'B', 'C'. Редът на редовете остава същият.

## Накратко:

Примерът показва, че `.sort_index()` може да се използва не само за сортиране по индекса на редовете (`axis=0`, което е по подразбиране), но и за сортиране по индекса на колоните (`axis=1`). Това е полезно, когато искате да подредите колоните на DataFrame-а по определен ред, например азбучен.

### в) Пример 3: Сортиране по ниво на MultiIndex

```
# Създаваме DataFrame с MultiIndex
index_multi = pd.MultiIndex.from_tuples([('Група1', 'A'), ('Група1',
'B'), ('Група2', 'A'), ('Група2', 'B')],
                                         names=['Група', 'Подгрупа'])

data_multi = {'Стойност': [10, 20, 30, 40]}
df_multi_index = pd.DataFrame(data_multi, index=index_multi)
print("\nОригинален DataFrame с MultiIndex:\n", df_multi_index)

# Сортиране по външното ниво ('Група') на индекса
df_sorted_level_group = df_multi_index.sort_index(level='Група')
print("\nDataFrame, сортиран по ниво 'Група' на индекса:\n",
df_sorted_level_group)

# Сортиране по вътрешното ниво ('Подгрупа') на индекса
df_sorted_level_subgroup = df_multi_index.sort_index(level='Подгрупа')
print("\nDataFrame, сортиран по ниво 'Подгрупа' на индекса:\n",
df_sorted_level_subgroup)

# Сортиране по двете нива в низходящ ред
```

```
df_sorted_levels_desc = df_multi_index.sort_index(level=['Група',
'Подгрупа'], ascending=[False, False])
print("\nDataFrame, сортиран по нива (низходящо):\n",
df_sorted_levels_desc)
```

*Този пример илюстрира как да сортираме DataFrame с MultiIndex по едно или няколко нива на индекса, като контролираме посоката на сортиране за всяко ниво.*

- 1) **Създава се DataFrame с MultiIndex:** Инициализира се Pandas DataFrame `df_multi_index` с MultiIndex за редовете. MultiIndex-ът има две нива: 'Група' (със стойности 'Група1', 'Група2') и 'Подгрупа' (със стойности 'A', 'B'). DataFrame-ът има една колона 'Стойност'. Оригиналният DataFrame с този MultiIndex се извежда на конзолата.
- 2) **Сортиране по външното ниво ('Група'):** Методът `.sort_index(level='Група')` се използва с параметъра `level='Група'`. Това указва, че DataFrame-ът трябва да бъде сортиран въз основа на етикетите на външното ниво на MultiIndex-а ('Група'). По подразбиране, `ascending` е `True`, така че групите ще бъдат сортирани във възходящ ред ('Група1', 'Група2'). В рамките на всяка група, подредбата на вътрешното ниво ('Подгрупа') се запазва (или се сортира, ако `sort_remaining=True`, което е по подразбиране). Резултатът е нов DataFrame `df_sorted_level_group`, който се извежда на конзолата, показващ редовете, групирани и сортирани първо по 'Група'.
- 3) **Сортиране по вътрешното ниво ('Подгрупа'):** Методът `.sort_index(level='Подгрупа')` се използва с параметъра `level='Подгрупа'`. Това сортира DataFrame-а въз основа на етикетите на вътрешното ниво на MultiIndex-а ('Подгрупа'). Редът на външното ниво ('Група') се запазва, но редовете в рамките на всяка група се сортират по 'Подгрупа' ('A', 'B'). Резултатът е нов DataFrame `df_sorted_level_subgroup`, който се извежда на конзолата, показващ редовете, сортирани по 'Подгрупа' в рамките на всяка 'Група'.
- 4) **Сортиране по двете нива в низходящ ред:** Методът `.sort_index(level=['Група', 'Подгрупа'], ascending=[False, False])` се използва с параметъра `level`, който приема списък от нива ['Група', 'Подгрупа'], и параметъра `ascending`, който приема списък от булеви стойности [False, False]. Това указва, че сортирането трябва да се извърши първо по ниво 'Група' в низходящ ред, а след това по ниво 'Подгрупа' също в низходящ ред (в рамките на всяка група). Резултатът е нов DataFrame `df_sorted_levels_desc`, който се извежда на конзолата, показващ редовете, сортирани по 'Група' ('Група2', 'Група1') и след това по 'Подгрупа' ('B', 'A') в рамките на всяка група.

## Накратко:

Примерът демонстрира как `.sort_index()` позволява гъвкаво сортиране на DataFrame-и с MultiIndex по едно или няколко нива на индекса. Параметърът `level` определя кои нива да се използват за сортиране, а параметърът `ascending` контролира посоката на сортиране за всяко ниво. Това е важно за организиране на данни с йерархична структура.

## 3. Обобщение на методите за сортиране:

- `.sort_values(by, axis=0, ascending=True, inplace=False, na_position='last')`:
  - Използва се за сортиране на DataFrame или Series по стойностите в една или повече колони (за DataFrame) или по самите стойности (за Series).
  - Основният параметър `by` указва кои колони да се използват за сортиране. Може да бъде единично име на колона или списък от имена.



- `axis=0` (по подразбиране) сортира редовете.
- `ascending=True` (по подразбиране) сортира във възходящ ред. За низходящ ред се използва `ascending=False`. Възможно е да се подаде списък от булеви стойности, съответстващ на колоните в `by`, за да се зададе различна посока на сортиране за всяка колона.
- `inplace=True` модифицира оригиналния обект, а `inplace=False` (по подразбиране) връща нов сортиран обект.
- `na_position` контролира позицията на липсващите стойности (`NaN`) при сортиране. 'last' (по подразбиране) поставя `NaN` в края, а 'first' ги поставя в началото.
- **`.sort_index(axis=0, level=None, ascending=True, inplace=False, sort_remaining=True)`:**
  - Използва се за сортиране на `DataFrame` или `Series` **по етикетите на техния индекс** (редовете или колоните).
  - `axis=0` (по подразбиране) сортира по индекса на редовете. За сортиране по индекса на колоните се използва `axis=1`.
  - `level` е важен при `MultiIndex` и указва кое ниво (или нива) на индекса да се използва за сортиране. Може да бъде име на ниво или пореден номер. Ако е `None`, се сортира по всички нива.
  - `ascending=True` (по подразбиране) сортира във възходящ ред на индекса. За низходящ ред се използва `ascending=False`. Възможно е да се подаде списък от булеви стойности, съответстващ на нивата в `level`, за да се зададе различна посока на сортиране за всяко ниво.
  - `inplace=True` модифицира оригиналния обект, а `inplace=False` (по подразбиране) връща нов сортиран обект.
  - `sort_remaining=True` (по подразбиране) сортира и останалите нива на `MultiIndex` в рамките на групите, определени от сортираното ниво.

## 4. Насоки при използване на сортиране в *Pandas*:

- **Изберете правилния метод:**
  - Използвайте `.sort_values()`, когато искате да подредите данните си въз основа на стойностите в една или повече колони. Това е най-често използваният метод за сортиране на `DataFrame`.
  - Използвайте `.sort_index()`, когато искате да подредите данните си въз основа на етикетите на индекса. Това е полезно, когато индексът има логически ред (например, времеви серии, азбучни етикети) или когато работите с `MultiIndex`.
- **Определете критериите за сортиране:**
  - За `.sort_values()`, внимателно изберете колоните, по които искате да сортирате (`by`). Редът на колоните в списъка `by` определя приоритета на сортиране.
  - За `.sort_index()`, определете по коя ос (`axis`) и по кое ниво (`level`, при `MultiIndex`) искате да сортирате.
- **Контролирайте посоката на сортиране:** Използвайте параметъра `ascending` (единична булева стойност или списък) за да зададете желаната посока на сортиране (възходяща или низходяща).
- **Управлявайте липсващите стойности:** Когато използвате `.sort_values()`, решете къде искате да бъдат поставени `NaN` стойностите с помощта на `na_position`.
- **Решете дали да сортирате на място:** Обмислете дали искате да модифицирате оригиналния `DataFrame` (`inplace=True`) или да създадете нов сортиран `DataFrame` (`inplace=False`).



Препоръчително е да се избягва `inplace=True`, освен ако не сте сигурни, че нямате нужда от оригиналния `DataFrame`.

- **Игнорирайте стария индекс при сортиране по стойности:** Ако след сортиране по стойности искате да имате нов последователен числов индекс, използвайте `ignore_index=True` в `.sort_values()`.

Чрез разбирането и правилното използване на тези методи и техните параметри, можете ефективно да организирате данните си в Pandas за по-лесен анализ и интерпретация.

## Казус 1: Сортиране на продукти по цена и наличност (`.sort_values()` по няколко колони)

Представете си, че имате `DataFrame` със списък на продукти, техните цени и наличност на склад. Искате да сортирате продуктите първо по цена (възходящо), а след това по наличност (низходящо), за да видите най-евтините продукти и сред тях тези с най-голяма наличност.

```
import pandas as pd

# Създаваме DataFrame със списък на продукти
products_data = {
    'продукт': ['Телевизор', 'Лаптоп', 'Мишка', 'Клавиатура',
               'Монитор'],
    'цена': [500.00, 1200.00, 25.00, 40.00, 25.00],
    'наличност': [5, 10, 50, 20, 30]
}

df_products = pd.DataFrame(products_data)
print("Оригинален DataFrame:\n", df_products)

# Казус: Сортирайте DataFrame-а първо по 'цена' (възходящо), а след това
по 'наличност' (низходящо).
```

## Решение на Казус 1:

```
# Сортиране по няколко колони с различни посоки
df_sorted_products = df_products.sort_values(by=['цена', 'наличност'],
                                             ascending=[True, False])
```

```
print("\nDataFrame, сортиран по цена (възходящо) и наличност  
(низходящо): \n", df_sorted_products)
```

## Казус 2: Сортиране на времеви данни по дата (`.sort_values()` по една колона с `na_position`)

Представете си, че имате `DataFrame` с времеви данни, където някои от датите може да липсват (`NaN`). Искате да сортирате данните по дата, като липсващите дати бъдат поставени в началото.

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с времеви данни и липсващи стойности
time_series_data = {
    'дата': pd.to_datetime(['2023-01-05', np.nan, '2023-01-01', '2023-01-10', np.nan]),
    'стойност': [10, 20, 15, 25, 30]
}
df_time_series = pd.DataFrame(time_series_data)
print("Оригинален DataFrame с липсващи дати: \n", df_time_series)

# Казус: Сортирайте DataFrame-а по колона 'дата', като липсващите стойности (NaN) бъдат в началото.
```

## Решение на Казус 2:

```
# Сортиране по колона с липсващи стойности, поставяйки ги в началото
df_sorted_time_series = df_time_series.sort_values(by='дата',
na_position='first')
print("\nDataFrame, сортиран по дата (NaN в началото): \n",
df_sorted_time_series)
```

### Казус 3: Сортиране на `Series` по неговите стойности (`.sort_values()` за `Series`)

Представете си, че имате `Series`, съдържащ броя на гласовете за различни кандидати. Искате да ги сортирате в низходящ ред, за да видите кой е получил най-много гласове.

```
import pandas as pd

# Създаваме Series с гласове за кандидати
votes = pd.Series([1200, 850, 1500, 920], index=['Алекс', 'Борис', 'Вера', 'Георги'])
print("Оригинален Series с гласове:\n", votes)

# Казус: Сортирайте Series-а по броя на гласовете в низходящ ред.
```

### Решение на Казус 3:

```
# Сортиране на Series по стойности
sorted_votes = votes.sort_values(ascending=False)
print("\nSeries, сортиран по гласове (низходящо):\n", sorted_votes)
```

### Казус 4: Сортиране на `DataFrame` по индекс (`.sort_index()` по ос)

Представете си, че имате `DataFrame` с продукти като колони и дати като индекс. Искате да сортирате `DataFrame`-а по дати (индекса).

```
import pandas as pd

# Създаваме DataFrame с дати като индекс
sales_over_time = pd.DataFrame({
    'Телевизор': [10, 12, 15],
    'Лаптоп': [5, 8, 7]
}, index=pd.to_datetime(['2023-01-05', '2023-01-01', '2023-01-10']))
print("Оригинален DataFrame с времеви индекс:\n", sales_over_time)

# Казус: Сортирайте DataFrame-а по неговия индекс (дата) във възходящ ред.
```

## Решение на Казус 4:

```
# Сортиране по индекс
sorted_sales_over_time = sales_over_time.sort_index()
print("\nDataFrame, сортиран по индекс (дата):\n",
      sorted_sales_over_time)
```

## Казус 5: Сортиране на DataFrame с мултииндекс (.sort\_index() по ниво)

Представете си, че имате DataFrame с мултииндекс (например, регион и продукт) и искате да го сортирате по конкретно ниво на индекса (например, по продукт в рамките на всеки регион).

```
import pandas as pd

# Създаваме DataFrame с мултииндекс
multi_indexed_sales = pd.DataFrame({
    'продажби': [100, 120, 150, 130, 90, 110]
}, index=pd.MultiIndex.from_tuples([
    ('Север', 'Телевизор'), ('Север', 'Лаптоп'),
    ('Юг', 'Телевизор'), ('Юг', 'Лаптоп'),
    ('Изток', 'Телевизор'), ('Изток', 'Лаптоп')
], names=['регион', 'продукт']))
print("Оригинален DataFrame с мултииндекс:\n", multi_indexed_sales)

# Казус: Сортирайте DataFrame-а по ниво 'продукт' (възходящо) в рамките
на всеки 'регион'.
```

## Решение на Казус 5:

```
# Сортиране по ниво на мултииндекс
sorted_multi_indexed_sales =
multi_indexed_sales.sort_index(level='продукт')
```

```
print("\nDataFrame, сортиран по ниво 'продукт':\n",
sorted_multi_indexed_sales)
```

Тези казуси илюстрират как `.sort_values()` позволява гъвкаво сортиране на `DataFrame` и `Series` по техните стойности, като се контролира по кои колони да се сортира, посоката на сортиране и как да се обработват липсващите стойности. От друга страна, `.sort_index()` е полезен за сортиране на данни въз основа на техния индекс, като позволява сортиране по различни нива при мултииндекси и определяне на посоката на сортиране.

## XII. Промяна на типа данни на колони (`.astype()` , `pd.to_numeric`, `pd.to_datetime`, `pd.to_timedelta`).

Често при работа с данни се сблъскваме с колони, които имат неправилен тип данни. Например, числови стойности могат да бъдат прочетени като низове, или дати могат да бъдат представени като обекти (което може да затрудни аритметичните операции с дати). Pandas предоставя няколко мощни функции и методи за преобразуване на типа данни на колони в `DataFrame` и `Series`.

Основните инструменти за тази цел са:

- `.astype()`: Гъвкав метод за явно преобразуване на типа данни.
- `pd.to_numeric()`: Специализирана функция за преобразуване на колони в числов тип.
- `pd.to_datetime()`: Специализирана функция за преобразуване на колони в тип `datetime` (за работа с дати и часове).
- `pd.to_timedelta()`: Специализирана функция за преобразуване на колони в тип `timedelta` (за работа с времеви разлики).

Разбирането на тези инструменти и кога да ги използваме е ключово за правилното манипулиране и анализ на данни в Pandas. Нека разгледаме всеки от тях по-подробно.

### 1. `.astype()`:

Методът `.astype()` е един от най-често използваните и гъвкави начини за явно преобразуване на типа данни на `Series` или една или повече колони в `DataFrame`. Той приема като аргумент типа данни, към който искаме да преобразуваме.

## Синтаксис:

За `Series`:

```
series.astype(dtype, copy=True, errors='raise')
```

За DataFrame (прилагане към една колона):

```
df['име_на_колона'].astype(dtype, copy=True, errors='raise')
```

За DataFrame (прилагане към множество колони - връща нов DataFrame):

```
df.astype(dtype, copy=True, errors='raise') # където dtype е речник
```

## Параметри:

- **dtype:** Типът данни, към който искаме да преобразуваме. Може да бъде:
  - Python вградени типове (напр., int, float, str, bool).
  - NumPy типове данни (напр., np.int64, np.float64, np.datetime64[ns]).
  - Pandas специфични типове данни (напр., 'category', 'datetime64[ns]', 'timedelta64[ns]').
- **copy:** Булев параметър. Ако е True (по подразбиране), се връща ново копие на данните. Ако е False, се опитва да върне изглед (view), ако е възможно.
- **errors:** Стринг, контролиращ обработката на грешки при преобразуване. Възможни стойности:
  - 'raise' (по подразбиране): Ако преобразуването не е валидно, се хвърля изключение (TypeError или ValueError).
  - 'ignore': Ако преобразуването не е валидно за дадена стойност, грешката се игнорира и стойността остава непроменена.

### а) Пример 1: Преобразуване на колона от низ в числов тип (int)

```
import pandas as pd

# Създаваме DataFrame с колона от низови числа
data = {'Числа_като_низове': ['10', '20', '30']}
df_strings = pd.DataFrame(data)
print("Оригинален DataFrame (тип данни на колоната):",
      df_strings.dtypes)
print(df_strings)

# Преобразуваме колоната в целочислен тип
df_strings['Числа_като_низове'] =
df_strings['Числа_като_низове'].astype(int)
print("\nDataFrame след преобразуване (тип данни на колоната):",
      df_strings.dtypes)
print(df_strings)
```

**В този пример преобразуваме колона, съдържаща низови представления на числа, в целочислен тип (int). Това позволява да извършваме аритметични операции с тези стойности.**

Разбира се, ето резюме на предоставения пример, който демонстрира преобразуването на колона от тип `object` (съдържаща низове, които представляват числа) в целочислен тип (`int`) с помощта на метода `.astype()` в Pandas:

- 1) **Създава се DataFrame с колона от низови числа:** Инициализира се Pandas `DataFrame` `df_strings` с една колона, наречена 'Числа\_като\_низове'. Тази колона съдържа низови стойности ('10', '20', '30'), които представляват цели числа. Типът данни на тази колона е `object`, тъй като Pandas интерпретира списъка от низове като обекти. Оригиналният `DataFrame` и типът данни на колоната се извеждат на конзолата.
- 2) **Преобразуване на колоната в целочислен тип:** Методът `.astype(int)` се прилага към колоната 'Числа\_като\_низове'. Това указва, че искаме да преобразуваме стойностите в тази колона към целочислен тип (`int`). Тъй като всички низове в колоната могат успешно да бъдат интерпретирани като цели числа, преобразуването е успешно. Резултатът е, че типът данни на колоната се променя от `object` на `int64` (или друга подходяща целочислена разновидност в зависимост от платформата). Обновеният `DataFrame` и новият тип данни на колоната се извеждат на конзолата.

## Накратко:

Примерът показва основното използване на `.astype()` за преобразуване на колона с низови числа в числов тип, което е често срещана операция при почистване и подготовка на данни за анализ. След преобразуването, с числовите стойности могат да се извършват математически операции.

### *б) Пример 2: Преобразуване на колона в булев тип (bool)*

```
import pandas as pd

# Създаваме DataFrame с колона, която може да бъде интерпретирана като
# булева
data_bool_like = {'Стойности': [0, 1, 1, 0, 1]}
df_bool_like = pd.DataFrame(data_bool_like)
print("Оригинален DataFrame (тип данни на колоната):",
      df_bool_like.dtypes)
print(df_bool_like)

# Преобразуваме колоната в булев тип
df_bool_like['Стойности'] = df_bool_like['Стойности'].astype(bool)
print("\nDataFrame след преобразуване (тип данни на колоната):",
      df_bool_like.dtypes)
print(df_bool_like)
```



*Тук показваме как числови стойности (0 и 1) могат да бъдат преобразувани в булев тип (bool), където 0 става False, а всяко друго число (включително 1) става True.*

- 1) Създава се `DataFrame` с колона, която може да бъде интерпретирана като булева: Инициализира се `Pandas DataFrame df_bool_like` с една колона, наречена 'Стойности'. Тази колона съдържа целочислени стойности (0 и 1). Типът данни на тази колона е `int64`. Оригиналният `DataFrame` и типът данни на колоната се извеждат на конзолата.
- 2) Преобразуване на колоната в булев тип: Методът `.astype(bool)` се прилага към колоната 'Стойности'. При преобразуване към булев тип, `Pandas` интерпретира стойността 0 като `False`, а всяка друга ненулева стойност (включително 1) като `True`. Резултатът е, че типът данни на колоната се променя от `int64` на `bool`. Обновеният `DataFrame` и новият тип данни на колоната се извеждат на конзолата, показвайки `True` и `False` стойности.

## Накратко:

Примерът показва как `.astype(bool)` може да се използва за преобразуване на числови колони в булев тип, което е полезно за логически операции и филтриране на данни въз основа на истинностни стойности.

### *в) Пример 3: Преобразуване на колона в категориален тип (category)*

Категориалният тип данни е полезен за колони с ограничен брой уникални стойности, особено ако тези стойности се повтарят често. Той може да подобри производителността и да намали използването на памет.

```
import pandas as pd

# Създаваме DataFrame с колона от низови категории
data_categories = {'Категория': ['A', 'B', 'A', 'C', 'B', 'A']}
df_categories = pd.DataFrame(data_categories)
print("Оригинален DataFrame (тип данни на колоната):",
      df_categories.dtypes)
print(df_categories)

# Преобразуваме колоната в категориален тип
df_categories['Категория'] =
df_categories['Категория'].astype('category')
print("\nDataFrame след преобразуване (тип данни на колоната):",
      df_categories.dtypes)
print(df_categories)
```

*В този случай, колоната 'Категория', съдържаща повтарящи се низови стойности, се преобразува в категориален тип.*

- 1) **Създава се DataFrame с колона от низови категории:** Инициализира се Pandas DataFrame `df_categories` с една колона, наречена 'Категория'. Тази колона съдържа низови стойности ('A', 'B', 'A', 'C', 'B', 'A'), представляващи различни категории. Типът данни на тази колона е `object`. Оригиналният DataFrame и типът данни на колоната се извеждат на конзолата.
- 2) **Преобразуване на колоната в категориален тип:** Методът `.astype('category')` се прилага към колоната 'Категория'. Категориалният тип данни е полезен за колони с ограничен брой уникални стойности, които се повтарят често. Той може да подобри производителността и да намали използването на памет, тъй като Pandas вътрешно представя тези стойности като числови кодове и поддържа речник на уникалните категории. След преобразуването, типът данни на колоната се променя на `category`. Обновеният DataFrame и новият тип данни на колоната се извеждат на конзолата. Забележете, че изходът може да покаже и уникалните категории.

## Накратко:

Примерът показва как `.astype('category')` може да се използва за преобразуване на низови колони с повтарящи се стойности в категориален тип, което е ефективен начин за оптимизиране на паметта и потенциално ускоряване на някои операции при анализ на данни.

### *г) Пример 4: Опит за преобразуване към невалиден тип данни (грешка)*

Ако опитаме да преобразуваме колона към тип данни, който не е съвместим със съдържанието ѝ, `.astype()` ще хвърли грешка (ако `errors='raise'`, което е по подразбиране).

```
import pandas as pd

# Създаваме DataFrame с колона, която не може да бъде директно
# преобразувана в int
data_invalid_int = {'Смесени_стойности': ['10', 'hello', '20']}
df_invalid_int = pd.DataFrame(data_invalid_int)
print("Оригинален DataFrame:\n", df_invalid_int)

# Опит за преобразуване към int (ще доведе до ValueError)
try:
    df_invalid_int['Смесени_стойности'] =
df_invalid_int['Смесени_стойности'].astype(int)
except ValueError as e:
    print(f"\nВъзникна грешка при преобразуване към int: {e}")
```

*Този пример показва, че опитът за преобразуване на низ 'hello' към целочислен тип води до ValueError.*

- 1) Създава се `DataFrame` с колона, която не може да бъде директно преобразувана в `int`: Инициализира се `Pandas DataFrame df_invalid_int` с една колона, наречена 'Смесени\_стойности'. Тази колона съдържа низови стойности ('10', 'hello', '20'), като една от тях ('hello') не може да бъде интерпретирана като цяло число. Оригиналният `DataFrame` се извежда на конзолата.
- 2) **Опит за преобразуване към `int` (ще доведе до `ValueError`):** Блокът `try...except` се опитва да приложи метода `.astype(int)` към колоната 'Смесени\_стойности'. Тъй като колоната съдържа низ ('hello'), който не може да бъде преобразуван към целочислен тип, `Pandas` хвърля изключение от тип `ValueError`.
- 3) **Обработване на грешката:** Блокът `except ValueError as e:` улавя възникналата `ValueError`. След това се извежда съобщение на конзолата, което съдържа описание на грешката (`e`), което обикновено указва, че невалидна буквална стойност е била подадена за целочислено преобразуване.

## Накратко:

Примерът показва, че когато се използва `.astype()` за преобразуване към конкретен тип данни и колоната съдържа стойности, които не могат да бъдат интерпретирани като този тип, `Pandas` по подразбиране хвърля грешка (`ValueError`). Това подчертава важността на предварителното почистване и валидиране на данните преди опит за преобразуване на типа им.

### д) Пример 5: Използване на `errors='ignore'`

Ако зададем `errors='ignore'`, `Pandas` ще пропусне редовете, които не могат да бъдат преобразувани, и ще запази оригиналния тип данни за тези стойности.

```
import pandas as pd

# Използваме DataFrame от предходния пример
data_invalid_int = {'Смесени_стойности': ['10', 'hello', '20']}
df_invalid_int = pd.DataFrame(data_invalid_int)
print("Оригинален DataFrame:\n", df_invalid_int)

# Опит за преобразуване към int с игнориране на грешки
df_invalid_int['Смесени_стойности'] =
df_invalid_int['Смесени_стойности'].astype(int, errors='ignore')
print("\nDataFrame след опит за преобразуване (грешките са
игнорирани):\n", df_invalid_int)
print("Тип данни на колоната:", df_invalid_int.dtypes)
```

*В този случай, низът 'hello' не може да бъде преобразуван към int, и тъй като errors='ignore', стойността остава непроменена, а типът данни на колоната вероятно ще остане object (или ще бъде повишен до тип, който може да побере всички стойности).*

Разбира се, ето резюме на предоставения пример, който демонстрира използването на параметъра errors='ignore' в метода .astype() при опит за преобразуване на колона към целочислен тип (int) в Pandas, когато колоната съдържа невалидни за преобразуване стойности:

- 1) **Използва се DataFrame от предходния пример:** Инициализира се Pandas DataFrame df\_invalid\_int с една колона, наречена 'Смесени\_стойности', която съдържа низови стойности ('10', 'hello', '20'), включваща невалидна за целочислено преобразуване стойност ('hello'). Оригиналният DataFrame се извежда на конзолата.
- 2) **Опит за преобразуване към int с игнориране на грешки:** Методът .astype(int, errors='ignore') се прилага към колоната 'Смесени\_стойности'. Параметърът errors='ignore' указва на Pandas да не хвърля грешка (ValueError), когато срещне стойност, която не може да бъде успешно преобразувана към посочения тип данни (int). Вместо това, Pandas запазва оригиналната стойност (в случая 'hello') в колоната и продължава с останалите стойности.
- 3) **Извежда се DataFrame след опита за преобразуване:** Обновеният DataFrame се извежда на конзолата. Забелязва се, че стойностите, които могат да бъдат преобразувани към int ('10', '20'), са останали като низове, а не са станали числа. Това е така, защото когато errors='ignore' е зададено и има поне една стойност, която не може да бъде преобразувана към целевия тип, Pandas обикновено запазва оригиналния тип данни на колоната (в този случай object) за всички елементи, за да може да побере и непреобразуваните стойности. Типът данни на колоната също се извежда, което ще покаже, че колоната най-вероятно все още е от тип object.

## Накратко:

Примерът показва, че използването на errors='ignore' в .astype() позволява да се избегне грешка при опит за преобразуване на колона с невалидни стойности. Въпреки това, важно е да се разбере, че непреобразуваните стойности ще запазят оригиналния си тип данни, което може да не е желаното поведение за последващ анализ, който изисква конкретен тип данни. В такива случаи е препоръчително предварително да се почистят или обработят невалидните стойности.

Методът .astype() е мощен инструмент за промяна на типа данни, но е важно да се уверите, че преобразуването е валидно за съдържанието на колоната. В противен случай ще възникне грешка (по подразбиране) или преобразуването ще бъде игнорирано (errors='ignore'), което може да доведе до неочаквани резултати при последващ анализ.

## 2. pd.to\_numeric

Функцията pd.to\_numeric() е изключително полезна за надеждно преобразуване на Series (едноколонен обект) в числов тип данни. Тя предлага по-голям контрол върху обработката на грешки и непреобразуваеми стойности в сравнение с .astype().

**Синтаксис:**

```
pd.to_numeric(arg, errors='raise', downcast=None)
```

## Параметри:

- **arg**: Series, list-подобен обект, или скаларна стойност, която искате да преобразувате. Най-често се използва за преобразуване на колона от DataFrame.
- **errors**: Стринг, контролиращ обработката на грешки при преобразуване. Възможни стойности:
  - 'raise' (по подразбиране): Ако преобразуването не е валидно, се хвърля изключение (ValueError).
  - 'coerce': Ако преобразуването не е валидно, невалидните стойности се заменят с NaN (Not a Number), което е стандартното представяне на липсващи числови данни в Pandas.
  - 'ignore': Ако преобразуването не е валидно, грешката се игнорира и оригиналните непреобразувани стойности се запазват (резултатът ще бъде Series с оригиналния тип данни).
- **downcast**: Стринг, указващ до какъв по-малък числов тип (ако е възможно) да се извърши преобразуването с цел оптимизация на паметта. Възможни стойности са 'integer', 'signed', 'unsigned', 'float'. Например, ако всички числа могат да бъдат представени като 32-битови цели числа, downcast='integer' ще ги преобразува до int32.

### а) Пример 1: Преобразуване на Series от низови числа към числов тип (float)

```
import pandas as pd

# Създаваме Series от низови числа
string_numbers = pd.Series(['1.5', '2', '-3.7', '4'])
print("Оригинален Series (тип данни):", string_numbers.dtype)
print(string_numbers)

# Преобразуваме Series към числов тип (float по подразбиране)
numeric_series = pd.to_numeric(string_numbers)
print("\nПреобразуван Series (тип данни):", numeric_series.dtype)
print(numeric_series)
```

В този пример, Series от низови числа се преобразува към числов тип float64 (по подразбиране, ако не е указан downcast).

- 1) **Създава се Series от низови числа:** Инициализира се Pandas Series string\_numbers с четири елемента, които са низови представяния на числа (включително десетични и отрицателни). Типът данни на този Series е object, тъй като Pandas интерпретира списъка от низове като обекти. Оригиналният Series и неговият тип данни се извеждат на конзолата.
- 2) **Преобразуване към числов тип:** Функцията pd.to\_numeric(string\_numbers) се използва за преобразуване на елементите на string\_numbers към числов тип. По подразбиране, ако не е указан параметърът downcast, pd.to\_numeric() ще се опита да преобразува към най-подходящия числов тип, който може да побере всички стойности. В този случай, тъй като има десетични числа, резултатът ще бъде float64.

- 3) **Извежда се преобразуваният series:** Резултатът е нов Series `numeric_series`, който съдържа същите числови стойности, но вече с тип данни `float64`. Преобразуваният Series и неговият тип данни се извеждат на конзолата.

## Накратко:

Примерът показва как `pd.to_numeric()` може лесно да преобразува Series от низови числа в Series с числов тип данни, което е необходимо за извършване на математически операции и анализ на данните. За разлика от `.astype()`, `pd.to_numeric()` е по-гъвкав при обработката на невалидни стойности чрез параметъра `errors`.

### б) Пример 2: Обработка на невалидни стойности с `errors='coerce'`

```
import pandas as pd

# Създаваме Series, съдържащ невалидна за числово преобразуване стойност
mixed_series = pd.Series(['10', 'invalid', '20.5', '-5'])
print("Оригинален Series:\n", mixed_series)

# Преобразуваме с errors='coerce'
coerced_series = pd.to_numeric(mixed_series, errors='coerce')
print("\nПреобразуван Series (errors='coerce'):\n", coerced_series)
```

Тук, низът `'invalid'` не може да бъде преобразуван към число и е заменен с `NaN` в резултата. Типът данни на резултата ще бъде `float64`, тъй като `NaN` е *floating-point* стойност.

Разбира се, ето резюме на предоставения пример, който демонстрира използването на параметъра `errors='coerce'` във функцията `pd.to_numeric()` за обработка на невалидни за числово преобразуване стойности в Pandas Series:

- 1) **Създава се series, съдържащ невалидна стойност:** Инициализира се Pandas Series `mixed_series` с четири елемента, като един от тях (`'invalid'`) не може да бъде преобразуван към числов тип. Оригиналният Series се извежда на конзолата.
- 2) **Преобразуване с `errors='coerce'`:** Функцията `pd.to_numeric()` се използва с аргумента `errors='coerce'`. Когато тази опция е зададена, Pandas се опитва да преобразува всеки елемент от Series-а към числов тип. Ако преобразуването е успешно, елементът се превръща в число (най-вероятно `float64`, за да побере всички възможни резултати, включително `NaN`). Ако преобразуването е неуспешно (както е в случая с `'invalid'`), невалидната стойност се заменя със специалната *floating-point* стойност `NaN` (Not a Number), която Pandas използва за представяне на липсващи числови данни.
- 3) **Извежда се преобразуваният series:** Резултатът е нов Series `coerced_series`, където валидните числови низове са преобразувани в числа (вероятно `float64`), а невалидната стойност `'invalid'` е заменена с `NaN`. Преобразуваният Series се извежда на конзолата.

## Накратко:



Примерът показва, че `errors='coerce'` е полезен начин за справяне с нечислови данни, които могат да присъстват в колони, които трябва да бъдат от числов тип. Чрез принудителното преобразуване на невалидните стойности в `NaN`, можем да продължим с числови анализи, като тези липсващи стойности могат да бъдат обработени по-късно (например, чрез запълване или премахване).

### в) Пример 3: Игнориране на грешки с `errors='ignore'`

```
import pandas as pd

# Използваме Series от предходния пример
mixed_series = pd.Series(['10', 'invalid', '20.5', '-5'])
print("Оригинален Series:\n", mixed_series)

# Преобразуваме с errors='ignore'
ignored_series = pd.to_numeric(mixed_series, errors='ignore')
print("\nПреобразуван Series (errors='ignore'):\n", ignored_series)
print("Тип данни:", ignored_series.dtype)
```

В този случай, непреобразуваемият низ `'invalid'` е запазен в резултата, и типът данни на Series-а остава `object`.

- 1) **Използва се Series от предходния пример:** Инициализира се Pandas Series `mixed_series` с четири елемента, като един от тях (`'invalid'`) не може да бъде преобразуван към числов тип. Оригиналният Series се извежда на конзолата.
- 2) **Преобразуване с `errors='ignore'`:** Функцията `pd.to_numeric()` се използва с аргумента `errors='ignore'`. Когато тази опция е зададена, Pandas се опитва да преобразува всеки елемент от Series-а към числов тип. Ако преобразуването е успешно, елементът се превръща в число. Ако преобразуването е неуспешно (както е в случая с `'invalid'`), Pandas **игнорира грешката** и запазва оригиналната стойност в Series-а.
- 3) **Извежда се преобразуваният Series и неговият тип данни:** Резултатът е нов Series `ignored_series`, който съдържа оригиналните стойности. Тъй като има поне една стойност (`'invalid'`), която не може да бъде преобразувана към числов тип, типът данни на целия Series остава `object`. Това е така, защото Pandas трябва да използва най-общия тип данни, който може да побере всички стойности в Series-а, включително непреобразуваните. Типът данни на `ignored_series` (`ignored_series.dtype`) също се извежда, което ще покаже `object`.

## Накратко:

Примерът показва, че `errors='ignore'` предотвратява хвърлянето на грешка при срещане на невалидни за числово преобразуване стойности. Въпреки това, той не извършва никакво преобразуване на тези невалидни стойности и запазва оригиналния тип данни на Series-а, ако има такива стойности. Тази опция може да бъде полезна, когато искате да избегнете прекъсване на кода, но е важно да се има предвид, че колоната може да не бъде от желания числов тип след тази операция.

### г) Пример 4: Оптимизация на паметта с `downcast`

```
import pandas as pd
```



```

# Създаваме Series от цели числа (като низове)
integer_strings = pd.Series(['10', '20', '30', '40'])
print("Оригинален Series (dtype):", integer_strings.dtype)
print(integer_strings)

# Преобразуваме към числов тип и downcast към 'integer'
downcasted_series = pd.to_numeric(integer_strings, downcast='integer')
print("\nПреобразуван Series (downcast='integer', dtype):",
downcasted_series.dtype)
print(downcasted_series)

# Създаваме Series от числа, които могат да бъдат представени като
float32
float_numbers = pd.Series([1.1, 2.5, 3.9, 4.0])
print("\nОригинален Series (dtype):", float_numbers.dtype)
print(float_numbers)

# Преобразуваме и downcast към 'float'
downcasted_float_series = pd.to_numeric(float_numbers, downcast='float')
print("\nПреобразуван Series (downcast='float', dtype):",
downcasted_float_series.dtype)
print(downcasted_float_series)

```

*Тези примери показват как параметърът `downcast` може да бъде използван за преобразуване към по-специфичен (и често по-ефективен по отношение на паметта) числов тип, ако данните го позволяват.*

**1) Преобразуване към числов тип и `downcast='integer'`:**

- Създава се Series `integer_strings` от низови представяния на цели числа. Типът данни е `object`.
- Функцията `pd.to_numeric()` се използва с `downcast='integer'`. Това първо преобразува низовете към числов тип, а след това се опитва да ги преобразува към най-малкия възможен целочислен тип, който може да побере всички стойности без загуба на информация (например `int8`, `int16`, `int32` или `int64`). В този случай, тъй като стойностите са малки, вероятно ще бъде избран по-малък целочислен тип като `int8` или `int16`. Резултатът е `downcasted_series` с оптимизиран целочислен тип данни.

**2) Преобразуване и `downcast='float'`:**

- Създава се Series `float_numbers` от десетични числа. Типът данни е `float64` (по подразбиране за десетични числа).

- Функцията `pd.to_numeric()` се използва с `downcast='float'`. Това се опитва да преобразува числата към най-малкия възможен floating-point тип, който може да ги представи без значителна загуба на точност (например `float32` или `float64`). В този случай, ще бъде направен опит за преобразуване към `float32`, което използва по-малко памет от `float64`. Резултатът е `downcasted_float_series` с потенциално по-малък floating-point тип данни.

## Накратко:

Примерът показва как параметърът `downcast` в `pd.to_numeric()` може да бъде използван за автоматично преобразуване на числови данни към по-компактни типове данни (`integer` или `float`), ако това е възможно без загуба на информация или значителна точност. Това е важна техника за оптимизиране на използването на памет, особено при работа с големи набори от данни. Pandas автоматично избира най-подходящия подтип въз основа на диапазона на стойностите.

Функцията `pd.to_numeric()` е предпочитан начин за преобразуване към числов тип в Pandas, особено когато има вероятност за наличие на невалидни стойности, тъй като позволява гъвкаво управление на тези ситуации чрез параметъра `errors`. Параметърът `downcast` предлага допълнителна възможност за оптимизация на паметта.

## 3. `pd.to_datetime()`

Функцията `pd.to_datetime()` е специализирана за преобразуване на аргументи, които могат да бъдат разбрани като дати и часове, в Pandas `datetime` обекти. Това е от съществено значение за работа с времеви серии и за извършване на операции, свързани с дати и часове (например, извличане на ден от седмицата, изчисляване на времеви разлики, филтриране по времеви интервали).

### Синтаксис:

```
pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False,
               utc=None, format=None, exact=True, unit=None,
               infer_datetime_format=False, origin='unix', cache=True)
```

### Основни параметри:

- **arg**: Series, DataFrame (една колона), списък-подобен обект, NumPy array, или скаларна стойност, която искате да преобразувате.
- **errors**: Стринг, контролиращ обработката на грешки при преобразуване. Възможни стойности са `'raise'` (по подразбиране), `'coerce'` (невалидните стойности се заменят с `NaT` - Not a Time), и `'ignore'` (невалидните стойности се запазват).
- **format**: Стринг, указващ формата на входните дати и часове. Ако не е зададен, Pandas се опитва автоматично да го определи (което понякога може да е бавно или да не работи правилно).

Задаването на формат може значително да ускори процеса и да гарантира правилното парсване. Използват се директиви за форматиране, подобни на тези в Python (`strftime` / `strptime`).

- **dayfirst**: Булев параметър. Ако е `True`, приема се, че първата част от датата е денят (например, `10/11/2023` се интерпретира като 10 ноември, а не 11 октомври). По подразбиране е `False`.
- **yearfirst**: Булев параметър. Ако е `True`, приема се, че първата част от датата е годината. По подразбиране е `False`.
- **utc**: Булев параметър. Ако е `True`, резултатът се нормализира до UTC часова зона.
- **unit**: Стринг, указващ мерната единица (например, `'s'`, `'ms'`, `'us'`, `'ns'`) за числови входни данни, които представляват брой единици от началото на епохата (определена от `origin`).
- **origin**: Скаларна стойност, указваща епохата за числови входни данни. По подразбиране е `'unix'` (1970-01-01).

#### *a) Пример 1: Преобразуване на Series от низове с дати към datetime min*

```
import pandas as pd

# Създаваме Series от низове, представляващи дати
date_strings = pd.Series(['2023-10-26', '2023/11/15', '12-01-2024'])
print("Оригинален Series (dtype):", date_strings.dtype)
print(date_strings)

# Преобразуваме Series към datetime тип (автоматично разпознаване на формата)
datetime_series = pd.to_datetime(date_strings)
print("\nПреобразуван Series (dtype):", datetime_series.dtype)
print(datetime_series)
```

*В този пример, `pd.to_datetime()` автоматично разпознава различните формати на датите и ги преобразува в datetime обекти.*

Разбира се, ето резюме на предоставения пример, който демонстрира основното използване на функцията `pd.to_datetime()` за преобразуване на Pandas Series, съдържащ низови представяния на дати, към datetime тип данни:

- 1) **Създава се Series от низове с дати:** Инициализира се Pandas Series `date_strings` с три елемента, които са низови представяния на дати в различни формати (`'YYYY-MM-DD'`, `'YYYY/MM/DD'`, `'DD-MM-YYYY'`). Типът данни на този Series е `object`. Оригиналният Series и неговият тип данни се извеждат на конзолата.
- 2) **Преобразуване към datetime тип (автоматично разпознаване на формата):** Функцията `pd.to_datetime(date_strings)` се използва за преобразуване на елементите на `date_strings` към datetime обекти. В този случай, Pandas се опитва автоматично да разпознае формата на всеки низ и да го парсне като дата. Ако форматът е стандартен или често срещан, Pandas обикновено успява да го определи правилно.

- 3) **Извежда се преобразуваният Series:** Резултатът е нов Series `datetime_series`, който съдържа същите дати, но вече с тип данни `datetime64[ns]`. Преобразуваният Series и неговият тип данни се извеждат на конзолата.

## Накратко:

Примерът показва, че `pd.to_datetime()` може да бъде много удобна за бързо преобразуване на колони или Series-и, съдържащи дати в различни често срещани формати, без да е необходимо изрично задаване на формата. Въпреки това, за по-нестандартни или нееднородни формати, е препоръчително да се използва параметърът `format` за по-надеждно преобразуване.

### *б) Пример 2: Задаване на формат с параметъра `format`*

Когато форматът на датите е консистентен, задаването му може да ускори процеса на парсане и да избегне грешки при автоматичното разпознаване.

```
import pandas as pd

# Създаваме Series с дати в специфичен формат
date_series_specific_format = pd.Series(['10-26-2023', '11-15-2023',
'01-12-2024'])

print("Оригинален Series:\n", date_series_specific_format)

# Преобразуваме с указване на формата '%m-%d-%Y' (месец-ден-година)
datetime_series_formatted = pd.to_datetime(date_series_specific_format,
format='%m-%d-%Y')

print("\nПреобразуван Series (с формат):\n", datetime_series_formatted)
```

*Тук указваме, че датите са във формат месец-ден-година, което помага на Pandas да ги парсне правилно.*

Разбира се, ето резюме на предоставения пример, който демонстрира използването на параметъра `format` във функцията `pd.to_datetime()` за преобразуване на Pandas Series с дати в специфичен формат:

- 1) **Създава се Series с дати в специфичен формат:** Инициализира се Pandas Series `date_series_specific_format` с три елемента, които са низови представяния на дати във формата 'месец-ден-година' (например, '10-26-2023' за 26 октомври 2023 г.). Оригиналният Series се извежда на конзолата.
- 2) **Преобразуване с указване на формата:** Функцията `pd.to_datetime()` се използва с параметъра `format='%m-%d-%Y'`. Този параметър указва на Pandas как точно са форматираните низове с дати.
  - `%m` указва месеца като число с водеща нула.
  - `%d` указва деня от месеца като число с водеща нула.

- %Y указва годината като четирицифрено число. Чрез предоставянето на точния формат, Pandas може да парсне датите правилно и ефективно.
- 3) **Извежда се преобразуваният Series:** Резултатът е нов Series `datetime_series_formatted`, който съдържа същите дати, но вече като `datetime` обекти с тип данни `datetime64[ns]`. Преобразуваният Series се извежда на конзолата.

## Накратко:

Примерът подчертава важноста на използването на параметъра `format` в `pd.to_datetime()`, когато работите с дати в неясен или нестандартен формат. Задаването на правилния формат не само гарантира правилното преобразуване, но и може да ускори процеса на парсане, особено при големи набори от данни.

### в) Пример 3: Обработка на невалидни стойности с `errors='coerce'`

Подобно на `pd.to_numeric()`, можем да използваме `errors='coerce'` за замяна на невалидни дати с `NaT` (Not a Time).

```
import pandas as pd

# Създаваме Series с невалидна дата
date_series_invalid = pd.Series(['2023-10-26', 'invalid date', '2024-01-12'])

print("Оригинален Series:\n", date_series_invalid)

# Преобразуваме с errors='coerce'
datetime_series_coerced = pd.to_datetime(date_series_invalid,
errors='coerce')

print("\nПреобразуван Series (errors='coerce'):\n",
datetime_series_coerced)
```

*Невалидният низ 'invalid date' е заменен с NaT.*

- 1) **Създава се Series с невалидна дата:** Инициализира се Pandas Series `date_series_invalid` с три елемента, като един от тях ('invalid date') не може да бъде разпознат като валидна дата или час. Оригиналният Series се извежда на конзолата.
- 2) **Преобразуване с `errors='coerce'`:** Функцията `pd.to_datetime()` се използва с аргумента `errors='coerce'`. Когато тази опция е зададена, Pandas се опитва да преобразува всеки елемент от Series-а към `datetime` обект. Ако преобразуването е успешно (както е за '2023-10-26' и '2024-01-12'), стойността се превръща в `datetime` обект. Ако преобразуването е неуспешно (както е за 'invalid date'), невалидната стойност се заменя със специалната стойност `NaT` (Not a Time), която Pandas използва за представяне на липсващи или невалидни `datetime` стойности.
- 3) **Извежда се преобразуваният Series:** Резултатът е нов Series `datetime_series_coerced`, където валидните дати са преобразувани в `datetime` обекти, а невалидната стойност 'invalid date' е заменена с `NaT`. Преобразуваният Series се извежда на конзолата.

## Накратко:

Примерът показва, че `errors='coerce'` е полезен начин за справяне със стойности, които не могат да бъдат разпознати като валидни дати или часове. Чрез принудителното им преобразуване в `NaT`, можем да продължим с анализ на времеви данни, като тези невалидни стойности могат да бъдат обработени по-късно (например, чрез премахване или попълване).

### г) Пример 4: Използване на `dayfirst=True`

Когато датите са във формат ден/месец/година.

```
import pandas as pd

# Създаваме Series с дати във формат ден/месец/година
date_series_dayfirst = pd.Series(['26/10/2023', '15/11/2023',
                                  '12/01/2024'])

print("Оригинален Series:\n", date_series_dayfirst)

# Преобразуваме с dayfirst=True
datetime_series_df = pd.to_datetime(date_series_dayfirst, dayfirst=True)
print("\nПреобразуван Series (dayfirst=True):\n", datetime_series_df)
```

Тук указваме, че първата част от низа е денят.

1. **Създава се series с дати във формат ден/месец/година:** Инициализира се Pandas Series `date_series_dayfirst` с три елемента, които са низови представяния на дати във формата 'ден/месец/година' (например, '26/10/2023' за 26 октомври 2023 г.). Оригиналният Series се извежда на конзолата.
2. **Преобразуване с `dayfirst=True`:** Функцията `pd.to_datetime()` се използва с параметъра `dayfirst=True`. Това указва на Pandas, че при парсване на низовете, първата числова стойност трябва да се интерпретира като деня, а втората като месеца. Без тази опция, Pandas по подразбиране би интерпретирал '26/10/2023' като 10 януари 2023 г. (ако `yearfirst` е `False`, което е по подразбиране).
3. **Извежда се преобразуваният series:** Резултатът е нов Series `datetime_series_df`, който съдържа същите дати, но вече като `datetime` обекти с тип данни `datetime64[ns]`, парснати правилно според указания формат ден/месец/година. Преобразуваният Series се извежда на конзолата.

## Накратко:

Примерът показва важността на използването на параметъра `dayfirst=True` в `pd.to_datetime()`, когато форматът на датите в данните ви е ден-месец-година. Правилното задаване на тази опция гарантира, че датите се интерпретират вярно и се преобразуват към `datetime` обекти по желания начин.

### д) Пример 5: Преобразуване на числови данни (timestamp) с unit и origin

Можем да преобразуваме числови стойности, представляващи брой секунди/милисекунди и т.н. от определена епоха, в datetime обекти.

```
import pandas as pd

# Series от брой секунди от Unix epoch (1970-01-01)
timestamp_seconds = pd.Series([1698307200, 1700035200])
print("Оригинален Series (timestamp seconds):\n", timestamp_seconds)

# Преобразуваме с unit='s' (секунди) и origin по подразбиране ('unix')
datetime_series_from_seconds = pd.to_datetime(timestamp_seconds,
unit='s')
print("\nПреобразуван Series (from seconds):\n",
datetime_series_from_seconds)

# Series от брой милисекунди от 2023-01-01
timestamp_ms = pd.Series([0, 1000, 2000])
print("\nОригинален Series (timestamp milliseconds):\n", timestamp_ms)

# Преобразуваме с unit='ms' (милисекунди) и origin
datetime_series_from_ms = pd.to_datetime(timestamp_ms, unit='ms',
origin='2023-01-01')
print("\nПреобразуван Series (from milliseconds, origin):\n",
datetime_series_from_ms)
```

#### 1. Преобразуване на брой секунди от Unix epoch:

- Създава се Pandas Series timestamp\_seconds, съдържащ цели числа, които представляват броя на секундите, изминали от Unix epoch (1 януари 1970 г., 00:00:00 UTC). Оригиналният Series се извежда на конзолата.
- Функцията pd.to\_datetime() се използва с параметъра unit='s', който указва, че входните числа са в секунди. Параметърът origin не е зададен, така че се използва стойността по подразбиране 'unix'. Резултатът е datetime\_series\_from\_seconds, който съдържа datetime обекти, съответстващи на тези времеви печати.

#### 2. Преобразуване на брой милисекунди от определена начална дата:

- Създава се Pandas Series timestamp\_ms, съдържащ цели числа, които представляват броя на милисекундите, изминали от 1 януари 2023 г. Оригиналният Series се извежда на конзолата.
- Функцията pd.to\_datetime() се използва с параметъра unit='ms', който указва, че входните числа са в милисекунди, и параметъра origin='2023-01-01', който задава



началната дата (епохата), спрямо която се измерват милисекундите. Резултатът е `datetime_series_from_ms`, който съдържа `datetime` обекти, съответстващи на тези времеви печати, започвайки от указаната начална дата.

## Накратко:

Примерът показва как `pd.to_datetime()` може да преобразува числови данни, представляващи времеви моменти, в Pandas `datetime` обекти. Параметърът `unit` указва мерната единица на числата (секунди, милисекунди и др.), а параметърът `origin` задава началната времева точка (епохата), спрямо която се измерват тези единици. Това е полезно, когато работите с данни, където времето е представено като числени отмествания.

Тези примери показват гъвкавостта на `pd.to_datetime()` при работа с различни формати и типове входни данни, представляващи дати и часове. Правилното използване на тази функция е ключово за ефективен анализ на времеви данни в Pandas.

## 4. `pd.to_timedelta()`

Функцията `pd.to_timedelta()` е специализирана за преобразуване на аргументи, които могат да бъдат разбрани като времеви разлики (например, продължителност, интервали), в Pandas `timedelta` обекти. Тези обекти са полезни за извършване на аритметични операции с дати и часове (например, добавяне или изваждане на времеви интервали).

Синтаксис:

```
pd.to_timedelta(arg, unit=None, errors='raise')
```

## Основни параметри:

- **arg:** `Series`, `DataFrame` (една колона), списък-подобен обект, NumPy array, или низ, който искате да преобразувате във `Timedelta`. Низът може да бъде във формат, разпознаван от Pandas (например, '1 day', '2 hours 30 minutes', 'PT1H30M'). Числовите стойности се интерпретират като брой единици, указани в параметъра `unit`.
- **unit:** Стринг, указващ мерната единица за числови входни данни. Възможни стойности са 'D' (дни), 'h' (часове), 'm' (минути), 's' (секунди), 'ms' (милисекунди), 'us' (микросекунди), 'ns' (наносекунди). Ако `arg` е низ или списък от низове, обикновено не е необходимо да се указва `unit`, тъй като Pandas се опитва да го извлече от низа.
- **errors:** Стринг, контролиращ обработката на грешки при преобразуване. Възможни стойности са 'raise' (по подразбиране), 'coerce' (невалидните стойности се заменят с `NaT`), и 'ignore' (невалидните стойности се запазват).

*а) Пример 1: Преобразуване на Series от низови времеви разлики и числови стойности с указана единица*

```
import pandas as pd
```

```
# Series от низови времеви разлики
time_diff_strings = pd.Series(['1 day', '2 hours', '30 minutes', '1 day
2 hours 30 minutes'])
print("Оригинален Series (низове):\n", time_diff_strings)

timedelta_from_strings = pd.to_timedelta(time_diff_strings)
print("\nПреобразуван Series (от низове):\n", timedelta_from_strings)
print("Тип данни:", timedelta_from_strings.dtype)

# Series от числови стойности (в секунди)
time_diff_seconds = pd.Series([3600, 7200, 10800])
print("\nОригинален Series (секунди):\n", time_diff_seconds)

timedelta_from_seconds = pd.to_timedelta(time_diff_seconds, unit='s')
print("\nПреобразуван Series (от секунди):\n", timedelta_from_seconds)
print("Тип данни:", timedelta_from_seconds.dtype)
```

*Този пример обединява преобразуването както на низови представления на времеви разлики, така и на числови стойности, където изрично указваме, че мерната единица е секунди.*

#### 1) Преобразуване на Series от низови времеви разлики:

- Създава се Pandas Series `time_diff_strings`, съдържащ низове, които описват времеви интервали в човешки четим формат (например, '1 day', '2 hours').
- Функцията `pd.to_timedelta(time_diff_strings)` се използва за преобразуване на тези низове във Timedelta обекти. Pandas автоматично разпознава единиците (дни, часове, минути) в низовете.
- Резултатът е `timedelta_from_strings`, който е Series от Timedelta обекти. Типът данни на този Series е `timedelta64[ns]`.

#### 2) Преобразуване на Series от числови стойности (в секунди):

- Създава се Pandas Series `time_diff_seconds`, съдържащ числови стойности.
- Функцията `pd.to_timedelta(time_diff_seconds, unit='s')` се използва за преобразуване на тези числа във Timedelta обекти. Параметърът `unit='s'` указва, че всяко число в Series-а представлява брой секунди.
- Резултатът е `timedelta_from_seconds`, който също е Series от Timedelta обекти. Типът данни е `timedelta64[ns]`.

## Накратко:

Примерът показва как `pd.to_timedelta()` може да преобразува различни видове входни данни, представляващи времеви разлики, в Pandas Timedelta обекти. Той демонстрира както парсването на низове с човешки четими описания на времеви интервали, така и преобразуването на числови стойности, където е необходимо да се укаже мерната единица. Timedelta обектите са полезни за извършване на времеви аритметични операции.

## б) Пример 2: Обработка на невалидни стойности с `errors='coerce'` и преобразуване на `DataFrame` колона

```
import pandas as pd

# DataFrame с колона, съдържаща валидни и невалидни времеви разлики
data = {'Времеви_интервал': ['1 hour', 'not a time', '2 days', '0.5 hours']}

df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)
print("Тип данни на колоната:", df['Времеви_интервал'].dtype)

# Преобразуваме колоната към timedelta, като невалидните стойности стават NaT
df['Времеви_интервал_timedelta'] =
pd.to_timedelta(df['Времеви_интервал'], errors='coerce')
print("\nDataFrame след преобразуване:\n", df)
print("Тип данни на новата колона:",
df['Времеви_интервал_timedelta'].dtype)
```

Тук показваме как `pd.to_timedelta()` може да се приложи към колона на `DataFrame` и как `errors='coerce'` заменя неразпознаваемите времеви интервали с `NaT`.

- 1) **Създава се `DataFrame` с колона от времеви интервали:** Инициализира се Pandas `DataFrame` `df` с една колона, наречена 'Времеви\_интервал'. Тази колона съдържа низове, някои от които представляват валидни времеви разлики ('1 hour', '2 days', '0.5 hours'), а други не ('not a time'). Типът данни на тази колона е `object`. Оригиналният `DataFrame` и типът данни на колоната се извеждат на конзолата.
- 2) **Преобразуване на колоната към `timedelta` с `errors='coerce'`:** Функцията `pd.to_timedelta()` се прилага към колоната 'Времеви\_интервал'. Параметърът `errors='coerce'` указва, че ако Pandas срещне низ, който не може да бъде разпознат като валидна времева разлика, вместо да хвърли грешка, той трябва да го замени със специалната стойност `NaT` (Not a Time), която е еквивалентна на `NaN` за времеви разлики.
- 3) **Извежда се `DataFrame` след преобразуването:** Създава се нова колона 'Времеви\_интервал\_timedelta' в `DataFrame`-а, която съдържа резултата от преобразуването. Валидните времеви разлики са успешно преобразувани във `Timedelta` обекти, а невалидната стойност 'not a time' е заменена с `NaT`. Типът данни на новата колона е `timedelta64[ns]`. Преобразуваният `DataFrame` и типът данни на новата колона се извеждат на конзолата.

## Накратко:

Примерът показва как `errors='coerce'` може да бъде полезен при работа с колони, съдържащи потенциално нечисти данни за времеви разлики. Чрез принудителното преобразуване на невалидните стойности в `NaT`, можем да избегнем грешки и да продължим с анализ, като `NaT` стойностите могат да бъдат обработени по-късно.

### в) Пример 3: Преобразуване на числови стойности с различни мерни единици

```
import pandas as pd

# Series от числови стойности с различни мерни единици
time_diff_mixed_units = pd.Series([1, 2.5, 60, 0.001])
print("Оригинален Series (числа):\n", time_diff_mixed_units)

# Преобразуваме към timedelta с указване на различни единици
timedelta_days = pd.to_timedelta(time_diff_mixed_units, unit='D')
print("\nВремеви разлики в дни:\n", timedelta_days)

timedelta_hours = pd.to_timedelta(time_diff_mixed_units, unit='h')
print("\nВремеви разлики в часове:\n", timedelta_hours)

timedelta_milliseconds = pd.to_timedelta(time_diff_mixed_units,
unit='ms')
print("\nВремеви разлики в милисекунди:\n", timedelta_milliseconds)
```

Този пример илюстрира как един и същ *Series* от числови стойности може да бъде интерпретиран като различни времеви продължителности в зависимост от зададената мерна единица (*unit*).

Разбира се, ето резюме на предоставения пример, който демонстрира как функцията `pd.to_timedelta()` може да се използва за преобразуване на *Pandas Series* от числови стойности във *Timedelta* обекти, като се интерпретират тези числа като различни мерни единици (дни, часове, милисекунди) чрез използване на параметъра `unit`:

- 1) **Създава се Series от числови стойности:** Инициализира се *Pandas Series* `time_diff_mixed_units`, съдържащ числови стойности (1, 2.5, 60, 0.001). Оригиналния *Series* се извежда на конзолата.
- 2) **Преобразуване с `unit='D'` (дни):** Функцията `pd.to_timedelta()` се използва с параметъра `unit='D'`. Това интерпретира всяко число в `time_diff_mixed_units` като брой дни и ги преобразува в съответните *Timedelta* обекти. Резултатът `timedelta_days` показва тези времеви разлики в дни.
- 3) **Преобразуване с `unit='h'` (часове):** Функцията `pd.to_timedelta()` се използва с параметъра `unit='h'`. Това интерпретира всяко число в `time_diff_mixed_units` като брой часове и ги

преобразува в съответните `Timedelta` обекти. Резултатът `timedelta_hours` показва тези времеви разлики в часове.

- 4) **Преобразуване с `unit='ms'` (милисекунди):** Функцията `pd.to_timedelta()` се използва с параметъра `unit='ms'`. Това интерпретира всяко число в `time_diff_mixed_units` като брой милисекунди и ги преобразува в съответните `Timedelta` обекти. Резултатът `timedelta_milliseconds` показва тези времеви разлики в милисекунди.

## Накратко:

Примерът илюстрира как параметърът `unit` в `pd.to_timedelta()` позволява гъвкаво интерпретиране на числови данни като различни времеви продължителности. Една и съща поредица от числа може да представлява съвсем различни времеви интервали в зависимост от мерната единица, която е указана при преобразуването. Това е полезно, когато работите с данни, където времевите разлики са представени числено в различни единици.

### г) Пример 4: Интерпретация на цели числа и комбиниране с низови единици

```
import pandas as pd

# Series от цели числа (интерпретирани като секунди по подразбиране)
time_diff_integers = pd.Series([3600, 86400])
print("Оригинален Series (цели числа):\n", time_diff_integers)

timedelta_from_integers = pd.to_timedelta(time_diff_integers)
print("\nПреобразуван Series (от цели числа - в секунди):\n",
      timedelta_from_integers)
print("Тип данни:", timedelta_from_integers.dtype)

# Комбиниране на числа с низови единици
time_diff_mixed = pd.Series(['1 day', 7200, '0.5 hours'])
print("\nОригинален Series (смесени):", time_diff_mixed)

timedelta_from_mixed = pd.to_timedelta(time_diff_mixed)
print("\nПреобразуван Series (смесени):\n", timedelta_from_mixed)
print("Тип данни:", timedelta_from_mixed.dtype)
```

Този пример показва как цели числа се интерпретират като секунди по подразбиране и как `pd.to_timedelta()` може да обработва `Series`, съдържащ както низови представления на времеви разлики, така и числови стойности (които се интерпретират като секунди).

#### 1) Преобразуване на `Series` от цели числа:

- Създава се Pandas `Series` `time_diff_integers`, съдържащ цели числа (3600 и 86400). Когато `unit` не е изрично зададен за числови входни данни, `pd.to_timedelta()` ги

интерпретира като наносекунди. Въпреки това, Pandas обикновено показва `Timedelta` обектите в по-големи, по-разбираеми единици, когато е възможно (в този случай, като секунди).

- Функцията `pd.to_timedelta(time_diff_integers)` се използва за преобразуване на тези числа във `Timedelta` обекти.
- Резултатът `timedelta_from_integers` е `Series` от `Timedelta` обекти, представляващи съответно 3600 секунди (1 час) и 86400 секунди (1 ден). Типът данни е `timedelta64[ns]`.

## 2) Комбиниране на числа с низови единици:

- Създава се `Pandas Series time_diff_mixed`, съдържащ както низов ('1 day'), така и числени (7200) и отново низов ('0.5 hours') представяния на времеви разлики.
- Функцията `pd.to_timedelta(time_diff_mixed)` се използва за преобразуване на тези смесени стойности във `Timedelta` обекти. `Pandas` е в състояние да парсне низовите представяния и да интерпретира числовите стойности като секунди (когато са в `Series` без изричен `unit` и в съседство с други, които могат да бъдат парснати с единици).
- Резултатът `timedelta_from_mixed` е `Series` от `Timedelta` обекти, представляващи 1 ден, 7200 секунди (2 часа) и 0.5 часа (30 минути). Типът данни е `timedelta64[ns]`.

## Накратко:

Примерът показва, че `pd.to_timedelta()` може да работи както с числови, така и с низови представяния на времеви разлики в един и същ `Series`. Когато се подават само цели числа без указан `unit`, те се интерпретират като наносекунди, но се показват в по-удобни единици. Когато числата се комбинират с низове, съдържащи единици, `Pandas` може да ги интерпретира в контекста на тези единици (например, като секунди в този случай).

### д) Пример 5: Работа с ISO 8601 формат за времеви интервали

```
import pandas as pd

# Series от времеви интервали в ISO 8601 формат
iso_time_diffs = pd.Series(['PT1H30M', 'P1DT2H', 'PT45S'])
print("Оригинален Series (ISO 8601):\n", iso_time_diffs)

timedelta_from_iso = pd.to_timedelta(iso_time_diffs)
print("\nПреобразуван Series (от ISO 8601):\n", timedelta_from_iso)
print("Тип данни:", timedelta_from_iso.dtype)
```

Този пример демонстрира как `pd.to_timedelta()` може директно да парсва низове, които са във формат `ISO 8601` за продължителност (където 'P' указва период, 'T' указва време, следвани от числа и съответните единици като 'D' за дни, 'H' за часове, 'M' за минути, 'S' за секунди).

- 1) Създава се `Series` от `ISO 8601` времеви интервали: Инициализира се `Pandas Series iso_time_diffs` с три низови елемента, всеки от които представлява времеви интервал в `ISO 8601` формат за продължителност:
  - 'PT1H30M' - 1 час и 30 минути



- 'P1DT2H' - 1 ден и 2 часа
- 'PT45S' - 45 секунди

Оригиналният Series се извежда на конзолата.

- 2) **Преобразуване към timedelta:** Функцията `pd.to_timedelta(iso_time_diffs)` се използва за преобразуване на тези ISO 8601 низове директно във Timedelta обекти. Pandas е вградено способен да разпознава и парсва този стандартен формат.
- 3) **Извежда се преобразуваният Series:** Резултатът е `timedelta_from_iso`, който е Series от Timedelta обекти, съответстващи на предоставените ISO 8601 интервали. Типът данни на този Series е `timedelta64[ns]`. Преобразуваният Series и неговият тип данни се извеждат на конзолата.

## Накратко:

Примерът показва, че `pd.to_timedelta()` е удобен инструмент за работа с времеви интервали, представени в ISO 8601 формат. Той позволява директното им преобразуване в Pandas Timedelta обекти, което улеснява последващи времеви аритметични операции и анализ.

### е) Пример 6: Използване на `errors='ignore'`

```
import pandas as pd

# Series с валидни и невалидни стойности за времеви разлики
mixed_time_diffs_with_errors = pd.Series(['1 hour', 'not a time', 7200])
print("Оригинален Series (с грешки):\n", mixed_time_diffs_with_errors)

timedelta_ignored_errors = pd.to_timedelta(mixed_time_diffs_with_errors,
errors='ignore')
print("\nПреобразуван Series (errors='ignore'):\n",
timedelta_ignored_errors)
print("Тип данни:", timedelta_ignored_errors.dtype)
```

Този пример показва как при използване на `errors='ignore'`, невалидният низ `'not a time'` се запазва в Series-а, а типът данни на Series-а остава `object`, тъй като не всички елементи могат да бъдат успешно преобразувани във Timedelta. Числовата стойност (7200) също остава непроменена, тъй като без указан `unit` и в присъствието на невалиден низ, Pandas не може да я преобразува автоматично в Timedelta.

- 1) **Създава се Series със смесени стойности:** Инициализира се Pandas Series `mixed_time_diffs_with_errors` с три елемента: валиден низов представяне на времева разлика ('1 hour'), невалиден низ ('not a time'), и число (7200). Оригиналният Series се извежда на конзолата.



- 2) **Преобразуване с `errors='ignore'`:** Функцията `pd.to_timedelta()` се използва с параметъра `errors='ignore'`. Когато тази опция е зададена, Pandas се опитва да преобразува всеки елемент във `Timedelta`. Ако преобразуването е успешно ('1 hour' става `Timedelta('0 days 01:00:00')`), се връща съответният `Timedelta` обект. Ако преобразуването е неуспешно ('not a time'), оригиналната стойност се запазва в резултата. Числовата стойност (7200) също се запазва, тъй като без изрично указване на `unit` и в присъствието на непреобразуваем низ, Pandas не може да определи мерната единица и я оставя непроменена.
- 3) **Извежда се преобразуваният `Series` и неговият тип данни:** Резултатът е `timedelta_ignored_errors`. Тъй като `Series`-ът съдържа стойност ('not a time'), която не може да бъде преобразувана във `Timedelta`, типът данни на целия `Series` остава `object`, за да може да побере всички типове елементи. Преобразуваният `Series` и неговият тип данни се извеждат на конзолата.

## Накратко:

Примерът показва, че `errors='ignore'` предотвратява хвърлянето на грешка при срещане на невалидни за `Timedelta` преобразуване стойности. Вместо това, тези стойности се запазват в оригиналния си вид, което може да доведе до `Series` с тип данни `object`, ако има елементи, които не са `Timedelta`. Тази опция е полезна, когато искате да избегнете прекъсване на кода, но трябва да сте внимателни при последващ анализ, тъй като колоната може да не е от желанния тип `timedelta64[ns]` за всички елементи.

Тези три примера трябва да дадат добра представа за основните начини за използване на функцията `pd.to_timedelta()` за преобразуване на различни видове данни във времеви разлики в Pandas.

## Казус 1: Преобразуване на колона с цени от низ в числов тип (`pd.to_numeric()` с обработка на грешки)

Представете си, че имате `DataFrame` с информация за продукти, където цените са прочетени като низове и някои от тях съдържат невалидни знаци (например, запетая вместо десетична точка или други символи). Искате да преобразувате тази колона в числов тип, като обработите грешките.

```
import pandas as pd

# Създаваме DataFrame с цени като низове (с потенциални грешки)
prices_data = {
    'продукт': ['A', 'B', 'B', 'Г'],
    'цена_низ': ['12,50', '25.75', 'invalid', '10.00']
}

df_prices_str = pd.DataFrame(prices_data)
print("Оригинален DataFrame (цена като низ):\n", df_prices_str)
```

```
# Казус: Преобразувайте колоната 'цена_низ' в числов тип (float), като
замените невалидните стойности с NaN.
```

## Решение на Казус 1:

```
# Използване на pd.to_numeric() за преобразуване в числов тип с
обработка на грешки
df_prices_numeric = df_prices_str.copy()
df_prices_numeric['цена_число'] =
pd.to_numeric(df_prices_numeric['цена_низ'].str.replace(',', '.', ''),
errors='coerce')
print("\nDataFrame с цена като число:\n", df_prices_numeric)
```

В този случай, първо заменяме запетаята с точка, за да осигурим правилното разпознаване на десетичната част. След това, `pd.to_numeric()` се използва с `errors='coerce'`, което означава, че ако преобразуването на дадена стойност е неуспешно, тя ще бъде заменена с `NaN`.

## Казус 2: Преобразуване на колона с дати от низ в `datetime` тип (`pd.to_datetime()` с указване на формат)

Представете си, че имате `DataFrame` с информация за събития, където датите са записани като низове в специфичен формат (например, 'дд.мм.гггг'). Искате да преобразувате тази колона в `datetime` тип, за да можете да извършвате времеви анализи.

```
import pandas as pd

# Създаваме DataFrame с дати като низове (в различен формат)
events_data = {
    'събитие': ['А', 'Б', 'В'],
    'дата_низ': ['26.10.2023', '15-11-2023', '2024/01/12']
}
df_events = pd.DataFrame(events_data)
print("Оригинален DataFrame (дата като низ):\n", df_events)

# Казус: Преобразувайте колоната 'дата_низ' в datetime тип.
```

## Решение на Казус 2:

```
# Използване на pd.to_datetime() за преобразуване в datetime тип (Pandas
ще опита автоматично)
df_events_datetime = df_events.copy()
df_events_datetime['дата_време'] =
pd.to_datetime(df_events_datetime['дата_низ'], errors='coerce')
print("\nDataFrame с дата като datetime:\n", df_events_datetime)

# Ако форматът е консистентен, може да се укаже изрично:
events_data_consistent_format = {'събитие': ['А', 'Б'], 'дата_низ':
['26.10.2023', '27.10.2023']}
df_events_consistent = pd.DataFrame(events_data_consistent_format)
df_events_consistent['дата_време'] =
pd.to_datetime(df_events_consistent['дата_низ'], format='%d.%m.%Y',
errors='coerce')
print("\nDataFrame с дата като datetime (указан формат):\n",
df_events_consistent)
```

В този случай, `pd.to_datetime()` се опитва автоматично да разпознае формата на низовете с дати. Ако форматът е специфичен и консистентен, е препоръчително да се укаже изрично чрез параметъра `format` за по-бързо и надеждно преобразуване.

### Казус 3: Преобразуване на колона с продължителности от низ в `timedelta min` (`pd.to_timedelta()` с указване на единици)

Представете си, че имате `DataFrame` с информация за задачи и тяхната продължителност, записана като низове (например, '1 hour', '30 minutes'). Искате да преобразувате тази колона в `timedelta` тип, за да можете да извършвате времеви изчисления.

```
import pandas as pd

# Създаваме DataFrame с продължителности като низове
tasks_data = {
    'задача': ['А', 'Б', 'В'],
    'продължителност_низ': ['1 hour', '30 minutes', '1.5 hours']
}
df_tasks = pd.DataFrame(tasks_data)
print("Оригинален DataFrame (продължителност като низ):\n", df_tasks)
```

```
# Казус: Преобразувайте колоната 'продължителност_низ' в timedelta тип.
```

## Решение на Казус 3:

```
# Използване на pd.to_timedelta() за преобразуване в timedelta тип
df_tasks_timedelta = df_tasks.copy()
df_tasks_timedelta['продължителност_време'] =
pd.to_timedelta(df_tasks_timedelta['продължителност_низ'],
errors='coerce')
print("\nDataFrame с продължителност като timedelta:\n",
df_tasks_timedelta)

# Ако имате числови стойности с определени единици, може да се укаже
'unit':
numeric_duration_data = {'задача': ['Г', 'Д'],
'продължителност_секунди': [3600, 1800]}
df_numeric_duration = pd.DataFrame(numeric_duration_data)
df_numeric_duration['продължителност_време'] =
pd.to_timedelta(df_numeric_duration['продължителност_секунди'],
unit='s', errors='coerce')
print("\nDataFrame с продължителност като timedelta (от секунди):\n",
df_numeric_duration)
```

*pd.to\_timedelta() може да разпознае различни низови формати за времеви разлики. Ако имате числови данни, представляващи продължителност в определени единици (например, секунди, минути, часове), можете да укажете това чрез параметъра unit.*

## Казус 4: Преобразуване на колона в категориален тип (.astype('category'))

Представете си, че имате DataFrame с информация за държави, където имената на държавите се повтарят многократно. Преобразуването на тази колона в категориален тип може да намали използването на памет и да ускори някои операции.

```
import pandas as pd
```

```
# Създаваме DataFrame с колона от низове с повтарящи се стойности
countries_data = {
    'град': ['София', 'Варна', 'Пловдив', 'Бургас', 'София', 'Пловдив'],
    'държава': ['България', 'България', 'България', 'България',
'Bългария', 'България']
}
df_countries = pd.DataFrame(countries_data)
print("Оригинален DataFrame (държава като низ):\n", df_countries)

# Казус: Преобразувайте колоната 'държава' в категориален тип.
```

## Решение на Казус 4:

```
# Използване на .astype('category') за преобразуване в категориален тип
df_countries_categorical = df_countries.copy()
df_countries_categorical['държава_категория'] =
df_countries_categorical['държава'].astype('category')
print("\nDataFrame с държава като категория:\n",
df_countries_categorical)
print("\nТип данни на колоната 'държава_категория':",
df_countries_categorical['държава_категория'].dtype)
```

*.astype('category')* е ефективен начин за представяне на колони с ограничен брой уникални стойности, което може да доведе до оптимизация на паметта и производителността.

Тези казуси илюстрират как различните методи за промяна на типа данни в Pandas се използват в зависимост от изходния тип данни, желаната трансформация и потенциалните грешки в данните. Правилното преобразуване на типа данни е ключова стъпка в процеса на анализ на данни.

## XIII. Работа с текстови данни (String Handling): Аксесорът `.str` (методи за низове).

Когато работим с данни, често се налага да обработваме текстови колони. Pandas предоставя мощен и удобен начин за извършване на операции с низове върху Series (например, колона от DataFrame) чрез специалния **аксесор** `.str`.

Аксесорът `.str` позволява да прилагаме методи за низове, подобни на тези, които са налични в Python, директно върху всеки елемент от Series, без да е необходимо да използваме цикли или други по-бавни

конструкции. Резултатът от тези операции обикновено е нов Series (или DataFrame, в зависимост от метода).

Някои от често използваните методи, достъпни чрез `.str`, включват:

- `.lower()` / `.upper()` / `.capitalize()` / `.title()`: Промяна на регистъра на низовете.
- `.strip()` / `.lstrip()` / `.rstrip()`: Премахване на водещи и/или завършващи интервали (или други знаци).
- `.split()` / `.rsplit()`: Разделяне на низове по разделител.
- `.join()`: Съединяване на елементи от списък или друг итерируем обект в низ.
- `.replace()`: Заместване на подниз с друг.
- `.contains()`: Проверка дали низ съдържа определен подниз (връща булев Series).
- `.startswith()` / `.endswith()`: Проверка дали низ започва или завършва с определен подниз (връща булев Series).
- `.len()`: Връща дължината на всеки низ.
- `.get()` / `[]`: Достъп до елемент по индекс (след разделяне на низ).
- `.extract()` / `.extractall()`: Извличане на групи от регулярни изрази.
- `.findall()`: Намиране на всички съвпадения на регулярен израз.
- `.count()`: Преброяване на появяванията на подниз или шаблон (регулярен израз).

### Пример 1: Преобразуване към малки букви (`.str.lower()`)

```
import pandas as pd

# Създаваме Series от низове с различен регистър
text_series = pd.Series(['Hello World', 'PYTHON is FUN', 'pandas
LIBRARY'])

print("Оригинален Series:\n", text_series)

# Преобразуваме всички низове към малки букви
lower_case_series = text_series.str.lower()
print("\nSeries след .str.lower():\n", lower_case_series)
```

Този пример показва как `.str.lower()` преобразува всички знаци във всеки низ от `text_series` към малки букви.

### Пример 2: Преобразуване към големи букви (`.str.upper()`)

```
import pandas as pd

# Използваме същия Series от предходния пример
text_series = pd.Series(['Hello World', 'PYTHON is FUN', 'pandas
LIBRARY'])
```

```
print("Оригинален Series:\n", text_series)

# Преобразуваме всички низове към големи букви
upper_case_series = text_series.str.upper()
print("\nSeries след .str.upper():\n", upper_case_series)
```

Тук `.str.upper()` преобразува всички знаци във всеки низ към големи букви.

### Пример 3: Преобразуване към формат "Capitalize" (`.str.capitalize()`)

Методът `.str.capitalize()` преобразува само първия знак на всеки низ към голяма буква, а останалите знаци стават малки.

```
import pandas as pd

# Използваме същия Series
text_series = pd.Series(['hello world', 'pYTHON is FUN', 'PANDAS library'])
print("Оригинален Series:\n", text_series)

# Преобразуваме към capitalize формат
capitalized_series = text_series.str.capitalize()
print("\nSeries след .str.capitalize():\n", capitalized_series)
```

Забележете как само първата буква на всяка дума (ако е първата буква на целия низ) става голяма, а останалите букви се преобразуват към малки.

### Пример 4: Преобразуване към формат "Title Case" (`.str.title()`)

Методът `.str.title()` преобразува първата буква на всяка дума в низа към голяма буква, а останалите букви в думата стават малки.

```
import pandas as pd

# Използваме същия Series
text_series = pd.Series(['hello world', 'pYTHON is FUN', 'PANDAS library the great'])
print("Оригинален Series:\n", text_series)
```



```
# Преобразуваме към title case формат
title_case_series = text_series.str.title()
print("\nSeries след .str.title():\n", title_case_series)
```

Тук виждаме как първата буква на всяка дума (разделена от интервал) е преобразувана към голяма буква.

Тези примери илюстрират основните методи за промяна на регистъра, достъпни чрез аксесора `.str` в Pandas. Те са често използвани при почистване и стандартизиране на текстови данни.

## XIV. Работа с данни за дата и час: Аксесорът `.dt` (компоненти на дата/час).

Когато Series в Pandas съдържа данни от тип `datetime64[ns]` (резултат от преобразуване с `pd.to_datetime()`), можем да използваме специалния аксесор `.dt` за лесен достъп до различни компоненти на датата и часа за всеки елемент от Series-a. Това ни позволява да извличаме информация като година, месец, ден, час, минута, секунда, ден от седмицата, ден от годината и много други.

Аксесорът `.dt` предоставя свойства (attributes) и методи, които връщат Series със съответните компоненти. Някои от често използваните свойства и методи на `.dt` включват:

### 1. Свойства (Attributes):

- `.year`: Връща годината като цяло число.
- `.month`: Връща месеца като цяло число (1-12).
- `.day`: Връща деня от месеца като цяло число (1-31).
- `.hour`: Връща часа като цяло число (0-23).
- `.minute`: Връща минутата като цяло число (0-59).
- `.second`: Връща секундата като цяло число (0-59).
- `.microsecond`: Връща микросекундата като цяло число (0-999999).
- `.nanosecond`: Връща наносекундата като цяло число (0-999999999).
- `.dayofweek`: Връща деня от седмицата като цяло число (понеделник=0, неделя=6).
- `.day_of_week`: Същото като `.dayofweek`.
- `.dayofyear`: Връща поредния ден от годината като цяло число (1-365 или 1-366 за високосна година).
- `.day_of_year`: Същото като `.dayofyear`.
- `.quarter`: Връща тримесечието от годината като цяло число (1-4).
- `.week`: Връща седмицата от годината като цяло число (0-53).
- `.weekofyear`: Същото като `.week`.
- `.month_name()`: Връща името на месеца като низ.
- `.day_name()`: Връща името на деня от седмицата като низ.
- `.is_leap_year`: Връща булева стойност, указваща дали годината е високосна.
- `.tz`: Връща часовата зона.

## 2. Методи:

- `.normalize()`: Преобразува времето в 00:00:00 (запазва само датата).
- `.strftime(date_format)`: Форматира `datetime` обектите като низове според зададен формат (подобно на `strftime` в Python).
- `.round(freq)` / `.floor(freq)` / `.ceil(freq)`: Закръгляне, закръгляне надолу или закръгляне нагоре до зададена честота (например, 'D' за ден, 'H' за час).
- `.to_period(freq)`: Преобразува `datetime` към `Period` обект с зададена честота.
- `.to_pydatetime()`: Връща масив от Python `datetime` обекти.

Използването на аксесора `.dt` е много по-ефективно и четимо от итерирането през `Series` от дати и часове за извличане на тези компоненти.

### Пример 1: Извличане на година, месец и ден

```
import pandas as pd

# Създаваме Series от datetime обекти
dates = pd.to_datetime(['2023-10-26', '2023-11-15', '2024-01-12
10:30:00'])
print("Оригинален Series (datetime):\n", dates)

# Извличаме годината
years = dates.dt.year
print("\nГодини:\n", years)

# Извличаме месеца
months = dates.dt.month
print("\nМесеци:\n", months)

# Извличаме деня от месеца
days = dates.dt.day
print("\nДни:\n", days)
```

Този пример показва как лесно можем да получим `Series` със съответните години, месеци и дни от оригиналния `datetime Series`.

Разбира се, ето резюме на предоставения пример, който демонстрира как да използвате аксесора `.dt` за извличане на основните календарни компоненти (година, месец и ден) от `Pandas Series`, съдържащ `datetime` обекти:

- 1) **Създава се Series от datetime обекти:** Инициализира се `Pandas Series` `dates` чрез преобразуване на списък от низове, представляващи дати (и час в един от елементите), в

datetime обекти с помощта на `pd.to_datetime()`. Оригиналният `Series` се извежда на конзолата.

- 2) **Извличане на годината (`.dt.year`):** Аксесорът `.dt` се използва за достъп до свойството `.year` на всеки datetime обект в `dates`. Резултатът е нов `Series years`, съдържащ само годините от съответните дати.
- 3) **Извличане на месеца (`.dt.month`):** По същия начин, свойството `.month` на аксесора `.dt` се използва за получаване на `Series months`, съдържащ месеците (като числа от 1 до 12) от всяка дата.
- 4) **Извличане на деня от месеца (`.dt.day`):** Свойството `.day` на `.dt` се използва за извличане на `Series days`, съдържащ деня от месеца (като числа от 1 до 31) за всяка дата.

## Накратко:

Примерът показва основното използване на аксесора `.dt` за лесно извличане на основните календарни компоненти (година, месец, ден) от `Series` с datetime данни. Това е много полезно за анализ и филтриране на данни по времеви критерии.

### Пример 2: Извличане на час, минута и секунда

```
import pandas as pd

# Използваме същия Series от предходния пример
dates = pd.to_datetime(['2023-10-26', '2023-11-15', '2024-01-12
10:30:00'])
print("Оригинален Series (datetime):\n", dates)

# Извличаме часа
hours = dates.dt.hour
print("\nЧасове:\n", hours)

# Извличаме минутата
minutes = dates.dt.minute
print("\nМинути:\n", minutes)

# Извличаме секундата
seconds = dates.dt.second
print("\nСекунди:\n", seconds)
```

Тук демонстрираме как да извлечем часовете, минутите и секундите от datetime Series. За първите две дати, където часът не е указан, стойностите са 0.

- 1) **Използва се Series от datetime обекти:** Припомня се Pandas Series dates, съдържащ datetime обекти, включително информация за час, минути и секунди в последния елемент. Оригиналният Series се извежда на конзолата.
- 2) **Извличане на часа (.dt.hour):** Аксесорът .dt се използва за достъп до свойството .hour на всеки datetime обект в dates. Резултатът е нов Series hours, съдържащ часа (в 24-часов формат) от съответните дати и часове. За датите, където часът не е бил изрично указан при създаването на Series-a, стойността е 0.
- 3) **Извличане на минутата (.dt.minute):** По същия начин, свойството .minute на аксесора .dt се използва за получаване на Series minutes, съдържащ минутите (от 0 до 59) от всяка дата и час. За датите без изрично указани минути, стойността е 0.
- 4) **Извличане на секундата (.dt.second):** Свойството .second на .dt се използва за извличане на Series seconds, съдържащ секундите (от 0 до 59) за всяка дата и час. За датите без изрично указани секунди, стойността е 0.

## Накратко:

Примерът показва как аксесорът .dt улеснява достъпа до времевите компоненти (час, минута, секунда) на Series с datetime данни. Това е полезно за анализ на данни, свързани с времето на събития, или за агрегиране на данни по определени времеви интервали.

### Пример 3: Извличане на ден от седмицата и име на месеца

```
import pandas as pd

# Създаваме Series от datetime обекти
dates = pd.to_datetime(['2023-10-26', '2023-11-15', '2024-01-12'])
print("Оригинален Series (datetime):\n", dates)

# Извличаме деня от седмицата (понеделник=0, неделя=6)
day_of_week = dates.dt.dayofweek
print("\nДен от седмицата (0=Пн, 6=Нд):\n", day_of_week)

# Извличаме името на месеца
month_name = dates.dt.month_name()
print("\nИме на месеца:\n", month_name)
```

*Този пример показва как да получим числено представяне на деня от седмицата и как да извлечем името на месеца като низ.*

- 1) **Създава се Series от datetime обекти:** Инициализира се Pandas Series dates чрез преобразуване на списък от низове, представляващи дати, в datetime обекти с помощта на pd.to\_datetime(). Оригиналният Series се извежда на конзолата.

- 2) **Извличане на деня от седмицата (`.dt.dayofweek`):** Аксесорът `.dt` се използва за достъп до свойството `.dayofweek` на всеки `datetime` обект в `dates`. Това свойство връща цяло число, представляващо деня от седмицата, където понеделник е 0, вторник е 1 и така до неделя, която е 6. Резултатът е нов `Series` `day_of_week`, съдържащ тези числови представяния.
- 3) **Извличане на името на месеца (`.dt.month_name()`):** Аксесорът `.dt` се използва за достъп до метода `.month_name()` на всеки `datetime` обект в `dates`. Този метод връща низово представяне на името на месеца за всяка дата. Резултатът е нов `Series` `month_name`, съдържащ тези имена на месеци.

## Накратко:

Примерът показва как аксесорът `.dt` предоставя удобен начин за извличане както на числени (например, ден от седмицата), така и на текстови (например, име на месец) компоненти от `Series` с `datetime` данни, което е полезно за анализ и визуализация на данни, свързани с времето.

### Пример 4: Използване на `.dt.strftime()` за форматиране на дати

```
import pandas as pd

# Създаваме Series от datetime обекти
dates = pd.to_datetime(['2023-10-26', '2023-11-15', '2024-01-12
10:30:00'])

print("Оригинален Series (datetime):\n", dates)

# Форматираме датите като 'ден/месец/година'
formatted_dates = dates.dt.strftime('%d/%m/%Y')
print("\nФорматирани дати (ден/месец/година):\n", formatted_dates)

# Форматираме датите и часа
formatted_datetime = dates.dt.strftime('%Y-%m-%d %H:%M')
print("\nФорматирани дата и час (YYYY-MM-DD HH:MM):\n",
formatted_datetime)
```

Тук използваме `.dt.strftime()` за да форматираме `datetime` обектите в желани низови представяния, използвайки стандартни Python директиви за форматиране на дати и часове.

- 1) **Създава се Series от datetime обекти:** Инициализира се Pandas Series `dates` чрез преобразуване на списък от низове, представляващи дати (и час в един от елементите), в `datetime` обекти с помощта на `pd.to_datetime()`. Оригиналният Series се извежда на конзолата.
- 2) **Форматиране на дати като 'ден/месец/година' (`.dt.strftime('%d/%m/%Y')`):** Аксесорът `.dt` се използва за достъп до метода `.strftime()` на всеки `datetime` обект в `dates`. Методът `.strftime()` приема като аргумент форматен низ, който указва как да бъде представен `datetime` обектът като низ. В този случай, форматният низ `'%d/%m/%Y'` указва:

- %d: Ден от месеца като число с водеща нула.
- %m: Месец като число с водеща нула.
- %Y: Година като четирицифрено число. Резултатът е нов Series formatted\_dates, съдържащ датите, форматирувани като низове в желанния формат.

3) **Форматиране на дата и час като 'YYYY-MM-DD HH:MM' (.dt.strftime('%Y-%m-%d %H:%M')):** Отново се използва методът .strftime() с друг форматен низ '%Y-%m-%d %H:%M', който указва:

- %Y: Година като четирицифрено число.
- %m: Месец като число с водеща нула.
- %d: Ден от месеца като число с водеща нула.
- %H: Час (24-часов формат) като число с водеща нула.
- %M: Минути като число с водеща нула. Резултатът е нов Series formatted\_datetime, съдържащ датите и часовете, форматирувани като низове в указания формат.

## Накратко:

Примерът показва как методът .dt.strftime() предоставя голяма гъвкавост при представянето на datetime данни като низове в различни желани формати. Това е особено полезно за генериране на отчети, визуализации или за записване на данни във файлове с определен формат за дата и час.

Тези примери илюстрират някои от основните възможности на аксесора .dt за лесен достъп до компонентите на данни за дата и час в Pandas. Той предоставя много други полезни свойства и методи, които могат да бъдат използвани за анализ и манипулиране на времеви данни.

## Казус 1: Анализ на обратна връзка от клиенти (.str.lower(), .str.contains())

Представете си, че имате DataFrame с колона, съдържаща обратна връзка от клиенти. Искате да анализирате тази обратна връзка, като преброите колко отзива съдържат думата "доволен" (без значение от регистъра на буквите).

```
import pandas as pd

# Създаваме DataFrame с обратна връзка от клиенти
feedback_data = {
    'клиент': ['А', 'Б', 'В', 'Г'],
    'отзив': ['Много съм доволен от продукта!', 'Не съм доволен от обслужването.',
             'Продуктът е Доволен и лесен за употреба.', 'Като цяло съм доволен. ']
}
```

```
df_feedback = pd.DataFrame(feedback_data)
print("Оригинален DataFrame с отзиви:\n", df_feedback)

# Казус: Пребройте колко отзиви съдържат думата "доволен" (без значение от регистъра).
```

## Решение на Казус 1:

```
# Преобразуваме всички отзиви в малки букви
df_feedback['отзив_малки'] = df_feedback['отзив'].str.lower()

# Проверяваме кои отзиви съдържат думата "доволен"
contains_доволен = df_feedback['отзив_малки'].str.contains('доволен')

# Преброяваме True стойностите
count_доволен = contains_доволен.sum()

print("\nБрой отзиви, съдържащи 'доволен':", count_доволен)
```

В този казус, първо използваме `.str.lower()` за да преобразуваме всички отзиви в малки букви, което позволява търсене без значение от регистъра. След това използваме `.str.contains('доволен')` за да създадем булев Series, указващ кои отзиви съдържат думата "доволен". Накрая, сумираме булевия Series (True се интерпретира като 1, False като 0), за да получим общия брой на отзивите, съдържащи търсената дума.

## Казус 2: Извличане на потребителски имена от имейл адреси

`(.str.split(), .str.get())`

Представете си, че имате DataFrame с колона, съдържаща имейл адреси на потребители. Искате да извлечете потребителското име (частта преди символа '@') от всеки имейл адрес.

```
import pandas as pd

# Създаваме DataFrame с имейл адреси
emails_data = {
    'потребител': ['Иван Иванов', 'Петър Петров'],
    'имейл': ['ivan.ivanov@example.com', 'petar_petrov@another.org']
}
```



```
df_emails = pd.DataFrame(emails_data)
print("Оригинален DataFrame с имейл адреси:\n", df_emails)

# Казус: Извлечете потребителското име от всеки имейл адрес.
```

## Решение на Казус 2:

```
# Разделяме имейл адресите по символа '@'
split_emails = df_emails['имейл'].str.split('@')

# Извличаме първия елемент от всяко разделяне (потребителското име)
usernames = split_emails.str.get(0)

df_emails['потребителско_име'] = usernames
print("\nDataFrame с извлечени потребителски имена:\n", df_emails)
```

В този казус, използваме `.str.split('@')` за да разделим всеки имейл адрес на две части по символа '@'. Резултатът е Series от списъци. След това използваме `.str.get(0)` за да извлечем първия елемент (с индекс 0) от всеки списък, който представлява потребителското име.

## Казус 3: Форматиране на телефонни номера (`.str.replace()`)

Представете си, че имате Series с телефонни номера, които са в различни формати (с интервали, тирета, скоби). Искате да ги стандартизирате до единен формат (например, само цифри).

```
import pandas as pd

# Създаваме Series с телефонни номера в различни формати
phone_numbers = pd.Series(['0888 12 34 56', '(02) 987-65-43',
                           '0899555111', '0700/12345'])
print("Оригинален Series с телефонни номера:\n", phone_numbers)

# Казус: Форматирайте телефонните номера, като премахнете всички
# нецифрови символи.
```

## Решение на Казус 3:

```
# Използваме .str.replace() с регулярен израз за премахване на нецифрови  
символи  
formatted_numbers = phone_numbers.str.replace(r'\D+', '', regex=True)  
  
print("\nФорматирани телефонни номера:\n", formatted_numbers)
```

В този казус, използваме `.str.replace(r'\D+', '', regex=True)`. Регулярният израз `\D+` съвпада с един или повече нецифрови символи. Замества всички съвпадения с празен низ (`''`), което ефективно премахва всички нецифрови символи от телефонните номера. Параметърът `regex=True` указва, че първият аргумент е регулярен израз.

Тези казуси илюстрират някои от многото полезни методи, достъпни чрез аксесора `.str` за обработка и анализ на текстови данни в Pandas. Тези методи позволяват ефективно почистване, трансформиране и извличане на информация от текстови колони.

## XV. Работа с категорийни данни: Аксесорът `.cat`

Категорийните данни са тип данни в Pandas, който е полезен за представяне на колони, съдържащи ограничен и обикновено повтарящ се набор от стойности (наречени категории). Използването на категорийни данни може да доведе до по-ефективно използване на паметта и по-бързи операции в сравнение с използването на обекти (низове) за такива данни. Освен това, категорийният тип данни позволява задаването на ред между категориите, което може да бъде важно за някои видове анализ и визуализация.

Когато `Series` в Pandas е от категориален тип (`category`), можем да използваме специалния аксесор `.cat` за достъп до специфични свойства и методи, свързани с категориите.

Някои от често използваните свойства и методи на `.cat` включват:

### 1. Свойства (Attributes):

- `.categories`: Връща `Index` обект, съдържащ уникалните категории.
- `.ordered`: Връща булева стойност, указваща дали категориите имат зададен ред.
- `.codes`: Връща `Series` от целочислени кодове, представящи всяка стойност в оригиналния `Series` спрямо категориите.

### 2. Методи:

- `.rename_categories(new_categories)`: Преименува категориите.
- `.reorder_categories(new_categories, ordered=None)`: Пренарежда категориите и може да зададе или промени реда.

- `.add_categories(new_categories)`: Добавя нови категории (съществуващите стойности в Series-a не се променят, но новите категории стават валидни).
- `.remove_categories(removals)`: Премахва посочени категории (всички съответстващи стойности в Series-a стават NaN).
- `.set_categories(new_categories, ordered=None, rename=False)`: Задава нови категории, може да зададе ред и опционално да преименува съществуващи категории.
- `.remove_unused_categories()`: Премахва категории, които не се срещат в Series-a.
- `.as_ordered(ordered=True)`: Задава реда на категориите.
- `.as_unordered()`: Премахва реда на категориите.

Използването на аксесора `.cat` е ключово за ефективна работа с категорични данни в Pandas, позволявайки ни да инспектираме, модифицираме и управляваме категориите по удобен начин.

## Пример 1: Достъп до категориите и кодовете

```
import pandas as pd

# Създаваме Series и го преобразуваме в категориален тип
data = pd.Series(['a', 'b', 'c', 'a', 'b', 'a'])
categorical_series = data.astype('category')
print("Оригинален категориален Series:\n", categorical_series)

# Достъп до уникалните категории
categories = categorical_series.cat.categories
print("\nКатегории:", categories)

# Достъп до кодовете на всяка стойност
codes = categorical_series.cat.codes
print("\nКодове:", codes)
```

Този пример показва как да създадем категориален Series и как да използваме `.cat.categories` за да видим уникалните категории и `.cat.codes` за да видим целочисленото представяне на всяка стойност спрямо тези категории.

Разбира се, ето резюме на предоставения пример, който демонстрира основните начини за инспектиране на категориален Pandas Series чрез аксесора `.cat`:

- 1) **Създаване и преобразуване в категориален тип:** Първо, създава се обикновен Pandas Series `data` от низови стойности. След това, този Series се преобразува в категориален тип данни с помощта на метода `.astype('category')`, като резултатът се присвоява на променливата `categorical_series`. Извежда се оригиналният категориален Series.
- 2) **Достъп до уникалните категории (`.cat.categories`):** Аксесорът `.cat` се използва за достъп до свойството `.categories` на `categorical_series`. Това свойство връща Index обект,

съдържащ всички уникални категории, които присъстват в категориалния Series. В този случай, категориите са 'a', 'b' и 'c', подредени лексикографски.

- 3) **Достъп до кодовете (.cat.codes):** Аксесорът .cat се използва също за достъп до свойството .codes. Това свойство връща Pandas Series от цели числа. Всяко цяло число представлява кода, съответстващ на категорията на съответната стойност в оригиналния категориален Series. Кодовете се присвояват на категориите въз основа на тяхното подреждане в .categories. В този пример, 'a' има код 0, 'b' има код 1, а 'c' има код 2.

## Накратко:

Примерът показва как да създадете категориален Series и как да използвате .cat.categories за да видите уникалните категории и .cat.codes за да получите целочисленото представяне на всяка стойност спрямо тези категории. Това е полезно за разбиране на вътрешното представяне на категориалните данни в Pandas.

### Пример 2: Преименуване на категории

```
import pandas as pd

# Създаваме категориален Series
data = pd.Series(['cat', 'dog', 'cat', 'fish'])
categorical_series = data.astype('category')
print("Оригинален категориален Series:\n", categorical_series)

# Преименуваме категориите
new_categories = ['котка', 'куче', 'риба']
renamed_series =
categorical_series.cat.rename_categories(new_categories)
print("\nSeries след преименуване на категориите:\n", renamed_series)
print("\nНови категории:", renamed_series.cat.categories)
```

Тук демонстрираме как да преименуваме съществуващите категории с помощта на .cat.rename\_categories().

- 1) **Създаване на категориален Series:** Първо, създава се Pandas Series data от низови стойности ('cat', 'dog', 'cat', 'fish'). След това, този Series се преобразува в категориален тип данни с помощта на метода .astype('category'), като резултатът се присвоява на променливата categorical\_series. Извежда се оригиналният категориален Series.
- 2) **Дефиниране на нови категории:** Създава се списък new\_categories, съдържащ новите имена на категориите ['котка', 'куче', 'риба']. Важно е да се отбележи, че броят на новите категории трябва да съответства на броя на оригиналните уникални категории и тяхното подреждане в списъка трябва да съответства на подреждането на оригиналните категории (лексикографско по подразбиране).

- 3) **Преименуване на категориите (.cat.rename\_categories()):** Аксесорът .cat се използва за достъп до метода .rename\_categories(). На този метод се подава списъкът с новите имена на категории new\_categories. Методът връща нов категориален Series renamed\_series, в който старите категории са заменени с новите.
- 4) **Извеждане на резултата и новите категории:** Извежда се renamed\_series, показващ стойностите с новите имена на категориите. След това се извежда .cat.categories на renamed\_series, за да се потвърди, че категориите са успешно преименувани.

## Накратко:

Примерът показва как .cat.rename\_categories() позволява лесно да се заменят имената на категориите в категориален Series, което е полезно за привеждане на данните към желана номенклатура или за превод на категориите.

### Пример 3: Добавяне на нови категории

```
import pandas as pd

# Създаваме категориален Series
data = pd.Series(['A', 'B', 'A'])
categorical_series = data.astype('category')
print("Оригинален категориален Series:\n", categorical_series)
print("Категории преди добавяне:", categorical_series.cat.categories)

# Добавяме нови категории
new_categories = ['A', 'B', 'C', 'D']
added_categories_series = categorical_series.cat.add_categories(['C', 'D'])
print("\nSeries след добавяне на категориите:\n",
      added_categories_series)
print("Категории след добавяне:",
      added_categories_series.cat.categories)
```

Този пример показва как да добавим нови категории към съществуващите с .cat.add\_categories(). Забележете, че добавянето на категории не променя съществуващите стойности в Series-а.

- 1) **Създаване на категориален Series:** Първо, създава се Pandas Series data от низови стойности ('A', 'B', 'A'). След това, този Series се преобразува в категориален тип данни с помощта на метода .astype('category'), като резултатът се присвоява на променливата categorical\_series. Извежда се оригиналният категориален Series, както и неговите първоначални категории (които са 'A' и 'B').

- 2) **Дефиниране на нови категории и добавяне:** Създава се списък `new_categories` (въпреки че не се използва директно за добавяне в този пример). След това, методът `.cat.add_categories(['C', 'D'])` се прилага към `categorical_series`, като му се подава списък с новите категории, които трябва да бъдат добавени ('C' и 'D'). Резултатът се присвоява на `added_categories_series`.
- 3) **Извеждане на резултата и новите категории:** Извежда се `added_categories_series`, който показва същите оригинални стойности ('A', 'B', 'A'), но вече с разширен набор от валидни категории. След това се извежда `.cat.categories` на `added_categories_series`, за да се потвърди, че новите категории ('C' и 'D') са успешно добавени към съществуващите ('A' и 'B').

## Накратко:

Примерът показва как `.cat.add_categories()` позволява да се разшири наборът от валидни категории за категориален Series. Важно е да се отбележи, че добавянето на категории не променя съществуващите стойности в Series-a. Ако Series-ът съдържа стойности, които не са в новия набор от категории, те ще станат NaN след използване на методи като `.cat.set_categories()` с `drop=True`.

### Пример 4: Премахване на неизползвани категории

```
import pandas as pd

# Създаваме категориален Series с неизползвана категория
data = pd.Series(['apple', 'banana', 'apple', 'grape'])
categorical_series =
data.astype('category').cat.add_categories(['orange'])
print("Оригинален категориален Series:\n", categorical_series)
print("Категории преди премахване:", categorical_series.cat.categories)

# Премахваме неизползваните категории
removed_unused_series =
categorical_series.cat.remove_unused_categories()
print("\nSeries след премахване на неизползваните категории:\n",
removed_unused_series)
print("Категории след премахване:",
removed_unused_series.cat.categories)
```

Тук показваме как `.cat.remove_unused_categories()` премахва категории, които не се срещат като стойности в Series-a.

- 1) **Създаване на категориален Series с неизползвана категория:** Първо, създава се Pandas Series `data` от низови стойности ('apple', 'banana', 'apple', 'grape'). След това, този Series се преобразува в категориален тип данни с `.astype('category')`, и след това към него се добавя

нова категория 'orange' с помощта на `.cat.add_categories(['orange'])`. Извежда се оригиналният категориален Series, както и неговите категории преди премахването ('apple', 'banana', 'grape', 'orange'). Забележете, че категорията 'orange' не се среща като стойност в самия Series.

- 2) **Премахване на неизползваните категории (`.cat.remove_unused_categories()`):** Аксесорът `.cat` се използва за достъп до метода `.remove_unused_categories()`. Този метод преглежда всички категории, дефинирани за Series-a, и премахва тези, които не се срещат като нито една от стойностите в Series-a. Резултатът е нов категориален Series `removed_unused_series`, в който са останали само категориите, които действително имат съответстващи стойности.
- 3) **Извеждане на резултата и новите категории:** Извежда се `removed_unused_series`, който показва същите оригинални стойности ('apple', 'banana', 'apple', 'grape'), но вече с намален набор от валидни категории. След това се извежда `.cat.categories` на `removed_unused_series`, за да се потвърди, че неизползваната категория 'orange' е била успешно премахната, оставяйки само 'apple', 'banana' и 'grape'.

## Накратко:

Примерът показва как `.cat.remove_unused_categories()` е полезен за почистване на категориални данни, като се премахват излишни дефинирани категории, които не заемат място в самия Series. Това може да подобри ефективността и да опрости представянето на категориите.

### Пример 5: Задаване на ред на категориите

```
import pandas as pd

# Създаваме категориален Series
data = pd.Series(['low', 'high', 'medium', 'low'])
categorical_series = data.astype('category')
print("Оригинален категориален Series (unordered):\n",
      categorical_series)
print("Подреден:", categorical_series.cat.ordered)

# Задаваме ред на категориите
ordered_categories = ['low', 'medium', 'high']
ordered_series =
categorical_series.cat.reorder_categories(ordered_categories,
ordered=True)
print("\nSeries след задаване на ред:\n", ordered_series)
print("Подреден:", ordered_series.cat.ordered)
print("Категории (с ред):", ordered_series.cat.categories)
```



В този пример показваме как да зададем ред на категориите с помощта на `.cat.reorder_categories()` и параметъра `ordered=True`. След като е зададен ред, сравненията между стойностите в Series-а могат да имат смисъл.

- 1) **Създаване на категориален Series (неподреден):** Първо, създава се Pandas Series data от низови стойности ('low', 'high', 'medium', 'low'). След това, този Series се преобразува в категориален тип данни с `.astype('category')`. Извежда се оригиналният категориален Series, както и неговото свойство `.cat.ordered`, което по подразбиране е False, тъй като не е зададен ред на категориите.
- 2) **Дефиниране на подредба на категориите:** Създава се списък `ordered_categories`, който указва желаната подредба на категориите: ['low', 'medium', 'high']. Редът в този списък ще бъде използван като логическа подредба на категориите.
- 3) **Задаване на ред на категориите (`.cat.reorder_categories()`):** Аксесорът `.cat` се използва за достъп до метода `.reorder_categories()`. На този метод се подават два аргумента: `ordered_categories` (списъкът с желаната подредба) и `ordered=True` (булев флаг, който указва, че категориите трябва да бъдат третираны като подредени). Резултатът е нов категориален Series `ordered_series` с зададена подредба на категориите.
- 4) **Извеждане на резултата и свойствата:** Извежда се `ordered_series`, показващ същите оригинални стойности, но вече с асоциирана подредба на категориите. След това се извеждат свойствата `.cat.ordered` (което вече е True) и `.cat.categories` (което показва категориите в зададения ред).

## Накратко:

Примерът показва как `.cat.reorder_categories()` позволява да се зададе логическа подредба на категориите в категориален Series. Това е важно за сравнения (например, `ordered_series[ordered_series > 'low']`) и за някои статистически анализи и визуализации, където редът на категориите има значение.

Тези примери илюстрират някои от основните операции, които могат да бъдат извършвани с категорични данни чрез аксесора `.cat` в Pandas. Той предоставя мощен начин за управление и анализ на данни с ограничен брой повтарящи се стойности.

## Казус 1: Анализ на предпочитания за размер на дрехи (`.cat.categories`, `.cat.codes`)

Представете си, че имате Series, съдържащ информация за предпочитания размер дрехи на клиенти. Искате да анализирате разпределението на размерите и да получите числено представяне на тези предпочитания.

```
import pandas as pd

# Създаваме Series с предпочитани размери дрехи
clothing_sizes = pd.Series(['S', 'M', 'L', 'S', 'XL', 'M', 'S'])
```

```
# Преобразуваме в категориален тип
categorical_sizes = clothing_sizes.astype('category')
print("Оригинален категориален Series:\n", categorical_sizes)

# Казус 1.1: Изведете уникалните размери (категории).
# Казус 1.2: Получете целочислено представяне на всеки размер.
```

## Решение на Казус 1:

```
# 1.1: Достъп до уникалните категории
categories = categorical_sizes.cat.categories
print("\nУникални размери (категории):", categories)

# 1.2: Достъп до кодовете на всяка стойност
codes = categorical_sizes.cat.codes
print("\nЧислено представяне (кодове):", codes)
```

## Казус 2: Преименуване на категории на статус на поръчка (.cat.rename\_categories())

Представете си, че имате Series, съдържащ статуси на поръчки на английски език. Искате да ги преведете на български.

```
import pandas as pd

# Създаваме категориален Series със статуси на поръчки
order_status_en = pd.Series(['Pending', 'Completed', 'Processing',
                              'Completed', 'Pending']).astype('category')
print("Оригинален категориален Series (английски):\n", order_status_en)

# Казус: Преименувайте категориите на български: 'Pending' -> 'Чакаща',
'Completed' -> 'Завършена', 'Processing' -> 'Обработка се'.
```

## Решение на Казус 2:

```
# Създаваме речник за преименуване
translation = {'Pending': 'Чакаща', 'Completed': 'Завършена',
               'Processing': 'Обработка се'}

# Преименуваме категориите
order_status_bg = order_status_en.cat.rename_categories(translation)
print("\nКатегориален Series (български):\n", order_status_bg)
print("\nНови категории:", order_status_bg.cat.categories)
```

### Казус 3: Задаване на ред на категории за оценка (.cat.reorder\_categories(), .cat.as\_ordered())

Представете си, че имате Series, съдържащ оценки (ниски, средни, високи). Искате да зададете логически ред на тези оценки, за да можете да ги сравнявате.

```
import pandas as pd

# Създаваме категориален Series с оценки (без ред по подразбиране)
ratings = pd.Series(['средна', 'висока', 'ниска',
                    'средна']).astype('category')
print("Оригинален категориален Series (unordered):\n", ratings)
print("Подреден:", ratings.cat.ordered)

# Казус: Задайте ред на категориите: 'ниска' < 'средна' < 'висока'.
```

### Решение на Казус 3:

```
# Задаваме желанния ред на категориите
ordered_categories = ['ниска', 'средна', 'висока']
ordered_ratings = ratings.cat.reorder_categories(ordered_categories,
                                                ordered=True)

print("\nКатегориален Series с зададен ред:\n", ordered_ratings)
print("Подреден:", ordered_ratings.cat.ordered)
```

```
print("Категории (с ред):", ordered_ratings.cat.categories)
```

### 3. Казус 4: Добавяне и премахване на категории за тип продукт

`(.cat.add_categories(), .cat.remove_categories())`

Представете си, че имате Series, съдържащ типове продукти. Искате да добавите нова категория и след това да премахнете категория, която вече не се използва.

```
import pandas as pd

# Създаваме категориален Series с типове продукти
product_types = pd.Series(['Електроника', 'Дрехи', 'Храни',
                           'Електроника']).astype('category')
print("Оригинален категориален Series:\n", product_types)
print("Категории преди промени:", product_types.cat.categories)

# Казус 4.1: Добавете категория 'Книги'.
# Казус 4.2: Премахнете категория 'Дрехи'.
```

### Решение на Казус 4:

```
# 4.1: Добавяне на категория
product_types_added = product_types.cat.add_categories(['Книги'])
print("\nSeries след добавяне на 'Книги':\n", product_types_added)
print("Категории след добавяне:", product_types_added.cat.categories)

# 4.2: Премахване на категория
product_types_removed =
product_types_added.cat.remove_categories(['Дрехи'])
print("\nSeries след премахване на 'Дрехи':\n", product_types_removed)
print("Категории след премахване:",
      product_types_removed.cat.categories)
```

Тези казуси илюстрират как аксесорът `.cat` предоставя мощни инструменти за управление и анализ на категорийни данни в Pandas, включително инспектиране на категориите и кодовете, преименуване, задаване на ред, добавяне и премахване на категории. Работата с категорийни данни може да доведе до по-ефективно използване на паметта и по-добро представяне при определени анализи.

## XVI. Използване на `.pipe()` за верижни операции.

Методът `.pipe()` в Pandas е изключително полезен инструмент за създаване на по-четим и организиран код при изпълнение на последователни операции върху `DataFrame` или `Series`. Той позволява да "вмъкнете" функции, които приемат `DataFrame` или `Series` като първи аргумент и връщат модифициран `DataFrame` или `Series`, в средата на верига от операции.

Вместо да пишете дълги и трудно четими вериги от `.method1().method2().method3()`, `.pipe()` позволява да прехвърлите обекта (`DataFrame` или `Series`) към отделни функции, което прави кода по-модулен и по-лесен за разбиране и поддръжка.

### Синтаксис:

```
df.pipe(func, *args, **kwargs)
```

- **func:** Функцията, която ще бъде приложена към `DataFrame` (или `Series`). Първият аргумент на тази функция трябва да бъде самият `DataFrame` (или `Series`).
- **\*args:** Позиционни аргументи, които ще бъдат предадени на `func` след `DataFrame`-а (или `Series`-а).
- **\*\*kwargs:** Ключови аргументи, които ще бъдат предадени на `func`.

### Предимства на използването на `.pipe()`:

- **Подобрява четимостта:** Веригите от операции стават по-лесни за проследяване, тъй като всяка стъпка е представена от извикване на функция.
- **Подобрява организацията на кода:** Логиката за всяка операция може да бъде капсулирана в отделна функция, което прави кода по-модулен и по-лесен за тестване.
- **Позволява използване на външни функции:** Можете лесно да включвате както вградени методи на Pandas, така и свои собствени или такива от други библиотеки във веригата от операции.
- **Улеснява предаването на допълнителни аргументи:** `.pipe()` позволява предаването на допълнителни аргументи към функциите във веригата.

### Пример 1: Проста верига от операции без `.pipe()`

```
import pandas as pd

# Създаваме DataFrame
data = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
df = pd.DataFrame(data)

# Верига от операции без .pipe()
result = (df
          .assign(col3=df['col1'] + df['col2']))
```

```
.query('col3 > 5')
.rename(columns={'col3': 'сума'})
)
print(result)
```

*Този пример показва стандартна верига от Pandas методи.*

Разбира се, ето резюме на предоставения пример, който демонстрира стандартна верига от операции върху Pandas DataFrame без използването на метода `.pipe()`:

- 1) **Създаване на DataFrame:** Първо, създава се Pandas DataFrame `df` с две колони: `'col1'` и `'col2'`, съдържащи съответно стойностите `[1, 2, 3]` и `[4, 5, 6]`.
- 2) **Верига от операции:** В скоби се прилагат последователно няколко метода към DataFrame-a:
  - `.assign(col3=df['col1'] + df['col2'])`: Създава се нова колона `'col3'`, чиито стойности са резултат от сумирането на стойностите от колоните `'col1'` и `'col2'` за всеки ред.
  - `.query('col3 > 5')`: Филтрират се редовете на DataFrame-a, като се запазват само тези, за които стойността в колона `'col3'` е по-голяма от 5.
  - `.rename(columns={'col3': 'сума'})`: Преименува се колона `'col3'` на `'сума'`.
- 3) **Извеждане на резултата:** Крайният резултат от тази верига от операции се присвоява на променливата `result` и след това се извежда на конзолата. Резултатът ще бъде нов DataFrame, съдържащ само реда, където сумата на `'col1'` и `'col2'` е по-голяма от 5, и колоната ще бъде преименувана на `'сума'`. В този случай, това ще бъде редът с оригинални стойности `col1=2, col2=5` (сума 7) и `col1=3, col2=6` (сума 9), с колона, наречена `'сума'`.

## Накратко:

Примерът показва как множество операции за трансформация и филтриране могат да бъдат верижно приложени към DataFrame с помощта на последователни извиквания на методи. Въпреки че този синтаксис е често срещан, методът `.pipe()` (както беше показано в следващите примери от темата) предлага алтернативен начин за организиране на такива вериги от операции, което може да подобри четимостта и поддръжката на кода, особено при по-сложни обработки.

## Пример 2: Същата верига от операции с `.pipe()`

```
import pandas as pd

# Създаваме същия DataFrame
data = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
df = pd.DataFrame(data)

# Дефинираме функции за всяка операция
def add_column(df):
    return df.assign(col3=df['col1'] + df['col2'])
```

```
def filter_rows(df, threshold):
    return df.query(f'col3 > {threshold}')
```

```
def rename_column(df, old_name, new_name):
    return df.rename(columns={old_name: new_name})
```

```
# Верига от операции с .pipe()
result = (df
    .pipe(add_column)
    .pipe(filter_rows, threshold=5)
    .pipe(rename_column, old_name='col3', new_name='сума')
)
print(result)
```

*В този пример всяка операция е изнесена в отделна функция, която се прилага към DataFrame-a чрез .pipe(). Това прави кода по-структуриран и четим.*

- 1) **Създаване на DataFrame:** Създава се същият Pandas DataFrame df с колони 'col1' ([1, 2, 3]) и 'col2' ([4, 5, 6]).
- 2) **Дефиниране на функции за всяка операция:** Дефинират се три отделни функции, всяка от които капсулира една от операциите, които искаме да приложим към DataFrame-a:
  - o add\_column(df): Приема DataFrame като аргумент и връща нов DataFrame с добавена колона 'col3', която е сума от 'col1' и 'col2'.
  - o filter\_rows(df, threshold): Приема DataFrame и праг като аргументи и връща нов DataFrame, съдържащ само редовете, за които стойността в колона 'col3' е по-голяма от зададения праг. Използва f-string за динамично вмъкване на прага в query стринга.
  - o rename\_column(df, old\_name, new\_name): Приема DataFrame, старо име на колона и ново име на колона като аргументи и връща нов DataFrame с преименувана колона.
- 3) **Верига от операции с .pipe():** Методът .pipe() се използва за последователно прилагане на тези функции към DataFrame-a:
  - o .pipe(add\_column): Първо, функцията add\_column се прилага към df. Резултатът (DataFrame с колона 'col3') се предава на следващия .pipe().
  - o .pipe(filter\_rows, threshold=5): След това, функцията filter\_rows се прилага към резултата от add\_column, като се подава и допълнителният аргумент threshold=5. Резултатът (DataFrame, филтриран по стойности в 'col3') се предава на следващия .pipe().
  - o .pipe(rename\_column, old\_name='col3', new\_name='сума'): Накрая, функцията rename\_column се прилага към резултата от filter\_rows, като се подават и допълнителните аргументи old\_name='col3' и new\_name='сума'.
- 4) **Извеждане на резултата:** Крайният резултат от тази верига от .pipe() извиквания се присвоява на променливата result и се извежда на конзолата. Резултатът ще бъде същият като в предходния пример (DataFrame с филтрирани редове и преименувана колона), но кодът е по-организиран и по-лесен за разбиране.



## Накратко:

Примерът илюстрира как `.pipe()` позволява да се създават по-четими и модулни вериги от операции върху `DataFrame`-и, като се използват отделни функции за всяка стъпка. Това улеснява разбирането на логиката и поддръжката на кода.

### Пример 3: Използване на *lambda* функции с `.pipe()`

```
import pandas as pd

# Създаваме DataFrame
data = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
df = pd.DataFrame(data)

# Вери́га от операции с .pipe() и lambda функции
result = (df
          .pipe(lambda x: x.assign(col3=x['col1'] * 2))
          .pipe(lambda x: x[x['col3'] < 6])
          .pipe(lambda x: x.rename(columns={'col3': 'удвоено'})))

print(result)
```

*.pipe()* може да се използва и с анонимни (*lambda*) функции за по-кратки операции, които не е необходимо да бъдат дефинирани отделно.

Разбира се, ето резюме на предоставения пример, който демонстрира използването на метода `.pipe()` във верига от операции върху Pandas `DataFrame`, като този път операциите се дефинират директно с помощта на анонимни (*lambda*) функции:

- 1) **Създаване на `DataFrame`:** Създава се Pandas `DataFrame` `df` с колони 'col1' ([1, 2, 3]) и 'col2' ([4, 5, 6]).
- 2) **Вери́га от операции с `.pipe()` и *lambda* функции:** Методът `.pipe()` се използва последователно, като на всяко извикване се подава анонимна функция (*lambda* функция), която дефинира съответната операция:
  - `.pipe(lambda x: x.assign(col3=x['col1'] * 2))`: Първата *lambda* функция приема `DataFrame` (който е представен с `x`) и връща нов `DataFrame` с добавена колона 'col3', чиито стойности са резултат от умножаването на стойностите в колона 'col1' по 2.
  - `.pipe(lambda x: x[x['col3'] < 6])`: Втората *lambda* функция приема резултата от предходната операция (отново представен с `x`) и връща нов `DataFrame`, съдържащ само редовете, за които стойността в колона 'col3' е по-малка от 6.
  - `.pipe(lambda x: x.rename(columns={'col3': 'удвоено'}))`: Третата *lambda* функция приема резултата от предходната операция и връща нов `DataFrame`, в който колона 'col3' е преименувана на 'удвоено'.
- 3) **Извеждане на резултата:** Крайният резултат от тази верига от `.pipe()` извиквания се присвоява на променливата `result` и се извежда на конзолата. Резултатът ще бъде `DataFrame`,

съдържащ редовете, където удвоената стойност на 'coll' е по-малка от 6, с колона, наречена 'удвоено'. В този случай, това ще бъдат редовете с оригинални 'coll' стойности 1 и 2.

## Накратко:

Примерът показва, че `.pipe()` може да бъде използван и с кратки, еднократни операции, дефинирани директно като `lambda` функции. Това може да бъде по-компактен начин за верижно прилагане на прости трансформации, без да е необходимо да се дефинират отделни именувани функции. Въпреки това, за по-сложни операции, дефинирането на отделни функции често подобрява четимостта.

`.pipe()` е мощен инструмент за създаване на по-чист и по-организиран код при работа с **Pandas**, особено когато се изпълняват множество последователни операции. Той насърчава функционалния стил на програмиране и подобрява поддръжката на кода.

### Казус 1: Предварителна обработка на данни за продажби (`.pipe()` с дефинирани функции)

Представете си, че имате суров `DataFrame` с данни за продажби, който трябва да бъде обработен през няколко стъпки: премахване на дублиращи се редове, филтриране на продажби под определена сума и преименуване на колони.

```
import pandas as pd

# Създаваме суров DataFrame с данни за продажби
raw_sales_data = {
    'product_id': [101, 102, 101, 103, 104, 102],
    'quantity': [2, 1, 2, 3, 1, 1],
    'price': [50.00, 120.00, 50.00, 85.50, 200.00, 120.00],
    'customer_id': ['A', 'B', 'A', 'C', 'D', 'B']
}

df_raw_sales = pd.DataFrame(raw_sales_data)
print("Суров DataFrame:\n", df_raw_sales)

# Казус: Използвайте .pipe() за да извършите следните стъпки:
# 1. Премахнете дублиращите се редове.
# 2. Филтрирайте продажбите, където общата стойност (quantity * price) е над 100 лв.
```

```
# 3. Преименувайте колоните на 'ID на продукт', 'Количество', 'Цена',  
'ID на клиент'.
```

## Решение на Казус 1:

```
# Дефинираме функции за всяка стъпка от обработката  
def remove_duplicates(df):  
    return df.drop_duplicates()  
  
def filter_high_value_sales(df, min_value):  
    df['total_value'] = df['quantity'] * df['price']  
    filtered_df = df[df['total_value'] >  
min_value].drop(columns=['total_value'])  
    return filtered_df  
  
def rename_columns(df):  
    return df.rename(columns={  
        'product_id': 'ID на продукт',  
        'quantity': 'Количество',  
        'price': 'Цена',  
        'customer_id': 'ID на клиент'  
    })  
  
# Използваме .pipe() за верижно изпълнение на функциите  
df_processed_sales = (df_raw_sales  
                        .pipe(remove_duplicates)  
                        .pipe(filter_high_value_sales, min_value=100)  
                        .pipe(rename_columns)  
                        )  
  
print("\nОбработен DataFrame:\n", df_processed_sales)
```

*В този казус, всяка стъпка от предварителната обработка е капсулирана в отделна функция, която приема DataFrame като първи аргумент и връща обработен DataFrame. Методът .pipe() позволява тези функции да бъдат верижно приложени към оригиналния DataFrame, което прави кода по-четим и по-лесен за разбиране и поддръжка.*

## Казус 2: Анализ на резултати от тест с условно премахване на слаби предмети (`.pipe()` с `lambda` функции)

Представете си, че имате `DataFrame` с резултати от тест на ученици по няколко предмета. Искате да изчислите средния резултат на всеки ученик, но преди това условно да премахнете резултатите по предмети, по които ученикът има оценка под 50.

```
import pandas as pd

# Създаваме DataFrame с резултати от тест
grades_data = {
    'ученик': ['Алекс', 'Борис', 'Вера'],
    'математика': [70, 45, 88],
    'физика': [80, 60, 78],
    'химия': [55, 30, 92]
}

df_grades = pd.DataFrame(grades_data).set_index('ученик')
print("Оригинален DataFrame с резултати:\n", df_grades)

# Казус: Използвайте .pipe() с lambda функции за:
# 1. Създаване на функция, която приема DataFrame и премахва колони
#    (предмети),
#    за които резултатът на ученика е под 50.
# 2. Изчисляване на средния резултат на всеки ученик след евентуалното
#    премахване на слаби предмети.
```

### Решение на Казус 2:

```
# Използваме .pipe() с lambda функция за условно премахване на слаби
# предмети и изчисляване на среден резултат
df_final_grades = (df_grades
    .pipe(lambda df: df.apply(lambda row: row[row >= 50],
axis=1))
    .pipe(lambda df:
df.mean(axis=1).rename('среден_резултат'))
)
```

```
print("\nСредни резултати след условно премахване на слаби предмети:\n",  
df_final_grades)
```

В този казус, първата `.pipe()` използва `lambda` функция, която прилага функция към всеки ред. Тази функция запазва само оценките, които са по-големи или равни на 50, като ефективно премахва слабите предмети за всеки ученик (тези стойности стават NaN). Втората `.pipe()` използва `lambda` функция, която изчислява средната стойност по редове (`axis=1`) на резултата от предходната стъпка и преименува получената `Series` на 'среден резултат'.

Тези казуси илюстрират гъвкавостта на `.pipe()` както при използване на отделно дефинирани функции за по-сложни стъпки, така и при използване на кратки `lambda` функции за по-бързи и директни трансформации във веригата от операции. `.pipe()` помага за създаване на по-подреден и четим код при работа с `Pandas`.

## XVII. Въпроси и Задачи

### I. Векторизирани операции (аритметични, сравнителни, логически)

1. Даден е `DataFrame` с колони 'A' и 'B' с числови стойности. Напишете код, който създава нова колона 'C', съдържаща сумата на стойностите от колони 'A' и 'B'.
2. Имате `Series` с резултати от тест (числа). Напишете код, който връща булев `Series`, указващ кои резултати са по-високи от средната стойност на всички резултати.
3. Даден е `DataFrame` с колони 'статус' (текст) и 'цена' (числа). Напишете код, който филтрира `DataFrame`-а, запазвайки само редовете, където статусът е 'Активен' и цената е по-малка от 100.

### II. Прилагане на функции

- а) Елемент по елемент (`.map()` за `Series`, `.applymap()` за `DataFrame`)
  4. Имате `Series` с имена на градове. Напишете код, който преобразува всички имена в горни букви, използвайки `.map()` и `lambda` функция.
  5. Даден е `DataFrame` с числови стойности. Напишете код, който форматира всяка стойност като низ с две десетични места, използвайки `.map()` и `f-string`.
- б) По редове/колони (`.apply()`)
  6. Даден е `DataFrame` с колони 'цена1', 'цена2', 'цена3'. Напишете код, който създава нова колона 'средна\_цена', съдържаща средната стойност от тези три колони за всеки ред, използвайки `.apply()` с `axis=1`.
  7. Имате `DataFrame` с числови колони. Напишете код, който намира максималната стойност за всяка колона, използвайки `.apply()` с `axis=0`.
- в) Използване на `lambda` функции

8. Даден е `Series` с оценки (числа). Напишете код, който създава нов `Series`, където всяка оценка се увеличава с 5, използвайки `.map()` и `lambda` функция.

9. Имате `DataFrame` с колона 'име'. Напишете код, който създава нова колона 'първа\_буква', съдържаща първата буква от всяко име, използвайки `.apply()` с `lambda` функция.

### III. Създаване на нови колони

- а) С константни стойности

10. Даден е `DataFrame`. Напишете код, който добавя нова колона 'валута' с константна стойност 'USD'.

- б) На базата на съществуващи колони (чрез операции и функции)

11. Имате `DataFrame` с колони 'брой' и 'единична\_цена'. Напишете код, който създава нова колона 'обща\_стойност', като умножава стойностите от тези две колони.

12. Даден е `DataFrame` с колона 'дата\_раждане' (`datetime` тип). Напишете код, който създава нова колона 'година\_раждане', съдържаща годината от датата на раждане.

- в) Методът `.assign()`

13. Даден е `DataFrame` с колона 'цена'. Използвайте `.assign()` за да добавите нова колона 'цена\_с\_ддс', която е цената, умножена по 1.20.

14. Имате `DataFrame` с колони 'първо\_име' и 'фамилия'. Използвайте `.assign()` за да добавите нова колона 'пълно\_име', съдържаща обединените първо и фамилно име (разделени с интервал).

- г) Условно създаване на колони (`np.where`, `.loc`)

15. Даден е `DataFrame` с колона 'резултат' (числа). Напишете код, който създава нова колона 'статус', която е 'Успех', ако резултатът е  $\geq 60$ , и 'Провал' в противен случай (използвайте `np.where`).

16. Имате `DataFrame` с колона 'категория' и 'цена'. Напишете код, който създава нова колона 'отстъпка', която е 10% от цената, само за редовете, където категорията е 'Промоция' (използвайте `.loc`).

### IV. Преименуване на колони и индекси

17. Даден е `DataFrame` с колони 'old\_name\_1' и 'old\_name\_2'. Напишете код, който ги преименува на 'new\_name\_1' и 'new\_name\_2' съответно, използвайки `.rename()`.

18. Имате `DataFrame` с индекс, който няма име. Напишете код, който задава име 'ID' на индекса, използвайки `.rename_axis()`.

19. Даден е `DataFrame` с колони, които са числови индекси (0, 1, 2). Напишете код, който ги преименува на 'Колона А', 'Колона В', 'Колона С', използвайки `.set_axis()`.

### V. Задаване и нулиране на индекса

20. Даден е `DataFrame` с колона 'ID'. Напишете код, който задава тази колона като индекс на `DataFrame-a`, използвайки `.set_index()`.
21. Имате `DataFrame` с индекс, който искате да превърнете обратно в обикновена колона. Напишете код, който нулира индекса, използвайки `.reset_index()`.

## VI. Сортиране на данни

### • а) Сортиране по стойности (`.sort_values()`)

22. Даден е `DataFrame` с колони 'име' и 'възраст'. Напишете код, който сортира `DataFrame-a` по възраст в низходящ ред.
23. Имате `DataFrame` с колони 'дата' и 'приход'. Напишете код, който сортира `DataFrame-a` първо по дата (възходящо), а след това по приход (низходящо).
24. Даден е `Series` с числови стойности, включващ `NaN`. Напишете код, който сортира `Series-a` във възходящ ред, като `NaN` стойностите бъдат поставени в края.

### • б) Сортиране по индекс (`.sort_index()`)

25. Даден е `DataFrame` с индекс от дати. Напишете код, който сортира `DataFrame-a` по дати във възходящ ред.
26. Имате `DataFrame` с мултииндекс (нива 'регион' и 'продукт'). Напишете код, който сортира `DataFrame-a` по ниво 'продукт' във възходящ ред.

## VII. Промяна на типа данни на колони

27. Даден е `Series` с числови стойности, прочетени като низове. Напишете код, който ги преобразува в числа (`float`), като обработи евентуални грешки, замествайки ги с `NaN`.
28. Имате `DataFrame` с колона 'дата' във формат 'гггг-мм-дд'. Напишете код, който преобразува тази колона в `datetime` тип.
29. Даден е `Series` с времеви разлики във формат 'HH:MM:SS'. Напишете код, който го преобразува в `Timedelta` тип.
30. Имате `DataFrame` с колона 'категория' (низови стойности с много повторения). Напишете код, който преобразува тази колона в категориален тип.

## VIII. Работа с текстови данни (`.str`)

31. Даден е `Series` с имейл адреси. Напишете код, който извлича домейн името (частта след '@') от всеки имейл адрес.
32. Имате `DataFrame` с колона 'текст'. Напишете код, който преброява колко пъти се среща думата 'важно' (без значение от регистъра) във всеки текст.
33. Даден е `Series` с телефонни номера в различни формати. Напишете код, който ги стандартизира, като премахне всички нецифрови символи.

## IX. Работа с данни за дата и час (`.dt`)

34. Даден е `Series` от `datetime` обекти. Напишете код, който извлича годината от всяка дата.
35. Имате `DataFrame` с колона 'време' (`datetime` тип). Напишете код, който създава нова колона 'ден\_от\_седмичата' (текст), съдържаща името на деня от седмицата за всяка дата.



36. Даден е `Series` от `datetime` обекти. Напишете код, който изчислява разликата в дни между всяка дата и днешната дата.

#### **X. Работа с категорийни данни (`.cat`)**

37. Даден е категориален `Series` с размери ('S', 'M', 'L'). Напишете код, който добавя категория 'XL'.
38. Имате категориален `Series` със статуси ('Ниско', 'Средно', 'Високо'). Напишете код, който задава ред на категориите: 'Ниско' < 'Средно' < 'Високо'.
39. Даден е категориален `Series` с цветове ('червен', 'син', 'зелен', 'червен'). Напишете код, който преименува категория 'червен' на 'ален'.

#### **XI. Използване на `.pipe()` за верижни операции**

40. Даден е `DataFrame`. Използвайте `.pipe()` за да приложите последователно две функции: първата, която филтрира редовете, където стойността в колона 'A' е по-голяма от 10, и втората, която преименува колона 'B' на 'Ново име'.
41. Създайте сценарий, в който `.pipe()` значително подобрява четимостта на кода при сложна последователност от операции върху `DataFrame`.