Глава VI. Обработка на Липсващи Данни

Липсващите данни са често срещано явление в реалните набори от данни. Те могат да възникнат по различни причини: грешки при въвеждане, липса на събрана информация, проблеми при трансфер на данни и други. Ефективното идентифициране и обработка на тези липси е **критична стъпка** в процеса на анализ на данни, тъй като те могат да повлияят значително на резултатите от нашите изчисления и модели.

В тази глава ще се фокусираме върху начините, по които библиотеката Pandas ни помага да се справим с липсващите данни. Ще разгледаме следните основни аспекти:

- **Как Pandas представя липсващите данни:** Ще се запознаем с различните стойности, които Pandas използва за обозначаване на липсващи данни, като `NaN` (Not a Number), `None`, `pd.NaT` (Not a Time), и експерименталната стойност `pd.NA`.
- **Идентифициране на липсващи данни:** Ще научим как да използваме методите `.isnull()`, `.isna()`, `.notnull()` и `.notna()` за откриване на липсващи стойности в `Series` и `DataFrame`.
- **Преброяване на липсващи данни:** Ще видим как да преброим броя на липсващите стойности в отделни колони или в целия `DataFrame`.
- **Премахване на липсващи данни:** Ще разгледаме метода `.dropna()` и неговите параметри, които ни позволяват да премахваме редове или колони, съдържащи липсващи стойности, с различни критерии.
- **Запълване на липсващи данни:** Ще се научим да използваме метода `.fillna()` за замяна на липсващи стойности с конкретни стойности, с методи за последователно и обратно запълване, както и със статистически мерки като средна, медиана и мода. Ще разгледаме и параметъра `limit` за контролиране на броя на запълнените стойности.
- **Интерполиране на липсващи данни:** Ще се запознаем с метода `.interpolate()` и различните техники за попълване на липсващи стойности въз основа на съседни стойности, което е особено полезно за времеви серии и други последователни данни.

Чрез усвояването на тези техники ще придобиете необходимите умения за почистване и подготовка на вашите данни, което е съществена стъпка за провеждането на надежден и смислен анализ.

I. Какво са липсващите данни? (NaN, None, pd.NaT, pd.NA - експериментално)

В Pandas, липсващите данни се представят по различни начини, в зависимост от типа на данните и източника им. Ето основните стойности, които ще срещнете:

1. `NaN` (Not a Number):

- Това е стандартното представяне на липсващи числови данни (floating-point numbers) в Pandas и NumPy.
- `NaN` е специална стойност от типа `float`.
- Всички аритметични операции с `NaN` обикновено водят до друг `NaN`.
- Сравнението на `NaN` с друга стойност (включително друг `NaN`) винаги връща `False`. Затова не трябва да използвате `==` за проверка за `NaN`; вместо това използвайте `pd.isna()` или `.isnull()`.

```
import pandas as pd
import numpy as np

missing_number = np.nan
print(f"Тип на missing_number: {type(missing_number)}")
print(f"missing_number == missing_number: {missing_number ==
missing_number}") # Резултатът е False!
print(f"pd.isna(missing_number): {pd.isna(missing_number)}")
```

2. None:

- `None` е вградена константа в Python, която често се използва за обозначаване на липса на стойност.
- Pandas обикновено конвертира `None` в `NaN`, когато се използва в числови колони (за да се запази консистентност и да се възползва от NumPy функционалността за работа с `NaN`).
- В колони с тип `object`, `None` може да остане като `None`.

```
missing_none = None
series_with_none = pd.Series([1, 2, None, 4])
print(f"Series с None:\n{series_with_none}")
print(f"Тип на данните в series_with_none: {series_with_none.dtype}") #
Забележете, че None е конвертиран във float (NaN)

series_object = pd.Series(['a', None, 'c'])
```

```
print(f"\nSeries с None (object dtype):\n{series_object}")
print(f"Тип на данните в series_object: {series_object.dtype}")
```

3. `pd.NaT` (Not a Time):

- `pd.NaT` е специална стойност в Pandas, използвана за обозначаване на липсващи стойности в колони с данни от тип datetime или timedelta.
- Аналогично на `NaN` за числови данни, `pd.NaT` се използва за времеви данни.

```
missing_date = pd.NaT
print(f"Тип на missing_date: {type(missing_date)}")

date_series = pd.Series(['2023-01-01', None, '2023-01-03'])
date_series = pd.to_datetime(date_series)
print(f"\nSeries с времеви данни и None:\n{date_series}") # None е
конвертиран в NaT

timedelta_series = pd.Series(['1 day', pd.NaT, '3 days'])
timedelta_series = pd.to_timedelta(timedelta_series)
print(f"\nSeries с времеви разлики и NaT:\n{timedelta_series}")
```

4. pd.NA:

- `pd.NA` е **стабилен и препоръчителен** начин за обозначаване на липсващи стойности в Pandas, въведен с цел осигуряване на по-консистентно поведение при различни типове данни (integer, boolean, string, datetime).
- Той е проектиран да адресира някои от проблемите и неочакваното поведение, което може да възникне при използване на `NaN` за не-числови типове данни. Например, при булеви колони, където `NaN` се третира като `float` и може да доведе до неинтуитивни резултати при логически операции.
- За да използвате `pd.NA`, Pandas автоматично ще го използва за липсващи стойности в колони с новите "nullable" типове данни (като `Int64`, `Boolean`, `String`). Можете също така изрично да създавате `Series` или `DataFrame` с тези типове данни.

```
import pandas as pd

missing_general = pd.NA
print(f"Тип на missing_general: {type(missing_general)}")

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

```
print(f"\nInteger Series с pd.NA:\n{integer_series_with_na}")

# Boolean Series с pd.NA (Използвай типа 'boolean', а не 'bool')
boolean_series_with_na = pd.Series([True, False, pd.NA],
dtype='boolean')
print(f"\nBoolean Series с pd.NA:\n{boolean_series_with_na}")

string_series_with_na = pd.Series(['apple', pd.NA, 'banana'],
dtype='string')
print(f"\nString Series с pd.NA:\n{string_series_with_na}")
```

Липсваща стойност	Тип данни, за които е основно предназначена	Произход/Забележки
`NaN`	Числови (float)	Стандартно за липсващи числови данни в NumPy/Pandas
`None`	Обект (може да бъде конвертиран в NaN или NaT или pd.NA в нови типове)	Вградена Python константа
`pd.NaT`	Datetime и Timedelta	Специфично за липсващи времеви данни в Pandas
`Pd.NA`	Integer (nullable), boolean (nullable), string, (и други нови типове)	Стабилен и препоръчителен начин за представяне на липсващи данни в Pandas 2.0+, осигуряващ по-консистентно поведение при различни типове данни. Автоматично се използва за nullable типове.

II. Идентифициране на липсващи данни (.isnull(), .isna(), .notnull(), .notna())

Pandas предоставя няколко удобни метода за откриване на липсващи стойности в `Series` и `DataFrame`. Тези методи връщат булев масив (или `Series` от булеви стойности), където `True` означава, че стойността на съответната позиция е липсваща, а `False` - че е валидна.

Основните методи за идентифициране на липсващи данни са:

1. .isnull() и .isna():

- Тези два метода вършат **абсолютно същото нещо**. `.isna()` е по-нов псевдоним (alias) на `.isnull()`, въведен за по-голяма консистентност с R и други езици за анализ на данни.
- Прилагат се към `Series` или `DataFrame` и връщат обект със същата структура, но съдържащ булеви стойности.

```
import pandas as pd
import numpy as np

# Series
series_data = pd.Series([1, np.nan, 'hello', None, pd.NaT, pd.NA],
dtype='object')
print("Оригинален Series:\n", series_data)
print("\n.isnull() на Series:\n", series_data.isnull())
print("\n.isna() на Series:\n", series_data.isna()) # Същият резултат

# DataFrame
df_data = pd.DataFrame({
    'A': [1, np.nan, 3],
    'B': ['a', None, 'c'],
    'C': [pd.Timestamp('2023-01-01'), pd.NaT, pd.Timestamp('2023-01-
03')],
    'D': [True, pd.NA, False]
})
print("\nОригинален DataFrame:\n", df_data)
print("\n.isnull() на DataFrame:\n", df_data.isnull())
print("\n.isna() на DataFrame:\n", df_data.isna()) # Същият резултат
```

- абележете как `np.nan`, `None`, и `pd.NaT` се отбелязват като `True`. В колоната 'D' (която Pandas вероятно ще интерпретира като object или boolean с nullable поддръжка), `pd.NA` също се отбелязва като `True`.

2. .notnull() и .notna():

- Тези два метода също вършат **абсолютно същото нещо**. `.notna()` е по-нов псевдоним на `.notnull()`.
- Те връщат булев масив (или `Series` от булеви стойности), където `True` означава, че стойността е **валидна (не е липсваща)**, а `False` - че е липсваща.

- Те са логическата инверсия на `.isnull()` и `.isna()`.

```
print("\n.notnull() на Series:\n", series_data.notnull())
print("\n.notna() на Series:\n", series_data.notna()) # Същият резултат

print("\n.notnull() на DataFrame:\n", df_data.notnull())
print("\n.notna() на DataFrame:\n", df_data.notna()) # Същият резултат
```

- **Използване на булевите масиви за селекция:**

Резултатите от `.isnull()` и `.notnull()` (или техните псевдоними) могат директно да се използват за булево индексиране, за да селектират редове или елементи, които съдържат или не съдържат липсващи данни.

```
# Селектиране на елементите от Series, които са липсващи
missing_values_series = series_data[series_data.isnull()]
print("\nЛипсващи стойности в Series:\n", missing_values_series)

# Селектиране на редовете от DataFrame, които имат поне една липсваща
стойност
rows_with_missing = df_data[df_data.isnull().any(axis=1)]
print("\nРедове в DataFrame с поне една липсваща стойност:\n",
rows_with_missing)

# Селектиране на редовете от DataFrame, които нямат липсващи стойности
rows_without_missing = df_data[df_data.notnull().all(axis=1)]
print("\nРедове в DataFrame без липсващи стойности:\n",
rows_without_missing)
```

Разбирането и използването на тези методи е първата стъпка към ефективната обработка на липсващи данни във вашите Pandas обекти. В следващата тема ще разгледаме как да преброим тези липсващи стойности.

III. Преброяване на липсващи данни.

След като вече знаем как да идентифицираме липсващите стойности с `.isnull()` (или `.isna()`), често е необходимо да преброим колко такива стойности има в нашите `Series` или `DataFrame`. Pandas предоставя няколко начина за това:

1. `.sum()` върху булев масив:

Тъй като `.isnull()` и `.isna()` връщат булеви масиви (където `True` представлява липсваща стойност и `False` - валидна), можем да използваме метода `.sum()` върху тези резултати. В Python (и Pandas), `True` се интерпретира като 1, а `False` като 0 при сумиране.

```
import pandas as pd
import numpy as np

# Series
series_data = pd.Series([1, np.nan, 'hello', None, pd.NaT, pd.NA],
dtype='object')
missing_count_series = series_data.isnull().sum()
print(f"Брой на липсващите стойности в Series: {missing_count_series}")

# DataFrame
df_data = pd.DataFrame({
    'A': [1, np.nan, 3],
    'B': ['a', None, 'c'],
    'C': [pd.Timestamp('2023-01-01'), pd.NaT, pd.Timestamp('2023-01-
03')],
    'D': [True, pd.NA, False]
})

# Брой на липсващите стойности във всяка колона на DataFrame
missing_count_df_column = df_data.isnull().sum()
print("\nБрой на липсващите стойности по колони в DataFrame:\n",
missing_count_df_column)

# Брой на липсващите стойности за всеки ред
missing_count_df_row = df_data.isnull().sum(axis=1)
```

```
print("\nБрой на липсващите стойности по редове в DataFrame:\n",
missing_count_df_row)

# Общ брой на липсващите стойности в целия DataFrame
total_missing_count_df = df_data.isnull().sum().sum()
print(f"\nОбщ брой на липсващите стойности в DataFrame:
{total_missing_count_df}")
```

2. .count():

Методът `.count()` връща броя на **не-липсващите** стойности. Следователно, за да намерим броя на липсващите стойности, можем да извадим резултата от `.count()` от общия брой на елементите (който може да бъде получен с `.size` за `Series` или `.shape[0]` за брой на редовете в `DataFrame`).

```
# Series
total_elements_series = series_data.size
non_missing_count_series = series_data.count()
missing_count_series_alternative = total_elements_series -
non_missing_count_series
print(f"\nАлтернативен брой на липсващите стойности в Series:
{missing_count_series_alternative}")

# DataFrame
total_rows_df = df_data.shape[0]
non_missing_count_df_column = df_data.count()
missing_count_df_column_alternative = total_rows_df -
non_missing_count_df_column
print("\nАлтернативен брой на липсващите стойности по колони в
DataFrame:\n", missing_count_df_column_alternative)
```

3. Избор между `.sum()` и `.count()`:

Обикновено `.isnull().sum()` (или `.isna().sum()`) е по-прямият и често предпочитан начин за преброяване на липсващите стойности, тъй като директно сумира булевите стойности, представляващи липсите. `.count()` изисква допълнителна стъпка за изваждане от общия брой на елементи.

Разбирането как да преброявате липсващите данни е важна стъпка, която помага да оцените степента на липса в набора от данни и да вземете информирани решения за това как да ги обработите (премахване, запълване и т.н.).

4. Още примери .isna().sum()

а) Пример 1: Преброяване на липсващи стойности в Series

```
import pandas as pd
import numpy as np

# Създаваме Series с няколко липсващи стойности
data = pd.Series([10, np.nan, 20, None, 30, pd.NA])
print("Оригинален Series:\n", data)

# Използваме .isna() за да създадем булев Series (True за липсващи,
False за валидни)
is_na = data.isna()
print("\nБулев Series (True където стойността е липсваща):\n", is_na)

# Използваме .sum() върху булевия Series, за да преброим True
стойностите (липсващите)
missing_count = is_na.sum()
print("\nБрой на липсващите стойности в Series:", missing_count)
```

б) Пример 2: Преброяване на липсващи стойности по колони в DataFrame

```
import pandas as pd
import numpy as np

# Създаваме DataFrame с липсващи стойности в различни колони
data = {'A': [1, np.nan, 3, None],
        'B': ['x', None, 'z', 'w'],
        'C': [True, False, pd.NA, True],
        'D': [pd.Timestamp('2023-01-01'), pd.NaT, pd.Timestamp('2023-01-
03'), pd.Timestamp('2023-01-04')]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)
```

```
# Използваме .isna() върху DataFrame-а, което връща DataFrame от булеви стойности
is_na_df = df.isna()
print("\nDataFrame от булеви стойности (True където стойността е липсваща):\n", is_na_df)

# Използваме .sum() върху резултата от .isna(). По подразбиране .sum() се прилага по колони (axis=0)
missing_counts_by_column = is_na_df.sum()
print("\nБрой на липсващите стойности по колони:\n", missing_counts_by_column)
```

в) Пример 3: Преброяване на общия брой на липсващи стойности в DataFrame

```
import pandas as pd
import numpy as np

# Създаваме същия DataFrame като в Пример 2
data = {'A': [1, np.nan, 3, None],
        'B': ['x', None, 'z', 'w'],
        'C': [True, False, pd.NA, True],
        'D': [pd.Timestamp('2023-01-01'), pd.NaT, pd.Timestamp('2023-01-03'), pd.Timestamp('2023-01-04')]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Първо прилагаме .isna(), след това .sum() веднъж, за да получим броя по колони.
missing_counts_by_column = df.isna().sum()

# След това прилагаме .sum() още веднъж върху резултата, за да сумираме броя на липсващите стойности във всички колони
total_missing_count = missing_counts_by_column.sum()
print("\nОбщ брой на липсващите стойности в DataFrame:", total_missing_count)
```

В тези примери `.isna().sum()` е ефективен и кратък начин да получите информация за броя на липсващите стойности в Pandas обекти. Резултатът е `Series`, когато се прилага към `DataFrame`, показващ броя на липсващите стойности за всяка колона, и скаларна стойност, когато се прилага към `Series` или когато се приложи `.sum()` втори път върху резултата за `DataFrame`.

IV. Премахване на липсващи данни (.dropna()) - параметри axis, how, thresh, subset, inplace

Методът `.dropna()` в Pandas се използва за премахване на редове или колони, които съдържат липсващи стойности. Той предлага няколко важни параметъра, които контролират кои редове/колони ще бъдат премахнати:

Синтаксис:

```
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

Нека разгледаме всеки от тези параметри по-подробно:

1. axis:

- Определя по коя ос да се търсят липсващи стойности за премахване.
  - `axis=0` (или `'index'`): Премахва редове, които съдържат липсващи стойности (това е стойността по подразбиране).
  - `axis=1` (или `'columns'`): Премахва колони, които съдържат липсващи стойности.

```
import pandas as pd
import numpy as np

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

```
# Премахване на редове, съдържащи поне една липсваща стойност (по подразбиране)
df_rows_dropped = df.dropna(axis=0)
print("\nDataFrame след премахване на редове с липсващи стойности:\n", df_rows_dropped)

# Премахване на колони, съдържащи поне една липсваща стойност
df_columns_dropped = df.dropna(axis=1)
print("\nDataFrame след премахване на колони с липсващи стойности:\n", df_columns_dropped)
```

➢ Обобщение на `axis`:

За да избегнеш объркване, мисли за `axis` по следния начин:

- **`axis=0`**: Операцията засяга **редовете**.
  - При `sum(axis=0)`, сумираш **по колони**, за да получиш резултат за всеки ред.
  - При `dropna(axis=0)`, премахваш **редове**.
- **`axis=1`**: Операцията засяга **колоните**.
  - При `sum(axis=1)`, сумираш **по редове**, за да получиш резултат за всяка колона.
  - При `dropna(axis=1)`, премахваш **колони**.

2. how:

- Определя дали ред или колона се премахва, когато има *каквито и да е* или *всички* липсващи стойности.
  - `'any'` (по подразбиране): Премахва реда/колоната, ако съдържа **поне една** липсваща стойност.
  - `'all'`: Премахва реда/колоната, само ако **всички** стойности в него са липсващи.

```
data = {'A': [1, np.nan, np.nan],
        'B': [np.nan, 2, np.nan],
        'C': [4, np.nan, np.nan],
        'D': [1, 2, 3]}
df = pd.DataFrame(data)
print("\nОригинален DataFrame:\n", df)

# Премахване на редове, съдържащи поне една липсваща стойност
df_rows_any_na = df.dropna(axis=0, how='any')
```

```
print("\nDataFrame след премахване на редове (how='any'):\n",
df_rows_any_na)

# Премахване на редове, където всички стойности са липсващи
df_rows_all_na = df.dropna(axis=0, how='all')
print("\nDataFrame след премахване на редове (how='all'):\n",
df_rows_all_na)

# Добавяме ред с всички NaN стойности за илюстрация на how='all'
df.loc[3] = [np.nan, np.nan, np.nan, np.nan]
print("\nDataFrame с добавен ред с всички NaN:\n", df)
df_rows_all_na_updated = df.dropna(axis=0, how='all')
print("\nDataFrame след премахване на редове (how='all') - с добавен
ред:\n", df_rows_all_na_updated)
```

3. thresh:

- Определя **минималния брой на не-липсващи стойности**, които трябва да присъстват в ред/колона, за да **не бъде премахнат**.
- Ако броят на не-липсващите стойности е по-малък от `thresh`, редът/колоната се премахва.
- Този параметър е полезен, когато искате да запазите редове/колони с определен брой валидни данни.

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

# Запазване на редове с поне 2 не-липсващи стойности
df_thresh_2 = df.dropna(axis=0, thresh=2)
print("\nDataFrame след премахване на редове (thresh=2):\n",
df_thresh_2)

# Запазване на колони с поне 3 не-липсващи стойности
df_thresh_3_columns = df.dropna(axis=1, thresh=3)
```

```
print("\nDataFrame след премахване на колони (thresh=3):\n",
df_thresh_3_columns)
```

4. subset:

- Определя списък от етикети на редове или колони, които да бъдат **взети предвид** за откриване на липсващи стойности.
- Например, ако `axis=0`, можете да укажете конкретни колони, в които да се търсят липсващи стойности за премахване на редове. Други колони не се вземат предвид.
- Ако `axis=1`, можете да укажете конкретни редове.

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

# Премахване на редове, където има липсваща стойност само в колоните 'A'
или 'B'
df_subset_ab = df.dropna(axis=0, subset=['A', 'B'])
print("\nDataFrame след премахване на редове (subset=['A', 'B']):\n",
df_subset_ab)

# Да илюстрираме с колони (макар и по-рядко използвано):
data_transposed = df.T
print("\nТранспониран DataFrame:\n", data_transposed)
# Премахване на колони (оригинално редове), където има липсваща стойност
в редове с индекс 0 или 2
df_subset_0_2_columns = data_transposed.dropna(axis=1, subset=[0, 2])
print("\nТранспониран DataFrame след премахване на колони (subset=[0,
2]):\n", df_subset_0_2_columns)
```

5. inplace:

- Булева стойност, която определя дали операцията да се извърши **на място** върху оригиналния `DataFrame`.
    - `inplace=False` (по подразбиране): Връща нов `DataFrame` с премахнатите редове/колони, а оригиналният остава непроменен.

- `inplace=True`: Модифицира оригиналния `DataFrame` директно и не връща нищо (`None`). Използвайте тази опция внимателно, тъй като промените са необратими.

```
data = {'A': [1, np.nan, 3],
        'B': [np.nan, 2, np.nan]}
df = pd.DataFrame(data.copy()) # Използваме .copy(), за да не променяме оригиналните данни случайно

print("\nОригинален DataFrame:\n", df)

df.dropna(inplace=True)
print("\nDataFrame след премахване на липсващи стойности (inplace=True):\n", df)
```

Разбирането и правилното използване на параметрите на `.dropna()` е важно за контролиране на процеса на почистване на данни чрез премахване на липсващи стойности. В зависимост от вашите данни и целите на анализа, може да изберете различни стратегии за премахване.

V. Запълване на липсващи данни (.fillna())

Методът `.fillna()` в Pandas се използва за замяна на липсващи стойности (`NaN`, `None`, `pd.NaT`, `pd.NA`) с други стойности. Той предлага голяма гъвкавост при определяне на стойностите за запълване.

Синтаксис:

```
df.fillna(value=None, method=None, axis=0, inplace=False, limit=None, downcast=None)
```

Нека разгледаме основните параметри:

1. value:

- Стойността, с която да се заменят всички липсващи стойности. Може да бъде скаларна стойност (едно число, низ и т.н.), речник или `Series`.
    - **Скаларна стойност:** Запълва всички липсващи стойности с предоставената стойност.

```
import pandas as pd
import numpy as np

data = {'A': [1, np.nan, 3],
        'B': [np.nan, 2, np.nan]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

df_filled_0 = df.fillna(value=0)
print("\nDataFrame след запълване с 0:\n", df_filled_0)
```

- **Речник:** Позволява да се задават различни стойности за запълване на липсващи данни във всяка колона. Ключовете на речника трябва да съвпадат с имената на колоните.

```
fill_values = {'A': 100, 'B': 200}
df_filled_dict = df.fillna(value=fill_values)
print("\nDataFrame след запълване с речник:\n", df_filled_dict)
```

- **Series:** Подобно на речника, но индексът на `Series` трябва да съответства на индекса на `DataFrame` (ако `axis=1`) или на имената на колоните (ако `axis=0`, което е по подразбиране).

```
fill_series = pd.Series([50, 60], index=['A', 'B'])
df_filled_series = df.fillna(value=fill_series)
print("\nDataFrame след запълване със Series:\n", df_filled_series)
```

2. method:

- Определя метода за запълване на липсващи стойности въз основа на съседни валидни стойности.
  - `'ffill'` или `'pad'`: **Forward fill**. Запълва липсващите стойности с последната наблюдавана валидна стойност (по посока на оста).
  - `'bfill'` или `'backfill'`: **Backward fill**. Запълва липсващите стойности със следващата наблюдавана валидна стойност (по посока на оста).

```
data = {'A': [np.nan, 1, np.nan, np.nan, 2, np.nan]}
series = pd.Series(data['A'])
print("\nОригинален Series:\n", series)

filled_ffill = series.fillna(method='ffill')
print("\nSeries след ffill:\n", filled_ffill)
```

```
filled_bfill = series.fillna(method='bfill')
print("\nSeries след bfill:\n", filled_bfill)

df_filled_ffill_col = df.fillna(method='ffill') # По подразбиране axis=0
(по редове)
print("\nDataFrame след ffill (по колони):\n", df_filled_ffill_col)

df_filled_bfill_row = df.fillna(method='bfill', axis=1) # Запълване по
редове
print("\nDataFrame след bfill (по редове):\n", df_filled_bfill_row)
```

3. axis:

- Определя оста, по която да се извърши запълването, когато се използва `method`.
  - `axis=0` (или `'index'`): Запълва по редове (надолу).
  - `axis=1` (или `'columns'`): Запълва по колони (надясно).

```
import pandas as pd
import numpy as np

data = {'A': [np.nan, 1, np.nan],
        'B': [2, np.nan, 3],
        'C': [np.nan, np.nan, 4]}
df = pd.DataFrame(data)
print("Оригинален DataFrame:\n", df)

# Запълване с 0 по колони (axis=0 - по подразбиране)
df_filled_column = df.fillna(value=0, axis=0)
print("\nЗапълване с 0 по колони (axis=0):\n", df_filled_column)

# Запълване с -1 по редове (axis=1)
df_filled_row = df.fillna(value=-1, axis=1)
print("\nЗапълване с -1 по редове (axis=1):\n", df_filled_row)
```

Обяснение:

- Когато `axis=0`, `.fillna()` обхожда всяка колона поотделно и запълва липсващите стойности в тази колона със зададената `value`.
- Когато `axis=1`, `.fillna()` обхожда всеки ред поотделно и запълва липсващите стойности в този ред със зададената `value`.

4. inplace:

- Булева стойност, която определя дали операцията да се извърши на място върху оригиналния `DataFrame` (както при `.dropna()`).

```
import pandas as pd
import numpy as np

data = {'A': [1, np.nan, 3],
        'B': [np.nan, 2, np.nan]}
df = pd.DataFrame(data.copy()) # Използваме .copy(), за да не променяме
оригиналните данни случайно
print("Оригинален DataFrame:\n", df)

# Запълване с 0 на място (inplace=True)
df.fillna(value=0, inplace=True)
print("\nDataFrame след запълване на място (inplace=True):\n", df)

# Създаваме нов DataFrame, за да демонстрираме inplace=False
df_original = pd.DataFrame(data)
print("\nОригинален DataFrame (нов):\n", df_original)

# Запълване с -1, създавайки нов DataFrame (inplace=False - по
подразбиране)
df_filled_new = df_original.fillna(value=-1)
print("\nНов DataFrame след запълване (inplace=False):\n",
df_filled_new)
print("\nОригиналният DataFrame (остава непроменен):\n", df_original)
```

Обяснение:

- Когато `inplace=True`, методът `.fillna()` **модифицира директно** `DataFrame`-а, върху който е приложен, и **не връща нов** `DataFrame`.
- Когато `inplace=False` (което е по подразбиране), `.fillna()` **не променя** оригиналния `DataFrame`, а **връща нов** `DataFrame` с приложеното запълване.

> **!!ВНИМАНИЕ!!! Използвайте `inplace=True` внимателно, тъй като промените са необратими. В много случаи е по-безопасно да оставите `inplace=False` и да присвоите резултата на нова променлива, за да запазите оригиналните данни.**

5. limit:

- Определя максималния брой последователни липсващи стойности, които могат да бъдат запълнени. Ако има повече последователни липси от `limit`, останалите няма да бъдат запълнени с метода `ffill` или `bfill`.

```
import pandas as pd
import numpy as np

data = {'A': [np.nan, np.nan, 1, np.nan, np.nan, np.nan, 2]}
series = pd.Series(data['A'])
print("Оригинален Series:\n", series)

# Запълване напред (ffill) с лимит 1 последователна липсваща стойност
filled_ffill_limit_1 = series.fillna(method='ffill', limit=1)
print("\nЗапълване напред (ffill) с limit=1:\n", filled_ffill_limit_1)

# Запълване назад (bfill) с лимит 2 последователни липсващи стойности
filled_bfill_limit_2 = series.fillna(method='bfill', limit=2)
print("\nЗапълване назад (bfill) с limit=2:\n", filled_bfill_limit_2)

data_df = pd.DataFrame({'B': [np.nan, np.nan, 3, np.nan, np.nan]})
print("\nОригинален DataFrame:\n", data_df)

# Запълване напред по колони с лимит 1
filled_ffill_df_limit_1 = data_df.fillna(method='ffill', limit=1)
print("\nDataFrame след ffill с limit=1:\n", filled_ffill_df_limit_1)
```

Обяснение:

- Параметърът `limit` се използва, когато запълвате липсващи стойности с методи като `'ffill'` или `'bfill'`. Той контролира колко **последователни** липсващи стойности ще бъдат запълнени.
- В първия случай (`ffill`, `limit=1`), само първата последователна `NaN` след валидна стойност се запълва.
- Във втория случай (`bfill`, `limit=2`), до две последователни `NaN` стойности преди валидна стойност се запълват.
- Параметърът `axis` може да се използва заедно с `limit`, за да определи посоката на запълване (по редове или колони).

6. Запълване със статистически мерки (средна, медиана, мода):

- Често е полезно да запълните липсващите числови стойности със статистически мерки от съответната колона.

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

# Запълване с средната стойност на всяка колона
df_filled_mean = df.fillna(df.mean())
print("\nDataFrame след запълване със средна стойност:\n",
df_filled_mean)

# Запълване с медианата на всяка колона
df_filled_median = df.fillna(df.median())
print("\nDataFrame след запълване с медиана:\n", df_filled_median)

# Запълване с най-често срещаните стойности на всяка колона (може да има
повече от една най-често срещаната стойност)
df_filled_mode = df.apply(lambda x: x.fillna(x.mode()[0] if not
x.mode().empty else np.nan), axis=0)
print("\nDataFrame след запълване с най-често срещаната стойност:\n",
df_filled_mode)
```

Методът `.fillna()` е много гъвкав и позволява различни стратегии за справяне с липсващите данни в зависимост от контекста на вашите данни и анализа, който искате да проведете.

VI. Интерполиране на липсващи данни (.interpolate()) - различни методи

***изисква** библиотеката `scipy`

Методът `.interpolate()` в Pandas е мощен инструмент за запълване на липсващи стойности чрез използване на съществуващи стойности в данните. Вместо просто да ги заменяме с константи или статистически мерки, интерполацията се опитва да "приблизи" липсващите стойности въз основа на тенденцията на съседните точки от данни. Това е особено полезно за времеви редове и други последователни данни.

Синтаксис:

```
series.interpolate(method='linear', axis=0, limit=None, inplace=False,
limit_direction='forward', limit_area=None, downcast=None)
df.interpolate(method='linear', axis=0, limit=None, inplace=False,
limit_direction='forward', limit_area=None, downcast=None)
```

Нека разгледаме някои от най-често използваните методи за интерполация:

1. `linear` (по подразбиране):

- Запълва липсващите стойности чрез линейна интерполация. Това означава, че се приема, че стойностите между две съседни точки се променят по права линия.

```
import pandas as pd
import numpy as np

data = pd.Series([1, np.nan, 3, np.nan, 5])
print("Оригинален Series:\n", data)

interpolated_linear = data.interpolate(method='linear')
print("\nИнтерполиран Series (linear):\n", interpolated_linear)
```

2. polynomial:

- Запълва липсващите стойности, използвайки полином от определена степен. Трябва да укажете степента на полинома с параметъра `order`.
- Подходящ е, когато имате криволинейна тенденция в данните.

```
data = pd.Series([1, np.nan, 4, np.nan, 9])
print("\nОригинален Series:\n", data)

interpolated_poly_2 = data.interpolate(method='polynomial', order=2)
print("\nИнтерполиран Series (polynomial, order=2):\n",
interpolated_poly_2)
```

3. spline:

- Запълва липсващите стойности, използвайки сплайн интерполация. Трябва да укажете степента на сплайна с параметъра `order`.
- Сплайните са по-гъвкави от полиномите и могат да осигурят по-гладки криви.

```
data = pd.Series([1, np.nan, 4, np.nan, 9])
print("\nОригинален Series:\n", data)

interpolated_spline_2 = data.interpolate(method='spline', order=2)
print("\nИнтерполиран Series (spline, order=2):\n",
interpolated_spline_2)
```

4. pad / ffill:

Вече разглеждани в `.fillna()`, тези методи могат да се използват и с `.interpolate()`. Те просто запълват с последната валидна стойност.

```
data = pd.Series([1, np.nan, np.nan, 4])
print("\nОригинален Series:\n", data)

interpolated_pad = data.interpolate(method='pad')
print("\nИнтерполиран Series (pad):\n", interpolated_pad)
```

5. nearest:

- Запълва липсващите стойности с най-близката валидна стойност.

```
data = pd.Series([1, np.nan, 3, np.nan, 5])
print("\nОригинален Series:\n", data)

interpolated_nearest = data.interpolate(method='nearest')
print("\nИнтерполиран Series (nearest):\n", interpolated_nearest)
```

6. time:

- Специализиран метод за данни от времеви редове. Той интерполира, като взема предвид времевия индекс на данните. Ако индексът не е времеви, ще работи като `linear`.

```
dates = pd.to_datetime(['2023-01-01', '2023-01-03', '2023-01-06'])
time_data = pd.Series([10, np.nan, 30], index=dates)
print("\nОригинален Time Series:\n", time_data)

interpolated_time = time_data.interpolate(method='time')
print("\nИнтерполиран Time Series (time):\n", interpolated_time)
```

7. Други важни параметри:

- **`axis`:** Определя оста за интерполация (0 за редове, 1 за колони). По подразбиране е 0.
- **`limit`:** Максимален брой последователни липсващи стойности за попълване.
- **`inplace`:** Булева стойност, указваща дали да се модифицира оригиналният обект.
- **`limit_direction`:** Посока за запълване при достигане на лимита ('forward', 'backward', 'both').
- **`limit_area`:** Ограничава прилагането на лимита до определена област ('inside', 'outside').

```
data_df = pd.DataFrame({
    'A': [1, np.nan, 3, np.nan, 5],
    'B': [np.nan, 2, np.nan, 4, np.nan]
})
print("\nОригинален DataFrame:\n", data_df)

interpolated_df_linear = data_df.interpolate(method='linear')
print("\nИнтерполиран DataFrame (linear):\n", interpolated_df_linear)
```

```
interpolated_df_polynomial = data_df.interpolate(method='polynomial',
order=2)
print("\nИнтерполиран DataFrame (polynomial, order=2):\n",
interpolated_df_polynomial)
```

Изборът на подходящ метод за интерполация зависи от естеството на вашите данни и предполагаемата тенденция на липсващите стойности. Визуализацията на данните преди и след интерполация може да помогне за оценка на ефективността на избрания метод.

VII. Казуси от Реалния Живот: Обработка на Липсващи Данни

1. Казус 1: Анализ на данни от сензори с прекъсвания

Ситуация:

Имате данни от сензор, който измерва температура на всеки час. Поради проблеми с връзката, за някои часове данните липсват. Искате да анализирате температурните тенденции във времето.

Данни (пример - част от DataFrame):

```
  Температура
2023-01-01 00:00:00    20.5
2023-01-01 01:00:00     NaN
2023-01-01 02:00:00    21.2
2023-01-01 03:00:00     NaN
2023-01-01 04:00:00    20.8
...
```

Задача:

1. Идентифицирайте броя на липсващите температурни показания.
2. Запълнете липсващите стойности, като използвате линейна интерполация, тъй като се предполага плавна промяна на температурата.
3. След интерполацията, проверете дали все още има липсващи стойности.

Решение:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
dates = pd.to_datetime(['2023-01-01 00:00:00', '2023-01-01 01:00:00',
'2023-01-01 02:00:00',
                        '2023-01-01 03:00:00', '2023-01-01 04:00:00',
'2023-01-01 05:00:00'])
temperatures = pd.Series([20.5, np.nan, 21.2, np.nan, 20.8, np.nan],
index=dates)
df_temp = pd.DataFrame({'Температура': temperatures})

# 1. Идентифициране на броя на липсващите стойности
missing_count = df_temp['Температура'].isna().sum()
print(f"Брой на липсващите температурни показания: {missing_count}")

# 2. Запълване на липсващите стойности с линейна интерполация
df_temp['Температура_Интерполирана'] =
df_temp['Температура'].interpolate(method='linear')
print("\nDataFrame след линейна интерполация:\n", df_temp)

# 3. Проверка за останали липсващи стойности
remaining_missing = df_temp['Температура_Интерполирана'].isna().sum()
print(f"\nБрой на липсващите стойности след интерполация:
{remaining_missing}")
```

Обяснение:

Линейната интерполация е подходяща тук, защото предполагаме, че температурата не се променя рязко между последователните измервания.

2. Казус 2: Попълване на липсващи потребителски данни въз основа на предишни записи

Ситуация:

Имате данни за потребители, включително информация за последния им активен ден. За някои потребители тази информация липсва, но имате записи за предишни активни дни. Искате да попълните липсващата информация, като използвате последния известен активен ден за всеки потребител.

Данни (пример - част от DataFrame):

```
Потребител Последен_активен_ден
0          А            2023-10-20
1          Б                   NaT
2          В            2023-11-15
3          Г                   NaT
4          Д            2023-10-28
```

Задача:

1. Идентифицирайте потребителите с липсваща информация за последния активен ден.
2. Запълнете липсващите дати, като използвате метода 'ffill' (forward fill) в рамките на всяка група потребител (в този прост пример, приемаме, че данните са вече подредени по потребител и време, ако има такива).

Решение:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
data = {'Потребител': ['А', 'Б', 'В', 'Г', 'Д'],
        'Последен_активен_ден': pd.to_datetime(['2023-10-20', np.nan,
'2023-11-15', np.nan, '2023-10-28'])}
df_users = pd.DataFrame(data)

# 1. Идентифициране на потребители с липсващи данни
missing_active_day = df_users[df_users['Последен_активен_ден'].isna()]
```

```
print("Потребители с липсващ последен активен ден:\n",
missing_active_day)

# 2. Запълване на липсващите дати с ffill
df_users['Последен_активен_ден_Запълнен'] =
df_users['Последен_активен_ден'].ffill()
print("\nDataFrame след запълване с ffill:\n", df_users)
```

Обяснение:

Forward fill е подходящ тук, защото предполагаме, че ако няма записан последен активен ден, можем да използваме предходния известен такъв за същия потребител (в по-сложни сценарии може да се наложи групиране по потребител).

3. Казус 3: Обработка на липсващи оценки в учебен процес

Ситуация:

Имате данни за оценки на ученици по различни предмети. Някои ученици не са получили оценки по определени предмети, което е отбелязано като липсваща стойност. Искате да премахнете учениците, които нямат оценки по повече от два предмета.

Данни (пример - част от DataFrame):

```
Ученик Математика Български История Физика
0      А         85.0       92.0     NaN     78.0
1      Б          NaN        NaN    65.0     88.0
2      В         76.0       80.0    91.0     94.0
3      Г          NaN       88.0     NaN      NaN
4      Д         90.0        NaN    82.0     NaN
```

Задача:

1. Пребройте броя на липсващите оценки за всеки ученик.
2. Премахнете редовете (учениците), които имат повече от две липсващи оценки.

Решение:

```
import pandas as pd
import numpy as np

# Създаване на примерен DataFrame
data = {'Ученик': ['А', 'Б', 'В', 'Г', 'Д'],
        'Математика': [85.0, np.nan, 76.0, np.nan, 90.0],
        'Български': [92.0, np.nan, 80.0, 88.0, np.nan],
        'История': [np.nan, 65.0, 91.0, np.nan, 82.0],
        'Физика': [78.0, 88.0, 94.0, np.nan, np.nan]}
df_grades = pd.DataFrame(data).set_index('Ученик')

# 1. Преброяване на броя на липсващите оценки за всеки ученик (по редове)
missing_grades_count = df_grades.isna().sum(axis=1)
print("Брой на липсващите оценки за всеки ученик:\n", missing_grades_count)

# 2. Премахване на учениците с повече от две липсващи оценки
students_to_drop = missing_grades_count[missing_grades_count > 2].index
df_grades_cleaned = df_grades.drop(students_to_drop)
print("\nDataFrame след премахване на ученици с повече от две липсващи оценки:\n", df_grades_cleaned)
```

Обяснение:

Тук използваме `.isna().sum(axis=1)` за да преброим липсващите стойности по редове (за всеки ученик). След това филтрираме учениците, които имат повече от две липсващи оценки, и ги премахваме от DataFrame-а с `.drop()`.

Тези казуси илюстрират как различните техники за обработка на липсващи данни могат да бъдат приложени в реални ситуации, в зависимост от естеството на данните и целите на анализа. Изборът между премахване, запълване или интерполиране зависи от контекста и потенциалното влияние на липсващите стойности върху крайните резултати.

VIII. Въпроси

1. Обяснете различните начини, по които липсващите данни могат да бъдат представени в Pandas (`NaN`, `None`, `pd.NaT`, `pd.NA`). В какви ситуации е по-вероятно да срещнете всяка от тези стойности?
2. Как методите `.isnull()` (или `.isna()`) и `.notnull()` (или `.notna()`) помагат при идентифицирането на липсващи данни? Какъв е типът на резултата, който те връщат? Дайте пример за тяхното използване при селектиране на данни.
3. Опишете как можете да преброите общия брой на липсващи стойности в един `DataFrame` и броя на липсващите стойности за всяка колона. Кой метод е най-често използван за тази цел и защо?
4. Обяснете действието на метода `.dropna()`. Как параметрите `axis` и `how` влияят на резултата от този метод? Дайте примери за ситуации, в които бихте използвали различни стойности за тези параметри.
5. Каква е ролята на параметъра `thresh` в метода `.dropna()`? В какви сценарии може да бъде полезно да се използва този параметър вместо стандартното премахване на редове/колони с липсващи стойности?
6. Опишете различните начини за запълване на липсващи данни с метода `.fillna()`. Кога бихте използвали запълване с конкретна стойност, а кога методи като `'ffill'` или `'bfill'`? Как параметърът `limit` контролира процеса на запълване?
7. Какви са основните принципи на интерполирането на липсващи данни с метода `.interpolate()`? Обяснете действието на поне три различни метода за интерполация (`linear`, `polynomial`, `time`) и дайте пример за сценарий, в който всеки от тях би бил подходящ.
8. В контекста на обработка на липсващи данни, обсъдете разликата между премахване на липсващи стойности и запълването им. Кои фактори трябва да вземете предвид, когато решавате коя стратегия да приложите?
9. Как параметърът `inplace=True` влияе на методите за обработка на липсващи данни като `.dropna()` и `.fillna()`? Какви са предимствата и недостатъците от използването на `inplace=True`?
10. Представете си, че работите с голям набор от данни, където много колони имат значителен брой липсващи стойности. Какви стъпки бихте предприели, за да анализирате и обработите тези липсващи данни по информиран начин?

IX. Задачи

Използвайте следния примерен DataFrame за решаване на задачите:

```
import pandas as pd
import numpy as np

data = {'Име': ['Алиса', 'Боб', 'Чарли', 'Дейвид', 'Ева'],
        'Възраст': [25, np.nan, 22, 35, np.nan],
        'Оценка': [8.5, 9.2, np.nan, 9.5, 8.9],
```

```
        'Град': ['София', None, 'София', 'Варна', 'Бургас']}
df_students = pd.DataFrame(data)
```

1. Идентифицирайте всички липсващи стойности в `df_students` и изведете булев DataFrame.
2. Пребройте броя на липсващите стойности за всяка колона в `df_students`.
3. Премахнете редовете от `df_students`, които съдържат поне една липсваща стойност, и запазете резултата в нов DataFrame `df_students_dropped_any`.
4. Създайте нов DataFrame, който съдържа само учениците, за които има информация както за възрастта, така и за оценката.
5. Премахнете редовете от `df_students`, които имат липсващи стойности само в колоните 'Възраст' и 'Оценка', използвайки параметъра `subset`.
6. Запълнете липсващите стойности в колоната 'Възраст' със средната възраст.
7. Запълнете липсващите стойности в колоната 'Оценка' с предишната валидна оценка (използвайте `ffill`).
8. Интерполирайте липсващите стойности в колоната 'Възраст', използвайки линейна интерполация.
9. Създайте нов DataFrame, който съдържа само редовете от `df_students`, където броят на не-липсващите стойности е поне 3.
10. За колоната 'Град', запълнете липсващите стойности с най-често срещания град (модата).
11. Създайте нов DataFrame, в който липсващите стойности в числовите колони ('Възраст', 'Оценка') са запълнени със средната стойност на съответната колона, а липсващата стойност в колоната 'Град' е запълнена с 'Неизвестен'.
12. След изпълнението на горните задачи, проверете отново за наличие на липсващи стойности в оригиналния `df_students`. Обяснете защо (или защо не) има все още липсващи стойности.
13. Проверете дали след всички операции все още има липсващи стойности в оригиналния `df_students` (ако сте използвали `inplace=False`).