Intermediate Data Science

Time Series Data

Joanna Bieri DATA201

Time Series Data

Time series are a form of data that are common in many fields: Economics, Finance, Ecology, Neuroscience, Physics, and Applied Math. Any data that is recorded at many points over time, with a certain frequency of observations, is time series data. Time plays an important role in the data - maybe you want to know how observations change over time.

Time Series Data

- *Fixed Frequency* observations are recorded at a fixed interval. The intervals are regular and time steps consistent.
- *Irregular* observations do not have a fixed interval, although time is recorded and could be important to your analysis.

Time Series Data

There are many ways that you might mark time series data:

- **Timestamps** time marked by recording specific instants in time.
- **Fixed Periods** time marked by recording the month or the year.
- **Intervals of Time** mark both a start and end timestamp.
- **Experimental or Elapsed Time** time as recorded relative to a start time.

We will mostly explore timestamps, since the other types can usually be converted into timestamps.

Date and Time Tools

We will start by exploring tools that are available for interacting with dates and times. We will look at the datetime package.

```
from datetime import datetime
now = datetime.now()
print(now)
```

```
2025-09-30 15:46:23.953545
```

We see the format is year, month, day, hour, minute, second, microsecond. We can access items from this datetime object. Try now. and press the tab button!

Date and Time Tools

You can add or subtract a `timedelta` to shift a datetime object or create a series of datetimes.

```
from datetime import timedelta
# timedelta(days=0, seconds=0,
            microseconds=0, milliseconds=0,
            minutes=0, hours=0, weeks=0)

# You can quickly generate lists of time data
delta = timedelta(days=1)
days_of_year = [datetime(2025,1,1)]

for i in range(30):
    days_of_year.append(days_of_year[-1]+delta)
```

Converting Strings

You can go back and forth between datetime objects and strings.

Convert to string

```
dt = datetime(2000,1,1)
# General String
str(dt)
# Formated String
dt.strftime("%m-%d-%Y")
```

Convert to datetime

```
str_date = '1-1-2000'
datetime.strptime(str_date, "%m-%d-%Y")
```

Time Series Basics - Weather data

```
Warning: Looks like you're using an outdated `kagglehub` versi
Path to dataset files: /home/bellajagu/.cache/kagglehub/datase
['Weather_dataset.csv', 'Weather_Data_1980_2024(hourly).csv']
```

	time	temperature	relative_humidity	dew_point	preci
0	1980-01-01T00:00	12.7	83	10.0	0.0
1	1980-01-01T01:00	12.9	82	9.9	0.0

Notice that the objects in the time column are strings

```
'1980-01-01T00:00'
```

Time Series Basics - Weather data

The lecture notes show a longer method to change date times with a function but we could also:

```
df = pd.read_csv(path+'/'+'Weather_Data_1980_2024(hourly).csv'
                 parse_dates=True,index_col=0)
```

Time Series Basics

Timestamps are a Pandas/Numpy object, basically what happens is when you send a datetime into pandas it interprets it as a Timestamp.

Note: pandas.Timestamp stores extra data: nanosecond level precision and frequency information. So it is always safe to convert from datetime to Timestamp, but you might lose some information if you go the other direction.

Indexing, Selection, and Subsetting

How do you select data that is in Timestamp format?

```
df.loc['1980-01-01 00:00:00']
```

Indexing, Selection, and Subsetting

Pandas will interpret our results if we leave out hours and minutes

```
df.loc['1980-01-01']
```

Indexing, Selection, and Subsetting

It will also interpret strings, even if they are not in the exact order

```
df.loc['01-01-1980']
```

Indexing, Selection, and Subsetting

You can slice the data by sending in date ranges

Since time data is chronological you can even used dates not in the range

```
df.loc['1908':'1981']
```

Date Ranges, Frequencies, and Shifting

Often when dealing with dates you need to do some work to make them regular relative to a fixed frequency, even if that means introducing missing variables into your data set.

You can check the frequency using the pandas function `pd.infer_freq`

Generating Date Ranges

If you have some data that you know has a particular date range, you can generate date range data using pandas without typing in the dates individually. To do this you need to choose a frequency.

Generating Date Ranges

Here is a very short list of common ones:

Alias	Description
`B`	Business day frequency
`D`	Calendar day
`W` / `W-MON`	Weekly (optionally anchored)
`QE`	Quarter end
`QE-JAN`	Quarter end (ending in January)
`QS`	Quarter start
`h`	Hourly
`bh`	Business hour
`min`	Minutely

Generating Date Ranges

- By default the frequency is Day

```
dates = pd.date_range('1-1-2025','1-1-2026')
```

- Months

```
dates = pd.date_range('1-1-2025','1-1-2026',
                      freq='ME')
```

- Quarters

```
dates = pd.date_range('1-1-2025','1-1-2026',
                        freq='QE-JAN')
```

- Generate a certain number starting a a date

```
dates = pd.date_range(start='1-1-2025',
                      periods=10, freq='D')
```

Frequencies and Offsets

You can use fancier frequencies to get more refined offsets for your dates

- Here 4h is every 4 hours

```
dates = pd.date_range(start='1-1-2025',
                      periods=10, freq='4h')
```

- we use the WOM = week of month to get every 3rd Friday

```
dates = pd.date_range(start='1-1-2025',
                      periods=10, freq='WOM-3FRI')
```

Shifting Date Data

Sometimes you want to move data backward or forward in time. Pandas has a `.shift` method for doing this.

```
mwf_days.shift(1,freq='d')
```

Time Zones

One of the hardest things to deal with in time series data is often time zones. Users who enter data using different time zones can really confuse the ordering of a data set. We typically will reference time zones with respect to UTC or coordinated universal time. Then time zones are referenced from the UTC, so for example Redlands is UTC - 7 during Daylight Saving Time (PDT) and UTC - 8 during Standard Time (PST).

Time Zones

Another thing to beware of is that historically, the UTC offsets and things like Daylight Savings have been changed. So be very careful when comparing times across historical data. If you run into issues with time zones for your data you should explore the pytz package. This package has access to a database that contains world time zone information.

The book has a chapter on dealing with Time Zone data starting on pate 374. I am going to skip it here so we don't get too into the weeds!

Periods and Period Arithmetic

A `Period` in pandas represents a **span of time** (e.g., a day, a month, a quarter), not just a single timestamp. It is useful for period-based time series data where the concept of a **time interval** is more relevant than an exact point in time.

Periods and Period Arithmetic

```
import pandas as pd

pd.Period('2025-09', freq='M')   # Represents September 2025
pd.Period('2025Q3', freq='Q')    # Represents Q3 of 2025
pd.Period('2025-09-30', freq='D')  # Represents the full day o
```

Why would we use Periods?

1. Time Logic - Grouping

`Period` makes it easy to **group and summarize** time series data by months, quarters, etc.

This avoids confusion from grouping by exact timestamps and ensures consistent aggregation.

2. Avoids Timestamp Precision Errors

Timestamps are overly precise (down to nanoseconds), which may be unnecessary or even problematic for grouped data like "September 2025". `Period` avoids that overprecision.

3. Time Arithmetic at the Period Level

You can do intuitive arithmetic with `Period`:

Periods and Period Arithmetic

We can use the period to group our data. Let's find the average monthly temperature in our weather data:

```
df['period'] = df['time'].dt.to_period('M')
df['temperature'].groupby(by=df['period']).mean()
```

Quarterly Data

Financial data is often reported quarterly or relative to a fiscal year end. Using periods can help us get dates depending on the quarter and fiscal year. Here is a quick example:

```
p = pd.Period('2025Q4', freq='Q-JAN')
# Look to see the start and ends dates of this quarter
print(p.asfreq('D', how='start'))
print(p.asfreq('D', how='end'))
```

Resampling and Frequency Conversion

But what if you had data that was missing some measurements or data that contained too many measurements? In these cases you want to use resampling. Resampling is the process of changing the frequency of your time series data. It lets you:

Resampling and Frequency Conversion

- Downsample: Convert high-frequency data (e.g., minute-level) to lower frequency (e.g., daily), usually by aggregating.
- Upsample: Convert lower-frequency data (e.g., daily) to higher frequency (e.g., hourly), often by filling or interpolating values.

Resampling and Frequency Conversion

Pandas has the `.resample` method to help us with this process. It is similar to `.groupby()` in that it requires a way to aggregate the data before you get back a data frame.

NOTE - your data frame must have a a datetime-like index such as:

- DatetimeIndex
- PeriodIndex
- TimedeltaIndex

for resample to work. It always uses the index values

Downsampling

Downsampling is converting from higher frequency to lower frequency.

```
# Let's downsample to get data only yearly
sample = df[cols].resample('YE')
sample
```

Downsampling

At this point pandas is ready to return the information but needs to know how to combine the groups. In this case let's return the average values.

```
sample.mean()
```

Downsampling

There are lots of ways to play with the data using sampling!!

Open-high-low-close

Open-high-low-close resampling

In finance, often we want to compute four important values:

Term	Meaning
Open	First price in the time window
High	Highest price in the time window
Low	Lowest price in the time window
Close	Last price in the time window

Open-high-low-close

Pandas has a function for this called `.ohlc()`. Let's see what this does with our temperature data on a daily frequency.

```
sample = df['temperature'].resample('D')
sample.ohlc()
```

Upsampling

When we did a downsample we had to aggregate the data so that many rows are grouped into one. Upsampling is converting from lower frequency to higher frequency. When we upsample we have to add new rows and decide how we might fill them in.

Upsampling

Method	Code Example	Use Case
Forward fill	`df.resample('H').ffill()`	Stock prices, step functions
Backward fill	`df.resample('H').bfill()`	Data where future value applies earlier
Interpolate	`df.resample('H').interpolate()`	Continuous numeric data
As-is (NaN)	`df.resample('H').asfreq()`	When you want to leave gaps

Upsampling

Let's start with our weather data, but pretend like we only know the values monthly:

Upsampling

We are pretending that we don't know the original data - these are our only observations! Now what if we wanted to expand this data to daily observations?

- as frequency

  sample = df_example.resample('D') sample.asfreq().head(15)

Upsampling

- forward fill

```
sample.ffill().head(15)
```

Upsampling

- interpolate

```
sample.interpolate().head(15)
```

Moving Window Functions

Next we will consider functions that are evaluated over a sliding window to time or evaluated with exponentially decaying weights. Our book calls these "moving window functions"

When analyzing time series data, we often want to extract meaningful trends without being overwhelmed by short-term noise. Two powerful techniques for this are **sliding window functions** and **exponentially weighted functions**.

Sliding Window Functions (Rolling Windows)

These compute metrics (like `mean`, `sum`, `std`, etc.) over a fixed-size window that "slides" across the time series.

Use cases: - 7-day moving average of temperature - 30-day rolling volatility of returns - Smoothing daily sales data

Why it's useful: - Helps observe short-term trends over time - Reduces the influence of sudden spikes or dips

Sliding Window Functions (Rolling Windows)

For this we will use the `.rolling()` method. Instead of looking at the weather data here we will read in the stock data from before, this data is more illustrative of the method and follows the book.

Load the stocks data:

Sliding Window Functions (Rolling Windows)

Now we will resample. This data looks to be daily frequency, but maybe we actually want to have the information based on the business day frequency.

Frequency	Code	Includes
Daily	`'D'`	All calendar days (Mon–Sun)
Business Daily	`'B'`	Only weekdays (Mon–Fri) — excludes weekends

Sliding Window Functions (Rolling Windows)

Sliding Window Functions (Rolling Windows)

We notice how financial data has lots of ups and downs and if we zoom into the data the change from one day to the next actually tells us very little. This is why we often use rolling functions to understand the data. Here we will calculate a rolling average and plot it with the data.

Sliding Window Functions (Rolling Windows)

Here we will calculate a mean over a 250 day window. You have to choose what window to use! As the window slides across the timeseries the data on the right becomes part of the average and the data from the left leaves the average.

```
rolling_ave = df_resample[my_col].rolling(250).mean()
```

Sliding Window Functions (Rolling Windows)

Sliding Window Functions (Rolling Windows)

By default `.rolling()` cannot deal with NaN values. However there is an optional flag 'min_periods=' which lets you specify the minumum number of non-nan values that can be used to calculate. This way NaNs can be dropped.

Sliding Window Functions (Expanding Windows)

Sometimes instead of a rolling window, you want an expanding window. For example we could calculate the average as we expand our data over time. In this case the right edge of the window expands to include more data in the average.

```
expand_ave = df_resample[my_col].expanding().mean()
```

Sliding Window Functions (Expanding Windows)

Exponentially Weighted Functions (EWM)

These compute statistics using exponentially decaying weights, giving more importance to more recent data points.

Use cases: - Real-time trend tracking (e.g., financial indicators) - Adaptive smoothing for changing behavior - Faster reaction to recent changes compared to rolling averages

Why it's useful: - More responsive to recent data - Does not require a fixed window size - Better suited for evolving or rapidly changing data

Exponentially Weighted Functions (EWM)

In Pandas we will use the `ewm()` exponentially weighted moving function. Here we thing of applying a decay factor based on the span which determines how much memory the ewm has. Often we combine exponential weighting with moving average so that the most recent data has more of an impact on the outcome.

```
rolling_ave = df_resample[my_col].rolling(250).mean()
ewm_rolling_ave = rolling_ave.ewm(span=30).mean()
```

Exponentially Weighted Functions (EWM)

Binary Moving Window Functions

When you are calculating things that need more than one set of timeseries data, for example correlation or covariance, you need to send in additional data into the functions. Here we will plot the correlation between the percent change in the stock price of 'AAPL' compared to the percent change in the benchmark index 'SPX'

Binary Moving Window Functions

```
# Get the percent changes
pcng_spx = df_resample['SPX'].pct_change()
pcng_aapl = df_resample['AAPL'].pct_change()

# Now calculate the correlation
corr = pcng_aapl.rolling(250).corr(pcng_spx)
```

Binary Moving Window Functions