😊 Time Methods

Time Methods in Pandas (Python)

Time methods in Pandas refer to a wide range of functions and properties used to work with date and time data. These are essential for handling time series data, performing time-based indexing, resampling, and extracting specific time components.

Pandas provides a powerful datetime functionality through two key features:

1. **Datetime-like objects** such as Timestamp, DatetimeIndex, Timedelta, and Period.
2. **Datetime properties and methods** accessed using the .dt accessor for Series of datetime-like values.

Key Time-Related Data Types in Pandas

- **datetime64[ns]**: Numpy-based datetime type used in Pandas for dates and timestamps.
- **Timedelta**: Represents the difference between two datetime values.
- **Period**: Represents a time span like a month, quarter, or year.

Important Time Methods and Properties

1. Datetime Conversion

- pd.to_datetime() – Converts a column or Series into datetime format.
- pd.to_timedelta() – Converts strings or numbers to timedelta (duration).

2. Datetime Properties with .dt

Used on Series with datetime-like values to access components:

- .year, .month, .day, .hour, .minute, .second
- .weekday – Returns the day of the week (0=Monday).
- .dayofweek, .day_name(), .month_name() – Readable names for time elements.
- .date – Returns only the date part.
- .time – Returns only the time part.
- .is_month_start, .is_month_end, .is_quarter_start, .is_quarter_end – Boolean flags for special times.
- .daysinmonth – Returns number of days in the month.

3. Date Range Generation

- pd.date_range() – Generates a range of dates.
- pd.timedelta_range() – Generates a range of time deltas.
- pd.period_range() – Generates a range of periods.

4. Resampling and Frequency Conversion

Used mainly in time series data:

- .resample('D'), .resample('M') – Resamples time series data to a new frequency (e.g., daily, monthly).
- .asfreq() – Changes the frequency without aggregation.
- Aggregation methods can be chained (like .resample('M').mean()).

5. Shifting and Lagging

- .shift() – Shifts data by a certain number of periods.
- .tshift() – Shifts the time index without modifying the data (older versions).

6. Timedelta Operations

- Difference between dates results in a Timedelta.
- Supports arithmetic operations (e.g., adding or subtracting days, hours, etc.).

7. Indexing and Filtering by Dates

- You can use datetime-based indexing if the DataFrame has a DatetimeIndex.
- Slicing works naturally with dates (e.g., filtering by year, month).

Common Use Cases

- Time series analysis (stock prices, weather data, sales data).
- Feature extraction from datetime (e.g., day of the week for modeling).
- Resampling data for plotting or aggregation.
- Filtering records within specific time windows.
- Calculating durations or aging of items (like age from birthdate).

Summary

Time methods in Pandas provide comprehensive support for manipulating and analyzing datetime and time series data. They are critical for any domain that deals with time-based patterns, such as finance, operations, marketing, and data engineering.

😄 To begin with the Lab

1. The code demonstrates creating and using Python datetime objects.
2. It sets specific date and time values using datetime(year, month, day, hour, minute, second).
3. my_date is created with only the date, so it defaults to midnight (00:00).
4. my_date_time includes full time information.
5. You can access specific components like the day with .day and the hour with .hour.
6. This is useful for working with dates and times in Python in a structured way.

```
[4]: from datetime import datetime
```

```
[5]: # To illustrate the order of arguments
my_year = 2017
my_month = 1
my_day = 2
my_hour = 13
my_minute = 30
my_second = 15
```

```
[6]: # January 2nd, 2017
my_date = datetime(my_year,my_month,my_day)
```

```
[7]: # Defaults to 0:00
my_date
```

```
[7]: datetime.datetime(2017, 1, 2, 0, 0)
```

```
[8]: # January 2nd, 2017 at 13:30:15
my_date_time = datetime(my_year,my_month,my_day,my_hour,my_minute,my_second)
```

```
[9]: my_date_time
```

```
[9]: datetime.datetime(2017, 1, 2, 13, 30, 15)
```

You can grab any part of the datetime object you want

```
[11]: my_date.day
```

```
[11]: 2
```

```
[12]: my_date_time.hour
```

```
[12]: 13
```

7. Below, we have used Pandas to convert to a date time. The code creates a Pandas Series myser containing date strings and a None value.
8. The Series is initialized with two date formats: 'Nov 3, 2000' and '2000-01-01'.
9. Accessing myser[0] retrieves the first element, 'Nov 3, 2000'. This is useful for working with mixed date formats in a Series.

```
[14]: import pandas as pd
```

```
[15]: myser = pd.Series(['Nov 3, 2000', '2000-01-01', None])
```

```
[16]: myser
```

```
[16]: 0    Nov 3, 2000
      1     2000-01-01
      2           None
      dtype: object
```

```
[17]: myser[0]
```

```
[17]: 'Nov 3, 2000'
```

10. The code demonstrates various ways to handle date parsing in Pandas using pd.to_datetime().

11. Attempting to use format='mixed' with errors='coerce' is not valid; instead, setting exact=False allows for flexible format matching.
12. The euro_date = '10-12-2000' example illustrates potential ambiguity between December 10th and October 12th.
13. By setting dayfirst=True, Pandas interprets the date as December 10th.
14. This approach is useful when dealing with datasets where date formats vary or are ambiguous.

```
[19]: pd.to_datetime(myser, format='mixed', errors='coerce')

[19]: 0   2000-11-03
      1   2000-01-01
      2          NaT
      dtype: datetime64[ns]

[20]: pd.to_datetime(myser, format='mixed')[0]

[20]: Timestamp('2000-11-03 00:00:00')

[21]: obvi_euro_date = '31-12-2000'

[22]: pd.to_datetime(obvi_euro_date, format='%d-%m-%Y')

[22]: Timestamp('2000-12-31 00:00:00')

[23]: # 10th of Dec OR 12th of October?
      # We may need to tell pandas
      euro_date = '10-12-2000'

[24]: pd.to_datetime(euro_date)

[24]: Timestamp('2000-10-12 00:00:00')

[25]: pd.to_datetime(euro_date,dayfirst=True)

[25]: Timestamp('2000-12-10 00:00:00')
```

15. The code uses Pandas' pd.to_datetime() function to convert strings into Timestamp objects.
16. By providing format='%d--%b--%Y', pd.to_datetime() can parse the custom-formatted string '12--Dec--2000' (day, abbreviated month, year) exactly.
17. For the more natural language string '12th of Dec 2000', pd.to_datetime() automatically infers the format and converts it without an explicit format string.
18. This method is essential for turning varied date representations into standardized datetime objects for analysis.

```
[28]: style_date = '12--Dec--2000'

[29]: pd.to_datetime(style_date, format='%d--%b--%Y')

[29]: Timestamp('2000-12-12 00:00:00')

[30]: strange_date = '12th of Dec 2000'

[31]: pd.to_datetime(strange_date)

[31]: Timestamp('2000-12-12 00:00:00')
```

19. This code reads a CSV file into a DataFrame with pd.read_csv().

20. Using sales.iloc[0]['DATE'] retrieves the first entry in the “DATE” column, whose type is initially a plain string.
21. Calling pd.to_datetime(sales['DATE']) parses all those strings into pandas datetime objects (Timestamps), and reassigning it back to sales['DATE'] updates the column.
22. After conversion, sales.iloc[0]['DATE'] returns a Timestamp and its type is now a pandas datetime, enabling time-series operations.

[33]:
```
sales = pd.read_csv('RetailSales_BeerWineLiquor.csv')
```

[34]:
```
sales
```

[34]:

	DATE	MRTSSM4453USN
0	1992-01-01	1509
1	1992-02-01	1541
2	1992-03-01	1597
3	1992-04-01	1675
4	1992-05-01	1822
...	...	...
335	2019-12-01	6630
336	2020-01-01	4388
337	2020-02-01	4533
338	2020-03-01	5562
339	2020-04-01	5207

340 rows × 2 columns

[35]:
```
sales.iloc[0]['DATE']
```

[35]:
```
'1992-01-01'
```

[36]:
```
type(sales.iloc[0]['DATE'])
```

[36]:
```
str
```

[37]:
```
sales['DATE'] = pd.to_datetime(sales['DATE'])
```

[38]:
```
sales
```

[38]:

	DATE	MRTSSM4453USN
0	1992-01-01	1509
1	1992-02-01	1541
2	1992-03-01	1597
3	1992-04-01	1675
4	1992-05-01	1822
...	...	...
335	2019-12-01	6630
336	2020-01-01	4388
337	2020-02-01	4533
338	2020-03-01	5562

```
[39]: sales.iloc[0]['DATE']

[39]: Timestamp('1992-01-01 00:00:00')

[40]: type(sales.iloc[0]['DATE'])

[40]: pandas._libs.tslibs.timestamps.Timestamp
```

23. The code reads the CSV file into a DataFrame with pd.read_csv(), and the parse_dates=[0] argument automatically converts the column at index 0 (assumed to be 'DATE') into datetime objects during the import.
24. After loading the data, sales.iloc[0]['DATE'] accesses the first row's "DATE" value, which is now a pandas Timestamp.
25. This method is efficient for directly parsing date columns during file import, eliminating the need to manually convert the column later.

```
[43]: # Parse Column at Index 0 as Datetime
      sales = pd.read_csv('RetailSales_BeerWineLiquor.csv',parse_dates=[0])

[44]: sales
```

[44]:

	DATE	MRTSSM4453USN
0	1992-01-01	1509
1	1992-02-01	1541
2	1992-03-01	1597
3	1992-04-01	1675
4	1992-05-01	1822
...	...	...
335	2019-12-01	6630
336	2020-01-01	4388
337	2020-02-01	4533
338	2020-03-01	5562
339	2020-04-01	5207

340 rows × 2 columns

```
[45]: type(sales.iloc[0]['DATE'])

[45]: pandas._libs.tslibs.timestamps.Timestamp
```

26. The code accesses the current index of the sales DataFrame using sales.index.
27. Then, sales.set_index("DATE") sets the "DATE" column as the new index of the DataFrame.
28. This operation reorders the data by date, making it easier to perform time-series analysis or filtering by date.
29. The updated DataFrame, now with "DATE" as the index, is stored back into sales.

[48]:
```
# Our index
sales.index
```

[48]:
```
RangeIndex(start=0, stop=340, step=1)
```

[49]:
```
# Reset DATE to index
```

[50]:
```
sales = sales.set_index("DATE")
```

[51]:
```
sales
```

[51]:

DATE	MRTSSM4453USN
1992-01-01	1509
1992-02-01	1541
1992-03-01	1597
1992-04-01	1675
1992-05-01	1822
...	...
2019-12-01	6630
2020-01-01	4388
2020-02-01	4533
2020-03-01	5562

30. The code first resets the index of the sales DataFrame back to the default integer index with sales.reset_index().
31. Then, help(sales['DATE'].dt) displays documentation for the .dt accessor, which allows you to work with date-related attributes and methods in the "DATE" column.
32. sales['DATE'].dt.month extracts the month from each date, while
33. sales['DATE'].dt.is_leap_year checks if the year of each date is a leap year.
34. The .dt accessor is useful for performing operations on datetime columns.

[57]:
```
sales = sales.reset_index()
```

[58]:
```
sales
```

[58]:

	DATE	MRTSSM4453USN
0	1992-01-01	1509
1	1992-02-01	1541
2	1992-03-01	1597
3	1992-04-01	1675
4	1992-05-01	1822
...	...	...
335	2019-12-01	6630
336	2020-01-01	4388
337	2020-02-01	4533
338	2020-03-01	5562
339	2020-04-01	5207

340 rows × 2 columns

[59]:
```
help(sales['DATE'].dt)
```

```
Help on DatetimeProperties in module pandas.core.indexes.accessors object:

class DatetimeProperties(Properties)
 |  DatetimeProperties(data: 'Series', orig) -> 'None'
 |
 |  Accessor object for datetimelike properties of the Series values.
 |
 |  Examples
 |  --------
 |  >>> seconds_series = pd.Series(pd.date_range("2000-01-01", periods=3, freq="s"))
 |  >>> seconds_series
 |  0   2000-01-01 00:00:00
 |  1   2000-01-01 00:00:01
 |  2   2000-01-01 00:00:02
 |  dtype: datetime64[ns]
 |  >>> seconds_series.dt.second
 |  0    0
 |  1    1
```

[60]:
```
sales['DATE'].dt.month
```

[60]:
```
0       1
1       2
2       3
3       4
4       5
       ..
335    12
336     1
337     2
338     3
339     4
Name: DATE, Length: 340, dtype: int32
```

[61]:
```
sales['DATE'].dt.is_leap_year
```

[61]:
```
0       True
1       True
2       True
3       True
4       True
       ...
335    False
336     True
337     True
338     True
339     True
Name: DATE, Length: 340, dtype: bool
```