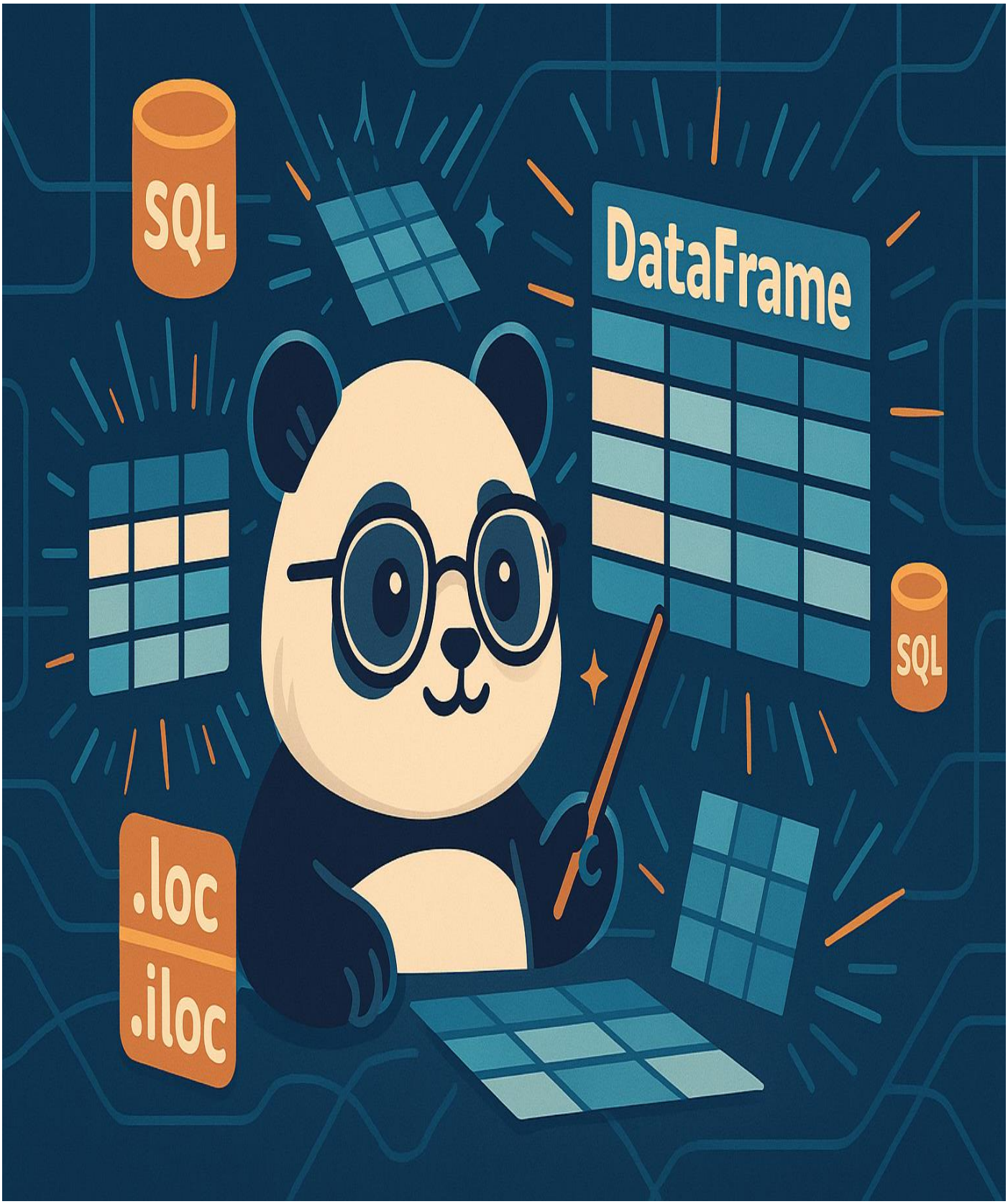
Глава IV. Индексиране и Селектиране на Данни



Индексиране и Селектиране на Данни. Това е фундаментално умение за ефективна работа с Pandas, тъй като често ще се налага да извличате, филтрирате и модифицирате конкретни части от вашите DataFrame-и и Series-и.

В тази глава ще разгледаме различните начини за достъп до данни в Pandas, като ще обърнем специално внимание на предимствата и недостатъците на всеки метод, както и на потенциалните опасности и най-добрите практики. Ето кратко обобщение на темите, които ще покрием:

1. **Селектиране на колони (`[]`, `.` `нотация` - внимание):** Ще започнем с най-основните начини за избиране на една или няколко колони от DataFrame, като ще разгледаме както използването на квадратни скоби (`[]`), така и достъпа чрез точкова нотация (атрибут). Ще обърнем специално внимание на случаите, когато точковата нотация може да бъде подвеждаща или да не работи.
2. **Селектиране на редове чрез слайсинг (`[]`):** Ще видим как можем да избираме определени редове от DataFrame, използвайки познатия от Python синтаксис за слайсинг. Важно е да се отбележи, че при DataFrame, слайсингът без конкретно указване на колони обикновено се прилага върху редовете.
3. **Селектиране по етикет (`.loc`):** Това е един от най-мощните и гъвкави методи за селектиране на данни в Pandas. Ще разгледаме как да избираме единични или множество редове и колони по техните етикети (имена на индекси и колони), както и как да използваме слайсинг по етикет и булеви масиви с `.loc`.
4. **Селектиране по позиция (`.iloc`):** Подобно на `.loc`, но вместо етикети, `.iloc` използва целочислени позиции (базирани на нула) за селектиране на редове и колони. Ще разгледаме селектиране на единични елементи, множества и слайсинг по позиция.
5. **Комбинирано селектиране (Опасности и алтернативи):** Ще обсъдим случаите, когато може да се опитаме да комбинираме различни методи за селектиране (например, верижно индексиране като `df['колона'][ред]`) и ще подчертаем потенциалните опасности и по-надеждните алтернативи, като използването на `.loc` и `.iloc`.
6. **Булево индексиране (логически условия):** Ще се науча как да използваме логически условия за филтриране на редове в DataFrame въз основа на стойностите в една или няколко колони. Ще разгледаме операторите за сравнение, логическите оператори (`&`, `|`, `~`, `^`), както и полезните методи `.isin()` и `.between()`.
7. **Методът `.query()` за селекция чрез низ:** Ще представим един по-експресивен начин за филтриране на DataFrame-и, използвайки низови изрази, които наподобяват SQL `WHERE` клауза.
8. **Бърз скаларен достъп (`.at`, `.iat`):** За бърз достъп и задаване на стойности на единични скаларни стойности (един елемент) в DataFrame или Series, ще разгледаме специализираните атрибути `.at` (по етикет) и `.iat` (по позиция).
9. **Индексиране с извикваеми обекти (callables):** Ще видим как можем да използваме функции или други извикваеми обекти като аргументи при селектиране с `.loc` и `.iloc`, което добавя още по-голяма гъвкавост.
10. **Задаване на стойности чрез селекция:** В края на главата ще разгледаме как можем да използваме методите за селектиране, за да променяме стойности в DataFrame-а въз основа на определени условия или местоположения.

Тази глава ще ви даде солидни познания за различните начини за достъп и манипулация на данни в Pandas, което е абсолютно необходимо за всякакъв по-нататъшен анализ.

I. Селектиране на колони ([], . нотация - внимание)

В Pandas има няколко начина да изберете една или повече колони от DataFrame. Ще разгледаме два основни подхода: използване на квадратни скоби (`[]`) и достъп чрез точкова нотация (`.`). Важно е да бъдем внимателни при използването на точкова нотация, тъй като тя има своите ограничения.

1. Селектиране на колони с квадратни скоби (`[]`)

Това е най-често използваният и най-гъвкав начин за селектиране на колони.

a) Селектиране на една колона:

За да изберете една колона, поставете името ѝ като стринг във вътрешността на квадратните скоби след името на DataFrame-а. Резултатът ще бъде Pandas Series, съдържащ данните от тази колона, запазвайки индекса на оригиналния DataFrame.

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'продукт': ['А', 'Б', 'А', 'В'],
        'цена': [10.5, 20.0, 10.5, 15.0],
        'количество': [5, 2, 3, 1]}
df = pd.DataFrame(data)

# Селектиране на колоната 'продукт'
product_column = df['продукт']
print("Селектирана колона 'продукт' (Series):\n", product_column)
print("\nТип на резултата:", type(product_column))

# Селектиране на колоната 'цена'
price_column = df['цена']
print("\nСелектирана колона 'цена' (Series):\n", price_column)
```

б) Селектиране на множество колони:

За да изберете множество колони, поставете списък от имената на колоните (като стрингове) във вътрешността на квадратните скоби. Резултатът ще бъде нов Pandas DataFrame, съдържащ само избраните колони и запазващ индекса на оригиналния DataFrame.

```
# Селектиране на колоните 'продукт' и 'цена'
multiple_columns = df[['продукт', 'цена']]
print("\nСелектирани колони 'продукт' и 'цена' (DataFrame):\n",
multiple_columns)
print("\nТип на резултата:", type(multiple_columns))

# Редът на имената на колоните в списъка определя реда на колоните в
новия DataFrame.
multiple_columns_another_row = df[['цена', 'продукт']]
print("\nСелектирани колони 'цена' и 'продукт' (DataFrame) с друг
ред:\n", multiple_columns_another_row)
```

2. Селектиране на колони с точкова нотация (.) - Внимание!

В някои случаи е възможно да достъпите колони на DataFrame като атрибути, използвайки точкова нотация след името на DataFrame-а. Това работи само ако името на колоната е валиден Python идентификатор (не съдържа интервали, не започва с цифра и не е запазена ключова дума на Python). Резултатът е винаги Pandas Series.

```
# Селектиране на колоната 'цена' с точкова нотация
price_column_dot = df.цена
print("\nСелектирана колона 'цена' (Series) с точкова нотация:\n",
price_column_dot)
print("\nТип на резултата:", type(price_column_dot))
```

- **Внимание! Ограничения и потенциални проблеми при точковата нотация:**
    - **Имена на колони с интервали или специални символи:** Ако името на колоната съдържа интервали, специални символи или не е валиден Python идентификатор, няма да можете да го достъпите с точкова нотация. Например, `df.име на колона с интервали` ще доведе до грешка `AttributeError`.

```
data_with_space = {'име на продукт': ['А', 'Б', 'А', 'В'],
                   'цена (лв)': [10.5, 20.0, 10.5, 15.0]}
df_space = pd.DataFrame(data_with_space)
```

```
# df_space.име на продукт  # Това ще доведе до AttributeError
print("\nСелектиране на колона с интервал с квадратни скоби:\n",
df_space['име на продукт'])
```

- **Имена на колони, съвпадащи с атрибути на DataFrame:** Ако имате колона с име, което съвпада с вграден атрибут или метод на DataFrame (например, `shape`, `index`, `head`), точковата нотация ще върне този атрибут/метод, а не колоната. Това може да доведе до объркване и неочаквано поведение.

```
data_with_shape_col = {'продукт': ['А', 'Б'], 'shape': [(1, 2), (3,)],
'цена': [10, 20]}
df_shape_col = pd.DataFrame(data_with_shape_col)

print("\nДостъп до атрибута 'shape' на DataFrame:\n",
df_shape_col.shape)
print("\nОпит за достъп до колоната 'shape' с точкова нотация (връща
атрибута):\n", df_shape_col.shape)
print("\nДостъп до колоната 'shape' с квадратни скоби:\n",
df_shape_col['shape'])
```

- **Динамично селектиране:** Точковата нотация не е подходяща, когато името на колоната, която искате да селектирате, се съхранява в променлива. В такива случаи трябва да използвате квадратни скоби.

```
име_на_колона = 'цена'
колона = df[име_на_колона]  # Правилен начин
# колона_грешно = df.име_на_колона  # Грешен начин (ще търси атрибут
'име_на_колона')
print("\nСелектиране на колона чрез променлива с квадратни скоби:\n",
колона)
```

- **Препоръка:**

Въпреки че точковата нотация може да изглежда по-кратка и удобна в някои случаи, **препоръчително е да се използва квадратни скоби (`[]`) за селектиране на колони**, тъй като този метод е по-гъвкав и избягва потенциалните проблеми, свързани с имената на колоните и конфликтите с атрибути на DataFrame. Квадратните скоби са също така единственият начин за селектиране на множество колони едновременно.

II. Селектиране на редове чрез слайсинг ([])

Подобно на работата със списъци и други последователности в Python, можем да използваме оператора за слайсинг (`[:]`) върху DataFrame, за да извлечем определен диапазон от редове. Когато се използва самостоятелно (без указване на колони), слайсингът върху DataFrame се прилага върху редовете.

1. Синтаксис:

```
dataframe[start:stop:step]
```

където:

- `start`: Индексът на първия ред, който ще бъде включен (по подразбиране е 0).
- `stop`: Индексът на последния ред, който **няма** да бъде включен.
- `step`: Стъпката между редовете (по подразбиране е 1).

Важно е да се отбележи, че когато се използва слайсинг по този начин, той работи **по подразбиране с целочислените позиции (индексите) на редовете**, дори ако DataFrame-ът има друг тип индекс (например, стрингов или DatetimeIndex).

2. Примери (използвайки DataFrame `df` от предната тема):

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0],
        'количество': [5, 2, 3, 1, 4, 6]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Селектиране на първите 3 реда (индекси 0, 1, 2)
първи_три_реда = df[0:3]
print("\nПървите 3 реда:\n", първи_три_реда)

# Селектиране на редове от индекс 2 до края
от_трети_ред_накрая = df[2:]
print("\nРедове от индекс 2 до края:\n", от_трети_ред_накрая)
```

```
# Селектиране на всеки втори ред
всеки_втори_ред = df[::2]
print("\nВсеки втори ред:\n", всеки_втори_ред)

# Селектиране на последните 2 реда (използване на отрицателен индекс)
последни_два_реда = df[-2:]
print("\nПоследните 2 реда:\n", последни_два_реда)

# Селектиране на всички редове в обратен ред
обратен_ред = df[::-1]
print("\nВсички редове в обратен ред:\n", обратен_ред)
```

3. Важно:

- Резултатът от слайсинг на DataFrame е винаги нов DataFrame (view или copy, в зависимост от конкретния случай, но най-добре е да се третира като нов обект).
- Слайсингът с `[]` работи **по позиция (целочислен индекс)**, независимо от етикетите на индекса на DataFrame-а. Ако DataFrame-ът има нечислов индекс, този метод все пак ще използва подлежащите целочислени позиции.

```
# Създаваме DataFrame с нечислов индекс
df_non_numeric_index = df.set_index('продукт')
print("\nDataFrame с нечислов индекс:\n", df_non_numeric_index)

# Слайсингът все още работи по целочислена позиция, а не по етикет 'А',
'Б' и т.н.
първите_два_реда_нечислов_индекс = df_non_numeric_index[0:2]
print("\nПървите два реда (по позиция) на DataFrame с нечислов
индекс:\n", първите_два_реда_нечислов_индекс)
```

4. Кога е полезен слайсингът с `[]` за редове?

- Когато искате да извлечете определен брой от първите или последните редове.
- Когато искате да вземете подмножество от редове въз основа на тяхната позиция в DataFrame.
- За прости операции за разделяне на DataFrame-а на части.

Въпреки че слайсингът с `[]` е удобен за селектиране на редове по позиция, за по-гъвкаво селектиране по етикет (стойностите на индекса) или за едновременно селектиране на редове и колони, ще използваме методите `.loc` и `.iloc`, които ще разгледаме по-късно.

III. Селектиране по етикет (.loc)

Методът `.loc` е изключително мощен и гъвкав инструмент за селектиране на данни в Pandas DataFrame по етикетите на редовете (индекса) и колоните. Синтаксисът на `.loc` е следният:

```
dataframe.loc[row_labels, column_labels]
```

където:

- `row_labels`: Може да бъде единичен етикет, списък от етикети, слайс от етикети или булев масив.
- `column_labels`: Подобно на `row_labels`, може да бъде единичен етикет, списък от етикети или слайс от етикети. Ако искате да селектирате всички колони, можете да използвате `:`.

1. Селектиране на редове по етикет:

a) Селектиране на единичен ред:

За да селектирате един ред, подайте етикета на този ред (стойността на индекса) като първи аргумент на `.loc`. Резултатът ще бъде Pandas Series, където имената на колоните са индекс, а стойностите са данните от съответната колона за този ред.

```
import pandas as pd

# Създаваме примерен DataFrame с нечислов индекс
data = {'продукт': ['A', 'B', 'A', 'B'],
        'цена': [10.5, 20.0, 10.5, 15.0],
        'количество': [5, 2, 3, 1]}
df = pd.DataFrame(data, index=['ред_1', 'ред_2', 'ред_3', 'ред_4'])
print("Оригинален DataFrame с нечислов индекс:\n", df)

# Селектиране на реда с етикет 'ред_2'
row_2 = df.loc['ред_2']
print("\nСелектиран ред с етикет 'ред_2' (Series):\n", row_2)
print("\nТип на резултата:", type(row_2))
```

б) Селектиране на множество редове:

За да селектирате множество редове, подайте списък от етикети на редовете като първи аргумент на `.loc`. Резултатът ще бъде нов DataFrame, съдържащ само избраните редове.

```
# Селектиране на редовете с етикети 'ред_1' и 'ред_3'
multy_row= df.loc[['ред_1', 'ред_3']]
print("\nСелектирани редове с етикети 'ред_1' и 'ред_3' (DataFrame):\n",
multy_row)
print("\nТип на резултата:", type(multy_row))
```

2. Селектиране на колони по етикет:

За да селектирате колони, използвате втория аргумент на `.loc`.

а) Селектиране на единична колона:

За да селектирате една колона, подайте името на колоната като стринг като втори аргумент на `.loc`, като за първия аргумент използвате `:` (което означава "всички редове").

```
# Селектиране на колоната 'цена' за всички редове
column_price_loc= df.loc[:, 'цена']
print("\nСелектирана колона 'цена' (Series) с .loc:\n",
column_price_loc)
print("\nТип на резултата:", type(column_price_loc))
```

б) Селектиране на множество колони:

За да селектирате множество колони, подайте списък от имената на колоните като втори аргумент на `.loc`, като за първия аргумент използвате `:`.

```
# Селектиране на колоните 'продукт' и 'количество' за всички редове
multiple_columns_loc = df.loc[:, ['продукт', 'количество']]
print("\nСелектирани колони 'продукт' и 'количество' (DataFrame) с
.loc:\n", multiple_columns_loc)
print("\nТип на резултата:", type(multiple_columns_loc))
```

в) Селектиране на конкретни клетки (комбинация от редове и колони):

Можете да селектирате конкретни клетки или подмножества от данни, като комбинирате селекция на редове и колони по техните етикети.

г) Селектиране на единична клетка:

Подайте етикета на реда и етикета на колоната.

```
# Селектиране на стойността в ред 'ред_2' и колона 'цена'
value = df.loc['ред_2', 'цена']
print("\nСтойност в ред 'ред_2', колона 'цена':", value)
print("\nТип на резултата:", type(value))
```

д) Селектиране на множество клетки:

Можете да използвате списъци за етикети както за редове, така и за колони.

```
# Селектиране на стойностите за редове 'ред_1' и 'ред_3' в колоните
'цена' и 'количество'
subset = df.loc[['ред_1', 'ред_3'], ['цена', 'количество']]
print("\nПодмножество от данни:\n", subset)
print("\nТип на резултата:", type(subset))
```

Методът `.loc` е изключително важен, защото прави селекцията на данни много ясна и лесна за разбиране, тъй като се базира на видимите етикети на DataFrame-а.

3. Селектиране чрез слайсиране по етикет

Освен единични етикети и списъци от етикети, `.loc` позволява да селектирате диапазон от редове или колони, използвайки синтаксис за слайсиране, базиран на етикетите.

а) Синтаксис за слайсиране по етикет:

```
dataframe.loc[start_row_label:stop_row_label,
start_column_label:stop_column_label]
```

Важно е да се отбележи, че **при слайсиране по етикет с `.loc`, крайният етикет (`stop_row_label` и `stop_column_label`) е ВКЛЮЧЕН** в резултата, за разлика от стандартния Python слайсинг по индекс.

б) Примери (използвайки DataFrame `df` от предната тема):

```
import pandas as pd

# Създаваме примерен DataFrame с нечислов индекс
data = {'продукт': ['А', 'Б', 'А', 'В', 'Г', 'Б'],
        'цена': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0],
        'количество': [5, 2, 3, 1, 4, 6]}
df = pd.DataFrame(data, index=['ред_1', 'ред_2', 'ред_3', 'ред_4',
'ред_5', 'ред_6'])
print("Оригинален DataFrame с нечислов индекс:\n", df)

# Селектиране на редове от 'ред_2' до 'ред_4' (включително) и всички
колони
rows_2_to_4= df.loc['ред_2':'ред_4', :]
print("\nРедове от 'ред_2' до 'ред_4':\n", rows_2_to_4)

# Селектиране на всички редове и колоните от 'продукт' до 'количество'
(включително)
product_to_quantity_columns = df.loc[:, 'продукт':'количество']
print("\nКолони от 'продукт' до 'количество':\n",
product_to_quantity_columns)

# Селектиране на редове от 'ред_3' до края и колоните 'цена' и
'количество'
subset_with_end = df.loc['ред_3':, ['цена', 'количество']]
print("\nРедове от 'ред_3' до края, колони 'цена' и 'количество':\n",
subset_with_end)

# Селектиране на първите три реда и колоната 'цена'
first_three_rows_price = df.loc[:'ред_3', 'цена']
print("\nПървите три реда, колона 'цена':\n", first_three_rows_price)
```

в) Важни аспекти при слайсиране по етикет:

- **Включване на крайната точка:** Запомнете, че при слайсиране с `.loc` по етикет, както началният, така и крайният етикет са включени в резултата. Това е ключова разлика от стандартния Python слайсинг по индекс.
- **Ред на етикетите:** Слайсирането по етикет работи логично само ако етикетите на индекса (или колоните, когато слайсирате колони) са подредени. Ако редът на етикетите не е логически (например, не е азбучен или хронологичен), резултатът от слайсирането може да не е това, което очаквате.
- **Използване на `:` за всички редове или колони:** Както видяхме, `:` може да се използва като заместител на слайс, за да селектират всички редове или всички колони.

Слайсирането по етикет с `.loc` е много удобно, когато работите с DataFrame-и, които имат смислени, нечислови индекси (например, дати, имена на продукти, ID-та). То позволява интуитивно да селектират диапазони от данни въз основа на тези етикети.

4. Селектиране с булев масив

Булевият масив (или булев вектор) е последователност от `True` и `False` стойности. Когато се използва за селектиране на редове или колони в Pandas DataFrame с `.loc`, се избират само тези редове (или колони), за които съответната стойност в булевия масив е `True`.

- Селектиране на редове с булев масив:

За да селектирате редове с булев масив, подайте булев Series или списък с дължина, съвпадаща с броя на редовете на DataFrame-а, като първи аргумент на `.loc`

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C', 'B'],
        'price': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0],
        'quantity': [5, 2, 3, 1, 4, 6]}
df = pd.DataFrame(data, index=['row_1', 'row_2', 'row_3', 'row_4',
'row_5', 'row_6'])
print("Original DataFrame:\n", df)

# Създаваме булев масив (Series) със същата дължина като броя на
редовете
boolean_array_rows = pd.Series([True, False, True, False, True, False],
index=df.index)
```

```
# Селектираме редовете, за които стойността в булевия масив е True
selected_rows_boolean = df.loc[boolean_array_rows]
print("\nSelected rows with boolean array (Series):\n",
selected_rows_boolean)

# Можем да използваме и списък от булеви стойности
boolean_list_rows = [False, True, False, True, False, True]
selected_rows_boolean_list = df.loc[boolean_list_rows]
print("\nSelected rows with boolean list:\n",
selected_rows_boolean_list)

# Създаваме булев масив въз основа на условие върху колоната 'price'
high_price = df['price'] > 15.0
print("\nBoolean array for 'price' > 15.0:\n", high_price)

# Селектираме редовете, за които 'price' е по-висока от 15.0
rows_with_high_price = df.loc[high_price]
print("\nRows with 'price' > 15.0:\n", rows_with_high_price)

# Създаваме булев масив (Series) със същата дължина като броя на
колоните
boolean_array_columns = pd.Series([True, False, True], index=df.columns)

# Селектираме всички редове и колоните, за които стойността в булевия
масив е True
selected_columns_boolean = df.loc[:, boolean_array_columns]
print("\nSelected columns with boolean array (Series):\n",
selected_columns_boolean)

# Можем да използваме и списък от булеви стойности
boolean_list_columns = [False, True, False]
selected_columns_boolean_list = df.loc[:, boolean_list_columns]
print("\nSelected columns with boolean list:\n",
selected_columns_boolean_list)
```

```
# Създаваме булев масив въз основа на типа данни на колоните
numeric_columns = df.dtypes != 'object'
print("\nBoolean array for numeric columns:\n", numeric_columns)

# Селектираме всички редове и само числовите колони
numeric_data = df.loc[:, numeric_columns]
print("\nNumeric data:\n", numeric_data)

# Комбинирано селектиране с булеви масиви
selection_combined = df.loc[df['price'] > 15.0, df.dtypes != 'object']
print("\nCombined selection (price > 15.0 and numeric columns):\n",
selection_combined)
```

Най-често булевите масиви се създават в резултат на прилагане на логически условия върху колоните на DataFrame-а.

IV. Селектиране по позиция (.iloc)

Методът `.iloc` е подобен на `.loc`, но вместо да селектира по етикети, той селектира по **целочислени позиции** (индекси) на редовете и колоните. Индексирането с `.iloc` е базирано на нула, както при стандартните Python списъци и масиви.

1. Синтаксисът на `.iloc` е аналогичен на `.loc`:

```
dataframe.iloc[row_positions, column_positions]
```

където:

- `row_positions`: Може да бъде единично цяло число, списък от цели числа, слайс от цели числа или булев масив (със същата дължина като броя на редовете).
- `column_positions`: Подобно на `row_positions`, може да бъде единично цяло число, списък от цели числа или слайс от цели числа. За селектиране на всички колони се използва `:`.

2. Селектиране на единични/множество редове/колони по позиция (целочислено):

а) Селектиране на единичен ред:

Подава се целочислената позиция на реда (започвайки от 0). Резултатът е Pandas Series.

```
import pandas as pd
```

```
# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B'],
        'price': [10.5, 20.0, 10.5, 15.0],
        'quantity': [5, 2, 3, 1]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Селектиране на реда на позиция 1 (вторият ред)
row_at_position_1 = df.iloc[1]
print("\nRow at position 1 (second row):\n", row_at_position_1)
print("\nType of result:", type(row_at_position_1))
```

б) Селектиране на множество редове:

Подава се списък от целочислени позиции на редовете. Резултатът е DataFrame.

```
# Селектиране на редовете на позиции 0 и 2 (първият и третият ред)
rows_at_positions_0_and_2 = df.iloc[[0, 2]]
print("\nRows at positions 0 and 2:\n", rows_at_positions_0_and_2)
print("\nType of result:", type(rows_at_positions_0_and_2))
```

в) Селектиране на единична колона:

Подава се : за всички редове и целочислената позиция на колоната. Резултатът е Series.

```
# Селектиране на колоната на позиция 0 (първата колона - 'product')
column_at_position_0 = df.iloc[:, 0]
print("\nColumn at position 0 (first column - 'product'):\n",
column_at_position_0)
print("\nType of result:", type(column_at_position_0))
```

г) Селектиране на множество колони:

Подава се : за всички редове и списък от целочислени позиции на колоните. Резултатът е DataFrame.

```
# Селектиране на колоните на позиции 0 и 2 (първата и третата колона - 'product' и 'quantity')
columns_at_positions_0_and_2 = df.iloc[:, [0, 2]]
print("\nColumns at positions 0 and 2:\n", columns_at_positions_0_and_2)
print("\nType of result:", type(columns_at_positions_0_and_2))
```

д) Селектиране на единична клетка:

Подават се целочислената позиция на реда и целочислената позиция на колоната. Резултатът е скаларна стойност.

```
# Селектиране на стойността в реда на позиция 1 и колоната на позиция 2
cell_value = df.iloc[1, 2]
print("\nValue at row position 1, column position 2:", cell_value)
print("\nType of result:", type(cell_value))
```

е) Селектиране на множество клетки:

Комбинират се списъци от целочислени позиции за редове и колони. Резултатът е DataFrame.

```
# Селектиране на стойностите за редовете на позиции 0 и 2 в колоните на позиции 1 и 2
subset_by_position = df.iloc[[0, 2], [1, 2]]
print("\nSubset of data by position:\n", subset_by_position)
print("\nType of result:", type(subset_by_position))
```

Методът `.iloc` е полезен, когато не знаете или не искате да използвате етикетите на индекса и колоните и предпочитате да работите с познатото целочислено индексиране. Той е особено удобен при итерация през редове или колони или при прилагане на операции, които зависят от позицията на данните.

3. Селектиране чрез слайсинг по позиция

Подобно на слайсинга, който използвахме за селектиране на редове с `[]`, можем да използваме слайсинг и с `.iloc`, но този път оперираме с целочислените позиции на редовете и колоните.

а) Синтаксис за слайсиране по позиция:

```
dataframe.iloc[start_row:stop_row:step_row,
start_column:stop_column:step_column]
```

където `start`, `stop` и `step` са цели числа, указващи началото, края (изключен) и стъпката на слайса съответно.

б) Примери (използвайки DataFrame `df` от предната тема):

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C', 'B'],
        'price': [10.5, 20.0, 10.5, 15.0, 25.0, 20.0],
        'quantity': [5, 2, 3, 1, 4, 6]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Селектиране на първите 3 реда (позиции 0, 1, 2) и всички колони
first_three_rows = df.iloc[0:3, :]
print("\nFirst three rows:\n", first_three_rows)

# Селектиране на всички редове и първите 2 колони (позиции 0, 1)
first_two_columns = df.iloc[:, 0:2]
print("\nFirst two columns:\n", first_two_columns)

# Селектиране на редове от позиция 2 до края и колони от позиция 1 до
края
subset_from_position = df.iloc[2:, 1:]
print("\nSubset from row position 2 and column position 1:\n",
subset_from_position)

# Селектиране на всеки втори ред и всички колони
```

```
every_second_row = df.iloc[::2, :]
print("\nEvery second row:\n", every_second_row)

# Селектиране на последните 2 реда и последните 2 колони (използване на отрицателни индекси)
last_two_rows_cols = df.iloc[-2:, -2:]
print("\nLast two rows and last two columns:\n", last_two_rows_cols)
```

в) Важни аспекти при слайсиране по позиция с `.iloc`:

- **Крайната точка е изключена:** Както при стандартния Python слайсинг, крайният индекс в слайса **не е включен** в резултата. Например, `0:3` ще селектира редовете на позиции 0, 1 и 2.
- **Целочислени позиции:** `.iloc` работи изключително с целочислени позиции, независимо от типа на индекса или етикетите на колоните.
- **Използване на `:` за всички редове или колони:** Подобно на `.loc`, `:` може да се използва за селектиране на всички редове или всички колони.

Слайсирането по позиция с `.iloc` е полезно, когато искате да извлечете подмножества от данни въз основа на тяхното абсолютно местоположение в DataFrame-а, без да се интересувате от етикетите.

V. Комбинирано селектиране (Опасности и алтернативи)

Понякога може да възникне желание да се комбинират различни методи за селектиране на данни в Pandas, например верижно индексиране (chained indexing) като `df['колона'][ред]` или `df.loc[ред]['колона']`. Въпреки че тези методи могат да изглеждат интуитивни, те често крият опасности и могат да доведат до неочаквано поведение, особено при опит за модифициране на данни.

1. Опасности при верижното индексиране:

Верижното индексиране се случва, когато използвате повече от един оператор за индексиране последователно. Например:

1) `df['колона']['ред']` - Първо селектира колоната, след което селектира ред от резултата (който е Series).
2) `df.loc['ред']['колона']` - Първо селектира реда, след което селектира колона от резултата (който е Series).

Основната опасност при верижното индексиране е свързана с това дали резултатът от първата операция е **view** (изглед) или **copy** (копие) на оригиналните данни. Pandas не винаги гарантира дали ще получите view или copy при верижно индексиране.

- **Проблеми при модификация:** Ако първата операция върне copy, модификацията на резултата няма да се отрази на оригиналния DataFrame, което може да доведе до объркване и грешки в анализа. Ако върне view, модификацията ще се отрази, но поведението може да бъде непредсказуемо в определени ситуации.

Нека илюстрираме с пример:

```
import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Опит за модификация с верижно индексиране
df['A'][0] = 10  # Възможно е да работи, но не се препоръчва
print("\nDataFrame след опит за модификация (вероятно е променен):\n",
df)

# Опит за модификация на копие (може да не промени оригиналния
DataFrame)
subset = df['B']
subset[0] = 40
print("\nDataFrame след модификация на subset (може да не е
променен):\n", df)
print("\nSubset след модификация:\n", subset)
```

Pandas често ще издаде предупреждение (`SettingWithCopyWarning`), когато открие потенциален опит за модификация върху copy, но не винаги може да го засече.

2. Алтернативи за безопасно и ясно селектиране:

Препоръчителният начин за селектиране и модифициране на данни е чрез използване на `.loc` и `.iloc`, които осигуряват по-ясен и предвидим начин за достъп до данни.

a) Използване на `.loc` за комбинирано селектиране по етикети:

```
import pandas as pd
```

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row_1', 'row_2',
'row_3']
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Селектиране на елемента в ред 'row_1' и колона 'A'
value_loc = df.loc['row_1', 'A']
print("\nValue using .loc:", value_loc)

# Модификация с .loc (препоръчителен начин)
df.loc['row_1', 'A'] = 100
print("\nDataFrame след модификация с .loc:\n", df)
```

б) Използване на `.iloc` за комбинирано селектиране по позиции:

```
import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Селектиране на елемента в ред на позиция 0 и колона на позиция 1
value_iloc = df.iloc[0, 1]
print("\nValue using .iloc:", value_iloc)

# Модификация с .iloc (препоръчителен начин)
df.iloc[0, 1] = 400
print("\nDataFrame след модификация с .iloc:\n", df)
```

3. Еднократно индексиране:

Най-безопасният и препоръчителен начин е да се извършва селекцията в една стъпка с помощта на `.loc` или `.iloc`, като се указват едновременно етикетите (за `.loc`) или позициите (за `.iloc`) на редовете и колоните.

```
import pandas as pd
```

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row_1', 'row_2',
'row_3']
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Безопасно селектиране и модификация с .loc
df.loc['row_2', 'B'] = 500
print("\nDataFrame след безопасна модификация с .loc:\n", df)

# Безопасно селектиране и модификация с .iloc
df.iloc[2, 0] = 300
print("\nDataFrame след безопасна модификация с .iloc:\n", df)
```

Избягвайте верижното индексиране, когато искате да модифицирате DataFrame. Винаги използвайте `.loc` или `.iloc` за ясна и безопасна селекция и модификация на данни в една стъпка. Това ще ви помогне да избегнете неочаквани резултати и `SettingWithCopyWarning`.

VI. Булево индексиране (логически условия)

1. Оператори за сравнение (==, !=, >, <, >=, <=)

Булевото индексиране е мощен метод в Pandas за филтриране на данни въз основа на логически условия. Операторите за сравнение играят ключова роля в създаването на тези условия. Когато приложите оператор за сравнение между Series (например, колона от DataFrame) и скаларна стойност или друг Series (със същия индекс), резултатът е булев Series, където всеки елемент е `True` или `False` в зависимост от това дали условието е изпълнено за съответния елемент.

Нека разгледаме всеки от операторите за сравнение с примери:

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C'],
        'price': [10.5, 20.0, 10.5, 15.0, 25.0],
        'quantity': [5, 2, 3, 1, 4]}
df = pd.DataFrame(data)
```

```
print("Original DataFrame:\n", df)

# 1. Равно (==)
# Проверяваме кои продукти имат цена 10.5
price_equal_10_5 = df['price'] == 10.5
print("\nProducts with price == 10.5:\n", price_equal_10_5)

# 2. Различно (!=)
# Проверяваме кои продукти имат количество, различно от 2
quantity_not_equal_2 = df['quantity'] != 2
print("\nProducts with quantity != 2:\n", quantity_not_equal_2)

# 3. По-голямо от (>)
# Проверяваме кои продукти имат цена по-голяма от 15.0
price_greater_than_15 = df['price'] > 15.0
print("\nProducts with price > 15.0:\n", price_greater_than_15)

# 4. По-малко от (<)
# Проверяваме кои продукти имат количество по-малко от 4
quantity_less_than_4 = df['quantity'] < 4
print("\nProducts with quantity < 4:\n", quantity_less_than_4)

# 5. По-голямо или равно на (>=)
# Проверяваме кои продукти имат цена по-голяма или равна на 15.0
price_greater_equal_15 = df['price'] >= 15.0
print("\nProducts with price >= 15.0:\n", price_greater_equal_15)

# 6. По-малко или равно на (<=)
# Проверяваме кои продукти имат количество по-малко или равно на 3
quantity_less_equal_3 = df['quantity'] <= 3
print("\nProducts with quantity <= 3:\n", quantity_less_equal_3)

# Използване на булевия Series за селектиране на редове с .loc
expensive_products = df.loc[df['price'] > 15.0]
print("\nExpensive products (price > 15.0):\n", expensive_products)
```

```
low_quantity_products = df.loc[df['quantity'] < 4]
print("\nProducts with low quantity (quantity < 4):\n",
low_quantity_products)
```

Както виждате от примерите, прилагането на оператори за сравнение към колона на DataFrame (Series) връща булев Series със същия индекс. След това този булев Series може да бъде използван като вход за `.loc`, за да селектират само редовете, за които стойността е `True`.

В следващите подтеми ще разгледаме как да комбинираме множество логически условия и други методи за булево индексиране.

2. Логически оператори (&, |, ~, ^)

Когато искаме да филтрираме данни въз основа на **множество условия**, трябва да комбинираме булеви Series, използвайки логически оператори. В Pandas, стандартните Python оператори `and`, `or`, и `not` **не работят** поелементно върху булеви Series, както очакваме. Вместо тях се използват следните поелементни логически оператори:

- `&` **(амперсанд): Логическо И (AND)** - Връща `True` само ако и двете съответни стойности в сравняваните булеви Series са `True`.
- `|` **(вертикална черта): Логическо ИЛИ (OR)** - Връща `True`, ако поне една от съответните стойности в сравняваните булеви Series е `True`.
- `~` **(тилда): Логическо НЕ (NOT)** - Инвертира булевите стойности; `True` става `False`, а `False` става `True`.
- `^` **(карет): Логическо ИЗКЛЮЧИТЕЛНО ИЛИ (XOR)** - Връща `True` само ако точно една от съответните стойности в сравняваните булеви Series е `True`.

Важно правило: Когато комбинирате множество булеви условия, е необходимо да заградите всяко отделно условие в **кръгли скоби** `()`. Това гарантира правилния ред на изпълнение на логическите операции.

Нека разгледаме примери, използвайки DataFrame `df` от предната тема:

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C', 'A'],
        'price': [10.5, 20.0, 10.5, 15.0, 25.0, 10.5],
        'quantity': [5, 2, 3, 1, 4, 2]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Условие 1: Продуктът е 'A'
is_product_a = df['product'] == 'A'
print("\nCondition: product == 'A'\n", is_product_a)
```

```
# Условие 2: Цената е по-голяма от 10.0
price_gt_10 = df['price'] > 10.0
print("\nCondition: price > 10.0\n", price_gt_10)

# 1. Логическо И (&) - Продуктът е 'A' И цената е по-голяма от 10.0
condition_and = (df['product'] == 'A') & (df['price'] > 10.0)
print("\nCondition: product == 'A' AND price > 10.0\n", condition_and)
filtered_and = df.loc[condition_and]
print("\nFiltered DataFrame (AND):\n", filtered_and)

# 2. Логическо ИЛИ (|) - Продуктът е 'B' ИЛИ количеството е по-голямо от
3
condition_or = (df['product'] == 'B') | (df['quantity'] > 3)
print("\nCondition: product == 'B' OR quantity > 3\n", condition_or)
filtered_or = df.loc[condition_or]
print("\nFiltered DataFrame (OR):\n", filtered_or)

# 3. Логическо НЕ (~) - Продуктът НЕ е 'B'
condition_not = ~(df['product'] == 'B')
print("\nCondition: product is NOT 'B'\n", condition_not)
filtered_not = df.loc[condition_not]
print("\nFiltered DataFrame (NOT):\n", filtered_not)

# 4. Логическо ИЗКЛЮЧИТЕЛНО ИЛИ (^) - Цената е 10.5 ИЛИ количеството е
2, НО НЕ И ДВЕТЕ ЕДНОВРЕМЕННО
condition_xor = (df['price'] == 10.5) ^ (df['quantity'] == 2)
print("\nCondition: price == 10.5 XOR quantity == 2\n", condition_xor)
filtered_xor = df.loc[condition_xor]
print("\nFiltered DataFrame (XOR):\n", filtered_xor)
```

Разбирането и правилното използване на тези логически оператори е ключово за извършване на комплексни филтрации на данни във вашите Pandas DataFrame-и. Не забравяйте за скобите!

3. Методът .isin()

Методът `.isin()` е много удобен, когато искате да филтрирате редове, където стойността в дадена колона съвпада с някоя от стойностите в списък (или друг итерируем обект като Series или set). Той връща булев Series, който е `True` за всеки елемент в колоната, който се съдържа в предоставения списък, и `False` в противен случай.

a) Синтаксисът на `.isin()` е:

```
series.isin(values)
```

където:

- `series`: е Pandas Series (например, колона от DataFrame), върху която искате да приложите филтъра.
- `values`: е списък, tuple, set или друг Series, съдържащ стойностите, които искате да търсите.

Нека разгледаме примери, използвайки DataFrame `df` от предната тема:

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'C', 'A', 'D', 'B'],
        'color': ['red', 'blue', 'green', 'red', 'yellow', 'blue']}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Искаме да филтрираме редовете, където продуктът е 'A' или 'B'
products_to_include = ['A', 'B']
filter_by_product = df['product'].isin(products_to_include)
print("\nBoolean Series for products 'A' or 'B':\n", filter_by_product)

# Селектираме редовете, където продуктът е 'A' или 'B'
selected_products = df.loc[filter_by_product]
print("\nDataFrame with products 'A' or 'B':\n", selected_products)

# Можем да използваме и set за стойностите
colors_to_include = {'red', 'green'}
filter_by_color = df['color'].isin(colors_to_include)
```

```
print("\nBoolean Series for colors 'red' or 'green':\n",
filter_by_color)

# Селектираме редовете, където цветът е 'red' или 'green'
selected_colors = df.loc[filter_by_color]
print("\nDataFrame with colors 'red' or 'green':\n", selected_colors)

# Можем да използваме и друг Series като стойности за търсене
search_values = pd.Series(['C', 'blue'])
filter_by_search_values = df.isin(search_values)
print("\nBoolean DataFrame checking if any value matches 'C' or
'blue':\n", filter_by_search_values)

# За да филтрираме редове, където поне една колона съдържа 'C' или
'blue'
rows_with_c_or_blue = df[df.isin(search_values).any(axis=1)]
print("\nRows where at least one column contains 'C' or 'blue':\n",
rows_with_c_or_blue)
```

б) Предимства на метода `.isin()`:

- **Четливост:** Прави филтрирането по множество стойности много по-ясно и кратко в сравнение с използването на множество оператори `==` и `|`.
- **Ефективност:** Особено при голям брой стойности за търсене, `.isin()` обикновено е по-ефективен.
- **Гъвкавост:** Приема различни итерируеми обекти като стойности за търсене.

Методът `.isin()` е незаменим инструмент, когато трябва да филтрирате данни въз основа на принадлежност към определен набор от стойности.

4. Методът `.isnull()` и `.notnull()`

Липсващите данни са често срещано явление при работа с реални набори от данни. Pandas използва стойността `NaN` (от NumPy) за представяне на липсващи числови данни и `None` (от Python) или `NaN` за други типове данни. Методите `.isnull()` и `.notnull()` са ключови за идентифицирането на тези липсващи стойности.

а) Методът `.isnull()`:

Методът `.isnull()` се прилага към Series или DataFrame и връща булев Series или DataFrame със същата структура. Върнатата стойност е `True` за всяка клетка, съдържаща липсваща стойност (`NaN` или `None`), и `False` за всички останали стойности.

- Синтаксис:

```
series.isnull()
dataframe.isnull()
```

- Пример:

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с липсващи стойности
data = {'A': [1, 2, np.nan, 4],
        'B': [5, np.nan, 7, 8],
        'C': [np.nan, np.nan, 9, 10],
        'D': [11, 12, 13, None]}
df = pd.DataFrame(data)
print("Original DataFrame with missing values:\n", df)

# Прилагаме .isnull() към DataFrame
is_null_df = df.isnull()
print("\nBoolean DataFrame indicating missing values:\n", is_null_df)

# Прилагаме .isnull() към колона (Series)
is_null_column_a = df['A'].isnull()
print("\nBoolean Series for missing values in column 'A':\n",
is_null_column_a)

# Използваме булевия Series за филтриране на редовете, където 'A' е NaN
rows_where_a_is_null = df[df['A'].isnull()]
print("\nRows where column 'A' contains NaN:\n", rows_where_a_is_null)
```

б) Методът `.notnull()`:

Методът `.notnull()` е противоположен на `.isnull()`. Той също се прилага към Series или DataFrame и връща булев Series или DataFrame, но със стойност `True` за всяка клетка, която **не** съдържа липсваща стойност, и `False` за липсващите стойности.

- Синтаксис:

```
series.notnull()
dataframe.notnull()
```

- Пример (използвайки същия DataFrame `df`):

```
# Прилагаме .notnull() към DataFrame
not_null_df = df.notnull()
print("\nBoolean DataFrame indicating non-missing values:\n",
not_null_df)

# Прилагаме .notnull() към колона (Series)
not_null_column_b = df['B'].notnull()
print("\nBoolean Series for non-missing values in column 'B':\n",
not_null_column_b)

# Използваме булевия Series за филтриране на редовете, където 'B' не е
NaN
rows_where_b_is_not_null = df[df['B'].notnull()]
print("\nRows where column 'B' does not contain NaN:\n",
rows_where_b_is_not_null)

# Можем да комбинираме условия с логически оператори
rows_where_a_is_null_and_b_is_not_null = df[(df['A'].isnull()) &
(df['B'].notnull())]
print("\nRows where 'A' is NaN AND 'B' is not NaN:\n",
rows_where_a_is_null_and_b_is_not_null)
```

в) Използване за филтриране:

Основното приложение на `.isnull()` и `.notnull()` в контекста на булевото индексиране е за филтриране на редове въз основа на наличието или отсъствието на липсващи стойности в определени колони.

Разбирането и използването на тези методи е от съществено значение за почистване и анализ на данни, които често съдържат липсващи стойности.

Готови ли сте да продължим към следващата тема: **Query метод (`.query()`)**

5. Методът .between()

Методът .between() се използва за проверка дали стойностите в Series се намират в даден затворен интервал (включително крайните точки). Той връща булев Series, който е True за всеки елемент, който е между (или равен на) зададените начална и крайна стойност, и False в противен случай.

Синтаксисът на .between() е:

```
series.between(left, right, inclusive='both')
```

където:

- `left`: Долната граница на интервала (включена по подразбиране).
- `right`: Горната граница на интервала (включена по подразбиране).
- `inclusive`: Определя кои граници да бъдат включени. Възможни стойности са:
  - `'both'` (по подразбиране): Включва и двете граници.
  - `'left'`: Включва само лявата граница.
  - `'right'`: Включва само дясната граница.
  - `'neither'`: Не включва нито една от границите.

Нека разгледаме примери, използвайки DataFrame `df` от предните теми:

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C', 'A'],
        'price': [10.5, 20.0, 15.0, 15.0, 22.5, 18.0],
        'quantity': [5, 2, 3, 1, 4, 2]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Филтриране на продукти с цена между 15 и 20 (включително)
price_between_15_20 = df['price'].between(15, 20)
```

```
print("\nBoolean Series for price between 15 and 20 (inclusive):\n",
price_between_15_20)
filtered_price_between = df[price_between_15_20]
print("\nProducts with price between 15 and 20:\n",
filtered_price_between)

# Филтриране на продукти с количество между 2 и 4 (лявата граница
включена)
quantity_between_2_4_left = df['quantity'].between(2, 4,
inclusive='left')
print("\nBoolean Series for quantity between 2 and 4 (left
inclusive):\n", quantity_between_2_4_left)
filtered_quantity_between_left = df[quantity_between_2_4_left]
print("\nProducts with quantity between 2 and 4 (left inclusive):\n",
filtered_quantity_between_left)

# Филтриране на продукти с цена между 15 и 20 (нито една граница не е
включена)
price_between_15_20_neither = df['price'].between(15, 20,
inclusive='neither')
print("\nBoolean Series for price between 15 and 20 (neither
inclusive):\n", price_between_15_20_neither)
filtered_price_between_neither = df[price_between_15_20_neither]
print("\nProducts with price between 15 and 20 (neither inclusive):\n",
filtered_price_between_neither)
```

Методът .between() е много полезен за ясно и ефективно филтриране на числови данни в определен диапазон.

VII. Методът .query() за селекция чрез низ

Методът `.query()` предоставя по-експресивен начин за филтриране на DataFrame, като използва низов израз, който наподобява SQL `WHERE` клауза или условен израз в Python. Това може да направи кода за филтриране по-четлив, особено при сложни условия.

а) Синтаксисът на `.query()` е:

```
dataframe.query(expr, inplace=False, **kwargs)
```

където:

- `expr`: е низов израз, който определя условието за филтриране. Той може да включва имена на колони (които се третират като променливи), оператори за сравнение, логически оператори (`and`, `or`, `not`), както и променливи от Python средата (с префикс `@`).
- `inplace`: ако е `True`, модифицира DataFrame-а на място (по подразбиране е `False`, връща нов DataFrame).
- `**kwargs`: допълнителни ключови думи аргументи, които могат да се използват в израза `expr`.

Нека разгледаме примери, използвайки DataFrame `df` от предните теми (ще го създадем наново за яснота):

б) Пример:

```
import pandas as pd

# Създаваме примерен DataFrame
data = {'product': ['A', 'B', 'A', 'B', 'C', 'A'],
        'price': [10.5, 20.0, 10.5, 15.0, 25.0, 10.5],
        'quantity': [5, 2, 3, 1, 4, 2]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Филтриране по единично условие
filtered_price_gt_15 = df.query('price > 15')
print("\nProducts with price > 15 using .query():\n",
filtered_price_gt_15)

# Филтриране по множество условия с 'and'
filtered_a_and_gt_2 = df.query('product == "A" and quantity > 2')
print("\nProducts 'A' with quantity > 2 using .query():\n",
filtered_a_and_gt_2)

# Филтриране по множество условия с 'or'
filtered_b_or_price_lt_12 = df.query('product == "B" or price < 12')
print("\nProducts 'B' or price < 12 using .query():\n",
filtered_b_or_price_lt_12)

# Използване на променливи от Python средата с префикс '@'
min_price = 13
```

```
filtered_price_gt_min = df.query('price > @min_price')
print(f"\nProducts with price > {min_price} using .query():\n",
filtered_price_gt_min)

valid_products = ['A', 'C']
filtered_isin = df.query('product in @valid_products')
print(f"\nProducts in {valid_products} using .query():\n",
filtered_isin)

# Използване на 'not in'
invalid_products = ['B']
filtered_notin = df.query('product not in @invalid_products')
print(f"\nProducts not in {invalid_products} using .query():\n",
filtered_notin)
```

в) Предимства на метода `.query()`:

- **Четливост:** За сложни филтрирания, низовият израз може да бъде по-лесен за разбиране от еквивалентните булеви условия с оператори `&`, `|`, `~`.
- **По-близък до SQL:** Ако сте запознати със SQL, синтаксисът на `.query()` може да ви се стори по-интуитивен.
- **Възможност за използване на външни променливи:** Лесно интегриране на променливи от вашата Python среда във филтриращите условия.

г) Ограничения и внимание:

- **Стрингове в условията:** Стринговите стойности в низовия израз трябва да бъдат оградени в кавички (ако съдържат апострофи, използвайте двойни кавички и обратно).
- **Имена на колони с интервали или специални символи:** Ако имената на колоните съдържат интервали или специални символи, може да се наложи да ги оградите в обратни апострофи (backticks - `).

> **Методът `.query()` е полезен инструмент за селекция, особено когато условията станат по-сложни и искате да имате по-четим код.**

VIII. Бърз скаларен достъп (.at, .iat)

Методите `.at` и `.iat` в Pandas се използват за **бърз достъп и задаване на скаларна стойност** (единична стойност) в DataFrame или Series. Те са оптимизирани за тази конкретна задача и често са по-бързи от `.loc` и `.iloc`, когато трябва да получите или промените само една клетка.

1. `.at` - Достъп по етикет:

Методът `.at` се използва за достъп до скаларна стойност по **етикети** на ред и колона.

а) Синтаксис за DataFrame:

```
dataframe.at[row_label, column_label]
```

б) Синтаксис за Series:

```
series.at[index_label]
```

в) Пример с DataFrame:

```
import pandas as pd

# Създаваме примерен DataFrame с нечислов индекс
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row_1', 'row_2',
'row_3']
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Достъп до стойността в ред 'row_2' и колона 'B'
value_at = df.at['row_2', 'B']
print("\nValue at ['row_2', 'B']:", value_at)

# Задаване на нова стойност
df.at['row_3', 'A'] = 10
print("\nDataFrame after setting value at ['row_3', 'A']:\n", df)
```

г) Пример със Series:

```
# Създаваме примерен Series с нечислови индекси
```

```
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print("\nOriginal Series:\n", s)

# Достъп до стойността с индекс 'b'
value_at_series = s.at['b']
print("\nValue at index 'b' in Series:", value_at_series)

# Задаване на нова стойност
s.at['c'] = 35
print("\nSeries after setting value at index 'c':\n", s)
```

2. `iat` - Достъп по целочислена позиция:

Методът `.iat` се използва за достъп до скаларна стойност по **целочислени позиции** (индекси) на ред и колона. Той е базиран на нула.

а) Синтаксис за DataFrame:

```
dataframe.iat[row_position, column_position]
```

б) Синтаксис за Series:

```
series.iat[index_position]
```

в) Пример с DataFrame:

```
# Използваме същия DataFrame df
print("\nDataFrame:\n", df)

# Достъп до стойността в ред на позиция 1 (втори ред) и колона на
позиция 0 (първа колона)
value_iat = df.iat[1, 0]
print("\nValue at [1, 0]:", value_iat)

# Задаване на нова стойност
df.iat[0, 1] = 40
print("\nDataFrame after setting value at [0, 1]:\n", df)
```

г) Пример със Series:

```
# Използваме същия Series s
print("\nSeries:\n", s)

# Достъп до стойността на позиция 0 (първи елемент)
value_iat_series = s.iat[0]
print("\nValue at position 0 in Series:", value_iat_series)

# Задаване на нова стойност
s.iat[2] = 45
print("\nSeries after setting value at position 2:\n", s)
```

3. Кога да използваме `.at` и `.iat`:

- **Бърз достъп до единична стойност:** Когато знаете точния етикет или позиция на реда и колоната, `.at` и `.iat` са по-ефективни от `.loc` и `.iloc`.
- **Задаване на единична стойност:** Те също са бързи при промяна на стойността на конкретна клетка.
- **Вътре в цикли (с внимание):** Въпреки че са бързи за единичен достъп, при голям брой последователни достъпи в цикъл, други методи като `.iloc` с предварително определени слайсове могат да бъдат по-ефективни.

4. Важни разлики:

- **`.at` работи с етикети**, докато **`.iat` работи с целочислени позиции**.
- И двата метода са предназначени за **достъп до една-единствена скаларна стойност**. Опитът за достъп до множество редове или колони ще доведе до грешка.

В общия случай, ако работите с етикети, използвайте `.at`. Ако работите с целочислени позиции, използвайте `.iat`. Когато селектирате повече от една стойност, използвайте `.loc` или `.iloc`.

IX. Индексиране с извикваеми обекти (callables)

Pandas предоставя гъвкавост при селектирането на данни, като позволява използването на функции (или други извикваеми обекти) като аргументи на `.loc`, `.iloc` и `.at`, `.iat`. Извикваемият обект се

извиква върху индекса (или колоните) и трябва да върне валиден резултат за селекция (например, единичен етикет, списък от етикети, булев масив или слайс).

Това може да бъде полезно за динамично генериране на критерии за селекция въз основа на стойностите в индекса или колоните.

1. Индексиране с извикваеми обекти с `.loc`:

Когато използвате извикваем обект с `.loc`, той се прилага към индекса (за селекция на редове) или към колоните (за селекция на колони). Функцията трябва да приеме индекса (или колоните) като аргумент и да върне резултат, който може да бъде използван за селекция.

Пример за селекция на редове, чийто индекс съдържа определен стринг:

```
import pandas as pd

# Създаваме примерен DataFrame с нечислов индекс
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row_a', 'row_b_long',
'row_c']
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Функция за селектиране на индекси, съдържащи '_long'
def select_long_indices(idx):
    return [i for i in idx if '_long' in i]

# Използваме функцията с .loc за селекция на редове
selected_rows = df.loc[select_long_indices]
print("\nRows with index containing '_long':\n", selected_rows)

# Можем да използваме и lambda функция
selected_rows_lambda = df.loc[lambda idx: [i for i in idx if 'a' in i]]
print("\nRows with index containing 'a' (using lambda):\n",
selected_rows_lambda)

# Селекция на колони, чието име започва с 'A'
selected_cols_lambda = df.loc[:, lambda cols: [c for c in cols if
c.startswith('A')]]
print("\nColumns starting with 'A':\n", selected_cols_lambda)
```

2. Индексиране с извикваеми обекти с `.iloc`:

Подобно на `.loc`, можете да използвате извикваеми обекти и с `.iloc`, но в този случай функцията ще се прилага към целочислените позиции на индекса или колоните.

Пример за селекция на редове на четни позиции:

```
# Селекция на редове на четни позиции (0, 2, ...)
selected_rows_iloc_lambda = df.iloc[lambda idx: [i for i in
range(len(idx)) if i % 2 == 0]]
print("\nRows at even positions:\n", selected_rows_iloc_lambda)

# Селекция на колони на нечетни позиции (1, 3, ...)
selected_cols_iloc_lambda = df.iloc[:, lambda cols: [i for i in
range(len(cols)) if i % 2 != 0]]
print("\nColumns at odd positions:\n", selected_cols_iloc_lambda)
```

3. Индексиране с извикваеми обекти с `.at` и `.iat`:

Тъй като `.at` и `.iat` очакват единичен етикет или позиция, използването на извикваеми обекти с тях е по-рядко срещано и трябва да върне единична валидна стойност.

Пример (по-скоро за илюстрация, не е типична употреба):

```
# Извикваем обект, връщащ етикет
def get_row_label(idx):
    return idx[1]

# Използване с .at
value_at_callable = df.at[get_row_label, 'B']
print("\nValue using callable with .at:", value_at_callable)

# Извикваем обект, връщащ позиция
def get_row_position(idx_len):
    return idx_len - 1

# Използване с .iat
value_iat_callable = df.iat[lambda length: length - 1, 0]
print("\nValue using callable with .iat:", value_iat_callable(len(df)))
```

4. Предимства на индексирането с извикваеми обекти:

- **Динамичност:** Позволява генериране на критерии за селекция по време на изпълнение.
- **Гъвкавост:** Може да се използват сложни логики, капсулирани във функции.
- **Четливост (в някои случаи):** За определени сложни условия, използването на именувана функция може да направи кода по-разбираем.

Въпреки, че не е най-често използваният метод за селекция, индексирането с извикваеми обекти може да бъде много полезно в специфични сценарии, където логиката за селекция е динамична или сложна.

X. Задаване на стойности чрез селекция

Една от основните операции при работа с данни е модифицирането на стойности в DataFrame или Series въз основа на определени условия или местоположения. Pandas позволява да задавате нови стойности на подмножества от данни, селектирани с помощта на вече разгледаните методи (`[]`, `.loc`, `.iloc`).

1. Задаване на стойности с `[]` (за колони):

Можете да зададете една и съща стойност на цяла колона или на подмножество от редове в колона.

```
import pandas as pd

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

# Задаване на една и съща стойност на цялата колона 'A'
df['A'] = 100
print("\nDataFrame after setting column 'A' to 100:\n", df)

# Задаване на различни стойности на колона 'B' (трябва да съвпадат по дължина)
df['B'] = [40, 50, 60]
print("\nDataFrame after setting column 'B' with a list:\n", df)
```

```
# Задаване на стойност на подмножество от редове в колона (чрез булев масив)
df['A'][df['B'] > 50] = 200
print("\nDataFrame after setting 'A' where 'B' > 50:\n", df)
```

Внимание: Както споменахме в темата за комбинирано селектиране, верижното присвояване (например `df['A'][условие] = стойност`) може да доведе до неочаквано поведение и е препоръчително да се избягва в полза на `.loc`.

2. Задаване на стойности с `.loc` (по етикети):

`.loc` е предпочитаният начин за задаване на стойности въз основа на етикети на редове и колони, тъй като е по-ясен и избягва проблемите с `SettingWithCopyWarning`.

```
# Създаваме нов DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row_1', 'row_2', 'row_3']
df_loc = pd.DataFrame(data)
print("\nOriginal DataFrame (for .loc):\n", df_loc)

# Задаване на стойност на конкретна клетка
df_loc.loc['row_1', 'A'] = 1000
print("\nDataFrame after setting value at ['row_1', 'A'] with .loc:\n", df_loc)

# Задаване на една и съща стойност на цял ред
df_loc.loc['row_2'] = 500
print("\nDataFrame after setting row 'row_2' with .loc:\n", df_loc)

# Задаване на стойности на подмножество от редове и колони
df_loc.loc[['row_1', 'row_3'], 'B'] = [400, 600]
print("\nDataFrame after setting values for specific rows and column 'B' with .loc:\n", df_loc)

# Задаване на стойности въз основа на булев масив (по етикети на индекса)
df_loc.loc[df_loc['A'] > 100, 'B'] = 700
```

```
print("\nDataFrame after setting 'B' where 'A' > 100 with .loc:\n",
df_loc)
```

3. Задаване на стойности с `.iloc` (по позиции):

`.iloc` се използва за задаване на стойности въз основа на целочислени позиции на редове и колони.

```
# Създаваме нов DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df_iloc = pd.DataFrame(data)
print("\nOriginal DataFrame (for .iloc):\n", df_iloc)

# Задаване на стойност на конкретна клетка
df_iloc.iloc[0, 1] = 4000
print("\nDataFrame after setting value at [0, 1] with .iloc:\n",
df_iloc)

# Задаване на една и съща стойност на цял ред
df_iloc.iloc[1] = 5000
print("\nDataFrame after setting row at position 1 with .iloc:\n",
df_iloc)

# Задаване на стойности на подмножество от редове и колони
df_iloc.iloc[[0, 2], 0] = [1000, 3000]
print("\nDataFrame after setting values for specific rows and column at
position 0 with .iloc:\n", df_iloc)

# Задаване на стойности въз основа на булев масив (по подразбиращ се
целочислен индекс)
df_iloc.iloc[df_iloc['B'] > 4000, 1] = 7000
print("\nDataFrame after setting column at position 1 where 'B' > 4000
with .iloc:\n", df_iloc)
```

4. Задаване на стойности с `.at` и `.iat` (бърз скаларен достъп):

Тези методи са най-бързи за задаване на стойност на **единична клетка**, когато знаете нейния етикет (`.at`) или позиция (`.iat`).

```
# Използваме df_loc
```

```
df_loc.at['row_3', 'B'] = 800
print("\nDataFrame after setting single value with .at:\n", df_loc)

# Използваме df_iloc
df_iloc.iat[2, 1] = 9000
print("\nDataFrame after setting single value with .iat:\n", df_iloc)
```

При задаване на стойности чрез селекция е важно да се уверите, че формата на присвояваната стойност (или последователност от стойности) е съвместима с формата на селектираната част от DataFrame-а.

XI. Обобщение на глава "Индексиране и Селектиране на Данни"

Тази глава разгледа основните методи за достъп и модификация на данни в Pandas DataFrame и Series. Ефективното индексиране и селектиране са ключови умения за всеки, който работи с данни, тъй като позволяват извличане на конкретни подмножества от данни за анализ или промяна.

Ето основните методи и концепции, които разгледахме:

- **Селектиране на колони (`[]`, `.` нотация):** Използва се за избиране на една или повече колони. Квадратните скоби са по-гъвкави и се препоръчват пред точковата нотация поради потенциални проблеми с имена на колони и конфликти с атрибути.
- **Селектиране на редове чрез слайсинг (`[]`):** Позволява избиране на диапазон от редове по тяхната целочислена позиция.
- **Селектиране по етикет (`.loc`):** Мощен метод за селектиране по етикети на редове и колони, включително единични етикети, списъци, слайси (с включени крайни точки) и булеви масиви.
- **Селектиране по позиция (`.iloc`):** Аналогичен на `.loc`, но използва целочислени позиции за селекция.
- **Комбинирано селектиране:** Подчертахме опасностите от верижното индексиране и препоръчахме използването на `.loc` и `.iloc` за ясна и безопасна селекция.
- **Булево индексиране:** Филтриране на данни въз основа на логически условия, използващи оператори за сравнение (`==`, `!=`, `>`, `<`, `>=`, `<=`), логически оператори (`&`, `|`, `~`, `^`), методите `.isin()`, `.between()`, `.isnull()`, `.notnull()`.
- **Методът `.query()`:** Селекция на данни чрез низов израз, наподобяващ SQL `WHERE` клауза.
- **Бърз скаларен достъп (`.at`, `.iat`):** Оптимизирани методи за достъп и задаване на единични стойности по етикет (`.at`) или позиция (`.iat`).
- **Индексиране с извикваеми обекти:** Използване на функции за динамично генериране на критерии за селекция.
- **Задаване на стойности чрез селекция:** Промяна на стойности в DataFrame въз основа на селектирани подмножества от данни.

XII. Практически казуси и решения:

Казус 1: Филтриране на продажби по регион и период.

Представете си, че имате DataFrame с данни за продажби, включващ колони като 'Дата', 'Регион', 'Продукт', 'Количество', 'Приход'. Искате да извлечете всички продажби за определен регион ('Север') през месец януари 2023 г.

Решение:

```
import pandas as pd

# Примерни данни
data = {'Дата': pd.to_datetime(['2023-01-05', '2023-01-15', '2023-02-
10', '2023-01-25', '2023-02-20']),
        'Регион': ['Север', 'Юг', 'Север', 'Север', 'Юг'],
        'Продукт': ['A', 'Б', 'A', 'В', 'Б'],
        'Количество': [10, 5, 12, 8, 6],
        'Приход': [100, 75, 120, 96, 90]}
df_sales = pd.DataFrame(data)

# Филтриране по регион и месец
sales_north_january = df_sales.loc[(df_sales['Регион'] == 'Север') &
(df_sales['Дата'].dt.month == 1)]
print(sales_north_january)
```

Казус 2: Извличане на топ 10 най-скъпи продукта.

Имате DataFrame със списък на продукти и техните цени. Искате да извлечете информация за 10-те продукта с най-висока цена.

Решение:

```
# Примерни данни
data_products = {'Продукт': ['X1', 'Y2', 'Z3', 'A4', 'B5', 'C6', 'D7',
'E8', 'F9', 'G10', 'H11', 'I12'],
```

```
                  'Цена': [150, 220, 180, 300, 190, 250, 210, 280, 160,
230, 270, 200]}
df_products = pd.DataFrame(data_products)

# Сортиране по цена и извличане на първите 10
top_10_expensive = df_products.sort_values(by='Цена',
ascending=False).head(10)
print(top_10_expensive)
```

Казус 3: Маркиране на редове с липсващи данни в определени колони.

Имате DataFrame с много колони и искате да идентифицирате редовете, където има липсващи стойности в колоните 'Адрес' или 'Телефон'.

Решение:

```
# Примерни данни с липсващи стойности
data_missing = {'Име': ['Иван', 'Петър', 'Мария', 'Георги'],
                'Възраст': [30, None, 25, 40],
                'Адрес': ['София', None, 'Пловдив', 'Варна'],
                'Телефон': ['123', '456', None, '789']}
df_missing = pd.DataFrame(data_missing)

# Създаване на булев Series, показващ редове с липсващи данни в 'Адрес'
или 'Телефон'
missing_address_phone = df_missing['Адрес'].isnull() |
df_missing['Телефон'].isnull()

# Извличане на редовете с липсващи данни
rows_with_missing = df_missing[missing_address_phone]
print(rows_with_missing)
```

Казус 4: Актуализиране на цените на продукти от определена категория.

Имате DataFrame с продукти, техните категории и цени. Искате да увеличите цените на всички продукти от категория 'Електроника' с 10%.

Решение:

```
# Примерни данни
data_update = {'Продукт': ['Лаптоп', 'Мишка', 'Монитор', 'Телевизор',
'Клавиатура'],
               'Категория': ['Електроника', 'Аксесоари', 'Електроника',
'Електроника', 'Аксесоари'],
               'Цена': [1200, 25, 350, 800, 75]}
df_update = pd.DataFrame(data_update)

# Увеличаване на цените за категория 'Електроника'
df_update.loc[df_update['Категория'] == 'Електроника', 'Цена'] *= 1.10
print(df_update)
```

Тези казуси илюстрират как различните методи за индексиране и селектиране могат да бъдат използвани за решаване на реални проблеми при анализ на данни. Разбирането на тези методи е от съществено значение за ефективната работа с Pandas.

Въпроси

1. Кога е препоръчително да използвате `.loc`, а кога `.iloc` за селектиране на данни? Каква е основната разлика между тях?
2. Обяснете разликата между селектиране на колона с `df['име_на_колона']` и `df[['име_на_колона']]`. Какъв е типът на резултата в двата случая?
3. Какво представлява верижното индексиране и защо е препоръчително да го избягвате при задаване на стойности? Какви са алтернативите?
4. Как се използват логическите оператори `&`, `|`, `~` при булево индексиране в Pandas? Защо не можем да използваме стандартните `and`, `or`, `not`?
5. Обяснете предназначението на методите `.isin()` и `.between()` при булево индексиране. Дайте примери за тяхното използване.
6. Как `.isnull()` и `.notnull()` помагат при работа с липсващи данни по време на селекция?

7. В какви ситуации може да бъде полезен методът `.query()` за селектиране на данни? Какви са неговите предимства и недостатъци?
8. Кога е най-подходящо да използвате `.at` и `.iat` вместо `.loc` и `.iloc`? Какви са ограниченията им?
9. Как можем да използваме извикваеми обекти (функции) за селектиране на данни с `.loc` или `.iloc`? Дайте пример.
10. Как се задават нови стойности на подмножества от данни, селектирани с `[]`, `.loc` и `.iloc`?

Задачи

Използвайте следния примерен DataFrame за решаване на задачите:

```
import pandas as pd

data = {'Име': ['Алиса', 'Боб', 'Чарли', 'Дейвид', 'Ева'],
        'Възраст': [25, 30, 22, 35, 28],
        'Град': ['София', 'Пловдив', 'София', 'Варна', 'Бургас'],
        'Оценка': [8.5, 9.2, 7.8, 9.5, 8.9]}
df_students = pd.DataFrame(data)
```

1. Изведете колоната 'Име'. Какъв е типът на резултата?
2. Изведете DataFrame, съдържащ само колоните 'Име' и 'Оценка'.
3. Изведете първите три реда от DataFrame-а.
4. Изведете реда с индекс 2, използвайки `.loc`. Какъв е етикетът на този индекс?
5. Изведете реда на позиция 1, използвайки `.iloc`.
6. Изведете името на студента на ред с индекс 3, използвайки `.loc`.
7. Изведете оценката на студента на ред на позиция 0, използвайки `.iloc`.
8. Изведете редовете, където възрастта е по-голяма от 25.
9. Изведете редовете, където градът е 'София' и оценката е по-голяма от 8.0.
10. Изведете редовете, където градът е или 'Пловдив', или 'Бургас', използвайки `.isin()`.
11. Изведете редовете, където оценката е между 8.0 и 9.0 (включително), използвайки `.between()`.
12. Изведете всички редове, където поне една от числовите колони ('Възраст', 'Оценка') е по-голяма от 30 или 9.0 съответно.
13. Използвайте `.query()` за да изведете студентите, чиято възраст е по-малка от 30 и градът им е 'София'.
14. Променете оценката на студента с име 'Чарли' на 9.0, използвайки `.loc`.
15. Увеличете възрастта на всички студенти с 1 година.
16. За студентите с оценка по-висока от 9.0, задайте стойност 'Отличен' в нова колона 'Статус'.