Глава X. Зареждане и Записване на Данни

В тази обширна глава ще се задълбочим в многообразните начини, по които библиотеката Pandas позволява да четем и записваме данни от и към различни източници и формати. Успешното справяне с I/O операциите е критично за всеки процес на анализ на данни, тъй като данните често идват в различни форми и е необходимо да бъдат преобразувани в удобни за работа Pandas структури (Series и DataFrame), а след обработката - запазени за бъдеща употреба или споделяне.

Ще започнем с най-често срещаните текстови файлове, като CSV, и ще разгледаме основните параметри на функциите за четене (`read_csv`) и запис (`to_csv`), които ни дават гъвкавост при работа с различни диалекти на тези файлове, както и с големи файлове чрез метода `chunksize`. Ще се спрем и на четенето на файлове с фиксирана ширина.

След това ще разгледаме специфични за Pandas методи за сериализация на обекти (`to_pickle`, `read_pickle`), както и удобните функции за работа с данни от и към клипборда.

Голяма част от главата ще бъде посветена на работата с популярния формат Excel, включително четене от различни работни листове и запис на множество листове в един файл.

Ще продължим с разглеждане на JSON (JavaScript Object Notation) и HTML форматите и как Pandas може да ги чете и записва. Ще засегнем и работата с XML файлове, за която е необходима допълнителна библиотека.

След това ще се запознаем с възможността за запис на данни в LaTeX формат, което е полезно за генериране на таблици за научни публикации.

Следват по-специализирани формати за съхранение на големи набори от данни като HDFStore (HDF5), Feather и Parquet, като ще отбележим и необходимите допълнителни библиотеки за тяхната употреба. Ще разгледаме и по-редки формати като ORC, SAS и SPSS.

В края на главата ще се фокусираме върху интеграцията на Pandas с SQL бази данни, като ще разгледаме методите за четене на данни чрез SQL заявки или директно от таблици, както и за запис на Pandas DataFrames в SQL бази данни. Ще споменем и специфична интеграция с Google BigQuery.

И накрая, ще разгледаме работата със STATA файлове.

Целта на тази глава е да ви предостави изчерпателен преглед на възможностите на Pandas за Input/Output операции, като ви научи как да четете данни от и да записвате данни в голямо разнообразие от формати, с които може да се сблъскате в практиката. След като усвоите тези умения, ще бъдете много по-ефективни в процеса на анализ на данни.

I. Общ преглед на I/O операциите в pandas

Pandas е проектирана да работи безпроблемно с голямо разнообразие от файлови формати и източници на данни. Нейните I/O възможности са мощни и гъвкави, като позволяват на потребителите да четат данни в `DataFrame` и `Series` обекти и да записват тези обекти обратно във файлове или други хранилища на данни.

Основната идея зад I/O функциите на Pandas е да се осигури лесен и ефективен начин за:

- **Зареждане на данни (Reading Data):** Преобразуване на данни от външни източници (файлове, бази данни, уеб API и др.) в Pandas структури от данни (`DataFrame` или `Series`).
- **Записване на данни (Writing Data):** Запазване на Pandas структури от данни във външни формати (файлове, бази данни и др.).

Pandas поддържа голям брой формати "извън кутията", а за някои други изискват инсталирането на допълнителни библиотеки. Ето общ преглед на основните категории I/O операции, които ще разгледаме в тази глава:

1. Работа с плоски файлове (Flat Files):

- **CSV (Comma Separated Values):** Най-често използваният формат за таблични данни. Pandas предлага мощни функции за четене (`pd.read_csv()`) и запис (`df.to_csv()`) на CSV файлове с множество опции за персонализиране на поведението (разделители, заглавни редове, индекси, кодиране и др.).
- **Файлове с фиксирана ширина (Fixed-Width Files):** Файлове, в които данните са подравнени в колони с определена ширина. Pandas осигурява функцията `pd.read_fwf()` за четене на такива файлове.

2. Сериализация на Pandas обекти (Pickling):

- Pandas обектите (Series и DataFrame) могат да бъдат директно сериализирани (запазени в двоичен формат) с помощта на `pd.to_pickle()` и заредени обратно с `pd.read_pickle()`. Този метод е удобен за запазване на междинни резултати или за бързо зареждане на сложни структури от данни.

3. Работа с клипборда:

- Pandas улеснява четенето на данни от клипборда (`pd.read_clipboard()`) и записването на данни в клипборда (`df.to_clipboard()`), което е полезно за бързо прехвърляне на данни между приложения.

4. Работа с Excel файлове:

- Pandas може да чете данни от Excel файлове (`pd.read_excel()`), включително поддръжка на множество работни листове. Записът в Excel файлове (`df.to_excel()`) също е възможен, както и записът на множество DataFrames в различни листове на един файл.

5. JSON (JavaScript Object Notation):

- Pandas предоставя функции за четене на данни във формат JSON (`pd.read_json()`) и за експортиране на Pandas обекти в JSON формат (`df.to_json()`). JSON е често използван формат за уеб базирани приложения и API-та.

6. HTML (HyperText Markup Language):

- Функцията `pd.read_html()` може да чете таблици директно от HTML файлове или уеб страници. Методът `df.to_html()` позволява експортиране на DataFrame в HTML формат.

7. XML (Extensible Markup Language):

- Pandas предлага функция за четене на XML файлове (`pd.read_xml()`) и за записване в XML формат (`df.to_xml()`), но за тези операции често се изисква допълнителната библиотека `lxml`.

8. LaTeX:

- Методът `df.to_latex()` позволява експортиране на DataFrame в LaTeX формат, което е полезно за създаване на таблици за научни документи.

9. HDFStore (PyTables / HDF5):

- Pandas може да взаимодейства с HDF5 файлове чрез библиотеката PyTables. Класът `pd.HDFStore` и функциите `pd.read_hdf()` и `df.to_hdf()` позволяват ефективно съхранение и четене на големи набори от данни в структуриран формат.

10. Feather, Parquet и ORC формати:

- Тези columnar формати са оптимизирани за бързо четене и запис на големи набори от данни. Pandas поддържа тези формати, но обикновено изисква инсталирането на допълнителни библиотеки като `pyarrow` или `fastparquet`.

11. SAS и SPSS файлове:

- Pandas може да чете данни от файлове, създадени със статистически софтуер като SAS (`pd.read_sas()`) и SPSS (`pd.read_spss()`), като за SPSS се изисква библиотеката `pyreadstat`.

12. SQL бази данни:

- Pandas предоставя гъвкави начини за свързване и взаимодействие с SQL бази данни. Функциите `pd.read_sql_query()`, `pd.read_sql_table()` и `pd.read_sql()` позволяват изпълнение на SQL заявки и четене на данни в DataFrame. Методът `df.to_sql()` позволява запис на DataFrame в SQL таблици. За тази функционалност обикновено е необходима библиотека като `SQLAlchemy` и съответният драйвер за конкретната база данни.

13. Google BigQuery:

- Pandas има специфична интеграция с Google BigQuery чрез функциите `pd.read_gbq()` и `df.to_gbq()`, но за тях е необходима библиотеката `pandas-gbq`.

14. STATA файлове:

- Pandas може да чете (`pd.read_stata()`) и записва (`df.to_stata()`) файлове във формат, използван от статистическия софтуер STATA.

В следващите секции ще разгледаме по-подробно всеки от тези видове I/O операции, като ще се фокусираме върху най-често използваните формати и основните параметри на съответните функции.

В следващите секции ще разгледаме по-подробно всеки от тези видове I/O операции, като ще се фокусираме върху най-често използваните формати и основните параметри на съответните функции.

II. Работа с текстови файлове (Flat file): Четене и Запис на CSV

CSV файловете са текстови файлове, в които стойностите са разделени със запетая (или друг разделител), а всеки ред представлява един запис (ред в таблицата).

1. Четене на CSV (read_csv)

Функцията `pd.read_csv()` е основният инструмент в Pandas за четене на данни от CSV файлове (и други текстови файлове с разделители) в DataFrame. Тя предлага множество параметри, които позволяват да се справим с различни структури и особености на CSV файловете. Ето някои от най-важните параметри:

- **`filepath_or_buffer`:** Първият и основен аргумент, който указва пътя до файла (локален или URL адрес) или друг обект, който може да предостави данни (например текстов буфер).
- **`sep` или `delimiter`:** Задава разделителя между полетата. По подразбиране е `,` (запетая). Може да бъде зададен друг символ (например `';'`, `'\t'` за табулация) или дори регулярен израз.
- **`header`:** Определя кои ред(ове) от файла да се използват като заглавни редове (имена на колоните).
    - `header=0`: Използва първия ред (индексиран от 0) като заглавен ред. Това е поведението по подразбиране, ако Pandas може да заключи, че има заглавен ред.
    - `header=None`: Файлът няма заглавен ред и Pandas ще присвои целочислени индекси на колоните (0, 1, 2...).
    - `header=n`: Използва реда с индекс `n` (започвайки от 0) като заглавен ред. Ако има няколко реда заглавия, може да се подаде списък с индекси.
- **`names`:** Списък с имена на колони, които да се използват. Този параметър е полезен, когато файлът няма заглавен ред или искате да презапишете съществуващите имена на колони. Ако се подаде `names`, а `header` не е `None`, то редът, посочен от `header`, ще бъде пропуснат като данни.
- **`index_col`:** Указва коя колона (по име или индекс) да се използва като индекс на DataFrame-а. Може да бъде подаден низ (име на колона) или цяло число (индекс на колона, започвайки от 0). Може да бъде и списък от колони, ако искате MultiIndex.
- **`dtype`:** Речник, в който ключовете са имената на колоните, а стойностите са желаните типове данни (например `'int64'`, `'float64'`, `'str'`, `'datetime64[ns]'`). Използва се за принудително задаване на типа на данните в определени колони.
- **`parse_dates`:** Булев флаг или списък от имена на колони или списъци от имена на колони. Ако е `True`, Pandas ще се опита да парсне всички колони, които изглеждат като дати. Ако е списък с имена на колони, само тези колони ще бъдат парснати като дати. Ако е списък от списъци, ще се опита да комбинира множество колони в една колона с дати (полезно за разделени дата и час).
- **`na_values`:** Списък или речник от стойности, които да се интерпретират като липсващи (`NaN`). Например, може да искате `'NA'`, `'NULL'`, `'-'` да се разпознават като липсващи стойности.

- **encoding:** Задава кодирането на файла (например `'utf-8'`, `'latin-1'`, `'cp1251'`). Важно е да се укаже правилното кодиране, за да се избегнат проблеми с четенето на специални символи.

Примери за четене на CSV:

Да предположим, че имаме файл `data.csv` със следното съдържание:

```
Име,Възраст,Град
Алиса,25,София
Борис,30,Пловдив
Ваня,27,Варна
```

```
import pandas as pd

# Четене на CSV файл с настройки по подразбиране (запетая като разделител,
първи ред като заглавен)
df = pd.read_csv('data.csv')
print("DataFrame от data.csv:\n", df)

# Четене на CSV файл с точка и запетая като разделител, без заглавен ред,
задаване на имена на колони
df2 = pd.read_csv('data_semicolon.csv', sep=';', header=None, names=['Име',
'Години', 'Населено място'])
print("\nDataFrame от data_semicolon.csv:\n", df2)

# Четене на CSV файл с указване на колона за индекс
df3 = pd.read_csv('data.csv', index_col='Име')
print("\nDataFrame от data.csv с 'Име' като индекс:\n", df3)

# Четене на CSV файл с указване на кодиране
df4 = pd.read_csv('data_utf8.csv', encoding='utf-8')
print("\nDataFrame от data_utf8.csv (UTF-8):\n", df4)

# Четене на CSV файл с указване на стойности за NaN
df5 = pd.read_csv('data_with_na.csv', na_values=['NA', '-'])
print("\nDataFrame от data_with_na.csv (с NaN стойности):\n", df5)
```

Разбирането и използването на тези параметри на `pd.read_csv()` е от съществено значение за успешното зареждане на данни от различни CSV файлове.

2. Запис в CSV (to_csv)

Методът `to_csv()` се използва за записване на DataFrame обект в CSV файл. Ето някои от основните му параметри:

- **path_or_buf:** Път до файла (локален или URL адрес) или друг обект, в който да се запишат данните. Ако е `None`, връща се низов формат на CSV.
- **sep:** Разделителят между полетата в записания файл. По подразбиране е `,` (запетая).
- **header:** Булев флаг, който определя дали да се записват имената на колоните (заглавният ред). По подразбиране е `True`. Ако е `False`, заглавният ред няма да бъде включен във файла.

- **`index`:** Булев флаг, който определя дали да се записва индексът на DataFrame-а като колона в CSV файла. По подразбиране е `True`. Ако е `False`, индексът няма да бъде записан. Ако индексът е MultiIndex, могат да се укажат имената на индексните нива.
- **`encoding`:** Задава кодирането на файла (например `'utf-8'`, `'latin-1'`). Препоръчително е да се използва `'utf-8'` за по-добра съвместимост.
- **`mode`:** Режим на отваряне на файла. По подразбиране е `'w'` (запис, презаписване, ако файлът съществува). Може да бъде `'a'` за добавяне към съществуващ файл.

Примери за запис в CSV:

Използвайки DataFrame `df`, създаден в предходния пример:

```
import pandas as pd

# Запис на DataFrame в CSV файл с настройки по подразбиране
df.to_csv('output.csv')

# Запис на DataFrame без индекс
df.to_csv('output_no_index.csv', index=False)

# Запис на DataFrame с различен разделител
df.to_csv('output_semicolon.csv', sep=';')

# Запис на DataFrame без заглавен ред
df.to_csv('output_no_header.csv', header=False)

# Запис на DataFrame с указване на кодиране
df.to_csv('output_utf8.csv', encoding='utf-8')
```

След изпълнението на тези команди ще бъдат създадени няколко CSV файла с различни формати, съответстващи на зададените параметри.

Разбирането на параметрите на `pd.read_csv()` и `df.to_csv()` ви дава голям контрол върху това как данните се четат от и записват в CSV файлове, което е изключително важно при работа с реални набори от данни, които могат да имат различни структури и кодирания.

3. Работа с големи файлове (chunksize)

Когато работите с много големи CSV файлове, зареждането на целия файл в паметта може да доведе до проблеми с производителността или дори до срив на програмата поради изчерпване на паметта. За да се справи с този сценарий, `pd.read_csv()` предлага параметъра `chunksize`.

a) Как работи `chunksize`:

Когато зададете стойност на `chunksize` (цяло число), `pd.read_csv()` не връща един DataFrame, а обект от тип `TextFileReader`, който е итерируем. Всеки път, когато итерирате този обект, той чете следващата "порция" (chunk) от файла и я връща като DataFrame. Размерът на всяка порция (броят на редовете в DataFrame-а) се определя от стойността на `chunksize`.

б) Предимства на използването на `chunksize`:

- **Намалена консумация на памет:** Само част от файла се зарежда в паметта в даден момент.
- **Възможност за обработка на файлове, по-големи от RAM паметта:** Можете да обработвате данните на порции, без да се налага да държите целия файл в паметта.
- **Постепенна обработка:** Позволява извършването на операции върху всяка порция от данните последователно.

в) Пример за четене на голям файл с `chunksize`:

Да предположим, че имаме много голям CSV файл наречен `large_data.csv`. Искаме да прочетем файла на порции от по 1000 реда и да обработим всяка порция.

```
import pandas as pd

# Задаване на размера на порцията
chunk_size = 1000

# Четене на файла на порции
reader = pd.read_csv('large_data.csv', chunksize=chunk_size)

# Итериране през порциите
for chunk in reader:
    # Обработка на текущата порция (chunk)
    print(f"Обработва се порция с {len(chunk)} реда.")
    # Тук можете да извършвате вашите анализи или трансформации върху
'chunk'
    # Например:
    # print(chunk.head())
    # average_value = chunk['колона'].mean()
    # ...

# След като цикълът завърши, всички порции са били обработени.
```

В този пример, `reader` е обект `TextFileReader`. Всяка итерация на цикъла `for` чете следващите 1000 реда от `large_data.csv` и ги връща като DataFrame, който е присвоен на променливата `chunk`. В тялото на цикъла можете да извършвате всякакви необходими операции върху тази порция данни.

г) Пример за агрегиране на данни от голям файл с `chunksize`:

Да предположим, че искаме да намерим средната стойност на дадена колона ('стойност') в голям файл, без да зареждаме целия файл в паметта.

```
import pandas as pd

chunk_size = 1000
total_sum = 0
total_count = 0
```

```
reader = pd.read_csv('large_data.csv', chunksize=chunk_size)

for chunk in reader:
    if 'стойност' in chunk.columns:
        total_sum += chunk['стойност'].sum()
        total_count += len(chunk['стойност'])

if total_count > 0:
    average_value = total_sum / total_count
    print(f"Средната стойност на колона 'стойност' е: {average_value}")
else:
    print("Файлът е празен или колона 'стойност' не съществува.")
```

В този пример, ние итерираме през порциите на файла, изчисляваме сумата и броя на стойностите в колона 'стойност' за всяка порция и ги акумулираме. Накрая, след като всички порции са обработени, изчисляваме средната стойност.

Използването на `chunksize` е мощен метод за работа с големи данни в Pandas, като позволява ефективна обработка с ограничена консумация на памет.

4. Четене на файлове с фиксирана ширина (read_fwf)

Файловете с фиксирана ширина са текстови файлове, в които данните във всяка колона са разположени в полета с предварително определена дължина (ширина). За разлика от CSV, където стойностите са разделени с определен разделител, при файловете с фиксирана ширина позицията на данните определя към коя колона принадлежат.

Pandas предоставя функцията `pd.read_fwf()` за четене на такива файлове. Тази функция е по-специализирана от `pd.read_csv()` и изисква да укажете как са разположени колоните във файла.

а) Основни параметри на `pd.read_fwf()`:

- **`filepath_or_buffer`:** Път до файла (локален или URL адрес) или друг обект, който може да предостави данни.
- **`colspecs`:** Указва разположението на колоните. Може да бъде:
  - Списък от двойки (tuples) от цели числа, задаващи началната и крайната позиция (с включена крайната) на всяка колона. Индексирането започва от 0. Например: `[(0, 5), (6, 15), (16, 20)]` указва три колони, съответно от 0 до 5, от 6 до 15 и от 16 до 20 позиция.
  - Списък от цели числа, указващи ширината на всяка колона. В този случай Pandas ще изчисли началните позиции. Например: `[5, 10, 5]` указва три колони с ширина 5, 10 и 5 знака.
- **`widths`:** Списък от цели числа, указващи ширината на всяка колона. Този параметър е алтернатива на `colspecs` и е по-удобен, когато всички колони имат фиксирана ширина.
- **`names`:** Списък с имена на колони, които да се използват. Подобно на `read_csv()`, този параметър е полезен, ако файлът няма заглавен ред или искате да презапишете съществуващите имена.
- **`header`:** Определя кои ред(ове) от файла да се използват като заглавни редове. Работи по същия начин както при `read_csv()`.
- **`index_col`:** Указва коя колона да се използва като индекс.
- **`dtype`:** Речник за задаване на типове данни на колоните.
- **`encoding`:** Задава кодирането на файла.

- **skiprows:** Брой редове или списък от номера на редове, които да бъдат пропуснати в началото на файла.

б) Примери за четене на файлове с фиксирана ширина:

Да предположим, че имаме файл `fixed_width_data.txt` със следното съдържание:

```
AAA  BBBBBB   CCC
123  45.678   X
987  65.432   Y
```

Пример 1: Използване на `colspecs` (начална и крайна позиция):

```
import pandas as pd

# Указване на позициите на колоните (индексирани от 0, краят е включен)
column_specs = [(0, 4), (6, 12), (15, 18)]
df_fwf1 = pd.read_fwf('fixed_width_data.txt', colspecs=column_specs,
names=['Колона1', 'Колона2', 'Колона3'])
print("DataFrame от fixed_width_data.txt (с colspecs):\n", df_fwf1)
```

```
Изход:

DataFrame от fixed_width_data.txt (с colspecs):
   Колона1  Колона2 Колона3
0    AAAA   BBBBBB      CCC
1     123   45.678        X
2     987   65.432        Y
```

Пример 2: Използване на `widths` (ширина на колоните):

```
import pandas as pd

# Указване на ширината на всяка колона
column_widths = [5, 7, 4]
df_fwf2 = pd.read_fwf('fixed_width_data.txt', widths=column_widths,
names=['Колона A', 'Колона B', 'Колона C'])
print("\nDataFrame от fixed_width_data.txt (с widths):\n", df_fwf2)
```

```
Изход:

DataFrame от fixed_width_data.txt (с widths):
   Колона A  Колона B Колона C
0    AAA    BBBBBB     CCC
1    123     45.678       X
2    987     65.432       Y
```

Забележете, че при използване на `widths`, Pandas автоматично изчислява началните позиции на колоните.

Важно е да се уверите, че правилно сте определили разположението или ширината на колоните във вашия файл с фиксирана ширина, за да може `pd.read_fwf()` да го прочете коректно. В противен случай данните могат да бъдат неправилно интерпретирани и разпределени по колоните.

5. Допълнително при работа с CSV

Въпреки че разгледахме основните параметри на `pd.read_csv()` и `df.to_csv()`, съществуват и други полезни аспекти и ситуации, с които може да се сблъскате:

a) Допълнителни аспекти при четене на CSV (`pd.read_csv()`):

- **Разделители, които са по-дълги от един символ:** Параметърът `sep` може да приеме регулярен израз, което позволява обработка на разделители с повече от един символ. Например, ако данните са разделени с `||`, можете да използвате `sep='\|\|'`.
- **Пропускане на коментари:** Ако CSV файлът съдържа редове с коментари (например започващи с `#`), можете да използвате параметъра `comment` и да зададете символа за коментар. Pandas ще игнорира редовете, започващи с този символ.

```
# Пример: Четене на CSV файл с коментари, започващи с '#'
df_with_comments = pd.read_csv('data_with_comments.csv', comment='#')
```

• **Обработка на празни редове:** Параметърът `skip_blank_lines` (по подразбиране `True`) указва дали да се пропускат празни редове във файла. Можете да го зададете на `False`, ако искате да ги интерпретирате по някакъв начин.

• **Четене на част от колоните:** Вече разгледахме `usecols`, но е важно да се отбележи, че можете да подадете както списък с имена, така и списък с индекси на колони.

• **Итерация през редове (алтернатива на `chunksize` за по-гъвкав контрол):** Въпреки че `chunksize` е ефективен за големи файлове, понякога може да имате нужда от по-финан контрол върху итерацията (например обработка на база ред). Можете да отворите файла ръчно и да го четете ред по ред, след което да парсвате данните.

• **Работа с различни символи за десетична запетая и хиляди:** В зависимост от локализацията на данните, може да се наложи да укажете различни символи за десетична запетая (например `,` вместо `.`) и за разделител на хиляди. Тези параметри са `decimal` и `thousands`.

```
# Пример: Четене на CSV с български формат на числата
df_bg_numbers = pd.read_csv('bulgarian_numbers.csv', decimal=',',
thousands=' ')
```

• **Стрипиране на whitespace:** Параметърът `skipinitialspace` (по подразбиране `False`) указва дали да се пропускат водещи интервали след разделителя.

- **Автоматично откриване на компресия:** Pandas може автоматично да декомпресира файлове, които са компресирани с gzip, bzip2, zip или xz, ако разширението на файла е съответно `.gz`, `.bz2`, `.zip` или `.xz`. Можете също да укажете компресията изрично с параметъра `compression`.

б) Допълнителни аспекти при запис в CSV (`df.to_csv()`):

- **Различни режими на запис:** Освен `'w'` (презаписване) и `'a'` (добавяне), могат да се използват и други режими, поддържани от стандартната функция `open()` на Python.
- **Запис на подмножество от редове:** Можете да филтрирате DataFrame-а преди запис, за да запазите само определени редове.
- **Контрол на символа за нов ред:** Параметърът `line_terminator` позволява да укажете символа или последователността от символи, които да се използват за край на ред в CSV файла (по подразбиране е операционната система-специфичен).
- **Записване без кавички или с различен символ за кавички:** Параметрите `quoting` и `quotechar` позволяват да контролирате как се обработват кавичките около текстовите полета (например дали винаги да се използват, само когато е необходимо и т.н.) и кой символ да се използва за кавички.

```
# Пример: Запис без кавички около текстовите полета
df.to_csv('output_no_quotes.csv', quoting=csv.QUOTE_NONE, sep=',')
```

За да използвате csv.QUOTE_NONE, трябва да импортирате модула csv.

в) Частни случаи

- **CSV файлове с неконсистентна структура:** Понякога CSV файловете могат да имат редове с различен брой колони или неконсистентни разделители. Pandas се опитва да се справи с такива ситуации, но може да се наложи допълнително почистване и предварителна обработка на данните.

- **Много големи CSV файлове с комплексни трансформации:** При изключително големи файлове, дори и с `chunksize`, сложните трансформации в цикъла могат да бъдат бавни. В такива случаи може да се обмисли използването на по-ефективни инструменти за обработка на големи данни (например Dask или PySpark), които могат да работят паралелно.

- **CSV файлове с вложени данни в една колона:** Понякога една колона в CSV файла може да съдържа структурирани данни (например JSON или списъци, представени като низове). В тези случаи ще трябва да прочетете файла и след това да парснете съдържанието на тази колона с помощта на други инструменти (например `json.loads()` или `ast.literal_eval()`).

Казус 1: Анализ на прости данни за продажби от CSV файл

Представете си, че имате CSV файл `sales.csv` със следното съдържание:

```
Продукт,Количество,Цена,Дата
Ябълки,10,1.20,2023-10-26
Банани,5,0.80,2023-10-26
Ябълки,12,1.25,2023-10-27
Портокали,8,1.50,2023-10-27
Банани,15,0.75,2023-10-28
```

Вашата задача е да прочетете този файл в Pandas DataFrame, да преобразувате колоната 'Дата' в datetime формат и да изведете средната цена на всички продадени артикули.

Решение:

```
import pandas as pd

# Четене на CSV файла
df_sales = pd.read_csv('sales.csv')

# Преобразуване на колоната 'Дата' в datetime
df_sales['Дата'] = pd.to_datetime(df_sales['Дата'])

# Изчисляване на общата стойност на всяка продажба
df_sales['Обща стойност'] = df_sales['Количество'] * df_sales['Цена']

# Изчисляване на средната цена на всички продадени артикули (претеглена
средна)
total_quantity = df_sales['Количество'].sum()
total_value = df_sales['Обща стойност'].sum()
average_price = total_value / total_quantity

print("DataFrame след четене и преобразуване:\n", df_sales)
print("\nСредна цена на продадените артикули: {:.2f}".format(average_price))
```

Казус 2: Обработка на голям CSV файл с данни за потребителски сесии

Имате голям CSV файл `user_sessions.csv`, който не може да се побере в паметта наведнъж. Файлът съдържа колони 'user_id', 'session_start', 'session_end', 'duration'. Вашата задача е да прочетете файла на части и да изчислите общия брой уникални потребители във файла.

Решение:

```
import pandas as pd

file_path = 'user_sessions.csv'
unique_users = set()
chunk_size = 10000  # Четене на части по 10000 реда

for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    unique_users.update(chunk['user_id'].unique())

total_unique_users = len(unique_users)
print(f"Общ брой уникални потребители: {total_unique_users}")
```

Забележка: За да работи този код, трябва да имате създаден фиктивен голям CSV файл user_sessions.csv с поне колона 'user_id'.

Казус 3: Четене на данни от файл с фиксирана ширина

Имате текстов файл `fixed_width_data.txt` със следното съдържание, където данните са подравнени с фиксирана ширина:

```
Име     ВъзрастГрад
Алиса   25      София
Борис   30      Пловдив
Ваня    27      Варна
```

Вашата задача е да прочетете този файл в Pandas DataFrame, като укажете правилната ширина на колоните и да зададете подходящи имена на колоните.

Решение:

```
import pandas as pd

file_path = 'fixed_width_data.txt'
column_widths = [8, 5, 10]  # Ширина на колоните 'Име', 'Възраст', 'Град'
column_names = ['Име', 'Възраст', 'Град']

df_fixed_width = pd.read_fwf(file_path, widths=column_widths,
names=column_names, skiprows=1)

print("DataFrame от файл с фиксирана ширина:\n", df_fixed_width)
```

Казус 4*: Анализ на потребителски сесии от уеб сървърни логове (комбиниране на CSV и FWL)

Ситуация: Получавате данни за потребителски сесии от уеб сървърен лог. Основната информация за всяко посещение (време на достъп, IP адрес на потребителя, достъпен URL, статус код) се съдържа в голям CSV файл (`web_access_logs.csv`). Поради специфична конфигурация на сървъра, информацията за продължителността на всяка сесия (време на първо посещение и време на последно посещение за даден потребител в рамките на сесия) се записва в отделен файл с фиксирана ширина (`session_durations.fwf`).

web_access_logs.csv (много голям файл, може да не се побере в паметта изцяло):

```
timestamp,ip_address,url,status_code,user_agent
2025-05-03 10:00:00,192.168.1.100,/home,200,"Mozilla/5.0 (Windows NT 10.0;
Win64; x64)"
2025-05-03 10:00:05,192.168.1.101,/products,200,"Chrome/113.0.0.0"
2025-05-03 10:00:10,192.168.1.100,/about,200,"Mozilla/5.0 (Windows NT 10.0;
Win64; x64)"
2025-05-03 10:00:15,192.168.1.102,/contact,404,"Safari/16.5"
```

```
... (милиони редове)
```

session_durations.fwf (по-малък файл):

```
IP Address      Start Time          End Time
192.168.1.100   20250503100000      20250503100010
192.168.1.101   20250503100005      20250503100005
192.168.1.102   20250503100015      20250503100015
192.168.1.100   20250503100500      20250503100530
...
```

Разстоянието между колоните е фиксирано, но може да варира. Трябва да се определи ширината на всяка колона.)

Задача:

1) Прочетете `web_access_logs.csv` на парчета, за да избегнете претоварване на паметта. Конвертирайте колоната `timestamp` в datetime формат.
2) Определете ширините на колоните в `session_durations.fwf` и го прочетете в DataFrame, като преобразувате колоните `Start Time` и `End Time` в datetime формат. Преименувайте колоните за по-лесно използване (например, 'ip_address', 'session_start', 'session_end').
3) За всяка сесия от `session_durations.fwf`, намерете броя на посещенията (редовете в `web_access_logs.csv`) на съответния IP адрес в рамките на времевия диапазон на сесията (между 'session_start' и 'session_end').
4) Създайте нов DataFrame, който съдържа IP адреса, началото на сесията, края на сесията и броя на посещенията в тази сесия.
5) Запишете резултата в нов CSV файл (`session_analysis.csv`).

Решение:

```
import pandas as pd

# 1. Четене на web_access_logs.csv на парчета
log_chunks = pd.read_csv('web_access_logs.csv', chunksize=100000,
parse_dates=['timestamp'])
all_visits = []
for chunk in log_chunks:
    all_visits.append(chunk)
web_logs_df = pd.concat(all_visits)

# 2. Четене на session_durations.fwf
# Определяне на ширините на колоните (трябва да се инспектира файла)
column_widths = [16, 15, 15]
session_df = pd.read_fwf('session_durations.fwf', widths=column_widths,
skiprows=1)
session_df.columns = ['IP Address', 'Start Time', 'End Time']
```

```
# Преобразуване на времеви колони в datetime
session_df['session_start'] = pd.to_datetime(session_df['Start Time'],
format='%Y%m%d%H%M%S')
session_df['session_end'] = pd.to_datetime(session_df['End Time'],
format='%Y%m%d%H%M%S')
session_df.rename(columns={'IP Address': 'ip_address'}, inplace=True)

# 3. Намиране на броя на посещенията за всяка сесия
session_visit_counts = []
for index, session in session_df.iterrows():
    ip = session['ip_address']
    start = session['session_start']
    end = session['session_end']

    visit_count = web_logs_df[
        (web_logs_df['ip_address'] == ip) &
        (web_logs_df['timestamp'] >= start) &
        (web_logs_df['timestamp'] <= end)
    ].shape[0]
    session_visit_counts.append({'ip_address': ip, 'session_start': start,
'session_end': end, 'visit_count': visit_count})

# 4. Създаване на нов DataFrame с резултатите
session_analysis_df = pd.DataFrame(session_visit_counts)

# 5. Записване на резултата в CSV файл
session_analysis_df.to_csv('session_analysis.csv', index=False)

print("Анализът на потребителските сесии завърши и резултатите са записани в
session_analysis.csv")
print(session_analysis_df.head())
```

Разбор на сложността:

- **Работа с големи файлове:** Налага се четене на CSV файла на парчета, което е ключово за обработка на данни, които не се побират в паметта.
- **Четене на файлове с фиксирана ширина:** Изисква се определяне на структурата на FWL файла и правилното му прочитане.
- **Различни формати на дати:** Датите в двата файла са в различен формат, което налага правилното им парсване.
- **Свързване на данни от различни източници:** Трябва да се свържат данните от двата DataFrame-а въз основа на IP адрес и времеви диапазон. Това е по-сложна операция от простото сливане по обща колона.
- **Итерация и условна логика:** Решението изисква итериране през сесиите и прилагане на условна логика за филтриране на логовете.

Въпроси:

1. Коя е основната функция в Pandas за четене на данни от CSV файл?
2. Избройте поне три често използвани параметъра на функцията `pd.read_csv()` и обяснете тяхното предназначение.
3. Как може да укажете на Pandas кой символ да използва като разделител между колоните във вашия CSV файл?
4. Ако вашият CSV файл няма ред със заглавия на колоните, как ще прочетете данните и ще зададете имена на колоните в Pandas?
5. Как може да зададете коя колона от CSV файла да се използва като индекс на DataFrame-а при четене?
6. Обяснете как параметърът `dtype` може да бъде полезен при четене на CSV файлове. Дайте пример.
7. Как Pandas автоматично разпознава дати при четене на CSV файл? Какво можете да направите, ако форматът на датите не се разпознава автоматично?
8. Как се обработват липсващите стойности (например, представени като `NA` или празни полета) по подразбиране при четене на CSV файл с Pandas? Как можете да персонализирате това поведение?
9. Защо е важно да указвате кодирането на CSV файла при четене и запис? Кои са някои често срещани кодирания?
10. Коя е основната функция в Pandas за записване на DataFrame в CSV файл? Избройте поне три важни параметъра на тази функция и обяснете тяхното предназначение.
11. Как можете да контролирате дали индексът на DataFrame-а да бъде записан в CSV файла?
12. Как може да укажете различен разделител при записване на DataFrame в CSV файл (например, да създадете TSV файл)?
13. Какво представлява параметърът `mode` при записване в CSV файл и кои са някои от неговите възможни стойности?
14. Как може да запишете DataFrame в CSV файл без да включвате заглавния ред?

Задачи:

1. **Прочетете CSV с персонализации:** Създайте CSV файл `data_custom.csv` със следните данни (използвайте запетая като разделител):

```
id;name;age;city;date
1;Alice;25;Sofia;2023/10/26
2;Bob;30;Plovdiv;2023/10/27
3;Charlie;22;Varna;2023/10/28
```

Прочетете този файл в Pandas DataFrame, като използвате `;` като разделител, зададете колоните `id` като индекс, прочетете колоната `date` като datetime обект и зададете имената на колоните съответно на 'ID', 'Име', 'Възраст', 'Град', 'Дата'. Изведете DataFrame-а.

2. **Запишете част от DataFrame в CSV:** Създайте DataFrame с произволни данни (поне 3 колони и 5 реда). След това изберете само две от колоните и запишете този подмножество в нов CSV файл `subset.csv`, като използвате табулация като разделител и не включвате индекса.
3. **Четене на CSV с липсващи стойности:** Създайте CSV файл `data_na.csv` със следните данни:

```
col1,col2,col3
1,a,
2,,c
,b,3
```

```
4,d,NA
```

Прочетете този файл в DataFrame, като укажете, че празните низове и 'NA' трябва да се интерпретират като `NaN`. Изведете DataFrame-а и проверете типа на данните.

4. **Запис с различно кодиране:** Създайте DataFrame с текст, съдържащ символи, които не са ASCII (например, букви на кирилица). Запишете DataFrame-а в CSV файл `bulgarian.csv`, като използвате кодиране 'utf-8'. След това прочетете файла отново с 'utf-8' кодиране и изведете съдържанието, за да се уверите, че данните са прочетени правилно. Опитайте да прочетете същия файл без да указвате кодиране и вижте дали ще има проблеми.

III. Pickling: Сериализация на pandas обекти (to_pickle, read_pickle)

Pickling е процесът на преобразуване на Python обект (като Pandas Series или DataFrame) в поток от байтове, който може да бъде записан във файл. Обратният процес, четенето на този поток от байтове и преобразуването му обратно в оригиналния Python обект, се нарича unpickling.

Pandas предоставя две удобни функции за работа с pickling: `pd.to_pickle()` за записване на обект във файл и `pd.read_pickle()` за четене на обект от файл.

`pd.to_pickle(obj, path):`

- `obj`: Pandas обектът (Series или DataFrame), който искате да сериализирате.
- `path`: Пътят до файла, където ще бъде записан сериализираният обект.

`pd.read_pickle(path):`

- `path`: Пътят до файла, от който искате да прочетете сериализирания Pandas обект.

1. Предимства на Pickling за Pandas обекти:

- **Запазва се типът на данните и структурата:** За разлика от текстовите формати като CSV, pickling запазва точния тип на данните (включително dtypes) и структурата на Pandas обекта, включително индекса и имената на колоните.
- **Ефективност:** Сериализацията и десериализацията на Pandas обекти обикновено са по-бързи от четенето и записването на големи текстови файлове.
- **Удобство за междинно съхранение:** Pickling е много удобен за запазване на междинни резултати от анализа, които могат да бъдат бързо заредени по-късно, без да се налага повторно обработване на данните.

2. Примери за използване на `to_pickle` и `read_pickle`:

Да създадем прост DataFrame и да го запишем във файл с помощта на `to_pickle()`, след което да го прочетем обратно с `read_pickle()`.

```
import pandas as pd

# Създаване на примерен DataFrame
data = {'Колона А': [1, 2, 3], 'Колона Б': [4.5, 5.6, 6.7]}
df_to_pickle = pd.DataFrame(data, index=['Ред 1', 'Ред 2', 'Ред 3'])

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

# Записване на DataFrame във файл 'my_data.pkl'
pickle_file_path = 'my_data.pkl'
df_to_pickle.to_pickle(pickle_file_path)
```

```
print(f"\nDataFrame-ът беше записан във файл: {pickle_file_path}")

# Четене на DataFrame от файла 'my_data.pkl'
df_from_pickle = pd.read_pickle(pickle_file_path)

print("\nDataFrame, прочетен от pickle файл:\n", df_from_pickle)

# Проверка дали оригиналният и прочетеният DataFrame са еднакви
print("\nДали оригиналният и прочетеният DataFrame са еднакви?",
df_to_pickle.equals(df_from_pickle))
```

```
Изход:

Оригинален DataFrame:
        Колона А  Колона Б
Ред 1          1       4.5
Ред 2          2       5.6
Ред 3          3       6.7

DataFrame-ът беше записан във файл: my_data.pkl

DataFrame, прочетен от pickle файл:
        Колона А  Колона Б
Ред 1          1       4.5
Ред 2          2       5.6
Ред 3          3       6.7

Дали оригиналният и прочетеният DataFrame са еднакви? True
```

Както виждате от примера, оригиналният DataFrame е успешно записан във файл my_data.pkl и след това е възстановен от този файл, като запазва своята структура, данни и типове данни.

3. Важно съображение за сигурност:

Трябва да бъдете внимателни, когато зареждате pickle файлове от ненадеждни източници. Pickle форматът може да съдържа произволен Python код, който може да бъде изпълнен по време на процеса на unpickling, което може да доведе до потенциални рискове за сигурността. Затова е препоръчително да зареждате pickle файлове само от доверени източници.

Въпреки това съображение, pickling е много полезен и ефективен начин за запазване и зареждане на Pandas обекти, особено когато работите в контролирана среда.

4. Допълнително за Pickling:

- **Нива на компресия:** Функциите `to_pickle()` и `read_pickle()` поддържат компресия на данните по време на сериализация и десериализация. Можете да укажете метод на компресия (например `'gzip'`, `'bz2'`, `'zip'`, `'xz'`) чрез параметъра `compression`. Компресията може да намали размера на файла на диска, но може да увеличи времето за запис и четене.

```python
import pandas as pd

# Запис с gzip компресия
df.to_pickle('data.pkl.gz', compression='gzip')

# Четене от gzip компресиран файл
df_loaded = pd.read_pickle('data.pkl.gz', compression='gzip')
```

- **Протоколи за Pickling:** Модулът `pickle` на Python предлага различни протоколи за сериализация, които се различават по съвместимост между версиите на Python и ефективност. Pandas използва протокола по подразбиране, но можете да го контролирате чрез параметър, ако директно използвате модула `pickle`. За функциите на Pandas това обикновено не е пряко изложено като параметър, но е важно да се знае, че различни версии на Pandas могат да използват различни версии на протокола по подразбиране.

- **Запазване на метаданни:** Pickling запазва не само данните, но и метаданните на Pandas обектите, като `dtype`, `index`, `columns`, `name` (за Series) и други атрибути. Това е едно от основните предимства пред текстовите формати.

- **Използване с `HDFStore`:** Pickling се използва вътрешно от `HDFStore` (PyTables) за сериализация на Pandas обекти, когато се записват във HDF5 формат. `HDFStore` предлага по-ефективен начин за съхранение и query на големи набори от данни.

5. Изключения и потенциални проблеми при Pickling:

- **Съвместимост между версии на Pandas:** Pickle файлове, създадени с една версия на Pandas, не винаги са съвместими с други версии. Възможно е да възникнат грешки при опит за четене на pickle файл с по-нова или по-стара версия на Pandas. Препоръчително е да се използва една и съща версия на Pandas за запис и четене на pickle файлове.

- **Съвместимост между версии на Python:** Подобно на съвместимостта с Pandas, съвместимостта между различни версии на Python също може да бъде проблем при pickle файлове, особено ако се използват по-нови протоколи на `pickle`.

- **Проблеми с преместване на код между среди:** Ако кодът, който създава и чете pickle файлове, се премества между среди с различни инсталирани библиотеки (например различни версии на NumPy или други зависимости), може да възникнат проблеми при десериализацията, ако сериализираният обект съдържа данни или метаданни, зависещи от тези библиотеки.

- **Размер на файла:** За много големи DataFrames, pickle файловете могат да бъдат доста големи, въпреки че компресията може да помогне. В такива случаи columnar формати като Parquet или HDF5 могат да бъдат по-ефективни по отношение на размера и скоростта на четене.

- **Сигурност (повторение):** Както беше споменато и преди, никога не трябва да unpickle файлове от ненадеждни източници, тъй като те могат да съдържат злонамерен код.

- **Проблеми при сериализация на сложни обекти в DataFrame:** Ако вашият DataFrame съдържа колони с много сложни или нестандартни Python обекти, те може да не се сериализират коректно с pickle или могат да доведат до неочаквано големи файлове. В такива случаи може да се наложи да трансформирате тези обекти в по-прости, сериализируеми формати, преди да използвате `to_pickle()`.

Казус 1: Запазване на междинни резултати от сложен анализ

Представете си, че извършвате дълъг и сложен анализ на голям набор от данни с Pandas. В процеса на работа създавате няколко междинни DataFrame-а, които съдържат важни резултати от различни етапи на анализа (например, почистени данни, агрегирани данни, резултати от feature engineering). Искате да запазите тези междинни резултати, за да можете по-късно да продължите работата си от същата точка, без да се налага да изпълнявате целия анализ отново.

Решение:

Използвайте `to_pickle()` за запазване на всеки междинен DataFrame в отделен `.pkl` файл.

```
import pandas as pd

# Да предположим, че имаме DataFrame след почистване на данни
cleaned_data = pd.DataFrame({'user_id': [1, 2, 1, 3], 'action': ['view',
'click', 'view', 'purchase']})
cleaned_data.to_pickle('cleaned_data.pkl')
print("Почистените данни бяха запазени.")

# Да предположим, че след това сме извършили агрегация
aggregated_data =
cleaned_data['user_id'].value_counts().reset_index(name='count')
aggregated_data.to_pickle('aggregated_data.pkl')
print("Агрегираните данни бяха запазени.")

# По-късно, за да продължите работата:
loaded_cleaned_data = pd.read_pickle('cleaned_data.pkl')
loaded_aggregated_data = pd.read_pickle('aggregated_data.pkl')

print("\nПочистените данни след зареждане:\n", loaded_cleaned_data)
print("\nАгрегираните данни след зареждане:\n", loaded_aggregated_data)
```

Казус 2: Споделяне на обработени данни между различни скриптове или процеси

Имате Python скрипт, който чете сурови данни, извършва сложна обработка с Pandas и създава DataFrame с готови за анализ резултати. Искате да използвате тези обработени данни в друг Python скрипт или да ги предоставите на друг процес, без да се налага да повтаряте цялата обработка.

Решение:

В първия скрипт запазете крайния DataFrame като `.pkl` файл.

```
import pandas as pd

# ... (четене на сурови данни и сложна обработка) ...
final_analyzed_data = pd.DataFrame({'feature1': [0.1, 0.5, 0.2], 'target':
[0, 1, 0]})
```

```
final_analyzed_data.to_pickle('analyzed_data.pkl')
print("Обработените данни бяха запазени за споделяне.")
```

Във втория скрипт заредете запазения DataFrame.

```
import pandas as pd

shared_data = pd.read_pickle('analyzed_data.pkl')
print("\nСподелените обработени данни:\n", shared_data)
# ... (продължаване на анализа или използване на данните) ...
```

Казус 3: Кеширане на резултати от API заявки

Представете си, че често извличате данни от API, които се връщат като JSON и ги преобразувате в Pandas DataFrame. Тъй като API заявките могат да бъдат бавни или да имат ограничения в броя на заявките, искате да кеширате получените DataFrames локално, за да ги използвате повторно, без да правите излишни API calls.

Решение:

След успешно получаване и преобразуване на данните от API, запазете DataFrame-а като `.pkl` файл. Преди да правите нова API заявка, проверете дали съответният `.pkl` файл съществува и го заредете, ако е наличен.

```
import pandas as pd
import os
import requests
import json
from datetime import datetime

def get_data_from_api(api_url, cache_file):
    if os.path.exists(cache_file):
        print(f"Зареждане на данни от кеш: {cache_file}")
        try:
            data = pd.read_pickle(cache_file)
            return data
        except Exception as e:
            print(f"Грешка при четене на кеш: {e}. Извличане на нови данни
от API.")
            pass

    print(f"Извличане на данни от API: {api_url}")
    response = requests.get(api_url)
    if response.status_code == 200:
        json_data = response.json()
        data = pd.DataFrame(json_data) # Предполагаме, че JSON може директно
да се превърне в DataFrame
        try:
            data.to_pickle(cache_file)
            print(f"Данните бяха кеширани в: {cache_file}")
        except Exception as e:
```

```
            print(f"Грешка при запазване на кеш: {e}")
        return data
    else:
        print(f"Грешка при API заявка: {response.status_code}")
        return None

api_url = "https://example-api.com/data" # Заменете с реален URL
cache_file = f"api_data_{datetime.now().strftime('%Y%m%d')}.pkl" # Кеш файл
с дата

df_api_data = get_data_from_api(api_url, cache_file)

if df_api_data is not None:
    print("\nДанни от API (или кеш):\n", df_api_data.head())
```

Казус 4*: Запазване и възстановяване на състоянието на сложен анализ на пазарни данни

Ситуация:

Разработвате скрипт за анализ на пазарни данни, който включва няколко сложни стъпки:

1. Четене на големи CSV файлове с исторически цени на акции (може да отнеме време).
2. Изчисляване на различни технически индикатори (например, пълзящи средни, RSI, MACD) за всяка акция. Тези изчисления могат да бъдат времеемки.
3. Извършване на статистически анализ и машинно обучение за идентифициране на потенциални възможности за търговия. Този процес включва обучение на модели и оценка на резултатите.
4. Генериране на отчети и визуализации.

Поради обема на данните и сложността на анализа, целият процес отнема значително време. Искате да имате възможност да запазите състоянието на анализа след всяка важна стъпка, за да можете:

- Да рестартирате скрипта от последната успешно завършена стъпка, без да се налага да повтаряте всички предходни изчисления.
- Да споделяте междинни резултати (например, DataFrame-и с изчислени индикатори или обучени модели) с други членове на екипа.
- Да извършвате експерименти, като зареждате запазено състояние, променяте параметри и сравнявате резултатите.

Задача:

1. Симулирайте четенето на голям CSV файл с пазарни данни и създайте Pandas DataFrame.
2. Напишете функции за изчисляване на поне два различни технически индикатора, които приемат DataFrame като вход и връщат DataFrame с добавени колони за индикаторите. Приложете тези функции към симулираните данни.
3. След изчисляването на индикаторите, запазете получения DataFrame (съдържащ оригиналните данни и индикаторите) като pickle файл.
4. В отделна част на скрипта (или при следващо изпълнение), прочетете DataFrame-а от pickle файла, без да повтаряте четенето на CSV и изчисляването на индикаторите.

5. Симулирайте обучението на прост модел за прогнозиране (например, използвайте `sklearn`). Запазете обучения модел като pickle файл.
6. В друга част на скрипта, прочетете обучения модел от pickle файла и го използвайте за правене на примерни прогнози върху прочетените пазарни данни.

Решение:

```
import pandas as pd
import numpy as np
import pickle
from sklearn.linear_model import LinearRegression

# --- 1. Симулиране на четене на пазарни данни ---
def simulate_market_data(num_rows=1000):
    dates = pd.to_datetime(pd.date_range('2024-01-01', periods=num_rows))
    data = {
        'Open': np.random.rand(num_rows) * 100,
        'High': np.random.rand(num_rows) * 100 + 5,
        'Low': np.random.rand(num_rows) * 100 - 5,
        'Close': np.random.rand(num_rows) * 100,
        'Volume': np.random.randint(1000, 100000, num_rows)
    }
    return pd.DataFrame(data, index=dates)

market_df = simulate_market_data()
print("Симулирани пазарни данни:")
print(market_df.head())

# --- 2. Функции за изчисляване на технически индикатори ---
def calculate_sma(df, window=20):
    df['SMA'] = df['Close'].rolling(window=window).mean()
    return df

def calculate_rsi(df, period=14):
    delta = df['Close'].diff(1)
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)
    avg_gain = gain.rolling(window=period, min_periods=period).mean()
    avg_loss = loss.rolling(window=period, min_periods=period).mean()
    rs = avg_gain / avg_loss
    df['RSI'] = 100 - (100 / (1 + rs))
    return df

# Прилагане на индикаторите
market_df = calculate_sma(market_df)
market_df = calculate_rsi(market_df)
print("\nПазарни данни с добавени индикатори:")
print(market_df.head(30))

# --- 3. Запазване на DataFrame като pickle файл ---
pickle_file_path_df = 'market_data_with_indicators.pkl'
try:
```

```
    market_df.to_pickle(pickle_file_path_df)
    print(f"\nDataFrame-ът е запазен като: {pickle_file_path_df}")
except Exception as e:
    print(f"Грешка при запазване на DataFrame като pickle: {e}")

# --- 4. Четене на DataFrame от pickle файл (при следващо изпълнение) ---
loaded_market_df = None
try:
    loaded_market_df = pd.read_pickle(pickle_file_path_df)
    print(f"\nDataFrame-ът е успешно прочетен от: {pickle_file_path_df}")
    print(loaded_market_df.head())
except FileNotFoundError:
    print(f"Файлът '{pickle_file_path_df}' не е намерен.")
except Exception as e:
    print(f"Грешка при четене на DataFrame от pickle: {e}")

if loaded_market_df is not None:
    # --- 5. Симулиране на обучение на модел и запазване като pickle ---
    model = LinearRegression()
    # Да предположим, че 'SMA' и 'RSI' са features, а 'Close' е target
    model.fit(loaded_market_df[['SMA', 'RSI']].dropna(),
loaded_market_df['Close'].dropna())

    pickle_file_path_model = 'trading_model.pkl'
    try:
        with open(pickle_file_path_model, 'wb') as file:
            pickle.dump(model, file)
        print(f"\nОбученият модел е запазен като: {pickle_file_path_model}")
    except Exception as e:
        print(f"Грешка при запазване на модела като pickle: {e}")

    # --- 6. Четене на модела от pickle и правене на прогнози ---
    loaded_model = None
    try:
        with open(pickle_file_path_model, 'rb') as file:
            loaded_model = pickle.load(file)
        print(f"\nМоделът е успешно прочетен от: {pickle_file_path_model}")

        # Правене на примерни прогнози
        last_data = loaded_market_df[['SMA', 'RSI']].dropna().tail(5)
        predictions = loaded_model.predict(last_data)
        print("\nПримерни прогнози:")
        print(predictions)

    except FileNotFoundError:
        print(f"Файлът '{pickle_file_path_model}' не е намерен.")
    except Exception as e:
        print(f"Грешка при четене на модела от pickle: {e}")
```

Въпроси:

1. Какво е "pickling" в контекста на Pandas и защо е полезно?
2. Кои функции в Pandas се използват за сериализация и десериализация на Pandas обекти с помощта на pickling?
3. Какви типове Pandas обекти могат да бъдат сериализирани с `to_pickle()`?
4. Каква е разликата между запазване на DataFrame в CSV файл и в pickle файл? В кои ситуации е по-подходящо да се използва pickle?
5. Какви са потенциалните проблеми със сигурността при десериализация на pickle файлове от ненадеждни източници?
6. Как може да запазите DataFrame в pickle файл? Дайте пример.
7. Как може да заредите DataFrame от pickle файл? Дайте пример.
8. Какви са предимствата на използването на pickle за запазване на междинни резултати от сложен анализ?
9. В какви ситуации е полезно да се използва pickle за споделяне на данни между различни Python скриптове или процеси?
10. Как може да използвате pickle за кеширане на резултати от API заявки?

Задачи:

1. **Запазване и зареждане на DataFrame:** Създайте DataFrame с произволни данни. Запазете го в pickle файл с име `my_data.pkl`. След това прочетете pickle файла обратно в нов DataFrame и изведете съдържанието му, за да се уверите, че данните са запазени и възстановени правилно.
2. **Запазване на няколко обекта:** Създайте два различни Pandas обекта (например, DataFrame и Series). Запазете ги в два отделни pickle файла (`df.pkl` и `series.pkl`). След това ги заредете обратно и изведете техните типове и съдържание.
3. **Кеширане на данни:** Симулирайте функция, която извлича данни (например, произволни числа). Добавете логика, която проверява дали съществува pickle файл с кеширани данни. Ако файлът съществува, заредете данните от него. В противен случай, генерирайте нови данни, запазете ги в pickle файл и върнете генерираните данни.
4. **Запазване на DataFrame с различни типове данни:** Създайте DataFrame, който съдържа различни типове данни (числа, низове, дати). Запазете го в pickle файл и след това го заредете обратно. Проверете типовете на данните в заредения DataFrame, за да се уверите, че са запазени правилно.

IV. Работа с клипборда (read_clipboard, to_clipboard)

Тези две функции в Pandas (read_clipboard, to_clipboard) предоставят удобен начин за бързо прехвърляне на данни между Pandas DataFrames/Series и други приложения, които поддържат копиране и поставяне на таблични данни (например електронни таблици, уеб страници с таблици).

1. Четен от Clipboard-а - `pd.read_clipboard (sep='\s+', **kwargs)`:

а) Основни параметри на pd.read_clipboard

- Тази функция чете данни от клипборда и ги връща като DataFrame.
- **`sep`:** Задава разделителя на данните в клипборда. По подразбиране е `'\s+'`, което означава един или повече whitespace символи (интервали, табулации, нови редове). Можете да зададете друг разделител, ако данните във вашия клипборд са разделени по различен начин (например `,` за CSV данни).
- **`**kwargs`:** Приема допълнителни ключови аргументи, които могат да бъдат подадени на `pd.read_clipboard()`. Това ви позволява да контролирате процеса на парсване на данните от клипборда (например указване на заглавен ред, имена на колони и др.).

б) Пример за използване на `read_clipboard()`:

1) Копирайте следните данни (например от текстов редактор или електронна таблица) във вашия клипборд:

```
Име     Възраст  Град
Алиса   25       София
Борис   30       Пловдив
Ваня    27       Варна
```

2) Изпълнете следния Python код:

```
import pandas as pd

df_from_clipboard = pd.read_clipboard()
print(df_from_clipboard)
```

3) Pandas ще прочете данните от клипборда, ще ги парсне (като използва whitespace като разделител по подразбиране) и ще създаде DataFrame:

```
Изход:

     Име  Възраст    Град
0  Алиса        25  София
```

```
1   Борис        30   Пловдив
2    Ваня        27     Варна
```

4) Ако данните във вашия клипборд са разделени с друг символ (например запетая), можете да го укажете чрез параметъра `sep`:

- Копирайте следните CSV данни в клипборда:

```
Име,Възраст,Град
Алиса,25,София
Борис,30,Пловдив
Ваня,27,Варна
```

- Изпълнете:

```
import pandas as pd

df_from_clipboard_csv = pd.read_clipboard(sep=',')
print(df_from_clipboard_csv)
```

- Изход:

```
     Име  Възраст     Град
0  Алиса        25   София
1  Борис        30  Пловдив
2   Ваня        27    Варна
```

5) Можете също да използвате други параметри на `pd.read_csv()` чрез `**kwargs`, например за да укажете, че няма заглавен ред:

```
# Ако клипбордът съдържа:
# Алиса 25 София
# Борис 30 Пловдив
# Ваня 27 Варна

df_no_header = pd.read_clipboard(header=None, sep='\s+', names=['Име',
'Възраст', 'Град'])
print(df_no_header)
```

2. Запис в Clipbord-a - df.to_clipboard(sep='\t', index=True, **kwargs):

a) Основни параметри на to_clipboard

- Този метод записва съдържанието на DataFrame (или Series) в клипборда.
- **sep:** Задава разделителя, който ще се използва между колоните в клипборда. По подразбиране е `'\t'` (табулация), което е удобно за поставяне в електронни таблици.

- **`index`:** Булев флаг, който определя дали индексът на DataFrame-а да бъде включен в данните, копирани в клипборда. По подразбиране е `True`.
- **`**kwargs`:** Приема допълнителни ключови аргументи, които могат да бъдат подадени на `df.to_csv()`.

б) Пример за използване на `to_clipboard()`:

Използвайки DataFrame `df_from_clipboard`, създаден по-рано:

```
import pandas as pd

# Запис на DataFrame в клипборда (с индекс, разделен с табулация)
df_from_clipboard.to_clipboard()
print("DataFrame-ът беше копиран в клипборда (с индекс, разделен с
табулация).")

# Запис на DataFrame в клипборда без индекс, разделен със запетая
df_from_clipboard.to_clipboard(index=False, sep=',')
print("DataFrame-ът беше копиран в клипборда (без индекс, разделен със
запетая).")
```

След изпълнението на тези команди, съдържанието на df_from_clipboard ще бъде копирано във вашия клипборд в съответния формат, готов за поставяне в друго приложение.

Тези функции за работа с клипборда са изключително удобни за бързи операции с данни, особено когато искате да прехвърлите малки набори от данни между Python и други приложения без да се налага да записвате и четете файлове.

3. Допълнително при работа с клипборда:

- **Зависимост от операционната система и графичната среда:** Функционалността на клипборда зависи от операционната система и графичната среда, която използвате. В някои среди (например сървърни без GUI) може да няма достъпен клипборд или поведението може да е различно.

- **Формати на данните в клипборда:** Данните в клипборда могат да бъдат в различни формати (например текст, RTF, HTML). `pd.read_clipboard()` очаква текстови данни, разделени по определен начин. Ако клипбордът съдържа данни в друг формат, `read_clipboard()` може да не работи коректно или да върне неочаквани резултати.

- **Копиране от различни приложения:** Различните приложения могат да копират таблични данни в клипборда с различни разделители (например табулация, запетая, интервали) и форматиране. Важно е да знаете как са форматирани данните, които копирате, за да зададете правилния `sep` параметър на `pd.read_clipboard()`.

- **Възможност за празен клипборд:** Ако клипбордът е празен, `pd.read_clipboard()` ще върне празен DataFrame.

- **Копиране на MultiIndex:** `df.to_clipboard()` може да копира DataFrame с MultiIndex. По подразбиране нивата на индекса се разделят с табулация. При поставяне в други приложения може да се наложи допълнително обработване на тези данни.

- **Копиране на голям обем данни:** Въпреки удобството, копирането на много голям обем данни в клипборда може да бъде бавно и да заеме значително количество памет. За големи набори от данни е по-препоръчително да се използват файлове.

4. Изключения и потенциални проблеми с Клипборда

- **`FileNotFoundError` (или подобни):** В редки случаи, свързани с проблеми с достъпа до системния клипборд, може да възникнат грешки, свързани с файлови операции или липса на достъп.

- **Неправилно парсване на данни:** Ако `sep` параметърът на `pd.read_clipboard()` не съответства на действителния разделител в клипборда, данните ще бъдат парснати неправилно, което може да доведе до грешен DataFrame.

- **Проблеми с кодирането:** Подобно на четенето от файлове, може да има проблеми с кодирането на текста в клипборда, особено ако се копират данни, съдържащи специални символи от приложения с различно кодиране. В повечето съвременни системи това не е често срещан проблем, но е възможно.

- **Ограничения на размера на клипборда:** Операционните системи могат да имат ограничения за размера на данните, които могат да бъдат съхранени в клипборда. При опит за копиране на много голям DataFrame с `to_clipboard()` може да възникне грешка или данните да бъдат отрязани.

- **Различно поведение в различни среди:** Поведението на `read_clipboard()` и `to_clipboard()` може леко да варира в зависимост от операционната система, инсталираните библиотеки (например `pyperclip`, която Pandas може да използва вътрешно) и графичната среда.

Казус 1: Бърз анализ на данни, копирани от уебсайт

Представете си, че разглеждате уебсайт с таблични данни (например, статистическа информация, ценови листи). Искате бързо да анализирате тези данни в Pandas, без да се налага да ги сваляте като файл. Можете да копирате таблицата директно от уебсайта (чрез select + Ctrl/Cmd + C) и след това да я прочетете в Pandas.

Решение:

Използвайте `pd.read_clipboard()` за директно прочитане на данните от клипборда в DataFrame.

```
import pandas as pd

# След като сте копирали таблицата от уебсайта в клипборда:
df_from_clipboard = pd.read_clipboard()

# Сега можете да работите с df_from_clipboard, сякаш сте го прочели от файл.
print("DataFrame от клипборда:\n", df_from_clipboard)

# Например, може да искате да видите основни статистически данни:
print("\nОсновни статистики:\n", df_from_clipboard.describe())
```

Важно: Успешното прочитане зависи от това как уебсайтът форматира таблицата при копиране (разделители, заглавни редове и т.н.). В някои случаи може да се наложи да използвате параметъра `sep` или други параметри на `read_clipboard()`, ако Pandas не разпознае автоматично структурата.

Казус 2: Експортиране на част от DataFrame за бързо споделяне или вмъкване в друг документ

Работите с голям DataFrame в Pandas и искате бързо да споделите малка част от него с колега през чат приложение или да я вмъкнете като таблица в имейл или текстов документ.

Решение:

Използвайте метода `df.to_clipboard()` на желания подмножество от DataFrame-а.

```
import pandas as pd

# Да предположим, че имаме DataFrame с резултати от анализ
results_df = pd.DataFrame({'Потребител': ['A', 'Б', 'В', 'Г'], 'Резултат 1':
[10, 12, 9, 15], 'Резултат 2': [2.5, 3.1, 2.8, 3.5]})

# Искаме да копираме само първите два реда и колоните 'Потребител' и
'Резултат 1'
subset_to_copy = results_df.head(2)[['Потребител', 'Резултат 1']]
subset_to_copy.to_clipboard(index=False) # Копиране без индекса

print("Първите два реда от DataFrame-а (колони 'Потребител' и 'Резултат 1')
бяха копирани в клипборда.")

# Сега можете да поставите съдържанието на клипборда директно в чат
приложение, имейл и т.н.
# (Обикновено с Ctrl/Cmd + V)
```

По подразбиране `to_clipboard()` използва табулация като разделител, което често е добре разпознато от други приложения. Параметърът `index=False` предотвратява копирането на индекса на DataFrame-а.

Казус 3: Временно прехвърляне на данни между различни Python среди или PyCharm

Работите в една Python среда (например, PyCharm) и искате бързо да прехвърлите малък DataFrame или Series в друга Python среда или друг PyCharm, без да записвате и четете файл.

Решение:

В първата среда копирайте DataFrame-а в клипборда:

```
import pandas as pd
```

```
# Да предположим, че имаме DataFrame
temp_df = pd.DataFrame({'col1': [1, 2, 3], 'col2': ['a', 'b', 'c']})
temp_df.to_clipboard(index=False)
print("DataFrame-ът беше копиран в клипборда.")
```

След това, във втората Python среда или PyCharm, прочетете данните от клипборда:

```
import pandas as pd

loaded_df = pd.read_clipboard()
print("DataFrame-ът беше прочетен от клипборда:\n", loaded_df)
```

Този метод е удобен за бързо прехвърляне на малки количества данни по време на интерактивна работа.

Казус 4*: Интегриране на данни от уеб приложение в локален анализ

Ситуация:

Работите като анализатор на данни и използвате уеб базирано приложение за генериране на справки за продажби. Приложението ви позволява да филтрирате данните по различни критерии (продукт, регион, период) и да копирате резултата като текст в клипборда. Текстът, копиран в клипборда, обаче не е в стандартен CSV формат. Той съдържа заглавна част с обща информация за справката, следвана от таблични данни с променлив брой празни редове между секциите и специфични символи за разделители, които не са консистентни.

Пример на текст, копиран в клипборда:

```
Отчет за продажбите - Период: 2025-01-01 до 2025-04-30
Регион: Европа

Продукт | Брой продажби | Обща стойност (EUR)
------- | -------- | --------
A       | 120      | 1500.50
B       | 85       | 925.75
Продукт | Брой продажби | Обща стойност (USD)
------- | -------- | --------
C       | 210      | 2520.00
D       | 150      | 1800.00
E       | 95       | 1140.25
```

Задача:

1. Прочетете съдържанието на клипборда в Pandas DataFrame. Тъй като форматът е неструктуриран, първоначално ще го прочетете като единична текстова колона.
2. Парсвайте текста, за да извлечете информацията за периода и региона от заглавната част.

3. Идентифицирайте началото на всяка таблица с данни (например, по наличието на линия с разделители '-------').
4. Разделете текста на отделни секции, представляващи таблиците с данни (за EUR и USD продажби).
5. За всяка секция, създайте Pandas DataFrame с подходящи имена на колони ('Продукт', 'Брой продажби', 'Обща стойност'). Преобразувайте колоната 'Брой продажби' в числов тип и 'Обща стойност' в числов тип (като запазите валутата).
6. Обединете DataFrame-ите от различните валути в един общ DataFrame, като добавите колона 'Валута' за указване на валутата на продажбите.
7. Изчислете общата стойност на продажбите в EUR, като конвертирате USD сумите по фиксиран курс (например, 1 USD = 0.92 EUR).
8. Създайте нов DataFrame, съдържащ 'Продукт', 'Брой продажби' и 'Обща стойност (EUR)'.
9. Копирайте получения обобщен DataFrame в клипборда, разделен с табулация, без индекс и със заглавен ред.

Решение:

```
import pandas as pd
import pyperclip  # Увери се, че библиотеката е инсталирана: pip install
pyperclip

# 1. Четене на клипборда като обикновен текст
clipboard_text = pyperclip.paste().splitlines()

# 2. Извличане на информация от заглавната част
period = None
region = None
for line in clipboard_text:
    if "Период:" in line:
        period = line.split(": ")[1]
    elif "Регион:" in line:
        region = line.split(": ")[1]

print(f"Отчетен период: {period}")
print(f"Регион: {region}")

# 3. Идентифициране на началото на таблиците
table_starts = [i for i, line in enumerate(clipboard_text) if '-------' in
line]

# 4. Разделяне на текста на секции с данни
data_sections = []
for i in range(len(table_starts)):
    start = table_starts[i] + 1
    end = table_starts[i+1] - 1 if i + 1 < len(table_starts) else
len(clipboard_text)
    data_sections.append(clipboard_text[start:end])

# 5. Създаване на DataFrame-и за всяка секция
dfs = []
for section_index, section in enumerate(data_sections):
    # Определяне на валутата от заглавния ред на таблицата
    header_line = clipboard_text[table_starts[section_index] - 1]
```

```
    currency = 'EUR' if 'EUR' in header_line else 'USD'

    # Обработка на таблицата
    data = [line.split('|') for line in section if line.strip()]
    df = pd.DataFrame(data, columns=['Продукт', ' Брой продажби ', ' Обща
стойност '])
    df['Валута'] = currency

    # Преобразуване на числовите стойности
    df['Брой продажби'] = pd.to_numeric(df[' Брой продажби '].str.strip(),
errors='coerce')
    df['Обща стойност'] = pd.to_numeric(df[' Обща стойност '].str.strip(),
errors='coerce')

    dfs.append(df[['Продукт', 'Брой продажби', 'Обща стойност', 'Валута']])

# 6. Обединяване на всички таблици
all_sales_df = pd.concat(dfs, ignore_index=True)

# 7. Конвертиране на USD в EUR
usd_to_eur = 0.92
all_sales_df['Обща стойност (EUR)'] = all_sales_df.apply(
    lambda row: row['Обща стойност'] * usd_to_eur if row['Валута'] == 'USD'
else row['Обща стойност'],
    axis=1
)

# 8. Обобщен резултат
summary_df = all_sales_df[['Продукт', 'Брой продажби', 'Обща стойност
(EUR)']]
print("\nОбобщени продажби в EUR:")
print(summary_df)

# 9. Копиране обратно в клипборда
summary_df.to_clipboard(index=False, header=True, sep='\t')
print("\nОбобщените данни са копирани в клипборда.")
```

Въпроси:

1. Каква е основната цел на функциите `pd.read_clipboard()` и `df.to_clipboard()` в Pandas?
2. В какви ситуации е полезно да използвате `pd.read_clipboard()`? Дайте поне два примера от реалния живот.
3. Какви данни може да прочете `pd.read_clipboard()` от клипборда? В какъв формат трябва да бъдат тези данни?
4. Ако данните в клипборда са разделени с различен символ от табулация (което е по подразбиране), как можете да укажете това на `pd.read_clipboard()`?
5. Какви са потенциалните проблеми при четене на данни от клипборда, които трябва да имате предвид?
6. В какви ситуации е полезно да използвате метода `df.to_clipboard()`? Дайте поне два примера от реалния живот.
7. Какъв е форматът на данните, които `df.to_clipboard()` поставя в клипборда по подразбиране? Кой е разделителят?
8. Как можете да предотвратите записването на индекса на DataFrame-а при използване на `df.to_clipboard()`?
9. Можете ли да укажете различен разделител (например, запетая) при използване на `df.to_clipboard()`? Ако да, как?
10. Как можете да копирате само определени колони от DataFrame-а в клипборда?

Задачи:

1. **Четене от уебсайт:** Отворете уебсайт с таблични данни (например, списък с най-високите сгради в света от Wikipedia). Копирайте част от таблицата (няколко реда и колони) в клипборда. След това използвайте `pd.read_clipboard()` в Pandas, за да прочетете данните в DataFrame. Изведете DataFrame-а и проверете неговата структура.
2. **Копиране на подмножество:** Създайте DataFrame с поне 4 колони и 5 реда с произволни данни. Изберете подмножество от DataFrame-а (например, първите 3 реда и две от колоните). Копирайте това подмножество в клипборда, без да включвате индекса и използвайки запетая като разделител. След това (симулирайки поставяне в текстов редактор) опишете как ще изглеждат данните в клипборда.
3. **Четене на данни, разделени със запетая:** Създайте текстов низ, който представлява таблични данни, разделени със запетая (включително заглавен ред). Копирайте този низ в клипборда. Използвайте `pd.read_clipboard()` с подходящ параметър, за да прочетете данните в DataFrame. Изведете DataFrame-а.
4. **Копиране с персонализиран разделител:** Създайте DataFrame с две числови колони. Копирайте DataFrame-а в клипборда, като използвате знака "|" (вертикална черта) като разделител и включите индекса. Опишете как ще изглеждат данните в клипборда.

V. Работа с Excel файлове

Pandas осигурява отлична поддръжка за четене и запис на данни във файловия формат на Microsoft Excel (`.xlsx` и по-стария `.xls`). За тази цел се използват функциите `pd.read_excel()` и методите `df.to_excel()` и `series.to_excel()`. Важно е да се отбележи, че за работа с Excel файлове често се изисква инсталирането на допълнителни библиотеки, като `openpyxl` за `.xlsx` файлове и `xlrd` за по-стари `.xls` файлове. Ако тези библиотеки не са инсталирани, може да получите грешка. Можете да ги инсталирате с `pip install openpyxl xlrd`.

1. Четене от Excel (read_excel)

Функцията `pd.read_excel()` е основният инструмент за четене на данни от Excel файлове в Pandas DataFrame. Тя предлага множество параметри, които позволяват гъвкавост при работа с различни Excel файлове и структури.

а) Основни параметри на `pd.read_excel()`:

- **`io`:** Първият аргумент, който указва пътя до Excel файла (локален или URL адрес) или обект, подобен на файл. Може да бъде и байтов поток.
- **`sheet_name`:** Указва кой работен лист (sheet) да бъде прочетен. Може да бъде:
  - Низ (име на листа, например `'Sheet1'`).
  - Цяло число (индекс на листа, започвайки от 0).
  - Списък от низове или цели числа (за четене на множество листове). В този случай функцията ще върне речник, където ключовете са имената (или индексите) на листовете, а стойностите са съответните DataFrames.
  - `None` (по подразбиране): Чете всички листове и връща речник от DataFrames.
- **`header`:** Определя кои ред(ове) да се използват като заглавен ред (имена на колоните). Поведението е същото като при `pd.read_csv()`.
- **`names`:** Списък с имена на колони, които да се използват. Полезен, когато файлът няма заглавен ред или искате да презапишете съществуващите.
- **`index_col`:** Указва коя колона (по име или индекс) да се използва като индекс на DataFrame-а.
- **`dtype`:** Речник за задаване на типове данни на колоните.
- **`parse_dates`:** Булев флаг или списък от колони за парсване като дати.
- **`na_values`:** Списък или речник от стойности, които да се интерпретират като `NaN`.
- **`skiprows`:** Брой редове или списък от номера на редове за пропускане в началото на файла.
- **`nrows`:** Брой редове за четене от началото на файла.
- **`encoding`:** Задава кодирането (обикновено не е необходимо за Excel файлове, но може да е полезно в някои случаи).

б) Примери за четене от Excel:

Да предположим, че имаме Excel файл `data.xlsx` с два работни листа: 'Sheet1' и 'Sheet2'.

Sheet1 съдържа:

Име	Възраст	Град
Алиса	25	София
Борис	30	Пловдив

Sheet2 съдържа:

Ваня	27	Варна

Продукт	Цена
Ябълка	1.2
Банан	0.8
Портокал	1.5

Пример 1: Четене на един работен лист по име (Sheet1):

```
import pandas as pd

df_sheet1 = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print("DataFrame от Sheet1:\n", df_sheet1)
```

```
Изход:

DataFrame от Sheet1:
      Име  Възраст    Град
0  Алиса        25   София
1  Борис        30  Пловдив
2   Ваня        27    Варна
```

Пример 2: Четене на един работен лист по индекс (първият лист, индекс 0):

```
df_sheet_index0 = pd.read_excel('data.xlsx', sheet_name=0)
print("\nDataFrame от лист с индекс 0:\n", df_sheet_index0)
```

Изход (ще бъде същият като горния, ако 'Sheet1' е първият лист):

```
DataFrame от лист с индекс 0:
      Име  Възраст    Град
0  Алиса        25   София
1  Борис        30  Пловдив
2   Ваня        27    Варна
```

Пример 3: Четене на всички работни листове:

```
all_sheets = pd.read_excel('data.xlsx', sheet_name=None)
print("\nРечник от всички листове:\n", all_sheets)
print("\nDataFrame от Sheet2:\n", all_sheets['Sheet2'])
```

```
Изход:

Речник от всички листове:
 {'Sheet1':       Име    Възраст         Град
0  Алиса       25   София
1  Борис       30   Пловдив
2  Ваня        27   Варна, 'Sheet2':      Продукт  Цена
0  Ябълка     1.2
1  Банан      0.8
2  Портокал   1.5}

DataFrame от Sheet2:
```

```
     Продукт  Цена
0    Ябълка   1.2
1     Банан   0.8
2   Портокал  1.5
```

В този случай read_excel() връща речник, където ключовете са имената на листовете, а стойностите са съответните DataFrames.

Пример 4: Четене на конкретни колони и задаване на индекс:

Да предположим, че в 'Sheet1' искаме да прочетем само колоните 'Име' и 'Възраст', а колоната 'Име' да бъде индекс.

```
df_subset = pd.read_excel('data.xlsx', sheet_name='Sheet1', usecols=['Име', 'Възраст'], index_col='Име')
print("\nDataFrame с избрани колони и индекс:\n", df_subset)
```

```
Изход:

DataFrame с избрани колони и индекс:
        Възраст
Име
Алиса         25
Борис         30
Ваня          27
```

Параметърът usecols приема списък с имена на колони или списък с индекси на колони (започващи от 0).

Пример 5: Четене с пропускане на редове и задаване на различен заглавен ред:

Да предположим, че Excel файлът има няколко реда с информация преди същинските данни и заглавния ред е на третия ред (индекс 2).

Някаква информация		
Допълнителна информация		
Име	**Възраст**	**Град**
Алиса	25	София
Борис	30	Пловдив
Ваня	27	Варна

Можем да прочетем този файл така:

```
df_skipped = pd.read_excel('data_with_extra_rows.xlsx', sheet_name='Sheet1', skiprows=2, header=0)
print("\nDataFrame с пропуснати редове и зададен заглавен ред:\n", df_skipped)
```

В този случай skiprows=2 пропуска първите два реда, а header=0 указва, че редът с индекс 0 (след пропускането, което е оригиналният трети ред) съдържа заглавията на колоните.

Тези примери илюстрират гъвкавостта на функцията `pd.read_excel()` при четене на данни от различни Excel файлове и работни листове. В следващата част ще разгледаме как да записваме Pandas DataFrames в Excel файлове.

2. Запис в Excel (.to_excel)

Методът `to_excel()` е основният инструмент в Pandas за записване на DataFrame (или Series) обект в Excel файл (`.xlsx`). Той предлага различни параметри за контролиране на формата и съдържанието на записания файл.

а) Основни параметри на `to_excel()`:

- **`excel_writer`:** Път до целевия Excel файл (низ). Може да бъде и обект `ExcelWriter`. Използването на `ExcelWriter` е необходимо, когато искате да записвате в множество работни листове на един и същ файл.
- **`sheet_name`:** Низ, указващ името на работния лист, в който ще бъдат записани данните. По подразбиране е `'Sheet1'`.
- **`na_rep`:** Низ, който да се използва за представяне на липсващи стойности (`NaN`) в Excel файла. По подразбиране `NaN` се записват като празни клетки.
- **`float_format`:** Форматен низ за числа с плаваща запетая (например `'%.2f'` за записване с два знака след десетичната запетая).
- **`columns`:** Списък от имена на колони, които да бъдат записани. Ако е `None` (по подразбиране), всички колони се записват.
- **`header`:** Булев флаг или списък от низове. Ако е `True` (по подразбиране), имената на колоните се записват като първи ред. Ако е `False`, имената на колоните не се записват. Ако е списък от низове, той се използва като заглавен ред (презаписва имената на колоните).
- **`index`:** Булев флаг. Ако е `True` (по подразбиране), индексът на DataFrame-а се записва като първа колона (или като множество колони, ако е MultiIndex). Ако е `False`, индексът не се записва.
- **`index_label`:** Низ или списък от низове, които да се използват като заглавие на колоната (или колоните) на индекса, ако `index=True`. Ако е `None` и `index=True`, името на индекса ще се използва, ако има такова.
- **`startrow`:** Цяло число, указващо на кой ред (започвайки от 0) да започне записът на данните.
- **`startcol`:** Цяло число, указващо в коя колона (започвайки от 0) да започне записът на данните.
- **`encoding`:** Задава кодирането на файла (обикновено `'utf-8'` е добър избор).
- **`engine`:** Двигателят, който ще се използва за писане в Excel файла (например `'openpyxl'` или `'xlsxwriter'`). По подразбиране Pandas се опитва да използва наличните инсталирани двигатели.

б) Примери за запис в Excel:

1) Използвайки DataFrame `df_sheet1`, създаден в предходната секция:

```
import pandas as pd

# Създаване на примерен DataFrame (ако не е наличен от предходните примери)
data = {'Име': ['Алиса', 'Борис', 'Ваня'],
        'Възраст': [25, 30, 27],
        'Град': ['София', 'Пловдив', 'Варна']}
df_to_excel = pd.DataFrame(data)

# Запис на DataFrame в Excel файл 'output.xlsx' в лист 'Sheet1' (по
подразбиране)
df_to_excel.to_excel('output.xlsx')
```

```
print("DataFrame-ът беше записан в 'output.xlsx' (Sheet1, с индекс и
заглавен ред).")

# Запис на DataFrame без индекс в лист 'Някакъв лист'
df_to_excel.to_excel('output_no_index.xlsx', sheet_name='Някакъв лист',
index=False)
print("DataFrame-ът беше записан в 'output_no_index.xlsx' ('Някакъв лист',
без индекс).")

# Запис само на колоните 'Име' и 'Град'
df_to_excel[['Име', 'Град']].to_excel('output_subset.xlsx',
sheet_name='Подмножество', index=False)
print("Подмножество от колони беше записано в 'output_subset.xlsx'
('Подмножество').")

# Запис с указване на представяне на NaN стойности
import numpy as np
df_with_nan = pd.DataFrame({'A': [1, np.nan, 3], 'B': [4, 5, np.nan]})
df_with_nan.to_excel('output_nan.xlsx', sheet_name='NaN стойности',
na_rep='ЛИПСВА')
print("DataFrame с NaN стойности беше записан в 'output_nan.xlsx' ('NaN
стойности', NaN като 'ЛИПСВА').")

# Запис с форматиране на числа с плаваща запетая
df_float = pd.DataFrame({'Цена': [1.2345, 2.3456]})
df_float.to_excel('output_float.xlsx', sheet_name='Формат',
float_format='%.2f', index=False)
print("DataFrame с числа с плаваща запетая беше записан в
'output_float.xlsx' ('Формат', с 2 знака след десетичната запетая).")
```

2) Запис на множество DataFrames в един Excel файл (работа с различни sheets):

За да запишете няколко DataFrames в различни работни листове на един и същ Excel файл, трябва да използвате класа `pd.ExcelWriter`.

```
import pandas as pd

# Създаване на два примера на DataFrame
data1 = {'Име': ['Алиса', 'Борис'], 'Възраст': [25, 30]}
df1 = pd.DataFrame(data1)

data2 = {'Продукт': ['Ябълка', 'Банан'], 'Цена': [1.20, 0.80]}
df2 = pd.DataFrame(data2)

# Създаване на обект ExcelWriter
with pd.ExcelWriter('multiple_sheets.xlsx') as writer:
    # Запис на df1 в лист 'Потребители'
    df1.to_excel(writer, sheet_name='Потребители', index=False)

    # Запис на df2 в лист 'Продукти'
    df2.to_excel(writer, sheet_name='Продукти', index=False, startrow=2) #
Започва запис от ред 3
```

```
print("Два DataFrame-а бяха записани в 'multiple_sheets.xlsx' на различни листове.")
```

В този пример, pd.ExcelWriter('multiple_sheets.xlsx') създава обект, който представлява Excel файла, в който ще се записва. Използването на with гарантира, че файлът ще бъде правилно затворен след запис. След това, за всеки DataFrame, използваме метода to_excel() и подаваме writer като първи аргумент, указвайки и името на листа. Можем също да контролираме къде точно в листа да започне записът (чрез startrow)

Примерен сценарий:

Представете си, че имате Excel файл с име `Cash_Flow.xlsx`. Този файл съдържа информация за паричните потоци на различни обекти (например магазини, клонове) за две последователни години: 2024 и 2025. Данните за всяка година се намират в отделен лист на Excel файла, съответно с имената "2025" и "2024".

Структурата на данните във всеки лист е идентична и може да изглежда по следния начин:

Лист "2025" (пример):

Дата	Обект	Оборот (€)	Разходи (€)	Печалба (€)
2025-01-01	Магазин А	1500	800	700
2025-01-01	Магазин Б	2200	1200	1000
2025-01-02	Магазин А	1600	850	750
2025-01-02	Магазин Б	2100	1150	950
...	...	...	...	...

Лист "2024" (пример):

Дата	Обект	Оборот (€)	Разходи (€)	Печалба (€)
2024-01-01	Магазин А	1400	750	650
2024-01-01	Магазин Б	2000	1100	900
2024-01-02	Магазин А	1550	820	730
2024-01-02	Магазин Б	1950	1050	900
...	...	...	...	...

Целта на скрипта е да прочете тези два листа и да създаде нов Excel файл (`interleaved_result.xlsx`), в който данните за 2025 и 2024 година за всеки обект и дата да бъдат представени един до друг за лесно сравнение. Очакваният резултат в `interleaved_result.xlsx` би изглеждал така:

Дата_1	Обект_1	Оборот (€)_1	Разходи (€)_1	Печалба (€)_1	Дата_2	Обект_2	Оборот (€)_2	Разходи (€)_2	Печалба (€)_2
2025-01-01	Магазин А	1500	800	700	2024-01-01	Магазин А	1400	750	650
2025-01-01	Магазин Б	2200	1200	1000	2024-01-01	Магазин Б	2000	1100	900
2025-01-02	Магазин А	1600	850	750	2024-01-02	Магазин А	1550	820	730
2025-01-02	Магазин Б	2100	1150	950	2024-01-02	Магазин Б	1950	1050	900
...	...	...	...	...	...	...	...	...	...

```python
import pandas as pd

# Стъпка 1: Зареждане на двата листа (или два файла)
df1 = pd.read_excel("Cash_Flow.xlsx", sheet_name="2025")
df2 = pd.read_excel("Cash_Flow.xlsx", sheet_name="2024")

# Уверяваме се, че имат еднакъв брой колони и редове
assert df1.shape == df2.shape, "DataFrames трябва да имат еднаква форма"

# Стъпка 2: Създаване на нов DataFrame с редуващи се колони
interleaved_columns = []

for col1, col2 in zip(df1.columns, df2.columns):
    interleaved_columns.append(df1[col1])
    interleaved_columns.append(df2[col2])

# Генериране на нови имена на колоните (можеш да ги персонализираш)
new_column_names = []
for col1, col2 in zip(df1.columns, df2.columns):
    new_column_names.append(f"{col1}_1")
    new_column_names.append(f"{col2}_2")

# Обединяване на колоните в нов DataFrame
result_df = pd.concat(interleaved_columns, axis=1)
result_df.columns = new_column_names

# Стъпка 3: Записване в Excel
result_df.to_excel("interleaved_result.xlsx", index=False)
```

Разбор на скрипта:

1. **`import pandas as pd`**: Тази команда импортира библиотеката Pandas и я присвоява на псевдонима `pd`, което позволява по-лесното ѝ използване в кода.
2. **`df1 = pd.read_excel("Cash_Flow.xlsx", sheet_name="2025")`**: Тази линия използва функцията `read_excel()` на Pandas, за да прочете данните от листа с име "2025" в Excel файла "Cash_Flow.xlsx" и ги съхранява в DataFrame обект, наречен `df1`. DataFrame е двуизмерна таблична структура от данни, която е основна структура в Pandas.

3. **`df2 = pd.read_excel("Cash_Flow.xlsx", sheet_name="2024")`**: Подобно на горната команда, тази линия чете данните от листа с име "2024" от същия Excel файл и ги съхранява в DataFrame обект `df2`.
4. **`assert df1.shape == df2.shape, "DataFrames must have same shape"`**: Тази команда използва ключовата дума `assert` за проверка дали размерите (броят на редовете и колоните) на двата DataFrame обекта са еднакви. Ако размерите не съвпадат, ще бъде генерирана грешка `AssertionError` с предоставеното съобщение. Това е важна стъпка, за да се гарантира, че данните могат да бъдат коректно сравнени ред по ред.
5. **`interleaved_columns = []`**: Създава се празен списък, който ще съдържа колоните от двата DataFrame обекта, подредени последователно.
6. **`for col1, col2 in zip(df1.columns, df2.columns):`**: Този цикъл `for` итерира едновременно през имената на колоните на `df1` и `df2`, използвайки функцията `zip()`. Тъй като се предполага, че колоните са еднакви и в същия ред, `zip()` ще върне двойки от съответните имена на колони.
7. **`interleaved_columns.append(df1[col1])`**: Всяка колона от `df1` (достъпена чрез името си `col1`) се добавя към списъка `interleaved_columns`.
8. **`interleaved_columns.append(df2[col2])`**: След това, съответната колона от `df2` (достъпена чрез името си `col2`) също се добавя към списъка `interleaved_columns`. По този начин колоните от двата DataFrame обекта се редуват в списъка.
9. **`new_column_names = []`**: Създава се празен списък за новите имена на колоните в резултата.
10. **`for col1, col2 in zip(df1.columns, df2.columns):`**: Още един цикъл `for`, който отново итерира през имената на колоните на двата DataFrame обекта.
11. **`new_column_names.append(f"{col1}_1")`**: За всяка колона от `df1`, се създава ново име, като към оригиналното име се добавя суфикс `_1` (например "Дата" става "Дата_1").
12. **`new_column_names.append(f"{col2}_2")`**: Аналогично, за всяка колона от `df2`, се създава ново име със суфикс `_2` (например "Дата" става "Дата_2").
13. **`result_df = pd.concat(interleaved_columns, axis=1)`**: Функцията `concat()` на Pandas се използва за обединяване на колоните, съдържащи се в списъка `interleaved_columns`. Аргументът `axis=1` указва, че обединяването трябва да се извърши по колони (хоризонтално). Резултатът е нов DataFrame, наречен `result_df`.
14. **`result_df.columns = new_column_names`**: На новосъздадения DataFrame `result_df` се присвояват генерираните нови имена на колоните от списъка `new_column_names`.
15. **`result_df.to_excel("interleaved_result.xlsx", index=False)`**: Накрая, DataFrame `result_df` се записва в нов Excel файл с име "interleaved_result.xlsx". Аргументът `index=False` предотвратява записването на индекса на DataFrame като колона в Excel файла.

Как да използвате скрипта:

1. Създайте Excel файл с име `Cash_Flow.xlsx`.
2. Добавете два листа към този файл с имената "2025" и "2024".
3. Попълнете данните за оборотите и другите показатели във всеки лист, като се уверите, че структурата на колоните и броят на редовете са еднакви.
4. Запазете файла `Cash_Flow.xlsx` в същата директория, където ще изпълнявате Python скрипта.

След изпълнението на скрипта, в същата директория ще бъде създаден нов Excel файл `interleaved_result.xlsx`, който ще съдържа данните от двата листа, разположени един до друг за сравнение.

3) Пример за използване на параметъра `startcol` в `to_excel()`

Да предположим, че имаме DataFrame с данни за продажби и искаме да го запишем в Excel файл, но да оставим няколко празни колони в началото на листа.

```
import pandas as pd

# Създаване на примерен DataFrame за продажби
data = {'Продукт': ['Телевизор', 'Хладилник', 'Пералня'],
        'Цена': [550.00, 800.00, 450.00],
        'Продадени бройки': [15, 8, 12]}
df_sales = pd.DataFrame(data)

# Запис на DataFrame в Excel файл, започвайки от колона с индекс 2 (трета колона)
file_path = 'sales_data_with_offset.xlsx'
df_sales.to_excel(file_path, sheet_name='Продажби', index=False, startcol=2)

print(f"DataFrame-ът беше записан в '{file_path}', лист 'Продажби', започвайки от колона C.")
```

Когато изпълните този код и отворите файла sales_data_with_offset.xlsx, ще видите, че данните от DataFrame-а започват от колона C (индекс 2). Първите две колони (A и B, индекси 0 и 1) ще бъдат празни. Заглавният ред ('Продукт', 'Цена', 'Продадени бройки') ще започне от клетка C1, а самите данни ще започнат от C2.

Обяснение:

- Създаваме DataFrame `df_sales` с данни за продажби.
- Използваме метода `to_excel()` за запис във файл `'sales_data_with_offset.xlsx'`.
- `sheet_name='Продажби'` указва името на работния лист.
- `index=False` предотвратява записването на индекса на DataFrame-а.
- `startcol=2` е ключовият параметър. Той казва на Pandas да започне да записва данните от колона с индекс 2, което е третата колона в Excel (A е 0, B е 1, C е 2).

Този параметър е полезен, когато искате да вмъкнете допълнителна информация, лого или други данни в началото на Excel листа, преди да запишете основната таблица с данни. Можете да комбинирате `startrow` и `startcol`, за да определите точното местоположение, където да започне записът на вашия DataFrame в Excel файла.

3. Допълнително за работа с Excel

Разбира се, има няколко допълнителни аспекта и интересни случаи, които могат да бъдат полезни при четене и писане на Excel файлове с Pandas:

a) Допълнения при четене (`pd.read_excel()`):

- **Четене на MultiIndex от Excel:** Ако вашият Excel файл има няколко заглавни реда, които трябва да формират MultiIndex за колоните, можете да използвате параметъра `header` с подаване на списък от индекси на редовете (например `header=[0, 1]`). Pandas ще ги интерпретира като нива на MultiIndex.

```
# Пример: Excel файл с два заглавни реда
df_multi_header = pd.read_excel('multi_header.xlsx', header=[0, 1])
print(df_multi_header)
```

- **Четене на MultiIndex за редовете (индекс):** Ако първите няколко колони в Excel файла трябва да формират MultiIndex за редовете, можете да използвате параметъра `index_col` с подаване на списък от индекси или имена на колони (например `index_col=[0, 1]`).

```
# Пример: Excel файл с две колони за индекс
df_multi_index = pd.read_excel('multi_index.xlsx', index_col=['Година',
'Месец'])
print(df_multi_index)
```

- **Контрол на типа на празните стойности:** Параметърът `keep_default_na` (булев, по подразбиране `True`) указва дали да се използват стандартните NaN стойности (като 'NaN', '#N/A' и др.). Можете да го зададете на `False`, ако искате тези стойности да се третират като обикновени низове.

- **Персонализиране на конвертирането на дати:** Въпреки че `parse_dates` е мощен, понякога се налага по-специфично парсване на дати. Можете да използвате параметъра `date_parser`, на който да подадете функция, която да обработва стойностите като дати.

- **Четене на формула клетки (зависи от двигателя):** Някои енджини (като `openpyxl`) могат да четат формулите от клетките, а не само резултатните стойности. Поведението може да се контролира чрез допълнителни параметри, специфични за енджина (вижте документацията на съответната библиотека).

б) Допълнения при писане (`df.to_excel()`):

- **Записване на MultiIndex:** Когато DataFrame-ът има MultiIndex (както за редовете, така и за колоните), `to_excel()` автоматично ще го запише в няколко реда/колони в Excel. Можете да контролирате как се записват етикетите на MultiIndex с параметрите `index_label` (за индекс) и като структурата на самия MultiIndex (за колони).
- **Контрол на записа на заглавния ред при добавяне към съществуващ файл:** Когато използвате `ExcelWriter` и записвате в съществуващ файл (или добавяте нови листове), може да искате да контролирате дали заглавният ред се записва всеки път. Това може да се управлява ръчно, като проверите дали листът е празен или като използвате допълнителна логика.
- **Използване на различни енджини:** Както споменахме, Pandas поддържа различни енджини за работа с Excel (`openpyxl`, `xlsxwriter`, `xlwt` - само за `.xls`). Всеки енджин има своите предимства и недостатъци по отношение на производителност, поддържани функционалности (като форматиране) и зависимости. Можете да изберете конкретен енджин с параметъра `engine`. Например, `xlsxwriter` предлага богати възможности за форматиране.

```
# Пример за запис с xlsxwriter и без индекс
df_to_excel.to_excel('output_xlsxwriter.xlsx', sheet_name='Данни',
index=False, engine='xlsxwriter')
```

- **Форматиране на Excel файла (с енджини като `xlsxwriter`):** Енджини като `xlsxwriter` позволяват много фин контрол върху форматирането на Excel файла (например стилове на клетки, формати на

числа, диаграми). За да използвате тези възможности, трябва да работите директно с обекта `ExcelWriter` и да използвате API-то на съответния енджин.

```
import pandas as pd

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

writer = pd.ExcelWriter('formatted_output.xlsx', engine='xlsxwriter')
df.to_excel(writer, sheet_name='Sheet1', index=False)

# Получаване на workbook и worksheet обекти на xlsxwriter
workbook = writer.book
worksheet = writer.sheets['Sheet1']

# Създаване на формат
bold_format = workbook.add_format({'bold': True})

# Прилагане на формата към заглавния ред
for col_num, value in enumerate(df.columns.values):
    worksheet.write(0, col_num, value, bold_format)

writer.close()
print("DataFrame-ът беше записан с форматиране.")
```

в) Изключения и интересни случаи:

- **Проблеми с кодирането (рядко за `.xlsx`):** При по-стари `.xls` файлове или при специфични ситуации може да възникнат проблеми с кодирането. Параметърът `encoding` може да помогне в такива случаи.
- **Големи Excel файлове:** За много големи Excel файлове може да възникнат проблеми с паметта, особено при четене. В такива случаи може да се наложи обработка на части от файла (ако е възможно) или използване на други формати, които са по-ефективни за големи данни (например CSV с `chunksize` или columnar формати като Parquet).
- **Специални символи в данните или имената на листовете:** Уверете се, че имената на листовете и данните не съдържат символи, които могат да причинят проблеми при записване или четене (въпреки че съвременните библиотеки обикновено се справят добре с това).

Разбирането на тези по-напреднали аспекти и потенциални проблеми ще ви направи още по-уверени и ефективни при работата с Excel файлове в Pandas.

Казус 1: Анализ на месечен финансов отчет

Представете си, че всеки месец получавате финансов отчет в Excel формат (`financial_report_october.xlsx`). Отчетът съдържа няколко листа, включително "Приходи", "Разходи" и "Баланс". Искате да прочетете данните от листа "Приходи" и да изчислите общите приходи за месеца.

Решение:

Използвайте `pd.read_excel()` с параметъра `sheet_name` за указване на конкретния лист.

```
import pandas as pd

file_path = 'financial_report_october.xlsx'
sheet_to_read = 'Приходи'

df_revenues = pd.read_excel(file_path, sheet_name=sheet_to_read)

# Да предположим, че колоната с приходите се нарича 'Сума'
total_revenues = df_revenues['Сума'].sum()

print(f"Общи приходи за октомври: {total_revenues:.2f}")
print("\nДанни за приходите:\n", df_revenues)
```

Казус 2: Комбиниране на данни от няколко Excel файла

Имате няколко Excel файла (например, отчети за продажби по региони: `sales_region_1.xlsx`, `sales_region_2.xlsx`), всеки с еднакъв формат на данните в първия лист. Искате да прочетете данните от всички файлове и да ги обедините в един DataFrame за общ анализ.

Решение:

Използвайте цикъл за четене на всеки файл и `pd.concat()` за обединяване на DataFrame-ите.

```
import pandas as pd
import glob

file_pattern = 'sales_region_*.xlsx'
all_files = glob.glob(file_pattern)
all_data = []

for file in all_files:
    df = pd.read_excel(file) # Чете първия лист по подразбиране
    all_data.append(df)

combined_sales_data = pd.concat(all_data, ignore_index=True)

print("Комбинирани данни за продажби от всички региони:\n",
combined_sales_data.head())
# Сега combined_sales_data съдържа всички данни.
```

*Библиотека glob в Python

Библиотеката `glob` е част от стандартната библиотека на Python (което означава, че не е необходимо да я инсталирате допълнително). Тя предоставя функция за намиране на всички пътища към файлове, които съответстват на определен шаблон (pattern) в дадена директория.

Основната функция, която използвахме, е `glob.glob(pathname)`.

- **`pathname`:** Това е низ, представляващ шаблон за търсене на файлове. Той може да съдържа стандартни символи за търсене, които се наричат уайлдкард символи (wildcards):
  - `*`: Съвпада с нула или повече символа. Например, `'*.txt'` ще намери всички файлове, завършващи на `.txt`. В нашия случай `'sales_region_*.xlsx'` ще намери всички файлове, които започват с `'sales_region_'` и завършват на `.xlsx` (като `sales_region_1.xlsx`, `sales_region_north.xlsx` и т.н.).
  - `?`: Съвпада с точно един произволен символ. Например, `'file?.txt'` ще намери `file1.txt`, `filea.txt`, но не и `file12.txt`.
  - `[]`: Използва се за задаване на набор или диапазон от символи, които могат да съвпаднат на дадена позиция. Например:
    - `'[0-9].txt'` ще намери `1.txt`, `9.txt`, но не и `a.txt`.
    - `'[abc].txt'` ще намери `a.txt`, `b.txt`, `c.txt`.
  - `[!...]` или `[^...]`: Съвпада с всеки символ, който НЕ е в зададения набор или диапазон. Например, `'[!0-9].txt'` ще намери `a.txt`, но не и `1.txt`.
- **Връщана стойност:** Функцията `glob.glob(pathname)` връща списък от всички пътища към файлове (включително директории, ако съвпадат с шаблона), които са били намерени в съответствие с подадения шаблон. Пътищата обикновено са относителни към текущата работна директория, освен ако не е подаден абсолютен път в шаблона.

Предимства на използването на `glob` в нашия случай:

- **Динамично откриване на файлове:** Вместо да изброяваме ръчно имената на всички Excel файлове, `glob` автоматично намира всички файлове, които следват определен модел. Това е особено полезно, когато броят на файловете може да варира или не е предварително известен.
- **Лесен начин за филтриране на файлове:** Използването на уайлдкард символи позволява лесно да селектират само файловете, които ни интересуват, въз основа на техните имена.
- **Кратък и четим код:** `glob` предлага елегантен начин за намиране на множество файлове с няколко реда код.

В нашия казус `'sales_region_*.xlsx'` гарантира, че ще бъдат обработени всички Excel файлове, чиито имена започват с `"sales_region_"` и завършват с `".xlsx"`, без да е необходимо да знаем точните им имена предварително. След това цикълът просто итерира през списъка от намерени файлови пътища и чете всеки файл с `pd.read_excel()`.

Казус 3: Експортиране на резултати от анализ в Excel с няколко листа

След като сте извършили анализ на данни в Pandas DataFrame, искате да експортирате резултатите в Excel файл (`analysis_results.xlsx`) с няколко листа, където всеки лист съдържа различни аспекти на анализа (например, "Обобщени данни", "Детайлни резултати", "Графики" - като таблични данни).

Решение:

Използвайте `pd.ExcelWriter` за записване на няколко DataFrame-а в различни листове на един Excel файл.

```
import pandas as pd
```

```
# Да предположим, че имаме два DataFrame-а с резултати
summary_df = pd.DataFrame({'Показател': ['Средна стойност', 'Максимална
стойност'], 'Стойност': [15.2, 25.8]})
details_df = pd.DataFrame({'ID': [1, 2, 3], 'Стойност': [10.1, 12.5, 9.8],
'Категория': ['A', 'B', 'A']})

file_path = 'analysis_results.xlsx'

with pd.ExcelWriter(file_path) as writer:
    summary_df.to_excel(writer, sheet_name='Обобщени данни', index=False)
    details_df.to_excel(writer, sheet_name='Детайлни резултати',
index=False)

print(f"Резултатите от анализа бяха записани в '{file_path}' с два листа.")
```

Казус 4*: Анализ на данни за продажби от регионални офиси

Ситуация:

Вашата компания има няколко регионални офиса и всеки офис изпраща месечни отчети за продажбите си в отделен Excel файл. Всеки Excel файл има една и съща структура, но съдържа данни за различни продукти и клиенти за съответния регион. Освен това, всеки файл съдържа и обобщена информация за продажбите за месеца в отделен sheet, наречен "Summary".

Структура на всеки Excel файл (например, 'sales_europe_january.xlsx'):

- **Sheet "Sales Data":** Съдържа подробни данни за продажбите с колони: 'ProductID', 'ProductName', 'CustomerID', 'CustomerName', 'Quantity', 'UnitPrice', 'TotalAmount'.
- **Sheet "Summary":** Съдържа обобщена информация за месеца: 'Total Sales', 'Number of Orders', 'Average Order Value'.

Може да използвате следния генратор:

```
import pandas as pd
import numpy as np
from pathlib import Path

# Папка за съхранение на файловете
output_dir = Path(r"C:/Users/DELL/Documents ")
output_dir.mkdir(parents=True, exist_ok=True)

# Параметри на симулацията
regions = ['europe', 'asia', 'america']
months = ['january', 'february']
num_customers = 10
num_products = 5

# Генериране на данни за продукти и клиенти
product_ids = [f"P{i:03}" for i in range(1, num_products + 1)]
```

```
product_names = [f"Product {i}" for i in range(1, num_products + 1)]
customer_ids = [f"C{i:03}" for i in range(1, num_customers + 1)]
customer_names = [f"Customer {i}" for i in range(1, num_customers + 1)]


# Генератор на данни за един файл
def generate_excel_file(region, month):
  np.random.seed(hash(region + month) % 2 ** 32)
  rows = []
  for _ in range(50):  # 50 продажби
    pid = np.random.choice(product_ids)
    pname = product_names[product_ids.index(pid)]
    cid = np.random.choice(customer_ids)
    cname = customer_names[customer_ids.index(cid)]
    quantity = np.random.randint(1, 20)
    unit_price = np.random.uniform(10.0, 100.0)
    total = round(quantity * unit_price, 2)
    rows.append([pid, pname, cid, cname, quantity, round(unit_price, 2),
total])

  sales_df = pd.DataFrame(rows, columns=[
    'ProductID', 'ProductName', 'CustomerID', 'CustomerName',
    'Quantity', 'UnitPrice', 'TotalAmount'
  ])

  # Обобщена информация
  total_sales = sales_df['TotalAmount'].sum()
  num_orders = sales_df['CustomerID'].nunique()
  avg_order_value = total_sales / num_orders if num_orders else 0
  summary_df = pd.DataFrame([{
    'Total Sales': round(total_sales, 2),
    'Number of Orders': num_orders,
    'Average Order Value': round(avg_order_value, 2)
  }])

  # Записване в Excel файл
  file_name = f"sales_{region}_{month}.xlsx"
  file_path = output_dir / file_name
  with pd.ExcelWriter(file_path) as writer:
    sales_df.to_excel(writer, sheet_name="Sales Data", index=False)
    summary_df.to_excel(writer, sheet_name="Summary", index=False)

  return file_path


# Генериране на файлове за всички региони и месеци
generated_files = [generate_excel_file(region, month) for region in regions
for month in months]

generated_files[:5]  # показваме първите няколко пътя към файлове като
потвърждение
```

Вие получавате множество такива Excel файлове за различни региони и месеци и трябва да извършите следния анализ:

1. Прочетете данните за продажбите от sheet "Sales Data" на всички Excel файлове.
2. Прочетете обобщената информация от sheet "Summary" на всички Excel файлове.
3. Обединете данните за продажбите от всички файлове в един голям DataFrame. Добавете колони 'Region' и 'Month' към този DataFrame, извлечени от името на файла.
4. Обединете обобщените данни от всички файлове в друг DataFrame, също с колони 'Region' и 'Month'.
5. Изчислете общите продажби, средното количество на поръчка и средната стойност на поръчката за всеки регион и месец, използвайки обединените данни.
6. Създайте нов Excel файл ('combined_sales_analysis.xlsx') с два sheets:
    - **Sheet "Detailed Sales":** Съдържа обединените подробни данни за продажбите.
    - **Sheet "Regional Summary":** Съдържа обобщените резултати, изчислени в стъпка 5.

Решение:

```
import pandas as pd
import os

# Директория, съдържаща Excel файловете
data_directory = 'sales_reports'  # Уверете се, че тази директория
съществува и съдържа вашите файлове

all_sales_data = []
all_summary_data = []

# Обхождане на всички файлове в директорията
for filename in os.listdir(data_directory):
    if filename.endswith('.xlsx'):
        file_path = os.path.join(data_directory, filename)
        region = filename.split('_')[1] if len(filename.split('_')) > 1 else
'Unknown'
        month_str = filename.split('_')[-1].replace('.xlsx', '') if
len(filename.split('_')) > 1 else 'Unknown'
        # Опитайте се да извлечете месеца, може да се наложи по-сложно
парсване
        month = month_str.split('.')[0] if '.' in month_str else month_str

        try:
            # 1. Четене на данните за продажбите
            sales_df = pd.read_excel(file_path, sheet_name='Sales Data')
            sales_df['Region'] = region
            sales_df['Month'] = month
            all_sales_data.append(sales_df)

            # 2. Четене на обобщените данни
            summary_df = pd.read_excel(file_path, sheet_name='Summary')
            summary_df['Region'] = region
            summary_df['Month'] = month
            all_summary_data.append(summary_df)
```

```
        except FileNotFoundError:
            print(f"Файлът '{filename}' не беше намерен.")
        except KeyError as e:
            print(f"Sheet '{e}' не беше намерен във файла '{filename}'.")
        except Exception as e:
            print(f"Възникна грешка при четене на файла '{filename}': {e}")

# 3. Обединяване на данните за продажбите
combined_sales_df = pd.concat(all_sales_data, ignore_index=True)

# 4. Обединяване на обобщените данни
combined_summary_df = pd.concat(all_summary_data, ignore_index=True)

# 5. Изчисляване на обобщени резултати по регион и месец
regional_summary = combined_sales_df.groupby(['Region', 'Month']).agg(
    TotalSales=('TotalAmount', 'sum'),
    TotalQuantity=('Quantity', 'sum'),
    NumberOfOrders=('CustomerID', 'nunique')
).reset_index()

regional_summary['AverageOrderValue'] = regional_summary['TotalSales'] /
regional_summary['NumberOfOrders']

# 6. Записване в нов Excel файл с множество sheets
output_file = 'combined_sales_analysis.xlsx'
with pd.ExcelWriter(output_file) as writer:
    combined_sales_df.to_excel(writer, sheet_name='Detailed Sales',
index=False)
    regional_summary.to_excel(writer, sheet_name='Regional Summary',
index=False)

print(f"Анализът завърши и резултатите са записани в '{output_file}'.")
```

Разбор на сложността:

- **Работа с множество файлове:** Скриптът трябва да обработи множество Excel файлове от директория.
- **Четене от различни sheets:** Данните се четат от различни sheet names ('Sales Data' и 'Summary') във всеки файл.
- **Извличане на информация от името на файла:** Допълнителна информация (регион и месец) се извлича динамично от името на всеки файл.
- **Обработка на грешки:** Включено е управление на потенциални грешки при четене на файлове или sheets.
- **Обединяване на данни:** Данните от множество файлове се обединяват в два отделни DataFrame-а.
- **Групиране и агрегиране:** Извършва се сложно групиране и агрегиране на данни за изчисляване на обобщени показатели.
- **Запис в множество sheets:** Резултатите се записват в нов Excel файл с два отделни sheets ('Detailed Sales' и 'Regional Summary').

Въпроси:

1. Кои са основните функции в Pandas за четене и запис на Excel файлове?
2. Как може да прочетете данни от конкретен лист в Excel файл, ако файлът съдържа няколко листа?
3. Как Pandas идентифицира заглавния ред в Excel файл по подразбиране? Как можете да укажете, че Excel файлът няма заглавен ред или че заглавният ред е на различен ред?
4. Как може да прочетете само определени колони от Excel файл?
5. Как се обработват липсващите стойности при четене на Excel файл с Pandas?
6. Как може да укажете коя колона от Excel файла да се използва като индекс на DataFrame-а при четене?
7. Какво представлява обектът `ExcelWriter` в Pandas и защо се използва при записване на няколко DataFrame-а в един Excel файл?
8. Как може да запишете DataFrame в конкретен лист на Excel файл? Какво се случва, ако лист с това име вече съществува?
9. Как може да запишете няколко различни DataFrame-а в различни листове на един Excel файл? Дайте примерен код.
10. Как можете да контролирате дали индексът на DataFrame-а да бъде записан в Excel файла?
11. Какви са някои от често използваните параметри на `df.to_excel()`?
12. В какъв формат се записват датите и часовете по подразбиране при експортиране на DataFrame в Excel?
13. Какви библиотеки трябва да бъдат инсталирани, за да може Pandas да работи с Excel файлове (както `.xlsx`, така и `.xls`)?
14. Как може да прочетете данни от Excel файл, който е защитен с парола? (Забележка: Pandas може да не поддържа директно тази функционалност, но какъв би бил евентуалният подход?)

Задачи:

1. **Четене от конкретен лист:** Създайте Excel файл `data.xlsx` с два листа: "Лист1" и "Лист2". Въведете произволни данни в двата листа (например, по 3 колони и 5 реда). Прочетете данните само от "Лист2" в Pandas DataFrame и изведете го.
2. **Четене без заглавен ред и задаване на имена:** Създайте Excel файл `no_header.xlsx` с 3 колони и 5 реда с произволни данни, без ред със заглавия. Прочетете този файл в Pandas DataFrame и задайте имената на колоните като 'Колона A', 'Колона B', 'Колона C'. Изведете DataFrame-а.
3. **Запис на няколко DataFrame-а:** Създайте два DataFrame-а с произволни данни. Запишете ги в един Excel файл `multiple_sheets.xlsx` на два отделни листа, съответно "Обобщени данни" и "Детайли". Не включвайте индексите при записването.
4. **Четене на определени колони и задаване на индекс:** Създайте Excel файл `data_with_id.xlsx` с колони 'ID', 'Име', 'Възраст', 'Град'. Прочетете файла, като извлечете само колоните 'Име' и 'Възраст', и зададете колоната 'Име' като индекс на DataFrame-а. Изведете резултата.
5. **Запис с контролиране на индекса:** Създайте DataFrame с произволни данни и индекс, който не е поредица от числа (например, букви). Запишете DataFrame-а в Excel файл `indexed_data.xlsx` веднъж с включен индекс и веднъж без индекс (в два различни файла или на два различни листа). Отворете Excel файла и сравнете резултатите.

VI. Работа с JSON (read_json, to_json)

1. JSON (JavaScript Object Notation)

JSON е лек формат за обмен на данни, който е лесен за четене и писане от хора и лесен за парсване и генериране от машини. Той се базира на подмножество на JavaScript синтаксиса. Данните в JSON се представят като ключ-стойност двойки, където ключовете са низове, а стойностите могат да бъдат примитивни типове (низ, число, булев, null) или други JSON обекти или масиви.

Тази функция чете JSON данни от низ, файл или URL адрес и ги преобразува в Pandas Series или DataFrame.

```
pd.read_json(path_or_buf=None, orient=None, typ='frame', dtype=None,
convert_dates=True, numpy=False, precise_float=False, date_unit=None,
encoding=None, lines=False, chunksize=None, compression='infer', nrows=None,
storage_options=None)
```

Някои от основните параметри включват:

- **`path_or_buf`:** Низ, съдържащ път до JSON файл, URL адрес или JSON string. Може да бъде и обект, подобен на файл.
- **`orient`:** Указва очаквания формат на JSON структурата. Възможни стойности са:
  - `'split'`: JSON е във формат `{index: [index], columns: [columns], data: [values]}`.
  - `'records'`: Списък от JSON обекти, където всеки обект представлява ред с ключ-стойност двойки за всяка колона.
  - `'index'`: JSON е във формат `{index: {column: value}}`.
  - `'columns'`: JSON е във формат `{column: {index: value}}`.
  - `'values'`: Просто 2D масив от стойности.
  - `'table'`: JSON е във формат, описан от табличната схема (с `'schema'` и `'data'`). По подразбиране Pandas се опитва да определи ориентацията автоматично.
- **`typ`:** Указва типа на върнатия обект (`'series'` или `'frame'`). По подразбиране е `'frame'` (DataFrame). Използва се само когато ориентацията позволява създаването и на Series (например при `'index'` или `'values'` с едномерни данни).
- **`dtype`:** Задава типа на данните за колоните (подобно на `read_csv`).
- **`convert_dates`:** Булев флаг. Ако е `True` (по подразбиране), Pandas ще се опита да парсне низове, които изглеждат като дати, в datetime обекти.
- **`encoding`:** Задава кодирането на файла.
- **`lines`:** Булев флаг. Ако е `True`, чете файла като JSON обект на всеки ред. Полезно за големи JSON файлове, където всеки ред е независим JSON обект.

2. Примери за четене на JSON:

а) Да предположим, че имаме JSON файл `data.json` със следното съдържание:

```
{
  "Име": ["Алиса", "Борис", "Ваня"],
```

```
  "Възраст": [25, 30, 27],
  "Град": ["София", "Пловдив", "Варна"]
}
```

```
import pandas as pd

# Четене на JSON файл (ориентация 'columns' по подразбиране)
df_from_json = pd.read_json('data.json')
print("DataFrame от data.json:\n", df_from_json)
```

РЕЗУЛТАТ:

```
[
  {"Име": "Алиса", "Възраст": 25, "Град": "София"},
  {"Име": "Борис", "Възраст": 30, "Град": "Пловдив"},
  {"Име": "Ваня", "Възраст": 27, "Град": "Варна"}
]
```

б) Ако JSON файлът е в ориентация 'records':

```
df_from_json_records = pd.read_json('data_records.json', orient='records')
print("\nDataFrame от data_records.json:\n", df_from_json_records)
```

РЕЗУЛТАТ:

```
{
  "0": {"Име": "Алиса", "Възраст": 25, "Град": "София"},
  "1": {"Име": "Борис", "Възраст": 30, "Град": "Пловдив"},
  "2": {"Име": "Ваня", "Възраст": 27, "Град": "Варна"}
}
```

в) Ако JSON файлът е в ориентация 'index':

Използвайки DataFrame `df_from_json`, създаден по-рано:

```
df_from_json_index = pd.read_json('data_index.json', orient='index')
print("\nDataFrame от data_index.json:\n", df_from_json_index)
```

3. Примери за запис в JSON, df.to_json:

Този метод конвертира DataFrame (или Series) в JSON string.

```
df.to_json(path_or_buf=None, orient=None, date_format=None,
double_precision=10, force_ascii=True, date_unit='ms', default_handler=None,
```

```
lines=False, compression='infer', index=True, indent=None,
storage_options=None)
```

Някои от основните параметри включват:

- **`path_or_buf`:** Път до файла, в който да се запише JSON string. Ако е `None`, връща JSON string.
- **`orient`:** Определя формата на JSON изхода. Същите стойности като при `read_json()` са възможни. По подразбиране е `'columns'` за DataFrame и `'index'` за Series.
- **`date_format`:** Как да се записват датите (`'epoch'` за UNIX timestamp или `'iso'` за ISO 8601 формат).
- **`indent`:** Брой интервали за отстъп при форматиране на JSON (за по-лесно четене от човек). Ако е `None`, JSON е на един ред.
- **`index`:** Булев флаг, указващ дали индексът да бъде включен в JSON изхода.
- **`lines`:** Булев флаг. Ако е `True`, записва всеки ред като JSON обект на нов ред (JSON Lines формат).

```
import pandas as pd

# Запис на DataFrame в JSON файл (ориентация 'columns' по подразбиране)
df_from_json.to_json('output.json')
print("DataFrame-ът беше записан в 'output.json' (ориентация 'columns').")

# Запис на DataFrame в JSON файл с ориентация 'records' и отстъп
df_from_json.to_json('output_records.json', orient='records', indent=4)
print("DataFrame-ът беше записан в 'output_records.json' (ориентация
'records' с отстъп).")

# Запис на DataFrame в JSON string с ориентация 'index' и включен индекс
json_string_index = df_from_json.to_json(orient='index', indent=2,
index=True)
print("\nJSON string (ориентация 'index'):\n", json_string_index)

# Запис на DataFrame в JSON Lines формат
df_from_json.to_json('output_lines.json', orient='records', lines=True)
print("DataFrame-ът беше записан в 'output_lines.json' (JSON Lines
формат).")
```

Работата с JSON в Pandas е доста гъвкава и позволява лесно преобразуване между таблични данни и JSON структури, което е изключително полезно при взаимодействие с уеб базирани системи.

4. Допълнително за работата с JSON

- **Вложени JSON структури:** `pd.read_json()` може да чете и обработва JSON файлове с вложени обекти или масиви. Как точно тези вложени структури се преобразуват в DataFrame зависи от ориентацията и структурата на JSON-а. Понякога може да се наложи допълнителна обработка (например с `json_normalize` от библиотеката `pandas`) за сплескване на вложените данни в табличен вид.

```
import pandas as pd
import json
```

```
nested_json_str = '''
{
    "потребители": [
        {"id": 1, "име": "Алиса", "детайли": {"възраст": 25, "град":
"София"}},
        {"id": 2, "име": "Борис", "детайли": {"възраст": 30, "град":
"Пловдив"}}
    ]
}
'''
nested_data = json.loads(nested_json_str)
df_nested = pd.json_normalize(nested_data, 'потребители',
record_prefix='потребител_', meta=['потребители'])
print(df_nested)
```

- **Различни типове данни в JSON:** Pandas се опитва да интерпретира типовете данни от JSON автоматично. Въпреки това, може да се наложи изрично задаване на `dtype` параметъра в `pd.read_json()`, ако искате различно поведение.

- **Обработка на JSON с множество нива:** За много сложни JSON структури с дълбоко вложеност, може да е по-подходящо първо да парснете JSON-а с вградения модул `json` на Python и след това да създадете DataFrame ръчно или с помощта на `pd.DataFrame.from_dict()` или `pd.DataFrame.from_records()`.

- **Сериализация на специфични Python обекти:** Когато записвате DataFrame в JSON с `to_json()`, Pandas се опитва да сериализира стандартните Python типове данни. За по-сложни обекти или потребителски типове може да се наложи да предоставите `default_handler` функция, която да ги преобразува в JSON-сериализируеми формати.

5. Изключения:

- **GeoJSON:** Въпреки че GeoJSON е базиран на JSON, той има специфична структура за представяне на географски данни. Работата с GeoJSON често изисква специализирани библиотеки като `geopandas`, които надграждат Pandas и добавят поддръжка за географски обекти.

- **JSON-LD:** JSON for Linked Data (JSON-LD) е формат за представяне на свързани данни с помощта на JSON. Той добавя семантични анотации към JSON данните. Pandas не предлага вградена поддръжка за JSON-LD и обикновено се използват други библиотеки за работа с него.

- **Други специализирани JSON базирани формати:** Съществуват и други формати, базирани на JSON, но със специфични конвенции и структури за определени домейни (например Mapbox Styles, Vega-Lite спецификации). Pandas може да чете тези като обикновен JSON, но интерпретацията и работата с данните често изискват познаване на съответния формат и може да се нуждаят от допълнителни инструменти.

Важно е да се има предвид, че въпреки че Pandas предоставя мощни инструменти за работа с JSON, сложността и структурата на самия JSON файл могат да наложат допълнителни стъпки за обработка и трансформация на данните, за да бъдат те в подходящ табличен вид за анализ.

Казус 1: Анализ на данни от API отговор

Представете си, че правите заявка към уеб API и получавате отговор във формат JSON. JSON отговорът съдържа списък от обекти, където всеки обект представлява информация за продукт (име, цена, наличност). Искате да преобразувате тези данни в Pandas DataFrame за по-лесен анализ.

Примерен JSON отговор:

```
[
  {"име": "Лаптоп", "цена": 1200.50, "наличност": 50},
  {"име": "Мишка", "цена": 25.99, "наличност": 100},
  {"име": "Клавиатура", "цена": 79.90, "наличност": 75}
]
```

Решение:

Използвайте `pd.read_json()` за директно прочитане на JSON стринга (или файл) в DataFrame.

```
import pandas as pd
import json

json_data = '''
[
  {"име": "Лаптоп", "цена": 1200.50, "наличност": 50},
  {"име": "Мишка", "цена": 25.99, "наличност": 100},
  {"име": "Клавиатура", "цена": 79.90, "наличност": 75}
]
'''

df_products = pd.read_json(json_data)

print("DataFrame от JSON данни:\n", df_products)

# Сега можете да анализирате данните, например да намерите средната цена:
average_price = df_products['цена'].mean()
print(f"\nСредна цена на продуктите: {average_price:.2f}")
```

Ако JSON данните са във файл `products.json`, просто подайте пътя към файла на `pd.read_json('products.json')`.

Казус 2: Записване на резултати от анализ за уеб приложение

След като сте извършили анализ на данни в Pandas DataFrame (например, обобщени данни за продажби по категории), искате да запишете резултатите във формат JSON, който да бъде използван от уеб приложение за визуализация или други цели.

Решение:

Използвайте метода `df.to_json()` за експортиране на DataFrame-а в JSON формат.

```
import pandas as pd

# Да предположим, че имаме DataFrame с обобщени данни за продажби
sales_summary = pd.DataFrame({
    'категория': ['Електроника', 'Книги', 'Дрехи'],
    'общо_продажби': [15000.00, 5000.50, 8000.75]
})

# Запис в JSON файл
sales_summary.to_json('sales_summary.json', orient='records', indent=4)
print("Обобщените данни за продажби бяха записани в 'sales_summary.json'.")

# Запис като JSON string
json_output = sales_summary.to_json(orient='records')
print("\nJSON string:\n", json_output)
```

Параметърът `orient='records'` форматира JSON като списък от обекти (както в първия казус), което често е удобно за консумация от уеб приложения. `indent=4` добавя отстъпи за по-лесно четене на JSON файла.

Казус 3: Работа с JSON файлове с различна структура

Понякога JSON файловете могат да имат по-сложна структура, например вложени обекти или масиви. `pd.read_json()` може да обработва различни ориентации на JSON.

Примерен JSON файл (`complex_data.json`):

```
{
  "потребители": [
    {
      "id": 1,
      "име": "Алиса",
      "адрес": {"град": "София", "улица": "бул. България 1"}
    },
    {
      "id": 2,
      "име": "Борис",
      "адрес": {"град": "Пловдив", "улица": "ул. Тракия 10"}
    }
  ]
}
```

Решение:

В този случай може да се наложи да използвате параметъра `orient='index'` или да нормализирате JSON структурата с `pd.json_normalize()`, както беше споменато по-рано.

```
import pandas as pd
import json
```

```
with open('complex_data.json', 'r', encoding='utf-8') as f:
    complex_json_data = json.load(f)

# Използване на json_normalize за сплескване на вложената структура
df_complex = pd.json_normalize(complex_json_data, 'потребители')
print("DataFrame от сложен JSON (с json_normalize):\n", df_complex)

# Можете също да достъпите вложените данни с record_path и meta
df_complex_nested = pd.json_normalize(complex_json_data,
record_path='потребители',
                                      meta=['потребители'])
print("\nDataFrame от сложен JSON (с record_path и meta):\n",
df_complex_nested)
```

Казус 4*: Анализ на данни за онлайн поръчки с детайлна информация за артикулите

Ситуация:

Получавате данни за онлайн поръчки от API в JSON формат. Всяка поръчка съдържа основна информация (ID на поръчката, дата на създаване, обща сума), както и детайлен списък от закупени артикули. Всеки артикул в списъка има ID, име, количество и единична цена. Освен това, всяка поръчка съдържа информация за клиента, включваща вложен адрес с данни за улица, град и пощенски код.

Примерна структура на един JSON обект (една поръчка):

```
{
  "order_id": "ORD-2025-0504-1001",
  "created_at": "2025-05-04T09:30:00Z",
  "total_amount": 125.50,
  "customer": {
    "customer_id": "CUST-001",
    "name": "John Doe",
    "address": {
      "street": "123 Main St",
      "city": "Plovdiv",
      "zipcode": "4000"
    }
  },
  "items": [
    {
      "item_id": "ITEM-001",
      "name": "Laptop",
      "quantity": 1,
      "unit_price": 1000.00
    },
    {
      "item_id": "ITEM-002",
      "name": "Mouse",
```

```
      "quantity": 2,
      "unit_price": 12.75
    }
  ],
  "payment_method": "Credit Card",
  "shipping_address": {
    "street": "456 Oak Ave",
    "city": "Sofia",
    "zipcode": "1000"
  }
}
```

Получавате списък от такива JSON обекти и трябва да извършите следния анализ:

1. Прочетете JSON данните в Pandas DataFrame. Тъй като има вложени структури и масиви, ще се наложи да ги нормализирате.
2. Създайте отделен DataFrame, съдържащ информация за всеки закупен артикул, като разгънете масива "items". Този DataFrame трябва да съдържа ID на поръчката, ID на артикула, име на артикула, количество и единична цена.
3. Създайте друг DataFrame с информация за адресите на клиентите (улица, град, пощенски код), свързан с ID на поръчката и ID на клиента.
4. Изчислете общата стойност на всеки артикул в рамките на всяка поръчка (количество * единична цена) и добавете тази информация към DataFrame-а с артикулите.
5. Намерете средния брой артикули на поръчка.
6. Определете топ 5 на най-продаваните артикули по обща стойност.
7. Анализирайте поръчките по градове на клиентите и намерете града с най-висока обща стойност на поръчките.
8. Създайте нов JSON файл, който съдържа само ID на поръчката и общата стойност на всички артикули в тази поръчка.

Решение:

```
import pandas as pd
import json

# Примерни JSON данни (списък от поръчки)
json_data = [
    {
        "order_id": "ORD-2025-0504-1001",
        "created_at": "2025-05-04T09:30:00Z",
        "total_amount": 125.50,
        "customer": {
            "customer_id": "CUST-001",
            "name": "John Doe",
            "address": {
                "street": "123 Main St",
                "city": "Plovdiv",
                "zipcode": "4000"
            }
        },
```

```
        "items": [
            {"item_id": "ITEM-001", "name": "Laptop", "quantity": 1,
"unit_price": 1000.00},
            {"item_id": "ITEM-002", "name": "Mouse", "quantity": 2,
"unit_price": 12.75}
        ],
        "payment_method": "Credit Card",
        "shipping_address": {"street": "456 Oak Ave", "city": "Sofia",
"zipcode": "1000"}
    },
    {
        "order_id": "ORD-2025-0504-1002",
        "created_at": "2025-05-04T10:15:00Z",
        "total_amount": 55.00,
        "customer": {
            "customer_id": "CUST-002",
            "name": "Jane Smith",
            "address": {
                "street": "789 Pine Ln",
                "city": "Sofia",
                "zipcode": "1000"
            }
        },
        "items": [
            {"item_id": "ITEM-003", "name": "Keyboard", "quantity": 1,
"unit_price": 45.00},
            {"item_id": "ITEM-002", "name": "Mouse", "quantity": 1,
"unit_price": 10.00}
        ],
        "payment_method": "PayPal",
        "shipping_address": {"street": "101 Elm Rd", "city": "Plovdiv",
"zipcode": "4000"}
    },
    {
        "order_id": "ORD-2025-0504-1003",
        "created_at": "2025-05-04T11:00:00Z",
        "total_amount": 200.00,
        "customer": {
            "customer_id": "CUST-001",
            "name": "John Doe",
            "address": {
                "street": "123 Main St",
                "city": "Plovdiv",
                "zipcode": "4000"
            }
        },
        "items": [
            {"item_id": "ITEM-001", "name": "Laptop", "quantity": 1,
"unit_price": 180.00}
        ],
        "payment_method": "Credit Card",
        "shipping_address": {"street": "123 Main St", "city": "Plovdiv",
"zipcode": "4000"}
```

```
    }
]

# 1. Четене и нормализиране на JSON данните
orders_df = pd.json_normalize(json_data, sep='_')
print("Основен DataFrame с поръчки:")
print(orders_df.head())

# 2. Създаване на DataFrame с информация за артикулите
items_df = pd.json_normalize(json_data, record_path='items',
meta=['order_id'])
print("\nDataFrame с информация за артикулите:")
print(items_df.head())

# 3. Създаване на DataFrame с адреси на клиентите
customer_address_df = pd.json_normalize(json_data, record_path=['customer',
'address'], meta=['order_id', ('customer', 'customer_id')])
customer_address_df.rename(columns={'street': 'customer_street', 'city':
'customer_city', 'zipcode': 'customer_zipcode'}, inplace=True)
print("\nDataFrame с адреси на клиентите:")
print(customer_address_df.head())

# 4. Изчисляване на общата стойност на всеки артикул
items_df['item_total'] = items_df['quantity'] * items_df['unit_price']
print("\nDataFrame с артикули и обща стойност:")
print(items_df.head())

# 5. Среден брой артикули на поръчка
average_items_per_order =
items_df.groupby('order_id')['item_id'].count().mean()
print(f"\nСреден брой артикули на поръчка: {average_items_per_order:.2f}")

# 6. Топ 5 на най-продаваните артикули по обща стойност
top_selling_items =
items_df.groupby('name')['item_total'].sum().sort_values(ascending=False).he
ad(5)
print("\nТоп 5 на най-продаваните артикули по обща стойност:")
print(top_selling_items)

# 7. Анализ на поръчките по градове на клиентите
orders_with_city = pd.merge(orders_df, customer_address_df[['order_id',
'customer_city']], on='order_id', how='left')
city_sales =
orders_with_city.groupby('customer_city')['total_amount'].sum().sort_values(
ascending=False)
most_valuable_city = city_sales.index[0]
highest_sales_amount = city_sales.iloc[0]
print(f"\nГрадът с най-висока обща стойност на поръчките е:
{most_valuable_city} с обща сума: {highest_sales_amount:.2f}")

# 8. Създаване на нов JSON файл с ID на поръчката и обща стойност на
артикулите
```

```
order_totals =
items_df.groupby('order_id')['item_total'].sum().reset_index()
order_totals_json = order_totals.to_json(orient='records')

with open('order_totals.json', 'w') as f:
    f.write(order_totals_json)

print("\nРезултатите (ID на поръчка и обща стойност на артикулите) са
записани в 'order_totals.json'")
print(order_totals)
```

Разбор на сложността:

- **Вложени JSON структури:** Данните съдържат вложени обекти (например, 'customer', 'address', 'shipping_address'), които трябва да бъдат обработени.
- **JSON масиви:** Всяка поръчка има масив от артикули ('items'), който трябва да бъде разгънат, за да се анализира всеки артикул поотделно.
- **Нормализация на JSON:** Използва се `pd.json_normalize` с различни параметри (`record_path`, `meta`, `sep`) за преобразуване на JSON данните в плоски таблични структури (DataFrame-и).
- **Създаване на множество DataFrame-и:** Създават се няколко DataFrame-а за различните аспекти на данните (основни поръчки, артикули, адреси на клиенти).
- **Свързване на DataFrame-и:** Използва се `pd.merge` за свързване на DataFrame-и въз основа на общи колони (например, 'order_id').
- **Групиране и агрегиране:** Извършват се операции за групиране на данни по различни критерии (например, по ID на поръчка, име на артикул, град) и агрегиране (например, изчисляване на средна стойност, сума, брой).
- **Преобразуване на данни:** Създава се нова колона ('item_total') чрез прилагане на аритметична операция върху съществуващи колони.
- **Записване в JSON формат:** Резултатът от анализа се записва обратно в JSON формат с определена структура (`orient='records'`).

Тези казуси илюстрират как Pandas може да се използва за четене и записване на JSON данни в различни сценарии, от прости API отговори до по-сложни JSON структури. Разбирането на параметъра `orient` и използването на `pd.json_normalize()` са ключови за работа с разнообразни JSON формати.

Въпроси:

1. Кои са основните функции в Pandas за четене и запис на JSON данни?
2. Какво представлява JSON форматът и какви са неговите основни структури (обекти и масиви)?

3. Как `pd.read_json()` преобразува JSON данни в Pandas DataFrame по подразбиране? Как се определят колоните и редовете?
4. Обяснете ролята на параметъра `orient` в `pd.read_json()`. Кои са някои от често използваните стойности за този параметър и как те влияят на структурата на създадения DataFrame?
5. Как `pd.read_json()` обработва вложени JSON обекти и масиви? Как може да се използва `pd.json_normalize()` за справяне с такива структури?
6. Как може да прочетете JSON данни от файл, от URL адрес или от JSON низ в Pandas?
7. Какво прави параметърът `lines=True` в `pd.read_json()`? В какъв тип JSON файлове е полезен?
8. Как `df.to_json()` преобразува Pandas DataFrame в JSON формат по подразбиране?
9. Обяснете ролята на параметъра `orient` в `df.to_json()`. Кои са някои от често използваните стойности и как те оформят JSON изхода?
10. Как може да запишете Pandas DataFrame в JSON файл? Как да контролирате форматирането на JSON файла (например, добавяне на отстъпи)?
11. Каква е разликата между `orient='records'` и `orient='index'` при записване на DataFrame в JSON?
12. Как може да запишете Pandas Series в JSON формат? Как се контролира неговото представяне?
13. В какви ситуации е полезно да експортирате данни от Pandas в JSON формат? Дайте примери.
14. Какви са някои от потенциалните предизвикателства при работа с много големи JSON файлове в Pandas?

Задачи:

1. **Четене на JSON с различна ориентация:** Създайте JSON файл `data_records.json` със следните данни (ориентация 'records'):

```
{"name": "Alice", "age": 25}
{"name": "Bob", "age": 30}
{"name": "Charlie", "age": 22}
```

Прочетете го в DataFrame. След това създайте друг JSON файл `data_index.json` със същите данни, но с ориентация 'index' (като имената са индекси). Прочетете и този файл в DataFrame. Сравнете резултатите.

2. **Работа с вложен JSON:** Създайте JSON файл `nested_data.json` със следната структура:

```
    {
  "users": [
    {"id": 1, "info": {"name": "Alice", "age": 25}},
    {"id": 2, "info": {"name": "Bob", "age": 30}}
  ]
}
```

Прочетете го в DataFrame, като използвате `pd.json_normalize()`, за да "сплескате" вложената структура. Изведете DataFrame-а.

3. **Запис в JSON с различна ориентация и форматиране:** Създайте DataFrame с произволни данни (поне 3 колони и 5 реда). Запишете го в JSON файл `output_records.json` с ориентация 'records' и с отстъпи от 4 интервала. След това запишете същия DataFrame в друг JSON файл `output_index.json` с ориентация 'index'. Отворете и сравнете съдържанието на двата JSON файла.

4. **Четене на JSON Lines:** Създайте JSON Lines файл `data_lines.json` (всеки ред е валиден JSON обект):

```
{"name": "Alice", "age": 25}
{"name": "Bob", "age": 30}
{"name": "Charlie", "age": 22}
```

   Прочетете го в DataFrame, като използвате правилния параметър. Изведете DataFrame-а.

5. **Експортиране на Series в JSON:** Създайте Pandas Series с произволни данни и имена на индекси. Експортирайте го в JSON файл `series_output.json`. Разгледайте съдържанието на файла. Опитайте да експортирате Series с различна ориентация (например, 'index').

VII. HTML: (read_html, to_html)

HTML е стандартният език за маркиране, предназначен за създаване на уеб страници и уеб приложения. Данните често се представят в HTML таблици (`<table>` елементи). Pandas осигурява удобен начин за извличане на тези таблици в DataFrame обекти.

1. Четене на HTML таблици:

```
pd.read_html(io, *, match='.+', flavor=None, header=None, index_col=None,
skiprows=None, attrs=None, parse_dates=False, thousands=',', decimal='.',
na_values=None, keep_default_na=True, converters=None, dtype=None,
true_values=None, false_values=None, storage_options=None
```

Тази функция парсва HTML файлове, HTML низове или URL адреси и се опитва да намери таблични елементи (`<table>`) и да ги превърне в списък от DataFrame обекти.

- **`io`:** Низ, съдържащ път до HTML файл, URL адрес или HTML string. Може да бъде и обект, подобен на файл.
- **`match`:** Регулярен израз, по който да се търсят атрибути `id` или `class` на `<table>` елементите. Само таблици, които съответстват на този израз, ще бъдат върнати. По подразбиране търси всички таблици (`'.+'`).
- **`flavor`:** Указва парсера, който да се използва. Възможни стойности са `'lxml'`, `'html5lib'`, `'bs4'` (BeautifulSoup4). `lxml` е най-бързият и препоръчителен, но изисква инсталация (`pip install lxml`). Ако не е инсталиран, Pandas ще опита да използва други налични парсери.
- **`header`:** Редът (индексиран от 0), който да се използва за заглавия на колоните. По подразбиране Pandas се опитва да го определи.
- **`index_col`:** Колоната (по индекс или име), която да се използва като индекс.
- **`skiprows`:** Брой редове за пропускане преди началото на таблицата (ако има такива в `<table>`).
- **`attrs`:** Речник от атрибути (`{'attribute': 'value'}`) за търсене на конкретна таблица. Например `{'class': 'wikitable'}` ще търси таблици с клас 'wikitable'.
- **`parse_dates, thousands, decimal, na_values, converters, dtype, true_values, false_values, storage_options`:** Тези параметри работят подобно на тези в `pd.read_csv()` и `pd.read_excel()` за обработка на данните след парсване на HTML таблицата.

a) Примери за четене на HTML таблици:

Да предположим, че имаме HTML файл `tables.html` със следното съдържание:

```
<!DOCTYPE html>
<html>
<head>
<title>Примерни таблици</title>
</head>
<body>

<h1>Първа таблица</h1>
<table id="table1">
  <tr>
    <th>Име</th>
    <th>Възраст</th>
```

```
    <th>Град</th>
  </tr>
  <tr>
    <td>Алиса</td>
    <td>25</td>
    <td>София</td>
  </tr>
  <tr>
    <td>Борис</td>
    <td>30</td>
    <td>Пловдив</td>
  </tr>
</table>

<h2>Втора таблица</h2>
<table class="data-table">
  <tr>
    <th>Продукт</th>
    <th>Цена</th>
  </tr>
  <tr>
    <td>Ябълка</td>
    <td>1.20</td>
  </tr>
  <tr>
    <td>Банан</td>
    <td>0.80</td>
  </tr>
</table>

</body>
</html>
```

```
import pandas as pd

# Четене на всички таблици от HTML файла
tables = pd.read_html('tables.html')
print("Списък от DataFrames:\n", tables)
print("\nПървата таблица:\n", tables[0])
print("\nВтората таблица:\n", tables[1])

# Четене само на таблицата с id 'table1'
table1 = pd.read_html('tables.html', attrs={'id': 'table1'})
print("\nТаблица с id 'table1':\n", table1[0])

# Четене само на таблицата с клас 'data-table'
table2 = pd.read_html('tables.html', attrs={'class': 'data-table'})
print("\nТаблица с клас 'data-table':\n", table2[0])
```

б) Ако искате да прочетете таблици от URL адрес:

```
# Пример: Четене на таблици от уеб страница (може да не работи винаги поради структурата на сайта)
url = 'https://bg.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0_%D0%91%D1%8A%D0%BB%D0%B3%D0%B0%D1%80%D0%B8%D1%8F'
wiki_tables = pd.read_html(url, match='Гъстота на населението според надморската височина')
if wiki_tables:
    print("\nТаблица от Wikipedia:\n", wiki_tables[0])
else:
    print("\nНе бяха намерени таблици, съответстващи на критерия.")
```

Важно е да имате инсталиран подходящ парсер (lxml, html5lib или beautifulsoup4), за да може pd.read_html() да работи.

2. Запис на DataFrame в HTML таблица

```
df.to_html(buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, bold_rows=True, classes=None, escape=True, max_rows=None, max_cols=None, show_dimensions=False, decimal='.', border=1, render_links=False, storage_options=None)
```

а) Основни параметри на `to_html`

Този метод рендира DataFrame в HTML таблица (`<table>`).

- **`buf`:** Обект, подобен на файл, в който да се запише HTML кода. Ако е `None` (по подразбиране), връща HTML string.
- **`columns`:** Списък от колони за рендиране (ако е `None`, всички колони се рендират).
- **`index`:** Булев флаг, указващ дали да се рендира индексът като първа колона (или като `<th>` елементи, ако `header=True`).
- **`header`:** Булев флаг, указващ дали да се рендира заглавен ред (`<th>` елементи).
- **`na_rep`:** Низ за представяне на `NaN` стойности.
- **`classes`:** Низ или списък от низове, които да се добавят като класове към `<table>` елемента. Полезно за CSS стилизиране.
- **`escape`:** Булев флаг, указващ дали да се екранират HTML специални символи (`<`, `>`, `&`).
- **`border`:** Цяло число, указващо стойността на атрибута `border` на таблицата.

б) Примери за запис в HTML:

Използвайки DataFrame `df_from_json`:

```
import pandas as pd

# Запис на DataFrame в HTML таблица (като string)
html_table = df_from_json.to_html()
print("HTML таблица:\n", html_table)
```

```
# Запис на DataFrame в HTML файл с клас 'my-data-table' и без индекс
df_from_json.to_html('output_table.html', index=False, classes='my-data-table')
print("DataFrame-ът беше записан в 'output_table.html' като HTML таблица с клас 'my-data-table'.")
```

Можете да отворите output_table.html във уеб браузър, за да видите рендираната таблица. Класът 'my-data-table' може да бъде използван за стилизиране на таблицата с CSS.

3. Допълнително за работата с HTML

Работата с HTML в Pandas може да бъде много полезна, но има и някои допълнителни аспекти и потенциални проблеми, които е добре да се имат предвид:

а) Допълнително при четене на HTML (pd.read_html()):

- **Повече от една таблица, съответстваща на `match` или `attrs`:** Ако има няколко таблици, които отговарят на зададените критерии, `pd.read_html()` ще върне списък от всички съвпадения. Трябва да внимавате и да индексирате резултата, за да получите желания DataFrame.

- **Сложни HTML структури:** HTML страниците могат да бъдат много сложни, с таблици, вложени в други елементи, или с невалиден HTML. Парсерите (`lxml`, `html5lib`, `beautifulsoup4`) се справят с различна степен на успеваемост с такива случаи. `lxml` обикновено е най-стриктен, докато `html5lib` е по-толерантен към невалиден HTML.

- **Таблици без `<thead>` или `<tbody>`:** Pandas се опитва да определи заглавния ред дори и при липса на `<thead>` елемент, като използва първия ред с `<th>` елементи или първия ред с данни, ако няма `<th>`. Параметърът `header` може да се използва за изрично указване на реда за заглавие.

- **Сливане на клетки (`colspan` и `rowspan`):** Pandas се опитва да обработи сливането на клетки, но резултатът може да не винаги е идеален и може да се наложи допълнителна обработка на DataFrame-а.

- **Извличане на данни, които не са в `<table>`:** `pd.read_html()` е специализирана за таблици. За извличане на други данни от HTML (например списъци, параграфи) ще трябва да използвате други библиотеки за парсване на HTML като BeautifulSoup.

- **Динамично генерирано съдържание (JavaScript):** Ако таблиците на уеб страницата се генерират динамично с JavaScript, `pd.read_html()` няма да може да ги прочете директно, тъй като тя само парсва статичния HTML изход. В такива случаи може да се наложи използване на инструменти като Selenium или Puppeteer, които могат да рендират JavaScript и след това да извлекат HTML кода.

б) Допълнително при запис в HTML (`df.to_html()`):

- **Стилизиране с CSS:** Параметърът `classes` е много полезен за добавяне на CSS класове към `<table>` елемента, което позволява лесно стилизиране на таблицата. За по-сложно вградено стилизиране може да се наложи ръчно добавяне на `<style>` тагове или атрибути към HTML кода, генериран от `to_html()`.

- **Контрол на реда на колоните:** Можете да използвате параметъра `columns` за да укажете кои колони и в какъв ред да бъдат включени в HTML таблицата.

- **Форматиране на числа и дати:** Параметрите `formatters` и `float_format` могат да бъдат използвани за контролиране на начина, по който се рендират числата и датите в HTML таблицата.

- **Създаване на връзки (<a> тагове):** Параметърът `render_links` (по подразбиране `False`) указва дали да се рендират URL адреси като HTML връзки (`<a>` тагове).

4. Изключения при работа с HTML:

- **Липса на таблици:** Ако HTML файлът или URL адресът не съдържа таблици, `pd.read_html()` ще върне празен списък.

- **Грешки при парсване:** Ако HTML кодът е силно невалиден или парсерът срещне неочаквана структура, може да възникне грешка при парсване. Изборът на правилния `flavor` може да помогне в някои случаи.

- **Проблеми с кодирането:** Както при други файлови формати, може да има проблеми с кодирането на HTML страницата. Уверете се, че Pandas използва правилното кодиране при четене (обикновено се справя автоматично).

- **Големи HTML таблици:** Парсването на много големи HTML таблици може да отнеме време и да изисква значително количество памет.

Казус 1: Извличане на таблица с валутни курсове от уебсайт

Представете си, че искате да получите актуални валутни курсове от уебсайт, който ги представя в HTML таблица. Искате да прочетете тази таблица в Pandas DataFrame за по-нататъшен анализ или сравнение.

Решение:

Използвайте `pd.read_html()` с URL адреса на уебсайта. Може да се наложи да инспектирате HTML структурата на страницата, за да определите дали има повече от една таблица и евентуално да използвате параметъра `attrs` за по-точно насочване към желаната таблица.

```
import pandas as pd

url = 'https://www.bnb.bg/Statistics/StInterbankForexMarket/index.htm'

# Прочитане на всички таблици от URL адреса
html_tables = pd.read_html(url)

# Обикновено валутните курсове са в една от таблиците.
# Може да се наложи да проверите съдържанието на всяка таблица,
# за да намерите тази, която ви трябва.
if html_tables:
    currency_rates_df = html_tables[1] # Предполагаме, че е първата таблица
    print("Таблица с валутни курсове от БНБ:\n", currency_rates_df.head())

    # Може да се наложи допълнително почистване на данните (премахване на
ненужни редове/колони).
else:
    print("Не бяха намерени таблици на посочения URL адрес.")
```

Важно: Успешното извличане зависи от структурата на HTML страницата. Ако таблицата е динамично генерирана с JavaScript, `pd.read_html()` може да не я прочете. Също така, може да се наложи да инсталирате допълнителни библиотеки като `lxml` или `beautifulsoup4`, ако те не са налични.

Казус 2: Записване на обобщени резултати в HTML формат за уеб отчет

След като сте анализирали данни и сте получили обобщени резултати в Pandas DataFrame, искате да ги експортирате като HTML таблица, която да бъде вградена в уеб отчет или изпратена по имейл.

Решение:

Използвайте метода `df.to_html()` за преобразуване на DataFrame-а в HTML табличен формат. Можете да използвате параметъра `index=False`, за да избегнете записването на индекса, и `classes` за добавяне на CSS класове за стилизиране.

```
import pandas as pd

# Да предположим, че имаме DataFrame с обобщени данни за продажби
sales_summary = pd.DataFrame({
    'Категория': ['Електроника', 'Книги', 'Дрехи'],
    'Средни продажби': [150.50, 75.20, 90.80],
    'Общо количество': [100, 200, 150]
})

# Запис в HTML формат (като string)
html_output = sales_summary.to_html(index=False, classes='sales-table')
print("HTML таблица:\n", html_output)

# Можете да запишете HTML кода и във файл:
with open('sales_report.html', 'w', encoding='utf-8') as f:
    f.write('<h1>Отчет за продажбите</h1>\n')
    f.write(html_output)
    f.write('\n<style>\n.sales-table { border-collapse: collapse; width:
100%; }\n.sales-table th, .sales-table td { border: 1px solid black;
padding: 8px; text-align: left; }\n</style>')

print("HTML таблицата беше записана в 'sales_report.html'.")
```

В този пример добавяме и CSS стилове директно в HTML файла за по-добро представяне на таблицата.

Казус 3: Четене на таблица с исторически данни от локален HTML файл

Имате локален HTML файл (`historical_data.html`), който съдържа таблици с исторически данни. Искате да прочетете тази таблица в Pandas DataFrame за анализ на тенденции.

Може да използвате следните генратори:

Генератор 1:

```
import pandas as pd
```

```
import numpy as np

# Функция за създаване на случайна таблица с исторически данни
def generate_historical_table(year, num_rows=10):
    dates = pd.date_range(f'{year}-01-01', periods=num_rows, freq='M')
    data = {
        'Дата': dates.strftime('%Y-%m-%d'),
        'Цена': np.round(np.random.uniform(100, 500, num_rows), 2),
        'Обем': np.random.randint(1000, 10000, num_rows),
    }
    return pd.DataFrame(data)

# Генерираме няколко таблици за различни години
tables = []
for year in [2021, 2022, 2023]:
    df = generate_historical_table(year)
    df_html = df.to_html(index=False, border=1)
    tables.append(f"<h2>Исторически данни за {year}</h2>\n{df_html}")

# Обединяваме в един HTML документ
html_content = f"""
<!DOCTYPE html>
<html lang="bg">
<head>
    <meta charset="UTF-8">
    <title>Исторически Данни</title>
</head>
<body>
    <h1>Исторически Данни по Години</h1>
    {'<br>'.join(tables)}
</body>
</html>
"""

# Записваме във файл
with open("historical_data.html", "w", encoding="utf-8") as f:
    f.write(html_content)

print("Файлът 'historical_data.html' е създаден успешно.")
```

Генерато 2:

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

# Примерни данни
countries = ['България', 'Германия', 'Франция', 'Италия', 'САЩ', 'Япония']
```

```
cities = {
    'България': ['София', 'Пловдив', 'Варна'],
    'Германия': ['Берлин', 'Мюнхен', 'Хамбург'],
    'Франция': ['Париж', 'Лион', 'Марсилия'],
    'Италия': ['Рим', 'Милано', 'Неапол'],
    'САЩ': ['Ню Йорк', 'Лос Анджелис', 'Чикаго'],
    'Япония': ['Токио', 'Осака', 'Киото']
}
persons = ['Иван Петров', 'Анна Дюпон', 'Джон Смит', 'Мария Роси', 'Кенджи
Такада']
events = ['Подписване на договор', 'Среща на високо ниво', 'Икономически
форум',
          'Политически протест', 'Научна конференция', 'Културно събитие']

# Функция за генериране на една таблица
def generate_event_table(year, num_rows=10):
    dates = pd.date_range(f'{year}-01-01', periods=num_rows, freq='M')
    data = []
    for date in dates:
        country = random.choice(countries)
        city = random.choice(cities[country])
        person = random.choice(persons)
        event = random.choice(events)
        data.append([date.strftime('%Y-%m-%d'), country, city, person,
event])

    df = pd.DataFrame(data, columns=['Дата', 'Държава', 'Град или Област',
'Свързано лице', 'Събитие'])
    return df

# Създаваме няколко таблици за различни години
html_tables = []
for year in [2022, 2023, 2024]:
    df = generate_event_table(year)
    html_table = df.to_html(index=False, border=1)
    html_tables.append(f"<h2>Събития през {year}</h2>\n{html_table}")

# Финален HTML документ
html_content = f"""
<!DOCTYPE html>
<html lang="bg">
<head>
    <meta charset="UTF-8">
    <title>Исторически събития</title>
    <style>
        body {{ font-family: Arial, sans-serif; margin: 20px; }}
        table {{ border-collapse: collapse; width: 100%; margin-bottom:
30px; }}
        th, td {{ border: 1px solid #ccc; padding: 8px; text-align: left; }}
        h2 {{ color: #333; }}
    </style>
</head>
<body>
```

```
    <h1>Исторически събития по години</h1>
    {"<br>".join(html_tables)}
</body>
</html>
"""

# Записваме HTML файла
with open("historical_data.html", "w", encoding="utf-8") as f:
    f.write(html_content)

print("Файлът 'historical_data.html' е създаден успешно.")
```

Решение:

Използвайте `pd.read_html()` с пътя до локалния HTML файл.

```
import pandas as pd

file_path = 'historical_data.html'

# Да предположим, че historical_data.html съдържа една или повече таблици
historical_tables = pd.read_html(file_path)

if historical_tables:
    historical_data_df = historical_tables[0] # Предполагаме, че е първата таблица
    print("Исторически данни:\n", historical_data_df.head())
else:
    print(f"Не бяха намерени таблици във файла '{file_path}'.")
```

Тези казуси илюстрират как Pandas може да се използва за извличане на данни от HTML таблици в уебсайтове или локални файлове и за експортиране на DataFrame-и обратно в HTML формат за различни цели, като уеб отчети или споделяне на данни.

Казус 4*: Събиране и анализ на данни за футболни резултати от различни сайтове

Ситуация:

Вие сте запален футболен фен и искате да съберете и анализирате данни за резултатите от мачове от няколко различни уебсайта, които представят информацията в HTML таблици. Всеки сайт има малко по-различен формат на таблиците и съдържа различна информация (например, някои сайтове дават само краен резултат, други - и информация за полувремето, голмайстори и др.).

Задача:

1. Определете списък с URL адреси на поне два различни уебсайта, съдържащи таблици с футболни резултати за дадено първенство или кръг.
2. За всеки URL адрес, използвайте `pd.read_html()` за извличане на всички таблици от страницата.

3. Инспектирайте извлечените таблици, за да идентифицирате тази, която съдържа резултатите от мачовете (може да се наложи да използвате критерии като брой колони, имена на колони или съдържание).
4. Нормализирайте данните от всяка таблица, така че да съдържат поне следните колони (ако е възможно да се извлекат): 'Дата', 'Домакин', 'Гост', 'Резултат'. Може да се наложи да извършите почистване на данни (например, разделяне на колони, премахване на излишни символи).
5. Ако е възможно, извлечете допълнителна информация като резултат на полувремето или голмайстори (може да изисква по-сложно парсване на съдържанието на клетките).
6. Създайте един общ Pandas DataFrame, съдържащ резултатите от всички уебсайтове. Добавете колона 'Източник', указваща от кой сайт са взети данните.
7. Анализирайте събраните данни:
    - Колко мача са изиграни общо?
    - Кой е най-често срещаният резултат?
    - Кой отбор е имал най-много победи като домакин?
    - Кой отбор е имал най-много победи като гост?
8. Запазете обединения DataFrame с резултатите в CSV файл.

Примерно решение (ще изисква реални URL адреси и адаптация според структурата на сайтовете):

```
import pandas as pd

# 1. Списък с URL адреси
urls = [
    'https://www.example-football-results1.com/league/round1',  # Заменете с реален URL
    'https://www.another-football-site.net/results/round-one'   # Заменете с реален URL
]

all_results = []

# 2. Обхождане на URL адресите
for url in urls:
    try:
        # Извличане на всички таблици от страницата
        html_tables = pd.read_html(url)

        # 3. Идентифициране на таблицата с резултати (трябва да се адаптира според сайта)
        results_table = None
        for table in html_tables:
            if 'Домакин' in table.columns and 'Гост' in table.columns and 'Резултат' in table.columns:
                results_table = table
                break
            elif len(table.columns) >= 4 and 'vs' in table.iloc[0].astype(str).str.lower().any():
                # Примерна логика за друг формат
                results_table = table
                results_table.columns = ['Дата', 'Домакин - Гост', 'Резултат', 'Други'] # Примерни колони
```

```
                results_table[['Домакин', 'Гост']] = results_table['Домакин
- Гост'].str.split(' - ', expand=True)
                results_table.drop(columns=['Домакин - Гост', 'Други'],
inplace=True)
                # Може да се наложи допълнително преименуване на колони

        if results_table is not None:
            # 4. Нормализиране на данните (адаптирайте според колоните на
таблицата)
            if 'Date' in results_table.columns and 'Home' in
results_table.columns and 'Away' in results_table.columns and 'Score' in
results_table.columns:
                results_table.rename(columns={'Date': 'Дата', 'Home':
'Домакин', 'Away': 'Гост', 'Score': 'Резултат'}, inplace=True)
            elif 'Дата' in results_table.columns and 'Резултат' in
results_table.columns:
                # Допълнителна обработка, ако колоните са различни
                pass # Трябва да се имплементира специфична логика за
парсване

            # 6. Добавяне на колона 'Източник'
            results_table['Източник'] = url
            all_results.append(results_table)
        else:
            print(f"Не беше намерена таблица с резултати на страницата:
{url}")

    except Exception as e:
        print(f"Възникна грешка при четене на URL '{url}': {e}")

# 6. Обединяване на резултатите от всички сайтове
if all_results:
    combined_results_df = pd.concat(all_results, ignore_index=True)
    print("\nОбединен DataFrame с резултати:")
    print(combined_results_df.head())

    # 7. Анализ на данните
    total_matches = len(combined_results_df)
    print(f"\nОбщо изиграни мачове: {total_matches}")

    most_common_result = combined_results_df['Резултат'].mode()
    print(f"Най-често срещаният резултат: {most_common_result.tolist()}")

    home_wins =
combined_results_df[combined_results_df['Резултат'].str.split('-
').str[0].astype(int) > combined_results_df['Резултат'].str.split('-
').str[1].astype(int)]
    most_home_wins = home_wins['Домакин'].value_counts().nlargest(1)
    print("\nОтбор с най-много победи като домакин:")
    print(most_home_wins)

    away_wins =
combined_results_df[combined_results_df['Резултат'].str.split('-
```

```
').str[0].astype(int) < combined_results_df['Резултат'].str.split('-
').str[1].astype(int)]
    most_away_wins = away_wins['Гост'].value_counts().nlargest(1)
    print("\nОтбор с най-много победи като гост:")
    print(most_away_wins)

    # 8. Запазване в CSV файл
    combined_results_df.to_csv('football_results.csv', index=False,
encoding='utf-8')
    print("\nРезултатите са запазени във 'football_results.csv'")

else:
    print("Не бяха събрани данни от нито един URL.")
```

Разбор на сложността:

- **Работа с множество уебсайтове:** Данните се извличат от различни източници, което увеличава вероятността от разлики във формата.
- **Различен формат на HTML таблиците:** Всеки сайт може да има уникална структура на таблиците (различни имена на колони, различен брой колони, различна организация на данните в клетките).
- **Идентифициране на правилната таблица:** Необходимо е да се напише логика за намиране на таблицата с резултати сред всички таблици на страницата.
- **Нормализация и почистване на данни:** Данните често изискват значително почистване и преобразуване, за да се приведат към единен формат (например, разделяне на колона с резултат, преименуване на колони, премахване на излишни символи).
- **Възможно извличане на допълнителна информация:** По-сложното парсване може да е необходимо за извличане на детайли като резултат на полувремето или голмайстори от съдържанието на клетките.
- **Комбиниране на данни от различни източници:** Събраните данни трябва да бъдат обединени в един DataFrame, като се запазва информация за произхода им.
- **Анализ на обединените данни:** Извършват се различни аналитични операции върху комбинирания DataFrame.

!!!ВНИМАНИЕ!!! **Този казус демонстрира реално приложение на `pd.read_html()` за събиране на данни от уебсайтове, като подчертава необходимостта от гъвкавост при обработката на различни HTML структури и важността на почистването и нормализирането на данните за последващ анализ. Решението изисква адаптация в зависимост от конкретните уебсайтове, които се използват.**

Въпроси:

1. Коя е основната функция в Pandas за четене на HTML таблици от уебсайт или HTML файл?
2. Какво връща функцията `pd.read_html()`? Защо е възможно да върне списък от DataFrame-и?
3. Ако HTML страница съдържа няколко таблици, как може да изберете конкретна таблица, която ви интересува, използвайки `pd.read_html()`? Обяснете параметрите, които могат да бъдат полезни за това.
4. Какви библиотеки са необходими (допълнително към Pandas) за успешното парсване на HTML от `pd.read_html()`? Коя е препоръчителната библиотека и защо?
5. Как `pd.read_html()` се справя с HTML таблици, които имат сложна структура (например, обединени клетки, множество заглавни редове)?
6. Как може да прочетете HTML таблица от локален HTML файл?
7. Коя е основната функция в Pandas за преобразуване на DataFrame в HTML таблица?
8. Как може да контролирате дали индексът на DataFrame-а да бъде включен в генерирания HTML код с `df.to_html()`?
9. Как може да добавите CSS класове към HTML таблицата или към нейните елементи (например, `<table>`, `<th>`, `<td>`) при използване на `df.to_html()`? Защо би било полезно това?
10. Как може да запишете генерирания HTML код във файл?
11. В какви ситуации е полезно да конвертирате DataFrame в HTML формат? Дайте примери.
12. Как `df.to_html()` се справя с липсващи стойности (NaN) в DataFrame-а?
13. Как може да повлияете на начина, по който се форматират числовите стойности в HTML таблицата, генерирана от `df.to_html()`?
14. Какви са някои от ограниченията на `pd.read_html()` при парсване на HTML (например, динамично съдържание)?

Задачи:

1. **Четене на таблица от уебсайт:** Изберете уебсайт с поне една HTML таблица (например, таблица с данни за акции, спортни резултати и т.н.). Използвайте `pd.read_html()` за да прочетете таблицата в DataFrame. Изведете първите няколко реда и информация за DataFrame-а. Ако сайтът има повече от една таблица, опитайте да идентифицирате индекса на желаната таблица.
2. **Четене на таблица с атрибути:** Намерете уебсайт, където таблицата има специфични HTML атрибути (например, `class` или `id`). Използвайте параметъра `attrs` на `pd.read_html()` за да прочетете само тази конкретна таблица.
3. **Запис в HTML с клас и без индекс:** Създайте DataFrame с произволни данни. Конвертирайте го в HTML таблица, като не включвате индекса и добавите CSS клас с име "data-table" към елемента `<table>`. Изведете генерирания HTML код.
4. **Запис на част от DataFrame в HTML:** Създайте DataFrame и изберете подмножество от него (например, няколко колони и редове). Конвертирайте това подмножество в HTML таблица и го запишете във файл `subset_table.html`. Отворете файла в браузър, за да видите резултата.
5. **Четене на локален HTML файл:** Създайте прост HTML файл (`my_table.html`), който съдържа една таблица с няколко реда и колони. Прочетете този файл в Pandas DataFrame, използвайки `pd.read_html()`. Изведете DataFrame-а.

VIII. XML: (read_xml, to_xml)

XML е език за маркиране, предназначен за кодиране на документи в машинно четим и човешки четим формат. Той е йерархичен и използва тагове за дефиниране на елементи и техните атрибути.

От версия 1.4.0 на Pandas, вече съществуват вградени функции за четене и писане на XML файлове: `pd.read_xml()` и `df.to_xml()` (както и `series.to_xml()`).

1. Четене на XML

Тази функция парсва XML файлове, XML низове или URL адреси и ги преобразува в DataFrame.

```
(pd.read_xml(path_or_buffer, *, xpath='./*', namespaces=None, converters=None, dtype=None,
attrs_only=False, parse_dates=False, encoding='utf-8', storage_options=None)):
```

а) Основни параметри на `read_xml`

- **`path_or_buffer`:** Низ, съдържащ път до XML файл, URL адрес или XML string. Може да бъде и обект, подобен на файл.
- **`xpath`:** XPath израз, указващ кои възли от XML структурата да се използват за създаване на редове в DataFrame-а. По подразбиране е `'./*'`, което избира всички директни деца на корена.
- **`namespaces`:** Речник от namespace префикси към URL адреси, ако XML файлът използва namespaces.
- **`converters`, `dtype`, `parse_dates`, `encoding`, `storage_options`:** Тези параметри работят подобно на тези в други функции за четене на файлове в Pandas.
- **`attrs_only`:** Булев флаг. Ако е `True`, ще се четат само атрибутите на възлите, посочени от `xpath`.

б) Пример за четене на XML с `pd.read_xml()`:

Използвайки същия XML файл `data.xml`:

```
<data>
  <record>
    <name>Алиса</name>
    <age>25</age>
    <city>София</city>
  </record>
  <record>
    <name>Борис</name>
    <age>30</age>
    <city>Пловдив</city>
  </record>
  <record>
    <name>Ваня</name>
    <age>27</age>
    <city>Варна</city>
  </record>
</data>
```

```
import pandas as pd
```

```
df_from_xml_pd = pd.read_xml('data.xml', xpath='//record')
print("DataFrame от data.xml (с pd.read_xml):\n", df_from_xml_pd)
```

В този случай, XPath изразът `'//record'` избира всички елементи с таг `<record>` в XML файла, и Pandas автоматично създава DataFrame от техните под-елементи.

2. Запис на DataFrame в XML

Този метод конвертира DataFrame (или Series) в XML формат.

```
DataFrame.to_xml(path_or_buffer=None, *, index=True, root_name='data', row_name='row', na_rep=None, attr_cols=None, elem_cols=None, namespaces=None, prefix=None, encoding='utf8', xml_declaration=True, pretty_print=True, parser='lxml', stylesheet=None, compression='infer', storage_options=None)
```

a) Основни параметри на `to_xml`

- **`path_or_buffer`**:
  - **Предназначение:** Указва къде да се запише генерираният XML изход.
  - **Тип:** Път до файл (`str`, `PathLike`) или обект, подобен на буфер (например отворен файл или `io.StringIO`).
  - **Поведение:** Ако е зададен път към файл, XML съдържанието се записва във файла. Ако е `None`, методът **връща XML съдържанието като низ (string)**, което е полезно, ако искаш да работиш с XML данните в паметта, преди да ги запишеш или използваш.
- **`index`**:
  - **Предназначение:** Контролира дали **индексът** на `DataFrame`-а трябва да бъде включен в XML изхода.
  - **Тип:** `bool` (логическа стойност).
  - **Поведение:**
    - Ако `True` (по подразбиране), индексът на `DataFrame`-а ще бъде записан като отделен XML елемент (`<index>...</index>`) в рамките на всеки ред (`<row>...</row>`).
    - Ако `False`, индексът няма да бъде включен в изхода по този начин. За да експортираш индекса като **атрибут**, трябва първо да го превърнеш в колона и след това да използваш параметъра `attr_cols`.
- **`root_name`**:
  - **Предназначение:** Задава името на **кореновия XML елемент**. Този елемент обхваща целия набор от данни.
  - **Тип:** `str`.
  - **Поведение:** По подразбиране е `'data'`. Всички редове на `DataFrame`-а ще бъдат вложени вътре в този елемент.
- **`row_name`**:
  - **Предназначение:** Задава името на XML елемента, който представлява **всеки ред** от `DataFrame`-а.
  - **Тип:** `str`.
  - **Поведение:** По подразбиране е `'row'`. За всеки ред от `DataFrame`-а ще бъде създаден елемент с това име.
- **`na_rep`**:

    - **Предназначение:** Определя как да бъдат представени **липсващите стойности** (`NaN`, `None`, `NaT`) в XML изхода.
    - **Тип:** `str`.
    - **Поведение:** Ако е зададен низ (напр. `'NULL'`, `'N/A'`, `''` за празен низ), всички липсващи стойности в `DataFrame`-а ще бъдат заменени с този низ в XML файла. Ако е `None` (по подразбиране), елементите с липсващи стойности просто ще бъдат пропуснати в XML изхода.
- **`attr_cols`**:
    - **Предназначение:** Позволява да укажеш кои колони от `DataFrame`-а да бъдат изнесени като **XML атрибути** на елемента `row_name`, вместо като вложени елементи.
    - **Тип:** Списък от низове (`list` от `str`).
    - **Поведение:** Подаваш списък с имената на колоните, които искаш да станат атрибути. Останалите колони ще бъдат изнесени като вложени елементи. **Важно:** Ако искаш да включиш индекса като атрибут, той трябва първо да бъде превърнат в колона.
- **`elem_cols`**:
    - **Предназначение:** Позволява да укажеш кои колони от `DataFrame`-а да бъдат изнесени като **вложени XML елементи**. Използва се заедно с `attr_cols`, за да се разграничи как да се обработват колоните.
    - **Тип:** Списък от низове (`list` от `str`).
    - **Поведение:** Ако са посочени както `attr_cols`, така и `elem_cols`, само колоните, изброени в един от тези списъци, ще бъдат включени в изхода. Ако не са посочени нито `attr_cols`, нито `elem_cols`, всички колони ще бъдат изнесени като елементи по подразбиране (освен ако не са изрично изключени чрез `attr_cols`).
- **`namespaces`**:
    - **Предназначение:** Дефинира XML пространства от имена (namespaces) за генерирания изход.
    - **Тип:** Речник (`dict`).
    - **Поведение:** Речник, където ключовете са префиксите на пространствата от имена, а стойностите са техните URI.
- **`prefix`**:
    - **Предназначение:** Задава префикс на XML елементите, когато се използват пространства от имена.
    - **Тип:** `str`.
- **`encoding`**:
    - **Предназначение:** Указва кодирането на символите за XML файла.
    - **Тип:** `str`.
    - **Поведение:** По подразбиране е `'utf-8'`. Изключително важно за правилното изобразяване на специални символи.
- **`xml_declaration`**:
    - **Предназначение:** Контролира дали да се включи XML декларация (`<?xml version="1.0" encoding="..."?>`) в началото на файла.
    - **Тип:** `bool`.
    - **Поведение:** По подразбиране е `True`.
- **`pretty_print`**:
    - **Предназначение:** Контролира дали XML изходът да бъде форматиран с отстъпи (indentation) за по-добра четимост.
    - **Тип:** `bool`.
    - **Поведение:** По подразбиране е `True`, което прави XML файла по-лесен за четене от човек. Ако е `False`, XML ще бъде записан на един ред (или с минимални прекъсвания).
- **`parser`**:
    - **Предназначение:** Указва кой XML парсер да се използва за вътрешната обработка.

- **Тип:** `str`.
- **Поведение:** По подразбиране е `'lxml'`, който е бърз и мощен парсер. Може да бъде и `'etree'`.

- **`stylesheet`:**
  - **Предназначение:** Позволява да се включи референция към XSLT стилов лист в XML файла.
  - **Тип:** `str` (път до файла).
  - **Поведение:** Позволява прилагане на стилове или трансформация към XML при преглед в браузър или друг XSLT-съвместим софтуер.
- **`compression`:**
  - **Предназначение:** Позволява компресия на изходния файл.
  - **Тип:** `str` или `dict`.
  - **Поведение:** Например `'gzip'`, `'bz2'`, `'zip'`. По подразбиране `'infer'` опитва да познае компресията от разширението на файла.
- **`storage_options`:**
  - **Предназначение:** Допълнителни опции, специфични за файловата система или облачното хранилище, ако `path_or_buffer` сочи към такъв ресурс.
  - **Тип:** `dict`.
  - **Поведение:** Например, могат да се подават акредитивни данни за достъп до S3 кошче.

б) Пример за запис в XML с `df.to_xml()`:

Използвайки DataFrame `df_from_xml_pd`:

```
import pandas as pd

# Запис на DataFrame в XML файл
df_from_xml_pd.to_xml('output_pandas.xml')
print("\nDataFrame-ът беше записан в 'output_pandas.xml' (с pd.to_xml).")

# Запис на DataFrame в XML string с по-лесно четене
xml_string_pd = df_from_xml_pd.to_xml(index=False, pretty_print=True)
print("\nXML string:\n", xml_string_pd)
```

Съдържанието на output_pandas.xml (по подразбиране) ще включва индекс като отделен елемент. Ако използвате index=False, индексът няма да бъде записан като елемент.

3. Използване на библиотеката lxml

а) Четене на XML и преобразуване в DataFrame чрез lxml

Процесът обикновено включва следните стъпки:

1) **Парсване на XML файла:** Използване на библиотека като `lxml` или `xml.etree.ElementTree` за прочитане и парсване на XML структурата в дървовидна форма.
2) **Извличане на данни:** Навигиране през XML дървото, за да се намерят желаните елементи и техните атрибути (които ще станат колони и стойности в DataFrame-а).
3) **Създаване на DataFrame:** Организиране на извлечените данни в списъци от речници или други подходящи структури и създаване на Pandas DataFrame от тях.

Пример с `xml.etree.ElementTree` (от стандартната библиотека):

Да предположим, че имаме XML файл `data.xml` със следното съдържание:

```
<data>
  <record>
    <name>Алиса</name>
    <age>25</age>
    <city>София</city>
  </record>
  <record>
    <name>Борис</name>
    <age>30</age>
    <city>Пловдив</city>
  </record>
  <record>
    <name>Ваня</name>
    <age>27</age>
    <city>Варна</city>
  </record>
</data>
```

```
import pandas as pd
import xml.etree.ElementTree as ET

def parse_xml_to_dataframe(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    data = []
    columns = []
    for record in root.findall('record'):
        row = {}
        for element in record:
            tag = element.tag
            text = element.text
            if tag not in columns:
                columns.append(tag)
            row[tag] = text
        data.append(row)
    return pd.DataFrame(data, columns=columns)

df_from_xml = parse_xml_to_dataframe('data.xml')
print("DataFrame от data.xml:\n", df_from_xml)
```

Пример с `lxml` (изисква инсталация `pip install lxml`):

`lxml` обикновено е по-бърз и предлага повече възможности за XPath заявки, което улеснява навигацията в сложни XML структури.

```
import pandas as pd
from lxml import etree
```

```
def parse_xml_with_lxml(xml_file):
    tree = etree.parse(xml_file)
    records = tree.xpath('//record')
    data = []
    columns = set()
    for record in records:
        row = {}
        for element in record:
            tag = element.tag
            text = element.text
            row[tag] = text
            columns.add(tag)
        data.append(row)
    return pd.DataFrame(data, columns=list(columns))

df_from_xml_lxml = parse_xml_with_lxml('data.xml')
print("\nDataFrame от data.xml (с lxml):\n", df_from_xml_lxml)
```

В зависимост от структурата на вашия XML файл, ще трябва да адаптирате логиката за намиране на правилните елементи и атрибути. Ако XML файлът е по-сложен (например с вложени елементи или атрибути, които трябва да станат колони), може да се наложи по-сложно парсване и трансформация.

б) Запис на DataFrame в XML чрез lxml:

Записването на DataFrame в XML формат също може да използвате на външни библиотеки за генериране на XML структурата. Можете да итерирате през редовете на DataFrame-а и да създавате съответните XML елементи и атрибути.

Пример за запис в XML с `xml.etree.ElementTree`:

```
import pandas as pd
import xml.etree.ElementTree as ET

def dataframe_to_xml(df, output_file):
    root = ET.Element('data')
    for index, row in df.iterrows():
        record = ET.SubElement(root, 'record')
        for column, value in row.items():
            element = ET.SubElement(record, column)
            element.text = str(value)

    tree = ET.ElementTree(root)
    ET.indent(tree, space="\t", level=0) # За по-лесно четене
    ET.write(output_file, encoding='utf-8', xml_declaration=True)

# Използване на DataFrame-а, създаден по-рано
dataframe_to_xml(df_from_xml, 'output.xml')
print("\nDataFrame-ът беше записан в 'output.xml'.")
```

Съдържанието на `output.xml` ще бъде:

```
<?xml version='1.0' encoding='utf-8'?>
<data>
  <record>
    <name>Алиса</name>
    <age>25</age>
    <city>София</city>
  </record>
  <record>
    <name>Борис</name>
    <age>30</age>
    <city>Пловдив</city>
  </record>
  <record>
    <name>Ваня</name>
    <age>27</age>
    <city>Варна</city>
  </record>
</data>
```

За по-сложни XML структури (например с атрибути, вложени елементи) ще трябва да адаптирате функцията `dataframe_to_xml`.

4. Допълнително при работа с XML в Pandas

Въпреки че `pd.read_xml()` и `df.to_xml()` улесняват работата с XML, съществуват някои допълнителни аспекти и потенциални проблеми, които е добре да се имат предвид:

a) Допълнително при четене на XML (`pd.read_xml()`):

- **XPath изрази:** Познаването на XPath е ключово за ефективното извличане на данни от сложни XML структури. Можете да използвате по-сложни XPath изрази за филтриране, избиране на атрибути и навигиране в йерархията на XML документа.

```
# Пример: Четене само на имената на хората над 25 години
df_filtered = pd.read_xml('data.xml', xpath='//record[age > 25]/name')
print(df_filtered)
```

Забележете, че резултатът от такъв XPath израз може да не е директно DataFrame в очаквания формат и може да се нуждае от допълнителна обработка.

- **Атрибути като колони:** По подразбиране `pd.read_xml()` чете съдържанието на елементите като данни. Ако искате да прочетете атрибутите на елементите като колони, можете да използвате XPath за достъп до атрибутите (например `//record/@attribute_name`) или да настроите `attrs_only=True`.

```
# Пример: XML с атрибути
xml_with_attrs = '<data><record id="1" name="Алиса"/><record id="2"
name="Борис"/></data>'
df_attrs = pd.read_xml(xml_with_attrs, xpath='//record', attrs_only=True)
print(df_attrs)
```

- **Namespaces:** Ако XML файлът използва namespaces, трябва да ги декларирате в параметъра `namespaces` като речник, за да можете да ги използвате във вашите XPath изрази.
- **Различни структури на XML:** `pd.read_xml()` работи най-добре с XML файлове, където повтарящи се елементи съдържат сходни под-елементи, които могат да бъдат превърнати в колони. За по-различни структури може да се наложи по-сложен XPath или предварителна обработка на XML файла.

б) Допълнителни аспекти при запис в XML (`df.to_xml()`):

- **Контрол на имената на елементите:** По подразбиране `df.to_xml()` използва имената на колоните като имена на елементи. Няма пряк параметър за лесно персонализиране на тези имена за всички колони наведнъж, но можете да манипулирате DataFrame-а преди запис (например да преименувате колоните).
- **Добавяне на атрибути:** `df.to_xml()` не предоставя вградена функционалност за директно добавяне на атрибути към елементите въз основа на данните в DataFrame-а. За тази цел може да се наложи да използвате други XML библиотеки (като `lxml` или `xml.etree.ElementTree`) след като DataFrame-ът е записан като XML string (ако `to_xml` е върнал string).
- **По-сложни XML структури:** Ако имате нужда от по-сложна XML структура (например вложени елементи, различни начини за представяне на индекс), може да е по-подходящо да генерирате XML файла ръчно с помощта на специализирана XML библиотека, като използвате данните от DataFrame-а.

3. Изключения и потенциални проблеми:

- **`ValueError` при неправилен XPath:** Ако предоставеният XPath израз е невалиден или не намира съвпадения, `pd.read_xml()` може да върне грешка или празен DataFrame.

- **Проблеми с кодирането:** Уверете се, че кодирането, зададено в параметъра `encoding`, съответства на кодирането на XML файла.

- **XML файлове с много дълбока йерархия или много големи файлове:** Парсването на такива файлове може да бъде ресурсоемко и да отнеме много време. В някои случаи може да се наложи итеративно обработване на части от файла (ако структурата го позволява).

- **Невалиден XML:** Ако XML файлът е невалиден, `pd.read_xml()` може да върне грешка в зависимост от използвания парсер.

- **Ограничения на `to_xml()`:** Както беше споменато, `to_xml()` е сравнително базов метод за запис и може да не поддържа всички сложни XML сценарии без допълнителна обработка:

1) Избор на подходяща XML библиотека:

- **`lxml`:** Това е една от най-мощните и бързи библиотеки за обработка на XML и HTML в Python. Тя предлага поддръжка за XPath и XSLT, както и гъвкаво API за парсване и генериране на XML. Ако се нуждаете от висока производителност и разширени възможности за навигация и манипулация на XML, `lxml` е отличен избор.
- **`xml.etree.ElementTree` (стандартна библиотека):** Тази библиотека е част от стандартната инсталация на Python и е по-лека от `lxml`. Тя е подходяща за по-прости XML структури и не изисква допълнителна инсталация. Въпреки че е по-бавна и с по-малко възможности от `lxml`, тя е удобна за основни задачи.

- **`BeautifulSoup4 (bs4)`:** Въпреки че е по-известна с парсването на HTML, `BeautifulSoup` може да парсва и XML. Тя е особено полезна при работа с недобре форматиран или невалиден XML, тъй като е по-толерантна към грешки.

2) **Общ подход при сложни XML структури:**

- **Парсване с избраната библиотека:** Използвайте една от гореспоменатите библиотеки, за да прочетете и парсвате XML файла в дървовидна структура.
- **Навигиране и извличане на данни:** Използвайте методите на библиотеката (например XPath с `lxml` или `find`, `findall` с `ElementTree`) за навигиране в XML дървото и извличане на желаните данни. Това може да включва достъп до елементи, атрибути и текст.
- **Трансформиране на данните:** Тъй като сложните XML структури често не се превеждат директно в плоски таблици, ще трябва да трансформирате извлечените данни в подходящ формат за създаване на Pandas DataFrame. Това може да включва създаване на списъци от речници, където всеки речник представлява ред, а ключовете са имената на колоните.
- **Създаване на DataFrame:** Използвайте `pd.DataFrame()` с трансформираните данни, за да създадете Pandas DataFrame.
- **Запис обратно в XML (ако е необходимо):** Ако трябва да запишете DataFrame обратно в по-сложен XML формат, ще трябва да използвате методите на избраната XML библиотека, за да генерирате XML структурата, като итерирате през редовете и колоните на DataFrame-а и създавате съответните елементи и атрибути.

3) **Примерен сценарий (комбиниране на `lxml` и Pandas):**

Да предположим, че имате XML файл със следната структура, където информацията е вложена и има атрибути:

```
<catalog>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
  </book>
</catalog>
```

Можете да използвате `lxml` за парсване и извличане на данни, а след това Pandas за създаване на DataFrame:

```
import pandas as pd
from lxml import etree

xml_content = '''
<catalog>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
  </book>
```

```
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
  </book>
</catalog>
'''

root = etree.fromstring(xml_content)
data = []
for book in root.xpath('//book'):
    category = book.get('category')
    title = book.xpath('./title/text()')[0]
    author = book.xpath('./author/text()')[0]
    year = book.xpath('./year/text()')[0]
    data.append({'category': category, 'title': title, 'author': author,
'year': year})

df = pd.DataFrame(data)
print(df)
```

За запис обратно в подобен XML формат, ще трябва да използвате `lxml` (или друга XML библиотека) за създаване на елементи и атрибути, като итерирате през редовете на DataFrame-а.

4) Съвети:

- Разберете добре структурата на вашия XML файл.
- Изберете XML библиотеката, която най-добре отговаря на вашите нужди по отношение на скорост, функционалност и лекота на използване.
- Използвайте XPath (ако е приложимо) за по-лесно навигиране и извличане на данни.
- Планирайте как ще трансформирате йерархичните XML данни в табличен вид и обратно.

Казус 1: Анализ на данни за поръчки от XML файл

Представете си, че получавате данни за поръчки от партньорска система във формат XML (`orders.xml`). XML файлът съдържа информация за всяка поръчка, включително ID на поръчката, дата, клиент и списък с поръчани артикули (име, количество, цена). Искате да извлечете тази информация и да я анализирате с Pandas.

Примерен XML файл (`orders.xml`):

```
<orders>
  <order order_id="1001" date="2025-05-02">
    <customer name="Иван Иванов"/>
    <items>
      <item name="Телевизор" quantity="1" price="550.00"/>
      <item name="Кабели" quantity="2" price="15.50"/>
    </items>
  </order>
  <order order_id="1002" date="2025-05-01">
```

```
    <customer name="Петър Петров"/>
    <items>
      <item name="Лаптоп" quantity="1" price="1200.00"/>
    </items>
  </order>
</orders>
```

Решение:

Използвайте `pd.read_xml()` с подходящ XPath израз, за да извлечете информацията. В този случай може да се наложи да прочетете данните на няколко етапа или да използвате по-сложен XPath за сплескване на структурата.

```
import pandas as pd

file_path = 'orders.xml'

# Първо, да прочетем основните данни за поръчките
orders_df = pd.read_xml(file_path, xpath='//order')
print("Основни данни за поръчките:\n", orders_df)

# След това може да се наложи да обработим вложените елементи <items>
отделно.
# Това може да изисква допълнително парсване с друга XML библиотека или по-
сложен XPath,
# който може да не е директно поддържан от read_xml за пълно сплескване.

# Пример за извличане на артикулите (може да изисква допълнителна
обработка):
items_data = []
import xml.etree.ElementTree as ET
tree = ET.parse(file_path)
root = tree.getroot()
for order in root.findall('order'):
    order_id = order.get('order_id')
    date = order.get('date')
    customer_name = order.find('customer').get('name')
    for item in order.find('items').findall('item'):
        item_name = item.get('name')
        quantity = item.get('quantity')
        price = item.get('price')
        items_data.append({'order_id': order_id, 'date': date,
'customer_name': customer_name,
                           'item_name': item_name, 'quantity': quantity,
'price': price})

df_items = pd.DataFrame(items_data)
print("\nДетайли за артикулите в поръчките:\n", df_items)
```

В този случай, поради вложената структура, директното използване на `pd.read_xml()` за пълното извличане на всички детайли може да е сложно и може да се наложи комбинация с други методи за парсване на XML.

Казус 2: Експортиране на конфигурационни данни в XML формат

След като сте обработили някакви конфигурационни данни в Pandas DataFrame, искате да ги запазите във XML файл (`config.xml`), който да бъде използван от друга система.

Решение:

Използвайте метода `df.to_xml()` за записване на DataFrame-а в XML формат. Можете да контролирате дали индексът да бъде включен и как да бъде форматиран XML изходът.

```
import pandas as pd

# Да предположим, че имаме DataFrame с конфигурационни настройки
config_df = pd.DataFrame({
    'параметър': ['таймаут', 'максимални_опити', 'лог_ниво'],
    'стойност': [30, 5, 'INFO']
})

file_path = 'config.xml'
config_df.to_xml(file_path, index=False, root_name='configuration',
row_name='setting')
print(f"Конфигурационните данни бяха записани в '{file_path}'.")
```

```
Съдържанието на config.xml ще бъде подобно на:

 <configuration>
   <setting>
     <параметър>таймаут</параметър>
     <стойност>30</стойност>
   </setting>
   <setting>
     <параметър>максимални_опити</параметър>
     <стойност>5</стойност>
   </setting>
   <setting>
     <параметър>лог_ниво</параметър>
     <стойност>INFO</стойност>
   </setting>
 </configuration>
```

Тук използваме параметрите root_name и row_name за по-добро структуриране на XML файла.

Казус 3: Четене на данни от XML файл с атрибути

Представете си, че имате XML файл (`sensors.xml`) с данни от сензори, където стойностите на сензорите са представени като атрибути на елементите.

Примерен XML файл (`sensors.xml`):

```xml
<sensors>
  <sensor id="S1" type="temperature" value="25.5"/>
  <sensor id="S2" type="humidity" value="60.2"/>
  <sensor id="S3" type="pressure" value="1012.3"/>
</sensors>
```

Решение:

Можете да използвате `pd.read_xml()` с XPath, който да извлича атрибутите.

```
import pandas as pd

file_path = 'sensors.xml'
df_sensors = pd.read_xml(file_path, xpath='//sensor', attrs_only=True)
print("Данни от сензори (само атрибути):\n", df_sensors)

# Ако искате да включите и текста на елементите (ако има такива),
# може да се наложи по-сложна обработка.
```

Тези казуси илюстрират как `pd.read_xml()` и `df.to_xml()` могат да се използват за работа с XML данни в Pandas. Важно е да се разбира структурата на XML файла и да се използват подходящи XPath изрази и параметри за правилното извличане и записване на данните. За по-сложни XML структури може да се наложи комбинирането им с други XML-специфични библиотеки, както обсъдихме по-рано.

Казус 4*: Анализ на данни за книги от XML каталог

Ситуация:

Получавате голям XML файл (`books.xml`), който представлява каталог на книги. Структурата на XML файла е следната:

- Кореновият елемент е `<catalog>`.
- Всеки елемент `<book>` представлява информация за една книга.
- Във всеки елемент `<book>` има различни под-елементи като `<title>`, `<author>`, `<genre>`, `<price>`, `<publish_date>`, `<description>`.

- Елементът `<book>` има и атрибут `id`.
- Елементът `<author>` може да има под-елементи `<first_name>` и `<last_name>`.
- Елементът `<price>` има атрибут `currency`.

Примерна структура на XML файла:

```
<?xml version="1.0"?>
<catalog>
  <book id="bk101">
    <author>
      <first_name>Gambardella</first_name>
      <last_name>Matthew</last_name>
    </author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price currency="USD">44.99</price>
    <publish_date>2000-10-01</publish_date>
    <description>An in-depth look at creating applications with
XML.</description>
  </book>
  <book id="bk102">
    <author>
      <first_name>Ralls</first_name>
      <last_name>Kim</last_name>
    </author>
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price currency="EUR">5.95</price>
    <publish_date>2000-12-16</publish_date>
    <description>A former architect battles corporate zombies.</description>
  </book>
  </catalog>
```

Задача:

1. Прочетете XML файла (`books.xml`) и го парсвайте, за да извлечете информацията за всяка книга, включително нейните под-елементи и атрибути.
2. Създайте Pandas DataFrame, където всяка книга е ред, а колоните са: `book_id`, `title`, `genre`, `price`, `price_currency`, `publish_date`, `description`, `author_first_name`, `author_last_name`.
3. Анализирайте данните:
    - Колко книги има от всеки жанр?
    - Каква е средната цена на книгите във всяка валута?
    - Кои са първите 3 най-скъпи книги (независимо от валутата)? За целта, трябва да конвертирате всички цени в една обща валута (например, USD, като използвате фиксирани курсове: 1 EUR = 1.10 USD).
    - Кои са авторите с най-много книги в каталога?
4. Създайте нов Pandas DataFrame, съдържащ само `book_id`, `title` и пълното име на автора (`author_first_name + ' ' + author_last_name`).
5. Запазете този новия DataFrame в CSV файл.

Решение:

```
import pandas as pd
import xml.etree.ElementTree as ET

# 1. Парсване на XML файла
tree = ET.parse('books.xml')
root = tree.getroot()

# Списък за съхранение на данните за всяка книга
books_data = []

# Итериране през всички елементи <book>
for book in root.findall('book'):
    book_id = book.get('id')
    title = book.find('title').text
    genre = book.find('genre').text
    price = float(book.find('price').text)
    price_currency = book.find('price').get('currency')
    publish_date = book.find('publish_date').text
    description = book.find('description').text
    author = book.find('author')
    first_name = author.find('first_name').text
    last_name = author.find('last_name').text

    books_data.append({
        'book_id': book_id,
        'title': title,
        'genre': genre,
        'price': price,
        'price_currency': price_currency,
        'publish_date': publish_date,
        'description': description,
        'author_first_name': first_name,
        'author_last_name': last_name
    })

# 2. Създаване на Pandas DataFrame
books_df = pd.DataFrame(books_data)
print("DataFrame с информация за книгите:")
print(books_df.head())

# 3. Анализ на данните
print("\nБрой книги по жанр:")
print(books_df['genre'].value_counts())

print("\nСредна цена по валута:")
print(books_df.groupby('price_currency')['price'].mean())

# Конвертиране на цените в USD
exchange_rates = {'USD': 1.00, 'EUR': 1.10}
```

```
books_df['price_usd'] = books_df.apply(lambda row: row['price'] *
exchange_rates[row['price_currency']], axis=1)

top_3_expensive = books_df.sort_values(by='price_usd',
ascending=False).head(3)
print("\nТоп 3 най-скъпи книги (в USD):")
print(top_3_expensive[['title', 'price_usd']])

print("\nАвтори с най-много книги:")
books_df['full_author_name'] = books_df['author_first_name'] + ' ' +
books_df['author_last_name']
print(books_df['full_author_name'].value_counts())

# 4. Създаване на нов DataFrame с ID, заглавие и пълно име на автора
author_title_df = books_df[['book_id', 'title', 'full_author_name']]
print("\nDataFrame с ID, заглавие и пълно име на автора:")
print(author_title_df.head())

# 5. Запазване на новия DataFrame в CSV файл
author_title_df.to_csv('books_authors.csv', index=False, encoding='utf-8')
print("\nDataFrame-ът с ID, заглавие и автори е запазен във
'books_authors.csv'")
```

Разбор на сложността:

- **Структуриран XML с вложени елементи и атрибути:** XML файлът има йерархична структура, която трябва да бъде обходена, за да се извлекат данните. Наличието на вложени елементи (`<author>`) и атрибути (`currency` в `<price>`) усложнява процеса на четене.
- **Преобразуване на XML в табличен формат:** Данните от XML файла трябва да бъдат преобразувани в плосък табличен формат, който е подходящ за Pandas DataFrame. Това изисква извличане на стойности от различни нива на XML структурата и присвояването им на съответните колони.
- **Анализ, включващ преобразуване на данни:** Анализът включва преобразуване на цените в обща валута, което изисква условна логика и използване на външни курсове.
- **Комбиниране на данни от различни XML елементи:** Създаването на колоната с пълното име на автора изисква комбиниране на стойности от два различни под-елемента (`<first_name>` и `<last_name>`).
- **Създаване на нов DataFrame с подмножество от данните:** Финалната стъпка включва създаване на нов DataFrame само с избрани колони.

Този казус демонстрира как Pandas може да се използва за работа със сложни XML данни, като се използва библиотеката `xml.etree.ElementTree` за парсване на XML и след това се създава DataFrame за анализ и по-нататъшна обработка. Той подчертава необходимостта от разбиране на структурата на XML файла, за да се извлекат правилно нужните данни.

Въпроси:

1. Коя е основната функция в Pandas за четене на XML данни?
2. Как `pd.read_xml()` преобразува XML структура в Pandas DataFrame? Как се определят колоните и редовете?
3. Обяснете ролята на параметъра `xpath` в `pd.read_xml()`. Как се използва XPath за селектиране на елементи от XML файла?
4. Как `pd.read_xml()` обработва атрибути на XML елементи? Как можете да укажете дали да се четат само атрибутите или и съдържанието на елементите?
5. Как `pd.read_xml()` се справя с вложени XML структури? В какви случаи може да се наложи допълнителна обработка или използване на други XML библиотеки?
6. Как може да прочетете XML данни от файл или от XML низ в Pandas?
7. Коя е основната функция в Pandas за записване на DataFrame в XML формат?
8. Как може да контролирате името на корена (`root_name`) и името на всеки ред (`row_name`) в генерирания XML файл с `df.to_xml()`?
9. Как може да контролирате дали индексът на DataFrame-а да бъде включен като елемент или атрибут в генерирания XML?
10. Как може да запишете DataFrame в XML файл с определено кодиране?
11. В какви ситуации е полезно да експортирате данни от Pandas в XML формат? Дайте примери.
12. Как `df.to_xml()` се справя с липсващи стойности (NaN) в DataFrame-а при запис?
13. Какви са някои от ограниченията на `pd.read_xml()` при работа със сложни XML структури?
14. В какви случаи може да е необходимо да комбинирате Pandas с други XML-специфични библиотеки като `lxml` или `xml.etree.ElementTree` за по-сложни задачи?

Задачи:

1. **Четене на XML с XPath:** Създайте XML файл `data.xml` със следната структура:

```xml
<data>
  <person id="1">
    <name>Alice</name>
    <age>25</age>
    <city>Sofia</city>
  </person>
  <person id="2">
    <name>Bob</name>
    <age>30</age>
    <city>Plovdiv</city>
  </person>
</data>
```

Прочетете само имената и възрастите на хората в DataFrame, използвайки подходящ XPath израз. Задайте 'id' като индекс.

2. **Четене на XML с атрибути:** Създайте XML файл `sensors.xml` със следната структура:

```xml
<sensors>
  <sensor id="S1" type="temperature" value="25.5"/>
  <sensor id="S2" type="humidity" value="60.2"/>
  <sensor id="S3" type="pressure" value="1012.3"/>
</sensors>
```

Прочетете данните в DataFrame, като използвате само атрибутите 'id', 'type' и 'value' като колони.

3. **Запис в XML с персонализирани имена:** Създайте DataFrame с колони 'Име', 'Възраст' и 'Град'. Запишете го в XML файл `output.xml` с име на корена 'хора' и име на всеки ред 'човек'. Не включвайте индекса.
4. **Работа с вложен XML (просто):** Създайте XML файл `books.xml`:

```xml
<books>
  <book title="Книга 1">
    <author>Автор 1</author>
  </book>
  <book title="Книга 2">
    <author>Автор 2</author>
  </book>
</books>
```

Прочетете заглавията и авторите в DataFrame, като използвате XPath.

5. **Запис с включен индекс като атрибут:** Създайте DataFrame с произволен индекс (не поредица от числа) и една колона с данни. Запишете го в XML файл `indexed_output.xml`, като включите индекса като атрибут на елемента за ред.
6. **Четене на XML с различни типове данни:** Създайте XML файл `mixed_data.xml`:

```xml
<items>
  <item name="Продукт А" quantity="10" price="25.99" in_stock="true"/>
  <item name="Продукт Б" quantity="5" price="120.50" in_stock="false"/>
</items>
```

Прочетете данните в DataFrame, като се опитате да позволите на Pandas автоматично да определи типовете данни (числа, булеви стойности). Изведете DataFrame-а и проверете типовете на колоните.

7. **Обработка на XML с повтарящи се елементи*:** Представете си XML файл, където информацията за даден обект е разпределена в няколко повтарящи се елемента с еднакви имена, но различни атрибути. Създайте примерен XML файл `repeating_elements.xml`:

```xml
<record>
  <field name="name" value="Alice"/>
  <field name="age" value="25"/>
  <field name="city" value="Sofia"/>
  <field name="interest" value="reading"/>
  <field name="interest" value="hiking"/>
</record>
```

Напишете код, който да прочете този XML файл и да го преобразува в DataFrame, където 'name', 'age', 'city' са отделни колони, а 'interest' е колона, съдържаща списък от интереси за всеки запис. (Тази задача може да изисква малко по-напреднало използване на XML парсване, вероятно в комбинация с `xml.etree.ElementTree` преди създаването на DataFrame).

8. **Записване на DataFrame с вложена структура в XML (симулирано)**:** Въпреки че `pd.to_xml()` не поддържа директно създаване на напълно вложени XML структури от DataFrame, представете си DataFrame, където една от колоните съдържа списъци или други структурирани данни. Създайте такъв DataFrame и напишете код, който да го преобразува в XML формат, където тази колона се представя като вложен XML елемент (може да се наложи и тук да се използва `xml.etree.ElementTree` за по-гъвкаво генериране на XML). Например:

```
import pandas as pd

data = {'name': ['Alice', 'Bob'],
        'interests': [['reading', 'hiking'], ['coding', 'gaming']]}
df = pd.DataFrame(data)
```

```
Напишете код, който да генерира XML като:
 <data>
   <person name="Alice">
     <interests>
       <interest>reading</interest>
       <interest>hiking</interest>
     </interests>
   </person>
   <person name="Bob">
     <interests>
       <interest>coding</interest>
       <interest>gaming</interest>
     </interests>
   </person>
 </data>
```

!!!ВНИМАНИЕ: Задача 7 и 8 в частност може да изискват комбинация от Pandas и други XML библиотеки за постигане на желаната структура.

IX. LaTeX: Запис в LaTeX формат (to_latex)

Pandas предоставя удобен начин за експортиране на DataFrame-и в LaTeX табличен формат, което е особено полезно при създаване на научни доклади, статии и други документи, където се изисква висококачествено представяне на таблични данни.

Основният метод за тази цел е `DataFrame.to_latex()`. Той генерира низов (string) представяне на DataFrame-а, форматиран като LaTeX таблица.

1. Основен синтаксис:

```
df.to_latex(buf=None, columns=None, col_space=None, header=True, index=True,
na_rep='NaN', formatters=None, float_format=None,
bold_rows=False,column_format=None, multirow=False, longtable=False,
escape=True, encoding=None, decimal='.', line_terminator='\n', **kwargs)
```

a) Основби параметри на `to_latex`

Нека разгледаме някои от най-важните параметри:

- **`buf`**: Може да бъде обект, подобен на файл (например, отворен файл), в който да се запише LaTeX изходът. Ако е `None` (по подразбиране), методът връща низов (string) резултат.
- **`columns`**: Списък от колони, които да бъдат включени в LaTeX таблицата. Ако е `None`, се използват всички колони.
- **`header`**: Булева стойност, която определя дали да се включи заглавният ред (имената на колоните) в LaTeX таблицата. По подразбиране е `True`.
- **`index`**: Булева стойност, която определя дали да се включи индексът на DataFrame-а като първа колона (или като ред, ако `multirow=True`). По подразбиране е `True`.
- **`na_rep`**: Низ, който да се използва за представяне на липсващи стойности (NaN). По подразбиране е `'NaN'`.
- **`formatters`**: Речник, където ключовете са имена на колони, а стойностите са функции за форматиране на стойностите в тези колони.
- **`float_format`**: Функция за форматиране на числа с плаваща запетая. Например, `'{:.2f}'.format`.
- **`bold_rows`**: Булева стойност, която определя дали имената на редовете (индекса) да бъдат в удебелен шрифт. По подразбиране е `False`.
- **`column_format`**: Низ, който задава формата на колоните в LaTeX (например, `'lcr'` за ляво, центрирано и дясно подравняване).
- **`multirow`**: Булева стойност, която определя дали мултииндекс редовете да бъдат форматирани с помощта на `multirow` командата на LaTeX. Това може да направи таблиците с мултииндекс по-компактни и четими.
- **`longtable`**: Булева стойност, която определя дали да се използва `longtable` средата на LaTeX, която е подходяща за дълги таблици, които могат да се простират на няколко страници. По подразбиране е `False`.
- **`escape`**: Булева стойност, която определя дали LaTeX специалните символи (като `&`, `%`, `$`, `#`, `_`, `{`, `}`) в данните да бъдат екранирани с обратна наклонена черта (`\`). По подразбиране е `True`.
- **`encoding`**: Кодирането, което да се използва при запис във файл (ако `buf` е файл).

б) Пример:

```
import pandas as pd

# Създаване на примерен DataFrame
data = {'Име': ['Алиса', 'Борис', 'Ваня'],
        'Възраст': [25, 30, 22],
        'Среден успех': [5.50, 4.80, 6.00]}
df = pd.DataFrame(data)

# Експортиране в LaTeX формат (като низ)
latex_table = df.to_latex(index=False, float_format='{:.2f}'.format)
print(latex_table)
```

Изходът ще бъде нещо подобно на:

```
\begin{tabular}{lrr}
\toprule
Име & Възраст & Среден успех \\
\midrule
Алиса & 25 & 5.50 \\
Борис & 30 & 4.80 \\
Ваня & 22 & 6.00 \\
\bottomrule
\end{tabular}
```

Можете да копирате този код и да го поставите във вашия LaTeX документ.

Запис във файл:

```
with open('table.tex', 'w', encoding='utf-8') as f:
    df.to_latex(f, index=False, float_format='{:.2f}'.format)
```

Това ще запише LaTeX кода директно във файла `table.tex`

2. Допълнителни аспекти на df.to_latex():

- **Персонализиране на заглавния ред (header):** Въпреки че параметърът `header=True` включва имената на колоните като заглавен ред, може да имате нужда от по-сложно оформление на заглавния ред, особено при мултииндекс колони. В такива случаи може да се наложи предварително да манипулирате имената на колоните на DataFrame-а или да добавите ръчно допълнителни LaTeX команди след генериране на основната таблица.
- **Използване на `formatters` за гъвкаво форматиране:** Параметърът `formatters` е изключително мощен. Той приема речник, в който ключовете са имената на колони, а стойностите са функции (или lambda функции), които се прилагат към всяка стойност в съответната колона преди да бъде записана в LaTeX. Това позволява много фин контрол върху представянето на данните, включително форматиране на числа, дати, низове и други типове данни.

```
import pandas as pd
import numpy as np

data = {'Цена': [10.5, np.nan, 20.3],
        'Дата': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03'])}
df = pd.DataFrame(data)

formatters = {'Цена': '${:.2f}'.format,
              'Дата': lambda x: x.strftime('%d.%m.%Y') if pd.notna(x) else
''}

latex_table = df.to_latex(index=False, formatters=formatters, na_rep='-')
print(latex_table)
```

В този пример цените се форматират с валутен знак и два десетични знака, датите се преобразуват във формат ДД.ММ.ГГГГ, а липсващите стойности се представят като "-".

- **Контрол на десетичния разделител:** В някои езици и научни области се използва запетая като десетичен разделител вместо точка. Параметърът `decimal` позволява да укажете кой символ да се използва като десетичен разделител в генерирания LaTeX код.

```
data = {'Стойност': [3.14159, 2.71828]}
df = pd.DataFrame(data)
latex_table_comma = df.to_latex(index=False, float_format='{:.3f}'.format,
decimal=',')
print(latex_table_comma)
```

Резултатът ще използва запетая като десетичен разделител.

- **Допълнителни LaTeX пакети:** В зависимост от сложността на таблицата, може да се наложи да включите допълнителни LaTeX пакети във вашия документ (например, `booktabs` за по-добри линии в таблицата, `siunitx` за форматиране на мерни единици и числа). `df.to_latex()` генерира стандартен LaTeX код за таблица, но вие имате свободата да го допълвате с команди от тези пакети ръчно след генерирането.

- **Работа с MultiIndex колони:** Когато DataFrame има MultiIndex колони, `to_latex()` се опитва да ги представи по разумен начин, използвайки `\multicolumn` команди. Параметърът `multirow` може да бъде полезен и тук за по-компактно представяне на нивата на индекса.

```
data = {'Група А': {'Показател 1': [1, 2], 'Показател 2': [3, 4]},
        'Група Б': {'Показател 1': [5, 6], 'Показател 2': [7, 8]}}
df_multi_col = pd.DataFrame(data)
latex_table_multi_col = df_multi_col.to_latex(escape=False) # escape=False
за по-лесно четене
print(latex_table_multi_col)
```

Резултатът ще включва `\multicolumn` команди за групиране на подколоните под основните колони.

Разбирането и използването на тези допълнителни възможности ще ви позволи да генерирате LaTeX таблици, които са не само коректно представени, но и стилистично съобразени с вашите нужди.

Казус 1: Представяне на резултати от статистически анализ в научен доклад

Представете си, че сте провели статистически анализ на данни за продажби и сте получили обобщени резултати, които искате да представите в научен доклад, написан с LaTeX. Резултатите включват средни стойности, стандартни отклонения и p-стойности за различни продуктови категории.

Решение:

```
import pandas as pd
import numpy as np

data = {'Категория': ['Електроника', 'Книги', 'Дрехи'],
        'Средни продажби': [150.50, 75.20, 90.80],
        'Стандартно отклонение': [25.30, 12.15, 18.70],
        'p-стойност': [0.01, 0.05, 0.12]}
df_results = pd.DataFrame(data)

# Експортиране в LaTeX формат за научен доклад
latex_output = df_results.to_latex(index=False,
                                   float_format=lambda x: f'{x:.2f}' if
isinstance(x, float) else x,
                                   caption='Обобщени резултати от анализ на
продажбите',
                                   label='tab:sales_analysis')

print(latex_output)

# Този LaTeX код може да бъде директно включен във вашия .tex файл.
# Командите \caption и \label ще помогнат за номериране и кръстосани
препратки.
```

В този казус използваме `float_format` за да гарантираме, че числата с плаваща запетая са представени с два десетични знака, което е често срещано в научните публикации. Параметрите `caption` и `label` добавят LaTeX команди за описание и етикетиране на таблицата, което е важно за правилното оформление на доклада.

Казус 2: Генериране на таблица с финансови показатели за годишен отчет

Вие сте финансов анализатор и трябва да генерирате таблица с ключови финансови показатели за годишния отчет на компанията. Тези показатели включват приходи, нетна печалба, EBITDA и маржове за последните няколко години.

Решение:

```
import pandas as pd

data = {'Година': [2020, 2021, 2022, 2023],
        'Приходи (млн. лв.)': [120.5, 135.2, 150.8, 165.1],
```

```
        'Нетна печалба (млн. лв.)': [15.2, 18.7, 22.5, 25.9],
        'EBITDA (млн. лв.)': [25.8, 30.1, 35.7, 40.2],
        'Марж на нетната печалба (%)': [12.6, 13.8, 14.9, 15.7]}
df_financials = pd.DataFrame(data)

# Форматиране на процентите
df_financials['Марж на нетната печалба (%)'] = df_financials['Марж на
нетната печалба (%)'].map('{:.1f}%'.format)

# Експортиране в LaTeX с подравняване и без индекс
latex_output = df_financials.to_latex(index=False,
                                      column_format='lrrrr',
                                      caption='Ключови финансови
показатели',
                                      label='tab:financial_performance')

print(latex_output)
```

Тук използваме `column_format` за да зададем подравняване на колоните (първата е ляво, останалите са дясно подравнени, което е често срещано за финансови данни). Процентите са форматирани предварително, за да включат символа `%`.

Казус 3: Създаване на таблица с резултати от експеримент с няколко групи

Вие сте изследовател и провеждате експеримент с три различни групи. Събрали сте данни за няколко измервания (например, средна стойност и стандартна грешка) за всяка група. Искате да представите тези резултати в таблица във вашата научна статия.

Решение:

```
import pandas as pd
import numpy as np

data = {'Група': ['Контролна', 'Експериментална 1', 'Експериментална 2'],
        'Средна стойност': [10.2, 15.7, 12.9],
        'Стандартна грешка': [0.5, 0.8, 0.6]}
df_experiment = pd.DataFrame(data)

# Експортиране с удебелен шрифт за имената на групите (индекса)
latex_output = df_experiment.set_index('Група').to_latex(bold_rows=True,

float_format='{:.1f}'.format,
                                                         caption='Резултати
от експеримент',

label='tab:experiment_results')
```

```
print(latex_output)
```

В този случай задаваме колоната 'Група' като индекс и използваме bold_rows=True, за да бъдат имената на групите (сега индекс) представени в удебелен шрифт в LaTeX таблицата, което може да подобри четимостта.

Тези казуси илюстрират как `df.to_latex()` може да се използва за създаване на готови за включване в LaTeX документи таблици с различно форматиране и структура, подходящи за различни практически нужди.

Казус 4*: Генериране на LaTeX таблица с резултати от статистически анализ на множество модели

Ситуация:

Провели сте статистически анализ, в който сте сравнили ефективността на няколко различни модела за прогнозиране върху различни набори от данни. Резултатите от анализа включват различни метрики (средна квадратична грешка - MSE, коефициент на детерминация – R^2, средна абсолютна грешка - MAE), както и информация за сложността на модела (брой параметри). Искате да представите тези резултати в добре форматирана LaTeX таблица за включване в научен доклад.

Данните са организирани в Pandas DataFrame, където всеки ред представлява резултатите за даден модел върху даден набор от данни.

Примерен Pandas DataFrame:

```
import pandas as pd
import numpy as np

data = {
    'Model': ['Linear Regression', 'Polynomial Regression (degree 2)',
'Random Forest', 'Gradient Boosting'],
    'Dataset': ['A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'],
    'MSE': [0.15, 0.12, 0.08, 0.07, 0.22, 0.18, 0.11, 0.10],
    'R-squared': [0.85, 0.88, 0.92, 0.93, 0.78, 0.82, 0.89, 0.90],
    'MAE': [0.25, 0.22, 0.18, 0.17, 0.30, 0.27, 0.21, 0.20],
    'Parameters': [3, 6, 100, 500] * 2
}
results_df = pd.DataFrame(data)
print(results_df)
```

Задача:

Генерирайте LaTeX код от `results_df`, който да включва:

- Заглавие на таблицата (`\caption`).
- Етикетиране на таблицата (`\label`).

- Използване на среда `tabular` с подходящо подравняване на колоните.
- Форматиране на метриките (MSE, R2, MAE) до два знака след десетичната запетая.
- Представяне на R2 като R2 в LaTeX.
- Групиране на резултатите по 'Dataset' с добавяне на многоредови клетки (`\multirow`) за името на набора от данни.
- Добавяне на хоризонтални линии (`\hline`) за по-добра структура на таблицата.
- Включване на информация за броя на параметрите.

Решение:

```
import pandas as pd
import numpy as np

data = {
    'Model': ['Linear Regression', 'Polynomial Regression (degree 2)',
'Random Forest', 'Gradient Boosting',
              'Linear Regression', 'Polynomial Regression (degree 2)',
'Random Forest', 'Gradient Boosting'],
    'Dataset': ['A'] * 4 + ['B'] * 4,
    'MSE': [0.15, 0.12, 0.08, 0.07, 0.22, 0.18, 0.11, 0.10],
    'R-squared': [0.85, 0.88, 0.92, 0.93, 0.78, 0.82, 0.89, 0.90],
    'MAE': [0.25, 0.22, 0.18, 0.17, 0.30, 0.27, 0.21, 0.20],
    'Parameters': [3, 6, 100, 500] * 2
}
results_df = pd.DataFrame(data)

latex_code = """
\\documentclass{article}
\\usepackage{multirow}
\\usepackage{amsmath}

\\begin{document}

\\begin{table}[h!]
    \\centering
    \\caption{Сравнение на ефективността на модели за прогнозиране}
    \\label{tab:model_comparison}
    \\begin{tabular}{l|c|ccc|c}
        \\hline
        \\multirow{4}{*}{\\textbf{Модел}} & \\multirow{4}{*}{\\textbf{Набор
от данни}} & \\multicolumn{3}{c|}{\\textbf{Метрики за ефективност}} &
\\multirow{4}{*}{\\textbf{Брой параметри}} \\\\
        \\cline{3-5}
        & & \\textbf{MSE} & \\textbf{$R^2$} & \\textbf{MAE} & \\\\
        \\hline
"""

grouped = results_df.groupby('Dataset')
first_row = True
for dataset, group in grouped:
    rowspan = len(group)
    for index, row in group.iterrows():
```

```
        if first_row:
            latex_code += f"
\\multirow{{{rowspan}}}{{*}}{{{dataset}}} & "
            first_row = False
        else:
            latex_code += "        & "
        latex_code += f"{row['Model']} & {row['MSE']:.2f} & ${row['R-
squared']:.2f}$ & {row['MAE']:.2f} & {row['Parameters']} \\\\ \n"
    latex_code += "        \\hline\n"
    first_row = True

latex_code += """
    \\end{tabular}
\\end{table}

\\end{document}
"""

# Записване на LaTeX кода във файл
with open('model_comparison_table.tex', 'w') as f:
    f.write(latex_code)

print("LaTeX кодът за таблицата е генериран и записан във
'model_comparison_table.tex'")
```

Разбор на сложността:

- **Групиране на данни за многоредови клетки:** Необходимо е да се групират редовете по 'Dataset' и да се изчисли `rowspan` за командата `\multirow`, за да се покаже името на набора от данни само веднъж за всяка група.
- **Включване на математически формули:** Символът R2 трябва да бъде екраниран и поставен в математически режим (`$R^2$`) в LaTeX.
- **Персонализирано форматиране:** Метриките трябва да бъдат форматирани до определен брой знаци след десетичната запетая.
- **Създаване на сложна структура на таблицата:** Таблицата изисква многоредови заглавия (`\multirow`), хоризонтални линии (`\hline`) и вертикални разделители (`|`) за ясна структура.
- **Динамично генериране на LaTeX код:** Кодът се генерира динамично въз основа на съдържанието на DataFrame-а, което изисква итерация през редовете и групите.
- **Комбиниране на текст и данни:** LaTeX кодът съдържа както статичен текст (команди, заглавия), така и динамично вмъкнати данни от DataFrame-а.

Този казус показва как `to_latex()` може да бъде използван за създаване на сложни и добре форматирани таблици за научни публикации, като се контролират различни аспекти на LaTeX кода. Въпреки че самата функция `to_latex()` може да не поддържа директно всички тези сложни форматирания, генерираният от нея базов код може да бъде допълнително манипулиран, както е показано в решението, за постигане на желания вид на таблицата. В реални сценарии, може да се наложи по-фино контролиране на генерирания LaTeX код.

Въпроси:

1. Обяснете как параметрите `caption` и `label` в `df.to_latex()` се отразяват на генерирания LaTeX код и защо са полезни при създаване на научни документи.
2. Как можете да използвате параметъра `formatters` за да приложите специфично форматиране към различни колони на DataFrame при експортиране в LaTeX? Дайте пример за форматиране на валутна колона.
3. В какъв случай бихте използвали параметъра `decimal=','` при експортиране в LaTeX и какъв е ефектът му?
4. Ако имате DataFrame с MultiIndex колони, как `df.to_latex()` се опитва да ги представи в LaTeX и какъв ефект има параметърът `multirow` в този контекст?
5. Представете си, че искате да експортирате DataFrame с български символи в LaTeX. Какво трябва да направите, за да се уверите, че символите се показват правилно в крайния PDF документ? (Hint: свързано е с кодирането и LaTeX пакетите).

Задачи:

1. **Експортиране с caption и label:** Създайте DataFrame с данни за продажби (продукт, количество, цена). Експортирайте го в LaTeX, като добавите подходящ caption и label към таблицата.
2. **Форматиране на различни типове колони:** Създайте DataFrame с колони, съдържащи числа с плаваща запетая, цели числа и дати. Експортирайте го в LaTeX, като форматирате числата до два десетични знака и датите във формат ДД.ММ.ГГГГ.
3. **Експортиране с десетична запетая:** Създайте DataFrame с числа с плаваща запетая. Експортирайте го в LaTeX, като използвате запетая като десетичен разделител.
4. **Експортиране с MultiIndex колони:** Създайте DataFrame с MultiIndex колони (например, нива 'Статистика' и поднива 'Средно', 'Стандартно отклонение'). Експортирайте го в LaTeX и наблюдавайте как се представя структурата. Опитайте да използвате `escape=False` за по-лесно четене на кода.
5. **Запис в LaTeX файл с UTF-8 кодиране:** Създайте DataFrame, който съдържа български символи в една от колоните. Запишете го в LaTeX файл, като изрично укажете кодиране 'utf-8'. След това създайте прост LaTeX документ, който включва тази таблица, компилирайте го и се уверете, че българските символи се показват правилно. (За тази задача ще ви е необходима работеща LaTeX инсталация).

X. HDFStore: PyTables (HDF5): (HDFStore, read_hdf, to_hdf) - *Изисква tables*

HDF5 (Hierarchical Data Format version 5) е високоефективен формат за съхранение на големи количества числови данни. `HDFStore` в Pandas е клас, който позволява да съхранявате Pandas обекти (като DataFrame и Series) в HDF5 файлове, използвайки библиотеката PyTables. Това е особено полезно за работа с големи набори от данни, които не се побират в паметта, или когато искате бърз и ефективен начин за достъп до части от данните.

1. Основни методи и клас:

- **`pandas.HDFStore(path, mode='r', complevel=None, complib=None, fletcher32=False, dropna=None)`**: Конструктор за създаване или отваряне на HDF5 файл.
  - `path`: Пътят до HDF5 файла.
  - `mode`: Режим на отваряне ('r' - четене, 'w' - запис/изтриване на съществуващ, 'a' - запис/добавяне към съществуващ).
  - `complevel`: Ниво на компресия (0-9, където по-високите стойности означават по-голяма компресия и по-бавно записване).
  - `complib`: Библиотека за компресия ('zlib', 'lzo', 'bzip2').
  - `fletcher32`: Булева стойност, която определя дали да се използва контролна сума за откриване на грешки.
- **`pandas.read_hdf(path_or_buf, key=None, mode='r', **kwargs)`**: Функция за четене на Pandas обект от HDF5 файл.
  - `path_or_buf`: Пътят до HDF5 файла или вече отворен `HDFStore` обект.
  - `key`: Името (ключа) на обекта, който искате да прочетете от файла. Ако е `None`, се връща речник с всички обекти.
  - `mode`: Режим на отваряне (ако е подаден път, а не `HDFStore` обект).
- **`DataFrame.to_hdf(path_or_buf, key, mode='a', complevel=None, complib=None, fletcher32=False, format=None, append=False, data_columns=None, errors='strict', **kwargs)`** и **`Series.to_hdf(...)`**: Методи за записване на DataFrame или Series в HDF5 файл.
  - `path_or_buf`: Пътят до HDF5 файла или вече отворен `HDFStore` обект.
  - `key`: Името (ключа), под което обектът ще бъде записан във файла.
  - `mode`: Режим на отваряне (ако е подаден път, а не `HDFStore` обект).
  - `append`: Булева стойност, която определя дали да се добавят данните към съществуващ обект с този ключ (ако е възможно).
  - `data_columns`: Списък от колони, които да бъдат третирани като "data columns" за по-ефективни заявки.

а) Пример за запис и четене:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
df = pd.DataFrame({'A': np.random.rand(1000),
                   'B': np.random.randint(0, 100, 1000),
```

```
                    'C': pd.Categorical(['a', 'b', 'c'] * (1000 // 3 +
1)[:1000])})

# Запис на DataFrame в HDF5 файл
df.to_hdf('my_data.h5', key='my_dataframe', mode='w', complevel=5,
complib='zlib')

# Четене на DataFrame от HDF5 файл
loaded_df = pd.read_hdf('my_data.h5', key='my_dataframe')

# Проверка дали данните са еднакви
print(df.equals(loaded_df))

# Можете да работите с HDFStore обект директно
with pd.HDFStore('my_data.h5', mode='a') as store:
    store['another_dataframe'] = df[:500]
    series_data = pd.Series(np.arange(10))
    store['my_series'] = series_data
    print(store.keys()) # Визуализиране на ключовете във файла
    loaded_series = store['my_series']
    print(loaded_series)
```

б) Предимства на HDFStore:

- **Ефективност:** Бързо четене и запис на големи набори от данни.
- **Компресия:** Поддържа различни нива и библиотеки за компресия, което намалява размера на файла на диска.
- **Селективно четене:** Възможност за четене само на определени редове или колони, ако DataFrame-ът е записан с `data_columns`.
- **Йерархична структура:** Позволява организиране на данните под различни ключове в един файл, подобно на файлова система.

в) Кога да използваме HDFStore:

- Когато работите с големи DataFrame-и, които могат да надхвърлят наличната RAM.
- Когато се нуждаете от бърз достъп до данни, съхранявани на диска.
- Когато искате да съхранявате множество Pandas обекти в един файл по организиран начин.

!!!Важно: За да работите с HDFStore, трябва да имате инсталирана библиотеката `pytables`. Можете да я инсталирате с `pip install pytables`.

2. Допълнително

а) Работа с `data_columns` за ефективни заявки:

Когато записвате DataFrame в HDF5 файл, можете да укажете кои колони да бъдат третирани като "data columns". Това позволява да извършвате заявки (queries) към файла, за да четете само редовете, които

отговарят на определени условия, без да зареждате целия DataFrame в паметта. Това може значително да ускори процеса при работа с големи файлове.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A': np.random.rand(1000),
                   'B': np.random.randint(0, 100, 1000),
                   'C': pd.Categorical((['a', 'b', 'c'] * ((1000 // 3) +
1))[:1000])})

# Записване с 'B' и 'C' като data_columns
df.to_hdf('my_data_query.h5', key='my_dataframe', mode='w',
data_columns=['B', 'C'])

# Четене само на редовете, където 'B' е по-голямо от 50
selected_df = pd.read_hdf('my_data_query.h5', key='my_dataframe', where='B >
50')
print(f"Брой редове, където B > 50: {len(selected_df)}")

# Четене само на редовете, където 'C' е 'a' или 'b'
selected_df_categorical = pd.read_hdf('my_data_query.h5',
key='my_dataframe', where="C in ['a', 'b']")
print(f"Брой редове, където C е 'a' или 'b':
{len(selected_df_categorical)}")
```

б) Различни формати за съхранение (`format`):

При записване с `to_hdf`, параметърът `format` може да бъде `'table'` (по-ефективен за заявки и добавяне) или `'fixed'` (по-бърз за четене и запис, но не поддържа добавяне или `data_columns`). По подразбиране е `'table'`. Изборът зависи от вашите нужди за производителност и функционалност.

```
# Записване във формат 'fixed'
df.to_hdf('my_data_fixed.h5', key='my_dataframe', mode='w', format='fixed')

# Четене от файл, записан във формат 'fixed'
loaded_fixed_df = pd.read_hdf('my_data_fixed.h5', key='my_dataframe')
```

в) Добавяне на данни към съществуващ обект (`append=True`):

Ако използвате формат `'table'`, можете да добавяте нови редове към вече съществуващ DataFrame във файла, като използвате `append=True`. Това е полезно за постепенно записване на големи набори от данни.

```
df_part1 = df[:500]
df_part2 = df[500:]

df_part1.to_hdf('my_appended_data.h5', key='my_dataframe', mode='w',
format='table', data_columns=['B'])
df_part2.to_hdf('my_appended_data.h5', key='my_dataframe', mode='a',
format='table', append=True)

loaded_appended_df = pd.read_hdf('my_appended_data.h5', key='my_dataframe')
```

```
print(f"Общ брой редове след добавяне: {len(loaded_appended_df)}")
```

3. Изключения и потенциални проблеми:

- **`FileNotFoundError`**: Възниква, ако се опитате да прочетете от файл, който не съществува.
- **`KeyError`**: Възниква, ако се опитате да прочетете обект по ключ, който не съществува в HDF5 файла.
- **`TypeError`**: Може да възникне при опит за записване на неподдържан тип обект в HDF5 файла. `HDFStore` е оптимизиран за Pandas DataFrame и Series.
- **Съвместимост на версии:** HDF5 форматът е сравнително стабилен, но е възможно да има проблеми със съвместимостта между много стари и много нови версии на PyTables. Препоръчително е да поддържате актуална версия на библиотеката.
- **Заключване на файла:** Когато HDF5 файл е отворен за писане от един процес, опитът за запис в него от друг процес може да доведе до грешки или повреди във файла. Препоръчително е да се избягва едновременен запис от множество процеси.
- **Производителност:** Въпреки че HDFStore е бърз, производителността може да варира в зависимост от размера на данните, нивото на компресия, формата на съхранение и сложността на заявките. Експериментирането с различни параметри може да помогне за оптимизиране на производителността.

4. Най-добри практики:

- Избирайте подходящо ниво и библиотека за компресия в зависимост от баланса между размер на файла и скорост на запис/четене.
- Използвайте `data_columns` за колони, по които често ще филтрирате или търсите, за да ускорите заявките.
- Избирайте формат на съхранение (`'table'` или `'fixed'`) според нуждите си за добавяне и заявки.
- Управлявайте отварянето и затварянето на `HDFStore` обекти, особено при работа с контекстния мениджър (`with pd.HDFStore(...) as store:`), за да гарантирате правилното затваряне на файла.
- При работа с големи файлове, планирайте внимателно структурата на данните и ключовете, за да осигурите лесен и ефективен достъп.

Казус 1: Обработка на големи времеви редици от сензори

Ситуация: Имате голям обем от данни от различни сензори, които измерват параметри през определени интервали от време. Данните се събират непрекъснато и трябва да бъдат анализирани периодично. Размерът на данните за всяка сесия на анализ може да надхвърли наличната RAM.

Решение:

1) **Съхранение:** Използвайте `HDFStore` за съхранение на данните. Записвайте данните от всеки сензор като отделен DataFrame под уникален ключ (например, името на сензора). Използвайте компресия (например, `complevel=5`, `complib='zlib'`) за намаляване на размера на файла. Тъй като времевите редици често се филтрират по времеви интервали, направете колоната с времето като `data_column`.

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

```
# Симулиране на данни от сензор
dates = pd.date_range('2023-01-01', periods=100000, freq='S')
sensor_data = pd.DataFrame({'timestamp': dates, 'temperature':
np.random.rand(100000) * 30 + 10,
                            'humidity': np.random.rand(100000) * 80 + 20})
sensor_data.set_index('timestamp', inplace=True)

# Създаване или отваряне на HDFStore
with pd.HDFStore('sensor_data.h5', mode='a', complevel=5, complib='zlib') as
store:
    store['sensor_1'] = sensor_data
```

1) **Анализ:** Когато е необходимо да се анализира определен период от време, прочетете само съответната част от данните, използвайки заявка (`where`) по колоната 'timestamp' (тъй като е `data_column`).

```
start_date = '2023-01-05 10:00:00'
end_date = '2023-01-05 11:00:00'

with pd.HDFStore('sensor_data.h5', mode='r') as store:
    query = f"timestamp >= '{start_date}' and timestamp <= '{end_date}'"
    hourly_data = store.select('sensor_1', where=query)
    print(f"Брой записи за избрания час: {len(hourly_data)}")
    # Извършете анализ на hourly_data
```

Казус 2: Кеширане на резултати от сложни изчисления

Ситуация: Извършвате сложни и времеемки изчисления върху голям набор от данни. Искате да кеширате резултатите, за да не се налага да ги преизчислявате всеки път, когато са необходими.

Решение:

1) **Кеширане:** След като изчисленията приключат, запишете резултатите (които могат да бъдат DataFrame-и или други Pandas обекти) в `HDFStore` под описателни ключове.

```
# Симулиране на сложни изчисления върху голям DataFrame (df_large)
df_large = pd.DataFrame(np.random.rand(1000000, 5), columns=list('ABCDE'))
result_1 = df_large.groupby('A').mean()
result_2 = df_large['B'].rolling(window=1000).std()

with pd.HDFStore('cached_results.h5', mode='w') as store:
    store['mean_by_group'] = result_1
    store['rolling_std'] = result_2
```

2) **Използване на кеша:** Когато имате нужда от резултатите, първо проверете дали съществуват в `HDFStore`. Ако да, прочетете ги оттам. В противен случай, извършете изчисленията и след това ги кеширайте.

```
def get_mean_by_group():
```

```
    try:
        with pd.HDFStore('cached_results.h5', mode='r') as store:
            return store['mean_by_group']
    except KeyError:
        df_large = pd.DataFrame(np.random.rand(1000000, 5),
columns=list('ABCDE'))
        result = df_large.groupby('A').mean()
        with pd.HDFStore('cached_results.h5', mode='a') as store:
            store['mean_by_group'] = result
        return result

mean_results = get_mean_by_group()
print(mean_results.head())
```

Казус 3: Съхранение на междинни резултати при сложен анализ

Ситуация: Извършвате многостъпков анализ на голям набор от данни. Искате да запазвате междинните резултати след всеки етап, за да можете лесно да се върнете към тях, ако възникне грешка или ако искате да експериментирате с различни следващи стъпки.

Решение:

1) **Междинно съхранение:** След всеки важен етап от анализа, запишете получения DataFrame в `HDFStore` под ключ, описващ етапа.

```
# Зареждане на първоначални данни (df_initial)
df_initial = pd.DataFrame(np.random.rand(500000, 10), columns=[f'col_{i}'
for i in range(10)])

with pd.HDFStore('analysis_steps.h5', mode='w') as store:
    store['step_1_raw_data'] = df_initial

    # Първи етап на обработка
    df_processed_1 = df_initial[df_initial['col_0'] > 0.5]
    store['step_2_filtered_data'] = df_processed_1

    # Втори етап на обработка
    df_aggregated = df_processed_1.groupby('col_1').mean()
    store['step_3_aggregated_data'] = df_aggregated
```

2) **Възстановяване на междинни резултати:** Ако е необходимо, можете лесно да прочетете DataFrame-а от всеки междинен етап, използвайки съответния ключ.

```
with pd.HDFStore('analysis_steps.h5', mode='r') as store:
    filtered_data = store['step_2_filtered_data']
    print(f"Размер на данните след филтриране: {len(filtered_data)}")
    # Продължете анализа от тази точка или я изследвайте
```

Казус 4*: Управление и анализ на исторически котировки на акции

Ситуация:

Разработвате система за анализ на финансови пазари. Получавате големи обеми от исторически котировки на акции за множество компании. Данните за всяка акция включват времеви серии от цени (Open, High, Low, Close), обем на търговия и други индикатори. Вместо да зареждате всички данни в паметта, искате да използвате `HDFStore` за ефективно съхранение и достъп до тези данни на диск.

Имате следните изисквания:

1. Съхранявайте данните за всяка акция в отделен DataFrame.
2. Записвайте DataFrame-ите в HDF5 файл (`stock_data.h5`), като използвате името на тикера на акцията като ключ (`key`) в `HDFStore`.
3. Добавете метаданни към всеки записан DataFrame, като например източника на данните и датата на последното обновяване.
4. Прочетете селективно данните за определен набор от акции от HDF5 файла.
5. Прочетете само част от данните за дадена акция (например, за определен период от време).
6. Извършете анализ върху прочетените данни (например, изчислете средната затваряща цена за определен период).
7. Добавете нов DataFrame с данни за друга акция към съществуващия HDF5 файл.
8. Изтрийте данните за определена акция от HDF5 файла.

Решение:

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

# 1. Генериране на примерни данни за няколко акции
tickers = ['AAPL', 'GOOG', 'MSFT', 'AMZN']
start_date = datetime.datetime(2023, 1, 1)
end_date = datetime.datetime(2024, 12, 31)
dates = pd.date_range(start=start_date, end=end_date, freq='B') # Работни
дни

stock_data = {}
for ticker in tickers:
    data = {
        'Open': np.random.rand(len(dates)) * 100 + ticker_index * 50,
        'High': np.random.rand(len(dates)) * 100 + 5 + ticker_index * 50,
        'Low': np.random.rand(len(dates)) * 100 - 5 + ticker_index * 50,
        'Close': np.random.rand(len(dates)) * 100 + ticker_index * 50,
        'Volume': np.random.randint(100000, 1000000, len(dates))
    }
    stock_data[ticker] = pd.DataFrame(data, index=dates)
    ticker_index = tickers.index(ticker) + 1

# 2. Записване на DataFrame-ите в HDF5 файл
```

```
h5_file = 'stock_data.h5'
with pd.HDFStore(h5_file, mode='w') as store:
    for ticker, df in stock_data.items():
        # 3. Добавяне на метаданни
        store.put(ticker, df, metadata={'source': 'Example Data Generator',
'last_updated': datetime.datetime.now().isoformat()})
        print(f"Данни за {ticker} записани в HDF5 файла.")

# 4. Четене на данни за определени акции
tickers_to_read = ['AAPL', 'MSFT']
loaded_data = {}
with pd.HDFStore(h5_file, mode='r') as store:
    for ticker in tickers_to_read:
        if ticker in store:
            loaded_data[ticker] = store.get(ticker)
            metadata = store.get_storer(ticker).attrs.metadata
            print(f"\nДанни за {ticker} прочетени. Метаданни: {metadata}")
        else:
            print(f"\nНяма данни за {ticker} в HDF5 файла.")

# 5. Четене на част от данните за определена акция (по дата)
ticker_to_read_partial = 'AAPL'
start_date_partial = '2024-01-01'
end_date_partial = '2024-06-30'

with pd.HDFStore(h5_file, mode='r') as store:
    if ticker_to_read_partial in store:
        partial_df = store.select(ticker_to_read_partial, where=f"index >=
'{start_date_partial}' and index <= '{end_date_partial}'")
        print(f"\nЧастични данни за {ticker_to_read_partial}
({start_date_partial} до {end_date_partial}):")
        print(partial_df.head())
    else:
        print(f"\nНяма данни за {ticker_to_read_partial} в HDF5 файла.")

# 6. Извършване на анализ върху прочетените данни
if 'AAPL' in loaded_data:
    aapl_data = loaded_data['AAPL']
    mean_closing_price_2024 = aapl_data['Close'][aapl_data.index.year ==
2024].mean()
    print(f"\nСредна затваряща цена на AAPL през 2024 г.:
{mean_closing_price_2024:.2f}")

# 7. Добавяне на нови данни за друга акция
new_ticker = 'TSLA'
new_data = {
    'Open': np.random.rand(len(dates)) * 200,
    'High': np.random.rand(len(dates)) * 200 + 10,
    'Low': np.random.rand(len(dates)) * 200 - 10,
    'Close': np.random.rand(len(dates)) * 200,
    'Volume': np.random.randint(200000, 2000000, len(dates))
}
new_df = pd.DataFrame(new_data, index=dates)
```

```
with pd.HDFStore(h5_file, mode='a') as store:
    store.put(new_ticker, new_df, metadata={'source': 'Example Data
Generator', 'last_updated': datetime.datetime.now().isoformat()})
    print(f"\nДанни за {new_ticker} добавени към HDF5 файла.")

# Проверка дали е добавена
with pd.HDFStore(h5_file, mode='r') as store:
    print("\nКлючове в HDF5 файла след добавяне на TSLA:", store.keys())

# 8. Изтриване на данни за определена акция
ticker_to_delete = 'GOOG'
with pd.HDFStore(h5_file, mode='a') as store:
    if ticker_to_delete in store:
        store.remove(ticker_to_delete)
        print(f"\nДанни за {ticker_to_delete} изтрити от HDF5 файла.")
    else:
        print(f"\nНяма данни за {ticker_to_delete} в HDF5 файла.")

# Проверка след изтриване
with pd.HDFStore(h5_file, mode='r') as store:
    print("\nКлючове в HDF5 файла след изтриване на GOOG:", store.keys())
```

Разбор на сложността:

- **Съхранение на множество DataFrame-и:** Данните за всяка акция се управляват като отделен DataFrame и се съхраняват под уникален ключ в HDF5 файла.
- **Добавяне и четене на метаданни:** Демонстрира се как да се асоциира допълнителна информация с всеки DataFrame, което е полезно за проследяване на произхода и актуализацията на данните.
- **Селективно четене на данни:** Показва се как да се четат само определени DataFrame-и или части от DataFrame (използвайки `where` клаузата за времеви серии), без да се зареждат всички данни в паметта.
- **Работа с времеви серии:** Казусът е базиран на времеви данни, което е често срещано приложение на `HDFStore` за големи набори от данни.
- **Добавяне и изтриване на данни:** Илюстрира се как да се модифицира съществуващ HDF5 файл чрез добавяне на нови данни и изтриване на съществуващи.
- **Ефективно управление на дисково пространство и памет:** `HDFStore` позволява работа с данни, които могат да бъдат по-големи от наличната RAM, като данните се четат от диска при нужда.

Този сложен казус демонстрира основните предимства на използването на `HDFStore` за управление на големи набори от данни, като ефективно съхранение, бърз селективен достъп и възможност за работа с данни, които не се побират в паметта. Той е особено полезен за приложения, които работят с големи обеми от времеви серии или други структурирани данни.

Въпроси:

1. Обяснете каква е основната цел на използването на `HDFStore` в Pandas. В какви ситуации е особено полезно?
2. Каква е връзката между `HDFStore` и библиотеката PyTables (HDF5)?
3. Опишете процеса на създаване или отваряне на HDF5 файл с помощта на `pd.HDFStore()`. Кои са някои важни параметри при създаването?
4. Как се записват и четат Pandas DataFrame-и и Series от HDF5 файл, използвайки съответно `to_hdf()` и `read_hdf()`?
5. Каква е ролята на параметъра `key` при запис и четене на обекти от `HDFStore`? Как трябва да се избират ключовете?
6. Обяснете концепцията за "data columns" при записване в HDF5 файл. Как се указват те и каква е ползата от тях при четене?
7. Как можете да четете само подмножество от редове от голям HDF5 файл въз основа на условия, без да зареждате целия файл в паметта? Дайте пример.
8. Сравнете и контрастирайте стойностите `'table'` и `'fixed'` за параметъра `format` при записване с `to_hdf()`. В кои ситуации е по-подходящ всеки от тях?
9. Как може да добавите нови данни към вече съществуващ DataFrame в HDF5 файл? Кои условия трябва да бъдат изпълнени, за да е възможно това?
10. Избройте някои от често срещаните изключения, които могат да възникнат при работа с `HDFStore`, и обяснете възможните причини за тях.
11. Какви са някои от най-добрите практики, които трябва да следвате при работа с `HDFStore`, особено когато става въпрос за производителност и управление на големи данни?
12. Какво трябва да имате предвид по отношение на съвместимостта на версиите при работа с HDF5 файлове, създадени с различни версии на PyTables?
13. Защо е важно да управлявате правилно отварянето и затварянето на `HDFStore` обекти? Как контекстният мениджър (`with ... as ...:`) помага в това?
14. В какви сценарии би било по-подходящо да използвате `HDFStore` вместо други методи за съхранение на данни като CSV или pickle?

Задачи:

1. **Запис и четене с data_columns:** Създайте голям DataFrame (например, с 10000 реда и няколко колони, включително колона с дати и колона с категорийни данни). Запишете го в HDF5 файл, като укажете две от колоните като `data_columns`. След това прочетете само редовете, където дадена `data_column` отговаря на определено условие.
2. **Използване на различни формати:** Създайте малък DataFrame и го запишете в два различни HDF5 файла, веднъж с `format='table'` и веднъж с `format='fixed'`. След това прочетете и двата файла и сравнете времето за четене (можете да използвате `%timeit` в Jupyter Notebook или `time` модула в Python).
3. **Добавяне на данни:** Създайте DataFrame и го запишете в HDF5 файл (използвайте `format='table'`). След това създайте друг DataFrame със същите колони, но с нови данни, и го добавете към съществуващия обект в HDF5 файла. Прочетете целия обект, за да се уверите, че данните са добавени правилно.
4. **Работа с множество обекти:** Създайте два различни DataFrame-а и една Series. Запишете ги в един HDF5 файл под различни ключове. След това прочетете обратно всеки от обектите, използвайки техните ключове.

5. **Изследване на компресия:** Създайте голям DataFrame и го запишете в HDF5 файл няколко пъти с различни нива на компресия (`complevel`) и различни библиотеки за компресия (`complib`). Наблюдавайте как се променя размерът на файла на диска. (Тази задача може да изисква инсталиране на допълнителни библиотеки за компресия като `lzo` или `bzip2`).

XI. Feather формат: (read_feather, to_feather)

Feather е бърз, лек и лесен за използване формат за двоично съхранение на DataFrame-и на Apache Arrow. Той е разработен с цел да осигури висока скорост при запис и четене на DataFrame-и между различни процеси и езици за програмиране (особено Python и R). Feather е оптимизиран за in-memory аналитични работни процеси.

1. Основни функции:

а) Основни параметри на `read_feather`

- **`pandas.read_feather(path, columns=None, use_threads=True, storage_options=None)`**: Функция за четене на DataFrame от Feather файл.
  - `path`: Пътят до Feather файла (може да бъде и обект, подобен на файл).
  - `columns`: Списък от колони, които да бъдат прочетени. Ако е `None`, се четат всички колони.
  - `use_threads`: Булева стойност, която определя дали да се използват множество нишки за четене (ако е възможно).
  - `storage_options`: Допълнителни опции за системи за съхранение, като AWS S3.
- **`DataFrame.to_feather(path, compression='uncompressed', compression_level=None, chunksize=None, storage_options=None)`**: Метод за записване на DataFrame във Feather файл.
  - `path`: Пътят до Feather файла (може да бъде и обект, подобен на файл).
  - `compression`: Тип на компресия ('uncompressed', 'lz4', 'zstd'). 'lz4' обикновено е много бърз, а 'zstd' предлага по-добра компресия при разумна скорост.
  - `compression_level`: Ниво на компресия (зависи от избраната компресия).
  - `chunksize`: Размер на парчетата за записване (за по-големи DataFrame-и).
  - `storage_options`: Допълнителни опции за системи за съхранение, като AWS S3.

б) Пример за запис и четене:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
df = pd.DataFrame({'A': np.random.rand(1000),
                   'B': np.random.randint(0, 100, 1000),
                   'C': pd.Categorical(['a', 'b', 'c'] * (1000 // 3 +
1)[:1000]),
                   'D': pd.to_datetime(np.random.randint(1609459200,
1640995200, 1000), unit='s')})
```

```
# Запис на DataFrame във Feather файл
df.to_feather('my_data.feather')

# Четене на DataFrame от Feather файл
loaded_df = pd.read_feather('my_data.feather')

# Проверка дали данните са еднакви
print(df.equals(loaded_df))

# Запис с компресия LZ4
df.to_feather('my_data_compressed.feather', compression='lz4')

# Четене само на определени колони
subset_df = pd.read_feather('my_data.feather', columns=['A', 'C'])
print(subset_df.head())
```

в) Предимства на Feather формат:

- **Висока скорост:** Оптимизиран за бърз запис и четене, особено в сравнение с CSV или pickle.
- **Лесен за използване:** Прости функции за запис и четене.
- **Крос-платформен и крос-езиков:** Поддържа се добре между Python (Pandas) и R.
- **Поддръжка на Apache Arrow:** Базиран на Apache Arrow, което осигурява ефективно представяне на данни в паметта.
- **Компресия:** Предлага опции за компресия (например, LZ4 и Zstandard) за намаляване на размера на файла при запазване на добра производителност.

г) Кога да използваме Feather формат:

- Когато скоростта на запис и четене е критична, особено при работа с големи DataFrame-и.
- За бърз обмен на данни между Python (Pandas) и R.
- За съхранение на междинни резултати по време на анализ.
- Когато размерът на файла на диска е важен и е необходима компресия с добра производителност.

д) Ограничения:

- Основно е предназначен за съхранение на един DataFrame на файл (за разлика от HDF5, който може да съдържа множество обекти).
- Може да не е толкова гъвкав за съхранение на много сложни структури от данни извън DataFrame-и.

2. Допълнително

1) **Избор на компресия:** Feather поддържа няколко алгоритъма за компресия.
    - `'uncompressed'` **(по подразбиране):** Без компресия. Най-бързо записване и четене, но най-голям размер на файла.
    - `'lz4'`**:** Много бърз алгоритъм за компресия и декомпресия, с разумно намаляване на размера на файла. Често е предпочитан заради скоростта си.
    - `'zstd'`**:** Предлага по-добра компресия от LZ4 при малко по-висока цена за скорост на компресия (скоростта на декомпресия обикновено е много добра). Подходящ, когато размерът на файла е по-важен, но все пак се изисква добра производителност.

Изборът на компресия зависи от баланса между скорост и размер на файла за вашия конкретен случай на употреба.

2) **Работа с различни типове данни:** Feather поддържа повечето стандартни типове данни на Pandas, включително числови, булеви, категорийни и времеви редици. Въпреки това, при много сложни или нестандартни типове данни може да има ограничения или загуба на специфична информация.
3) **Крос-езикова съвместимост:** Едно от основните предимства на Feather е неговата съвместимост между Python (Pandas) и R. DataFrame-и, записани във Feather формат от Pandas, могат лесно да бъдат прочетени в R и обратно, без загуба на данни или метаданни (в повечето случаи).
4) **Съхранение в паметта (Apache Arrow):** Тъй като Feather е базиран на Apache Arrow, данните се съхраняват във формат, който е оптимизиран за аналитични операции в паметта. Това може да доведе до по-добра производителност при последващ анализ на прочетените данни в Pandas.
5) **`storage_options` за облачно съхранение:** Подобно на други функции за четене/запис на файлове в Pandas, `read_feather` и `to_feather` поддържат `storage_options` за работа с файлове, съхранявани в облачни услуги като AWS S3 или Google Cloud Storage. Това позволява директно четене и запис на Feather файлове от и към облака.

3. Изключения и потенциални проблеми:

1. **`FileNotFoundError`**: Възниква, ако се опитате да прочетете Feather файл, който не съществува на указания път.
2. **`ArrowInvalid`**: Може да възникне, ако файлът не е валиден Feather файл или е повреден.
3. **Проблеми със специфични типове данни:** Въпреки че поддръжката е широка, е възможно някои много специфични или нестандартни обекти в DataFrame-а да не бъдат сериализирани коректно във Feather формат или да възникне грешка при записа или четенето.
4. **Липса на вградена поддръжка за множество обекти:** За разлика от HDF5, Feather обикновено съхранява един DataFrame на файл. Ако трябва да съхранявате множество Pandas обекти, ще трябва да използвате няколко Feather файла или друг формат.
5. **Зависимости:** За да работите с Feather, трябва да имате инсталирани библиотеките `pandas` и `pyarrow`. Ако `pyarrow` не е инсталиран, може да получите грешка при опит за използване на `read_feather` или `to_feather`.

4. Най-добри практики:

- Избирайте подходяща компресия в зависимост от вашите нужди за скорост и размер на файла. Експериментирайте с `'lz4'` и `'zstd'`, за да видите коя работи най-добре за вашите данни.
- Уверете се, че имате инсталирана библиотеката `pyarrow` (`pip install pyarrow`).
- Използвайте Feather, когато скоростта на I/O е критична и работите основно с DataFrame-и.
- Когато обменяте данни между Python и R, Feather е отличен избор поради своята крос-езикова съвместимост.
- При работа с облачно съхранение, използвайте параметъра `storage_options` за конфигуриране на връзката.

Казус 1: Бърз обмен на междинни резултати в pipeline за обработка на данни

Ситуация: Разработвате сложен pipeline за обработка на данни, който включва няколко последователни стъпки. Някои от тези стъпки генерират големи DataFrame-и като междинни резултати, които трябва да бъдат подадени към следващите стъпки. Скоростта на запис и четене на тези междинни данни е от решаващо значение за общата производителност на pipeline-а.

Решение:

Използвайте Feather формат за записване на междинните DataFrame-и между стъпките на pipeline-а. Поради високата скорост на запис и четене на Feather, времето, прекарано за I/O операции, ще бъде значително намалено в сравнение с други формати като CSV или pickle.

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

# Симулиране на голям DataFrame като резултат от първа стъпка
n_rows = 10_000_000
df_step1 = pd.DataFrame({'col1': np.random.rand(n_rows),
                         'col2': np.random.randint(0, 100, n_rows)})

# Записване във Feather формат
start_time = time.time()
df_step1.to_feather('step1_output.feather')
end_time = time.time()
print(f"Време за запис във Feather: {end_time - start_time:.2f} секунди")

# В следващата стъпка на pipeline-а:
start_time = time.time()
df_step2_input = pd.read_feather('step1_output.feather')
end_time = time.time()
print(f"Време за четене от Feather: {end_time - start_time:.2f} секунди")

# Продължете обработката с df_step2_input
# ...
```

Казус 2: Споделяне на големи набори от данни между Python и R за съвместен анализ

Ситуация: Екип от анализатори използва както Python (с Pandas), така и R за анализ на едни и същи големи набори от данни. Необходимо е бърз и лесен начин за обмен на тези данни между двете среди, без да се губи информация или да се налага сложна конверсия.

Решение:

Използвайте Feather формат за съхранение и обмен на данните. DataFrame-и, обработени и записани от Pandas в Python, могат директно да бъдат прочетени в R с помощта на пакета `feather`, и обратно. Това е много по-ефективно от експортирането в CSV и последващото парсване в другата среда.

В Python (запис):

```
import pandas as pd
```

```
# ... (зареждане или обработка на DataFrame df_for_r) ...
df_for_r.to_feather('shared_data.feather')
print("Данните са записани във Feather формат за използване в R.")
```

В R (четене):

```
library(feather)
shared_data <- read_feather("shared_data.feather")
print(head(shared_data))
# Продължете анализа в R
```

Казус 3: Кеширане на често използвани, големи DataFrame-и в memory-intensive приложения

Ситуация: Разработвате приложение за анализ на данни, което често използва няколко големи DataFrame-а. Зареждането на тези DataFrame-и от диска всеки път, когато са необходими, води до значително забавяне. Имате достатъчно RAM, за да ги кеширате.

Решение:

При стартиране на приложението, заредете необходимите големи DataFrame-и от Feather файлове (ако вече са записани веднъж). Feather осигурява бързо зареждане в паметта. След това приложението може да работи директно с тези DataFrame-и в паметта. Ако DataFrame-ите се променят по време на работа, те могат бързо да бъдат записани обратно във Feather формат при затваряне на приложението или при определени събития.

```
import pandas as pd
import os

data_file = 'large_dataframe.feather'

def load_large_dataframe():
    if os.path.exists(data_file):
        print("Зареждане на кеширан DataFrame от Feather...")
        return pd.read_feather(data_file)
    else:
        print("Създаване на голям DataFrame...")
        n_rows = 5_000_000
        df = pd.DataFrame({'col1': np.random.rand(n_rows),
                           'col2': np.random.randint(0, 100, n_rows)})
        df.to_feather(data_file)
        return df

large_df = load_large_dataframe()
print(f"Размер на заредения DataFrame: {large_df.shape}")

# ... (приложението използва large_df) ...
```

```
# При затваряне (или при необходимост от запазване на промените):
# large_df.to_feather(data_file)
```

Въпроси:

1. Каква е основната цел на Feather формата и защо е разработен? Какви са основните му предимства?
2. На каква технология е базиран Feather форматът и как това допринася за неговата ефективност?
3. Сравнете скоростта и размера на файла при използване на Feather формат с други формати за съхранение на DataFrame-и като CSV и pickle. В какви ситуации Feather може да бъде значително по-ефективен?
4. Какви опции за компресия се поддържат от `df.to_feather()`? Кои са основните характеристики на всеки метод на компресия по отношение на скорост и степен на компресия?
5. Какви типове данни на Pandas се поддържат добре от Feather формата? Има ли ограничения при работа с много специфични или нестандартни типове данни?
6. Обяснете как Feather улеснява обмена на данни между Python (Pandas) и R. Защо е по-удобен от други формати за тази цел?
7. Какви са основните параметри на функциите `pd.read_feather()` и `df.to_feather()` и как се използват?
8. В какви сценарии би било полезно да използвате параметъра `columns` при четене на Feather файл?
9. Как `storage_options` параметърът позволява работа с Feather файлове, съхранявани в облачни услуги?
10. Какви са някои от потенциалните изключения или проблеми, които могат да възникнат при работа с Feather формат?
11. Какво е необходимо да бъде инсталирано, за да се използва Feather в Pandas?
12. В какви ситуации може да е по-подходящо да използвате HDFStore вместо Feather и обратно? Сравнете ги по отношение на възможности и употреба.

Задачи:

1. **Сравнение на скорост:** Създайте голям DataFrame (например, с няколко милиона реда и няколко колони с различни типове данни). Измерете времето за запис и четене на този DataFrame, използвайки CSV, pickle и Feather (с и без компресия 'lz4'). Сравнете резултатите.
2. **Крос-езиков обмен:** Създайте DataFrame в Pandas и го запишете във Feather формат. След това (ако имате инсталиран R и пакета `feather`), прочетете този файл в R и покажете първите няколко реда. Създайте DataFrame в R и го запишете във Feather формат. Прочетете го обратно в Pandas и покажете първите няколко реда.
3. **Четене на подмножество от колони:** Създайте DataFrame с няколко колони и го запишете във Feather файл. След това прочетете файла обратно, като изберете само две от колоните.
4. **Изследване на компресия:** Запишете голям DataFrame във Feather формат, използвайки различни методи на компресия ('uncompressed', 'lz4', 'zstd'). Сравнете размера на получените файлове и времето за запис/четене за всеки метод.
5. **Работа с облачно съхранение (симулирано):** Ако имате достъп до облачно хранилище (например, AWS S3), опитайте да прочетете и запишете Feather файл, използвайки `storage_options`. Ако не, можете да симулирате това, като използвате локални файлове и се фокусирате върху синтаксиса на `storage_options` (например, за HTTP/HTTPS URL).

XII. Parquet формат: (read_parquet, to_parquet) - *Изисква pyarrow или fastparquet*

Apache Parquet е колонен формат за съхранение на данни, оптимизиран за бързи заявки и ефективна компресия. Той е проектиран за обработка на големи набори от данни и е често използван в екосистемите на Apache Hadoop и Apache Spark, както и в Python с Pandas и PyArrow. Колонният формат позволява по-ефективно четене на определени колони (вместо целия ред), което е много полезно при аналитични задачи.

1. Основни функции:

а) Основни методи и клас:

- **`pandas.read_parquet(path, columns=None, use_threads=True, storage_options=None, row_group_offsets=None, **kwargs)`**: Функция за четене на DataFrame от Parquet файл.
  - `path`: Пътят до Parquet файла (може да бъде и обект, подобен на файл или URL). Поддържа се и четене на директории, съдържащи партиционирани Parquet файлове.
  - `columns`: Списък от колони, които да бъдат прочетени. Ако е `None`, се четат всички колони.
  - `use_threads`: Булева стойност, която определя дали да се използват множество нишки за четене (ако е възможно).
  - `storage_options`: Допълнителни опции за системи за съхранение, като AWS S3, Google Cloud Storage и др.
  - `row_group_offsets`: Списък с индекси на групи редове за четене (за напреднали потребители).
  - `**kwargs`: Допълнителни аргументи, които се предават на основната Parquet engine (обикновено PyArrow).
- **`DataFrame.to_parquet(path, engine='auto', compression='snappy', index=None, partition_cols=None, storage_options=None, **kwargs)`**: Метод за записване на DataFrame във Parquet файл.
  - `path`: Пътят до Parquet файла или директория (ако се използва партициониране).
  - `engine`: Parquet engine, който да се използва ('auto', 'pyarrow', 'fastparquet'). 'pyarrow' е препоръчителният и по-гъвкав engine.
  - `compression`: Алгоритъм за компресия ('snappy', 'gzip', 'lzo', 'brotli', None). 'snappy' е бърз и осигурява разумна компресия.
  - `index`: Булева стойност, която определя дали индексът на DataFrame-а да бъде записан като колона. Ако е `None`, индексът се записва.
  - `partition_cols`: Списък от колони, по които да се партиционира DataFrame-ът при запис. Това създава структура от поддиректории.
  - `storage_options`: Допълнителни опции за системи за съхранение.
  - `**kwargs`: Допълнителни аргументи, които се предават на основната Parquet engine.

б) Пример за запис и четене:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
```

```
df = pd.DataFrame({'A': np.random.rand(1000),
                   'B': np.random.randint(0, 100, 1000),
                   'C': pd.Categorical(['a', 'b', 'c'] * (1000 // 3 +
1)[:1000]),
                   'D': pd.to_datetime(np.random.randint(1609459200,
1640995200, 1000), unit='s')})

# Запис на DataFrame в Parquet файл
df.to_parquet('my_data.parquet')

# Четене на DataFrame от Parquet файл
loaded_df = pd.read_parquet('my_data.parquet')

# Проверка дали данните са еднакви
print(df.equals(loaded_df))

# Запис с компресия Gzip и без индекс
df.to_parquet('my_data_compressed.parquet', compression='gzip', index=False)

# Четене само на определени колони
subset_df = pd.read_parquet('my_data.parquet', columns=['A', 'C'])
print(subset_df.head())

# Запис с партициониране по колона 'C'
df['partition_col'] = df['C']
df.to_parquet('partitioned_data', partition_cols=['partition_col'])

# Четене на партиционирани данни
partitioned_df = pd.read_parquet('partitioned_data')
print(partitioned_df.head())
```

в) Предимства на Parquet формат:

- **Колонен формат:** Позволява ефективно четене само на необходимите колони, което значително ускорява аналитичните заявки и намалява използването на памет.
- **Висока компресия:** Поддържа различни алгоритми за компресия, които могат значително да намалят размера на файла на диска, особено за данни с много повтарящи се стойности.
- **Схема вградена във файла:** Parquet файловете съдържат информация за схемата на данните, което улеснява четенето и гарантира консистентност.
- **Добра поддръжка в екосистемите за големи данни:** Широко използван в Hadoop, Spark и други инструменти за обработка на големи данни.
- **Поддръжка на партициониране:** Възможност за разделяне на данните на множество файлове в структура от директории въз основа на стойностите на една или повече колони, което допълнително оптимизира заявките.

г) Кога да използваме Parquet формат:

- Когато работите с големи набори от данни и производителността на четене е важна.
- За аналитични работни процеси, които често изискват само подмножество от колони.
- За съхранение на данни, които ще се обработват с инструменти като Spark или Dask.
- Когато е важна висока степен на компресия за намаляване на дисковото пространство.

!!!Важно: За да работите с Parquet, трябва да имате инсталирана поне една от поддържаните библиотеки за Parquet engine (PyArrow е силно препоръчителен: `pip install pyarrow`). `fastparquet` е друга опция (`pip install fastparquet`). Pandas ще използва 'pyarrow' по подразбиране, ако е инсталиран.

2. Допълнителни аспекти:

а) Избор на Parquet Engine:

Pandas поддържа няколко backend-а за работа с Parquet файлове, контролирани от параметъра `engine` в `to_parquet()` и `read_parquet()`:

- **`'auto'` (по подразбиране):** Pandas автоматично избира engine-а. Препоръчва се да се остави така, тъй като обикновено избира най-добрия наличен engine (предпочита PyArrow, ако е инсталиран).
- **`'pyarrow'`: ** Препоръчителният engine. Той е по-гъвкав, поддържа повече типове данни и компресии, както и партициониране.
- **`'fastparquet'`: ** Друг engine, който може да бъде по-бърз в някои случаи, но може да има ограничения в поддръжката на определени типове данни или функционалности.

Препоръчително е да използвате `'pyarrow'`, освен ако нямате специфични причини да използвате `'fastparquet'`.

б) Партициониране (`partition_cols`):

Партицонирането е мощен начин за организиране на големи набори от данни, като ги разделя на поддиректории въз основа на стойностите на една или повече колони. Когато четете партиционирани данни, можете да филтрирате по партиционните колони, което може значително да ускори заявките, тъй като се четат само съответните поддиректории.

```
# Записване с партициониране по две колони
df['year'] = pd.to_datetime(df['D']).dt.year
df['month'] = pd.to_datetime(df['D']).dt.month
df.to_parquet('partitioned_by_year_month', partition_cols=['year', 'month'])

# Четене само на данни за определена година
df_year_2023 = pd.read_parquet('partitioned_by_year_month/year=2023')

# Четене само на данни за определен месец и година
df_jan_2023 = pd.read_parquet('partitioned_by_year_month/year=2023/month=1')

# Четене на всички партиционирани данни
all_partitioned_data = pd.read_parquet('partitioned_by_year_month')
```

в) Различни алгоритми за компресия:

Parquet поддържа няколко алгоритъма за компресия, които могат да бъдат избрани чрез параметъра `compression`:

- **`'snappy'` (по подразбиране):** Бърз алгоритъм с разумна компресия.
- **`'gzip'`:** По-добра компресия от Snappy, но по-бавен.

- **`'lzo'`:** Бърз алгоритъм за компресия, но може да изисква допълнителни библиотеки.
- **`'brotli'`:** Нов алгоритъм, предлагащ по-добра компресия от gzip при сравнима скорост (може да изисква допълнителна инсталация).
- **`None`:** Без компресия.

Изборът на компресия зависи от приоритета между скорост и размер на файла.

г) Запазване на индекса (`index`):

Параметърът `index` в `to_parquet()` контролира дали индексът на DataFrame-а се записва като колона във файла. По подразбиране индексът се записва. Ако `index=False`, индексът не се запазва.

д) Метаданни:

Parquet файловете съдържат метаданни за структурата на данните, което позволява на четящите процеси да разберат схемата на данните без да четат целия файл.

3. Изключения и потенциални проблеми:

- **`FileNotFoundError`**: Възниква при опит за четене на несъществуващ Parquet файл или директория.
- **`ArrowInvalid` или подобни грешки от engine-а:** Могат да възникнат, ако файлът не е валиден Parquet файл или е повреден, или ако има проблеми със съвместимостта на engine-а с файла.
- **Поддръжка на типове данни:** Въпреки че Parquet поддържа много типове данни на Pandas, може да има редки случаи на непълна или неефективна поддръжка за някои много специфични типове. PyArrow обикновено има по-широка поддръжка.
- **Проблеми с партиционирани данни:** Неправилно структурирани партиционирани директории могат да доведат до грешки при четене. Уверете се, че структурата на директориите следва конвенцията `partition_column=value`.
- **Зависимости:** За работа с Parquet в Pandas е необходимо да имате инсталиран поне един от поддържаните Parquet engine-и (`pyarrow` или `fastparquet`).

4. Най-добри практики:

- Използвайте `'pyarrow'` като Parquet engine за по-добра съвместимост и поддръжка на функционалности.
- Обмислете партициониране на големи набори от данни по често използвани филтърни колони, за да оптимизирате заявките.
- Изберете подходящ алгоритъм за компресия в зависимост от вашите нужди за скорост и размер на файла.
- Бъдете внимателни при записване на индекса; решете дали е необходимо да го запазите като колона.
- Уверете се, че структурата на партиционираните директории е консистентна.
- Поддържайте актуални версии на `pandas` и `pyarrow` (или `fastparquet`).

Казус 1: Обработка на лог файлове от уеб сървър

Ситуация: Имате огромни дневни файлове (логи) от уеб сървър, които съдържат информация за всяко посещение (време, IP адрес, URL, потребителски агент и др.). Тези файлове са твърде големи, за да се обработват ефективно в паметта като обикновен текстов файл или CSV. Трябва да извършвате анализи, които често включват филтриране по определени колони (например, времеви периоди, IP адреси) и агрегиране на данни.

Решение:

1) **Конвертиране в Parquet:** Прочетете лог файловете (може би на парчета, ако са много големи) и ги запишете във Parquet формат. Изберете подходяща компресия (например, `snappy` за баланс между скорост и размер).

```
import pandas as pd

# Да предположим, че имаме функция за четене на лог файл и връщане на
DataFrame
def read_web_log(file_path):
    # ... (логика за парсване на лог файла) ...
    return pd.DataFrame(...)

log_file_path = 'web_server.log'
df_logs = read_web_log(log_file_path)
df_logs.to_parquet('web_server_logs.parquet', compression='snappy')
```

2) **Анализ с Parquet:** Когато е необходимо да се анализират данните, прочетете Parquet файла и извършете необходимите операции. Тъй като Parquet е колонен формат, четенето само на необходимите колони (например, 'timestamp' и 'IP address' за анализ на трафика) ще бъде много бързо и ефективно.

```
df_parquet = pd.read_parquet('web_server_logs.parquet',
columns=['timestamp', 'ip_address'])

# Филтриране по времеви диапазон
start_date = '2025-05-01'
end_date = '2025-05-02'
df_filtered = df_parquet[(df_parquet['timestamp'] >= start_date) &
(df_parquet['timestamp'] < end_date)]

# Агрегиране по IP адрес
ip_counts = df_filtered['ip_address'].value_counts()
print(ip_counts.head(10))
```

Казус 2: Съхранение на исторически финансови данни

Ситуация: Компания събира исторически данни за финансови транзакции (дата, вид транзакция, сума, валута и др.). Обемът на тези данни нараства с времето и е необходимо ефективно съхранение, което позволява бързо извличане на данни за определени периоди или видове транзакции за целите на отчитане и анализ.

Решение:

1) **Съхранение във Parquet с партициониране:** Записвайте финансовите данни във Parquet формат и ги партиционирайте по година и месец на транзакцията. Това ще създаде структура от директории (`year=YYYY/month=MM/`), което ще ускори заявките за конкретни периоди.

```
import pandas as pd

# Да предположим, че df_financial_data съдържа финансовите данни с колона
'date'
df_financial_data['year'] =
pd.to_datetime(df_financial_data['date']).dt.year
df_financial_data['month'] =
pd.to_datetime(df_financial_data['date']).dt.month

df_financial_data.to_parquet('financial_data_partitioned',
partition_cols=['year', 'month'], compression='gzip')
```

2) **Четене на партиционирани данни за анализ:** Когато е необходимо да се анализират данни за определен период, прочетете само съответните партиции.

```
# Анализ на данни за януари 2025
january_2025_data =
pd.read_parquet('financial_data_partitioned/year=2025/month=1')
print(f"Брой транзакции през януари 2025: {len(january_2025_data)}")

# Анализ на всички данни за 2024
year_2024_data = pd.read_parquet('financial_data_partitioned/year=2024')
print(f"Брой транзакции през 2024: {len(year_2024_data)}")
```

Казус 3: Интеграция с инструменти за Big Data (Spark)

Ситуация: Използвате Pandas за предварителна обработка и анализ на данни, но част от по-мащабната обработка и анализ се извършва с Apache Spark. Необходимо е ефективен начин за прехвърляне на DataFrame-и между Pandas и Spark.

Решение:

Parquet е отличен формат за тази цел, тъй като се поддържа нативно и ефективно както от Pandas (чрез PyArrow), така и от Spark.

В Pandas (запис за Spark):

```
import pandas as pd
```

```
# ... (обработка на DataFrame df_for_spark) ...
df_for_spark.to_parquet('data_for_spark.parquet', compression='snappy')
```

В Spark (четене от Pandas):

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder.appName("ReadParquet").getOrCreate()

val df_from_pandas = spark.read.parquet("data_for_spark.parquet")
df_from_pandas.printSchema()
df_from_pandas.show()

// ... (извършване на Spark анализ) ...

// Запис от Spark обратно към Parquet (може да се чете от Pandas)
// processed_df.write.parquet("processed_data.parquet")

spark.stop()
```

Тези казуси показват как колонният формат и възможностите за партициониране и компресия на Parquet го правят мощен инструмент за работа с големи набори от данни в различни реални сценарии, особено когато се изисква ефективност при четене и интеграция с други инструменти за обработка на данни.

Казус 4*: Анализ на разпределени лог файлове по дата и тип съобщение

Генерирате големи обеми от лог файлове от различни приложения. Тези лог файлове съдържат информация за времето на събитието, ниво на съобщението (INFO, WARNING, ERROR), източник на съобщението и самото съобщение. За да подобрите ефективността на съхранение и заявките, сте решили да съхранявате тези лог файлове в Parquet формат, като данните са партиционирани по година, месец и ниво на съобщение.

Структура на данните (примерни колони):

- `timestamp`: Timestamp на събитието.
- `level`: Ниво на съобщението (например: 'INFO', 'WARNING', 'ERROR').
- `source`: Източник на съобщението (например, 'AuthService', 'ApiService').
- `message`: Самото лог съобщение.

Данните са записани в Parquet файлове в директория, която има следната структура на партициите:

```
log_data/
├── year=2024/
│   ├── month=01/
```

```
│   │   ├── level=INFO/
│   │   │   └── part-00000-....parquet
│   │   │   └── ...
│   │   ├── level=WARNING/
│   │   │   └── part-00000-....parquet
│   │   │   └── ...
│   │   └── level=ERROR/
│   │       └── part-00000-....parquet
│   │       └── ...
│   ├── month=02/
│   │   └── ...
│   └── ...
└── year=2025/
    └── ...
```

Задача:

1. Симулирайте генерирането на голям Pandas DataFrame с лог данни за няколко месеца и различни нива на съобщения.
2. Запишете този DataFrame в Parquet формат в структура от партиции, както е описано по-горе, използвайки колоните 'year', 'month' (извлечени от `timestamp`) и 'level' като партиционни колони.
3. Прочетете ефективно всички лог данни от партиционираната директория в един Pandas DataFrame.
4. Прочетете само лог съобщенията за определена година (например, 2024).
5. Прочетете само лог съобщенията за определен месец от определена година (например, януари 2024).
6. Прочетете само лог съобщенията за определено ниво (например, 'ERROR') за всички години и месеци.
7. Извършете анализ върху прочетените данни:
   - Пресметнете броя на съобщенията за всяко ниво за всяка година.
   - Намерете топ 10 на най-често срещаните съобщения за грешки (ERROR ниво).
8. Запишете резултатите от анализа (например, DataFrame с броя на съобщенията по ниво и година) в нов Parquet файл без партициониране.

Решение:

```
import pandas as pd
import numpy as np
import os
import datetime

# 1. Симулиране на генериране на лог данни
num_rows = 100000
start_date = datetime.datetime(2024, 1, 1)
end_date = datetime.datetime(2025, 3, 31)
timestamps = pd.to_datetime(np.random.uniform(start_date.timestamp(),
end_date.timestamp(), num_rows), unit='s')
levels = np.random.choice(['INFO', 'WARNING', 'ERROR'], num_rows, p=[0.7,
0.2, 0.1])
sources = np.random.choice(['AuthService', 'ApiService', 'Database',
'Scheduler'], num_rows)
messages = [f"Log message {i}" for i in range(num_rows)]

log_df = pd.DataFrame({
```

```
    'timestamp': timestamps,
    'level': levels,
    'source': sources,
    'message': messages
})
log_df['year'] = log_df['timestamp'].dt.year
log_df['month'] = log_df['timestamp'].dt.month.astype(str).str.zfill(2)

# 2. Записване в Parquet с партициониране
output_dir = 'log_data'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

log_df.to_parquet(output_dir, partition_cols=['year', 'month', 'level'])
print(f"Лог данните са записани в Parquet формат с партициониране в
директория '{output_dir}'.")

# 3. Четене на всички лог данни
all_logs_df = pd.read_parquet(output_dir)
print(f"\nПрочетени са {len(all_logs_df)} записа от всички партиции.")
print(all_logs_df.head())

# 4. Четене на лог съобщения за определена година
logs_2024_df = pd.read_parquet(output_dir, filters=[('year', '=', 2024)])
print(f"\nПрочетени са {len(logs_2024_df)} записа за 2024 г.")
print(logs_2024_df.head())

# 5. Четене на лог съобщения за определен месец от определена година
logs_jan_2024_df = pd.read_parquet(output_dir, filters=[('year', '=', 2024),
('month', '=', '01')])
print(f"\nПрочетени са {len(logs_jan_2024_df)} записа за януари 2024 г.")
print(logs_jan_2024_df.head())

# 6. Четене на лог съобщения за определено ниво
error_logs_df = pd.read_parquet(output_dir, filters=[('level', '=',
'ERROR')])
print(f"\nПрочетени са {len(error_logs_df)} записа с ниво 'ERROR'.")
print(error_logs_df.head())

# 7. Анализ на данните
# a) Брой съобщения за всяко ниво за всяка година
yearly_level_counts = all_logs_df.groupby(['year',
'level']).size().reset_index(name='count')
print("\nБрой съобщения по ниво и година:")
print(yearly_level_counts)

# b) Топ 10 най-често срещани съобщения за грешки
top_10_error_messages = error_logs_df['message'].value_counts().nlargest(10)
print("\nТоп 10 най-често срещани съобщения за грешки:")
print(top_10_error_messages)

# 8. Записване на резултатите от анализа в нов Parquet файл
analysis_output_file = 'log_analysis_results.parquet'
```

```
yearly_level_counts.to_parquet(analysis_output_file)
print(f"\nРезултатите от анализа са записани в '{analysis_output_file}'.")
```

Разбор на сложността:

- **Генериране на голям набор от данни:** Симулира се сценарий с голям обем от данни, където ефективното съхранение е важно.
- **Партициониране на данни при запис:** Данните се разделят на множество файлове в йерархична структура от директории въз основа на стойностите на определени колони ('year', 'month', 'level').
- **Ефективно четене с филтриране на партиции:** `pd.read_parquet()` се използва с `filters` параметър за четене само на необходимите партиции, което значително намалява времето за четене и използването на памет.
- **Работа с дати и извличане на партиционни колони:** Годината и месецът се извличат от колоната с времеви печат за целите на партиционирането.
- **Анализ на големи набори от данни:** Извършват се групови операции и преброяване върху прочетените данни.
- **Записване на резултати в Parquet формат:** Резултатите от анализа се записват обратно в Parquet формат за ефикасно съхранение.

Този казус илюстрира как Parquet форматът, особено в комбинация с партициониране, може да бъде изключително полезен за управление и анализ на големи, структурирани набори от данни, като лог файлове, финансови данни или други видове данни, които могат да бъдат ефективно разделени по определени категории или времеви периоди. Четенето само на необходимите партиции значително подобрява производителността при заявки към големи обеми от данни.

Въпроси:

1. Какво представлява Parquet форматът и какви са основните му характеристики и предимства в сравнение с други формати като CSV или Feather?
2. Обяснете концепцията за колонен формат за съхранение на данни. Как това се отразява на ефективността при четене и компресия в Parquet?
3. Какви са различните Parquet engine-и, които могат да се използват в Pandas (`engine` параметър)? Кой е препоръчителният и защо?
4. Опишете процеса на записване на DataFrame във Parquet файл с помощта на `df.to_parquet()`. Кои са някои важни параметри, които могат да бъдат настроени?
5. Как се чете DataFrame от Parquet файл, използвайки `pd.read_parquet()`? Какви са възможностите за четене само на определени колони?
6. Какво е партициониране на данни при запис във Parquet формат (`partition_cols`)? Каква е структурата на директориите, която се създава, и каква е ползата от партицонирането при четене?
7. Избройте и опишете някои от алгоритмите за компресия, поддържани от Parquet в Pandas (`compression` параметър). Как се избира подходящият алгоритъм?
8. Как параметърът `index` в `df.to_parquet()` контролира запазването на индекса на DataFrame-а? Кога бихте искали да запазите или да не запазите индекса?
9. Как Parquet форматът се интегрира с други инструменти и екосистеми за обработка на големи данни като Apache Spark? Защо е предпочитан формат за обмен на данни с тях?
10. Какви са някои от потенциалните изключения или проблеми, които могат да възникнат при работа с Parquet файлове в Pandas?
11. Какво е необходимо да бъде инсталирано, за да се използва Parquet формат в Pandas?
12. В какви сценарии би бил Parquet по-подходящ от Feather и обратно? Сравнете ги по отношение на техните силни страни.

Задачи:

1. **Запис и четене с различни engine-и:** Създайте DataFrame и го запишете във Parquet файл, използвайки както `'pyarrow'`, така и `'fastparquet'` engine (ако е инсталиран). След това прочетете и двата файла и се уверете, че данните са еднакви. Забелязвате ли разлики в скоростта или размера на файла?
2. **Партициониране и четене на партиции:** Създайте DataFrame с колона за дата. Партиционирайте го по година и месец при запис във Parquet формат. След това прочетете само данните за определен месец от определена година, като укажете пътя към съответната партиция. Прочетете и всички данни наведнъж.
3. **Изследване на компресия:** Запишете голям DataFrame във Parquet формат, използвайки различни методи на компресия (`'snappy'`, `'gzip'`, `None`). Сравнете размера на получените файлове и времето за запис/четене.
4. **Четене само на подмножество от колони:** Създайте DataFrame с няколко колони и го запишете във Parquet файл. Прочетете файла обратно, като изберете само две от колоните. Измерете времето и сравнете го с четенето на всички колони.
5. **Работа с индекс:** Създайте DataFrame с нетривиален индекс (например, MultiIndex). Запишете го във Parquet файл веднъж със запазване на индекса (по подразбиране) и веднъж без (`index=False`). Прочетете обратно и двата файла и наблюдавайте как се обработва индексът.

XIII. ORC формат: (read_orc, to_orc) – Изисква pyarrow

ORC (Optimized Row Columnar) е колонен формат за съхранение на данни, подобен на Parquet, но разработен специално за Apache Hive. Той е оптимизиран за висока производителност при четене и запис на данни в среди на Hadoop и Hive. ORC предлага ефективна компресия и възможност за филтриране на данни на ниво блок, което може значително да ускори заявките.

!!!Важно: За да работите с ORC формат в Pandas, е *задължително* да имате инсталирана библиотеката `pyarrow` (`pip install pyarrow`).

1. Основни функции:

- **`pandas.read_orc(path, columns=None, use_threads=True, storage_options=None, **kwargs)`**: Функция за четене на DataFrame от ORC файл.
  - `path`: Път до ORC файла (може да бъде и обект, подобен на файл или URL).
  - `columns`: Списък от колони за четене. Ако е `None`, се четат всички.
  - `use_threads`: Булева стойност, определяща дали да се използват множество нишки за четене (ако е възможно).
  - `storage_options`: Допълнителни опции за системи за съхранение (напр., AWS S3).
  - `**kwargs`: Допълнителни аргументи, предавани на PyArrow.
- **`DataFrame.to_orc(path, index=None, compression='snappy', compression_level=None, storage_options=None, **kwargs)`**: Метод за записване на DataFrame в ORC файл.
  - `path`: Път до ORC файла.
  - `index`: Булева стойност, определяща дали индексът да се запише като колона. Ако е `None`, индексът се записва.
  - `compression`: Алгоритъм за компресия ('snappy', 'gzip', 'zstd', None). 'snappy' е бърз и ефективен.
  - `compression_level`: Ниво на компресия (зависи от алгоритъма).
  - `storage_options`: Допълнителни опции за системи за съхранение.
  - `**kwargs`: Допълнителни аргументи, предавани на PyArrow.

2. Пример за запис и четене:

```
import pandas as pd import numpy as np
# Създаване на примерен DataFrame
df = pd.DataFrame({'A': np.random.rand(1000),
'B': np.random.randint(0, 100, 1000),
'C': pd.Categorical(['a', 'b', 'c'] * (1000 // 3 +
1)[:1000]),
'D': pd.to_datetime(np.random.randint(1609459200, 1640995200, 1000),
unit='s')})
# Запис на DataFrame в ORC файл
```

```
df.to_orc('my_data.orc')


# Четене на DataFrame от ORC файл
loaded_df = pd.read_orc('my_data.orc')

# Проверка дали данните са еднакви
print(df.equals(loaded_df))

# Запис с компресия Gzip и без индекс
df.to_orc('my_data_compressed.orc', compression='gzip', index=False)
```

2. Предимства на ORC формат:

- **Колонен формат:** Позволява ефективно четене само на необходимите колони, ускорявайки заявките.
- **Висока компресия:** Поддържа различни алгоритми за компресия, намаляващи размера на файла.
- **Филтриране на ниво блок:** ORC позволява на Hive и други инструменти да пропускат цели блокове данни, които не отговарят на условията на заявката, което допълнително ускорява обработката.
- **Добра интеграция с Hive и Hadoop:** Разработен е специално за Hive и е оптимизиран за работа в среди на Hadoop.
- **Схема вградена във файла:** ORC файловете съдържат метаданни за структурата на данните.

3. Кога да използваме ORC формат:

- Когато работите с големи набори от данни и се изисква висока производителност при четене.
- За данни, които ще се обработват с Apache Hive или други инструменти в екосистемата на Hadoop.
- Когато е важна ефективната компресия.
- Когато се възползвате от възможността за филтриране на ниво блок.

4. Ограничения:

- Основно е предназначен за използване в рамките на екосистемата на Hadoop.
- Може да не е толкова широко поддържан извън тези среди, колкото Parquet.
- Задължително изисква `pyarrow` за работа с Pandas.

5. Допълнителни аспекти:

1) **Зависимост от PyArrow:** Както вече беше споменато, работата с ORC формат в Pandas изисква инсталирана библиотеката `pyarrow`. Ако `pyarrow` не е инсталиран, ще получите грешка при опит за използване на `pd.read_orc()` или `df.to_orc()`.
2) **Избор на компресия:** Подобно на Parquet и Feather, ORC поддържа различни алгоритми за компресия, които могат да бъдат избрани чрез параметъра `compression`:
    - **`'snappy'` (по подразбиране):** Бърз алгоритъм с добра компресия.
    - **`'gzip'`:** По-добра компресия от Snappy, но по-бавен.
    - **`'zstd'`:** Съвременен алгоритъм, предлагащ добра компресия и скорост.
    - **`None`:** Без компресия.

    Изборът на компресия зависи от вашите изисквания за скорост и размер на файла. Параметърът `compression_level` може да се използва за по- фина настройка на нивото на компресия при някои алгоритми (в зависимост от `pyarrow`).

3) **Запазване на индекса (`index`):** Параметърът `index` в `df.to_orc()` контролира дали индексът на DataFrame-а се записва като отделна колона в ORC файла. По подразбиране индексът се записва. За да не се записва индексът, задайте `index=False`.
4) **`storage_options` за отдалечено съхранение:** Функциите `read_orc()` и `to_orc()` поддържат параметъра `storage_options`, който позволява работа с файлове, съхранявани в отдалечени системи като AWS S3, Google Cloud Storage (GCS) и други, които се поддържат от `pyarrow`.
5) **Метаданни:** ORC файловете съдържат богати метаданни за структурата на данните, типовете на колоните и компресията, което улеснява ефективното четене и обработка.

6. Изключения и потенциални проблеми:

1) **`ImportError`**: Възниква, ако се опитате да използвате `read_orc()` или `to_orc()` без инсталиран `pyarrow`. Съобщението за грешка обикновено указва, че `pyarrow` е необходим.
2) **`FileNotFoundError`**: Възниква, ако указаният ORC файл не съществува.
3) **Грешки при четене на невалиден ORC файл**: Ако файлът не е валиден ORC файл или е повреден, `pyarrow` може да върне различни грешки при опит за четене.
4) **Поддръжка на типове данни**: Въпреки че `pyarrow` поддържа голям набор от Pandas типове данни, е възможно да има редки случаи на непълна поддръжка или неочаквано поведение при много специфични или нестандартни типове.
5) **Съвместимост на версии**: Възможно е да има проблеми със съвместимостта при четене на ORC файлове, създадени с по-стари или по-нови версии на `pyarrow` или други ORC инструменти. Препоръчително е да се използват съвместими версии.

7. Най-добри практики:

- Уверете се, че имате инсталиран `pyarrow` (`pip install pyarrow`).
- Избирайте подходяща компресия в зависимост от вашите нужди. `snappy` обикновено е добър компромис.
- Решете дали е необходимо да запазвате индекса на DataFrame-а при запис.
- Когато работите с отдалечено съхранение, конфигурирайте правилно `storage_options`.
- Поддържайте актуални версии на `pandas` и `pyarrow`.
- Ако срещате проблеми при четене на ORC файлове, проверете версията на `pyarrow`, с която са били създадени файловете.

Казус 1: Анализ на големи набори от данни от IoT устройства

Ситуация: Компания събира данни от голям брой IoT устройства, които изпращат различни показания (температура, влажност, налягане и др.) през определени интервали от време. Общият обем на данните е много голям и трябва да се анализира периодично, като често се филтрира по времеви диапазони и по идентификатор на устройството.

Решение:

1) **Съхранение в ORC:** Записвайте данните от IoT устройствата във ORC формат. Използвайте колонен формат, за да позволите ефективно четене само на необходимите колони (например, 'timestamp', 'device_id', 'temperature'). Компресията (например, 'snappy') ще помогне за намаляване на размера на файла. Ако данните се събират непрекъснато, може да е полезно да ги партиционирате по дата или час (въпреки че `pandas.to_orc` не поддържа директно партициониране, това може да се управлява на ниво структура на директориите и последващо четене).

```
import pandas as pd
import numpy as np

# Симулиране на данни от IoT устройства

n_rows = 1_000_000

dates = pd.date_range('2025-05-01', periods=n_rows, freq='S')
df_iot = pd.DataFrame({'timestamp': dates,

                       'device_id': np.random.choice(['device_1',
'device_2', 'device_3'], n_rows),

                       'temperature': np.random.rand(n_rows) * 30 + 10,

                       'humidity': np.random.rand(n_rows) * 80 + 20})

df_iot.to_orc('iot_data.orc', compression='snappy', index=False)
```

```
df_orc = pd.read_orc('iot_data.orc', columns=['timestamp', 'device_id',
'temperature'])

# Филтриране по времеви диапазон и устройство

start_date = '2025-05-01 10:00:00'

end_date = '2025-05-01 11:00:00'

device_to_analyze = 'device_2'

df_filtered = df_orc[(df_orc['timestamp'] >= start_date) &
(df_orc['timestamp'] < end_date) & (df_orc['device_id'] ==
device_to_analyze)]

print(f"Брой записи за устройство '{device_to_analyze}' в избрания час:

{len(df_filtered)}")

print(f"Средна температура: {df_filtered['temperature'].mean()}")
```

2) **Анализ с ORC:** Когато е необходимо да се анализират данните, прочетете ORC файла. Благодарение на колонния формат, филтрирането и агрегирането по определени колони ще бъде по-ефективно.

Казус 2: Интеграция с Hive за анализ на уеб трафик

Ситуация: Компания използва Apache Hive за анализ на големи обеми от данни за уеб трафик, съхранявани в HDFS. Екипът от анализатори използва Pandas за някои ad-hoc анализи и визуализации на малки подмножества от тези данни. Необходимо е лесен начин за четене на данни от ORC файлове (използвани от Hive) в Pandas.

Решение:

Използвайте `pd.read_orc()` за директно четене на ORC файлове, генерирани от Hive, в Pandas DataFrame. Това позволява на анализаторите да използват познатите инструменти на Pandas за изследване на данни, които са част от по-голям Hive базиран data warehouse.

```
import pandas as pd
# Път към ORC файл, генериран от Hive в HDFS (може да изисква специфична конфигурация за достъп до HDFS)
orc_file_path = 'hdfs://namenode:8020/user/hive/warehouse/web_traffic.orc'
try:
df_from_hive = pd.read_orc(orc_file_path) print(f"Брой редове, прочетени от Hive ORC файл:
{len(df_from_hive)}") print(df_from_hive.head())
# Извършете анализ или визуализация с Pandas
average_session_duration =
df_from_hive.groupby('user_id')['session_duration'].mean()
print(average_session_duration.head())
except ImportError as e:
print(f"Грешка: {e}. Уверете се, че pyarrow е инсталиран.") except FileNotFoundError:
print(f"Грешка: Файлът '{orc_file_path}' не е намерен.") except Exception as e:
print(f"Възникна грешка при четене на ORC файла: {e}")
```

Казус 3: Ефективно съхранение на междинни резултати в Spark workflow (с Pandas на edge нод)

Ситуация: Разработвате workflow, който използва Apache Spark за мащабна обработка на данни. В края на определени етапи, резултатите (които могат да бъдат по-малки подмножества от оригиналните данни) трябва да бъдат достъпни за по-нататъчен анализ или визуализация с помощта на Pandas, изпълняван на edge нод (възел извън основния Spark клъстер).

Решение:

Spark може да записва междинните резултати във ORC формат. Pandas (с `pyarrow`) може след това да прочете тези ORC файлове бързо и ефективно, без да се налага преобразуване в по-бавен формат като CSV.

В Spark (запис):

```
// Scala код на Spark
val processedData = ... // Резултат от Spark обработка като DataFrame
processedData.write.orc("hdfs://namenode:8020/user/spark/processed_data.orc")
```

В Pandas (четене):

```
import pandas as pd
orc_output_path = 'hdfs://namenode:8020/user/spark/processed_data.orc' try:
df_from_spark = pd.read_orc(orc_output_path) print(f"Брой редове, прочетени от Spark ORC файл:
{len(df_from_spark)}") print(df_from_spark.head())
# Извършете анализ или визуализация с Pandas
correlation_matrix = df_from_spark[['feature_1', 'feature_2']].corr()
print(correlation_matrix)
except ImportError as e:
print(f"Грешка: {e}. Уверете се, че pyarrow е инсталиран.") except FileNotFoundError:
print(f"Грешка: Файлът '{orc_output_path}' не е намерен.") except Exception as e:
print(f"Възникна грешка при четене на ORC файла: {e}")
```

Тези казуси илюстрират как ORC форматът, поддържан от Pandas чрез `pyarrow`, може да бъде полезен за работа с големи данни, особено в контекста на екосистемата на Hadoop и за ефективен обмен на данни между различни инструменти.

Казус 4*: Анализ на потребителски данни с богата структура

Ситуация:

Разполагате с голям набор от потребителски данни, съдържащ информация за потребителски профили и тяхната активност. Данните включват:

- `user_id`: Уникален идентификатор на потребителя.
- `name`: Име на потребителя.

- `age`: Възраст на потребителя.
- `country`: Държава на потребителя.
- `signup_date`: Дата на регистрация.
- `last_login`: Последна дата на влизане.
- `preferences`: JSON стринг, съдържащ вложени данни за предпочитанията на потребителя (например, любими категории продукти, езикови настройки).

- `activity_log`: JSON стринг, съдържащ масив от обекти, описващи действията на потребителя (например, време на действие, тип действие, детайли).

Имате нужда да съхраните тези данни ефективно и да извършвате различни анализи, включващи филтриране, агрегация и работа с вложените структури. Искате да сравните производителността на ORC формата с други често използвани формати като CSV и Parquet.

Задача:

1. Симулирайте генерирането на голям Pandas DataFrame с потребителски данни, включващ колони с различни типове, както и JSON стрингове за `preferences` и `activity_log`.
2. Запишете този DataFrame в три различни формата: CSV, Parquet и ORC.
3. Измерете времето и размера на файла за всеки от трите формата.
4. Прочетете данните обратно от всеки от трите формата в Pandas DataFrame.
5. Извършете следните аналитични задачи и измерете времето за всяка от тях при четене от всеки формат:
    - Филтриране на потребители от определена държава (например, 'USA').
    - Изчисляване на средната възраст на потребителите по държави.
    - Парсване на JSON колоната `preferences` и извличане на броя на любимите категории продукти за всеки потребител.
    - Парсване на JSON колоната `activity_log` и преброяване на броя на действията от тип 'purchase' за всеки потребител.

Забележка: За работа с ORC формат може да се наложи инсталирането на допълнителна библиотека като `pyarrow` (която поддържа и Parquet) или `orc`. В примера ще използваме `pyarrow`.

Решение:

```
import pandas as pd import numpy as np import json
import time import os
import pyarrow.parquet as pq import pyarrow.orc as orc
# 1. Симулиране на генериране на потребителски данни
num_rows = 100000 np.random.seed(42)
countries = ['USA', 'Canada', 'UK', 'Germany', 'France', 'Japan',
'Australia', 'Brazil', 'India', 'China']
activity_types = ['view', 'click', 'add_to_cart', 'purchase', 'logout']
user_data = pd.DataFrame({
'user_id': np.arange(1, num_rows + 1),
'name': [f'User {i}' for i in range(1, num_rows + 1)], 'age':
np.random.randint(18, 65, num_rows),
'country': np.random.choice(countries, num_rows), 'signup_date':
pd.to_datetime('2023-01-01') +
pd.to_timedelta(np.random.randint(0, 730, num_rows), unit='D'), 'last_login':
pd.to_datetime('2024-01-01') +
pd.to_timedelta(np.random.randint(0, 365, num_rows), unit='D'),
```

```
    'preferences': [json.dumps({'favorite_categories':
np.random.choice(['Books', 'Electronics', 'Clothing', 'Home', 'Sports'],
np.random.randint(1, 4)).tolist(),
                                    'language': np.random.choice(['en',
'fr', 'de', 'ja'])}) for _ in range(num_rows)],
    'activity_log': [json.dumps([{'timestamp': pd.to_datetime('2024-01-
01') + pd.to_timedelta(np.random.randint(0, 90, 1)[0],
unit='D').isoformat(),
                                    'type':
np.random.choice(activity_types),
                                    'details': f'Action detail
{np.random.randint(1, 10)}'} for _ in range(np.random.randint(5, 15))])
for _ in range(num_rows)]
})

# 2. Записване в различни формати
base_filename = 'user_data'
csv_file = f'{base_filename}.csv'
parquet_file = f'{base_filename}.parquet'
orc_file = f'{base_filename}.orc'

start_time = time.time()
user_data.to_csv(csv_file, index=False)
csv_size = os.path.getsize(csv_file)
csv_write_time = time.time() - start_time
print(f"CSV записан за {csv_write_time:.4f} секунди, размер: {csv_size /
(1024 * 1024):.2f} MB")

start_time = time.time()
user_data.to_parquet(parquet_file, index=False)
parquet_size = os.path.getsize(parquet_file)
parquet_write_time = time.time() - start_time
print(f"Parquet записан за {parquet_write_time:.4f} секунди, размер:
{parquet_size / (1024 * 1024):.2f} MB")

start_time = time.time()
user_data.to_orc(orc_file, index=False)
orc_size = os.path.getsize(orc_file)
orc_write_time = time.time() - start_time
print(f"ORC записан за {orc_write_time:.4f} секунди, размер: {orc_size /
(1024 * 1024):.2f} MB")
```

```
# 4. Четене от различните формати
start_time = time.time() df_csv = pd.read_csv(csv_file)
csv_read_time = time.time() - start_time
print(f"\nCSV прочетен за {csv_read_time:.4f} секунди.")

start_time = time.time()
df_parquet = pd.read_parquet(parquet_file) parquet_read_time = time.time() -
start_time
print(f"Parquet прочетен за {parquet_read_time:.4f} секунди.")

start_time = time.time()
df_orc = pd.read_orc(orc_file)
orc_read_time = time.time() - start_time

print(f"ORC прочетен за {orc_read_time:.4f} секунди.")

# 5. Извършване на аналитични задачи и измерване на времето

def analyze_data(df, format_name):

    print(f"\nАнализ на данни от формат: {format_name}")

    start_time = time.time()

    filtered_usa = df[df['country'] == 'USA']
    filter_time = time.time() - start_time

    print(f"  Филтриране на потребители от USA: {filter_time:.4f}
секунди, {len(filtered_usa)} резултата.")

    start_time = time.time()

    avg_age_by_country = df.groupby('country')['age'].mean()
    groupby_time = time.time() - start_time

    print(f"  Средна възраст по държави:\n{avg_age_by_country.head()}\n
Време за групиране: {groupby_time:.4f} секунди.")


    start_time = time.time()

    df['preferences_parsed'] = df['preferences'].apply(json.loads)
    df['num_favorite_categories'] =

df['preferences_parsed'].apply(lambda x:
len(x.get('favorite_categories', [])))

    parse_preferences_time = time.time() - start_time

    print(f"  Парсване на preferences и извличане на брой категории:

{parse_preferences_time:.4f} секунди.")

    start_time = time.time()
```

```
    df['activity_log_parsed'] = df['activity_log'].apply(json.loads)
    df['purchase_count'] = df['activity_log_parsed'].apply(lambda x:
sum(1 for event in x if event.get('type') == 'purchase'))
parse_activity_log_time = time.time() - start_time
print(f"  Парсване на activity_log и преброяване на покупки:
{parse_activity_log_time:.4f} секунди.")
print(f"  Примерни брой покупки:\n{df['purchase_count'].head()}")

analyze_data(df_csv.copy(), 'CSV') analyze_data(df_parquet.copy(), 'Parquet')
analyze_data(df_orc.copy(), 'ORC')

# Почистване на създадените файлове os.remove(csv_file)
os.remove(parquet_file) os.remove(orc_file)
```

Разбор на сложността:

- **Генериране на данни с различни типове и вложени структури:** Симулира се реален сценарий с данни, които не са само прости числови или текстови, а включват и сложни JSON структури.
- **Сравнение на различни формати:** Казусът изисква запис и четене на едни и същи данни в три различни формата (CSV, Parquet, ORC) за сравнение на ефективността по размер и скорост.
- **Измерване на производителност:** Използва се `time` модулът за измерване на времето, необходимо за запис и четене на данните, както и за извършване на аналитични задачи.
- **Работа с JSON данни в Pandas:** Демонстрира се как JSON стринговете могат да бъдат прочетени и парсвани в Pandas DataFrame за достъп до вложените данни.
- **Извършване на сложни аналитични задачи:** Включени са филтриране, групиране и прилагане на функции за работа с парсираните JSON данни.

Този казус показва предимствата на columnar формати като Parquet и ORC при работа с големи набори от данни, особено когато се извършват селективни заявки или се работи със сложни типове данни. Обикновено, columnar форматите предлагат по-добро компресиране и по-бързо четене на подмножества от колони в сравнение с row-based формати като CSV. ORC често може да предложи допълнителни оптимизации спрямо Parquet в определени сценарии. Резултатите от времевите измервания могат да варират в зависимост от размера на данните и хардуерната конфигурация.

Въпроси:

1. Какво е ORC (Optimized Row Columnar) формат и за какви среди за обработка на данни е основно предназначен?
2. Какви са основните предимства на ORC формата в сравнение с други колонни формати като Parquet? В какви специфични сценарии може да бъде по-изгоден?
3. Задължително ли е да имате инсталиран `pyarrow`, за да работите с ORC формат в Pandas? Обяснете защо.
4. Опишете процеса на записване на Pandas DataFrame във ORC файл с помощта на `df.to_orc()`. Кои са основните параметри, които могат да бъдат конфигурирани?
5. Как се чете ORC файл в Pandas DataFrame с помощта на `pd.read_orc()`? Възможно ли е да се четат само определени колони?
6. Какви алгоритми за компресия се поддържат от `df.to_orc()`? Сравнете ги по отношение на скорост и степен на компресия.
7. Как параметърът `index` в `df.to_orc()` влияе на записа на индекса на DataFrame-а в ORC файла?
8. Как `storage_options` параметърът позволява работа с ORC файлове, съхранявани в отдалечени файлови системи?
9. Как ORC форматът се интегрира с екосистемата на Apache Hadoop, особено с Apache Hive? Какви са ползите от използването на ORC в тази среда?
10. Какви са някои от потенциалните изключения или грешки, които могат да възникнат при работа с ORC файлове в Pandas?
11. Сравнете ORC и Parquet по отношение на тяхната поддръжка в Pandas, основни характеристики и сценарии на употреба.
12. В какви ситуации бихте избрали да използвате ORC вместо Parquet и обратно?

Задачи:

1. **Запис и четене с `pyarrow`:** Създайте DataFrame и го запишете във ORC файл. След това го прочетете обратно, като използвате `pd.read_orc()`. Уверете се, че данните са идентични.
2. **Четене на подмножество от колони:** Създайте DataFrame с няколко колони и го запишете във ORC файл. Прочетете го отново, като изберете само две от колоните.
3. **Изследване на компресия:** Запишете голям DataFrame във ORC формат, използвайки различни методи на компресия (`'snappy'`, `'gzip'`, `None`). Сравнете размера на получените файлове. (Може да изисквате измерване на времето за запис/четене, ако е приложимо).
4. **Работа с индекс:** Създайте DataFrame с нетривиален индекс и го запишете във ORC файл с и без запазване на индекса (`index=True` и `index=False`). Прочетете обратно и двата файла и наблюдавайте как се обработва индексът.
5. **Симулиране на четене от отдалечено хранилище:** Ако имате достъп до отдалечено хранилище (например, AWS S3), опитайте да прочетете ORC файл, използвайки `storage_options`. Ако не, можете да симулирате това, като използвате локален файл и се фокусирате върху синтаксиса на `storage_options` за HTTP/HTTPS URL (въпреки че локалните файлове ще се четат директно).

XIV. SAS файлове: (`read_sas`)

Pandas предоставя възможност за четене на файлове, създадени от статистическия софтуер SAS (Statistical Analysis System), с помощта на функцията `pd.read_sas()`. Тази функция поддържа четене на SAS Transport файлове (формат .sas7bdat) и SAS XPORT файлове (формат .xpt).

1. Основна функция:

- **`pandas.read_sas(filepath_or_buffer, format=None, index=None, encoding=None, chunksize=None, iterator=False, **kwargs)`**: Функция за четене на SAS файл в DataFrame.
  - `filepath_or_buffer`: Пътят до SAS файла (може да бъде и обект, подобен на файл).
  - `format`: Формат на SAS файла: `'infer'` (по подразбиране, опитва се да определи автоматично), `'sas7bdat'` (SAS Transport файл), или `'xport'` (SAS XPORT файл).
  - `index`: Колона(и), които да се използват като индекс на DataFrame-а.
  - `encoding`: Кодирането, което да се използва за четене на низови данни. Ако е `None`, Pandas ще опита да го определи.
  - `chunksize`: Ако е зададено цяло число, връща обект `SAS7BDATReader` или `XportReader` за итериране през файла на парчета с дадения размер.
  - `iterator`: Булева стойност. Ако е `True`, връща итератор вместо цял DataFrame. Използва се заедно с `chunksize`.
  - `**kwargs`: Допълнителни аргументи, които се предават на съответния парсер (за `sas7bdat` или `xport`).

2. Пример за четене на SAS7BDAT файл:

```
import pandas as pd

# Да предположим, че имаме SAS Transport файл 'data.sas7bdat'
try:
    df_sas7bdat = pd.read_sas('data.sas7bdat')
    print(f"Брой редове: {len(df_sas7bdat)}")
    print(df_sas7bdat.head())
except FileNotFoundError:
    print("Файлът 'data.sas7bdat' не е намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на SAS7BDAT файла: {e}")
```

3. Пример за четене на XPORT файл:

```
import pandas as pd

# Да предположим, че имаме SAS XPORT файл 'data.xpt'
try:
    df_xport = pd.read_sas('data.xpt', format='xport')
    print(f"Брой редове: {len(df_xport)}")
    print(df_xport.head())
except FileNotFoundError:
    print("Файлът 'data.xpt' не е намерен.")
except Exception as e:
```

```
    print(f"Възникна грешка при четене на SAS XPORT файла: {e}")
```

4. Важни аспекти:

- **Формат на файла:** Уверете се, че сте указали правилния формат (`'sas7bdat'` или `'xport'`) или оставете Pandas да го определи (`'infer'`).
- **Кодиране:** SAS файловете могат да бъдат записани с различни кодирания. Ако срещате проблеми с четенето на низови данни, опитайте да укажете кодиране (например, `'latin-1'`, `'utf-8'`).
- **Големи файлове:** За много големи SAS файлове, използването на `chunksize` или `iterator=True` може да бъде по-ефективно, тъй като позволява обработка на данните на парчета, без да се зарежда целият файл в паметта.
- **Индекс:** Можете да зададете една или повече колони от SAS файла като индекс на DataFrame-а.

5. Писане в SAS файлове

За съжаление, **Pandas в момента не предоставя вградена функционалност за записване на данни във SAS файлове (.sas7bdat или .xpt)**. Функцията `pd.read_sas()` е предназначена само за четене на SAS файлове в DataFrame.

Ако имате нужда да запишете DataFrame във SAS формат, ще трябва да използвате външни библиотеки или да обмислите алтернативни подходи:

1) **Използване на `pyreadstat` (запис само в .sas7bdat):**

Библиотеката `pyreadstat` е фокусирана основно върху четене на статистически файлове (SAS, SPSS, Stata), но поддържа и записване във формат `.sas7bdat`.

Първо, инсталирайте библиотеката:

```
pip install pyreadstat
```

След това можете да запишете DataFrame във `.sas7bdat` файл:

```
import pandas as pd
import pyreadstat

# Да предположим, че имате DataFrame df
data = {'col1': [1, 2, 3], 'col2': ['A', 'B', 'C']}
df = pd.DataFrame(data)

try:
    pyreadstat.write_sas7bdat(df, 'output.sas7bdat')
    print("DataFrame-ът е записан успешно в output.sas7bdat")
except Exception as e:
    print(f"Възникна грешка при запис във SAS7BDAT файл: {e}")
```

!!!Важно за `pyreadstat.write_sas7bdat()`:

- Библиотеката има някои ограничения по отношение на типовете данни и дължината на низовете, които могат да бъдат записани коректно във SAS формат.
- Проверете документацията на `pyreadstat` за подробности относно поддържаните типове данни и възможни ограничения.

2) **Експортиране в междинен формат и използване на SAS за запис:**

Друг подход е да експортирате DataFrame-а в междинен формат, който може лесно да бъде прочетен от SAS (например, CSV файл), и след това да използвате SAS процедури (например, `PROC IMPORT` и `PROC EXPORT` или `DATA` стъпки) за да запишете данните във SAS формат.

Python (експортиране в CSV):

```
import pandas as pd

# Да предположим, че имате DataFrame df
data = {'col1': [1, 2, 3], 'col2': ['A', 'B', 'C']}
df = pd.DataFrame(data)

df.to_csv('output.csv', index=False)
print("DataFrame-ът е експортиран в output.csv")
```

SAS (четене на CSV и запис във SAS формат):

```
/* Четене на CSV файл */
PROC IMPORT DATAFILE="output.csv"
            OUT=mydata
            DBMS=CSV;
RUN;

/* Запис на данните във SAS Transport файл (.sas7bdat) */
PROC COPY IN=WORK OUT=SASLIB;
  SELECT mydata;
RUN;

/* (Опционално) Запис на данните във SAS XPORT файл (.xpt) */
PROC XPORT OUTFILE="output.xpt" MEMTYPE=DATASET;
  SELECT mydata;
RUN;
```

(В този пример `SASLIB` трябва да бъде дефинирана като библиотека, където искате да запишете `.sas7bdat` файла.)

2) **Използване на други езици или инструменти:**

Ако работите в среда, където имате достъп до други езици за програмиране (например, R с пакета `haven`) или специализирани инструменти за работа със статистически данни, може да разгледате възможностите за записване във SAS формат чрез тях.

6. Допълнителни аспекти:

1) **Автоматично определяне на формата (`format='infer'`):** По подразбиране, Pandas се опитва автоматично да определи дали файлът е във формат `.sas7bdat` или `.xpt`. Въпреки че това често работи добре, в някои случаи може да е по-надеждно изрично да зададете формата, ако го знаете.
2) **Работа с различни кодирания (`encoding`):** SAS файловете могат да бъдат създадени с различни кодирания на знаците. Ако при четенето се появят проблеми с неразпознати символи или грешно интерпретиран текст, може да се наложи да експериментирате с различни кодирания като `'latin-1'`, `'utf-8'`, `'cp1252'` (често използвано за Windows) или други, специфични за езика или региона, в който е създаден файлът.
3) **Четене на големи файлове на парчета (`chunksize`, `iterator`):** Когато работите с много големи SAS файлове, зареждането на целия файл в паметта може да бъде неефективно или невъзможно. Параметърът `chunksize` позволява да четете файла на по-малки парчета (chunks) от редове, които се връщат като DataFrame. Ако зададете `iterator=True`, `read_sas()` ще върне итератор, който можете да използвате за последователно обработване на тези парчета.

```
# Четене на файл на парчета от по 1000 реда
try:
    reader = pd.read_sas('large_data.sas7bdat', chunksize=1000)
    for chunk in reader:
        print(f"Обработка на парче с {len(chunk)} реда...")
        # Извършете обработка на текущото парче (chunk)
except FileNotFoundError:
    print("Файлът 'large_data.sas7bdat' не е намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на SAS файла: {e}")
```

4) **Задаване на индекс (`index`):** Можете да използвате една или повече колони от SAS файла като индекс на DataFrame-а, като подадете името на колоната (или списък от имена) на параметъра `index`.
5) **Допълнителни аргументи (`**kwargs`):** Функцията `read_sas()` приема допълнителни аргументи, които се предават на съответния backend (за `.sas7bdat` се използва `sas7bdat` библиотеката, а за `.xpt` - вградена функционалност). Проверете документацията на тези библиотеки за възможни специфични опции.

7. Потенциални проблеми:

1. **Липса на файла (`FileNotFoundError`):** Най-честата грешка е, когато файлът, указан в `filepath_or_buffer`, не съществува.
2. **Грешно определен формат:** Ако форматът е грешно определен (автоматично или изрично), може да възникнат грешки при парсването на файла или да се прочетат некоректни данни.

3. **Проблеми с кодирането (`UnicodeDecodeError`):** Неправилното кодиране може да доведе до грешки при декодиране на низови данни, особено ако файлът съдържа символи, които не са съвместими с избраното кодиране.
4. **Неподдържани или неочаквани типове данни:** Въпреки че Pandas се стреми да обработва SAS типовете данни коректно, е възможно да има случаи на неочаквано поведение или загуба на прецизност при много специфични или нестандартни SAS формати на данни.
5. **Повредени файлове:** Ако SAS файлът е повреден, `read_sas()` може да върне грешки или да прочете само част от данните.
6. **Ограничения на паметта:** При много големи файлове, дори и при четене на парчета, общата обработка на данните може да изисква значително количество памет.

8. Най-добри практики:

- Ако знаете формата на SAS файла, задайте го изрично с параметъра `format`.
- Ако имате съмнения относно кодирането, опитайте с `'latin-1'` като първа стъпка, тъй като често се използва за SAS файлове. Ако това не работи, проверете документацията на файла или се консултирайте с източника на данните.
- За големи файлове, използвайте `chunksize` или `iterator=True`, за да избегнете проблеми с паметта.
- Проверявайте внимателно прочетените данни, за да се уверите, че са интерпретирани правилно, особено низовите стойности и типовете данни.
- Ако срещате неочаквани грешки, проверете версията на Pandas, която използвате, тъй като поддръжката за SAS файлове може да се е подобрила в по-нови версии.

Казус 1: Четене на демографски данни от здравно проучване

Ситуация: Вие сте изследовател в областта на общественото здраве и получавате данни от голямо национално здравно проучване, съхранени като SAS Transport файл (`demographics.sas7bdat`). Файлът съдържа демографска информация за участниците (възраст, пол, образование, местоживеене и др.). Трябва да анализирате тези данни с Pandas.

Решение:

Използвайте `pd.read_sas()` за директно четене на `.sas7bdat` файла в Pandas DataFrame. След това можете да използвате стандартните Pandas инструменти за почистване, преобразуване и анализ на данните.

```
import pandas as pd

try:
    df_demographics = pd.read_sas('demographics.sas7bdat')
    print(f"Брой участници в проучването: {len(df_demographics)}")
    print(df_demographics.head())

    # Извършете анализ: например, средна възраст по пол
    average_age_by_gender =
df_demographics.groupby('gender')['age'].mean()
    print("\nСредна възраст по пол:")
    print(average_age_by_gender)
```

```
except FileNotFoundError:
    print("Файлът 'demographics.sas7bdat' не е намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на SAS файла: {e}")
```

Казус 2: Интегриране на исторически клинични данни от SAS XPORT файл

Ситуация: Работите в болнична администрация и трябва да интегрирате исторически клинични данни, съхранени в стар SAS XPORT файл (`clinical_data.xpt`), в съвременна система за анализ на данни, базирана на Python и Pandas.

Решение:

Използвайте `pd.read_sas()` с изрично зададен формат `'xport'` за четене на `.xpt` файла в DataFrame. След това можете да преобразувате данните и да ги интегрирате с други източници на данни.

```
import pandas as pd

try:
    df_clinical = pd.read_sas('clinical_data.xpt', format='xport')
    print(f"Брой записи в клиничните данни: {len(df_clinical)}")
    print(df_clinical.head())

    # Извършете преобразуване: например, преименуване на колони
    df_clinical.rename(columns={'patient_id': 'ID_пациент',
'diagnosis_code': 'Код_диагноза'}, inplace=True)
    print("\nПреименувани колони:")
    print(df_clinical.head())

except FileNotFoundError:
    print("Файлът 'clinical_data.xpt' не е намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на SAS файла: {e}")
```

Казус 3*: Обработка на много голям SAS файл с неясно кодиране и специфична структура

Ситуация: Получавате много голям SAS Transport файл (`large_dataset.sas7bdat`) от външен партньор. Не сте сигурни за точното кодиране, и при първите опити за четене с подразбиране се появяват проблеми с неразпознати символи. Освен това, файлът е твърде голям, за да се побере изцяло в паметта на вашата работна станция. Има информация, че първите няколко реда съдържат метаданни, които не са част от същинствените данни. Трябва да прочетете данните, като пропуснете метаданните и се справите с евентуалното нестандартно кодиране, без да претоварвате паметта.

Решение:

1) **Опит за определяне на кодирането:** Опитайте да прочетете малка част от файла с различни кодирания (`'latin-1'`, `'cp1252'`, `'utf-8'`) с малък `nrows` параметър (ако е приложим, въпреки че `read_sas` директно няма `nrows`, може да се използва с `chunksize` и спиране след първия чанк) или като прочетете файла на един чанк и разгледате резултата, за да идентифицирате евентуални проблеми с кодирането.
2) **Четене на парчета с предполагаемото кодиране:** Използвайте параметъра `chunksize` за итериране през файла на по-малки парчета. Задайте кодирането, което сте определили в предходната стъпка.
3) **Пропускане на метаданни (ако е необходимо):** Ако знаете броя на редовете с метаданни в началото, можете да използвате брояч при итерирането през chunks и да започнете обработката само след като тези редове бъдат пропуснати.

```
import pandas as pd

try:
    # Първи опит за определяне на кодиране (може да се наложи повече експериментиране)
    possible_encoding = 'latin-1'  # Примерно кодиране

    reader = pd.read_sas('large_dataset.sas7bdat', encoding=possible_encoding, chunksize=1000)
    data_chunks = []
    rows_processed = 0
    metadata_rows_to_skip = 5  # Примерно броя редове метаданни

    for chunk in reader:
        if rows_processed >= metadata_rows_to_skip:
            data_chunks.append(chunk)
        rows_processed += len(chunk)
        # Можете да добавите условие за спиране след прочитане на достатъчно данни за тестване

    if data_chunks:
        df_large = pd.concat(data_chunks)
        print(f"Успешно прочетени {len(df_large)} реда данни.")
        print(df_large.head())
        # Продължете с анализа
    else:
        print("Не бяха прочетени данни след пропускане на метаданните.")

except FileNotFoundError:
    print("Файлът 'large_dataset.sas7bdat' не е намерен.")
except UnicodeDecodeError:
    print(f"Грешка при декодиране с кодиране '{possible_encoding}'. Опитайте друго кодиране.")
except Exception as e:
    print(f"Възникна неочаквана грешка при четене на SAS файла: {e}")
```

Въпроси:

1. Каква е основната цел на функцията `pd.read_sas()` в Pandas? Какви типове SAS файлове може да чете?
2. Обяснете разликата между SAS Transport файлове (.sas7bdat) и SAS XPORT файлове (.xpt). Как се указва форматът при четене с `pd.read_sas()`?
3. Каква е ролята на параметъра `encoding` в `pd.read_sas()`? Защо е важно да се указва правилното кодиране и какви проблеми могат да възникнат при грешно кодиране?
4. В какви ситуации би било полезно да използвате параметрите `chunksize` или `iterator=True` при четене на SAS файлове? Как се работи с върнатия обект в тези случаи?
5. Как можете да зададете една или повече колони от SAS файла като индекс на DataFrame-а при четене?
6. Какво означава `format='infer'` в `pd.read_sas()` и в какви случаи може да не е надеждно да се разчита на автоматичното определяне на формата?
7. Опишете някои от потенциалните проблеми или грешки, които могат да възникнат при четене на SAS файлове с Pandas, и как бихте могли да ги отстраните.
8. Възможно ли е да се записват Pandas DataFrame-и във SAS файлове с вградените функции на Pandas? Ако не, какви са алтернативните подходи?
9. Какви са предимствата на използването на Pandas за четене на SAS файлове в сравнение с други методи или инструменти?
10. Какви допълнителни аргументи могат да бъдат подадени на `pd.read_sas()` чрез `**kwargs` и за какво могат да бъдат използвани?

Задачи:

1. **Четене на SAS7BDAT файл:** Намерете или създайте примерен `.sas7bdat` файл. Прочетете го в Pandas DataFrame и покажете първите няколко реда и информация за колоните.
2. **Четене на XPORT файл:** Намерете или създайте примерен `.xpt` файл. Прочетете го в Pandas DataFrame, като изрично укажете формата, и покажете първите няколко реда.
3. **Работа с кодиране:** Създайте (или модифицирайте съществуващ) SAS файл с не-ASCII символи и го запишете с различно кодиране (например, използвайте SAS, ако имате достъп). Опитайте да го прочетете с Pandas, използвайки грешно и правилно кодиране, и наблюдавайте резултата.
4. **Четене на голям файл на парчета:** Създайте (или използвайте голям) SAS файл. Прочетете го на парчета с определен `chunksize` и обработете всеки парче (например, изчислете средна стойност на една от колоните за всеки парче).
5. **Задаване на индекс:** Прочетете SAS файл и задайте една от колоните като индекс на DataFrame-а. Покажете индекса и достъпете данни по индекс.

XV. SPSS файлове: (read_spss) - *Изисква pyreadstat*

SPSS (.sav) е често срещан формат за съхранение на данни от статистически анализи и социални науки. Библиотеката `pyreadstat` предоставя удобен начин за четене на SPSS файлове в Pandas DataFrame, като запазва важна метаинформация като етикети на променливи и стойности, както и информация за липсващи стойности.

1. Инсталация на `pyreadstat`

Преди да започнем, уверете се, че библиотеката `pyreadstat` е инсталирана във вашата Python среда. Ако не е, можете да я инсталирате с помощта на pip:

```
pip install pyreadstat
```

2. Четене на SPSS файлове с `read_sav()`

Основната функция за четене на SPSS файлове е `pyreadstat.read_sav()`. Тя приема пътя до `.sav` файла като аргумент и връща кортеж, съдържащ:

1) **DataFrame:** Pandas DataFrame, съдържащ данните от SPSS файла.
2) **Метаданни:** Обект, съдържащ информация за променливите, техните етикети, етикетите на стойностите и дефинициите за липсващи стойности.

Ето основен пример за четене на SPSS файл:

```
import pyreadstat
import pandas as pd

try:
    df, meta = pyreadstat.read_sav('вашият_файл.sav')
    print("Данните са прочетени успешно:")
    print(df.head())
    print("\nМетаданни:")
    print(meta)
except FileNotFoundError:
    print("Файлът не беше намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на файла: {e}")
```

Обектът `meta` съдържа следните полезни атрибути:

- `variable_labels`: Речник, където ключовете са имената на променливите, а стойностите са техните етикети.

- `variable_value_labels`: Речник, където ключовете са имената на променливите, а стойностите са речници, съпоставящи стойностите на променливите с техните етикети.
- `missing_user_value`: Речник, където ключовете са имената на променливите, а стойностите са списъци с кодовете, които в SPSS са дефинирани като липсващи за съответната променлива.
- `column_names_to_labels`: Речник, който съпоставя имената на колоните с техните етикети.

а) Достъп до метаданни

След като прочетете файла, можете да използвате обекта `meta`, за да получите информация за променливите и техните стойности:

```
if 'Q1_Gender' in meta.variable_labels:
    print(f"Етикет на променливата Q1_Gender:
{meta.variable_labels['Q1_Gender']}")

if 'Q1_Gender' in meta.variable_value_labels:
    print(f"Етикети на стойностите за Q1_Gender:
{meta.variable_value_labels['Q1_Gender']}")

if 'Q1_Gender' in meta.missing_user_value:
    print(f"Липсващи стойности за Q1_Gender:
{meta.missing_user_value['Q1_Gender']}")
```

б) Обработка на липсващи стойности

`pyreadstat` не конвертира автоматично дефинираните в SPSS липсващи стойности в `NaN` на Pandas. Ще трябва да го направите ръчно, ако е необходимо:

```
import numpy as np

for col, missing_codes in meta.missing_user_value.items():
    if col in df.columns and isinstance(missing_codes, list):
        df[col] = df[col].replace(missing_codes, np.nan)

print("\nDataFrame след обработка на липсващи стойности:")
print(df.head())
```

в) Използване на етикети на стойностите

Етикетите на стойностите могат да бъдат полезни за разбиране на кодировката на категорийните променливи. Можете да ги използвате за преобразуване на числовите кодове в техните текстови етикети, ако е необходимо:

```
for col, value_labels in meta.variable_value_labels.items():
    if col in df.columns:
        df[col + '_label'] = df[col].map(value_labels)

print("\nDataFrame с добавени етикети на стойностите:")
print(df.head())
```

2. Запазване на DataFrame в SPSS формат (`write_sav()`)

Библиотеката `pyreadstat` също позволява записване на Pandas DataFrame обратно в SPSS `.sav` файл, като може да запази или зададе метаданни:

```
# Примерни метаданни (можете да ги модифицирате или вземете от прочетен файл)
output_meta = pyreadstat.metadata_container(
    column_names=['колона1', 'колона2'],
    column_labels=['Етикет на колона 1', 'Етикет на колона 2'],
    variable_value_labels={'колона1': {1: 'Етикет 1', 2: 'Етикет 2'}}
)

try:
    pyreadstat.write_sav(df, 'нов_файл.sav',
column_labels=output_meta.column_labels,
variable_value_labels=output_meta.variable_value_labels)
    print("DataFrame-ът е успешно записан в 'нов_файл.sav'.")
except Exception as e:
    print(f"Възникна грешка при запис на файла: {e}")
```

3. Допълнителни аспекти

При работа със SPSS файлове и библиотеката `pyreadstat` могат да възникнат различни ситуации и е полезно да сме подготвени за тях.

а) Кодиране на символите

SPSS файловете могат да бъдат записани с различни кодирания на символите. По подразбиране, `pyreadstat` се опитва да определи кодирането автоматично. Ако имате проблеми с четенето на текст (например, некоректно показване на кирилица или други специални символи), може да се наложи да укажете кодирането изрично чрез аргумента `encoding` на функцията `read_sav()`:

```
try:
    df, meta = pyreadstat.read_sav('вашият_файл.sav', encoding='windows-1251') # Пример за кодировка
    print("Данните са прочетени успешно с указано кодиране.")
except Exception as e:
    print(f"Възникна грешка при четене на файла с указано кодиране: {e}")
```

Често използвани кодирания включват `'utf-8'`, `'latin-1'`, `'windows-1251'`. Ако не сте сигурни кое е правилното, може да се наложи да експериментирате или да проверите информацията за файла (ако е налична).

б) Работа с дати и часове

SPSS има специфичен начин за съхранение на дати и часове. `pyreadstat` обикновено ги конвертира в Pandas `datetime64` тип. Въпреки това, в зависимост от формата в SPSS, може да се наложи

допълнителна обработка, за да се гарантира правилната интерпретация. Проверете `dtypes` на вашия DataFrame след четене, за да видите как са интерпретирани датите и часовете. Ако е необходимо, можете да използвате Pandas функции като `pd.to_datetime()` с подходящ формат.

в) Големи файлове

При работа с много големи SPSS файлове, зареждането на целия файл в паметта може да бъде неефективно или да доведе до грешки. `pyreadstat` чете целия файл наведнъж. Ако се сблъскате с проблеми с паметта, може да се наложи да разгледате други инструменти или подходи за обработка на големи файлове, които работят на порции (chunking), но `pyreadstat` не предлага такава функционалност директно.

г) Нестандартни SPSS файлове

В редки случаи може да попаднете на SPSS файлове със нестандартни или повредени структури, които `pyreadstat` не може да прочете. В такива ситуации, съобщението за грешка от `pyreadstat` може да даде насоки за проблема. Възможно е да се наложи да използвате друг софтуер (например, самия SPSS) за конвертиране или поправяне на файла.

д) Записване с метаданни

Както беше показано в основния урок, при записване обратно в `.sav` формат с `write_sav()`, можете да предоставите метаданни като етикети на колони и стойности. Важно е да се отбележи, че не всички аспекти на оригиналните метаданни могат да бъдат запазени или пресъздадени при запис. Проверете документацията на `pyreadstat` за подробности относно поддържаните метаданни при запис.

4. Изключения при четене

При опит за четене на SPSS файл могат да възникнат различни изключения:

- **`FileNotFoundError`:** Файлът, указан в пътя, не съществува.
- **Грешки, свързани с кодирането:** Ако кодирането е неправилно или файлът е повреден, може да възникнат грешки при декодиране на текста.
- **Грешки, свързани със структурата на файла:** Ако `.sav` файлът е повреден или има неочаквана структура, `pyreadstat` може да не успее да го парсва и да върне грешка. Съобщението за грешка често може да даде информация за естеството на проблема.
- **Други `Exception`:** Възможни са и други неочаквани грешки по време на процеса на четене. Винаги е добра практика да обвивате кода за четене в `try...except` блок, за да обработите потенциални проблеми.

5. Съвети за отстраняване на проблеми

- **Проверете пътя до файла:** Уверете се, че сте предоставили правилния път до `.sav` файла.
- **Опитайте с различни кодирания:** Ако има проблеми с текста, опитайте да укажете различни кодирания.
- **Проверете файла с друг софтуер:** Отворете файла със SPSS или друг съвместим софтуер, за да се уверите, че е валиден и да разгледате неговите свойства (например, кодиране).
- **Консултирайте документацията на `pyreadstat`:** Официалната документация на библиотеката може да съдържа информация за специфични проблеми и решения.

- **Търсете помощ онлайн:** Ако срещнете необичаен проблем, потърсете информация в онлайн форуми или задайте въпрос в Stack Overflow, като предоставите възможно най-много детайли за грешката и файла (ако е възможно да споделите анонимизирана версия).

Разбирането на тези допълнителни аспекти и възможни изключения ще ви помогне да работите по-ефективно и да отстранявате проблеми при взаимодействието със SPSS файлове в Python с помощта на `pyreadstat`.

Казус 1: Интегриране на данни от множество проучвания с различни кодировки и липсващи стойности

Ситуация:

Работите в изследователски екип и трябва да обедините данни от три различни социологически проучвания, съхранени като SPSS файлове (`survey1.sav`, `survey2.sav`, `survey3.sav`). Тези проучвания са проведени в различни периоди от време и е възможно да използват различни кодировки на символите и различни кодове за обозначаване на липсващи стойности за едни и същи въпроси. Например, за отговор "Не знам" в едно проучване може да се използва код 99, в друго - -9, а в трето - да е дефиниран като системно липсваща стойност в SPSS (което `pyreadstat` може да интерпретира като `NaN` по подразбиране).

Задача:

1. Прочетете всеки от трите SPSS файла в отделни Pandas DataFrame-и, като се опитате да определите автоматично кодирането. Ако възникнат проблеми с текста, опитайте с често срещани кодировки като `'latin-1'` или `'windows-1251'`.
2. Разгледайте метаданните (етикети на променливи и стойности, липсващи стойности) за всеки DataFrame.
3. Идентифицирайте общите въпроси (променливи) между трите проучвания (например, "Възраст", "Пол", "Образование").
4. Стандартизирайте представянето на липсващите стойности за общите променливи във всички DataFrame-и, като ги замените с `np.nan` на Pandas. За целта, използвайте информацията от `meta.missing_user_value` за всеки файл.
5. Ако е необходимо, стандартизирайте етикетите на стойностите за общите категорийни променливи (например, "Мъж"/"Жена" да бъдат кодирани еднакво във всички DataFrame-и). Това може да изисква създаване на mapping речници въз основа на `meta.variable_value_labels`.
6. Обединете трите DataFrame-а в един голям DataFrame за общите променливи.
7. Изведете обобщена статистика (например, средна възраст, честотно разпределение по пол и образование) за обединения набор от данни.

Решение:

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

def read_spss_with_fallback(file_path):
    """
    Опитва се да прочете SPSS файл с автоматично кодиране,
```

```
    ако възникне грешка, пробва с 'latin-1' и 'windows-1251'.
    Връща DataFrame и метаданни или None, None при грешка.
    """
    try:
        df, meta = pyreadstat.read_sav(file_path)
        print(f"Успешно прочетен: {file_path} (автоматично кодиране)")
        return df, meta
    except Exception as e_auto:
        try:
            df, meta = pyreadstat.read_sav(file_path, encoding='latin-
1')
            print(f"Успешно прочетен: {file_path} (latin-1)")
            return df, meta
        except Exception as e_latin1:
            try:
                df, meta = pyreadstat.read_sav(file_path,
encoding='windows-1251')
                print(f"Успешно прочетен: {file_path} (windows-1251)")
                return df, meta
            except Exception as e_win1251:
                print(f"Грешка при четене на {file_path}:")
                print(f"  Автоматично: {e_auto}")
                print(f"  latin-1: {e_latin1}")
                print(f"  windows-1251: {e_win1251}")
                return None, None

# 1. Четене на файловете
file_paths = ['survey1.sav', 'survey2.sav', 'survey3.sav']
dfs = []
metas = []

for path in file_paths:
    df, meta = read_spss_with_fallback(path)
    if df is not None:
        dfs.append(df)
        metas.append(meta)

# 2. Разглеждане на метаданните (примерно за първия файл)
if metas:
    print("\nМетаданни за първия файл:")
    print("Етикети на променливите:", metas[0].variable_labels)
    print("Етикети на стойностите:", metas[0].variable_value_labels)
    print("Липсващи стойности:", metas[0].missing_user_value)

# 3. Идентифициране на общите въпроси (примерни общи променливи)
common_vars = ['Възраст', 'Пол', 'Образование'] # Заменете с реалните
общи имена

# 4. Стандартизиране на липсващите стойности
processed_dfs = []
for i, df in enumerate(dfs):
    df_copy = df.copy()
    meta = metas[i]
    for var in common_vars:
```

```
        if var in meta.missing_user_value and var in df_copy.columns:
            missing_codes = meta.missing_user_value[var]
            df_copy[var] = df_copy[var].replace(missing_codes, np.nan)
    processed_dfs.append(df_copy[common_vars]) # Вземаме само общите
променливи

# 5. Стандартизиране на етикетите на стойностите (пример за 'Пол')
gender_mapping = {
    1: 'Мъж',
    2: 'Жена',
    99: np.nan, # Ако 'Не знам' е кодирано като 99 в някои файлове
    -9: np.nan  # Ако 'Отказал' е кодирано като -9 в други
}

processed_dfs_standardized = []
for df in processed_dfs:
    df_copy = df.copy()
    if 'Пол' in df_copy.columns:
        # Прилагаме mapping, като се справяме с възможни липсващи
стойности
        df_copy['Пол'] = df_copy['Пол'].map(lambda x:
gender_mapping.get(x, np.nan))
    processed_dfs_standardized.append(df_copy)

# Подобна логика може да се приложи и за 'Образование' или други
категорийни променливи,
# като се създадат съответните mapping речници въз основа на
метаданните.

# 6. Обединяване на DataFrame-ите
if processed_dfs_standardized:
    merged_df = pd.concat(processed_dfs_standardized, ignore_index=True)
    print("\nОбединен DataFrame:")
    print(merged_df.head())

    # 7. Извеждане на обобщена статистика
    print("\nОбобщена статистика:")
    print("\nСредна възраст:", merged_df['Възраст'].mean())
    print("\nЧестотно разпределение по пол:")
    print(merged_df['Пол'].value_counts(dropna=False))
    print("\nЧестотно разпределение по образование:")
    print(merged_df['Образование'].value_counts(dropna=False))
else:
    print("\nНе бяха прочетени успешно файлове, обединяването не е
възможно.")
```

Разбор на решението:

1. **Четене с резервни кодировки:** Функцията `read_spss_with_fallback` се опитва да прочете всеки файл с автоматично кодиране и ако възникне грешка, пробва с `'latin-1'` и `'windows-1251'`. Това помага за справяне с потенциални разлики в кодирането.

2. **Разглеждане на метаданни:** Кодът (в коментари, тъй като няма реални файлове) показва как може да се достъпи информацията за етикети и липсващи стойности от обекта `meta`. В реална ситуация, тази информация ще бъде ключова за следващите стъпки.
3. **Идентифициране на общи променливи:** Променливата `common_vars` трябва да бъде заменена с реалните имена на общите въпроси, които сте идентифицирали между проучванията.
4. **Стандартизиране на липсващите стойности:** Итерира се през общите променливи и за всеки DataFrame, липсващите кодове, дефинирани в метаданните, се заменят с `np.nan`.
5. **Стандартизиране на етикетите на стойностите:** За пример е дадена стандартизация на променливата 'Пол'. Създаден е речник `gender_mapping`, който съпоставя различните възможни кодировки (1, 2, 99, -9) с общи стойности ('Мъж', 'Жена', `np.nan`). Подобен подход трябва да се приложи и за други категорийни променливи, като се вземат предвид етикетите на стойностите от метаданните на всеки файл.
6. **Обединяване на DataFrame-ите:** След като данните са стандартизирани (по отношение на липсващи стойности и евентуално етикети на стойности), `pd.concat()` се използва за обединяване на DataFrame-ите по редове.
7. **Извеждане на обобщена статистика:** Накрая, се изчисляват основни статистически мерки за обединения набор от данни.

Важни бележки:

- Това решение предполага, че знаете имената на общите променливи. В реална ситуация, може да се наложи да ги идентифицирате, като разгледате `meta.variable_labels` на всеки файл и потърсите съвпадения или сходни описания.
- Стандартизирането на етикетите на стойностите може да бъде по-сложно, ако кодировката е много различна между проучванията. В такива случаи, може да се наложи по-детайлно разглеждане на `meta.variable_value_labels` и създаване на по-сложни mapping правила.
- Възможно е да има и разлики в самите имена на променливите между проучванията за един и същ въпрос (например, "Възраст на респондента" в един и "Години" в друг). В такъв случай, ще трябва да направите и преименуване на колоните преди обединяването.

Това решение предоставя рамка за справяне с първия казус. В реална ситуация, ще трябва да го адаптирате според специфичните характеристики на вашите файлове с данни.

Казус 2: Анализ на данни с многоезични етикети и стойности

Ситуация:

Работите с международна изследователска организация и получавате SPSS файл (`global_survey.sav`) от проучване, проведено в няколко държави. Етикетите на променливите и стойностите във файла са предоставени на няколко езика (например, английски, френски, немски). `pyreadstat` може да прочете тези етикети, но те са съхранени в метаданните без ясна индикация за езика.

Задача:

1. Прочетете SPSS файла и разгледайте `meta.variable_labels` и `meta.variable_value_labels`. Забележете, че за някои променливи може да има етикети на различни езици, вероятно разделени със специален символ или по друг начин кодирани.

2. Разработете стратегия за извличане на етикетите на един конкретен език (например, английски), ако е възможно да се идентифицират (например, ако има конвенция за именуване на етикетите или ако един език е по подразбиране).
3. Ако етикетите са смесени или не могат лесно да бъдат разделени по език, създайте функция, която да показва наличните етикети за избрана променлива и да позволява на потребителя ръчно да избере или интерпретира правилния етикет.
4. Използвайте избраните или интерпретираните етикети, за да направите по-разбираеми резултатите от анализа (например, при именуване на осите на графики или при представяне на таблици с честоти).

Решение:

```
import pyreadstat
import pandas as pd

def read_spss_and_explore_labels(file_path):
    """
    Прочита SPSS файл и връща DataFrame и метаданни.
    Извежда примерни етикети на променливи и стойности.
    """
    try:
        df, meta = pyreadstat.read_sav(file_path)
        print(f"Успешно прочетен: {file_path}")
        print("\nПримерни етикети на променливи (първите 5):")
        for i, (var, label) in enumerate(meta.variable_labels.items()):
            if i < 5:
                print(f"  {var}: {label}")
        print("\nПримерни етикети на стойности (първите 5 променливи с
етикети):")
        for i, (var, labels) in
enumerate(meta.variable_value_labels.items()):
            if i < 5:
                print(f"  {var}: {labels}")
        return df, meta
    except FileNotFoundError:
        print(f"Файлът '{file_path}' не беше намерен.")
        return None, None
    except Exception as e:
        print(f"Грешка при четене на файла '{file_path}': {e}")
        return None, None

def extract_english_label(label):
    """
    Опитва се да извлече английски етикет по конвенция (пример:
'Question (EN)', 'Question [EN]').
    Връща намерения английски етикет или оригиналния, ако не бъде
намерен.
    """
    if '(EN)' in label:
        return label.split('(EN)')[0].strip()
    elif '[EN]' in label:
        return label.split('[EN]')[0].strip()
    return label
```

```
def display_and_select_label(variable, labels):
    """
    Показва наличните етикети за променлива и позволява на потребителя да избере един.
    Връща избрания етикет.
    """
    print(f"\nНалични етикети за променливата '{variable}':")
    for i, label in enumerate(labels):
        print(f"  [{i+1}] {label}")

    while True:
        try:
            choice = int(input("Изберете номер на етикет или въведете 'other' за ръчно въвеждане: "))
            if 1 <= choice <= len(labels):
                return labels[choice - 1]
            elif input_label := input("Въведете ръчно етикет: "):
                return input_label
            else:
                print("Невалиден избор.")
        except ValueError:
            print("Моля, въведете число или 'other'.")

# 1. Прочитане на файла и разглеждане на етикетите
file_path = 'global_survey.sav' # Заменете с реалния път до файла
df, meta = read_spss_and_explore_labels(file_path)

if df is not None:
    english_variable_labels = {}
    mixed_language_variables = {}

    # 2. Стратегия за извличане на английски етикети (по конвенция)
    for var, label in meta.variable_labels.items():
        english_label = extract_english_label(label)
        english_variable_labels[var] = english_label
        if english_label == label and any(lang_code in label for lang_code in ['(FR)', '[DE]', '(ES)']):
            mixed_language_variables[var] = [label] # Събираме за ръчна обработка

    print("\nАнглийски етикети на променливите (извлечени по конвенция):")
    for var, label in english_variable_labels.items():
        if label != meta.variable_labels.get(var):
            print(f"  {var}: Оригинален='{meta.variable_labels[var]}', Извлечен='{label}'")

    # 3. Ръчна обработка на променливи със смесени езици
    manual_labels = {}
    if mixed_language_variables:
        print("\nПроменливи със смесени езици, нуждаещи се от ръчна обработка:")
        for var, labels in mixed_language_variables.items():
```

```
            # В реална ситуация, може да има повече от един етикет в
списъка,
            # ако различни езици са слепени в един стринг.
            # Тук предполагаме, че имаме само оригиналния многоезичен
етикет.
            selected_label = display_and_select_label(var,
[meta.variable_labels[var]])
            manual_labels[var] = selected_label

    # Комбиниране на автоматично извлечени и ръчно избрани етикети
    final_variable_labels = english_variable_labels.copy()
    final_variable_labels.update(manual_labels)

    print("\nФинални етикети на променливите:")
    for var, label in final_variable_labels.items():
        print(f"  {var}: {label}")

    # Подобен процес може да се приложи и за етикетите на стойностите
(meta.variable_value_labels)
    # Ако етикетите на стойностите също съдържат езикови маркери, може
да се създаде
    # функция за извличане на етикети на конкретен език за всяка
стойност.

    english_value_labels = {}
    for var, value_labels in meta.variable_value_labels.items():
        english_value_labels[var] = {val: extract_english_label(label)
for val, label in value_labels.items()}

    print("\nПримерни английски етикети на стойностите (първите 5
променливи):")
    for i, (var, labels) in enumerate(english_value_labels.items()):
        if i < 5:
            print(f"  {var}: {labels}")

    # 4. Използване на избраните етикети при анализ (примерно)
    if 'country' in df.columns and 'country' in final_variable_labels:
        print(f"\nАнализ на база '{final_variable_labels['country']}'
(първите 5 записа):")
        print(df['country'].head())

        # Ако имаме етикети на стойностите за 'country'
        if 'country' in english_value_labels:
            df['country_label_en'] =
df['country'].map(english_value_labels['country'])
            print("\nЧестотно разпределение по държави (с английски
етикети):")
            print(df['country_label_en'].value_counts(dropna=False))
        else:
            print("\nНяма налични английски етикети на стойностите за
'country'.")

    # Подобно използване на етикетите може да се приложи при създаване
на графики
```

```
    # (като се използват final_variable_labels за надписи на осите) или при
    # представяне на таблици (като се използват english_value_labels за стойностите).

else:
    print("Не може да се продължи с анализа поради грешка при четене на файла.")
```

Разбор на решението:

1. **Прочитане и разглеждане на етикетите:** Функцията `read_spss_and_explore_labels` прочита файла и извежда примерни етикети, за да се добие представа за структурата им.
2. **Стратегия за извличане на английски етикети:** Функцията `extract_english_label` демонстрира прост подход за извличане на английски етикети, базиран на конвенция за добавяне на `(EN)` или `[EN]` в етикета. Този подход може да бъде адаптиран според реалната конвенция във файла.
3. **Ръчна обработка на смесени езици:** Функцията `display_and_select_label` показва наличните (оригинални) етикети за променлива и позволява на потребителя да избере един или да въведе ръчно. В реална ситуация, ако има слепени етикети на различни езици, тази функция може да бъде разширена, за да ги раздели и покаже поотделно.
4. **Използване на избраните етикети:** В примера се демонстрира как финалните (автоматично извлечени и ръчно избрани) етикети на променливите могат да се използват при извеждане на резултати от анализа (например, за описание на колоната 'country'). Също така се показва как могат да се използват английските етикети на стойностите за преобразуване на числови кодове в по-разбираеми текстови етикети при честотно разпределение.

Важни бележки:

- **Конвенция за етикети:** Успехът на автоматичното извличане зависи от наличието на последователна конвенция за обозначаване на езика в етикетите.
- **Сложни случаи:** Ако етикетите са много сложни или не следват ясна конвенция, може да се наложи по- sophisticated обработка на текста (например, използване на регулярни изрази) или дори ръчна интерпретация за голяма част от променливите.
- **Етикети на стойностите:** Аналогичен процес може да бъде приложен и за `meta.variable_value_labels`, като се създаде функция за извличане на етикети на стойностите на конкретен език за всяка променлива.
- **Съхранение на езикови карти:** Ако се налага многократна работа с файла, може да е полезно да се създаде речник или друг структуриран формат, който съпоставя имената на променливите и стойностите с техните английски (или друг желан) етикети, за да се използва многократно.

Това решение предоставя основни стратегии за справяне с многоезични етикети. В зависимост от структурата на вашите данни, може да се наложи да адаптирате тези подходи.

Казус 3: Запазване на резултати от анализ обратно в SPSS формат с коректни метаданни

Ситуация:

След като сте извършили анализ на данни, прочетени от SPSS файл, сте създали нови променливи (колони) във вашия Pandas DataFrame. Сега искате да запазите тези резултати обратно в SPSS `.sav` файл, като запазите възможно най-много от оригиналните метаданни (етикети на променливи и стойности) за съществуващите променливи и добавите подходящи метаданни за новите променливи.

Задача:

1. Прочетете оригинален SPSS файл (`original_data.sav`) и извършете някакъв анализ, който води до създаването на една или повече нови колони в DataFrame-а.
2. Създайте `pyreadstat.metadata_container` обект, който съдържа:
    - Имената на всички колони в модифицирания DataFrame.
    - Етикетите на оригиналните променливи (взети от `meta` на прочетения файл).
    - Етикети за новите променливи (създайте ги смислено).
    - Ако е приложимо, етикети на стойностите за новите променливи (ако те са категорийни).
3. Използвайте `pyreadstat.write_sav()` функцията, като подадете модифицирания DataFrame и създадения обект с метаданни, за да запишете резултата в нов `.sav` файл (`analyzed_data.sav`).
4. Проверете създадения `.sav` файл с друг софтуер (например, PSPP или SPSS) дали метаданните са запазени коректно.

Решение:

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

def read_spss_with_metadata(file_path):
    """Прочита SPSS файл и връща DataFrame и метаданни."""
    try:
        df, meta = pyreadstat.read_sav(file_path)
        print(f"Успешно прочетен: {file_path}")
        return df, meta
    except FileNotFoundError:
        print(f"Файлът '{file_path}' не беше намерен.")
        return None, None
    except Exception as e:
        print(f"Грешка при четене на файла '{file_path}': {e}")
        return None, None

def analyze_data(df):
    """Извършва примерен анализ и добавя нова колона."""
    df['Възраст_Квадрат'] = df['Възраст'] ** 2
    df['Висок_Доход'] = df['Доход'] > 50000
    return df

def create_metadata_for_new_columns(original_meta, new_columns):
    """Създава метаданни за новите колони."""
```

```
    new_variable_labels = {}
    new_variable_value_labels = {}
    for col in new_columns:
        if col == 'Възраст_Квадрат':
            new_variable_labels[col] = 'Възраст на респондента
(квадрат)'
        elif col == 'Висок_Доход':
            new_variable_labels[col] = 'Респондент с висок доход
(>50000)'
            new_variable_value_labels[col] = {True: 'Да', False: 'Не'}
    return new_variable_labels, new_variable_value_labels

def save_to_spss_with_metadata(df, original_meta, new_var_labels,
new_val_labels, output_file):
    """Записва DataFrame в SPSS формат с оригинални и нови метаданни."""
    column_labels = original_meta.column_labels.copy()
    variable_value_labels = original_meta.variable_value_labels.copy()

    # Добавяне на етикети за новите колони
    for col, label in new_var_labels.items():
        if col in df.columns and col not in original_meta.column_names:
            column_labels.append(label)

    # Добавяне на етикети на стойностите за новите колони
    variable_value_labels.update(new_val_labels)

    try:
        pyreadstat.write_sav(
            df,
            output_file,
            column_labels=column_labels,
            variable_value_labels=variable_value_labels,
            variable_formats=original_meta.variable_formats # Запазваме
оригиналните формати
            # user_missing=original_meta.missing_user_value # Възможно е
да искате да запазите и user_missing
        )
        print(f"Резултатите са записани успешно в '{output_file}' с
метаданни.")
    except Exception as e:
        print(f"Грешка при записване в SPSS формат: {e}")

# 1. Прочитане на оригиналния SPSS файл
original_file = 'original_data.sav' # Заменете с пътя до вашия файл
df_original, meta_original = read_spss_with_metadata(original_file)

if df_original is not None:
    # Уверете се, че колоните 'Възраст' и 'Доход' съществуват
    if 'Възраст' not in df_original.columns or 'Доход' not in
df_original.columns:
        print("Файлът не съдържа необходимите колони 'Възраст' и 'Доход'
за примера.")
        exit()
```

```
    # 2. Извършване на анализ и добавяне на нови колони
    df_analyzed = analyze_data(df_original.copy())
    new_columns = ['Възраст_Квадрат', 'Висок_Доход']

    # 3. Създаване на метаданни за новите колони
    new_var_labels, new_val_labels =
create_metadata_for_new_columns(meta_original, new_columns)

    # 4. Запазване на резултатите обратно в SPSS формат с метаданни
    output_file = 'analyzed_data.sav'
    save_to_spss_with_metadata(
        df_analyzed,
        meta_original,
        new_var_labels,
        new_val_labels,
        output_file
    )

    print("\nПроверете файла 'analyzed_data.sav' с PSPP или SPSS.")

else:
    print("Не може да се продължи поради грешка при четене на
оригиналния файл.")
```

Разбор на решението:

1. **Прочитане на оригиналния файл:** Функцията `read_spss_with_metadata` прочита `.sav` файла и връща DataFrame и неговите метаданни.
2. **Извършване на анализ:** Функцията `analyze_data` симулира анализ, като създава две нови колони: 'Възраст_Квадрат' и 'Висок_Доход'.
3. **Създаване на метаданни за нови колони:**
    - `create_metadata_for_new_columns` приема оригиналните метаданни и списък с нови колони.
    - Създава речници `new_variable_labels` за етикетите на новите променливи и `new_variable_value_labels` за етикетите на стойностите на новите (категорийни) променливи.
4. **Записване в SPSS формат с метаданни:**
    - `save_to_spss_with_metadata` приема DataFrame-а с резултати, оригиналните метаданни и метаданните за новите колони.
    - Копират се етикетите на колоните и етикетите на стойностите от оригиналните метаданни.
    - Добавят се етикетите за новите колони към списъка `column_labels`.
    - Речникът с етикетите на стойностите за новите колони се обединява със съществуващия `variable_value_labels`.
    - `pyreadstat.write_sav()` се използва за запис на DataFrame-а в новия `.sav` файл, като се подават `column_labels` и `variable_value_labels`. Също така се запазват оригиналните формати на променливите (`variable_formats`). Можете да решите дали да запазите и `user_missing`.

Важни бележки:

- **Съществуващи метаданни:** Решението се опитва да запази етикетите на оригиналните променливи. Ако имената на оригиналните колони са променени по време на анализа, ще трябва да актуализирате `column_labels` съответно.
- **Формати на променливите:** Аргументът `variable_formats` се използва, за да се запазят оригиналните формати на променливите (например, брой десетични знаци за числови променливи).
- **Липсващи стойности за нови променливи:** Ако новите променливи имат специфични кодове за липсващи стойности, ще трябва да ги дефинирате и да ги добавите към аргумента `user_missing` на `write_sav()`.
- **Типове на променливите:** `pyreadstat` се опитва да определи типовете на променливите въз основа на данните в DataFrame-а. В някои случаи може да се наложи да предоставите информация за типа на новите променливи чрез аргумента `variable_types` на `write_sav()`.
- **Проверка:** След запис е важно да проверите създадения `.sav` файл с друг софтуер, за да се уверите, че данните и метаданните са запазени както очаквате.

Тези казуси илюстрират някои от реалните предизвикателства при работа със SPSS файлове, като различия в кодирането и представянето на данни между различни файлове, многоезични етикети и необходимостта от запазване на метаданни. Решаването на тези казуси ще задълбочи разбирането ви за използването на `pyreadstat` и Pandas за ефективна обработка на SPSS данни.

Въпроси:

1. Защо е необходима външна библиотека като `pyreadstat` за четене на SPSS файлове (`.sav`) в Pandas? Какви възможности предоставя тя в сравнение с вградените функции на Pandas за други файлови формати?
2. Обяснете каква информация се съдържа в обекта с метаданни, който се връща заедно с DataFrame-а от функцията `pyreadstat.read_sav()`. Кои са основните атрибути на този обект и за какво служат?
3. Как `pyreadstat` се справя с дефинираните в SPSS липсващи стойности? Трябва ли да се предприемат допълнителни стъпки за тяхната обработка в Pandas? Ако да, какви?
4. Как можете да използвате етикетите на стойностите, прочетени от SPSS файла с `pyreadstat`, за да направите анализа на категорийни променливи по-информативен в Pandas? Дайте пример.
5. В какви ситуации може да се наложи да укажете кодирането изрично при четене на SPSS файл с `pyreadstat`? Какви са някои често срещани кодирания, които може да опитате?
6. Възможно ли е да се запише Pandas DataFrame обратно в SPSS `.sav` формат с помощта на `pyreadstat`? Ако да, какви аспекти от метаданните могат да бъдат запазени или зададени при запис?

Задачи:

1. **(Четене и метаданни):**
   - Прочетете SPSS файл (`survey.sav`).
   - Изведете списък с всички етикети на променливите.
   - За конкретна променлива (например, 'образование'), изведете нейните етикети на стойностите.
   - Проверете дали има дефинирани липсващи стойности за някоя от променливите и ги изведете.
2. **(Обработка на липсващи стойности):**
   - Прочетете SPSS файл (`survey_with_missing.sav`).
   - Преобразувайте всички дефинирани липсващи стойности в `NaN` стойности на Pandas DataFrame-а.
   - Пребройте броя на липсващите стойности за всяка променлива след преобразуването.
3. **(Използване на етикети на стойностите):**
   - Прочетете SPSS файл, съдържащ променлива за пол (кодирана като 1 и 2) и променлива за удовлетвореност (кодирана по скала от 1 до 5).
   - Използвайте етикетите на стойностите, за да създадете честотни таблици за тези две променливи, като показвате текстовите етикети вместо числовите кодове.
4. **(Работа с кодиране):**
   - Опитайте да прочетете SPSS файл, за който знаете, че използва специфично кодиране (например, кирилица в 'windows-1251'). Прочетете го първо без да указвате кодиране и наблюдавайте резултата. След това го прочетете отново, указвайки правилното кодиране.
5. **(Записване в SPSS формат - по-сложно)*:**
   - Прочетете SPSS файл и създайте нова колона във върнатия DataFrame (например, на базата на съществуващи данни).

- Създайте примерен обект с метаданни (`pyreadstat.metadata_container`), който включва етикет за новата променлива и (ако е приложимо) етикети на стойностите.
- Запишете модифицирания DataFrame в нов `.sav` файл, използвайки създадените метаданни. Проверете файла с друг софтуер (например, PSPP) дали етикетите са запазени.

XVI. SQL бази данни:

Pandas предоставя удобни функции за взаимодействие с SQL бази данни, позволявайки изпълнение на SQL заявки и трансфер на данни между DataFrame-и и SQL таблици. За тази цел Pandas използва SQLAlchemy, която осигурява гъвкав начин за свързване с различни SQL диалекти (например, SQLite, PostgreSQL, MySQL, MS SQL Server).

Преди да започнете, уверете се, че имате инсталирани необходимите библиотеки за връзка с вашата SQL база данни (например, `sqlite3` за SQLite, `psycopg2` за PostgreSQL, `mysql-connector-python` за MySQL, `pyodbc` за MS SQL Server). SQLAlchemy обикновено се инсталира автоматично с Pandas, но ако имате проблеми, можете да го инсталирате с:

```
pip install sqlalchemy
```

1. Четене от SQL

Pandas предлага три основни функции за четене на данни от SQL бази данни:

- **`read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None, dtype=None):`** Изпълнява произволна SQL заявка и връща резултата като DataFrame.
- **`read_sql_table(table_name, con, index_col=None, coerce_float=True, parse_dates=None, columns=None, schema=None, chunksize=None, dtype=None):`** Чете цяла SQL таблица в DataFrame.
- **`read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None, dtype=None):`** Тази функция е по-гъвкава и може да приема или SQL заявка (аргумент `sql` е низ), или име на таблица (аргумент `sql` е низ и се използва `table_name` вътрешно).

Всички тези функции изискват връзка с базата данни (`con`). Тази връзка обикновено се създава с помощта на SQLAlchemy `create_engine()` или чрез специфични за драйвера методи за свързване.

а) Пример за свързване с SQLite (в паметта):

```
from sqlalchemy import create_engine
import pandas as pd

# Създаване на връзка с SQLite база данни в паметта
engine = create_engine('sqlite:///:memory:')
```

```
# Създаване на примерна таблица
data = {'Име': ['Алиса', 'Боб', 'Чарли'],
        'Възраст': [25, 30, 28],
        'Град': ['София', 'Пловдив', 'Варна']}
df_example = pd.DataFrame(data)
df_example.to_sql('потребители', engine, if_exists='replace',
index=False)

# Четене с read_sql_query
query = "SELECT Име, Възраст FROM потребители WHERE Възраст > 25"
df_query = pd.read_sql_query(query, engine)
print("Резултат от SQL заявка:")
print(df_query)

# Четене с read_sql_table
df_table = pd.read_sql_table('потребители', engine)
print("\nПрочетена цялата таблица:")
print(df_table)

# Четене с read_sql (използване на заявка)
df_sql_query = pd.read_sql("SELECT Град FROM потребители", engine)
print("\nРезултат от SQL заявка с read_sql:")
print(df_sql_query)

# Четене с read_sql (използване на име на таблица)
df_sql_table = pd.read_sql('потребители', engine, index_col='Име')
print("\nПрочетена цялата таблица с read_sql (зададен индекс):")
print(df_sql_table)

# Затваряне на връзката (важно за външни бази данни)
engine.dispose()
```

б) Важни параметри при четене:

- **`sql`:** SQL заявка (за `read_sql_query` и `read_sql`) или име на таблица (за `read_sql_table` и `read_sql`).
- **`con`:** Обект за връзка с базата данни (резултат от `create_engine()` или друга валидна връзка).
- **`index_col`:** Колона(и) за използване като индекс на DataFrame-а.
- **`parse_dates`:** Списък от колони за парсване като дати.
- **`columns`:** Списък от колони за четене (ако не искате всички).
- **`chunksize`:** Ако е зададен, връща итератор, където всеки елемент е DataFrame с определен брой редове (полезно за големи таблици).
- **`dtype`:** Речник, указващ типа на данните за колоните.

2. Запис в SQL

Pandas предоставя функцията **`to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None)`** за запис на DataFrame в SQL база данни като таблица.

а) Пример за запис:

```
data_to_db = {'Продукт': ['Ябълка', 'Банан', 'Портокал'],
              'Цена': [1.20, 0.80, 1.00]}
df_to_db = pd.DataFrame(data_to_db)

# Запис на DataFrame в SQL таблица 'продукти'
df_to_db.to_sql('продукти', engine, if_exists='replace', index=False)

# Проверка дали данните са записани успешно
df_from_db = pd.read_sql_table('продукти', engine)
print("\nДанни, прочетени от таблица 'продукти':")
print(df_from_db)

# Добавяне на нови данни към съществуващата таблица
new_data = {'Продукт': ['Круша'], 'Цена': [1.50]}
df_new = pd.DataFrame(new_data)
df_new.to_sql('продукти', engine, if_exists='append', index=False)

df_updated = pd.read_sql_table('продукти', engine)
print("\nДанни след добавяне:")
print(df_updated)

# Затваряне на връзката
engine.dispose()
```

б) Важни параметри при запис:

- **`name`:** Името на таблицата, която ще бъде създадена (или в която ще се добавят данни).
- **`con`:** Обект за връзка с базата данни.
- **`if_exists`:** Какво да се прави, ако таблицата вече съществува:
  - `'fail'`: Вдига грешка.
  - `'replace'`: Изтрива таблицата и създава нова.
  - `'append'`: Добавя данните към съществуващата таблица.
- **`index`:** Дали да се записва индексът на DataFrame-а като колона в SQL таблицата.
- **`index_label`:** Име на колоната за индекса (ако `index=True`).
- **`chunksize`:** Брой редове за запис наведнъж (полезно за големи DataFrame-и).
- **`dtype`:** Речник, указващ SQL типа на данните за колоните.
- **`method`:** Как да се вмъкват редовете в SQL. `'multi'` (по подразбиране за повечето бази данни) използва bulk insert (по-бързо). Може да се зададе като функция за по-специфично поведение.

3. Задълбочено разглеждане на работата с SQL бази данни в Pandas

Взаимодействието между Pandas и SQL бази данни е ключов аспект при анализ на данни, особено когато данните се управляват в релационни системи. Pandas улеснява този процес чрез абстрахиране на сложността на SQL заявките и връзките с бази данни, като същевременно предоставя гъвкавост за изпълнение на специфични SQL операции.

а) Създаване на връзка с база данни

Преди да четем или пишем данни, трябва да установим връзка с базата данни. Както споменахме, SQLAlchemy е основният инструмент, който Pandas използва за тази цел. `create_engine()` е функция, която създава обект за връзка (engine). Синтаксисът на URL адреса за връзка зависи от конкретния SQL диалект и драйвер.

След като създадете `engine`, този обект се подава като аргумент `con` на функциите `read_sql` и `to_sql`.

Примери за URL адреси на връзка:

- **SQLite (файл):** `'sqlite:///path/to/your/database.db'`

```
'sqlite:///my_database.db'
```

В този пример, `'my_database.db'` е името на файла, където ще се съхранява SQLite базата данни. Ако файлът не съществува, той ще бъде създаден. Пътят може да бъде абсолютен (например, `'sqlite:///C:/Data/my_database.db'`) или относителен спрямо текущата работна директория.

- **SQLite (в паметта):** `'sqlite:///:memory:'`

```
'sqlite:///:memory:'
```

Този connection string създава временна SQLite база данни, която съществува само докато е активна връзката с нея. Данните се губят след затваряне на връзката.

- **PostgreSQL:** `'postgresql://user:password@host:port/database'`

```
'postgresql://myuser:mypassword@localhost:5432/mydatabase'
```

  - `myuser`: Вашето потребителско име за PostgreSQL.
  - `mypassword`: Паролата за този потребител.
  - `localhost`: Хостът, на който се намира PostgreSQL сървърът (може да е IP адрес или име на хост).
  - `5432`: Портът, на който PostgreSQL слуша за връзки (по подразбиране е 5432).
  - `mydatabase`: Името на базата данни, към която искате да се свържете.

- **MySQL:** `'mysql+mysqlconnector://user:password@host:port/database'`

```
'mysql+mysqlconnector://myuser:mypassword@localhost:3306/mydatabase'
```

  - `myuser`: Вашето потребителско име за MySQL.
  - `mypassword`: Паролата за този потребител.
  - `localhost`: Хостът на MySQL сървъра.
  - `3306`: Портът на MySQL сървъра (по подразбиране е 3306).

- **MS SQL Server (SQLAlchemy):** `'mssql+pyodbc://user:password@dsn_name'` (където `dsn_name` е конфигуриран ODBC източник на данни)
- `'mssql+pyodbc://mydsn'`
  - `mydsn`: Името на предварително конфигуриран ODBC Data Source Name (DSN) на вашата система, който съдържа информацията за връзка със SQL Server (сървър, база данни, удостоверяване и др.).

- **MS SQL Server (ADO.NET - може да изисква `pyodbc`):**
  `'mssql+pyodbc:///?odbc_connect=Driver={ODBC Driver 17 for SQL Server};Server=your_server_name;Database=your_database_name;UID=your_username;PWD=your_password;'`
  - `Driver={ODBC Driver 17 for SQL Server}`: Указва ODBC драйвера, който да се използва. Уверете се, че сте инсталирали правилния драйвер (може да е друга версия, например `ODBC Driver 13 for SQL Server`).
  - `Server=your_server_name`: Името на SQL Server инстанцията (може да е IP адрес или име на хост).
  - `Database=your_database_name`: Името на базата данни.
  - `UID=your_username`: Вашето потребителско име за SQL Server.
  - `PWD=your_password`: Паролата за този потребител.

Важно:

- Уверете се, че сте инсталирали необходимите драйвери за съответната база данни (например, `psycopg2` за PostgreSQL, `mysql-connector-python` за MySQL, `pyodbc` за MS SQL Server).
- Заменете placeholder стойностите с вашите реални данни за връзка.
- Съхранявайте чувствителна информация като пароли сигурно, като избягвате да ги пишете директно в кода. Използвайте променливи на средата или други методи за управление на секрети.

Тези примерни connection strings ще ви помогнат да започнете да се свързвате с различни SQL бази данни от Pandas. Не забравяйте да адаптирате детайлите според вашата конкретна конфигурация.

б) По-детайлно за четене на данни

1) `read_sql_query()`

Тази функция е най-гъвкава, тъй като позволява изпълнението на всяка валидна SQL `SELECT` заявка. Можете да използвате сложни `JOIN` клаузи, `WHERE` филтри, `GROUP BY` агрегации, `ORDER BY` сортиране и други SQL конструкции.

Пример със сложна заявка:

```
query_complex = """
SELECT p.Име AS Име_Потребител, o.Продукт, o.Количество, o.Дата
```

```
FROM потребители p
JOIN поръчки o ON p.ID = o.ПотребителID
WHERE p.Град = 'София' AND o.Количество > 1
ORDER BY o.Дата DESC;
"""
df_complex = pd.read_sql_query(query_complex, engine)
print("\nРезултат от сложна SQL заявка:")
print(df_complex)
```

2) read_sql_table()

Тази функция е по-ограничена, тъй като чете цяла таблица. Въпреки това, параметрите `columns` и `index_col` позволяват да селектират определени колони и да се зададе индекс. Параметърът `schema` е важен, ако таблицата се намира в определена схема, различна от подразбиращата се.

Пример с указване на схема и колони:

```
# Ако таблицата 'потребители' е в схема 'public' (за PostgreSQL)
df_table_schema = pd.read_sql_table('потребители', engine,
schema='public', columns=['Име', 'Възраст'])
print("\nПрочетена таблица с указване на схема и колони:")
print(df_table_schema)
```

3) read_sql()

Както беше споменато, тази функция може да приема или SQL заявка (поведението е идентично с `read_sql_query()`), или име на таблица (поведението е сходно с `read_sql_table()`). Pandas се опитва да определи дали аргументът `sql` е заявка или име на таблица въз основа на съдържанието му (например, ако съдържа `SELECT`, `FROM` и т.н.). Въпреки това, може да е по-явно да използвате `read_sql_query()` или `read_sql_table()` в зависимост от намерението.

в) По-детайлно за запис на данни (`to_sql()`)

Функцията `to_sql()` предлага различни начини за управление на съществуващи таблици и контрол върху типа на данните, които се записват.

1) `if_exists` параметър

Този параметър е критичен за предотвратяване на загуба на данни или грешки:

- `'fail'`: Използвайте, когато не искате да презаписвате съществуваща таблица и искате да сте сигурни, че записвате в нова таблица.
- `'replace'`: Използвайте внимателно, тъй като това ще изтрие цялата съществуваща таблица и ще я замени с данните от DataFrame-а.
- `'append'`: Използвайте, когато искате да добавите нови редове към съществуваща таблица. Уверете се, че структурата на DataFrame-а съответства на структурата на таблицата.

2) `index` и `index_label` параметри

Ако `index=True`, индексът на DataFrame-а ще бъде записан като колона в SQL таблицата. `index_label` позволява да зададете име на тази колона (по подразбиране е `None`, което може да доведе до име `'index'`). Ако индексът е съставен (MultiIndex), ще бъдат създадени множество индексни колони.

3) `dtype` параметър

Понякога Pandas може да не определи правилно SQL типа на данните за колоните. Параметърът `dtype` приема речник, където ключовете са имената на колоните, а стойностите са SQL типове (например, `VARCHAR(50)`, `INTEGER`, `FLOAT`). Това е полезно за осигуряване на съвместимост с дефиницията на таблицата в базата данни или за оптимизация на съхранението. SQLAlchemy диалектите предоставят свои собствени типове данни, които могат да бъдат използвани тук (например, `sqlalchemy.types.VARCHAR`).

Пример за указване на SQL типове данни:

```
from sqlalchemy import types

data_types = {'Име': types.VARCHAR(50),
              'Възраст': types.INTEGER(),
              'Град': types.VARCHAR(50)}
df_to_db.to_sql('потребители_типове', engine, if_exists='replace',
index=False, dtype=data_types)

# Проверка на създадената таблица (зависи от използваната SQL система)
# Например, в SQLite: PRAGMA table_info(потребители_типове);
```

4) `chunksize` параметър

За големи DataFrame-и, записването на всички редове наведнъж може да бъде неефективно или да доведе до проблеми с паметта. Параметърът `chunksize` позволява записването на данните на порции (партиди) с определен брой редове. Това може да подобри производителността и да намали използването на памет.

5) `method` параметър

Този параметър контролира как Pandas вмъква редовете в SQL таблицата.

- `None` (или `'multi'` за повечето диалекти): Използва bulk insert, което е по-бързо за вмъкване на много редове.
- Функция: Можете да предоставите функция, която приема обект за връзка с базата данни, име на таблица и DataFrame chunk, и извършва вмъкването по персонализиран начин. Това може да е полезно за специфични нужди за обработка на данни по време на вмъкването.

г) Работа с различни SQL диалекти

Тъй като Pandas използва SQLAlchemy, той поддържа голям брой SQL диалекти. Въпреки това, е важно да имате инсталирани правилните драйвери за вашата база данни (например, `psycopg2` за PostgreSQL, `mysql-connector-python` за MySQL). URL адресът за връзка (`create_engine()`) трябва да бъде съобразен с конкретния диалект и драйвер.

д) Сигурност

Когато работите с бази данни, е важно да имате предвид сигурността. Избягвайте да съхранявате пароли директно в кода. Вместо това, използвайте променливи на средата или други сигурни методи за управление на идентификационни данни.

4. Заключение

Работата с SQL бази данни в Pandas е мощен инструмент за анализ на данни, съхранявани в релационни системи. Разбирането на различните параметри на функциите `read_sql` и `to_sql`, както и основите на SQLAlchemy връзките, позволява ефективно прехвърляне и обработка на данни между DataFrame-и и SQL таблици. Задълбоченото познаване на тези аспекти е от съществено значение за всеки, който работи с данни, управлявани в SQL бази данни.

Казус 1: Анализ на продажби от SQLite база данни

Ситуация:

Имате SQLite база данни (`sales.db`), която съдържа две таблици: `customers` (информация за клиентите) и `orders` (информация за поръчките).

Структура на таблиците:

customers:

Column	Type
id	INTEGER
customer_id	INTEGER
order_date	TEXT
total_amount	REAL

orders:

Column	Type
id	INTEGER
customer_id	INTEGER
order_date	TEXT
total_amount	REAL

Задача:

1. Създайте връзка с SQLite базата данни `sales.db`.
2. Прочетете цялата таблица `customers` в Pandas DataFrame.
3. Използвайте SQL заявка, за да прочетете всички поръчки, направени от клиенти от град 'София', заедно с имената на клиентите.
4. Изчислете средната стойност на поръчките за всеки клиент от 'София'.
5. Запишете DataFrame, съдържащ имената на клиентите от 'София' и тяхната средна стойност на поръчките, в нова SQL таблица наречена `sofia_customer_avg_order`.

```
import pandas as pd
from sqlalchemy import create_engine
```

```
# 1. Създаване на връзка с SQLite
engine = create_engine('sqlite:///sales.db')

# Създаване на примерни данни и запис в базата данни (ако я няма)
data_customers = {'id': [1, 2, 3, 4],
                  'name': ['Алиса', 'Боб', 'Чарли', 'Деси'],
                  'city': ['София', 'Пловдив', 'София', 'Варна']}
df_customers = pd.DataFrame(data_customers)
df_customers.to_sql('customers', engine, if_exists='replace',
index=False)

data_orders = {'id': [1, 2, 3, 4, 5, 6],
               'customer_id': [1, 2, 1, 3, 1, 2],
               'order_date': ['2025-05-01', '2025-05-01', '2025-05-02',
'2025-05-03', '2025-05-04', '2025-05-05'],
               'total_amount': [100.50, 55.20, 200.00, 78.90, 120.00,
62.10]}
df_orders = pd.DataFrame(data_orders)
df_orders.to_sql('orders', engine, if_exists='replace', index=False)

# 2. Четене на таблицата customers
df_customers_from_db = pd.read_sql_table('customers', engine)
print("Прочетена таблица 'customers':")
print(df_customers_from_db)

# 3. Четене на поръчки от клиенти от София с имената на клиентите
query = """
SELECT c.name, o.order_date, o.total_amount
FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE c.city = 'София';
"""
df_sofia_orders = pd.read_sql_query(query, engine)
print("\nПоръчки от клиенти от София:")
print(df_sofia_orders)

# 4. Изчисляване на средната стойност на поръчките за всеки клиент от
София
avg_orders_sofia =
df_sofia_orders.groupby('name')['total_amount'].mean().reset_index()
avg_orders_sofia.rename(columns={'name': 'customer_name',
'total_amount': 'average_order_amount'}, inplace=True)
print("\nСредна стойност на поръчките за клиенти от София:")
print(avg_orders_sofia)

# 5. Запис на резултата в нова SQL таблица
avg_orders_sofia.to_sql('sofia_customer_avg_order', engine,
if_exists='replace', index=False)

# Проверка дали таблицата е създадена и съдържа данните
df_avg_orders_from_db = pd.read_sql_table('sofia_customer_avg_order',
engine)
print("\nПрочетена таблица 'sofia_customer_avg_order':")
print(df_avg_orders_from_db)
```

```
# Затваряне на връзката
engine.dispose()
```

Казус 2: Записване на резултати от анализ в PostgreSQL

Ситуация:

Извършили сте сложен анализ в Pandas DataFrame (`analysis_results`) и искате да запазите резултатите в PostgreSQL база данни в таблица наречена `final_analysis`. DataFrame-ът има колони 'product_id', 'average_rating', 'review_count'.

Задача:

1. Създайте връзка с вашата PostgreSQL база данни (заменете placeholder стойностите).
2. Запишете DataFrame `analysis_results` в таблицата `final_analysis`. Ако таблицата вече съществува, добавете данните към нея. Уверете се, че индексът на DataFrame-а не се записва като колона.
3. Проверете дали данните са записани успешно, като прочетете цялата таблица `final_analysis` обратно в Pandas DataFrame.

```
import pandas as pd
from sqlalchemy import create_engine

# Създаване на примерен DataFrame с резултати от анализ
data_analysis = {'product_id': [101, 102, 103],
                 'average_rating': [4.5, 3.9, 4.8],
                 'review_count': [150, 80, 220]}
analysis_results = pd.DataFrame(data_analysis)
print("DataFrame за запис:")
print(analysis_results)

# 1. Създаване на връзка с PostgreSQL
engine_pg =
create_engine('postgresql://user:password@host:port/database') #
Заменете с вашите данни

try:
    # 2. Запис на DataFrame в PostgreSQL
    analysis_results.to_sql('final_analysis', engine_pg,
if_exists='append', index=False)
    print("\nДанните са успешно записани в таблица 'final_analysis' в
PostgreSQL.")

    # 3. Проверка чрез четене на таблицата
    df_from_pg = pd.read_sql_table('final_analysis', engine_pg)
    print("\nПрочетени данни от таблица 'final_analysis':")
```

```
    print(df_from_pg)

except Exception as e:
    print(f"Възникна грешка при взаимодействие с PostgreSQL: {e}")

finally:
    # Затваряне на връзката
    if 'engine_pg' in locals():
        engine_pg.dispose()
```

Казус 3: Четене на част от голяма SQL таблица на порции

Ситуация:

Имате голяма MySQL таблица (`large_dataset`) с милиони редове и искате да я обработвате на порции, за да избегнете проблеми с паметта. Таблицата има колони 'id', 'timestamp', 'value'.

Задача:

1. Създайте връзка с вашата MySQL база данни (заменете placeholder стойностите).
2. Прочетете таблицата `large_dataset` на порции по 1000 реда всяка.
3. За всяка порция, изчислете средната стойност на колоната 'value' и изведете резултата.

Решение:

```
import pandas as pd
from sqlalchemy import create_engine

# 1. Създаване на връзка с MySQL
engine_mysql =
create_engine('mysql+mysqlconnector://user:password@host:port/database')
# Заменете с вашите данни

try:
    # 2. Четене на таблицата на порции
    chunk_iterator = pd.read_sql_table('large_dataset', engine_mysql,
chunksize=1000)

    # 3. Обработка на всяка порция
    all_averages = []
    for chunk in chunk_iterator:
        average_value = chunk['value'].mean()
        all_averages.append(average_value)
        print(f"Средна стойност за текущата порция: {average_value}")

    if all_averages:
        overall_average = sum(all_averages) / len(all_averages)
```

```
        print(f"\nОбща средна стойност за всички порции:
{overall_average}")
    else:
        print("\nТаблицата 'large_dataset' е празна.")

except Exception as e:
    print(f"Възникна грешка при взаимодействие с MySQL: {e}")

finally:
    # Затваряне на връзката
    if 'engine_mysql' in locals():
        engine_mysql.dispose()
```

Тези казуси илюстрират основните начини за взаимодействие между Pandas и SQL бази данни: четене на цели таблици, изпълнение на SQL заявки, записване на DataFrame-и и четене на големи набори от данни на порции. Разбирането на тези техники е важно за ефективна работа с данни, съхранявани в релационни системи. Не забравяйте да адаптирате connection strings и имената на таблици според вашата среда.

Казус 4*: Анализ на представянето на ученици от няколко училища

Разполагате с PostgreSQL база данни (`school_performance.db`), която съдържа информация за представянето на ученици от различни училища. Базата данни има следните таблици:

schools:

Column	Type
school_id	INTEGER PRIMARY KEY
name	TEXT
city	TEXT

students:

Column	Type
student_id	INTEGER PRIMARY KEY
school_id	INTEGER FOREIGN KEY REFERENCES schools(school_id)
name	TEXT
gender	TEXT
birth_date	TEXT

subjects:

Column	Type
subject_id	INTEGER PRIMARY KEY
name	TEXT

exams:

Column	Type
exam_id	INTEGER PRIMARY KEY
student_id	INTEGER FOREIGN KEY REFERENCES students(student_id)
subject_id	INTEGER FOREIGN KEY REFERENCES subjects(subject_id)
exam_date	TEXT
score	REAL

Задача:

1. Създайте връзка с PostgreSQL базата данни `school_performance.db`.
2. Прочетете всички таблици в Pandas DataFrame-и (`schools_df`, `students_df`, `subjects_df`, `exams_df`).
3. Използвайки SQL заявка, извлечете следната информация: име на училище, град, име на ученик, пол, име на предмет и резултат от изпит за всички записи.
4. Използвайте Pandas, за да:
   - Преобразувате колоната `birth_date` в datetime формат.
   - Изчислете възрастта на всеки ученик към последната дата на изпитване във всеки предмет (за всяка комбинация от ученик и предмет, вземете най-късната `exam_date`).
   - Намерете средния резултат на всеки ученик по всеки предмет.
   - Определете училището с най-висок среден успех (средно аритметично на средните резултати на всички ученици в училището по всички предмети).
   - Намерете топ 3 на най-трудните предмета (предметите с най-нисък среден успех за всички ученици).
   - За всеки град, намерете средната възраст на учениците, които имат среден успех над 85 по поне един предмет.
5. Създайте нов Pandas DataFrame, съдържащ училище (име), предмет (име) и среден успех за този предмет в това училище. Запишете този DataFrame в нова SQL таблица наречена `school_subject_avg_score`.

Решение:

```
import pandas as pd
from sqlalchemy import create_engine
from datetime import datetime

# 1. Създаване на връзка с PostgreSQL (заменете с вашите данни)
engine =
create_engine('postgresql://user:password@host:port/school_performance.d
b')

# Създаване на примерни данни и запис в базата данни (ако я няма)
data_schools = {'school_id': [1, 2, 3],
                'name': ['Първо СУ', 'Второ СУ', 'Трето СУ'],
                'city': ['София', 'Пловдив', 'Варна']}
schools_df = pd.DataFrame(data_schools)
schools_df.to_sql('schools', engine, if_exists='replace', index=False)

data_students = {'student_id': [1, 2, 3, 4, 5, 6],
                 'school_id': [1, 1, 2, 2, 3, 3],
                 'name': ['Иван Иванов', 'Петър Петров', 'Мария
Георгиева', 'Ангел Ангелов', 'Сияна Симеонова', 'Николай Николов'],
                 'gender': ['м', 'м', 'ж', 'м', 'ж', 'м'],
                 'birth_date': ['2008-05-10', '2007-11-20', '2009-02-
15', '2008-08-01', '2007-09-25', '2009-04-05']}
students_df = pd.DataFrame(data_students)
students_df.to_sql('students', engine, if_exists='replace', index=False)
```

```
data_subjects = {'subject_id': [1, 2, 3],
                 'name': ['Математика', 'Български', 'История']}
subjects_df = pd.DataFrame(data_subjects)
subjects_df.to_sql('subjects', engine, if_exists='replace', index=False)

data_exams = {'exam_id': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18],
              'student_id': [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 1, 2,
3, 4, 5, 6],
              'subject_id': [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 3, 3,
3, 3, 3, 3],
              'exam_date': ['2024-06-10', '2024-06-10', '2024-06-11',
'2024-06-11', '2024-06-12', '2024-06-12',
                            '2024-06-13', '2024-06-13', '2024-06-14',
'2024-06-14', '2024-06-15', '2024-06-15',
                            '2024-06-16', '2024-06-16', '2024-06-17',
'2024-06-17', '2024-06-18', '2024-06-18'],
              'score': [85, 92, 78, 88, 90, 82, 75, 80, 95, 88, 80, 76,
60, 70, 80, 90, 75, 65]}
exams_df = pd.DataFrame(data_exams)
exams_df.to_sql('exams', engine, if_exists='replace', index=False)

# 2. Четене на всички таблици
schools_df = pd.read_sql_table('schools', engine)
students_df = pd.read_sql_table('students', engine)
subjects_df = pd.read_sql_table('subjects', engine)
exams_df = pd.read_sql_table('exams', engine)

# 3. Извличане на информация с SQL заявка
query = """
SELECT s.name AS school_name, s.city AS school_city,
       st.name AS student_name, st.gender AS student_gender,
st.birth_date,
       sub.name AS subject_name, e.score, e.exam_date
FROM schools s
JOIN students st ON s.school_id = st.school_id
JOIN exams e ON st.student_id = e.student_id
JOIN subjects sub ON e.subject_id = sub.subject_id;
"""
performance_df = pd.read_sql_query(query, engine)

# 4. Използване на Pandas за анализ
# a) Преобразуване на birth_date в datetime
performance_df['birth_date'] =
pd.to_datetime(performance_df['birth_date'])

# b) Изчисляване на възрастта към последната дата на изпитване
performance_df['exam_date'] =
pd.to_datetime(performance_df['exam_date'])
latest_exam_dates = performance_df.groupby(['student_name',
'subject_name'])['exam_date'].max().reset_index()
performance_df_merged = pd.merge(performance_df, latest_exam_dates,
on=['student_name', 'subject_name', 'exam_date'], how='inner')
```

```
performance_df_merged['age_at_exam'] =
(performance_df_merged['exam_date'] -
performance_df_merged['birth_date']).dt.days / 365.25

# c) Среден резултат на всеки ученик по всеки предмет
average_scores = performance_df.groupby(['student_name',
'subject_name'])['score'].mean().reset_index()

# d) Училище с най-висок среден успех
school_avg_scores = pd.merge(average_scores, students_df,
on='student_name')
school_avg_scores = pd.merge(school_avg_scores, schools_df,
on='school_id')
school_overall_avg =
school_avg_scores.groupby('name_y')['score'].mean().sort_values(ascendin
g=False).index[0]
print(f"\nУчилище с най-висок среден успех: {school_overall_avg}")

# e) Топ 3 на най-трудните предмета
subject_avg_scores =
average_scores.groupby('subject_name')['score'].mean().sort_values(ascen
ding=True).head(3)
print("\nТоп 3 на най-трудните предмета:")
print(subject_avg_scores)

# f) Средна възраст на ученици с висок успех по град
high_achievers = average_scores[average_scores['score'] >
85]['student_name'].unique()
high_achievers_df =
students_df[students_df['name'].isin(high_achievers)]
high_achievers_df['birth_date'] =
pd.to_datetime(high_achievers_df['birth_date'])
current_date = datetime.now() # Използваме текущата дата за
приблизителна възраст
high_achievers_df['age'] = (current_date -
high_achievers_df['birth_date']).dt.days / 365.25
city_avg_age_high_achievers =
high_achievers_df.groupby('school_id')['age'].mean()
city_avg_age_high_achievers = pd.merge(city_avg_age_high_achievers,
schools_df, on='school_id')
print("\nСредна възраст на ученици с висок успех по град:")
print(city_avg_age_high_achievers[['name', 'age']])

# 5. Създаване и запис на school_subject_avg_score
school_subject_avg = pd.merge(average_scores, students_df,
on='student_name')
school_subject_avg = pd.merge(school_subject_avg, schools_df,
on='school_id')
school_subject_avg = school_subject_avg.groupby(['name_y',
'subject_name'])['score'].mean().reset_index()
school_subject_avg.rename(columns={'name_y': 'school_name', 'score':
'average_score'}, inplace=True)
```

```
school_subject_avg.to_sql('school_subject_avg_score', engine,
if_exists='replace', index=False)
print("\nТаблица 'school_subject_avg_score' е създадена.")

# Затваряне на връзката
engine.dispose()
```

Разбор на сложността:

- **Множество таблици и JOIN:** Казусът изисква обединяване на данни от четири различни таблици, за да се получи пълна информация за представянето на учениците.
- **Сложна SQL заявка:** Използва се `JOIN` клауза за комбиниране на данни от свързани таблици.
- **Обработка на дати:** Включва преобразуване на текстова колона в datetime формат и изчисляване на възрастта на учениците.
- **Групиране и агрегация в Pandas:** Използват се `groupby()` и `mean()` за намиране на средни резултати и средна възраст.
- **Филтриране и намиране на уникални стойности:** Използва се за идентифициране на ученици с висок успех.
- **Множествени сливания (merge):** Данните се сливат няколко пъти, за да се комбинира информация от различни DataFrame-и.
- **Запис на обобщени резултати обратно в SQL:** Създава се нов DataFrame с обобщена информация и се записва в нова SQL таблица.

Този казус демонстрира как Pandas може да се използва за ефективна обработка и анализ на данни, извлечени от SQL база данни, като комбинира мощта на SQL за извличане и обединяване на данни с гъвкавостта на Pandas за трансформация и анализ. Финалното записване на резултатите обратно в базата данни позволява по-нататъшно използване на анализираната информация.

Въпроси:

1. Обяснете ролята на SQLAlchemy при работата на Pandas с SQL бази данни. Защо е необходима тази библиотека?
2. Каква е разликата между функциите `read_sql_query()`, `read_sql_table()` и `read_sql()` на Pandas? В какви ситуации е по-подходящо да се използва всяка от тях?
3. Опишете различните опции за параметъра `if_exists` във функцията `to_sql()`. Кога бихте използвали всяка от тях?
4. Как можете да зададете кои колони от SQL таблица да бъдат прочетени в Pandas DataFrame?
5. Как може да се обработва голяма SQL таблица, която не може да се побере в паметта, с помощта на Pandas? Кой параметър на функциите за четене от SQL е ключов за това?
6. Обяснете как параметрите `index` и `index_label` във функцията `to_sql()` влияят на записа на DataFrame-а в SQL таблица.
7. Как можете да укажете специфични SQL типове данни за колоните, когато записвате DataFrame в SQL таблица с помощта на `to_sql()`? Защо може да е необходимо това?

Задачи:

1. **(Четене с конкретна заявка):**
   - Свържете се с SQLite база данни (`mydatabase.db`).
   - Използвайте `read_sql_query()`, за да извлечете всички записи от таблица `products`, където цената е по-висока от 50.
   - Изведете резултата като Pandas DataFrame.
2. **(Четене на определени колони):**
   - Свържете се с PostgreSQL база данни.
   - Използвайте `read_sql_table()`, за да прочетете само колоните 'name' и 'email' от таблица `users`.
   - Задайте колоната 'name' като индекс на DataFrame-а.
3. **(Записване с различни опции за `if_exists`):**
   - Създайте примерен Pandas DataFrame.
   - Свържете се с MySQL база данни.
   - Опитайте да запишете DataFrame-а в таблица `logs`, използвайки `if_exists='fail'`. Какво се случва, ако таблицата вече съществува?
   - Запишете същия DataFrame в таблица `logs`, използвайки `if_exists='replace'`. Проверете съдържанието на таблицата след записа.
   - Добавете нови данни към DataFrame-а и го запишете в таблица `logs`, използвайки `if_exists='append'`. Проверете отново съдържанието на таблицата.
4. **(Четене на порции):**
   - Свържете се с MS SQL Server база данни, съдържаща голяма таблица `sensor_data`.
   - Прочетете данните на порции по 5000 реда.
   - За всяка порция, изчислете средната стойност на колоната 'temperature' и я отпечатайте.
5. **(Записване с указване на dtype):**
   - Създайте Pandas DataFrame с колона 'id' (числа) и колона 'description' (текст).
   - Свържете се с SQLite база данни.
   - Запишете DataFrame-а в таблица `items`, като укажете, че колоната 'id' трябва да бъде от тип INTEGER в SQL, а 'description' от тип TEXT. Проверете типа на колоните в създадената таблица (ако е възможно с SQL команда).
6. **(Комбиниране на четене и анализ):**
   - Свържете се с PostgreSQL база данни с таблици `employees` (id, name, department_id) и `departments` (id, name).
   - Използвайте `read_sql_query()`, за да извлечете имената на всички служители и името на техния отдел.
   - Използвайте Pandas, за да намерите броя на служителите във всеки отдел.

Тези въпроси и задачи имат за цел да затвърдят разбирането ви за различните начини за взаимодействие между Pandas и SQL бази данни и да ви помогнат да приложите наученото в практически сценарии. Не забравяйте да адаптирате connection strings и имената на таблици според вашата локална среда и налични бази данни.

XVII. Google BigQuery: (read_gbq, to_gbq) - *Изисква pandas-gbq*

Google BigQuery е напълно управляван, мащабируем и рентабилен безсървърен складов продукт за данни. Библиотеката `pandas-gbq` осигурява интеграция между Pandas DataFrame-и и Google BigQuery, позволявайки лесно четене на данни от BigQuery в Pandas и записване на DataFrame-и в BigQuery таблици.

1. Инсталация на `pandas-gbq`

Преди да започнете, уверете се, че библиотеката `pandas-gbq` е инсталирана във вашата Python среда:

```
pip install pandas-gbq
```

Също така, трябва да имате настроена Google Cloud Platform (GCP) среда, проект с активиран BigQuery API и подходящи права за достъп. Библиотеката обикновено използва вашите GCP credentials, които могат да бъдат настроени чрез:

- **Google Cloud CLI (gcloud):** Ако работите в среда, където е инсталиран gcloud CLI, `pandas-gbq` автоматично ще използва credentials, с които сте се аутентикирали (`gcloud auth application-default login`).
- **Service Account Key File:** Можете да създадете service account key file в JSON формат от GCP конзолата и да го предоставите на `pandas-gbq`.
- **Environment Variables:** Можете да зададете пътя до service account key file чрез environment variable `GOOGLE_APPLICATION_CREDENTIALS`.

2. Четене от Google BigQuery (`read_gbq`)

Функцията `pandas.read_gbq(query, project_id=None, credentials=None, configuration=None, dialect='standard', location=None, progress_bar_type=None, job_config=None, use_bqstorage_api=None, bqstorage_client=None, max_results=None, api_method='auto', reauth=False, timeout=None, billing_project_id=None)` се използва за изпълнение на SQL заявки към Google BigQuery и връща резултата като Pandas DataFrame.

a) Основни параметри:

- **`query` (str):** SQL заявката, която ще бъде изпълнена в BigQuery. Използва стандартен SQL диалект по подразбиране.
- **`project_id` (str, optional):** ID на GCP проекта, в който се намира BigQuery dataset-ът. Може да бъде зададен тук или да се използва default-ният проект от credentials.
- **`credentials` (google.auth.credentials.Credentials, optional):** GCP credentials обект. Ако не е предоставен, `pandas-gbq` ще се опита да използва default credentials.
- **`dialect` (str, default='standard'):** SQL диалект, който да се използва ('standard' или 'legacy').
- **`location` (str, optional):** Локацията на BigQuery dataset-а (например, 'US', 'EU'). Препоръчително е да се зададе за по-добра производителност.

- **`progress_bar_type` (str, optional):** Тип на progress bar-а, който да се показва по време на изпълнение на заявката (например, 'tqdm').
- **`job_config` (google.cloud.bigquery.job.QueryJobConfig, optional):** Допълнителни конфигурационни опции за BigQuery job-а.

б) Пример за четене:

```
import pandas as pd

# SQL заявка за четене на данни
query = """
SELECT column1, column2
FROM `your-gcp-project.your_dataset.your_table`
LIMIT 10;
"""

# ID на вашия GCP проект
project_id = 'your-gcp-project'

try:
    # Четене на данни от BigQuery в DataFrame
    df_bq = pd.read_gbq(query, project_id=project_id, location='US')
    print("Данни, прочетени от BigQuery:")
    print(df_bq)

except Exception as e:
    print(f"Възникна грешка при четене от BigQuery: {e}")
```

Уверете се, че сте заменили `'your-gcp-project'`, `'your_dataset'` и `'your_table'` с вашите реални стойности. Също така, може да се наложи да промените `location` според местоположението на вашия BigQuery dataset.

3. Запис в Google BigQuery (`to_gbq`)

Функцията `pandas.DataFrame.to_gbq(destination_table, project_id=None, credentials=None, if_exists='fail', chunksize=None, reauth=False, table_config=None, api_method='auto', location=None, progress_bar=True, job_config=None, timeout=None, billing_project_id=None)` се използва за записване на Pandas DataFrame в Google BigQuery таблица.

а) Основни параметри:

- **`destination_table` (str):** Целевата таблица в BigQuery във формата `'your_dataset.your_table'`. Може да включва и project ID, например `'your-gcp-project.your_dataset.your_table'`. Ако project ID не е включен тук, трябва да бъде зададен в `project_id` параметъра.
- **`project_id` (str, optional):** ID на GCP проекта, в който ще бъде създадена таблицата.
- **`credentials` (google.auth.credentials.Credentials, optional):** GCP credentials обект.
- **`if_exists` (str, default='fail'):** Какво да се прави, ако целевата таблица вече съществува:
  - `'fail'`: Вдига грешка.
  - `'replace'`: Изтрива съществуващата таблица и създава нова.

    - `'append'`: Добавя данните към съществуващата таблица.
- **`chunksize` (int, optional):** Размер на партидите, на които да се разделят данните при качване (полезно за големи DataFrame-и).
- **`table_config` (dict, optional):** Конфигурация на BigQuery таблицата (например, schema, описание).
- **`location` (str, optional):** Локацията на BigQuery dataset-а. Препоръчително е да съвпада с локацията на dataset-а.
- **`progress_bar` (bool, default=True):** Дали да се показва progress bar по време на качването.
- **`job_config` (google.cloud.bigquery.job.LoadJobConfig, optional):** Допълнителни конфигурационни опции за BigQuery load job-а.

б) Пример за запис:

```
import pandas as pd

# Създаване на примерен DataFrame
data_to_bq = {'col1': [1, 2, 3], 'col2': ['a', 'b', 'c']}
df_to_bq = pd.DataFrame(data_to_bq)

# ID на вашия GCP проект
project_id = 'your-gcp-project'

# Целева таблица в BigQuery
destination_table = 'your_dataset.output_table'

try:
    # Запис на DataFrame в BigQuery
    df_to_bq.to_gbq(destination_table, project_id=project_id,
if_exists='replace', location='US')
    print(f"DataFrame-ът е успешно записан в BigQuery таблица
'{destination_table}'.")

except Exception as e:
    print(f"Възникна грешка при запис в BigQuery: {e}")
```

Отново, не забравяйте да замените placeholder стойностите с вашите реални данни.

4. Допълнителни аспекти

- **Credentials Management:** Управлението на GCP credentials е от съществено значение за сигурна и правилна работа с BigQuery. Препоръчва се използването на service accounts за автоматизирани процеси.
- **Billing:** Използването на BigQuery може да доведе до разходи, базирани на обема на обработените данни при заявки и съхранението на данни. Следете внимателно използването и настройките за биллинг на вашия GCP проект.
- **Performance:** За големи набори от данни, оптимизирането на SQL заявките и използването на подходящи локации може значително да подобри производителността.
- **Data Types:** `pandas-gbq` се опитва да конвертира Pandas data types към съответните BigQuery data types. В някои случаи може да се наложи явно указване на schema чрез `table_config`.

Работата с Google BigQuery в Pandas чрез `pandas-gbq` предоставя мощен начин за анализ на големи обеми от данни, съхранявани в облака, като комбинира гъвкавостта на Pandas с мащабируемостта на BigQuery.

5. Любопитни, полезни и интересни аспекти:

1. **SQL Диалект:** Въпреки че `pandas-gbq` по подразбиране използва Standard SQL, BigQuery поддържа и Legacy SQL. Можете да превключите диалекта чрез параметъра `dialect` във функцията `read_gbq()`. Standard SQL е по-съвременният и ANSI SQL 2011 съвместим диалект, който се препоръчва за нови проекти. Legacy SQL е по-стар и има някои разлики в синтаксиса и функционалността.
2. **Използване на BigQuery Storage API:** За по-бързо и по-ефективно четене на големи обеми от данни от BigQuery, можете да използвате BigQuery Storage API. За да го активирате, трябва да инсталирате библиотеката `google-cloud-bigquery-storage` и да зададете параметъра `use_bqstorage_api=True` във функцията `read_gbq()`. Това може значително да намали времето за четене и разходите за мрежов трафик.

```
import pandas as pd

query = "SELECT * FROM `your-project.your_dataset.your_table` LIMIT
100000"
project_id = "your-project"

try:
    df_bq_storage = pd.read_gbq(query, project_id=project_id,
use_bqstorage_api=True)
    print(f"Прочетени {len(df_bq_storage)} реда, използвайки BigQuery
Storage API.")
except Exception as e:
    print(f"Грешка при четене с Storage API: {e}")
```

1. **Конфигуриране на BigQuery Job:** Можете да предоставите обект `google.cloud.bigquery.job.QueryJobConfig` на параметъра `job_config` във функцията `read_gbq()`, за да настроите допълнителни опции за BigQuery заявката, като например приоритет, дестинационна таблица за резултатите (ако не искате да ги четете веднага в Pandas) и други. Подобно, за `to_gbq()`, можете да използвате `google.cloud.bigquery.job.LoadJobConfig` за контрол на процеса на записване.
2. **Интерактивна работа в Jupyter Notebook:** Когато работите в Jupyter Notebook, `pandas-gbq` може да показва progress bar за дълго изпълняващи се заявки и процеси на записване, което е много полезно за наблюдение на хода на изпълнение.
3. **Автоматично управление на Credentials:** Ако работите в Google Cloud среда (например, Compute Engine, Cloud Functions, Colab), `pandas-gbq` често може автоматично да открие и използва необходимите credentials, без да е необходимо изрично да ги предоставяте.

6. Изключения и потенциални проблеми:

1. **Проблеми с Credentials:** Най-честите проблеми при работа с `pandas-gbq` са свързани с неправилно настроени или липсващи GCP credentials. Уверете се, че имате валидни credentials

и че те имат необходимите права за достъп до BigQuery (например, `BigQuery Data Viewer`, `BigQuery User`, `BigQuery Data Editor`).

2. **Грешки в SQL заявките:** Ако предоставената SQL заявка има синтактични грешки или се опитва да достъпи несъществуващи таблици или колони, BigQuery ще върне грешка, която `pandas-gbq` ще прехвърли като Python изключение.
3. **Превишаване на квоти и лимити:** BigQuery има различни квоти и лимити (например, максимален размер на резултата от заявка, честота на заявките). Ако вашите операции ги надвишат, може да получите грешки. Проверете документацията на Google BigQuery за актуални квоти и лимити. Параметърът `max_results` в `read_gbq()` може да помогне за контролиране на размера на резултата.
4. **Проблеми с типове данни:** Въпреки че `pandas-gbq` се стреми да извършва разумни конверсии между Pandas и BigQuery типове данни, понякога може да възникнат несъответствия или загуба на прецизност. В такива случаи може да се наложи явно да дефинирате schema при записване (`table_config`) или да извършите допълнителна обработка на данните в Pandas след четене.
5. **Разходи:** Неправилно оптимизирани заявки могат да обработват големи обеми от данни и да доведат до по-високи разходи за BigQuery. Винаги преглеждайте плана за изпълнение на заявките (execution plan) в BigQuery конзолата, за да се уверите, че са ефективни.
6. **Време за изпълнение на заявки:** За много големи набори от данни или сложни заявки, времето за изпълнение може да бъде значително. Използването на BigQuery Storage API и оптимизирането на заявките са ключови за подобряване на производителността.
7. **Мрежови проблеми:** В редки случаи, проблеми с мрежовата връзка между вашата среда и Google Cloud могат да доведат до неуспешни операции.

Разбирането на тези аспекти и потенциални проблеми ще ви помогне да работите по-ефективно и да отстранявате грешки при използване на `pandas-gbq` за взаимодействие с Google BigQuery. Винаги е добра идея да се консултирате с официалната документация на `pandas-gbq` и Google BigQuery за най-актуална и детайлна информация.

Казус 1: Анализ на уеб трафик от BigQuery

Ситуация:

Имате Google BigQuery проект, който съдържа dataset с данни за уеб трафик от Google Analytics (или подобен източник). Една от таблиците в този dataset е `website_traffic`, която съдържа информация за посещенията на уебсайта, включително време на посещение, източник на трафик, разгледани страници и действия на потребителите.

Структура на (примерна) таблица `website_traffic`:

Column	Type
visit_timestamp	TIMESTAMP
source	STRING
page_viewed	STRING
user_id	STRING
action	STRING

Задача:

1. Настройте необходимите credentials за достъп до вашия Google BigQuery проект.
2. Използвайте `pandas-gbq` за четене на всички записи от таблицата `website_traffic` за последните 7 дни.
3. Използвайте Pandas, за да:
    - Преброите броя на посещенията за всеки източник на трафик.
    - Намерите топ 10 на най-разглежданите страници.
    - Определите броя на уникалните потребители за всеки ден от последните 7 дни.
4. Създайте Pandas DataFrame, съдържащ информация за дневния брой на уникалните потребители.
5. Запишете този DataFrame в нова BigQuery таблица във вашия проект, наречена `daily_unique_users`. Ако таблицата вече съществува, я презапишете.

Решение:

```
import pandas as pd
from google.cloud import bigquery
from datetime import datetime, timedelta

# 1. Настройка на credentials (ако не са настроени автоматично)
# Може да се наложи да зададете project_id, ако не е конфигуриран по
друг начин.
project_id = 'your-gcp-project-id'  # Заменете с вашия GCP project ID

try:
    # 2. Четене на данни от BigQuery за последните 7 дни
    seven_days_ago = datetime.now() - timedelta(days=7)
    timestamp_seven_days_ago = seven_days_ago.timestamp()

    query = f"""
    SELECT visit_timestamp, source, page_viewed, user_id
    FROM `{project_id}.your_dataset.website_traffic`
    WHERE visit_timestamp >=
TIMESTAMP_SECONDS({int(timestamp_seven_days_ago)});
    """

    df_traffic = pd.read_gbq(query, project_id=project_id,
location='US') # Заменете локацията, ако е необходимо
    print(f"Прочетени {len(df_traffic)} записа от BigQuery.")

    # 3. Анализ с Pandas
    # a) Брой посещения по източник
    source_counts = df_traffic['source'].value_counts().reset_index()
    source_counts.columns = ['source', 'visit_count']
    print("\nБрой посещения по източник:")
    print(source_counts)

    # b) Топ 10 най-разглеждани страници
    top_pages =
df_traffic['page_viewed'].value_counts().nlargest(10).reset_index()
    top_pages.columns = ['page', 'view_count']
    print("\nТоп 10 най-разглеждани страници:")
    print(top_pages)
```

```
    # c) Брой уникални потребители за всеки ден
    df_traffic['visit_date'] =
pd.to_datetime(df_traffic['visit_timestamp']).dt.date
    daily_unique_users =
df_traffic.groupby('visit_date')['user_id'].nunique().reset_index()
    daily_unique_users.columns = ['date', 'unique_users']
    print("\nБрой уникални потребители за всеки ден:")
    print(daily_unique_users)

    # 4. Създаване на DataFrame с дневния брой на уникалните потребители
(вече създаден: daily_unique_users)

    # 5. Запис на DataFrame в нова BigQuery таблица
    destination_table = 'your_dataset.daily_unique_users'
    daily_unique_users.to_gbq(destination_table, project_id=project_id,
if_exists='replace', location='US')
    print(f"\nDataFrame 'daily_unique_users' е записан в BigQuery
таблица '{destination_table}'.")

except Exception as e:
    print(f"Възникна грешка при работа с BigQuery: {e}")
```

Не забравяйте да замените `'your-gcp-project-id'` и `'your_dataset'` с вашите реални стойности. Също така, адаптирайте името на таблицата (`website_traffic`) и локацията (`'US'`) според вашата BigQuery структура и местоположение на данните.

Казус 2: Записване на резултати от машинен learning модел в BigQuery

Ситуация:

Обучили сте модел за машинно обучение, който предсказва вероятността даден потребител да закупи определен продукт. Резултатите от предсказанията са запазени в Pandas DataFrame (`predictions_df`) със следните колони: `user_id`, `product_id`, `predicted_probability`.

Задача:

1. Настройте необходимите credentials за достъп до вашия Google BigQuery проект.
2. Запишете DataFrame `predictions_df` в BigQuery таблица, наречена `product_purchase_predictions` във вашия dataset. Ако таблицата вече съществува, добавете новите предсказания към нея.
3. Създайте друга BigQuery таблица, наречена `high_probability_purchases`, която съдържа само предсказания с вероятност по-висока от 0.8. За целта, първо запишете всички предсказания и след това използвайте SQL заявка в BigQuery, за да създадете новата таблица.

Решение:

```
import pandas as pd
from google.cloud import bigquery

# 1. Настройка на credentials (ако не са настроени автоматично)
```

```
project_id = 'your-gcp-project-id'  # Заменете с вашия GCP project ID

# 2. Създаване на примерен DataFrame с предсказания
data_predictions = {'user_id': [1, 2, 3, 1, 4, 2],
                    'product_id': ['A', 'B', 'C', 'B', 'A', 'C'],
                    'predicted_probability': [0.92, 0.78, 0.85, 0.65,
0.98, 0.70]}
predictions_df = pd.DataFrame(data_predictions)
print("DataFrame с предсказания:")
print(predictions_df)

try:
    # Запис на всички предсказания в BigQuery
    destination_table_all = 'your_dataset.product_purchase_predictions'
    predictions_df.to_gbq(destination_table_all, project_id=project_id,
if_exists='append', location='US')
    print(f"\nDataFrame 'predictions_df' е записан в BigQuery таблица
'{destination_table_all}'.")

    # 3. Създаване на таблица с висока вероятност чрез SQL в BigQuery
    client = bigquery.Client(project=project_id)
    destination_table_high_prob =
f'{project_id}.your_dataset.high_probability_purchases'

    query_create_high_prob = f"""
    CREATE OR REPLACE TABLE `{destination_table_high_prob}` AS
    SELECT user_id, product_id, predicted_probability
    FROM `{project_id}.your_dataset.product_purchase_predictions`
    WHERE predicted_probability > 0.8;
    """

    query_job = client.query(query_create_high_prob)
    query_job.result()  # Изчаква завършване на заявката
    print(f"\nBigQuery таблица '{destination_table_high_prob}' е
създадена с предсказания с висока вероятност.")

except Exception as e:
    print(f"Възникна грешка при работа с BigQuery: {e}")
```

Отново, не забравяйте да замените `'your-gcp-project-id'` и `'your_dataset'` с вашите реални стойности. Тези казуси илюстрират основните операции за четене и записване на данни между Pandas и Google BigQuery, както и как можете да комбинирате Pandas за анализ с мощността на BigQuery за съхранение и обработка на големи обеми от данни.

Казус 3*: Анализ на поведението на потребители в e-commerce платформа

Ситуация:

Имате голям dataset в Google BigQuery, съдържащ информация за събития, генерирани от потребители на e-commerce платформа. Таблицата `user_activity` съдържа данни за сесии, действия (разглеждане на продукт, добавяне в количката, извършване на поръчка), времеви печати и информация за потребителите.

Структура на (примерна) таблица `user_activity`:

Column	Type	Description
session_id	STRING	**Уникален идентификатор на сесията на потребителя**
user_id	STRING	**Уникален идентификатор на потребителя**
event_timestamp	TIMESTAMP	**Времеви печат на събитието**
event_type	STRING	**Тип на събитието ('view_product', 'add_to_cart', 'purchase')**
product_id	STRING	**Идентификатор на продукта**
category	STRING	**Категория на продукта**
price	FLOAT	**Цена на продукта**
purchase_amount	FLOAT	**Сума на поръчката (само при 'purchase')**
traffic_source	STRING	**Източник на трафик (например, 'organic', 'paid', 'referral')**
device_type	STRING	**Тип на устройството на потребителя ('desktop', 'mobile')**
user_location	STRING	**Геолокация на потребителя**

Задача:

1. Настройте необходимите credentials за достъп до вашия Google BigQuery проект.
2. Използвайте `pandas-gbq` за четене на данни от таблицата `user_activity` за последните 30 дни. За големи обеми данни, прочетете на порции, ако е необходимо, и обединете ги в един DataFrame.
3. Използвайте Pandas, за да извършите следния анализ:
    - Определете средния брой продукти, добавени в количката на сесия за потребители, които впоследствие са извършили покупка.
    - Намерете кои са топ 5 на най-често купуваните продукти и техния среден брой в една поръчка.
    - Сегментирайте потребителите според броя на извършените покупки през периода (например, 1 покупка, 2-5 покупки, над 5 покупки).
    - За всеки сегмент потребители, определете средната стойност на поръчката и най-често използвания източник на трафик.
    - Анализирайте времето между добавяне в количката и извършване на покупка. Намерете средното време за различните категории продукти.
4. Използвайте BigQuery SQL заявка, за да намерите средната продължителност на сесия (в секунди) за всеки тип устройство (desktop/mobile). (Може да се наложи да дефинирате как се определя началото и края на сесията - например, първото и последното събитие в рамките на 30 минути).
5. Създайте Pandas DataFrame, съдържащ обобщена информация за потребителските сегменти (брой потребители в сегмент, средна стойност на поръчка, най-чест източник на трафик).
6. Запишете този обобщен DataFrame в нова BigQuery таблица, наречена `user_purchase_segments`.

Решение (концептуално и с основни примери на код):

```
import pandas as pd
from google.cloud import bigquery
from datetime import datetime, timedelta

# 1. Настройка на credentials
project_id = 'your-gcp-project-id'

try:
    # 2. Четене на данни от BigQuery за последните 30 дни (може да се
наложи chunking)
    thirty_days_ago = datetime.now() - timedelta(days=30)
    timestamp_thirty_days_ago = thirty_days_ago.timestamp()

    query = f"""
    SELECT *
    FROM `{project_id}.your_dataset.user_activity`
    WHERE event_timestamp >=
TIMESTAMP_SECONDS({int(timestamp_thirty_days_ago)});
    """

    # Ако данните са много големи, може да се чете на порции
    # all_data = []
    # for chunk in pd.read_gbq(query, project_id=project_id,
chunksize=100000):
    #     all_data.append(chunk)
    # df_activity = pd.concat(all_data)
    df_activity = pd.read_gbq(query, project_id=project_id,
location='US')
    print(f"Прочетени {len(df_activity)} записа.")

    # 3. Анализ с Pandas
    # a) Среден брой добавени в количката преди покупка
    purchasers = df_activity[df_activity['event_type'] ==
'purchase']['session_id'].unique()
    cart_additions_before_purchase =
df_activity[(df_activity['session_id'].isin(purchasers)) &
(df_activity['event_type'] == 'add_to_cart')]
    avg_cart_additions =
cart_additions_before_purchase.groupby('session_id').size().mean()
    print(f"\nСреден брой продукти, добавени в количката преди покупка:
{avg_cart_additions:.2f}")

    # b) Топ 5 най-често купувани продукти и среден брой
    purchases = df_activity[df_activity['event_type'] == 'purchase']
    top_products =
purchases['product_id'].value_counts().nlargest(5).index.tolist()
    avg_quantity_top_products =
purchases[purchases['product_id'].isin(top_products)].groupby('product_i
d')['session_id'].size().groupby('product_id').mean()
    print("\nТоп 5 най-често купувани продукти и среден брой в
поръчка:")
```

```
    print(avg_quantity_top_products)

    # c) Сегментиране на потребители по брой покупки
    purchase_counts =
purchases.groupby('user_id')['session_id'].nunique()
    def segment_users(count):
        if count == 1:
            return '1 Purchase'
        elif 2 <= count <= 5:
            return '2-5 Purchases'
        else:
            return '>5 Purchases'
    user_segments =
purchase_counts.apply(segment_users).reset_index(name='purchase_segment'
)
    df_activity_merged = pd.merge(df_activity, user_segments,
on='user_id', how='left')

    # d) Средна стойност на поръчка и най-чест източник на трафик за
сегмент
    segment_analysis =
df_activity_merged[df_activity_merged['event_type'] ==
'purchase'].groupby('purchase_segment').agg(
        average_order_value=('purchase_amount', 'mean'),
        most_common_traffic_source=('traffic_source', lambda x:
x.mode()[0] if not x.empty else None),
        num_users=('user_id', 'nunique')
    ).reset_index()
    print("\nАнализ на потребителски сегменти:")
    print(segment_analysis)

    # e) Време между добавяне в количката и покупка (пример за една
категория)
    cart_additions = df_activity[df_activity['event_type'] ==
'add_to_cart']
    purchases_category = df_activity[(df_activity['event_type'] ==
'purchase') & (df_activity['category'] == 'Electronics')]
    merged_time = pd.merge(cart_additions, purchases_category,
on=['user_id', 'product_id', 'session_id'], suffixes=('_cart',
'_purchase'))
    merged_time['time_diff'] = (merged_time['event_timestamp_purchase']
- merged_time['event_timestamp_cart']).dt.total_seconds()
    avg_time_to_purchase_electronics = merged_time['time_diff'].mean() /
60 if not merged_time.empty else None
    print(f"\nСредно време до покупка за категория 'Electronics'
(минути): {avg_time_to_purchase_electronics:.2f}")

    # 4. Използване на BigQuery SQL за средна продължителност на сесия
по устройство
    client = bigquery.Client(project=project_id)
    query_session_duration = f"""
    SELECT device_type, AVG(TIMESTAMP_DIFF(last_event, first_event,
SECOND)) AS avg_session_duration
    FROM (
```

```
        SELECT
            session_id,
            device_type,
            MIN(event_timestamp) AS first_event,
            MAX(event_timestamp) AS last_event
        FROM `{project_id}.your_dataset.user_activity`
        WHERE event_timestamp >=
TIMESTAMP_SECONDS({int(timestamp_thirty_days_ago)})
        GROUP BY session_id, device_type
    )
    GROUP BY device_type;
    """
    df_session_duration =
client.query(query_session_duration).to_dataframe()
    print("\nСредна продължителност на сесия по устройство (секунди):")
    print(df_session_duration)

    # 5. Създаване на обобщен DataFrame за потребителски сегменти (вече
създаден: segment_analysis)

    # 6. Запис на обобщения DataFrame в BigQuery
    destination_table_segments = 'your_dataset.user_purchase_segments'
    segment_analysis.to_gbq(destination_table_segments,
project_id=project_id, if_exists='replace', location='US')
    print(f"\nОбобщен DataFrame 'segment_analysis' е записан в BigQuery
таблица '{destination_table_segments}'.")

except Exception as e:
    print(f"Възникна грешка при работа с BigQuery: {e}")
```

Разбор на сложността:

- **Четене на голям обем данни:** Казусът предполага работа с голям dataset, което може да изисква четене на порции.
- **Сложен анализ с Pandas:** Включва филтриране, групиране, обединяване на данни, изчисляване на сложни метрики (среден брой, средна стойност, времеви разлики), и сегментиране на потребители.
- **Използване на специфични функции на BigQuery SQL:** Използва се `TIMESTAMP_DIFF` за изчисляване на продължителност на сесия, което е по-ефективно да се направи на ниво BigQuery за големи обеми.
- **Комбиниране на Pandas и BigQuery:** Данните се четат в Pandas за гъвкав анализ, а някои по-тежки операции (като изчисляване на продължителност на сесии) се извършват директно в BigQuery.
- **Записване на резултати:** Обобщената информация се записва обратно в BigQuery за по-нататъшно използване или визуализация.

Този казус демонстрира как Pandas и BigQuery могат да работят заедно за анализ на големи и сложни набори от данни, като се използват най-добрите инструменти за всяка част от процеса. Не забравяйте да адаптирате кода според структурата на вашата BigQuery таблица и вашите конкретни нужди за анализ.

Въпроси:

1. Обяснете каква е ролята на библиотеката `pandas-gbq`. Защо е необходима за взаимодействие между Pandas и Google BigQuery?
2. Кои са основните функции, предоставени от `pandas-gbq` за четене и запис на данни в BigQuery? Опишете накратко тяхната функционалност и основни параметри.
3. Какви са начините за управление на GCP credentials, които `pandas-gbq` може да използва? Кой метод се препоръчва за автоматизирани процеси?
4. Обяснете предимствата на използването на BigQuery Storage API при четене на данни с `pandas-gbq`. Как се активира тази функционалност?
5. Как може да се контролира поведението при запис на DataFrame в съществуваща BigQuery таблица с помощта на `pandas-gbq`? Кои са възможните опции?
6. В какви ситуации би било полезно да се четат данни от BigQuery на порции с помощта на `pandas-gbq`? Как се осъществява това?
7. Как може да се укаже SQL диалектът (Standard SQL или Legacy SQL), който да се използва при четене на данни от BigQuery с `pandas-gbq`?

Задачи:

1. **(Четене на данни):**
   - Свържете се с вашия Google BigQuery проект.
   - Използвайте `pandas.read_gbq()` за да прочетете всички колони от таблица `your_dataset.your_table` (заменете с реални имена).
   - Изведете първите 5 реда от получения DataFrame.
2. **(Филтриране при четене):**
   - Използвайки същата връзка и таблица от предходната задача, прочетете само редовете, където стойността на колона `event_type` е равна на 'purchase'.
   - Пребройте броя на тези редове в DataFrame-а.
3. **(Запис на DataFrame):**
   - Създайте примерен Pandas DataFrame с няколко колони и редове.
   - Запишете този DataFrame в нова BigQuery таблица `your_dataset.pandas_output` във вашия проект. Използвайте опцията да презапишете таблицата, ако вече съществува.
4. **(Добавяне на данни):**
   - Добавете няколко нови реда към DataFrame-а от предходната задача.
   - Запишете обновения DataFrame в същата BigQuery таблица `your_dataset.pandas_output`, като този път използвате опцията за добавяне на данни към съществуващата таблица.
   - Проверете в BigQuery конзолата дали новите редове са добавени.
5. **(Четене с BigQuery Storage API):**
   - Изпълнете заявката от задача 1, като използвате BigQuery Storage API (ако е инсталирана необходимата библиотека). Сравнете времето за изпълнение (ако е значително).
6. **(Използване на SQL диалект):**
   - Ако имате Legacy SQL заявка, която искате да изпълните, използвайте параметъра `dialect='legacy'` във функцията `pd.read_gbq()`. (Може да се наложи да намерите примерна Legacy SQL заявка).
7. **(Анализ след четене):**
   - Прочетете данни за уеб събития (например, с колони `user_id`, `session_id`, `event_type`, `event_timestamp`) от BigQuery.

- Използвайте Pandas, за да намерите броя на уникалните потребители и средния брой събития на сесия.

XVIII. STATA файлове: (read_stata, to_stata)

STATA е мощен статистически софтуер, който използва собствен формат за съхранение на данни (.dta файлове). Pandas предоставя вградена поддръжка за четене и записване на данни във STATA формат чрез функциите `pd.read_stata()` и `DataFrame.to_stata()`.

1. Четене на STATA файлове (`pd.read_stata`)

Функцията `pandas.read_stata(filepath_or_buffer, convert_dates=None, index_col=None, convert_categoricals=True, prefer_int_like=True, preserve_dtypes=True, columns=None, order=None, iterator=False, chunksize=None, compression='infer', storage_options=None)` се използва за четене на STATA .dta файлове и връща Pandas DataFrame.

а) Основни параметри:

- **`filepath_or_buffer` (str, path object, or file-like object):** Пътят до STATA файла (локален или URL) или вече отворен file-like обект.
- **`convert_dates` (bool or list-like of str, default=None):** Ако е True, опитва се да конвертира разпознати STATA формати за дати в Pandas datetime обекти. Ако е списък от имена на колони, се опитва да конвертира само тези колони. Ако е None, автоматично разпознава и конвертира дати.
- **`index_col` (str, list-like of str, default=None):** Колона(и) за използване като индекс на върнатия DataFrame.
- **`convert_categoricals` (bool, default=True):** Ако е True, STATA categorical променливи се конвертират в Pandas Categorical dtype.
- **`prefer_int_like` (bool, default=True):** Ако е True, променливи, които изглеждат като целочислени, се четат като такива (ако е възможно без загуба на информация).
- **`preserve_dtypes` (bool, default=True):** Ако е True, се опитва да запази оригиналните STATA dtypes.
- **`columns` (list-like of str, default=None):** Списък с колони за четене. Ако е None, се четат всички колони.
- **`order` (list-like of str, default=None):** Ред на колоните във върнатия DataFrame.
- **`iterator` (bool, default=False):** Ако е True, връща StataReader обект за итериране през файла на порции.
- **`chunksize` (int, default=None):** Размер на порциите, които да се връщат при `iterator=True`.
- **`compression` (str, default='infer'):** Тип на компресия ('infer', 'gzip', 'bz2', 'zip', 'xz', None). Pandas автоматично ще определи компресията за повечето файлове.
- **`storage_options` (dict, optional):** Допълнителни опции, които се предават на съответния storage backend (например, credentials за cloud storage).

б) Пример за четене на STATA файл:

```
import pandas as pd

try:
    # Четене на STATA файл
```

```
    df_stata = pd.read_stata('data.dta')
    print("Данни, прочетени от STATA файла:")
    print(df_stata.head())

except FileNotFoundError:
    print("Файлът 'data.dta' не беше намерен.")
except Exception as e:
    print(f"Възникна грешка при четене на STATA файл: {e}")
```

!!!ВНИМАНИЕ!!! Уверете се, че файлът `data.dta` съществува в същата директория или предоставете пълния път до него.

2. Записване във STATA файл (`DataFrame.to_stata`)

Методът `DataFrame.to_stata(path, write_index=True, encoding=None, data_label=None, variable_labels=None, value_labels=None, write_var_labels=True, write_value_labels=True, version=114)` се използва за записване на Pandas DataFrame във STATA .dta файл.

а) Основни параметри:

- **`path` (str, path object, or file-like object):** Пътят до файла, в който да се запишат STATA данните.
- **`write_index` (bool, default=True):** Дали да се запише индексът на DataFrame-а като първа колона (без име по подразбиране).
- **`encoding` (str, optional):** Кодиране, което да се използва при записване на файла (например, 'latin-1', 'utf-8'). Ако е None, STATA по подразбиране кодирането се използва.
- **`data_label` (str, optional):** Етикет за набора от данни в STATA файла.
- **`variable_labels` (dict, optional):** Речник, съдържащ етикети за променливите (колоните), където ключът е името на колоната, а стойността е етикетът.
- **`value_labels` (dict, optional):** Речник, съдържащ етикети за стойностите на categorical променливи. Ключът е името на колоната, а стойността е друг речник, където ключът е стойността, а стойността е етикетът.
- **`write_var_labels` (bool, default=True):** Дали да се записват етикетите на променливите.
- **`write_value_labels` (bool, default=True):** Дали да се записват етикетите на стойностите.
- **`version` (int, default=114):** Версия на STATA файла за записване (например, 105 за STATA 10, 114 за STATA 13-16).

б) Пример за записване във STATA файл:

```
import pandas as pd

# Създаване на примерен DataFrame
data = {'col1': [1, 2, 3], 'col2': ['A', 'B', 'C']}
df_to_stata = pd.DataFrame(data)

# Дефиниране на етикети на променливите
variable_labels = {'col1': 'Идентификатор', 'col2': 'Категория'}
```

```
# Дефиниране на етикети на стойностите (ако има categorical променливи)
value_labels = {'col2': { 'A': 'Първа', 'B': 'Втора', 'C': 'Трета'}}

try:
    # Записване на DataFrame във STATA файл
    df_to_stata.to_stata('output.dta', variable_labels=variable_labels,
value_labels=value_labels)
    print("DataFrame-ът е успешно записан във файл 'output.dta'.")

except Exception as e:
    print(f"Възникна грешка при записване в STATA файл: {e}")
```

Този пример показва как можете да запишете DataFrame и да добавите метаданни като етикети на променливите и стойностите, които могат да бъдат четени от STATA.

Работата със STATA файлове в Pandas е сравнително проста и позволява лесен обмен на данни между Python среди за анализ и STATA. Функциите `read_stata` и `to_stata` поддържат голям набор от опции за контрол на процеса на четене и записване, включително обработка на дати, categorical променливи, компресия и метаданни.

3. Допълнителни аспекти:

1) **Версии на STATA файлове:** Pandas поддържа четене и записване на различни версии на STATA .dta файлове. Параметърът `version` в `to_stata()` позволява да укажете версията, в която да бъде записан файлът, което може да е важно за съвместимост със стари версии на STATA. По подразбиране се използва версия 114 (STATA 13-16).
2) **Categorical Dtype:** Когато четете STATA файлове с `convert_categoricals=True` (по подразбиране), STATA categorical променливите се конвертират в Pandas Categorical dtype. Това е ефективен начин за представяне на категорийни данни и може да запази информацията за дефинираните в STATA етикети на стойностите. Тези етикети могат да бъдат достъпни чрез атрибута `.cat.categories` и `.cat.codes` на колоната в DataFrame-а.
3) **Кодиране на символи:** STATA файловете могат да бъдат записани с различни кодирания на символите. Ако срещате проблеми с четенето на файлове с не-ASCII символи, може да се наложи да експериментирате с различни кодирания, като зададете параметъра `encoding` в `read_stata()`. За записване, параметърът `encoding` в `to_stata()` може да бъде използван за указване на желаното кодиране.
4) **Работа с липсващи стойности:** Pandas и STATA използват различни конвенции за представяне на липсващи стойности. Pandas използва `NaN` (Not a Number), докато STATA има различни типове липсващи стойности (например, `.`, `.a`, `.b`, ..., `.z`). `read_stata()` обикновено конвертира всички STATA липсващи стойности в `NaN` на Pandas. При записване с `to_stata()`, Pandas `NaN` стойностите се записват като стандартната липсваща стойност на STATA (`.`).
5) **Итерация и четене на порции:** За много големи STATA файлове, които не могат да се поместят в паметта, можете да използвате параметрите `iterator=True` и `chunksize` във `read_stata()`. Това ще върне итератор, който може да се използва за четене на файла на по-малки порции (DataFrame-и).
6) **Съхранение в облака:** Подобно на други I/O функции на Pandas, `read_stata()` и `to_stata()` могат да работят с файлове, съхранени в облачни хранилища (например, AWS S3, Google Cloud Storage, Azure Blob Storage), като се предоставят подходящи `storage_options`.

4. Изключения и потенциални проблеми:

1) **Липса на библиотека:** За да работите със STATA файлове, Pandas трябва да има инсталирана зависимост, обикновено `tabulate` (за по-стари версии на Pandas може да е `tables`, но това е по-рядко за STATA). Ако срещнете грешка, свързана с липсващ модул, уверете се, че необходимите библиотеки са инсталирани (`pip install tabulate`).
2) **Неподдържани формати или версии:** Въпреки че Pandas поддържа повечето често използвани версии на STATA файлове, е възможно да има проблеми с четенето на много стари или много нови (все още неофициално поддържани) формати. В такива случаи може да се наложи да конвертирате файла с помощта на STATA или друг инструмент.
3) **Повреден файл:** Ако STATA файлът е повреден или не е записан правилно, `read_stata()` може да върне грешка или да прочете данните некоректно.
4) **Проблеми с кодирането:** Неправилното указване на кодирането при четене може да доведе до проблеми с показването на не-ASCII символи (например, кирилица, специални знаци). Уверете се, че знаете кодирането на файла или опитайте с различни опции (например, 'latin-1', 'utf-8').
5) **Загуба на специфични STATA метаданни:** Въпреки че Pandas се опитва да запази етикети на променливи и стойности, е възможно някои по-специфични метаданни, които STATA поддържа, да не бъдат напълно запазени при конвертирането към DataFrame.
6) **Разлики в типовете данни:** Въпреки че Pandas се стреми да извършва разумни конверсии на типове данни, може да има случаи, в които STATA тип данни няма точно съответствие в Pandas. Това може да доведе до леки разлики в представянето или прецизността на данните.

Казус 1: Анализ на данни от социологическо проучване

Ситуация:

Работите в изследователски екип, който е провел голямо социологическо проучване. Данните от проучването са ви предоставени като STATA .dta файл (`survey_data.dta`). Файлът съдържа отговори на множество въпроси, демографска информация за респондентите и други релевантни променливи.

Задача:

1. Прочетете файла `survey_data.dta` в Pandas DataFrame.
2. Разгледайте структурата на DataFrame-а (брой редове, колони, типове данни).
3. Проверете за липсващи стойности в основните демографски колони (например, възраст, пол, образование).
4. Изчислете средната възраст на участниците в проучването.
5. Намерете разпределението на участниците по пол и образователно ниво.
6. Ако файлът съдържа етикети на променливите и стойностите, уверете се, че Pandas ги е прочел коректно (като Categorical dtype, ако е приложимо).
7. Създайте нов DataFrame, съдържащ само респондентите с висше образование, и го запишете като нов STATA файл (`higher_education.dta`).

Решение:

```
import pandas as pd

try:
    # 1. Четене на STATA файла
    survey_df = pd.read_stata('survey_data.dta')
```

```
    # 2. Разглеждане на структурата
    print("Информация за DataFrame:")
    print(survey_df.info())
    print("\nПърви 5 реда:")
    print(survey_df.head())

    # 3. Проверка за липсващи стойности
    demographic_cols = ['възраст', 'пол', 'образование']  # Заменете с
реалните имена на колоните
    print("\nБрой липсващи стойности в демографските колони:")
    print(survey_df[demographic_cols].isnull().sum())

    # 4. Средна възраст
    if 'възраст' in survey_df.columns:
        mean_age = survey_df['възраст'].mean()
        print(f"\nСредна възраст на участниците: {mean_age:.2f}")
    else:
        print("\nКолоната 'възраст' не е намерена.")

    # 5. Разпределение по пол и образование
    if 'пол' in survey_df.columns:
        gender_distribution = survey_df['пол'].value_counts()
        print("\nРазпределение по пол:")
        print(gender_distribution)
    else:
        print("\nКолоната 'пол' не е намерена.")

    if 'образование' in survey_df.columns:
        education_distribution = survey_df['образование'].value_counts()
        print("\nРазпределение по образователно ниво:")
        print(education_distribution)
    else:
        print("\nКолоната 'образование' не е намерена.")

    # 6. Проверка за Categorical dtype (ако има такива колони)
    for col in survey_df.select_dtypes(include='category').columns:
        print(f"\nКатегорийна колона '{col}':")
        print(f"  Категории: {survey_df[col].cat.categories}")
        print(f"  Кодове: {survey_df[col].cat.codes.head()}")

    # 7. Създаване на DataFrame за висше образование и запис
    if 'образование' in survey_df.columns:
        higher_education_df =
survey_df[survey_df['образование'].str.contains('висше', case=False,
na=False)] # Адаптирайте филтъра според данните
        higher_education_df.to_stata('higher_education.dta')
        print("\nРеспондентите с висше образование са записани във файл
'higher_education.dta'.")
    else:
        print("\nНе може да се създаде файл за висше образование, тъй
като колоната 'образование' не е намерена.")

except FileNotFoundError:
```

```
    print("Файлът 'survey_data.dta' не беше намерен.")
except Exception as e:
    print(f"Възникна грешка при обработката на STATA файла: {e}")
```

Казус 2: Конвертиране на данни от STATA за уеб базирана визуализация

Ситуация:

Имате набор от икономически данни, съхранени в STATA файл (`economic_indicators.dta`). Искате да използвате тези данни за създаване на интерактивни графики на уебсайт, който използва JavaScript библиотеки като D3.js или Chart.js. За тази цел е необходимо да конвертирате данните в JSON формат.

Задача:

1. Прочетете файла `economic_indicators.dta` в Pandas DataFrame.
2. Изберете няколко ключови колони, които ще бъдат използвани за визуализацията (например, година, БВП, инфлация, безработица).
3. Преобразувайте DataFrame-а в списък от речници, където всеки речник представлява ред от данните, а ключовете са имената на колоните.
4. Запишете този списък от речници като JSON файл (`economic_data.json`).

Решение:

```
import pandas as pd
import json

try:
    # 1. Четене на STATA файла
    economic_df = pd.read_stata('economic_indicators.dta')

    # 2. Избор на ключови колони
    key_columns = ['година', 'БВП', 'инфлация', 'безработица']  # 
Заменете с реалните имена на колоните
    if all(col in economic_df.columns for col in key_columns):
        data_for_visualization = economic_df[key_columns]

        # 3. Преобразуване в списък от речници
        data_json = data_for_visualization.to_dict(orient='records')

        # 4. Записване като JSON файл
        with open('economic_data.json', 'w') as f:
            json.dump(data_json, f, indent=4)

        print("Икономическите данни са конвертирани и записани във файл 
'economic_data.json'.")

    else:
        print("Една или повече от ключовите колони не бяха намерени в 
STATA файла.")
```

```
except FileNotFoundError:
    print("Файлът 'economic_indicators.dta' не беше намерен.")
except Exception as e:
    print(f"Възникна грешка при обработката на STATA файла: {e}")
```

Въпроси:

1. Обяснете какви са предимствата на използването на Pandas за работа със STATA файлове (.dta) в сравнение с други методи.
2. Опишете основните параметри на функциите `pd.read_stata()` и `DataFrame.to_stata()`. Кога бихте използвали някои от по-специфичните параметри?
3. Как Pandas обработва STATA categorical променливи? Как можете да достъпите етикетите на стойностите, след като данните са прочетени в DataFrame?
4. Сравнете обработката на липсващи стойности между Pandas и STATA. Как се конвертират липсващите стойности при четене и записване на STATA файлове с Pandas?
5. Защо е важно да се обръща внимание на кодирането на символите при работа със STATA файлове? Как Pandas ви позволява да управлявате кодирането?
6. В какви ситуации би било полезно да четете STATA файлове на порции с помощта на Pandas? Как се реализира това?
7. Как Pandas се справя с различните версии на STATA файлове? Има ли ограничения или препоръки относно версиите?

Задачи:

1. **(Четене и основна информация):**
    - Прочетете STATA файл (`your_data.dta`).
    - Изведете броя на редовете и колоните, както и списък с имената на колоните.
    - Погледнете първите 10 реда от DataFrame-а.
2. **(Работа с Categorical данни):**
    - Прочетете STATA файл, който съдържа categorical променливи.
    - Изведете уникалните категории и техните кодове за една от categorical колоните.
    - Разпечатайте етикетите на стойностите за същата колона (ако има такива).
3. **(Филтриране и запис):**
    - Прочетете STATA файл.
    - Филтрирайте DataFrame-а, за да запазите само редовете, където стойността на определена колона отговаря на дадено условие.
    - Запишете резултата в нов STATA файл (`filtered_data.dta`).
4. **(Управление на липсващи стойности):**
    - Прочетете STATA файл, който съдържа липсващи стойности.
    - Запълнете липсващите стойности в една от числовите колони със средната стойност на колоната.
    - Запишете променения DataFrame в нов STATA файл.
5. **(Избор на колони и запис):**
    - Прочетете STATA файл, съдържащ много колони.
    - Създайте нов DataFrame, съдържащ само няколко избрани колони.
    - Запишете този подмножество от данни в STATA файл.
6. **(Четене на голям файл на порции):**
    - Ако разполагате с голям STATA файл, прочетете го на порции по 1000 реда.

    - За всяка порция, изчислете средната стойност на една от числовите колони и запазете резултатите.
    - Накрая, изчислете общата средна стойност от средните стойности на всяка порция.
7. **(Записване с метаданни):**
    - Създайте примерен Pandas DataFrame.
    - Създайте речник с етикети на променливите и речник с етикети на стойностите за някои от колоните.
    - Запишете DataFrame-а в STATA файл, включвайки тези метаданни. Отворете файла в STATA (ако имате достъп) и проверете дали етикетите са запазени.

XIX. Финални думи за I/O гъвкавостта на Pandas

Pandas се откроява като изключително гъвкава и мощна библиотека за работа с данни в Python, и нейните I/O възможности са ключов аспект от тази функционалност. Тя предоставя елегантен и интуитивен начин за четене и записване на данни в голям набор от файлови формати и източници на данни, което я прави незаменим инструмент за всеки, който се занимава с анализ на данни.

1. Гъвкавост и разнообразие на формати:

Както разгледахме в тази глава, Pandas поддържа широк спектър от формати, включително:

- **Текстови файлове:** CSV, TSV и други разделителни файлове, които са едни от най-често използваните формати за обмен на данни. Pandas предлага богати опции за персонализиране на процеса на парсване, обработка на различни кодирания, липсващи стойности и други нюанси.
- **Excel файлове:** Възможността за четене и записване на .xlsx и .xls файлове е изключително полезна за взаимодействие с данни, създадени или поддържани в Microsoft Excel.
- **JSON:** Поддръжката на JSON позволява лесно интегриране с уеб базирани приложения и NoSQL бази данни. Pandas може да обработва както плоски, така и по-сложни JSON структури.
- **HTML:** Функцията `read_html` е уникална и улеснява извличането на таблични данни директно от уеб страници.
- **XML:** Въпреки че може да изисква известна предварителна обработка или използване на допълнителни библиотеки, Pandas може да работи и с XML данни.
- **SQL бази данни:** Интеграцията със SQL чрез SQLAlchemy позволява безпроблемно четене и записване на данни от и към релационни бази данни.
- **Google BigQuery:** Специализираната поддръжка чрез `pandas-gbq` осигурява ефективна работа с големи обеми от данни в облачната платформа на Google.
- **STATA файлове:** Възможността за четене и записване на .dta файлове е важна за потребители, работещи със статистическия софтуер STATA.
- **Други формати:** Pandas също така поддържа формати като HDF5, Feather и Parquet, които са оптимизирани за съхранение и четене на големи набори от данни.

2. Предимства пред други подходи:

- **Интегрирана функционалност:** Pandas предоставя всички тези I/O възможности в рамките на една библиотека, което улеснява управлението на зависимостите и консистентността на кода.
- **Гъвкавост и контрол:** Всяка функция за четене и записване предлага множество параметри за фин контрол върху процеса на обработка на данните, като позволява адаптиране към специфични нужди и формати.

- **Автоматично преобразуване към DataFrame:** Резултатът от четенето на данни винаги е структуриран като DataFrame, което веднага позволява използването на мощните инструменти за анализ и манипулация на данни, които Pandas предлага.
- **Лесен синтаксис:** Функциите за I/O са проектирани да бъдат лесни за използване, като често се изисква само един ред код за четене или запис на данни.
- **Производителност:** Pandas е оптимизиран за работа с таблични данни и осигурява добра производителност дори при по-големи файлове (особено при използване на по-ефективни формати като Parquet или при четене на порции).

3. Обобщение на казусите и задачите:

- След всеки разгледан файлов формат, предоставените казуси и задачи имаха за цел да илюстрират реални сценарии, в които се налага взаимодействие с тези типове файлове. Те обхващаха основни операции като четене, записване, филтриране и трансформация на данни, както и по-специфични аспекти като работа с categorical данни (при STATA), използване на SQL заявки (при SQL бази данни и BigQuery) и извличане на данни от уеб страници (при HTML).
- Задачите бяха структурирани така, че да насърчат практическото прилагане на наученото и да подчертаят гъвкавостта на Pandas при справяне с различни източници на данни. Те също така засегнаха важни аспекти като обработка на грешки, управление на липсващи стойности и избор на подходящи параметри за конкретни ситуации.
- В заключение, I/O възможностите на Pandas са съществена част от нейната привлекателност като инструмент за анализ на данни. Разнообразието от поддържани формати, комбинирано с гъвкавостта и лекотата на използване на нейните функции, я прави незаменима библиотека за всеки, който работи с данни в Python. Разгледаните казуси и задачи само потвърждават тази гъвкавост и демонстрират как Pandas може да бъде ефективно приложен в различни реални сценарии.