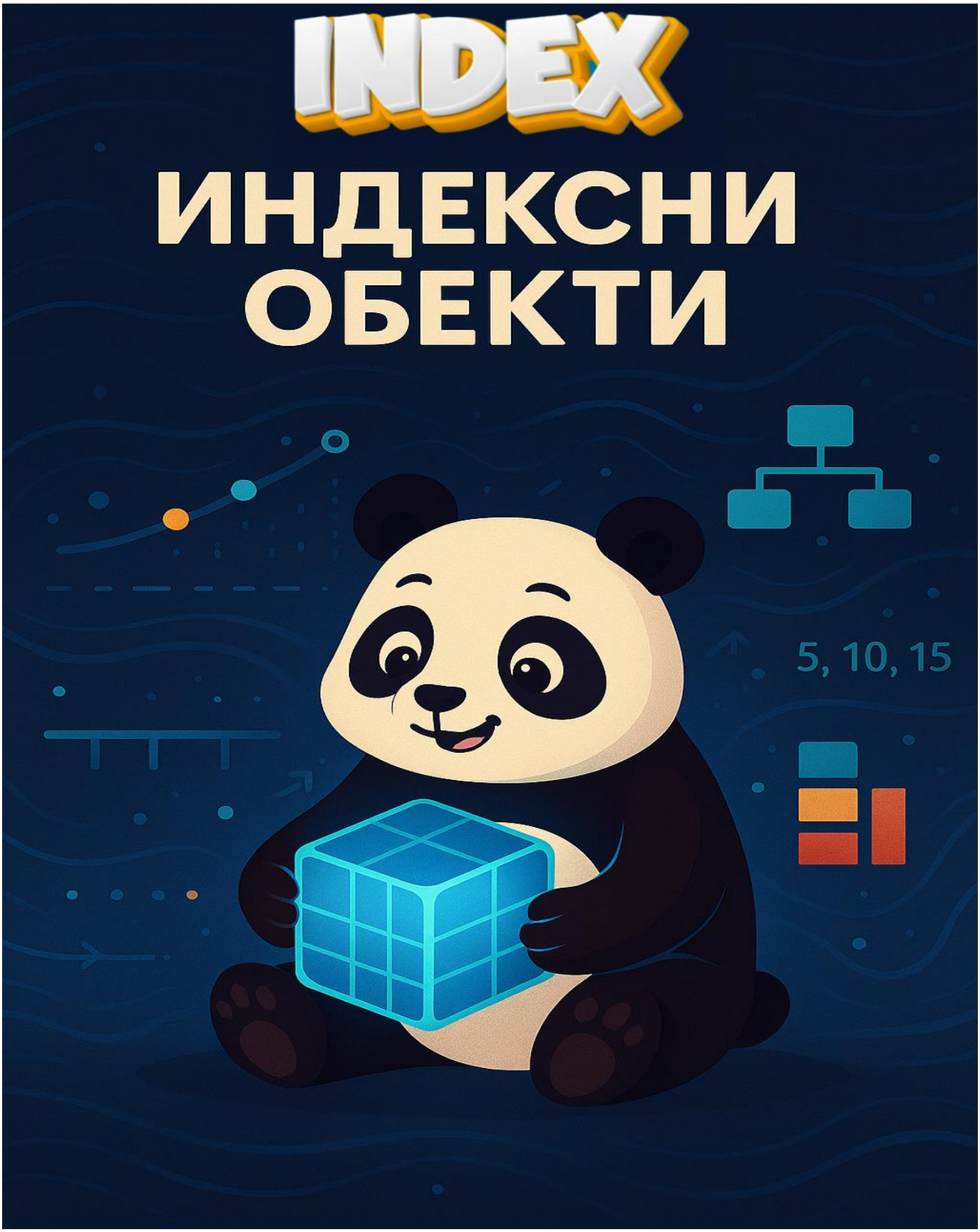
Глава V. Индексни Обекти (Index Objects)



В тази глава ще се задълбочим в разбирането на **индексите** в Pandas, които са фундаментален аспект на структурата на DataFrame и Series. Индексите не са просто етикети за редовете (и колоните при DataFrame), а мощни обекти, които осигуряват бърз достъп до данни, подравняване при операции и възможност за йерархично структуриране.

Ще разгледаме различните типове индексни обекти, които Pandas предлага, всеки от които е оптимизиран за специфични видове данни и сценарии. Разбирането на тези обекти и техните възможности ще ви даде по-голям контрол върху вашите данни и ще подобри ефективността на вашия анализ.

Ето какво ще покрием подробно в тази глава:

1. **Ролята и значението на индексите в pandas:** Ще започнем с обяснение защо индексите са толкова важни в Pandas. Ще разгледаме как те улесняват селекцията на данни, подравняването при операции между Series и DataFrame, и връзката им с концепцията за "маркирани данни".
2. **Основният клас Index и общи операции (атрибути, методи):** Ще се запознаем с базовия клас `Index`, от който произлизат всички други специфични типове индекси. Ще разгледаме общите атрибути (като `.shape`, `.dtype`, `.name`, `.is_unique`) и методи (като `.get_loc()`, `.reindex()`, `.drop()`, `.unique()`), които са налични за всички индексни обекти.
3. **Числови индекси (Int64Index, UInt64Index, Float64Index, NumericIndex):** Ще разгледаме индексите, които съдържат числови стойности от различни типове (цели и плаваща запетая). Ще видим как се създават и какви специфични операции могат да се прилагат върху тях.
4. **DatetimeIndex: Индекс за дата и час (обект Timestamp):** Този тип индекс е изключително важен при работа с времеви серии. Ще разгледаме как се създава `DatetimeIndex` от различни формати на дати и часове, както и специфичните методи за работа с времеви данни (например, селекция по дата, агрегиране по периоди).
5. **TimedeltaIndex: Индекс за времеви разлики (обект Timedelta):** Ще се запознаем с индекса, който представлява времеви разлики или продължителности. Ще видим как се създава и какви операции са приложими (например, аритметични операции с дати и часове).
6. **PeriodIndex: Индекс за периоди (обект Period):** Този тип индекс представлява периоди от време (например, месеци, години). Ще разгледаме създаването и специфичните операции за работа с периоди (например, преобразуване между честоти).
7. **CategoricalIndex: Индекс за категорийни данни:** Ще видим как можем да използваме категорийни данни като индекс, което може да доведе до оптимизация на паметта и по-бързи операции при определени сценарии.
8. **IntervalIndex: Индекс за интервали (обект Interval):** Този индекс представлява затворени, отворени или полуотворени интервали. Ще разгледаме неговото създаване и приложения (например, при групиране на числови данни в интервали).
9. **MultiIndex: Йерархично (многонивово) индексиране (кратко запознаване с MultiIndex):** Ще направим встъпително запознаване с `MultiIndex`, който позволява да имате повече от едно ниво на индексиране както за редовете, така и за колоните. Това е мощна концепция за представяне на многоизмерни данни в двуизмерна структура. Подробното разглеждане на `MultiIndex` ще бъде в следваща глава.

Чрез тази глава ще придобиете задълбочено разбиране за индексните обекти в Pandas и как те могат да бъдат ефективно използвани за управление и анализ на вашите данни.

I. Ролята и значението на индексите в pandas.

В Pandas, **индексът** е фундаментална концепция, която присъства както в `Series`, така и в `DataFrame`. Той изпълнява няколко ключови роли, които правят работата с данни по-ефективна и интуитивна. Нека разгледаме основните аспекти на ролята и значението на индексите:

1. Уникална идентификация и етикетиране на данни:

- Индексът осигурява **етикети** за всеки ред (в `Series` и `DataFrame`) и за всяка колона (в `DataFrame`). Тези етикети позволяват **уникално идентифициране** на отделни записи и колони.
- В много случаи индексът може да бъде съставен от смислени стойности (например, дати, имена на продукти, потребителски ID-та), което прави данните по-разбираеми и контекстуални.
- За разлика от позиционното индексиране (както при списъците в Python), базираното на етикети индексиране прави кода по-четим и по-малко податлив на грешки при промени в реда на данните.

2. Бърз и ефективен достъп до данни (селекция):

- Индексите в Pandas са оптимизирани структури данни, които позволяват **бързо извличане** на редове и колони въз основа на техните етикети.
- Когато имате голям набор от данни, използването на индекс за селекция (чрез `.loc`) е значително по-бързо от итерирането или филтрирането по условия върху самите данни.
- Това е особено важно при повтарящи се операции за извличане на конкретни подмножества от данни.

3. Подравняване на данни при операции:

- Една от ключовите сили на Pandas е способността му автоматично да **подравнява данни** въз основа на индексите при извършване на операции между `Series` и `DataFrame`.
- Когато извършвате аритметични или други поелементни операции между два обекта на Pandas, те се опитват да съпоставят елементите си по техните индекси. Ако индексните етикети не съвпадат, резултатът ще съдържа липсващи стойности (`NaN`) за несъвпадащите елементи.
- Това автоматично подравняване е изключително полезно за комбиниране на данни от различни източници или за извършване на времеви серии анализи, където датите служат като индекс.

4. Възможност за йерархично (многонивово) индексиране (MultiIndex):

- Pandas позволява създаването на **многонивови индекси** (`MultiIndex`), които дават възможност за представяне на данни с по-висока размерност в двуизмерна структура (DataFrame или Series).
- `MultiIndex` позволява групиране и анализ на данни по множество измерения (например, продажби по година и регион).

5. Поддържане на целостта на данните:

- Уникалните индекси могат да помогнат за гарантиране на **целостта на данните**, като предотвратяват дублирането на редове или колони (в зависимост от това дали индексът е зададен като уникален).

Пример, илюстриращ подравняване по индекс:

```
import pandas as pd

s1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
s2 = pd.Series([5, 15, 25], index=['b', 'c', 'd'])

print("Series s1:\n", s1)
print("\nSeries s2:\n", s2)

# Събиране на двата Series
s_sum = s1 + s2
print("\nSum of s1 and s2 (aligned by index):\n", s_sum)
```

В този пример, при събирането на s1 и s2, Pandas автоматично подравнява елементите по техните индекси ('a', 'b', 'c', 'd'). За индекса 'a' от s1 няма съответстващ индекс в s2, и обратното за 'd', затова резултатът съдържа NaN за тези случаи. Елементите с общи индекси ('b' и 'c') се събират коректно.

В заключение, индексите в Pandas са много повече от просто етикети. Те са ключова структура, която осигурява ефективен достъп, автоматично подравняване и възможност за сложна организация на данните, което ги прави съществена част от работата с Pandas.

II. Основният клас Index и общи операции (атрибути, методи).

В Pandas, класът `Index` е базовият клас, от който произлизат всички специфични типове индекси, като `Int64Index`, `DatetimeIndex`, `CategoricalIndex` и други. Той представлява неизменяема (immutable) последователност от етикети. Разбирането на основните атрибути и методи на класа `Index` е важно, тъй като те са общи за всички типове индекси и предоставят основни функционалности за работа с тях.

1. Създаване на Index обект:

Можете да създадете `Index` обект директно от списък или друг итерируем обект:

```
import pandas as pd

# Създаване на Index от списък
index_labels = pd.Index(['a', 'b', 'c', 'b'])
print("Index object:\n", index_labels)
print("\nType of index_labels:", type(index_labels))
```

2. Основни атрибути на Index обекта:

- **`.values`:** Връща NumPy array, съдържащ етикетите на индекса.

```
print("\nValues of the index:\n", index_labels.values)
print("\nType of values:", type(index_labels.values))
```

- **`.dtype`:** Връща типа на данните на етикетите в индекса.

```
print("\nData type of the index:", index_labels.dtype)
```

- **`.shape`:** Връща tuple, представляващ размерността на индекса (винаги ще бъде `(n,)`, където `n` е броят на етикетите).

```
print("\nShape of the index:", index_labels.shape)
```

- **`.size`:** Връща броя на етикетите в индекса (еквивалентно на първия елемент на `.shape`).

```
print("\nSize of the index:", index_labels.size)
```

- **`.name`:** Връща името на индекса (може да бъде `None`). Индексите могат да имат име, което е полезно при работа с `DataFrame` с много нива на индекс.

```
index_with_name = pd.Index(['x', 'y', 'z'], name='labels')
print("\nIndex with name:\n", index_with_name)
print("\nName of the index:", index_with_name.name)
```

- **`.is_unique`:** Връща `True`, ако всички етикети в индекса са уникални, и `False` в противен случай.

```
print("\nIs index_labels unique?", index_labels.is_unique)
index_unique = pd.Index(['p', 'q', 'r'])
```

```
print("\nIs index_unique unique?", index_unique.is_unique)
```

- **`.is_monotonic_increasing` / `.is_monotonic_decreasing`:** Връща `True`, ако етикетите в индекса са монотонно нарастващи или намаляващи съответно.

```
index_increasing = pd.Index([1, 2, 3, 5])
print("\nIs index_increasing monotonic increasing?",
index_increasing.is_monotonic_increasing)
index_decreasing = pd.Index([5, 3, 2, 1])
print("\nIs index_decreasing monotonic decreasing?",
index_decreasing.is_monotonic_decreasing)
```

3. Основни методи на Index обекта:

- **`.get_loc(label)`:** Връща целочисленото местоположение (или слайс, или булев масив) за дадения етикет. Ако етикетът се среща няколко пъти, може да върне `slice` или `np.ndarray`.

```
print("\nLocation of 'a' in index_labels:", index_labels.get_loc('a'))
print("\nLocation of 'b' in index_labels:", index_labels.get_loc('b'))
```

- **`.reindex(labels, method=None, level=None, copy=True, fill_value=np.nan, limit=None, tolerance=None)`:** Създава нов индекс и DataFrame/Series, съобразен с новите етикети. Може да се използва за добавяне, премахване или пренареждане на етикети.

```
s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
new_index = pd.Index(['a', 'b', 'd'])
reindexed_s = s.reindex(new_index, fill_value=0)
print("\nOriginal Series:\n", s)
print("\nReindexed Series:\n", reindexed_s)
```

- **`.drop(labels, level=None, errors='raise')`:** Връща нов Index с премахнати указани етикети.

```
dropped_index = index_labels.drop('b')
print("\nIndex after dropping 'b':\n", dropped_index)
```

- **`.unique()`:** Връща нов Index с уникалните етикети от оригиналния индекс.

```
unique_index = index_labels.unique()
```

```
print("\nUnique labels in index_labels:\n", unique_index)
```

4. Аритметични и сравнителни операции:

Индексните обекти поддържат някои аритметични (+, -) и сравнителни (==, !=, >, <) операции, които се прилагат поелементно. Резултатът обикновено е NumPy array или булев array.

```
import pandas as pd

index_int1 = pd.Index([10, 20, 30])
index_int2 = pd.Index([5, 20, 25])
index_str1 = pd.Index(['apple', 'banana', 'cherry'])
index_str2 = pd.Index(['banana', 'date', 'elderberry'])

# Аритметични операции (поддържат се основно за числови индекси)
try:
    sum_index = index_int1 + index_int2
    print("\nindex_int1 + index_int2:", sum_index)
except TypeError as e:
    print("\nError during addition of numeric indices:", e)

try:
    diff_index = index_int1 - index_int2
    print("\nindex_int1 - index_int2:", diff_index)
except TypeError as e:
    print("\nError during subtraction of numeric indices:", e)

# Сравнителни операции
equal_index = index_int1 == index_int2
print("\nindex_int1 == index_int2:", equal_index)

not_equal_index = index_int1 != index_int2
print("\nindex_int1 != index_int2:", not_equal_index)

greater_than_index = index_int1 > index_int2
print("\nindex_int1 > index_int2:", greater_than_index)
```

```
less_than_index = index_int1 < index_int2
print("\nindex_int1 < index_int2:", less_than_index)

# Сравнение на стрингови индекси (лексикографско сравнение)
equal_str_index = index_str1 == index_str2
print("\nindex_str1 == index_str2:", equal_str_index)

greater_than_str_index = index_str1 > index_str2
print("\nindex_str1 > index_str2:", greater_than_str_index)
```

Пояснения към примерите:

- **Аритметични операции:** Както се вижда от примера, аритметичните операции (`+`, `-`, `*`, `/` и др.) се поддържат основно за числови индексни обекти (`Int64Index`, `Float64Index` и др.). Опитът за извършване на аритметични операции върху стрингови индекси ще доведе до `TypeError`. Резултатът от аритметичните операции е нов `Index` обект със съответните резултати от поелементните операции.
- **Сравнителни операции:** Сравнителните операции (`==`, `!=`, `>`, `<`, `>=`, `<=`) се поддържат както за числови, така и за стрингови индексни обекти.
    - При числови индекси сравнението се извършва по стойност.
    - При стрингови индекси сравнението се извършва лексикографски (по азбучен ред). Резултатът от сравнителните операции е NumPy булев масив, показващ резултата от сравнението на всеки елемент.

Разбирането на тези основни атрибути и методи на класа `Index` е важно, тъй като те са градивните елементи за работа с всички специфични типове индекси в Pandas. В следващите теми ще разгледаме как тези общи концепции се прилагат и разширяват в различните видове индекси.

III. Числови индекси (Int64Index, UInt64Index, Float64Index, NumericIndex)

Числовите индекси в Pandas са специализирани типове индекси, които съдържат числови стойности. Те са оптимизирани за работа с числови данни и поддържат специфични операции, свързани с тяхната числена природа. Основните класове числови индекси са:

- **`Int64Index`:** За 64-битови цели числа.
- **`UInt64Index`:** За 64-битови беззнакови цели числа.
- **`Float64Index`:** За 64-битови числа с плаваща запетая.
- **`NumericIndex`:** Абстрактен базов клас за всички числови индекси.

В повечето случаи, когато създавате `Series` или `DataFrame` с числови етикети за редовете (и не указвате изрично друг тип индекс), Pandas автоматично ще използва един от тези числови индекси в зависимост от типа на предоставените числа.

1. Създаване на числови индекси:

```
import pandas as pd
import numpy as np

# Int64Index
int_index = pd.Index([1, 2, 3, 4, 5])
print("Int64Index:", int_index)
print("dtype:", int_index.dtype)

# UInt64Index
uint_index = pd.Index(np.array([1, 2, 3, 4, 5], dtype=np.uint64))
print("\nUInt64Index:", uint_index)
print("dtype:", uint_index.dtype)

# Float64Index
float_index = pd.Index([1.0, 2.5, 3.0, 4.5])
print("\nFloat64Index:", float_index)
print("dtype:", float_index.dtype)

# Създаване на Series и DataFrame с числови индекси
series_with_int_index = pd.Series(['a', 'b', 'c'], index=[10, 20, 30])
print("\nSeries with Int64Index:\n", series_with_int_index)

df_with_float_index = pd.DataFrame({'col1': [10, 20], 'col2': [30, 40]},
index=[1.5, 3.0])
print("\nDataFrame with Float64Index:\n", df_with_float_index)
```

2. Специфични операции и поведение на числовите индекси:

Тъй като съдържат числа, тези индекси поддържат всички общи операции на класа `Index`, както и някои допълнителни, свързани с тяхната числена природа:

- **Сортиране:** Числовите индекси могат да бъдат лесно сортирани по числова стойност с метода `.sort_values()`.

```
unsorted_int_index = pd.Index([3, 1, 4, 2])
sorted_int_index = unsorted_int_index.sort_values()
print("\nUnsorted Int64Index:", unsorted_int_index)
print("Sorted Int64Index:", sorted_int_index)
```

- **Сравнение и аритметични операции:** Както видяхме в предходната тема, числовите индекси поддържат поелементни сравнения и аритметични операции (резултатът е обикновено NumPy array или нов `NumericIndex`).
- **Селекция с помощта на слайсинг:** Когато числовият индекс е монотонно нарастващ или намаляващ, можете да използвате слайсинг с числови стойности (в допълнение към целочисленото позиционно слайсиране).

```
numeric_series = pd.Series(['p', 'q', 'r', 's'], index=[10, 15, 20, 25])
print("\nNumeric Series:\n", numeric_series)

# Селекция по стойност (включва крайните точки, ако съществуват)
sliced_series = numeric_series[12:22]
print("\nSliced Series by value (12:22):\n", sliced_series)

# Внимание: Това е различно от целочисленото позиционно слайсиране!
positional_slice = numeric_series[1:3]
print("\nPositional slice (1:3):\n", positional_slice)
```

!!!Важно: Когато използвате слайсинг с числови етикети, Pandas ще върне всички елементи, чиито етикети попадат в зададения диапазон (включително крайните точки, ако съществуват в индекса). Това е различно от стандартния Python слайсинг по индекс, където крайната точка е изключена.

• **Използване с `.loc` и `.iloc`:** Числовите индекси работят безпроблемно с методите `.loc` (за селекция по етикет - числовата стойност) и `.iloc` (за селекция по целочислена позиция).

```
print("\nSelection by label (.loc):", numeric_series.loc[15])
print("Selection by position (.iloc):", numeric_series.iloc[1])
```

3. Кога да използваме числови индекси?

Числовите индекси са естествени, когато етикетите на вашите данни са числови по природа, като например:

- Идентификатори (ID-та).
- Резултати от измервания.
- Дискретни времеви точки.

Въпреки че Pandas е гъвкав и позволява използването на числа като етикети, за времеви серии данни е по-подходящо да се използва `DatetimeIndex`, който предлага много повече специфични функционалности за работа с времеви данни.

IV. DatetimeIndex: Индекс за дата и час (обект Timestamp)

`DatetimeIndex` е изключително важен и мощен тип индекс в Pandas, специално предназначен за работа с **времеви серии данни**. Той е оптимизиран за съхранение и манипулиране на последователности от дати и часове. Всеки елемент в `DatetimeIndex` е обект от тип `Timestamp`, който представлява единичен момент във времето.

1. Създаване на DatetimeIndex:

`DatetimeIndex` може да бъде създаден по няколко начина:

а) Използване на `pd.to_datetime()`:

Тази функция може да конвертира различни формати на дати и часове в `DatetimeIndex`.

```
import pandas as pd

dates_list = ['2023-01-01', '2023-01-05', '2023-01-10']
datetime_index_from_list = pd.to_datetime(dates_list)
print("DatetimeIndex from list:\n", datetime_index_from_list)
print("dtype:", datetime_index_from_list.dtype)

dates_series = pd.Series(['2023/02/01', '2023/02/05', '2023/02/10'])
datetime_index_from_series = pd.to_datetime(dates_series)
print("\nDatetimeIndex from Series:\n", datetime_index_from_series)
```

б) Използване на `pd.date_range()`:

Тази функция генерира последователност от дати в определен диапазон с зададена честота.

```
date_range_index = pd.date_range(start='2023-03-01', end='2023-03-10')
print("\nDatetimeIndex from date_range (daily):\n", date_range_index)

hourly_range_index = pd.date_range(start='2023-04-01 09:00', periods=5,
freq='H')
```

```
print("\nDatetimeIndex from date_range (hourly):\n", hourly_range_index)
```

Честотните низове (като 'D' за ден, 'H' за час, 'M' за край на месец и др.) са много гъвкави и позволяват генериране на различни времеви последователности.

в) Директно подаване на списък или NumPy array към `pd.Index()`:

```
import numpy as np


numpy_dates = np.array(['2023-05-01', '2023-05-05', '2023-05-10'],
dtype='datetime64[D]')
datetime_index_from_numpy = pd.Index(numpy_dates)
print("\nDatetimeIndex from NumPy datetime64 array:\n",
datetime_index_from_numpy)
print("dtype:", datetime_index_from_numpy.dtype)
```

2. Специфични операции и предимства на DatetimeIndex:

`DatetimeIndex` предлага множество специфични методи и улеснява операции, свързани с времеви серии:

а) Селекция по дата и час:

Можете лесно да селектирате данни по конкретна дата, диапазон от дати или части от дата и час.

```
dates = pd.date_range('2023-01-01', periods=10, freq='D')
ts = pd.Series(np.random.randn(len(dates)), index=dates)
print("Time Series:\n", ts)

print("\nData for '2023-01-05':\n", ts['2023-01-05'])
print("\nData for '2023-01-03' to '2023-01-07':\n", ts['2023-01-
03':'2023-01-07'])
print("\nData for January 2023:\n", ts['2023-01'])
```

б) Слайсинг по дата:

Подобно на числовите индекси, можете да използвате слайсинг с дати, когато `DatetimeIndex` е сортиран.

в) Преобразуване на честота (`.resample()`):

Един от най-мощните методи за времеви серии, позволява агрегиране или преобразуване на данни в друга честота (например, от дневна към месечна).

```
daily_data = pd.Series(np.random.randn(30), index=pd.date_range('2023-
01-01', periods=30, freq='D'))
monthly_mean = daily_data.resample('M').mean()
print("\nDaily Data:\n", daily_data.head())
print("\nMonthly Mean:\n", monthly_mean)
```

г) Плъзгащи статистики (`.rolling()`):

Изчисляване на плъзгащи средни, суми и други статистики за определен времеви прозорец.

```
rolling_mean = daily_data.rolling(window=7).mean()
print("\n7-day Rolling Mean:\n", rolling_mean.head(10))
```

д) Работа с часови зони:

`DatetimeIndex` може да бъде направен aware за часови зони, което е важно при работа с данни от различни географски локации.

```
aware_index = pd.date_range('2023-01-01', periods=3, freq='D',
tz='Europe/Sofia')
print("\nDatetimeIndex with Timezone:\n", aware_index)
```

е) Различни методи за преместване на времеви точки (`.shift()`, `.tshift()`):

Позволяват преместване на данните напред или назад във времето.

ж) Извличане на компоненти на дата и час (.dt аксесор):

`DatetimeIndex` (както и колони с datetime данни) има `.dt` аксесор, който позволява лесно извличане на компоненти като година, месец, ден, час, минута, секунда, ден от седмицата и други.

`DatetimeIndex` е незаменим инструмент за анализ на времеви серии данни в Pandas. Неговите специфични функционалности улесняват често срещани задачи като филтриране, агрегиране и преобразуване на данни по времеви интервали.

V. TimedeltaIndex: Индекс за времеви разлики (обект Timedelta).

`TimedeltaIndex` е тип индекс в Pandas, който съдържа обекти от тип `Timedelta`. Тези обекти представляват **продължителност, разлика между две дати или часове**. `TimedeltaIndex` е полезен, когато искате да индексирате данни по времеви интервали или да извършвате операции, свързани с времеви разлики.

1. Създаване на TimedeltaIndex:

Подобно на `DatetimeIndex`, `TimedeltaIndex` може да бъде създаден по няколко начина:

а) Използване на `pd.to_timedelta()`:

Тази функция може да конвертира различни формати на времеви разлики в `TimedeltaIndex`.

```
import pandas as pd

durations_list = ['1 days', '2 days 05:30:00', '0 days 12:00:00']
timedelta_index_from_list = pd.to_timedelta(durations_list)
print("TimedeltaIndex from list:\n", timedelta_index_from_list)
print("dtype:", timedelta_index_from_list.dtype)

durations_series = pd.Series(['1 hours', '1.5 hours', '2 hours 30
minutes'])
timedelta_index_from_series = pd.to_timedelta(durations_series)
print("\nTimedeltaIndex from Series:\n", timedelta_index_from_series)
```

б) Използване на `pd.timedelta_range()`:

Тази функция генерира последователност от времеви разлики в определен диапазон с зададена честота.

```
timedelta_range_index = pd.timedelta_range(start='1 day', periods=5,
freq='6H')
print("\nTimedeltaIndex from timedelta_range:\n", timedelta_range_index)

week_range_index = pd.timedelta_range(end='7 days', periods=3,
freq='3.5D')
print("\nTimedeltaIndex from timedelta_range (fractional days):\n",
week_range_index)
```

Честотните низове за TimedeltaIndex включват 'D' (дни), 'H' (часове), 'min' (минути), 'sec' (секунди), 'ms' (милисекунди), 'us' (микросекунди), 'ns' (наносекунди) и техните кратни.

в) Директно подаване на списък или NumPy array към `pd.Index()`:

```
import numpy as np
```

```
numpy_timedeltas = np.array([np.timedelta64(1, 'D'),
np.timedelta64(3600, 's'), np.timedelta64(2, 'h')])
timedelta_index_from_numpy = pd.Index(numpy_timedeltas)
print("\nTimedeltaIndex from NumPy timedelta64 array:\n",
timedelta_index_from_numpy)
print("dtype:", timedelta_index_from_numpy.dtype)
```

2. Специфични операции и предимства на TimedeltaIndex:

`TimedeltaIndex` позволява извършването на операции, свързани с времеви разлики:

а) Аритметични операции с `DatetimeIndex`:

Можете да събирате или изваждате `TimedeltaIndex` от `DatetimeIndex`, за да получавате нови `DatetimeIndex` обекти, представляващи бъдещи или минали моменти във времето.

```
dates = pd.to_datetime(['2023-01-01', '2023-01-05', '2023-01-10'])
time_offsets = pd.to_timedelta(['1 day', '3 days', '7 days'])

future_dates = dates + time_offsets
print("\nFuture dates:", future_dates)

past_dates = dates - time_offsets
print("\nPast dates:", past_dates)
```

б) Аритметични операции между `TimedeltaIndex` обекти:

Можете да извършвате аритметични операции (събиране, изваждане, умножение, деление) между два `TimedeltaIndex` обекта или между `TimedeltaIndex` и скаларна стойност (число, което се интерпретира като наносекунди).

```
td_index1 = pd.to_timedelta(['1 day', '2 days'])
td_index2 = pd.to_timedelta(['0.5 days', '1.5 days'])

sum_td = td_index1 + td_index2
print("\nSum of TimedeltaIndex:", sum_td)
```

```
scalar_mult = td_index1 * 2
print("\nTimedeltaIndex multiplied by scalar:", scalar_mult)
```

в) Сравнителни операции:

Можете да сравнявате елементи на `TimedeltaIndex` помежду си или със скаларни стойности.

```
print("\ntd_index1 > '1 day':", td_index1 > pd.to_timedelta('1 day'))
```

г) Абсолютна стойност (`.abs()`):

Връща `TimedeltaIndex` с абсолютните стойности на времевите разлики (полезно, ако има отрицателни стойности).

д) Извличане на компоненти (`.components`):

Предоставя DataFrame с компоненти на времевите разлики (дни, часове, минути, секунди, милисекунди, микросекунди, наносекунди).

```
td_index = pd.to_timedelta(['1 day 02:30:45', '-1 day 01:00:00'])
components_df = td_index.components
print("\nTimedeltaIndex components:\n", components_df)
```

`TimedeltaIndex` е ключов инструмент за анализ на събития във времето, изчисляване на продължителности, управление на закъснения и други сценарии, където времевите разлики са от значение. Той често се използва в комбинация с `DatetimeIndex` при анализ на времеви серии.

VI. PeriodIndex: Индекс за периоди (обект Period).

`PeriodIndex` е тип индекс в Pandas, който съдържа обекти от тип `Period`. Тези обекти представляват **периоди от време с фиксирана честота**, като например ден, месец, година, тримесечие и други. `PeriodIndex` е полезен, когато искате да индексирате данни по времеви периоди, а не по конкретни моменти във времето.

1. Създаване на PeriodIndex:

`PeriodIndex` може да бъде създаден по няколко начина:

а) Използване на `pd.PeriodIndex()`:

Този конструктор приема различни входни данни, включително списъци от дати или периодични низове, както и информация за честотата.

```
import pandas as pd

# От списък с дати и честота
periods_from_dates = pd.PeriodIndex(['2023-01-01', '2023-02-01', '2023-03-01'], freq='M')
print("PeriodIndex from dates (monthly):\n", periods_from_dates)
print("dtype:", periods_from_dates.dtype)

# От списък с периодични низове
period_strings = ['2023Q1', '2023Q2', '2023Q3']
periods_from_strings = pd.PeriodIndex(period_strings, freq='Q')
print("\nPeriodIndex from strings (quarterly):\n", periods_from_strings)
```

б) Използване на `pd.date_range()` с параметър `freq` и конвертиране към `PeriodIndex` с `.to_period()`:

```
date_rng = pd.date_range(start='2023-01-01', periods=3, freq='D')
period_index_from_range = date_rng.to_period(freq='W')
print("\nPeriodIndex from date_range (weekly):\n",
period_index_from_range)
```

в) Използване на `pd.period_range()`:

Тази функция генерира последователност от периоди в определен диапазон с зададена честота.

```
period_range_index = pd.period_range(start='2023-01', end='2023-03', freq='M')
print("\nPeriodIndex from period_range (monthly):\n",
period_range_index)

yearly_period_index = pd.period_range(start='2020', periods=5, freq='Y')
```

```
print("\nPeriodIndex from period_range (yearly):\n",
yearly_period_index)
```

Честотните низове за PeriodIndex са същите като тези за DatetimeIndex (например, 'D', 'W', 'M', 'Q', 'Y'), но те интерпретират времето като периоди, а не като моменти.

2. Специфични операции и предимства на PeriodIndex:

`PeriodIndex` предоставя функционалности, специфични за работа с времеви периоди:

а) Аритметични операции:

Можете да извършвате аритметични операции с `PeriodIndex` и цели числа. Събирането или изваждането на цяло число премества периода с този брой честотни единици.

```
monthly_periods = pd.PeriodIndex(['2023-01', '2023-02', '2023-03'],
freq='M')
next_periods = monthly_periods + 1
print("\nNext monthly periods:", next_periods)

previous_periods = monthly_periods - 2
print("\nPrevious monthly periods:", previous_periods)
```

б) Преобразуване на честота (`.asfreq()`):

Можете да промените честотата на периодите в `PeriodIndex`. Например, от месечна на дневна (ще върне първия ден от месеца) или обратното.

```
quarterly_periods = pd.PeriodIndex(['2023Q1', '2023Q2'], freq='Q')
daily_periods = quarterly_periods.asfreq('D', 'start')
print("\nQuarterly periods:", quarterly_periods)
print("Daily periods (start):\n", daily_periods)

monthly_from_quarterly = quarterly_periods.asfreq('M', 'end')
print("\nMonthly periods (end):\n", monthly_from_quarterly)
```

Аргументът за позиция ('start' или 'end') определя коя част от по-финия период да се върне.

в) Сравнителни операции:

Можете да сравнявате `PeriodIndex` обекти помежду си.

```
periods1 = pd.PeriodIndex(['2023-01', '2023-02'], freq='M')
```

```
periods2 = pd.PeriodIndex(['2023-02', '2023-03'], freq='M')
print("\nperiods1 < periods2:", periods1 < periods2)
```

г) Извличане на компоненти (.year, .month, .quarter и др.):

Подобно на `DatetimeIndex`, `PeriodIndex` има атрибути за извличане на различни компоненти на периода.

```
print("\nYear:", monthly_periods.year)
print("Month:", monthly_periods.month)
print("Quarter:", monthly_periods.quarter)
```

`PeriodIndex` е полезен, когато естествената единица на вашите данни е времеви период, а не конкретен момент. Примери за това включват финансови отчети (тримесечни, годишни), обобщени данни за месец или седмица и други. Използването на `PeriodIndex` осигурява семантична яснота и улеснява операциите, свързани с тези периоди.

VII. CategoricalIndex: Индекс за категорийни данни.

`CategoricalIndex` е тип индекс в Pandas, който се основава на категорийни данни. Той е оптимизиран за индексиране с повтарящи се стрингови или други обекти, които имат ограничен брой уникални стойности (категории). Използването на `CategoricalIndex` може да доведе до **значително намаляване на използването на памет** и **повишаване на производителността** при определени операции в сравнение с обикновен `Index` от обекти.

1. Създаване на CategoricalIndex:

`CategoricalIndex` може да бъде създаден по няколко начина:

а) Директно от списък или Series с указване на `dtype='category'`:

```
import pandas as pd

colors_list = ['red', 'blue', 'green', 'red', 'blue']
categorical_index_from_list = pd.Index(colors_list, dtype='category')
print("CategoricalIndex from list:\n", categorical_index_from_list)
print("dtype:", categorical_index_from_list.dtype)
```

```
colors_series = pd.Series(['yellow', 'green', 'yellow'],
dtype='category')
categorical_index_from_series = pd.Index(colors_series)
print("\nCategoricalIndex from Series:\n",
categorical_index_from_series)
```

б) Използване на конструктора `pd.CategoricalIndex()`:

Този конструктор позволява по-гъвкаво създаване, включително указване на категории и дали те са подредени.

```
categories = ['A', 'B', 'C']
data = ['A', 'C', 'B', 'A']
categorical_index_explicit = pd.CategoricalIndex(data,
categories=categories)
print("\nExplicit CategoricalIndex (unordered):\n",
categorical_index_explicit)

ordered_categories = ['low', 'medium', 'high']
ordered_data = ['high', 'low', 'medium', 'high']
categorical_index_ordered = pd.CategoricalIndex(ordered_data,
categories=ordered_categories, ordered=True)
print("\nExplicit CategoricalIndex (ordered):\n",
categorical_index_ordered)
```

2. *Предимства и специфични операции на CategoricalIndex:*

- **Намалено използване на памет:** Pandas съхранява само уникалните категории веднъж, а след това използва целочислени кодове за представяне на всяка стойност в индекса. Това е особено ефективно при индекси с много повтарящи се стойности.
- **Повишена производителност:** Някои операции върху `CategoricalIndex` могат да бъдат по-бързи поради вътрешното представяне на данните.
- **Сортиране:** Категорийните индекси могат да бъдат сортирани според лексикографския ред на категориите или според зададен ред, ако категориите са подредени (`ordered=True`).

```
print("\nSorted categorical_index_from_list:",
categorical_index_from_list.sort_values())
```

```
print("Sorted categorical_index_ordered:",
categorical_index_ordered.sort_values())
```

- **Групиране:** Категорийните индекси са ефективни при операции за групиране (`groupby`).
- **Сравнения:** Сравненията се извършват на базата на стойностите (категориите).
- **Достъп до категории (`.categories`):** Атрибутът `.categories` връща обект `pd.Categorical` съдържащ уникалните категории.

```
print("\nCategories of categorical_index_explicit:",
categorical_index_explicit.categories)
```

- **Достъп до кодове (`.codes`):** Атрибутът `.codes` връща NumPy array с целочислените кодове, съответстващи на всяка стойност в индекса.

```
print("\nCodes of categorical_index_explicit:",
categorical_index_explicit.codes)
```

3. Кога да използваме CategoricalIndex?

Използвайте `CategoricalIndex`, когато:

- Индексът съдържа много повтарящи се стрингови или други обекти.
- Редът на категориите е важен (за подреждане или сравнение).
- Искате да оптимизирате използването на памет и евентуално да подобрите производителността.

Въпреки че `CategoricalIndex` предлага много предимства в определени сценарии, е важно да се отбележи, че не всички операции са еднакво бързи или интуитивни както при други типове индекси. Трябва да прецените дали предимствата надвишават потенциалните усложнения за вашия конкретен случай на употреба.

VIII. IntervalIndex: Индекс за интервали (обект Interval).

`IntervalIndex` е специализиран тип индекс в Pandas, който съдържа обекти от тип `Interval`. Тези обекти представляват **затворени, отворени или полуотворени интервали**. `IntervalIndex` е полезен, когато искате да индексирате данни по диапазони от стойности, а не по единични точки.

1. Създаване на IntervalIndex:

`IntervalIndex` може да бъде създаден по няколко начина:

а) Използване на pd.IntervalIndex.from_arrays():

Създава `IntervalIndex` от два масива, указващи левите и десните граници на интервалите. Можете да укажете и коя от границите е затворена ('both', 'left', 'right', 'neither').

```
import pandas as pd

left = [0, 1, 2]
right = [1, 2, 3]
interval_index_from_arrays = pd.IntervalIndex.from_arrays(left, right)
print("IntervalIndex from arrays (both closed):\n",
interval_index_from_arrays)

interval_index_left_closed = pd.IntervalIndex.from_arrays(left, right,
closed='left')
print("\nIntervalIndex from arrays (left closed):\n",
interval_index_left_closed)
```

б) Използване на `pd.IntervalIndex.from_tuples()`:

Създава `IntervalIndex` от списък или масив от tuples, където всеки tuple представлява (лява граница, дясна граница). Параметърът `closed` се използва както при `from_arrays()`.

```
tuples = [(0, 1), (1, 2), (2, 3)]
interval_index_from_tuples = pd.IntervalIndex.from_tuples(tuples)
print("\nIntervalIndex from tuples (both closed):\n",
interval_index_from_tuples)
```

в) Използване на `pd.interval_range()`:

Генерира последователност от интервали с еднаква дължина. Изисква задаване на `start`, `end` или `periods`, както и `freq` (честота на дължината на интервала) и `closed`.

```
interval_range_index = pd.interval_range(start=0, end=3, periods=3)
print("\nIntervalIndex from interval_range (periods):\n",
interval_range_index)
```

```
interval_range_freq = pd.interval_range(start=0, periods=3, freq=1.5)
print("\nIntervalIndex from interval_range (frequency):\n",
interval_range_freq)

interval_range_closed = pd.interval_range(start=0, periods=3, freq=1,
closed='right')
print("\nIntervalIndex from interval_range (right closed):\n",
interval_range_closed)
```

2. Специфични операции и предимства на IntervalIndex:

`IntervalIndex` позволява ефективно извършване на операции, свързани с интервали:

а) Селекция по интервал:

Можете да селектирате данни, чийто индекс (или друга колона) попада в определен интервал.

```
s = pd.Series([10, 20, 30], index=interval_index_from_arrays)
print("\nSeries with IntervalIndex:\n", s)

print("\nValue for interval containing 0.5:", s.loc[0.5])
print("Value for interval containing 1.8:", s.loc[1.8])
```

б) Проверка за съдържание (`.contains()`):

Методът `.contains()` на `IntervalIndex` връща булев масив, показващ дали дадена стойност се съдържа във всеки от интервалите.

```
print("\nDoes interval_index_from_arrays contain 0.5?",
interval_index_from_arrays.contains(0.5))
print("Does interval_index_from_arrays contain [0.5, 1.5, 2.5]?",
interval_index_from_arrays.contains([0.5, 1.5, 2.5]))
```

в) Проверка за припокриване (`.overlaps()`):

Методът `.overlaps()` проверява дали интервалите в `IntervalIndex` се припокриват с даден интервал или друг `IntervalIndex`.

```
other_interval = pd.Interval(1.5, 2.5)
```

```
print("\nDoes interval_index_from_arrays overlap with", other_interval,
"?", interval_index_from_arrays.overlaps(other_interval))

other_interval_index = pd.IntervalIndex.from_tuples([(1.2, 2.2), (2.8,
3.8)])
print("\nDoes interval_index_from_arrays overlap with\n",
other_interval_index, "?\n",
interval_index_from_arrays.overlaps(other_interval_index))
```

г) Достъп до граници (`.left`, `.right`, `.mid`, `.length`):

Атрибутите `.left` и `.right` връщат NumPy масиви с левите и десните граници на интервалите. `.mid` връща средната точка на всеки интервал, а `.length` - дължината.

```
print("\nLeft boundaries:", interval_index_from_arrays.left)
print("Right boundaries:", interval_index_from_arrays.right)
print("Midpoints:", interval_index_from_arrays.mid)
print("Lengths:", interval_index_from_arrays.length)
```

3. `IntervalIndex` е полезен в различни сценарии, включително:

- **Групиране на непрекъснати данни в интервали (binning).**
- **Представяне на времеви интервали.**
- **Анализ на събития, които се случват в определен диапазон.**

Разбирането на `IntervalIndex` ви дава още един мощен инструмент за работа и анализ на данни в Pandas, особено когато естествената структура на данните включва интервали.

IX. MultiIndex: Йерархично (многонивово) индексиране (кратко запознаване с MultiIndex)

`MultiIndex` (наричан още йерархичен индекс или многонивово индексиране) е изключително мощна характеристика на Pandas, която позволява да имате **повече от едно ниво на индексиране** както за редовете, така и за колоните на `Series` или `DataFrame`. Това дава възможност за представяне и работа с данни с по-висока размерност (например, данни, индексирани по година и месец, или по регион и продукт) в двуизмерна структура.

1. Създаване на MultiIndex:

`MultiIndex` може да бъде създаден по няколко начина:

а) От списъци или масиви:

Можете да предоставите списък от масиви (или списъци), където всеки масив представлява ниво на индекса.

```
import pandas as pd

levels = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
multi_index_from_lists = pd.MultiIndex.from_arrays(levels,
names=['first', 'second'])
print("MultiIndex from arrays:\n", multi_index_from_lists)
```

б) От списък от tuples:

```
tuples = [('bar', 'one'), ('bar', 'two'), ('baz', 'one'), ('baz',
'two'),
          ('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux',
'two')]
multi_index_from_tuples = pd.MultiIndex.from_tuples(tuples,
names=['first', 'second'])
print("\nMultiIndex from tuples:\n", multi_index_from_tuples)
```

в) От декартово произведение (`pd.MultiIndex.from_product()`):

Създава `MultiIndex` от декартовото произведение на няколко итерируеми обекта.

```
iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]
multi_index_from_product = pd.MultiIndex.from_product(iterables,
names=['first', 'second'])
print("\nMultiIndex from product:\n", multi_index_from_product)
```

г) Когато създавате `Series` или `DataFrame`, можете да зададете `MultiIndex` като индекс:

```
import numpy as np

data = pd.Series(np.random.randn(8), index=multi_index_from_tuples)
print("\nSeries with MultiIndex:\n", data)

df = pd.DataFrame(np.random.randn(8, 2), index=multi_index_from_tuples,
columns=['A', 'B'])
print("\nDataFrame with MultiIndex:\n", df)
```

2. Предимства и основни операции с MultiIndex (кратко):

а) Представяне на многомерни данни:

`MultiIndex` позволява естествено представяне на данни, които са категоризирани по повече от един фактор.

б) Йерархична селекция:

Можете да селектирате данни на различни нива на индекса, което осигурява голяма гъвкавост при достъпа до подмножества от данни.

```
print("\nSelect 'bar' from the first level:\n", df.loc['bar'])
print("\nSelect ('bar', 'one'):\n", df.loc[('bar', 'one')])
```

в) Частично индексиране:

Можете да използвате слайси и други методи за селекция на определени нива.

г) Преобразуване на структурата на данните (`.unstack()`, `.stack()`):

`MultiIndex` улеснява преместването на нива на индекса към колони и обратно, което е полезно за преструктуриране на данните за анализ или визуализация.

```
unstacked_df = df.unstack(level='second')
print("\nUnstacked DataFrame:\n", unstacked_df)
```

д) Групиране по нива на индекса:

Можете да групирате данни по едно или повече нива на `MultiIndex`.

> **`MultiIndex` е напреднала тема и ще бъде разгледана по-подробно в следваща глава. Това кратко представяне има за цел да ви даде първоначална представа за възможностите на йерархичното индексиране в Pandas.**

X. Казуси от реалния живот: Индексиране на обекти в Pandas

Казус 1: Анализ на продажби по дата

Ситуация:

Имате данни за дневните продажби на различни продукти за изминалата година. Данните са записани в CSV файл, където всяка колона представлява продукт, а всеки ред - дата. Искате да анализирате тенденциите в продажбите във времето за определен продукт.

Данни (пример):

```
Дата,Продукт_А,Продукт_Б,Продукт_В
2024-01-01,10,5,12
2024-01-02,15,8,10
2024-01-03,12,6,15
...
2024-12-31,18,10,11
```

Може да използвате следния генератор:

```
import pandas as pd
import numpy as np

# Създаваме списък с дати за 2024 година
dates = pd.date_range(start='2024-01-01', end='2024-12-31', freq='D')
```

```
# Списък с продукти
products = ['Хляб', 'Мляко', 'Яйца', 'Кафе', 'Захар']

# Генерираме случайни продажби за всеки продукт
np.random.seed(42)  # За повторяемост на резултата
data = {product: np.random.randint(0, 100, size=len(dates)) for product
in products}

# Създаваме DataFrame и задаваме датите като индекс
df_sales = pd.DataFrame(data, index=dates)
df_sales.index.name = 'Дата'

# Записваме във файл
df_sales.to_csv('sales_data_2024.csv', encoding='utf-8-sig')

print(df_sales.head())
```

Задача:

Заредете данните в Pandas DataFrame и използвайте `DatetimeIndex`, за да:

1. Извлечете продажбите за месец март 2024 г. за всички продукти.
2. Изчислете средните дневни продажби за продукт 'Продукт_А' за всяка седмица.

Решение:

```
import pandas as pd

# Зареждане на данните (предполагаме, че файлът се казва
'sales_data.csv')
sales_df = pd.read_csv('sales_data_2024.csv', index_col='Дата')

# Конвертиране на индекса към DatetimeIndex
sales_df.index = pd.to_datetime(sales_df.index)

# 1. Извличане на продажбите за март 2024 г.
march_sales = sales_df.loc['2024-03']
```

```
print("Продажби за март 2024 г.:\n", march_sales)

# 2. Изчисляване на средните седмични продажби за 'Продукт_А'
weekly_avg_product_a = sales_df['Хляб'].resample('W').mean()
print("\nСредни седмични продажби за Продукт_А през 2024 г.:\n",
weekly_avg_product_a)
```

Обяснение:

Използването на `DatetimeIndex` позволява лесно филтриране на данни по времеви периоди (месец) и използване на мощния метод `.resample()` за агрегиране на данни по друга честота (седмична средна).

Казус 2: Анализ на резултати от анкета по възрастови групи

Ситуация:

Имате данни от анкета, където всеки ред представлява отговор на участник, включително неговата възраст (цяло число). Искате да анализирате средното удовлетворение от анкетата за различни възрастови групи.

Данни (пример - част от DataFrame):

```
Възраст  Удовлетвореност
0        25                4
1        30                5
2        22                3
3        25                5
4        40                4
...
```

Задача:

Създайте възрастови групи (например, 18-25, 26-35, 36-45 и т.н.) и използвайте `IntervalIndex`, за да групирате резултатите и да изчислите средното удовлетворение за всяка група.

Решение:

```
import pandas as pd
```

```
# Създаване на примерен DataFrame
data = {'Възраст': [25, 30, 22, 25, 40, 32, 48, 28, 38, 42],
        'Удовлетвореност': [4, 5, 3, 5, 4, 4, 5, 3, 4, 5]}
survey_df = pd.DataFrame(data)

# Определяне на възрастовите граници
age_bins = [18, 25, 35, 45, 55]
age_labels = ['18-25', '26-35', '36-45', '46-55']

# Създаване на IntervalIndex от възрастовите интервали
age_intervals = pd.IntervalIndex.from_breaks(age_bins, closed='right')

# Създаване на нова колона с възрастовите интервали за всеки участник
survey_df['Възрастова група'] = pd.cut(survey_df['Възраст'],
bins=age_bins, labels=age_labels, right=True)

# Групиране по възрастова група и изчисляване на средното удовлетворение
average_satisfaction = survey_df.groupby('Възрастова
група')['Удовлетвореност'].mean()

print("Средно удовлетворение по възрастови групи:\n",
average_satisfaction)
```

Обяснение:

Въпреки че тук не използваме `IntervalIndex` като индекс на самия DataFrame, концепцията за интервали е ключова. Функцията `pd.cut()` използва границите, за да създаде категории, които могат да бъдат представени вътрешно като интервали. След това групираме данните по тези категории, което е пряко свързано с идеята за индексиране по интервали.

Казус 3: Анализ на представянето на ученици по предмети (категорийни данни)

Ситуация:

Имате данни за оценките на ученици по различни предмети. Искате да анализирате разпределението на оценките (например, 'Отличен', 'Много добър', 'Добър' и т.н.) за всеки предмет.

Данни (пример - част от DataFrame):

```
Ученик    Предмет    Оценка
0       А      Математика  Отличен
1       Б      Български   Добър
2       В      Математика  Много добър
3       А      Български   Среден
4       Б      Математика  Добър
...
```

Задача:

Използвайте `CategoricalIndex` за колоната 'Оценка', за да анализирате честотата на всяка категория оценка за всеки предмет.

Решение:

```
import pandas as pd

# Създаване на примерен DataFrame
data = {'Ученик': ['А', 'Б', 'В', 'А', 'Б', 'В', 'А', 'Б'],
        'Предмет': ['Математика', 'Български', 'Математика',
'Български', 'Математика', 'Български', 'Математика', 'Български'],
        'Оценка': ['Отличен', 'Добър', 'Много добър', 'Среден', 'Добър',
'Отличен', 'Много добър', 'Среден']}
grades_df = pd.DataFrame(data)

# Дефиниране на категориите и реда им
grade_categories = ['Среден', 'Добър', 'Много добър', 'Отличен']
grades_df['Оценка'] = pd.Categorical(grades_df['Оценка'],
categories=grade_categories, ordered=True)

# Използване на CategoricalIndex (в случая за колона, но концепцията е
същата)
grades_df_indexed = grades_df.set_index(['Предмет', 'Оценка'])
```

```
# Анализ на честотата на оценките за всеки предмет
grade_counts = grades_df.groupby('Предмет')['Оценка'].value_counts()
print("Честота на оценките по предмети:\n", grade_counts)
```

Обяснение:

Въпреки че в този пример не създаваме директно `CategoricalIndex` като индекс на DataFrame, преобразуването на колоната 'Оценка' в категориален тип (`pd.Categorical`) е тясно свързано с концепцията на `CategoricalIndex`. Това позволява Pandas да работи по-ефективно с тези данни и да запази информация за реда на категориите, което може да е важно при анализ и визуализация.

Тези казуси илюстрират как различните типове индексни обекти (или свързани концепции) в Pandas могат да бъдат приложени за решаване на реални задачи при анализ на данни с различни характеристики (времеви серии, количествени групи, категорийни данни).

Въпроси

1. Каква е основната роля на Index обекта в Pandas Series и DataFrame? Какви са някои от неговите основни характеристики (например, изменяемост, уникалност)?

2. Обяснете разликата между различните основни типове индекси в Pandas:
   - Index (базов клас)
   - Int64Index, UInt64Index, Float64Index
   - DatetimeIndex
   - TimedeltaIndex
   - PeriodIndex
   - CategoricalIndex
   - IntervalIndex

   Кога е препоръчително да използвате всеки от тези типове?

3. Как можем да създадем Index обект директно от Python списък или NumPy array?

4. Какви са основните атрибути на Index обекта (например, .values, .dtype, .shape, .name, .is_unique, .is_monotonic)? Дайте кратко обяснение за всеки.

5. Обяснете предназначението на основните методи на Index обекта (например, .get_loc(), .reindex(), .drop(), .unique(), .sort_values()). Дайте кратки примери за тяхното използване.

6. Какви са специфичните предимства на използването на DatetimeIndex при работа с времеви серии данни? Дайте примери за операции, които са улеснени от този тип индекс (например, селекция по дата, преобразуване на честота).

7. В какви ситуации е полезен TimedeltaIndex? Как можем да извършваме аритметични операции с DatetimeIndex и TimedeltaIndex?

8. Какво представлява PeriodIndex и в какви случаи е подходящ за използване? Как се различава от DatetimeIndex?

9. Обяснете предимствата на използването на CategoricalIndex, особено по отношение на памет и производителност. Как можем да укажем категории и дали те са подредени?

10. В какви сценарии е полезен IntervalIndex? Как можем да създадем IntervalIndex и какви специфични операции поддържа (например, .contains(), .overlaps())?

11. (Кратко въведение) Какво представлява MultiIndex и защо е полезен за представяне на данни с повече от едно ниво на индексиране? Дайте прост пример за създаване на MultiIndex.

Задачи

Използвайте следния примерен DataFrame за решаване на задачите:

```
import pandas as pd
import numpy as np

dates = pd.to_datetime(['2023-01-01', '2023-01-05', '2023-01-10',
'2023-01-15', '2023-01-20'])
durations = pd.to_timedelta(['1 day', '2 days', '0.5 days', '3 days',
'1.5 days'])
categories = pd.Categorical(['A', 'B', 'A', 'C', 'B'], categories=['A',
'B', 'C'], ordered=False)
intervals = pd.IntervalIndex.from_tuples([(0, 1), (1, 3), (2, 4), (3,
5), (4, 6)])

data = pd.DataFrame({
    'Стойност': np.random.randn(5),
    'Продължителност': durations,
    'Категория': categories
}, index=dates)
```

1. Изведете индекса на DataFrame-а и неговия тип.

2. Създайте нов DataFrame, съдържащ само редовете от оригиналния DataFrame, чиито дати са след 2023-01-07.

3. Изведете стойността на реда с етикет '2023-01-15', използвайки `.loc`.

4. Създайте нов `DatetimeIndex`, съдържащ всички дати от индекса на оригиналния DataFrame, но преобразувани към края на месеца.

5. Изчислете средната стойност за всеки ден от седмицата на колоната 'Стойност'. (Hint: Използвайте `.dt.day_name()` на индекса).

6. Създайте `TimedeltaIndex` от колоната 'Продължителност' на DataFrame-а. Изведете общата продължителност (сумата на всички елементи в индекса).

7. Създайте `PeriodIndex` от индекса на оригиналния DataFrame с честота 'седмица' ('W').

8. Изведете уникалните категории от колоната 'Категория'.

9. Създайте `IntervalIndex` от списъка `[(0, 2), (1.5, 3.5)]`. Проверете кои от стойностите `[0.5, 2.5, 4.5]` се съдържат във всеки от интервалите на този нов индекс.

10. За оригиналния DataFrame, създайте нов `MultiIndex`, използвайки оригиналния `DatetimeIndex` и колоната 'Категория' като нива. Преобразувайте DataFrame-а към този нов `MultiIndex`.