CPE21 Reviewer Prelim Compiled

March 15, 2025

1 Introduction to Time Series with Pandas

Most of our data will have a datatime index, so let's learn how to deal with this sort of data with pandas!

1.1 Python Datetime Review

In the course introduction section we discussed Python datetime objects.

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

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

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

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

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

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

[ ]:
```
my_date_time
```

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

You can grab any part of the datetime object you want

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

[ ]:
```
2
```

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

[ ]:
```
13
```

1.2 NumPy Datetime Arrays

We mentioned that NumPy handles dates more efficiently than Python's datetime format. The NumPy data type is called datetime64 to distinguish it from Python's datetime.

In this section we'll show how to set up datetime arrays in NumPy. These will become useful later on in the course. For more info on NumPy visit https://docs.scipy.org/doc/numpy-1.15.4/reference/arrays.datetime.html

[ ]:
```
import numpy as np
```

[ ]:
```
# CREATE AN ARRAY FROM THREE DATES
np.array(['2016-03-15', '2017-05-24', '2018-08-09'], dtype='datetime64')
```

[ ]:
```
array(['2016-03-15', '2017-05-24', '2018-08-09'], dtype='datetime64[D]')
```

NOTE: We see the dtype listed as 'datetime64[D]'. This tells us that NumPy applied a day-level date precision. If we want we can pass in a different measurement, such as [h] for hour or [Y] for year.

[ ]:
```
np.array(['2016-03-15', '2017-05-24', '2018-08-09'], dtype='datetime64[h]')
```

[ ]:
```
array(['2016-03-15T00', '2017-05-24T00', '2018-08-09T00'],
      dtype='datetime64[h]')
```

[ ]:
```
np.array(['2016-03-15', '2017-05-24', '2018-08-09'], dtype='datetime64[Y]')
```

[ ]:
```
array(['2016', '2017', '2018'], dtype='datetime64[Y]')
```

1.3 NumPy Date Ranges

Just as np.arange(start,stop,step) can be used to produce an array of evenly-spaced integers, we can pass a dtype argument to obtain an array of dates. Remember that the stop date is exclusive.

[ ]:
```
# AN ARRAY OF DATES FROM 6/1/18 TO 6/22/18 SPACED ONE WEEK APART
np.arange('2018-06-01', '2018-06-23', 7, dtype='datetime64[D]')
```

[ ]:
```
array(['2018-06-01', '2018-06-08', '2018-06-15', '2018-06-22'],
      dtype='datetime64[D]')
```

By omitting the step value we can obtain every value based on the precision.

[ ]:
```
# AN ARRAY OF DATES FOR EVERY YEAR FROM 1968 TO 1975
np.arange('1968', '1976', dtype='datetime64[Y]')
```

```
[ ]: array(['1968', '1969', '1970', '1971', '1972', '1973', '1974', '1975'],
           dtype='datetime64[Y]')
```

1.4 Pandas Datetime Index

We'll usually deal with time series as a datetime index when working with pandas dataframes. Fortunately pandas has a lot of functions and methods to work with time series! For more on the pandas DatetimeIndex visit https://pandas.pydata.org/pandas-docs/stable/timeseries.html

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

The simplest way to build a DatetimeIndex is with the pd.date_range() method:

```
[ ]: # THE WEEK OF JULY 8TH, 2018
     idx = pd.date_range('7/8/2018', periods=7, freq='D')
     idx
```

```
[ ]: DatetimeIndex(['2018-07-08', '2018-07-09', '2018-07-10', '2018-07-11',
                  '2018-07-12', '2018-07-13', '2018-07-14'],
                 dtype='datetime64[ns]', freq='D')
```

DatetimeIndex Frequencies: When we used pd.date_range() above, we had to pass in a frequency parameter 'D'. This created a series of 7 dates spaced one day apart. We'll cover this topic in depth in upcoming lectures, but for now, a list of time series offset aliases like 'D' can be found here.

Another way is to convert incoming text with the pd.to_datetime() method:

```
[ ]: idx = pd.to_datetime(['Jan 01, 2018','1/2/18','03-Jan-2018',None])
     idx
```

```
[ ]: DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', 'NaT'],
     dtype='datetime64[ns]', freq=None)
```

A third way is to pass a list or an array of datetime objects into the pd.DatetimeIndex() method:

```
[ ]: # Create a NumPy datetime array
     some_dates = np.array(['2016-03-15', '2017-05-24', '2018-08-09'],␣
      ↪dtype='datetime64[D]')
     some_dates
```

```
[ ]: array(['2016-03-15', '2017-05-24', '2018-08-09'], dtype='datetime64[D]')
```

```
[ ]: # Convert to an index
     idx = pd.DatetimeIndex(some_dates)
     idx
```

```
[ ]: DatetimeIndex(['2016-03-15', '2017-05-24', '2018-08-09'],
     dtype='datetime64[ns]', freq=None)
```

Notice that even though the dates came into pandas with a day-level precision, pandas assigns a nanosecond-level precision with the expectation that we might want this later on.

To set an existing column as the index, use .set_index() >df.set_index('Date',inplace=True)

1.5 Pandas Datetime Analysis

[ ]:
```
# Create some random data
data = np.random.randn(3,2)
cols = ['A','B']
print(data)
```

```
[[-1.64971705  1.07943894]
 [ 0.4587492  -0.04201784]
 [-1.2793774  -1.85383771]]
```

[ ]:
```
# Create a DataFrame with our random data, our date index, and our columns
df = pd.DataFrame(data,idx,cols)
df
```

[ ]:
```
                   A         B
2016-03-15 -1.649717  1.079439
2017-05-24  0.458749 -0.042018
2018-08-09 -1.279377 -1.853838
```

Now we can perform a typical analysis of our DataFrame

[ ]:
```
df.index
```

[ ]:
```
DatetimeIndex(['2016-03-15', '2017-05-24', '2018-08-09'],
dtype='datetime64[ns]', freq=None)
```

[ ]:
```
# Latest Date Value
df.index.max()
```

[ ]:
```
Timestamp('2018-08-09 00:00:00')
```

[ ]:
```
# Latest Date Index Location
df.index.argmax()
```

[ ]:
```
2
```

[ ]:
```
# Earliest Date Value
df.index.min()
```

[ ]:
```
Timestamp('2016-03-15 00:00:00')
```

[ ]:
```
# Earliest Date Index Location
df.index.argmin()
```

[ ]:
```
0
```

NOTE: Normally we would find index locations by running .idxmin() or .idxmax() on df['column'] since .argmin() and .argmax() have been deprecated. However, we still use .argmin() and .argmax() on the index itself.

2 Time Resampling

Let's learn how to sample time series data! This will be useful later on in the course!

[ ]:
```
import pandas as pd
%matplotlib inline
```

2.1 Import the data

For this exercise we'll look at Starbucks stock data from 2015 to 2018 which includes daily closing prices and trading volumes.

[ ]:
```
df = pd.read_csv('../Data/starbucks.csv', index_col='Date', parse_dates=True) #␣
 ↪modify to fit your needs (location of your data)
```

Note: the above code is a faster way of doing the following:

[ ]:
```
df.head()
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
2015-01-07  37.8848   9732554
2015-01-08  38.4961  13170548
```

2.2 resample()

A common operation with time series data is resampling based on the time series index. Let's see how to use the resample() method. [reference]

[ ]:
```
# Our index
df.index
```

[ ]:
```
DatetimeIndex(['2015-01-02', '2015-01-05', '2015-01-06', '2015-01-07',
               '2015-01-08', '2015-01-09', '2015-01-12', '2015-01-13',
               '2015-01-14', '2015-01-15',
               …
               '2018-12-17', '2018-12-18', '2018-12-19', '2018-12-20',
               '2018-12-21', '2018-12-24', '2018-12-26', '2018-12-27',
               '2018-12-28', '2018-12-31'],
              dtype='datetime64[ns]', name='Date', length=1006, freq=None)
```

When calling `.resample()` you first need to pass in a **rule** parameter, then you need to call some sort of aggregation function.

The **rule** parameter describes the frequency with which to apply the aggregation function (daily, monthly, yearly, etc.) It is passed in using an "offset alias" - refer to the table below. [reference]

The aggregation function is needed because, due to resampling, we need some sort of mathematical rule to join the rows (mean, sum, count, etc.)

2.2.1 TIME SERIES OFFSET ALIASES

ALIAS	DESCRIPTION
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
SM	semi-month end frequency (15th and end of month)
BM	business month end frequency
CBM	custom business month end frequency
MS	month start frequency
SMS	semi-month start frequency (1st and 15th)
BMS	business month start frequency
CBMS	custom business month start frequency
Q	quarter end frequency
	intentionally left blank
BQ	business quarter end frequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
BH	business hour frequency
H	hourly frequency
T, min	minutely frequency
S	secondly frequency
L, ms	milliseconds
U, us	microseconds
N	nanoseconds

[ ]:
```
# Yearly Means
df.resample(rule='A').mean()
```

[ ]:
```
                Close        Volume
Date
2015-12-31  50.078100  8.649190e+06
2016-12-31  53.891732  9.300633e+06
```

```
2017-12-31   55.457310   9.296078e+06
2018-12-31   56.870005   1.122883e+07
```

Resampling rule 'A' takes all of the data points in a given year, applies the aggregation function (in this case we calculate the mean), and reports the result as the last day of that year.

2.2.2 Custom Resampling Functions

We're not limited to pandas built-in summary functions (min/max/mean etc.). We can define our own function:

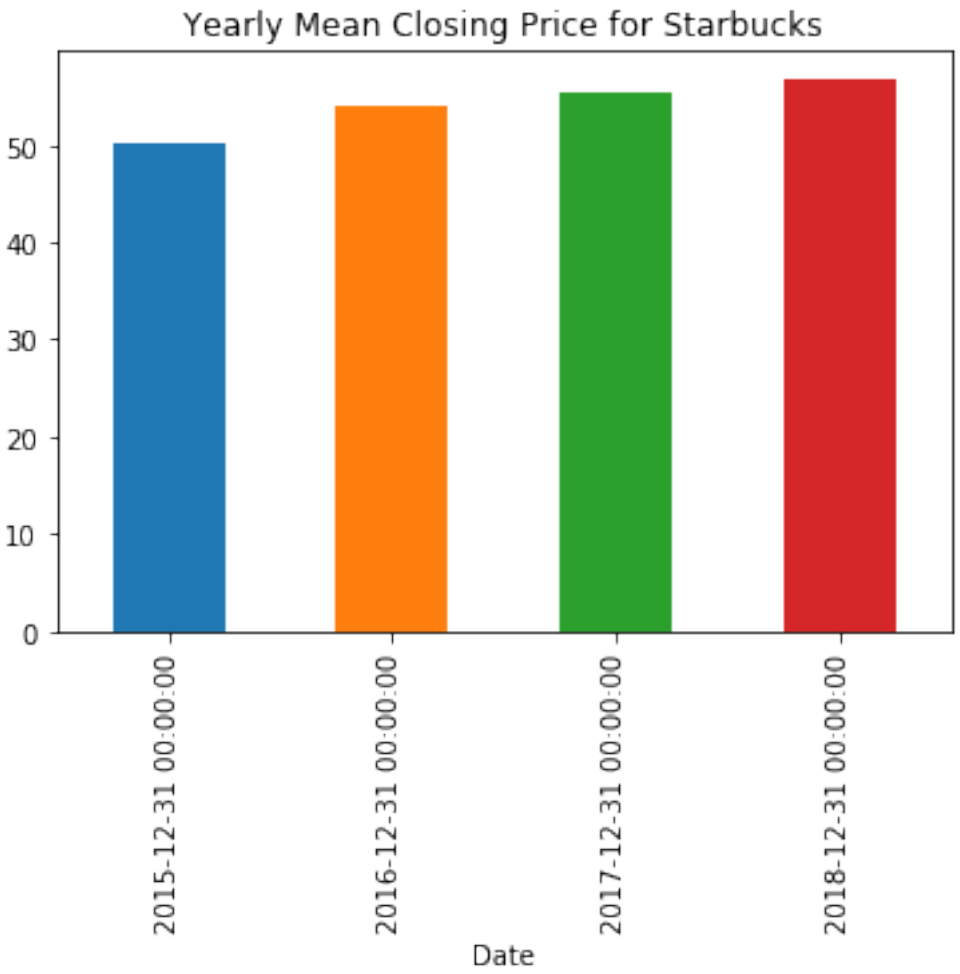
[ ]:
```
def first_day(entry):
    """
    Returns the first instance of the period, regardless of sampling rate.
    """
    if len(entry):  # handles the case of missing data
        return entry[0]
```

[ ]:
```
df.resample(rule='A').apply(first_day)
```

[ ]:
```
              Close    Volume
Date
2015-12-31  38.0061   6906098
2016-12-31  55.0780  13521544
2017-12-31  53.1100   7809307
2018-12-31  56.3243   7215978
```

2.2.3 Plotting

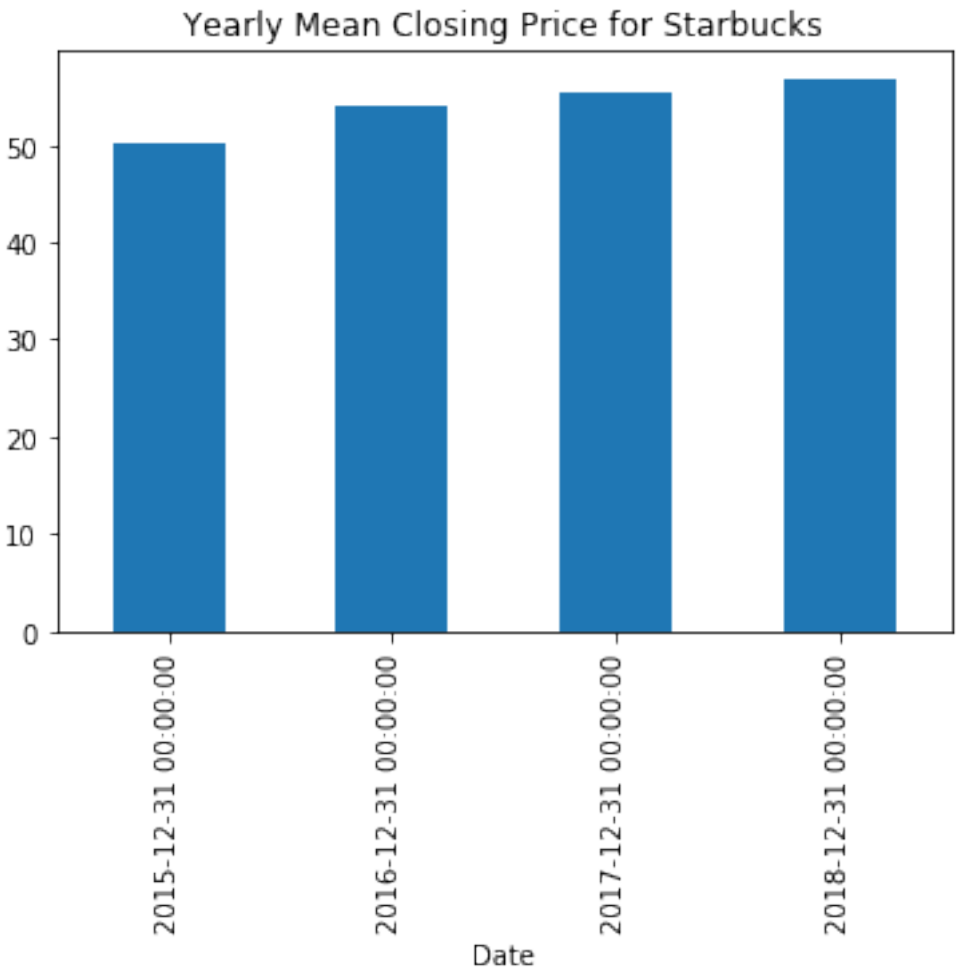
[ ]:
```
df['Close'].resample('A').mean().plot.bar(title='Yearly Mean Closing Price for␣
↪Starbucks');
```



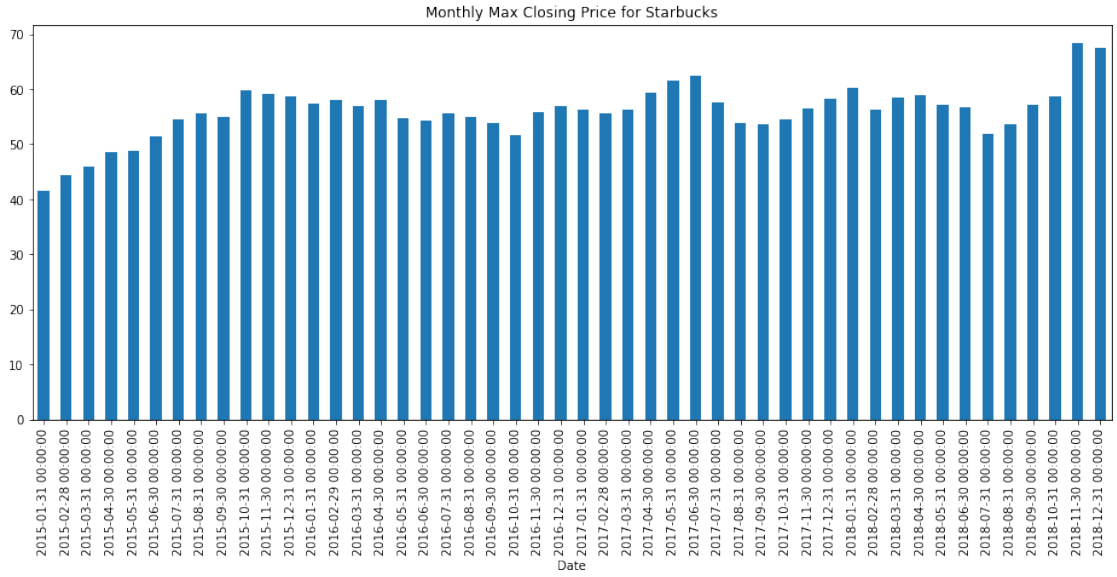
Pandas treats each sample as its own trace, and by default assigns different colors to each one. If you want, you can pass a color argument to assign your own color collection, or to set a uniform color. For example, color='#1f77b4' sets a uniform "steel blue" color.

Also, the above code can be broken into two lines for improved readability.

[ ]:
```
title = 'Yearly Mean Closing Price for Starbucks'
df['Close'].resample('A').mean().plot.bar(title=title,color=['#1f77b4']);
```



[ ]:
```
title = 'Monthly Max Closing Price for Starbucks'
df['Close'].resample('M').max().plot.bar(figsize=(16,6),␣
 ↪title=title,color='#1f77b4');
```



That is it! Up next we'll learn about time shifts!

3 Time Shifting

Sometimes you may need to shift all your data up or down along the time series index. In fact, a lot of pandas built-in methods do this under the hood. This isn't something we'll do often in the course, but it's definitely good to know about this anyways!

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

[ ]:
```
df = pd.read_csv('../Data/starbucks.csv',index_col='Date',parse_dates=True)
```

[ ]:
```
df.head()
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
2015-01-07  37.8848   9732554
2015-01-08  38.4961  13170548
```

[ ]:
```
df.tail()
```

[ ]:
```
             Close    Volume
Date
2018-12-24  60.56   6323252
2018-12-26  63.08  16646238
```

```
2018-12-27  63.20  11308081
2018-12-28  63.39   7712127
2018-12-31  64.40   7690183
```

3.1 .shift() forward

This method shifts the entire date index a given number of rows, without regard for time periods (months & years).It returns a modified copy of the original DataFrame.

[ ]:
```
df.shift(1).head()
```

[ ]:
```
              Close      Volume
Date
2015-01-02      NaN         NaN
2015-01-05  38.0061   6906098.0
2015-01-06  37.2781  11623796.0
2015-01-07  36.9748   7664340.0
2015-01-08  37.8848   9732554.0
```

[ ]:
```
# NOTE: You will lose that last piece of data that no longer has an index!
df.shift(1).tail()
```

[ ]:
```
            Close      Volume
Date
2018-12-24  61.39  23524888.0
2018-12-26  60.56   6323252.0
2018-12-27  63.08  16646238.0
2018-12-28  63.20  11308081.0
2018-12-31  63.39   7712127.0
```

3.2 .shift() backwards

[ ]:
```
df.shift(-1).head()
```

[ ]:
```
              Close      Volume
Date
2015-01-02  37.2781  11623796.0
2015-01-05  36.9748   7664340.0
2015-01-06  37.8848   9732554.0
2015-01-07  38.4961  13170548.0
2015-01-08  37.2361  27556706.0
```

[ ]:
```
df.shift(-1).tail()
```

[ ]:
```
            Close      Volume
Date
2018-12-24  63.08  16646238.0
2018-12-26  63.20  11308081.0
```

```
2018-12-27  63.39   7712127.0
2018-12-28  64.40   7690183.0
2018-12-31    NaN         NaN
```

3.3 Shifting based on Time Series Frequency Code

We can choose to shift index values up or down without realigning the data by passing in a freq argument. This method shifts dates to the next period based on a frequency code. Common codes are 'M' for month-end and 'A' for year-end. Refer to the Time Series Offset Aliases table from the Time Resampling lecture for a full list of values, or click here.

[ ]:
```
# Shift everything forward one month
df.shift(periods=1, freq='M').head()
```

[ ]:
```
              Close    Volume
Date
2015-01-31  38.0061   6906098
2015-01-31  37.2781  11623796
2015-01-31  36.9748   7664340
2015-01-31  37.8848   9732554
2015-01-31  38.4961  13170548
```

For more info on time shifting visit http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shift.html Up next we'll look at rolling and expanding!

4 Rolling and Expanding

A common process with time series is to create data based off of a rolling mean. The idea is to divide the data into "windows" of time, and then calculate an aggregate function for each window. In this way we obtain a simple moving average. Let's show how to do this easily with pandas!

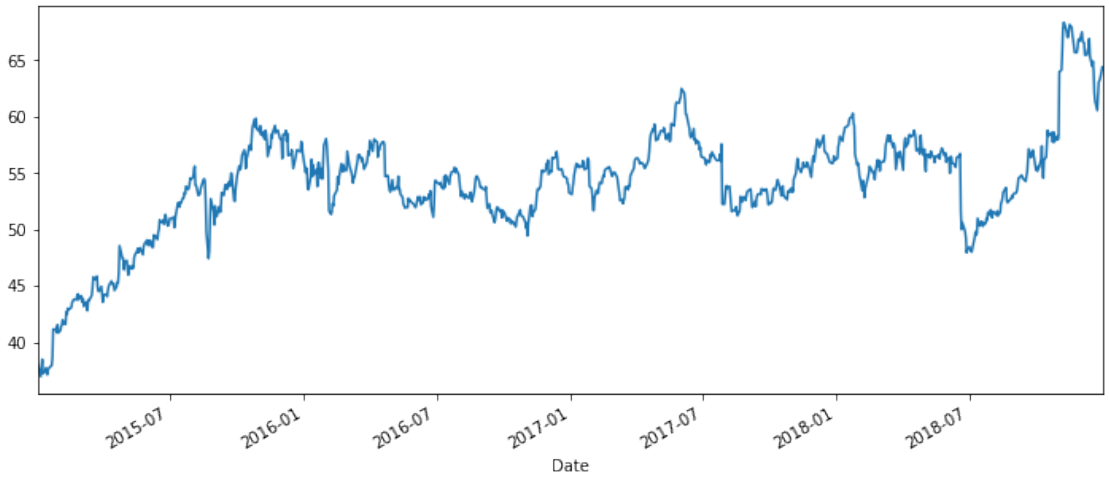
[ ]:
```
import pandas as pd
%matplotlib inline
```

[ ]:
```
# Import the data:
df = pd.read_csv('../Data/starbucks.csv', index_col='Date', parse_dates=True)
```

[ ]:
```
df.head()
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
2015-01-07  37.8848   9732554
2015-01-08  38.4961  13170548
```

[ ]:
```
df['Close'].plot(figsize=(12,5)).autoscale(axis='x',tight=True);
```



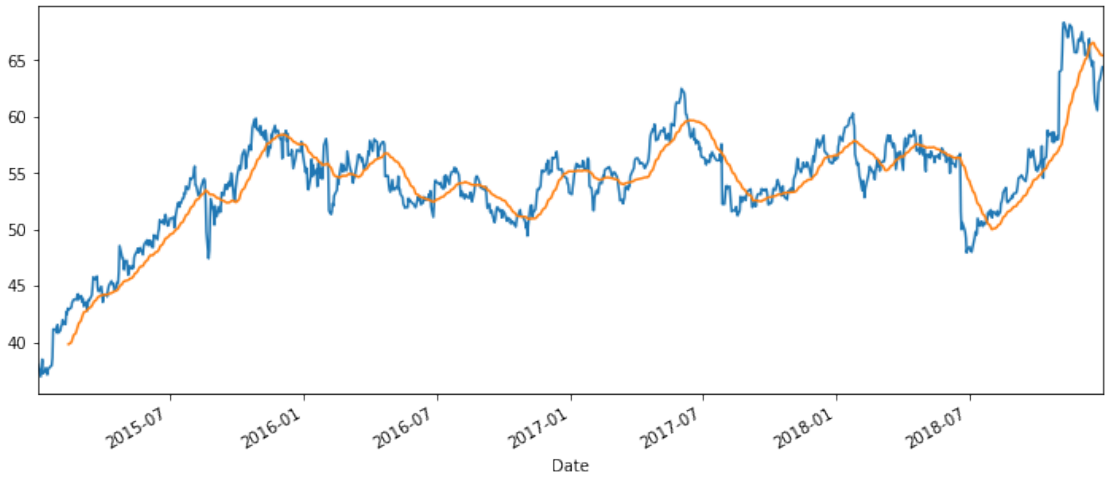
Now let's add in a rolling mean! This rolling method provides row entries, where every entry is then representative of the window.

[ ]:
```
# 7 day rolling mean
df.rolling(window=7).mean().head(15)
```

[ ]:

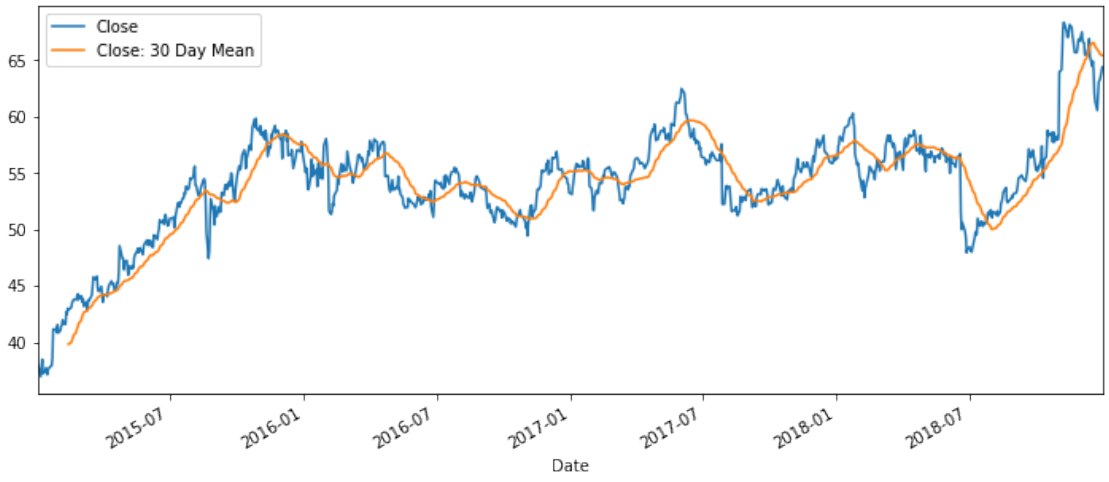
Date	Close	Volume
2015-01-02	NaN	NaN
2015-01-05	NaN	NaN
2015-01-06	NaN	NaN
2015-01-07	NaN	NaN
2015-01-08	NaN	NaN
2015-01-09	NaN	NaN
2015-01-12	37.616786	1.238222e+07
2015-01-13	37.578786	1.297288e+07
2015-01-14	37.614786	1.264020e+07
2015-01-15	37.638114	1.270624e+07
2015-01-16	37.600114	1.260380e+07
2015-01-20	37.515786	1.225634e+07
2015-01-21	37.615786	9.868837e+06
2015-01-22	37.783114	1.185335e+07
2015-01-23	38.273129	1.571999e+07

[ ]:
```
df['Close'].plot(figsize=(12,5)).autoscale(axis='x',tight=True)
df.rolling(window=30).mean()['Close'].plot();
```



The easiest way to add a legend is to make the rolling value a new column, then pandas does it automatically!

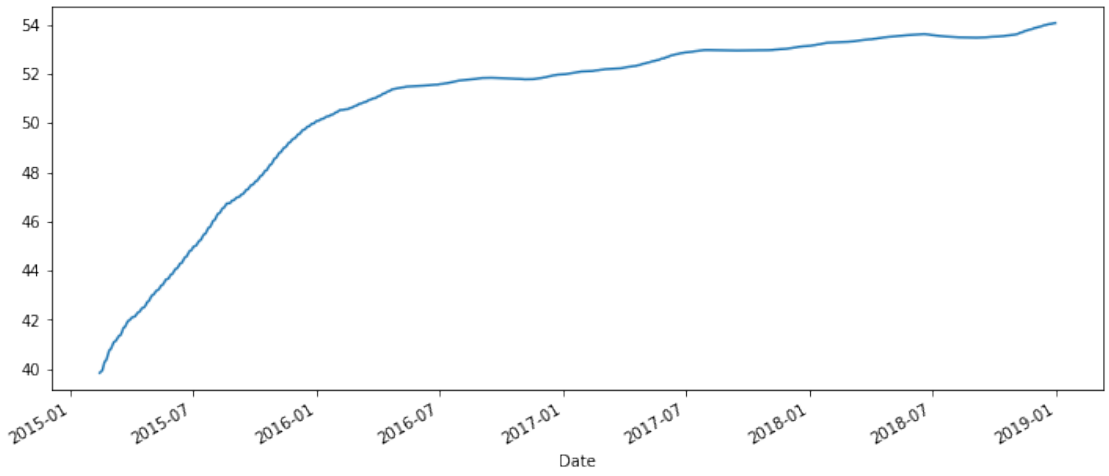
[ ]:
```
df['Close: 30 Day Mean'] = df['Close'].rolling(window=30).mean()
df[['Close','Close: 30 Day Mean']].plot(figsize=(12,5)).
  ↪autoscale(axis='x',tight=True);
```



4.1 Expanding

Instead of calculating values for a rolling window of dates, what if you wanted to take into account everything from the start of the time series up to each point in time? For example, instead of considering the average over the last 7 days, we would consider all prior data in our expanding set of averages.

[ ]:
```
# df['Close'].plot(figsize=(12,5)).autoscale(axis='x',tight=True)

# Optional: specify a minimum number of periods to start from
df['Close'].expanding(min_periods=30).mean().plot(figsize=(12,5));
```



That's it! It doesn't help much to visualize an expanding operation against the daily data, since all it really gives us is a picture of the "stability" or "volatility" of a stock. However, if you do want to see it, simply uncomment the first plot line above and rerun the cell.

Next up, we'll take a deep dive into visualizing time series data!

5 Visualizing Time Series Data

Let's go through a few key points of creating nice time series visualizations!

[ ]:
```
import pandas as pd
%matplotlib inline
```

[ ]:
```
df = pd.read_csv('../Data/starbucks.csv',index_col='Date',parse_dates=True)
```

[ ]:
```
df.head()
```

[ ]:

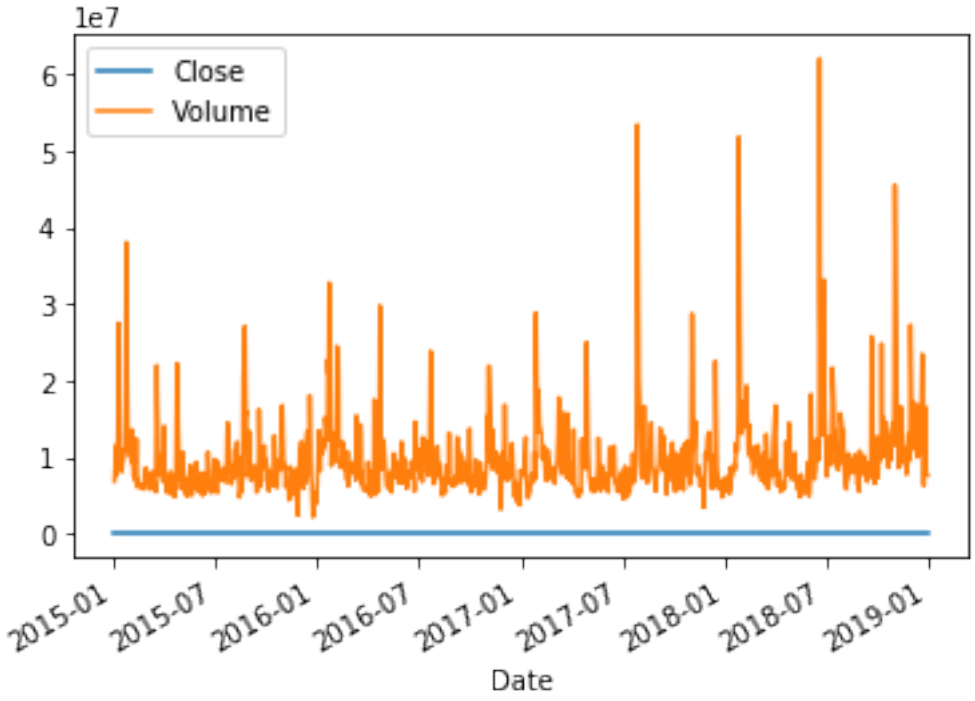
Date	Close	Volume
2015-01-02	38.0061	6906098
2015-01-05	37.2781	11623796
2015-01-06	36.9748	7664340
2015-01-07	37.8848	9732554
2015-01-08	38.4961	13170548

```
[ ]: # To show that dates are already parsed
     df.index
```

```
[ ]: DatetimeIndex(['2015-01-02', '2015-01-05', '2015-01-06', '2015-01-07',
                    '2015-01-08', '2015-01-09', '2015-01-12', '2015-01-13',
                    '2015-01-14', '2015-01-15',
                    …
                    '2018-12-17', '2018-12-18', '2018-12-19', '2018-12-20',
                    '2018-12-21', '2018-12-24', '2018-12-26', '2018-12-27',
                    '2018-12-28', '2018-12-31'],
                   dtype='datetime64[ns]', name='Date', length=1006, freq=None)
```

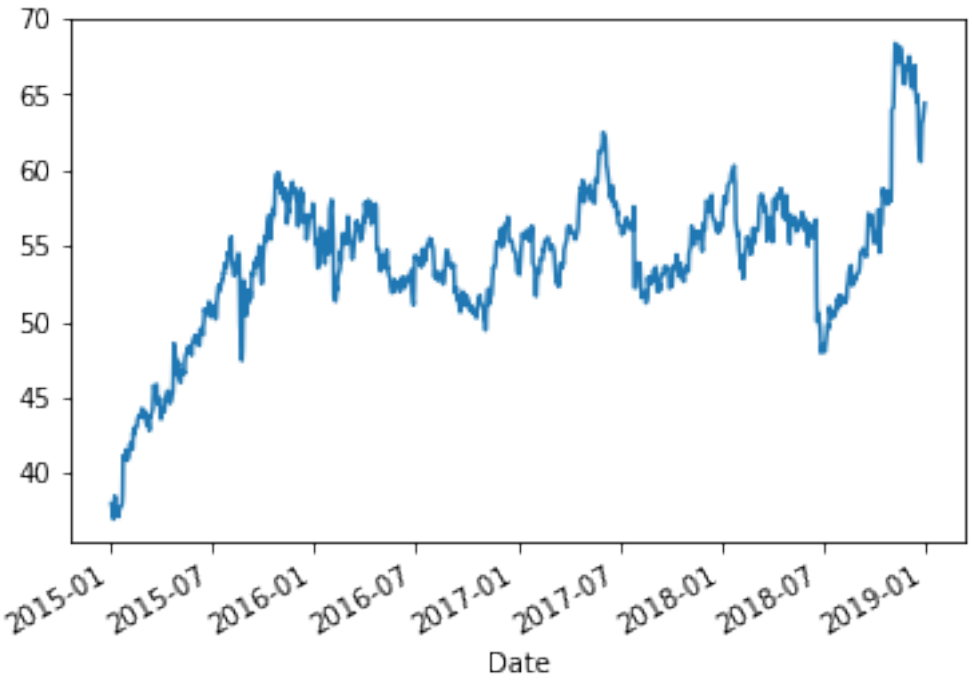
First we'll create a line plot that puts both 'Close' and 'Volume' on the same graph.Remember that we can use df.plot() in place of df.plot.line()

```
[ ]: df.plot();
```

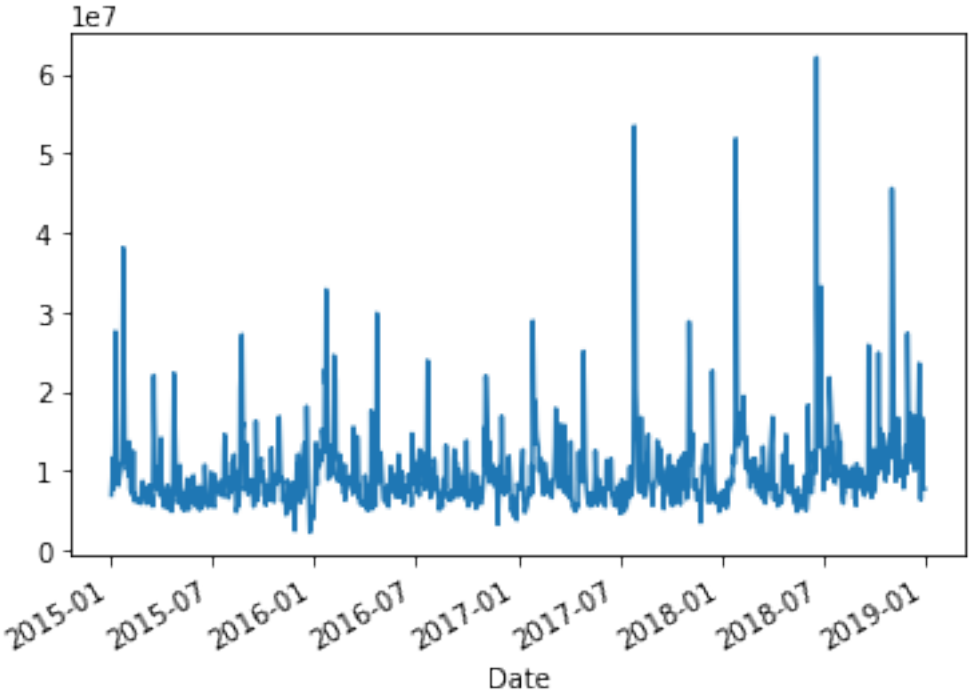


This isn't very helpful due to the difference in y-values, so we'll split them up.

```
[ ]: df['Close'].plot();
```



[ ]:
```
df['Volume'].plot();
```

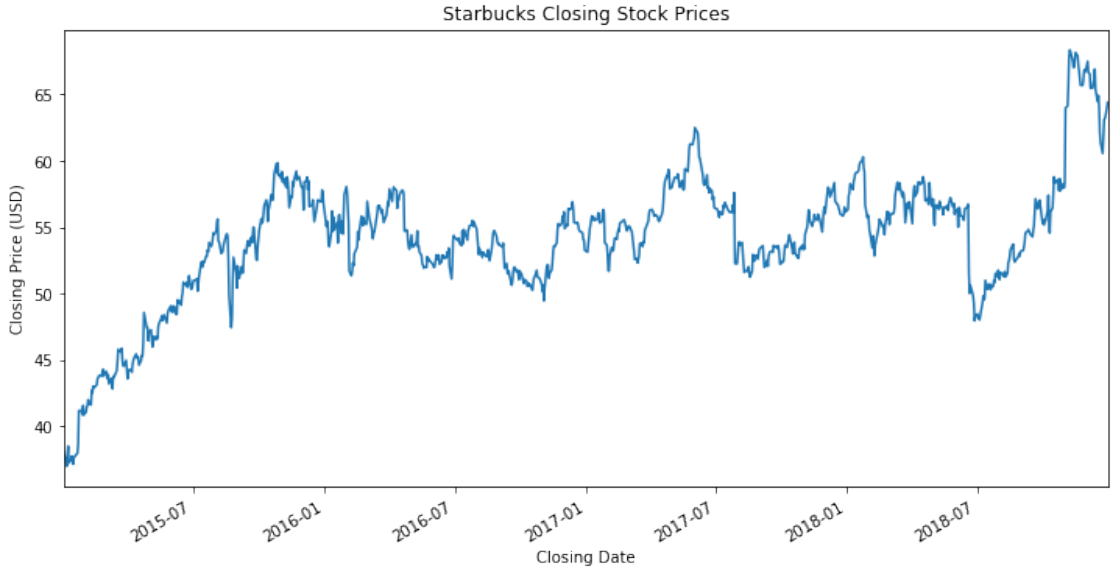


6 Plot Formatting

6.1 Adding a title and axis labels

NOTE: While we can pass a title into the pandas .plot() function, we can't pass x- and y-axis labels.However, since .plot() returns a matplotlib.axes.AxesSubplot object, we can set the labelson that object so long as we do it in the same jupyter cell. Setting an autoscale is done the same way.

[ ]:
```
title='Starbucks Closing Stock Prices'
ylabel='Closing Price (USD)'
xlabel='Closing Date'

ax = df['Close'].plot(figsize=(12,6),title=title)
ax.autoscale(axis='x',tight=True)
ax.set(xlabel=xlabel, ylabel=ylabel);
```



6.2 X Limits

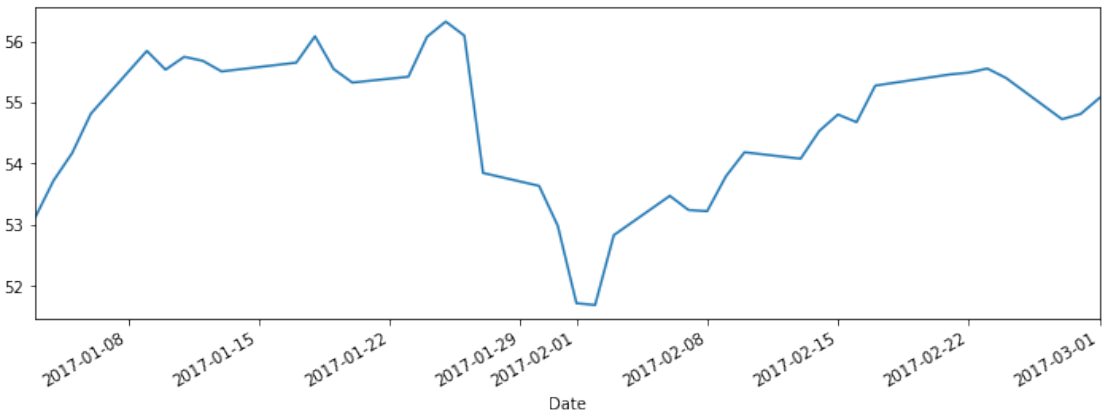
There are two ways we can set a specific span of time as an x-axis limit. We can plot a slice of the dataset, or we can pass x-limit values as an argument into df.plot().

The advantage of using a slice is that pandas automatically adjusts the y-limits accordingly.

The advantage of passing in arguments is that pandas automatically tightens the x-axis. Plus, if we're also setting y-limits this can improve readability.

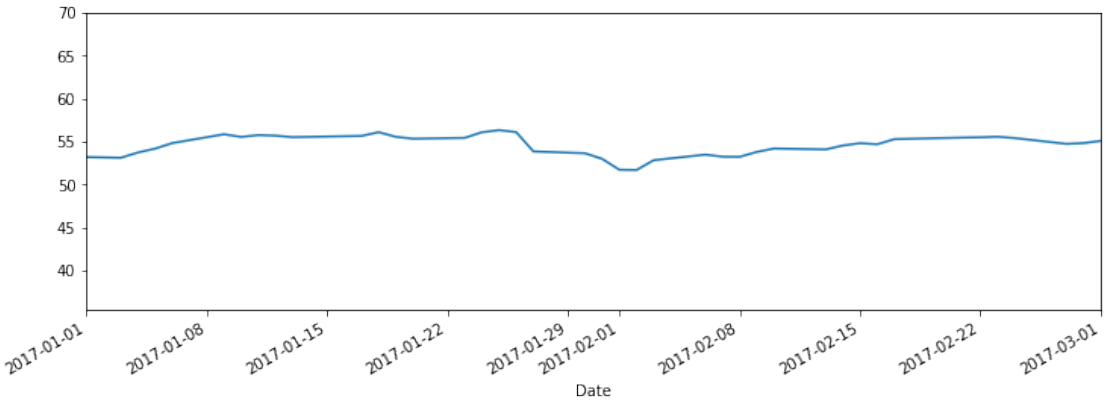
6.2.1 Choosing X Limits by Slice:

```
[ ]: # Dates are separated by a colon:
df['Close']['2017-01-01':'2017-03-01'].plot(figsize=(12,4)).
  ↪autoscale(axis='x',tight=True);
```



6.2.2 Choosing X Limits by Argument:

```
[ ]: # Dates are separated by a comma:
df['Close'].plot(figsize=(12,4),xlim=['2017-01-01','2017-03-01']);
```



NOTE: It's worth noting that the limit values do not have to appear in the index. Pandas will plot the actual dates based on their location in time.Also, another advantage of slicing over arguments is that it's easier to include the upper/lower bound as a limit.That is, df['column']['2017-01-01':].plot() is easier to type than df['column'].plot(xlim=('2017-01-01',df.index.max()))

Now let's focus on the y-axis limits to get a better sense of the shape of the data.First we'll find out what upper and lower limits to use.

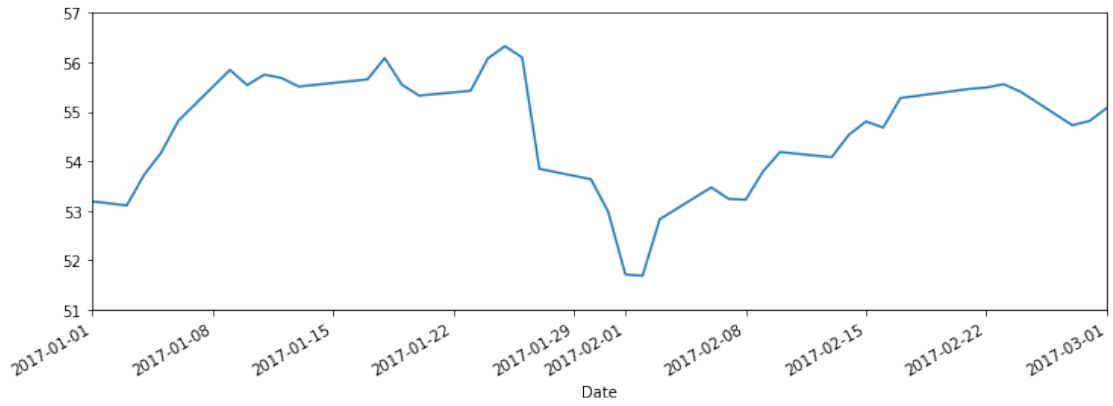
```
[ ]: # FIND THE MINIMUM VALUE IN THE RANGE:
     df.loc['2017-01-01':'2017-03-01']['Close'].min()
```

```
[ ]: 51.6899
```

```
[ ]: # FIND THE MAXIMUM VALUE IN THE RANGE:
     df.loc['2017-01-01':'2017-03-01']['Close'].max()
```

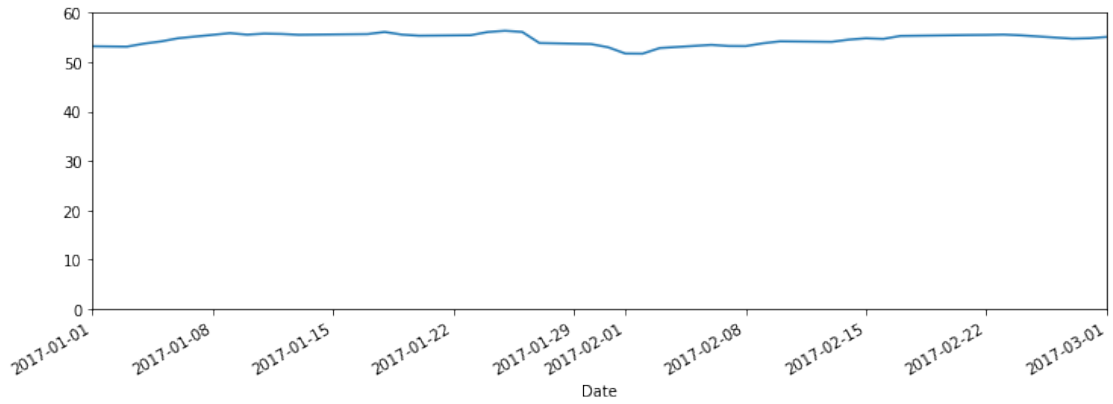
```
[ ]: 56.3244
```

```
[ ]: # PLUG THESE IN AS Y-LIMIT VALUES:
     df['Close'].plot(figsize=(12,4),xlim=['2017-01-01','2017-03-01'],ylim=[51,57]);
```



NOTE: Be careful when setting y-axis limits! Setting too narrow a slice can make graphs appear overly volatile. The above chart might lead you to believe that stocks were many times more valuable in January 2017 than in early February, but a look at them with the y-axis minimum set to zero tells a different story:

```
[ ]: df['Close'].plot(figsize=(12,4),xlim=['2017-01-01','2017-03-01'],ylim=[0,60]);
```

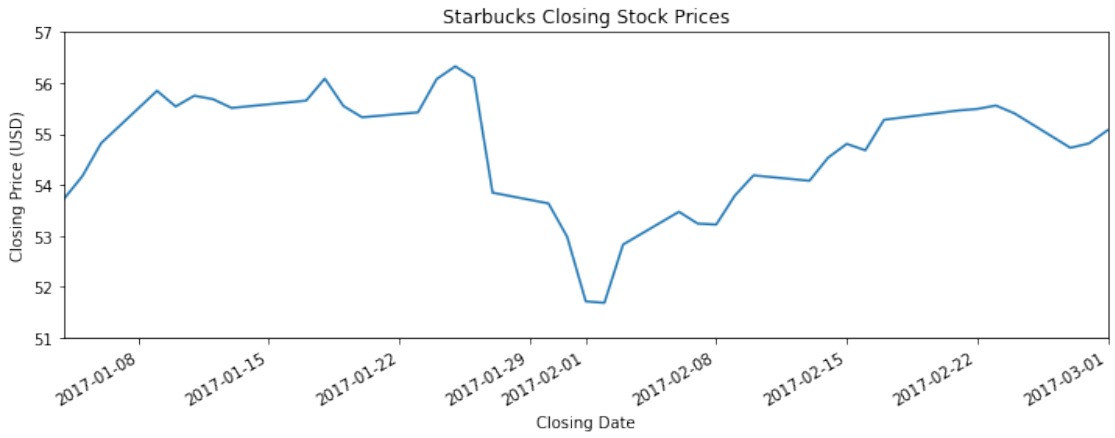


6.3 Title and axis labels

Let's add a title and axis labels to our subplot.

REMEMBER: ax.autoscale(axis='both',tight=True) is unnecessary if axis limits have been passed into .plot(). If we were to add it, autoscale would revert the axis limits to the full dataset.

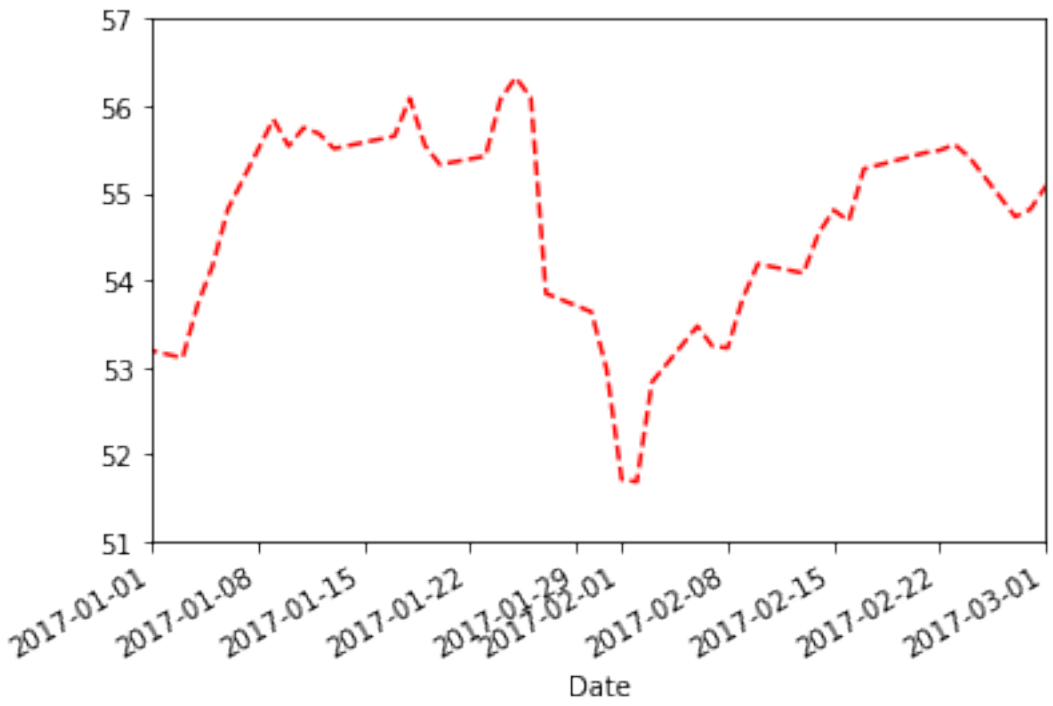
[ ]:
```
title='Starbucks Closing Stock Prices'
ylabel='Closing Price (USD)'
xlabel='Closing Date'

ax = df['Close'].
  ↪plot(xlim=['2017-01-04','2017-03-01'],ylim=[51,57],figsize=(12,4),title=title)
ax.set(xlabel=xlabel, ylabel=ylabel);
```



6.4 Color and Style

We can pass arguments into .plot() to change the linestyle and color. Refer to the Customizing Plots lecture from the previous section for more options.

[ ]:
```
df['Close'].plot(xlim=['2017-01-01','2017-03-01'],ylim=[51,57],ls='--',c='r');
```



6.5 X Ticks

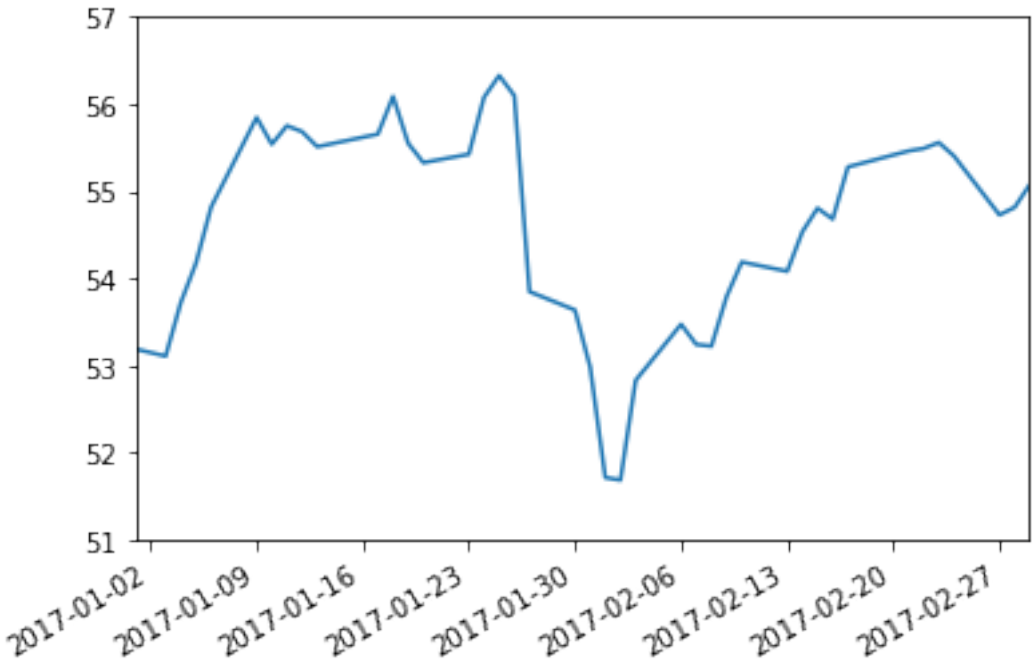
In this section we'll look at how to change the format and appearance of dates along the x-axis. To do this, we'll borrow a tool from matplotlib called dates.

[ ]:
```
from matplotlib import dates
```

6.5.1 Set the spacing

The x-axis values can be divided into major and minor axes. For now, we'll work only with the major axis and learn how to set the spacing with .set_major_locator().

[ ]:
```
# CREATE OUR AXIS OBJECT
ax = df['Close'].plot(xlim=['2017-01-01','2017-03-01'],ylim=[51,57])

# REMOVE PANDAS DEFAULT "Date" LABEL
ax.set(xlabel='')

# SET THE TICK LOCATOR AND FORMATTER FOR THE MAJOR AXIS
ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday=0))
```



Notice that dates are spaced one week apart. The dates themselves correspond with byweekday=0, or Mondays. For a full list of locator options available from matplotlib.dates visit https://matplotlib.org/api/dates_api.html#date-tickers

6.5.2 Date Formatting

Formatting follows the Python datetime strftime codes.
The following examples are based on `datetime.datetime(2001, 2, 3, 16, 5, 6)`:

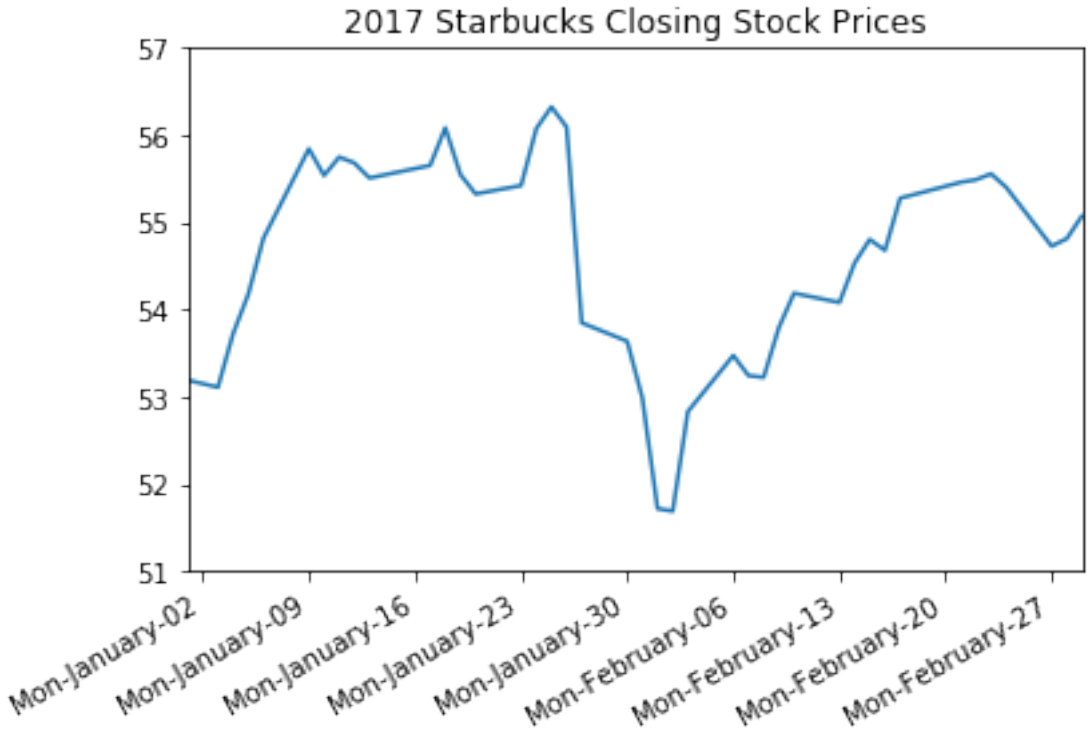
CODE	MEANING	EXAMPLE
%Y	Year with century as a decimal number.	2001
%y	Year without century as a zero-padded decimal number.	01
%m	Month as a zero-padded decimal number.	02
%B	Month as locale's full name.	February
%b	Month as locale's abbreviated name.	Feb
%d	Day of the month as a zero-padded decimal number.	03
%A	Weekday as locale's full name.	Saturday
%a	Weekday as locale's abbreviated name.	Sat
%H	Hour (24-hour clock) as a zero-padded decimal number.	16
%I	Hour (12-hour clock) as a zero-padded decimal number.	04
%p	Locale's equivalent of either AM or PM.	PM
%M	Minute as a zero-padded decimal number.	05

CODE	MEANING	EXAMPLE
%S	Second as a zero-padded decimal number.	06
%#m	Month as a decimal number. (Windows)	2
%-m	Month as a decimal number. (Mac/Linux)	2
%#x	Long date	Saturday, February 03, 2001
%#c	Long date and time	Saturday, February 03, 2001 16:05:06

[ ]:
```
# USE THIS SPACE TO EXPERIMENT WITH DIFFERENT FORMATS
from datetime import datetime
datetime(2001, 2, 3, 16, 5, 6).strftime("%A, %B %d, %Y  %I:%M:%S %p")
```

[ ]:
```
'Saturday, February 03, 2001  04:05:06 PM'
```

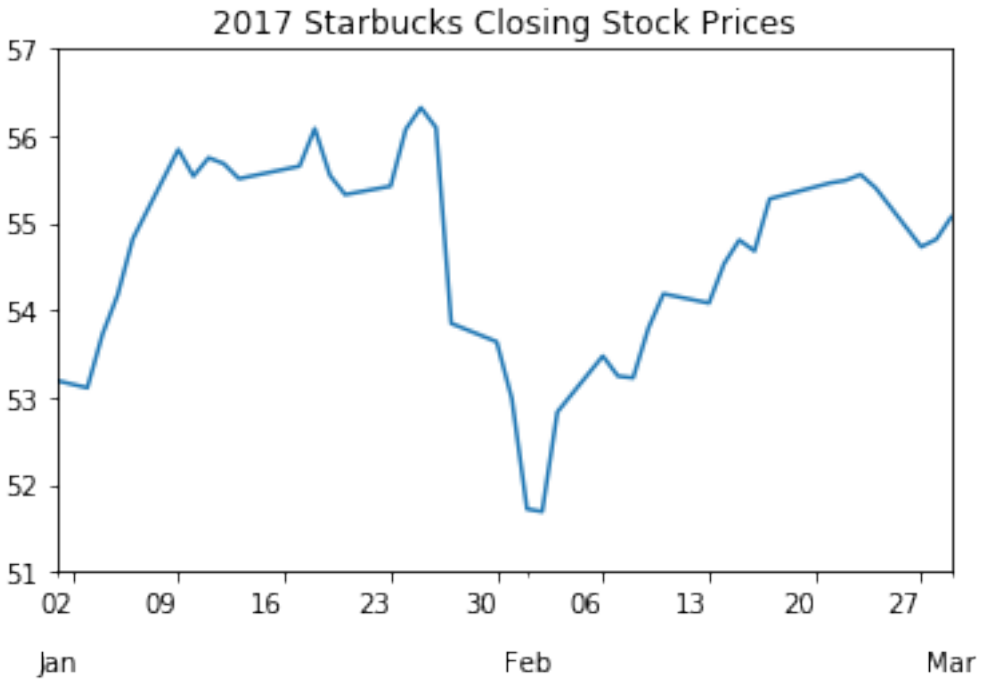
[ ]:
```
ax = df['Close'].plot(xlim=['2017-01-01','2017-03-01'],ylim=[51,57],title='2017␣
 ↪Starbucks Closing Stock Prices')
ax.set(xlabel='')

ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday=0))
ax.xaxis.set_major_formatter(dates.DateFormatter("%a-%B-%d"))
```



6.6 Major vs. Minor Axis Values

All of the tick marks we've used so far have belonged to the major axis. We can assign another level called the minor axis, perhaps to separate month names from days of the month.

[ ]:
```
ax = df['Close'].
  ↪plot(xlim=['2017-01-01','2017-03-01'],ylim=[51,57],rot=0,title='2017␣
  ↪Starbucks Closing Stock Prices')
ax.set(xlabel='')

ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday=0))
ax.xaxis.set_major_formatter(dates.DateFormatter('%d'))

ax.xaxis.set_minor_locator(dates.MonthLocator())
ax.xaxis.set_minor_formatter(dates.DateFormatter('\n\n%b'))
```



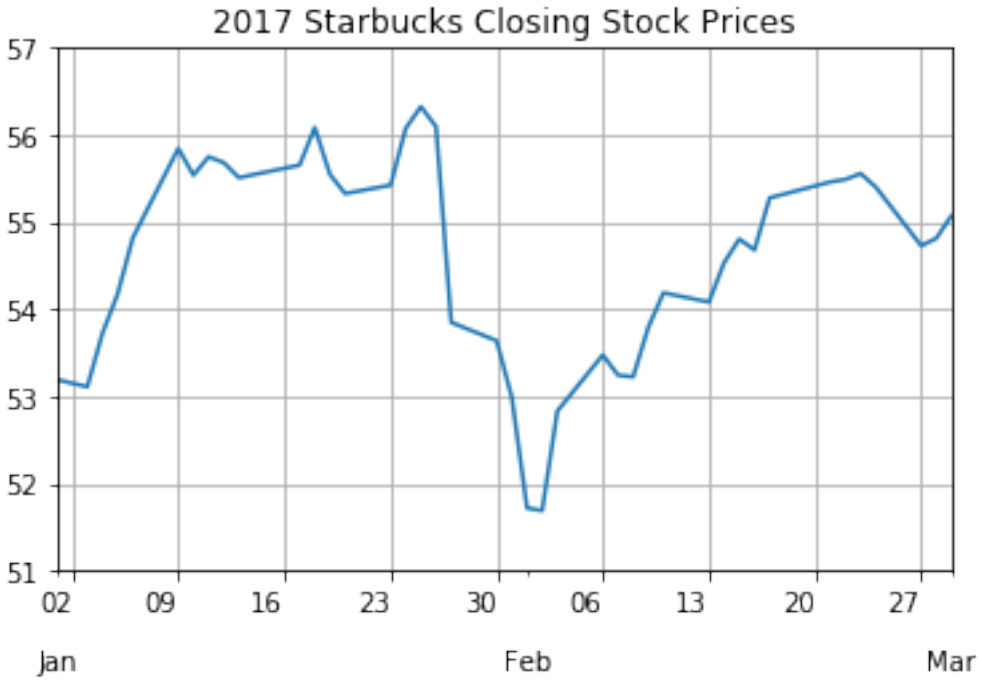
NOTE: we passed a rotation argument rot=0 into df.plot() so that the major axis values appear horizontal, not slanted.

6.7 Adding Gridlines

We can add x and y axis gridlines that extend into the plot from each major tick mark.

[ ]:

```
ax = df['Close'].
  ↪plot(xlim=['2017-01-01','2017-03-01'],ylim=[51,57],rot=0,title='2017␣
  ↪Starbucks Closing Stock Prices')
ax.set(xlabel='')

ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday=0))
ax.xaxis.set_major_formatter(dates.DateFormatter('%d'))

ax.xaxis.set_minor_locator(dates.MonthLocator())
ax.xaxis.set_minor_formatter(dates.DateFormatter('\n\n%b'))

ax.yaxis.grid(True)
ax.xaxis.grid(True)
```



[ ]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from datetime import datetime # datetime library
```

7 Working with Dates

```
[ ]: # Creating Dates with datetime (year, month, day, hour, min, secs)

     datetime(2024, 2, 14)
```

```
[ ]: datetime.datetime(2024, 2, 14, 0, 0)
```

```
[ ]: # Creating Dates from array
     # Capital D means 'Day level precision'

     np.array(['2020-03-15', '2020-03-16', '2020-03-17'], dtype='datetime64')
```

```
[ ]: array(['2020-03-15', '2020-03-16', '2020-03-17'], dtype='datetime64[D]')
```

```
[ ]: # Creating Dates from array
     # Capital Y means 'Year level precision'
     # Other settings, Y, M, D, h, s

     np.array(['2020-03-15', '2020-03-16', '2020-03-17'], dtype='datetime64[M]')
```

```
[ ]: array(['2020-03', '2020-03', '2020-03'], dtype='datetime64[M]')
```

```
[ ]: # Date Ranges (7 day step)

     np.arange('2018-06-01', '2130-06-23', 7, dtype='datetime64[Y]')
```

```
[ ]: array(['2018', '2025', '2032', '2039', '2046', '2053', '2060', '2067',
            '2074', '2081', '2088', '2095', '2102', '2109', '2116', '2123'],
           dtype='datetime64[Y]')
```

```
[ ]: # Date Ranges (Months, Year)

     np.arange('1968', '2024', dtype='datetime64[Y]')
```

```
[ ]: array(['1968', '1969', '1970', '1971', '1972', '1973', '1974', '1975',
            '1976', '1977', '1978', '1979', '1980', '1981', '1982', '1983',
            '1984', '1985', '1986', '1987', '1988', '1989', '1990', '1991',
            '1992', '1993', '1994', '1995', '1996', '1997', '1998', '1999',
            '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007',
            '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015',
            '2016', '2017', '2018', '2019', '2020', '2021', '2022', '2023'],
           dtype='datetime64[Y]')
```

```
[ ]: # Pandas date_range

     pd.date_range('2020-01-01', '2024-01-01')
```

```
[ ]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',
                    '2020-01-05', '2020-01-06', '2020-01-07', '2020-01-08',
                    '2020-01-09', '2020-01-10',
                    …
                    '2023-12-23', '2023-12-24', '2023-12-25', '2023-12-26',
                    '2023-12-27', '2023-12-28', '2023-12-29', '2023-12-30',
                    '2023-12-31', '2024-01-01'],
                   dtype='datetime64[ns]', length=1462, freq='D')
```

```
[ ]: # Pandas date_range (periods, freq)
     # freq values (D - daily, W - weekly, M - month end, Q - quarter end)
     # freq values (A - year end)
     # Full details

     pd.date_range('2020-01-01', periods = 7, freq = 'W')
```

```
[ ]: DatetimeIndex(['2020-01-05', '2020-01-12', '2020-01-19', '2020-01-26',
                    '2020-02-02', '2020-02-09', '2020-02-16'],
                   dtype='datetime64[ns]', freq='W-SUN')
```

```
[ ]: # Pandas date_range (start, end, freq)
     # freq values (D - daily, W - weekly, M - month end, Q - quarter end)
     # freq values (A - year end)
     # Full details

     pd.date_range(start = '1924-02-14', end = '2024-02-14', freq = 'D')
```

```
[ ]: DatetimeIndex(['1924-02-14', '1924-02-15', '1924-02-16', '1924-02-17',
                    '1924-02-18', '1924-02-19', '1924-02-20', '1924-02-21',
                    '1924-02-22', '1924-02-23',
                    …
                    '2024-02-05', '2024-02-06', '2024-02-07', '2024-02-08',
                    '2024-02-09', '2024-02-10', '2024-02-11', '2024-02-12',
                    '2024-02-13', '2024-02-14'],
                   dtype='datetime64[ns]', length=36526, freq='D')
```

```
[ ]: # Pandas to_datetime

     pd.to_datetime(['1/2/2018', '1/3/2018'])
```

```
[ ]: DatetimeIndex(['2018-01-02', '2018-01-03'], dtype='datetime64[ns]', freq=None)
```

```
[ ]: # Pandas to_datetime (format)

     pd.to_datetime(['1/2/2018', '1/3/2018'])
```

```
[ ]: DatetimeIndex(['2018-01-02', '2018-01-03'], dtype='datetime64[ns]', freq=None)
```

[ ]:
```
# Pandas to_datetime (format)

pd.to_datetime(['1/2/2018', '1/3/2018'], format = '%m/%d/%Y')
```

[ ]:
```
DatetimeIndex(['2018-01-02', '2018-01-03'], dtype='datetime64[ns]', freq=None)
```

8 Time Resampling

[ ]:
```
# Load Data

df = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/starbucks.csv')
df.head()
```

[ ]:
```
         Date    Close    Volume
0  2015-01-02  38.0061   6906098
1  2015-01-05  37.2781  11623796
2  2015-01-06  36.9748   7664340
3  2015-01-07  37.8848   9732554
4  2015-01-08  38.4961  13170548
```

[ ]:
```
# Load Data with Specific Index

df = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/starbucks.csv',
                 index_col = 'Date')
df.head(3)
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
```

[ ]:
```
# Load Data with Specific Index and Recognized as TRUE dates

df = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/starbucks.csv'
                  ,index_col = 'Date', parse_dates = True)
df.head(3)
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
```

[ ]:
```
# Set Date Column to Index (Alternative)

# df.set_index('Date')
```

[ ]:
```
# Time Resampling
# Rule describes the frequency with which to apply the aggregation function

a = df.resample(rule = 'Y').sum() # min, max, sum, etc.
a
```

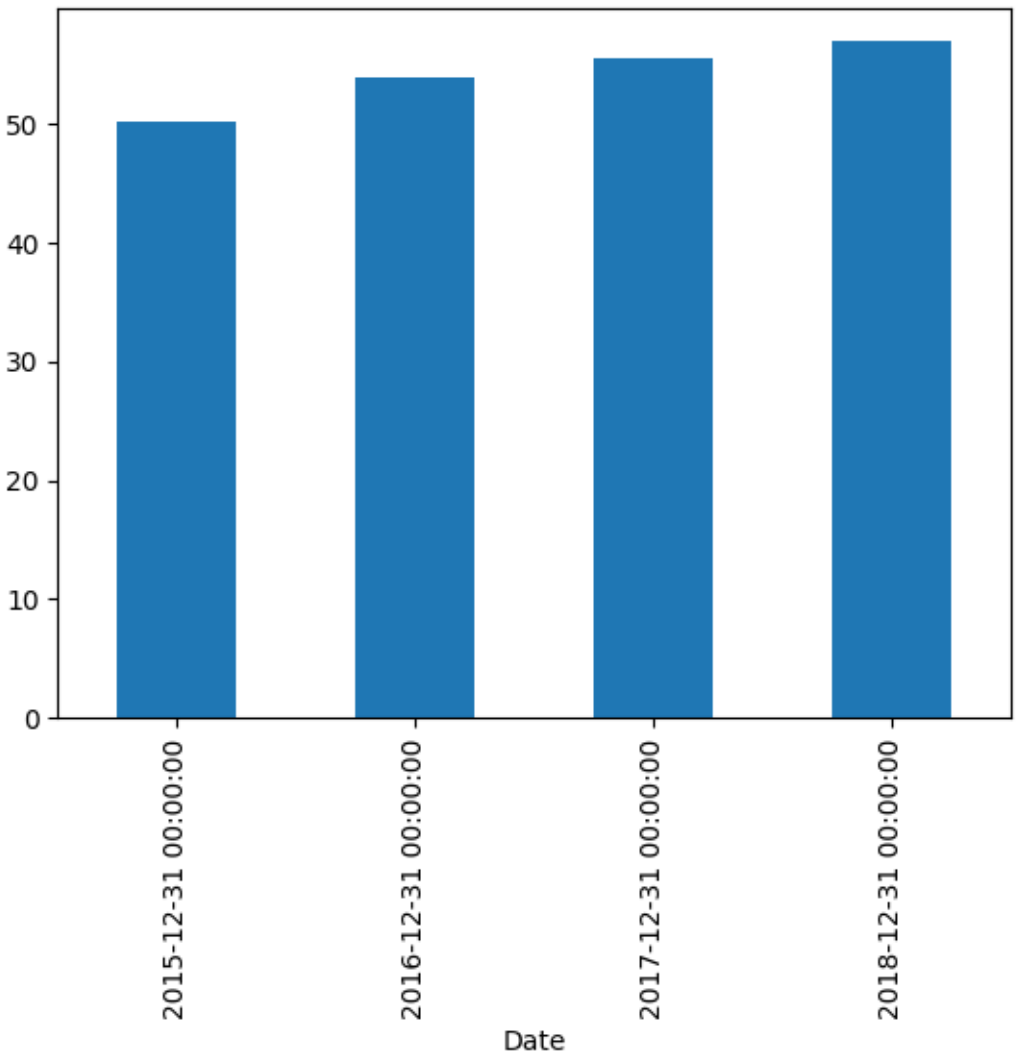
[ ]:
```
                 Close      Volume
Date
2015-12-31  12619.6811  2179595896
2016-12-31  13580.7164  2343759515
2017-12-31  13919.7849  2333315537
2018-12-31  14274.3712  2818435669
```

[ ]:
```
# Combine Resampling with Graphs for Analysis

df['Close'].resample('A').mean() # year-end frequency
```

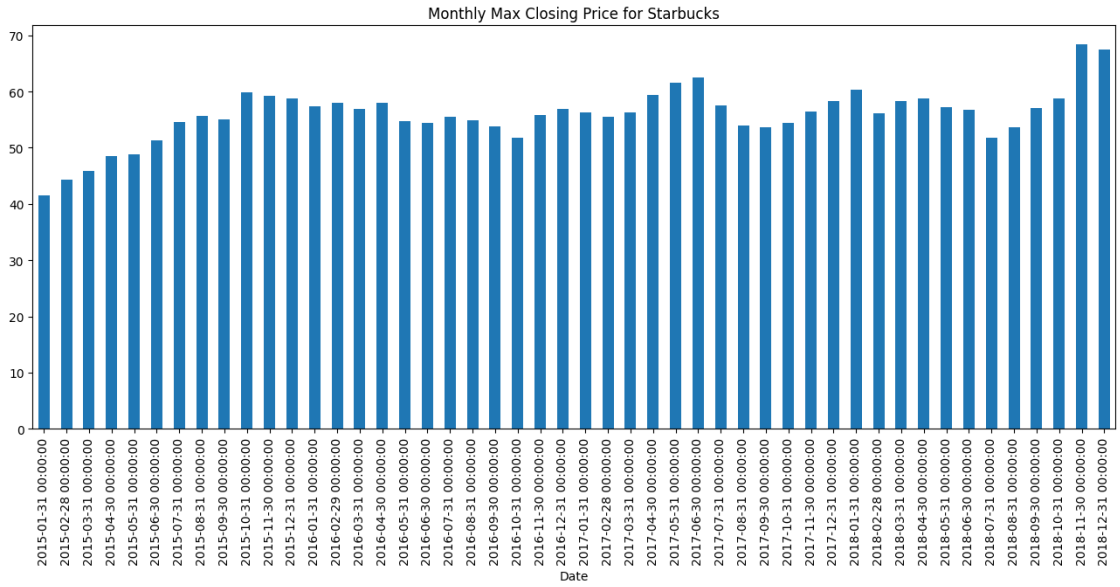
[ ]:
```
Date
2015-12-31    50.078100
2016-12-31    53.891732
2017-12-31    55.457310
2018-12-31    56.870005
Freq: A-DEC, Name: Close, dtype: float64
```

[ ]:
```
# Combine Resampling with Graphs for Analysis

# The closing price is often viewed as the most accurate valuation of a stock␣
 ↪at the end of a
# trading day because it reflects the consensus value agreed upon by buyers and␣
 ↪sellers.
# A stock's closing price can indicate the market's sentiment towards the stock␣
 ↪and can influence investor decisions for the
# next trading day.

df['Close'].resample('A').mean().plot(kind = 'bar')
```

[ ]:
```
<Axes: xlabel='Date'>
```



[ ]:
```
# Can you plot the 'Monthly Max Closing Price for Starbucks?

title = 'Monthly Max Closing Price for Starbucks'
df['Close'].resample('M').max().plot.bar(figsize=(16,6),␣
 ↪title=title,color='#1f77b4');
```



9 Time Shifting

[ ]:
```
df.head()
```

[ ]:
```
              Close    Volume
Date
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
2015-01-07  37.8848   9732554
2015-01-08  38.4961  13170548
```

[ ]:
```
df.tail()
```

[ ]:
```
             Close    Volume
Date
2018-12-24  60.56   6323252
2018-12-26  63.08  16646238
2018-12-27  63.20  11308081
2018-12-28  63.39   7712127
2018-12-31  64.40   7690183
```

[ ]:
```
# Time Shifting
# Number of rows?

df.shift(1) # Notice the NaN
```

[ ]:
```
              Close      Volume
Date
2015-01-02      NaN         NaN
2015-01-05  38.0061   6906098.0
2015-01-06  37.2781  11623796.0
2015-01-07  36.9748   7664340.0
2015-01-08  37.8848   9732554.0
…              …          …
2018-12-24  61.3900  23524888.0
2018-12-26  60.5600   6323252.0
2018-12-27  63.0800  16646238.0
2018-12-28  63.2000  11308081.0
2018-12-31  63.3900   7712127.0

[1006 rows x 2 columns]
```

[ ]:
```
df.shift(1).tail() # Notice we lost data [64.40        7690183]
```

[ ]:
```
             Close      Volume
Date
2018-12-24  61.39  23524888.0
2018-12-26  60.56   6323252.0
2018-12-27  63.08  16646238.0
2018-12-28  63.20  11308081.0
2018-12-31  63.39   7712127.0
```

[ ]:
```
# Time Shifting
# Number of rows?

df.shift(5) # Notice the NaN
```

[ ]:
```
             Close      Volume
Date
2015-01-02     NaN         NaN
2015-01-05     NaN         NaN
2015-01-06     NaN         NaN
2015-01-07     NaN         NaN
2015-01-08     NaN         NaN
…              …          …
2018-12-24  64.47  15143054.0
2018-12-26  64.92  10523476.0
2018-12-27  64.06  14390146.0
2018-12-28  62.15  20264918.0
2018-12-31  61.39  23524888.0

[1006 rows x 2 columns]
```

[ ]:
```
# Time Shifting Backwards
# Number of rows?

df.shift(-1) # Notice the NaN
```

[ ]:
```
              Close      Volume
Date
2015-01-02  37.2781  11623796.0
2015-01-05  36.9748   7664340.0
2015-01-06  37.8848   9732554.0
2015-01-07  38.4961  13170548.0
2015-01-08  37.2361  27556706.0
…              …         …
2018-12-24  63.0800  16646238.0
2018-12-26  63.2000  11308081.0
2018-12-27  63.3900   7712127.0
2018-12-28  64.4000   7690183.0
2018-12-31      NaN         NaN

[1006 rows x 2 columns]
```

[ ]:
```
# Time Shifting Backwards
# Number of rows?

df.shift(-1).head() # Notice the NaN
```

[ ]:
```
              Close      Volume
Date
2015-01-02  37.2781  11623796.0
2015-01-05  36.9748   7664340.0
2015-01-06  37.8848   9732554.0
2015-01-07  38.4961  13170548.0
2015-01-08  37.2361  27556706.0
```

[ ]:
```
# Shifting for a Month?
# periods, freq

df.shift(periods = 1, freq = 'Y') # shift TS every 1 month
```

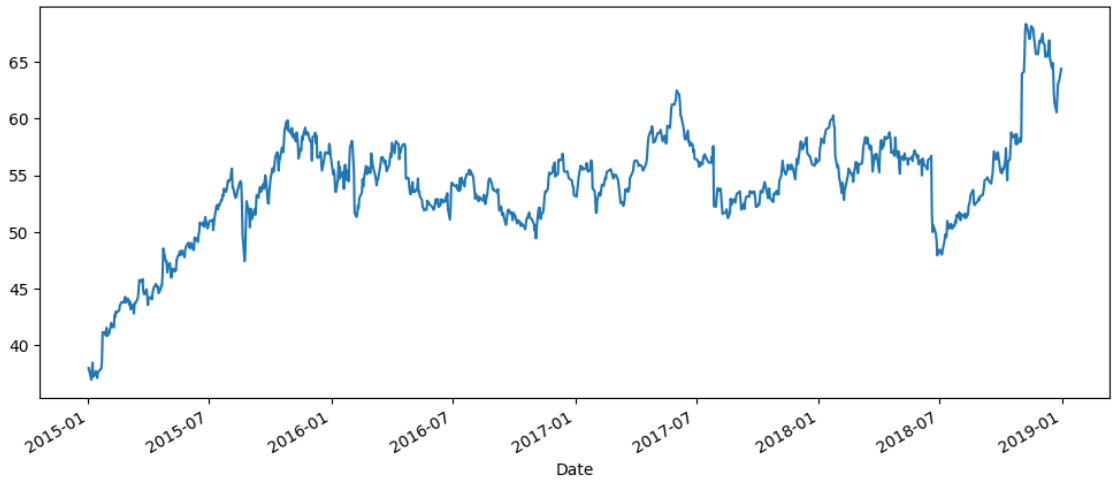
[ ]:
```
              Close    Volume
Date
2015-12-31  38.0061   6906098
2015-12-31  37.2781  11623796
2015-12-31  36.9748   7664340
2015-12-31  37.8848   9732554
2015-12-31  38.4961  13170548
…              …       …
```

```
2018-12-31   60.5600    6323252
2018-12-31   63.0800   16646238
2018-12-31   63.2000   11308081
2018-12-31   63.3900    7712127
2019-12-31   64.4000    7690183

[1006 rows x 2 columns]
```

10 Rolling Averages

[ ]:
```
# Original Closing Price Plot

df['Close'].plot(figsize = (12,5))
```

[ ]:
```
<Axes: xlabel='Date'>
```

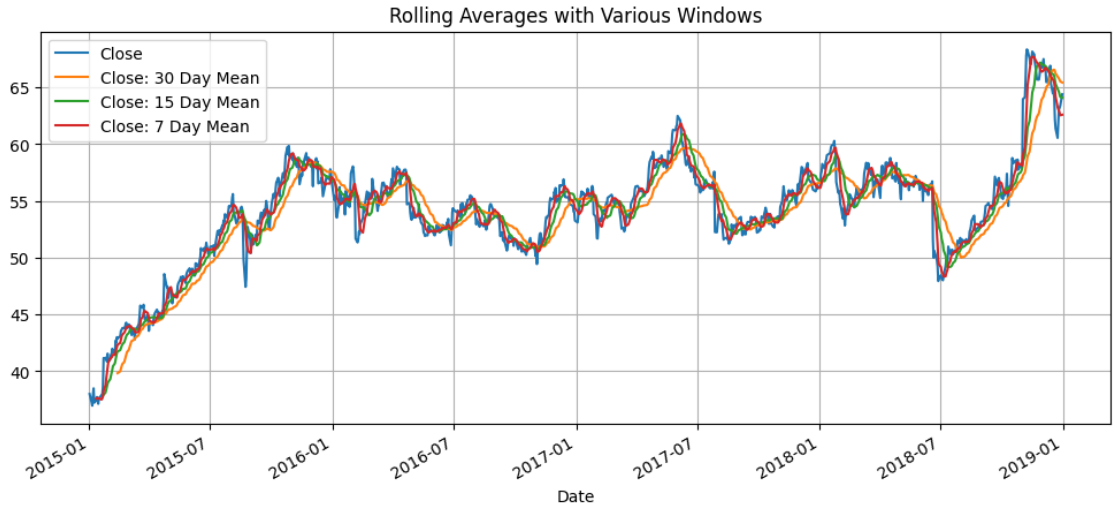


[ ]:
```
# Rolling Averages

# Rolling averages smooth out short-term fluctuations and highlight longer-term␣
 ↪trends or
# cycles in the data. This smoothing process makes it easier to visualize and␣
 ↪understand the
# underlying trends in the data by filtering out the "noise" from random␣
 ↪fluctuations.
# By averaging the data points over a specified period, it provides a clearer␣
 ↪view of the direction in which the series is moving.

# In time series analysis, identifying the underlying trend is crucial for␣
 ↪understanding the
```

```
# behavior of the series over time. Rolling averages can be used to easily␣
 ↪identify these trends.
# Whether the trend is upward, downward, or stable over a specific period, the␣
 ↪rolling average plots a
# smoother line that makes it easier to identify and analyze these trends␣
 ↪without the distraction of volatility in the data.

# Rolling averages can also be used as a simple forecasting method. By␣
 ↪projecting the moving average line into the future,
# analysts can make short-term forecasts based on the assumption that the␣
 ↪recent trend will continue for some time.
# Although it's a more naive approach compared to more sophisticated␣
 ↪forecasting models, it can serve as a good starting point or
# baseline for forecasting efforts, especially for initial exploratory analysis.

# Rolling averages serve as a powerful tool in the analysis and interpretation␣
 ↪of time series data,
# providing insights that might not be immediately apparent from the raw data␣
 ↪alone. By smoothing the data,
# highlighting underlying trends, and offering a basis for simple forecasting,␣
 ↪rolling averages facilitate a
# deeper understanding of the time series data's behavior over time.

df['Close'].plot(figsize = (12,5))
df.rolling(window = 15).mean()['Close'].plot(color = 'red')
```

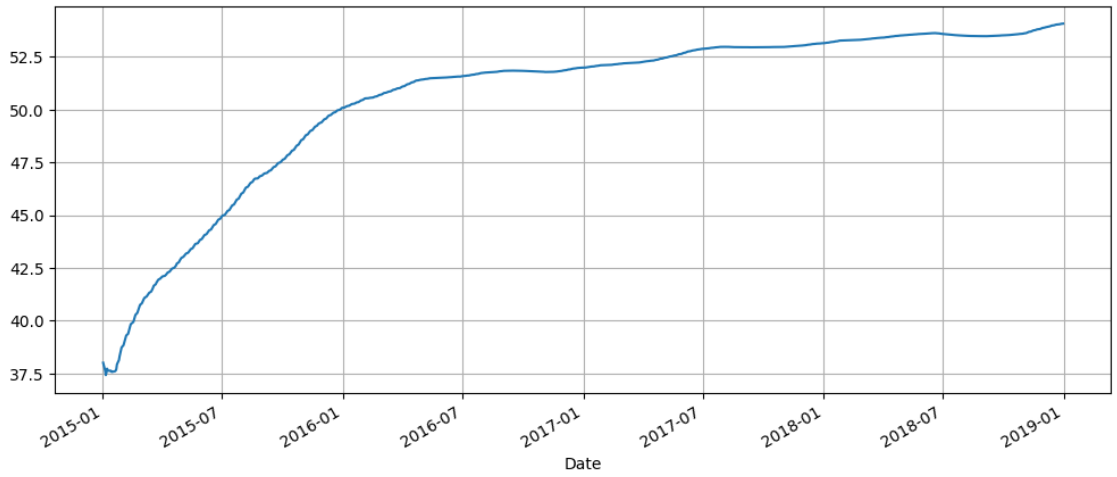
[ ]: <Axes: xlabel='Date'>

[ ]:
```
## Combining Windows in one plot

df['Close: 30 Day Mean'] = df['Close'].rolling(window = 30).mean()
df['Close: 15 Day Mean'] = df['Close'].rolling(window = 15).mean()
df['Close: 7 Day Mean'] = df['Close'].rolling(window = 7).mean()

df[['Close', 'Close: 30 Day Mean', 'Close: 15 Day Mean', 'Close: 7 Day Mean']].
 ↪plot(figsize = (12,5))
plt.title('Rolling Averages with Various Windows')
plt.grid(True)
```



[90]:
```
# Expanding

# Expanding windows allow for the calculation of cumulative metrics over time,
# providing insights into the overall behavior or characteristics of the data␣
 ↪up to each point in time.
# This can include cumulative sums, averages, or other statistics that show how␣
 ↪a metric has evolved from the start of
# the time series to the current moment. This is crucial in financial analysis␣
 ↪for calculating running totals of sales, stock prices,
# or cumulative returns, offering a broader view of performance trends.

# Expanding windows can help in identifying long-term trends and anomalies␣
 ↪within time series data.
# By analyzing how cumulative measures change over time, analysts can spot␣
 ↪trends that may not be apparent with a
# fixed or rolling window approach.

# Expanding windows offer a versatile tool in the analysis of time series data,
```

```
# providing insights that are not just snapshot-based but integrate historical␣
 ↪data up to the present.
# This approach enhances understanding, model development, and decision-making␣
 ↪processes across various applications,
# from finance and marketing to meteorology and beyond.

df['Close'].expanding().mean().plot(figsize = (12,5))
plt.grid(True)
```



[ ]:

[1]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

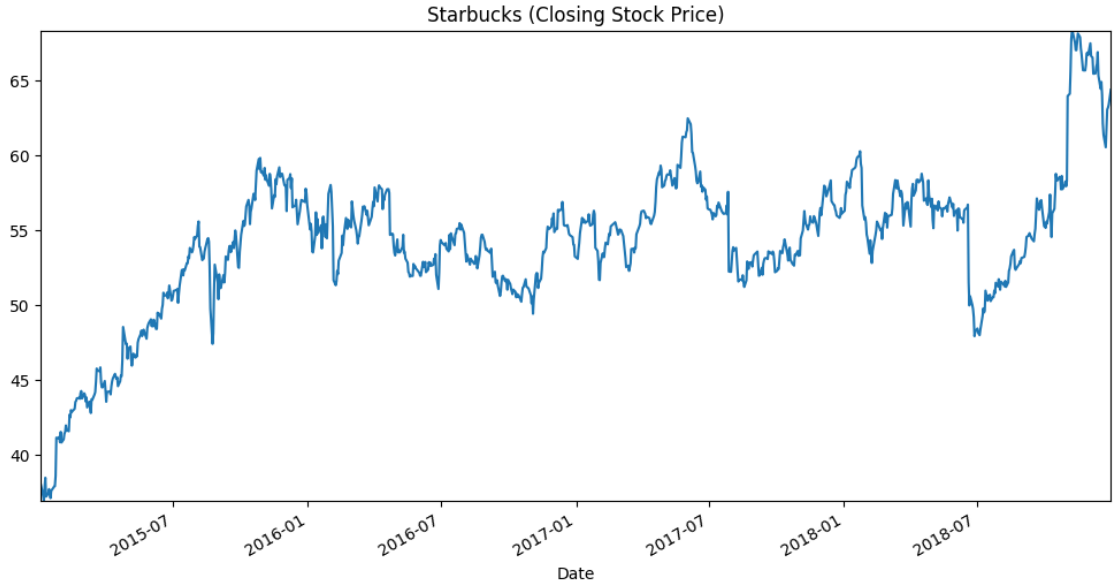
11 Some helpful Time-Series Visualization Customizations

[2]:
```
# Load Data

df = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/starbucks.csv',
                 index_col = 'Date', parse_dates = True)
df.head()
```

[2]:
```
              Close    Volume
Date
```

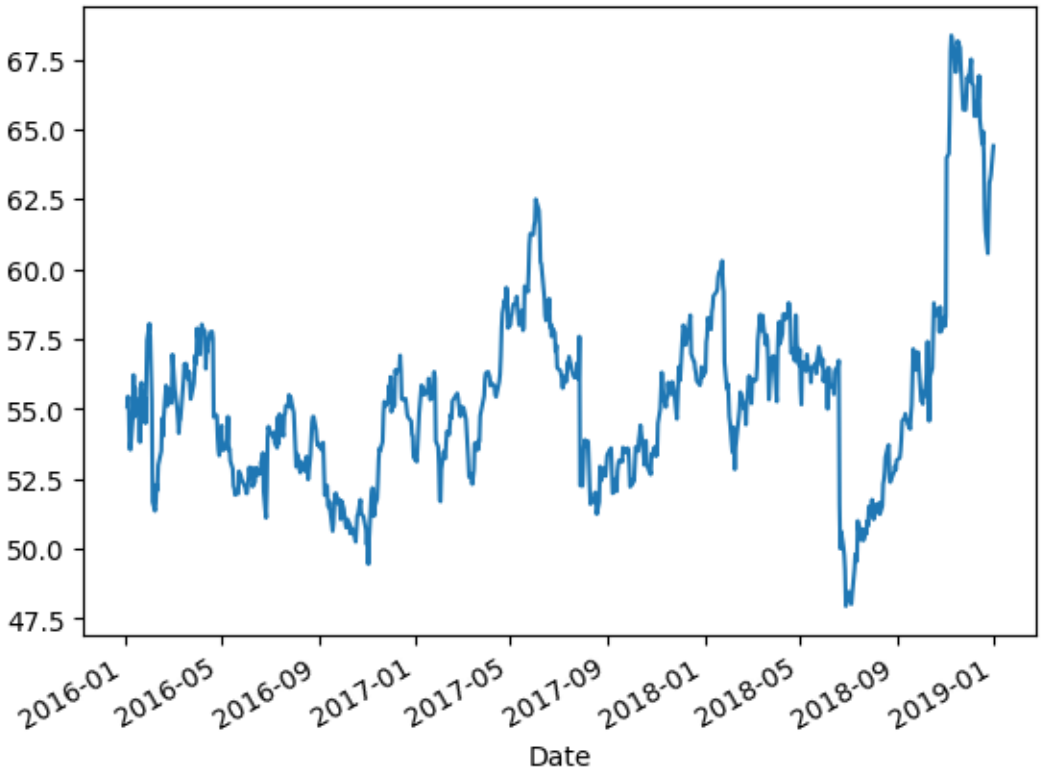
```
2015-01-02  38.0061   6906098
2015-01-05  37.2781  11623796
2015-01-06  36.9748   7664340
2015-01-07  37.8848   9732554
2015-01-08  38.4961  13170548
```

[3]:
```
# Auto Scaling and Layout

ax = df['Close'].plot(figsize = (12,6), title = 'Starbucks (Closing Stock␣
↪Price)')
ax.autoscale(axis = 'both', tight = True)
```



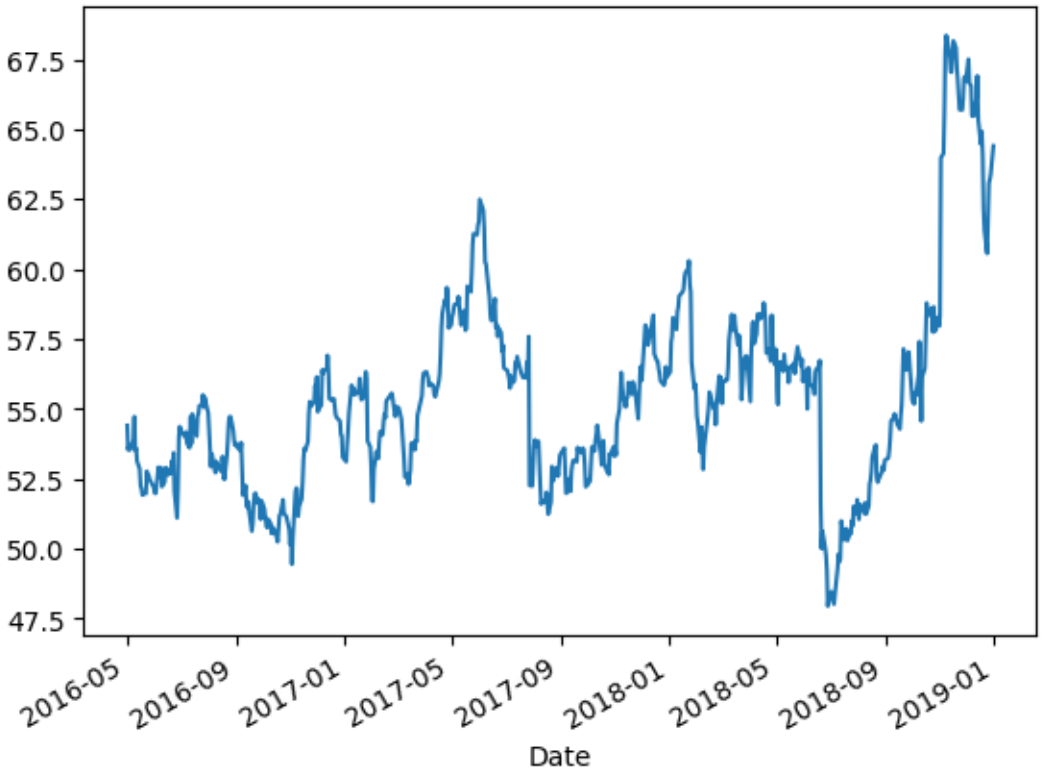
[4]:
```
# Selecting Specific Time Frame (Year)

df['Close']['2016':].plot()
```

[4]:
```
<Axes: xlabel='Date'>
```



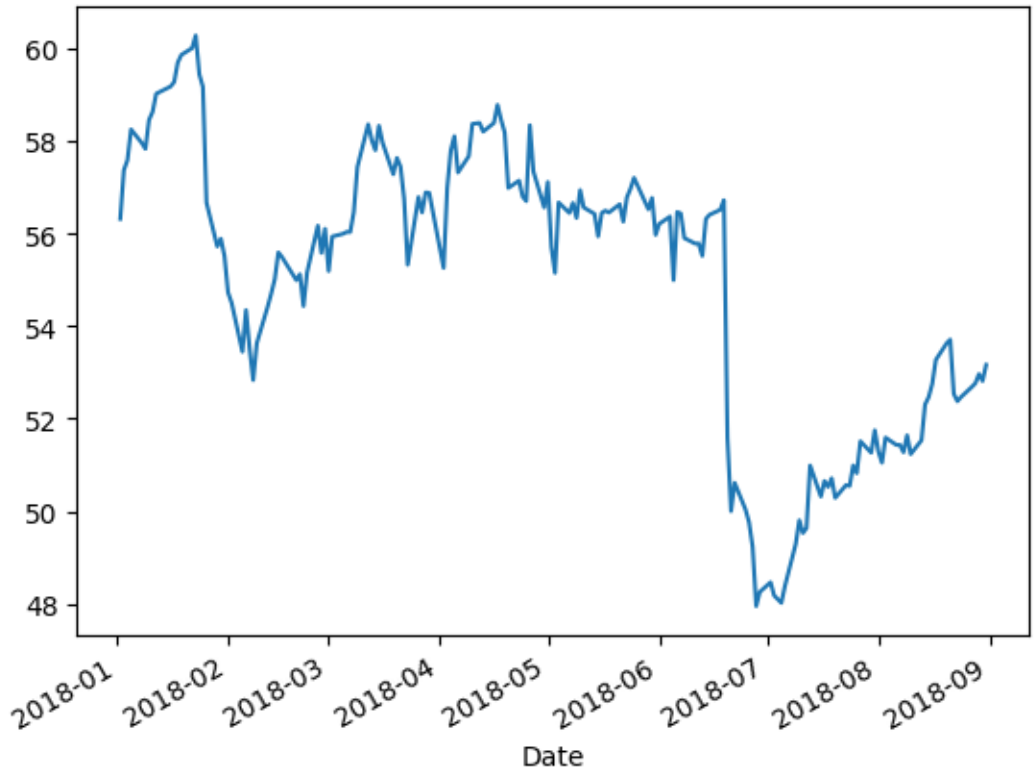
[5]:
```
# Selecting Specific Time Frame (Year, Day)

df['Close']['2016-05':].plot()
```

[5]: <Axes: xlabel='Date'>



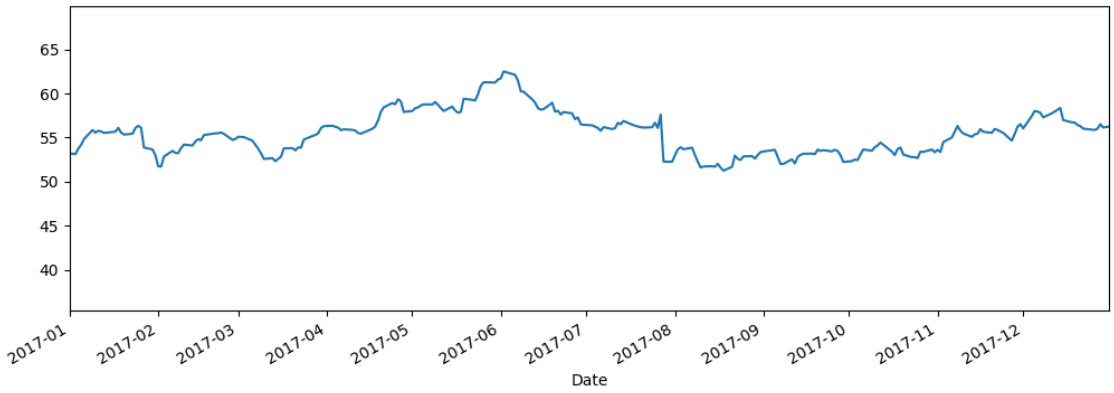
[6]:
```
# Selecting Specific Time Frame (Specific Range)

df['Close']['2018-01-01': '2018-09-01'].plot()
```

[6]: <Axes: xlabel='Date'>



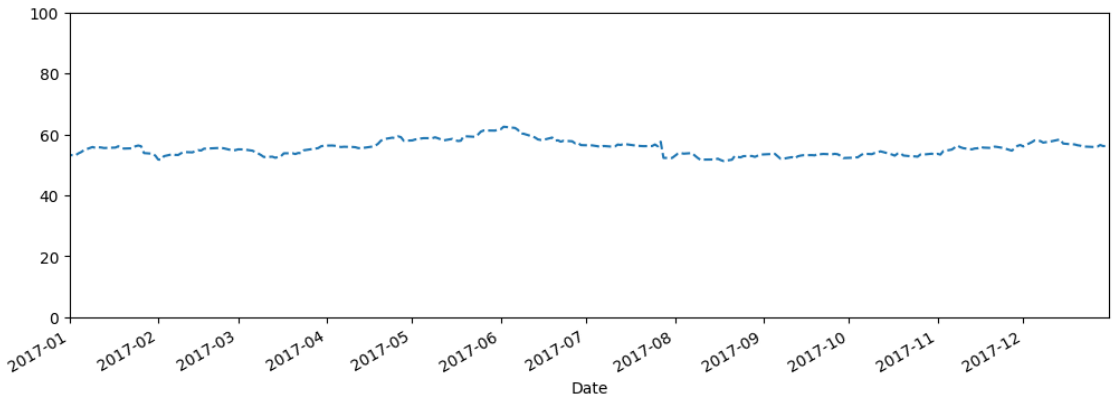
[7]:
```
# Selecting Specific Time Frame (Specific Range using xlim), figsize = (12, 4)

df['Close'].plot(figsize = (12,4), xlim = ['2017-01-01', '2017-12-31']) # Plot level selection
```

[7]: <Axes: xlabel='Date'>



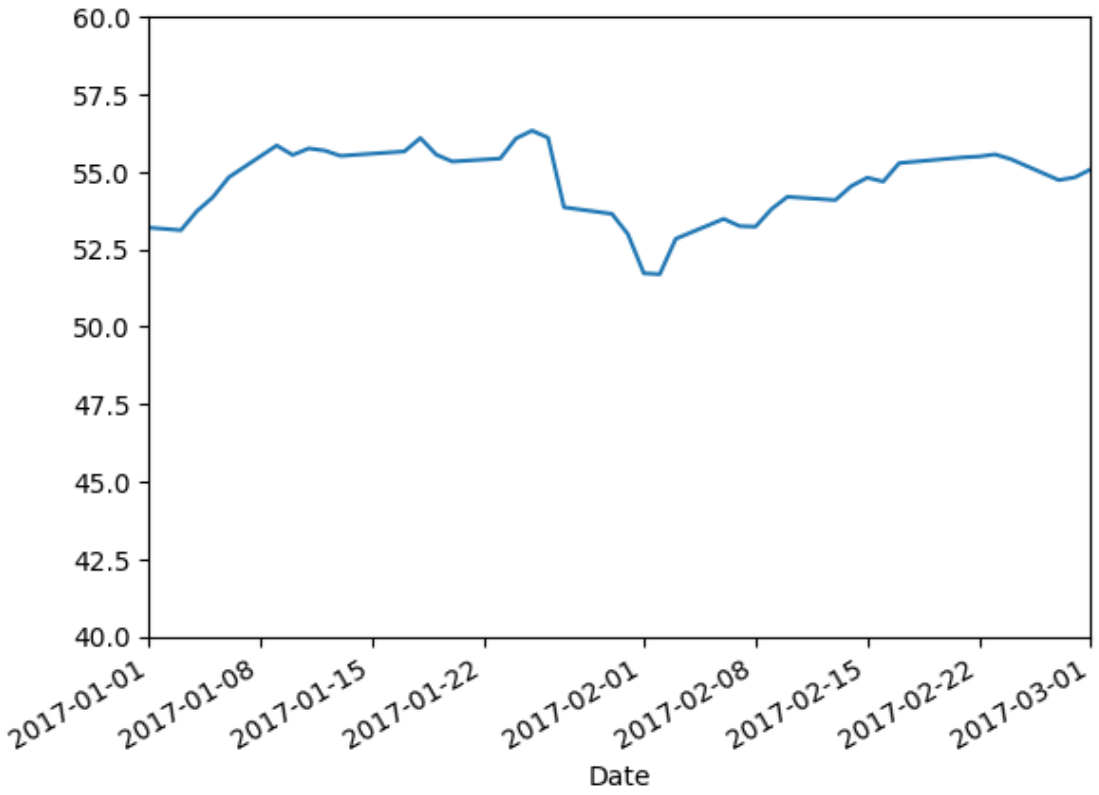
[8]:
```
# Selecting Specific Time Frame (Specific Range using xlim, adding ylim),␣
↪figsize = (12, 4)

df['Close'].plot(figsize = (12,4), xlim = ['2017-01-01', '2017-12-31'],
                 ylim = [0, 100], linestyle = "--") # Plot level selection
```

[8]: <Axes: xlabel='Date'>

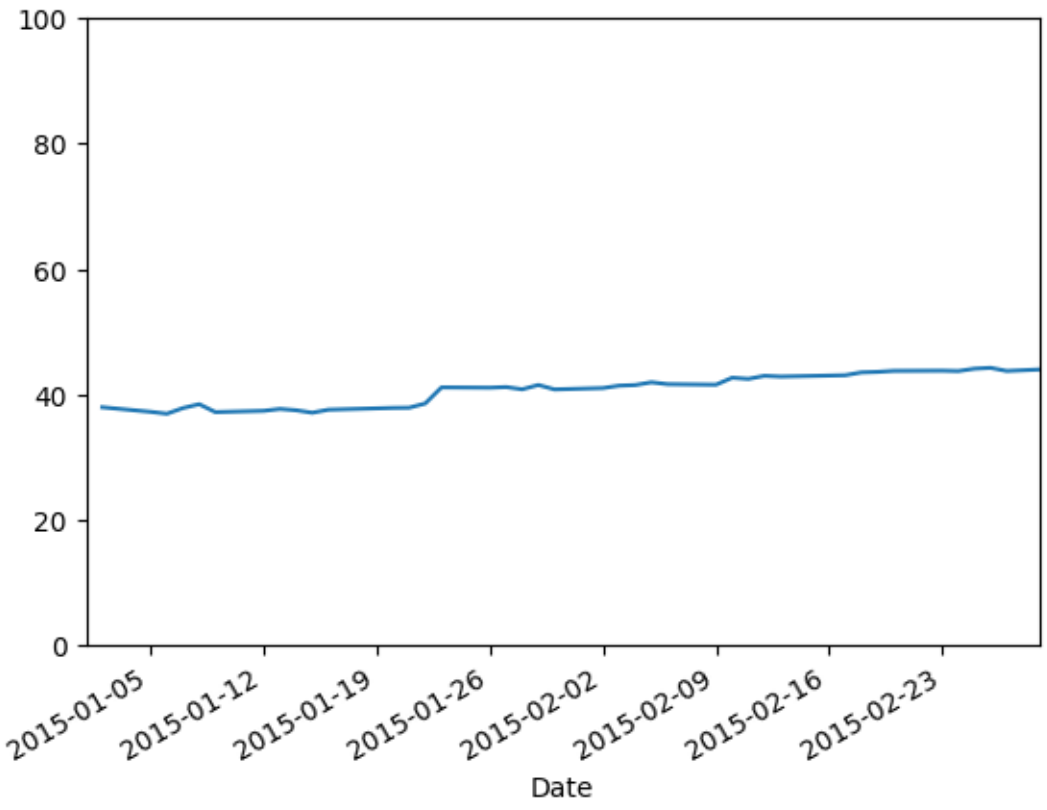


[9]:
```
# xticks, y_ticks, etc.

from matplotlib import dates

df['Close'].plot(xlim = ['2017-01-01', '2017-03-01'], ylim = [40, 60])
```

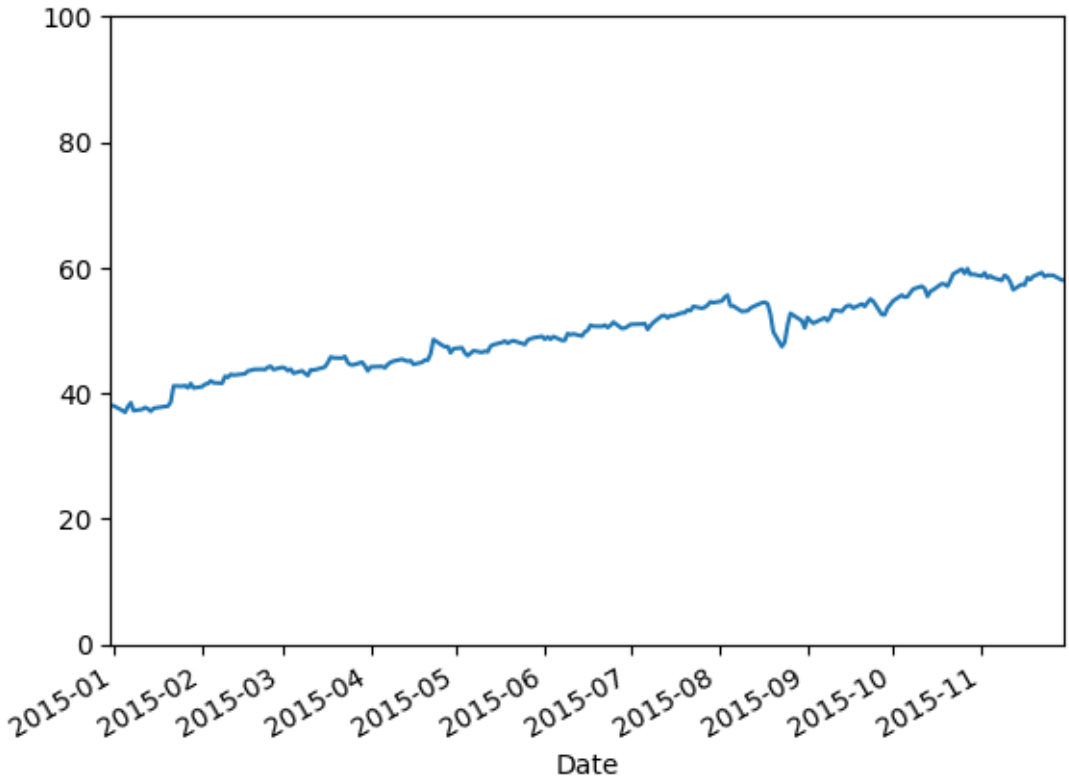
[9]: <Axes: xlabel='Date'>



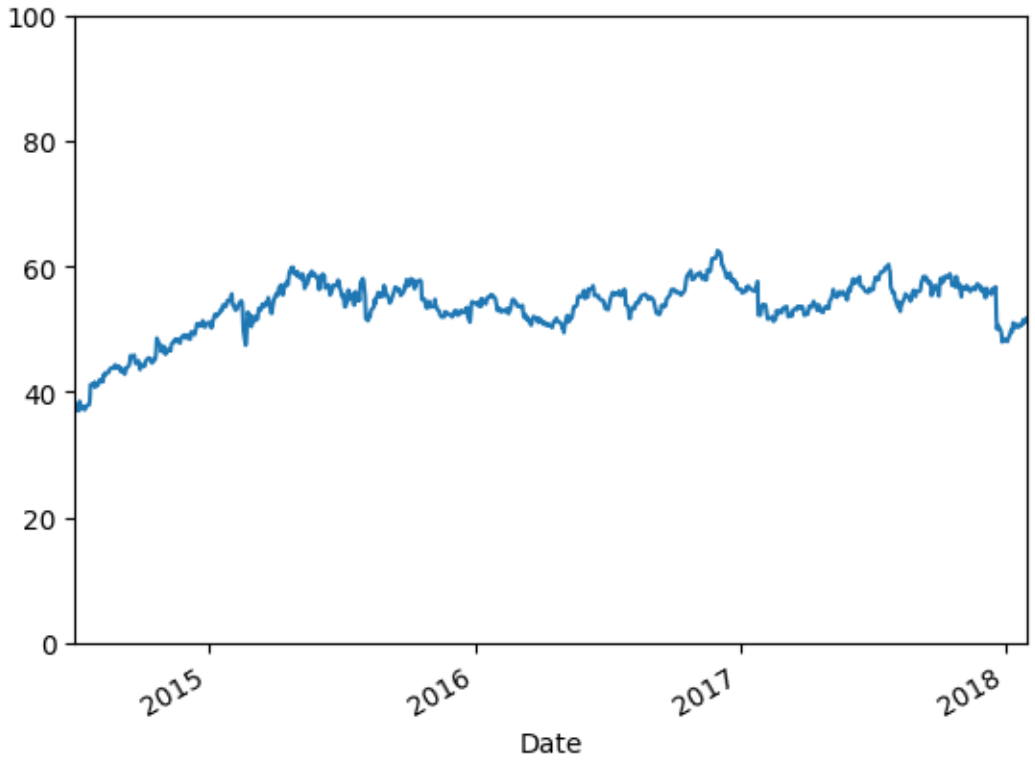
[10]:
```
# set_major_locator

ax = df['Close'].plot(xlim = ['2015-01-01', '2015-03-01'], ylim = [0, 100])
ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday = 0)) # every week,␣
 ↪start on Monday (0)
```



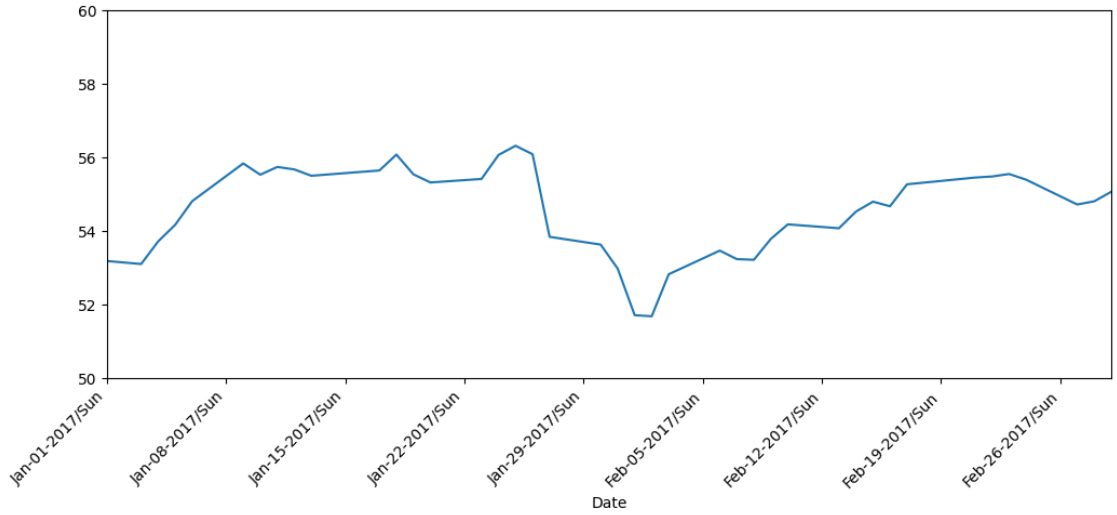
[11]:
```
# set_major_locator

ax = df['Close'].plot(xlim = ['2015-01-01', '2015-12-01'], ylim = [0, 100])
ax.xaxis.set_major_locator(dates.MonthLocator(bymonthday = 2)) # every month,␣
 ↪start at 2nd day
```



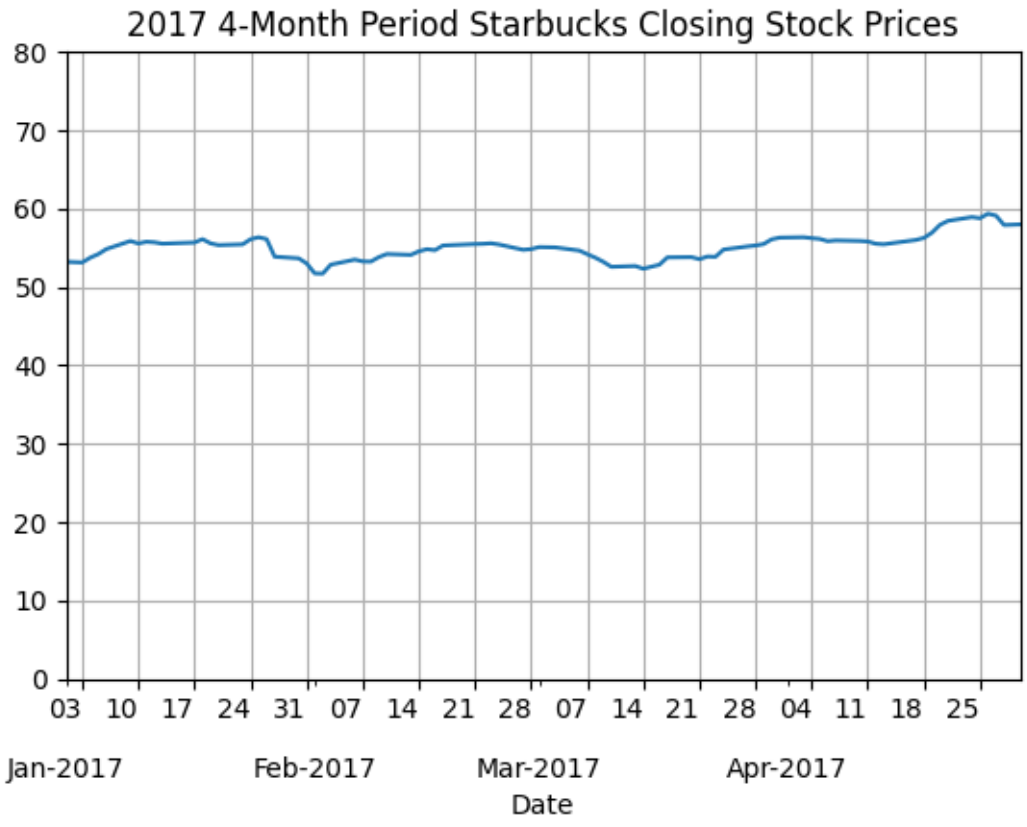
[12]:
```
# set_major_locator

ax = df['Close'].plot(xlim = ['2015-01-01', '2018-08-01'], ylim = [0, 100])
ax.xaxis.set_major_locator(dates.YearLocator(base = 1, month = 7, day = 4)) #␣
 ↪every year, 4th of July each year on x-axis
```



[13]:
```
# set_major_formatter

ax = df['Close'].plot(xlim = ['2017-01-01', '2017-03-01'], ylim = [50, 60], rot␣
 ↪= 45, figsize = (12,5))
ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday = 6))
ax.xaxis.set_major_formatter(dates.DateFormatter('%b-%d-%Y/%a'))
```

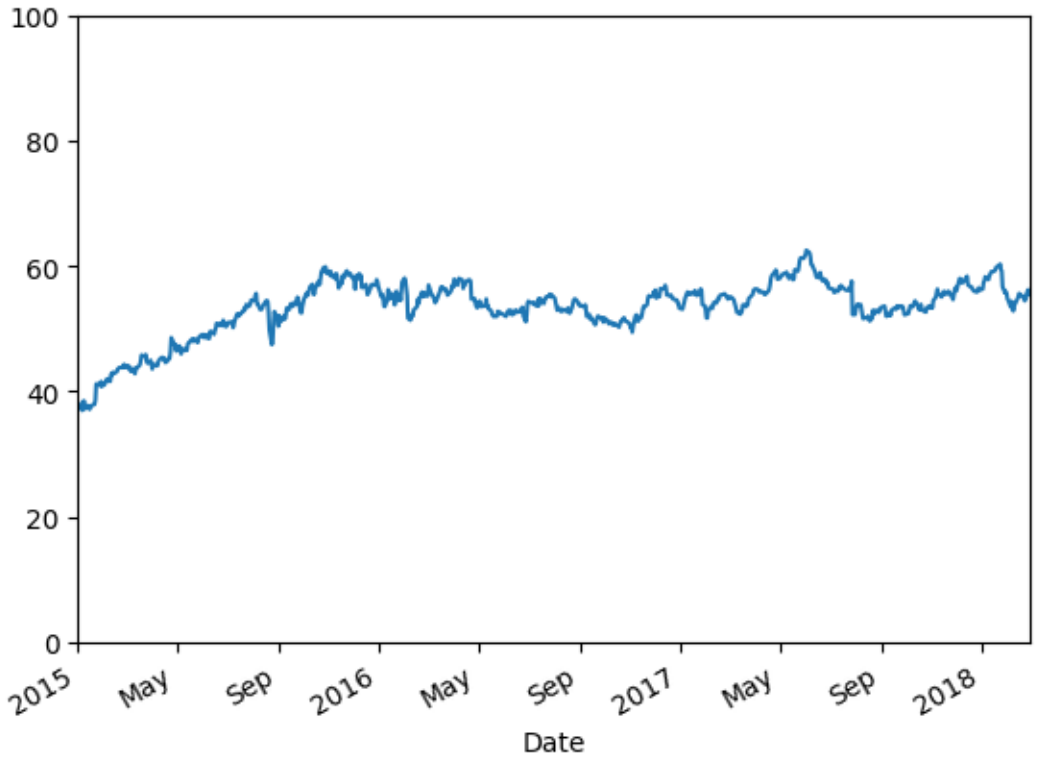


[14]:
```
# Major and Minor Axis

ax = df['Close'].plot(xlim = ['2017-01-01','2017-04-30'], ylim = [0, 80], rot =␣
 ↪0 ,title = '2017 4-Month Period Starbucks Closing Stock Prices')

ax.xaxis.set_major_locator(dates.WeekdayLocator(byweekday = 1)) # Starts at␣
 ↪Tuesday, Monday = 0
ax.xaxis.set_major_formatter(dates.DateFormatter('%d'))

ax.xaxis.set_minor_locator(dates.MonthLocator())
ax.xaxis.set_minor_formatter(dates.DateFormatter('\n\n%b-%Y'))

ax.yaxis.grid(True)
ax.xaxis.grid(True)
```



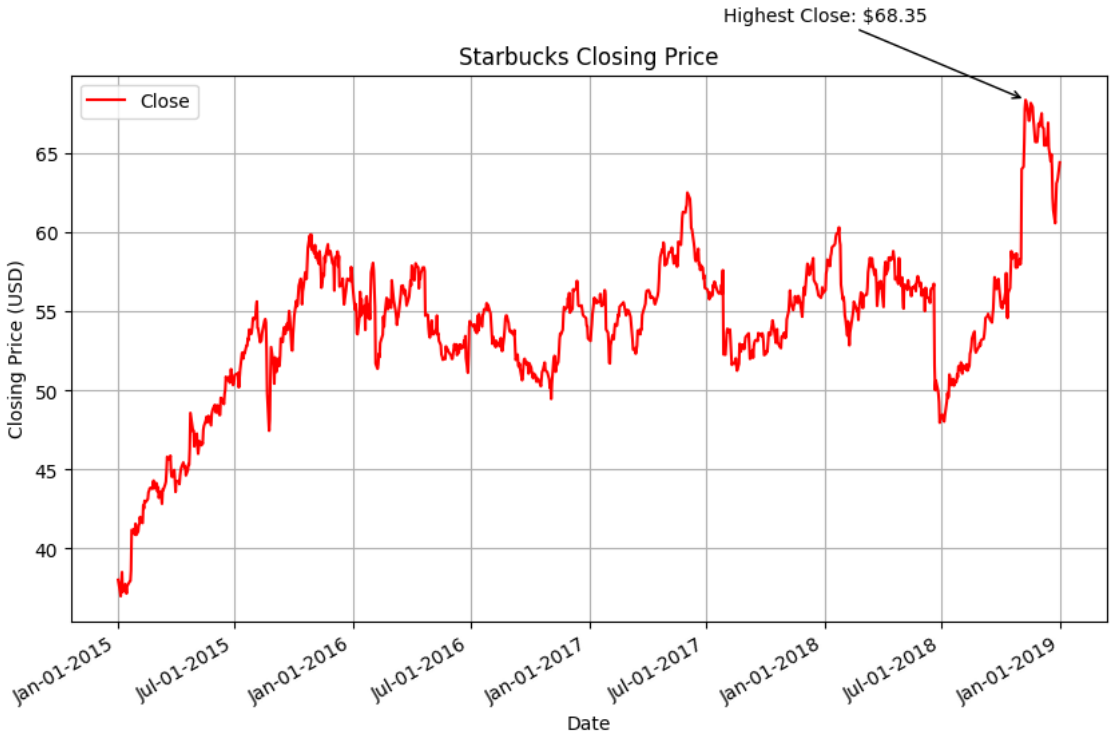
[15]:
```
# ConciseDateFormatter()
```

```
ax = df['Close'].plot(xlim = ['2015-01-01', '2018-03-01'], ylim = [0, 100])
ax.xaxis.set_major_formatter(dates.ConciseDateFormatter(dates.MonthLocator()))␣
 ↪# concise
```



[16]:
```
# Annotation

# Find the date and value of the highest 'Close' price

peak_close_date = df['Close'].idxmax()
peak_close_value = df['Close'].max()

# Plot the 'Close' column with annotation for the highest close value

ax = df['Close'].plot(figsize=(10, 6), color = 'red')
ax.set_title('Starbucks Closing Price')
ax.set_ylabel('Closing Price (USD)')

# Annotate the highest 'Close' price
ax.annotate(f'Highest Close: ${peak_close_value:.2f}', xy = (peak_close_date,␣
 ↪peak_close_value),
            xytext=(peak_close_date - pd.DateOffset(months = 5),␣
 ↪peak_close_value + 5),
```

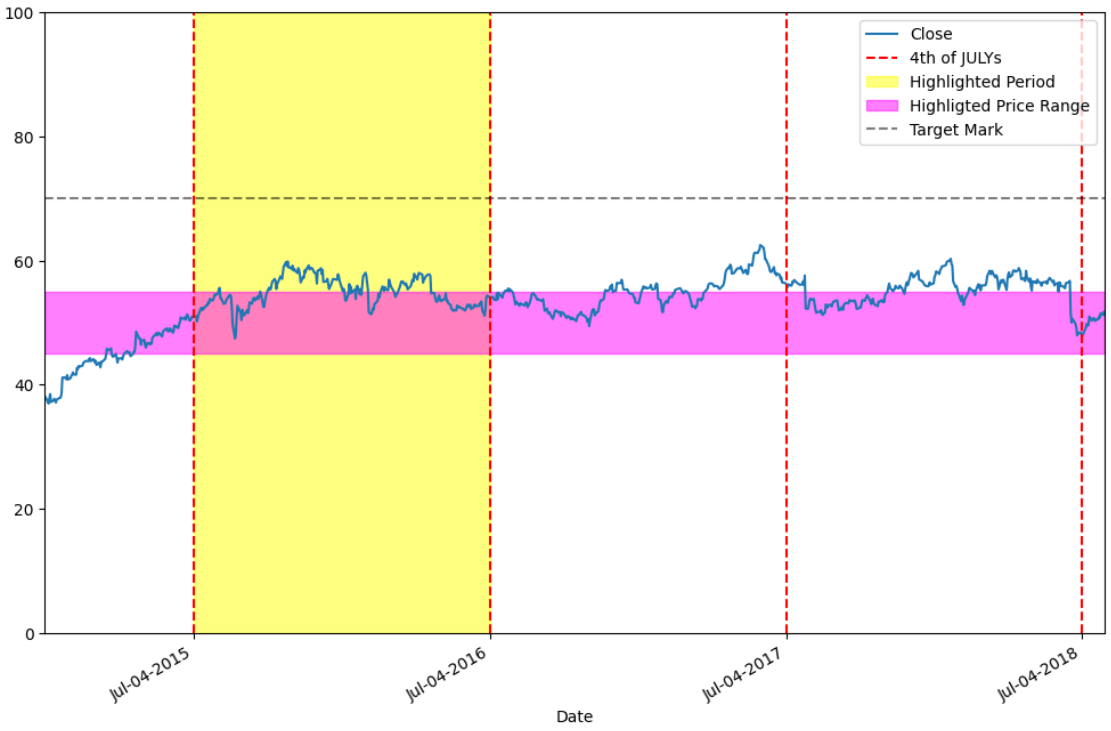
```
            arrowprops=dict(facecolor='black', arrowstyle='->'),
            horizontalalignment='right')

ax.xaxis.set_major_formatter(dates.DateFormatter('%b-%d-%Y'))

ax.legend()
ax.grid(True)
```



[17]:
```
# Emphasis via Lines and Regions

important_dates = pd.to_datetime(['2015-07-04', '2016-07-04', '2017-07-04',␣
 ↪'2018-07-04']) # 4th of JULYs

ax = df['Close'].plot(xlim = ['2015-01-01', '2018-08-01'], ylim = [0, 100],␣
 ↪figsize = (12,8)) # Line Graph with Limits
ax.xaxis.set_major_locator(dates.YearLocator(base = 1, month = 7, day = 4)) #␣
 ↪show every year, 4th of July each year on x-axis
ax.xaxis.set_major_formatter(dates.DateFormatter('%b-%d-%Y')) # Format of Dates␣
 ↪on x-axis

for day in important_dates:
```

```
    ax.axvline(day, color = 'red', linestyle = '--', label = '4th of JULYs' if␣
 ↪day == important_dates[0] else "") # iterate every 4th of July for each year␣
 ↪(vertical lines)

ax.axvspan('2015-07-04', '2016-07-04', color='yellow', alpha = 0.5, label =␣
 ↪'Highlighted Period') # Vertical Span
ax.axhspan(45, 55, color = 'magenta', alpha = 0.5, label = 'Highligted Price␣
 ↪Range') # Horizontal Span
ax.axhline(70, color = 'black', alpha = 0.5, label = 'Target Mark', linestyle =␣
 ↪'--') # Horizontal Line

ax.legend()
```

[17]: <matplotlib.legend.Legend at 0x79762ce9f0a0>



12 Web Data Scraping

Web Data Scraping is a technique used to extract data from websites. This process involves programmatically accessing web pages and pulling out the information that you need. Web scraping can be used to gather data from websites that do not provide an ***Application Program Interface (API)*** for easy data access or when you need large amounts of data quickly and the site's API limits do not allow for this. Here are the key aspects of web scraping:

1. **Sending a Request:** The first step is to send a request to the web server hosting the website from which data is to be scraped. This request is typically done using HTTP or HTTPS protocols.
2. **Receiving the Response:** The server responds to the request by sending back the requested web page, often in HTML format. Other formats like JSON and XML can also be received depending on the API or web service.
3. **Parsing the Data:** Once the data is received, it needs to be parsed. For HTML, this usually involves using libraries like BeautifulSoup in Python, which allow for easy navigation of the structure of the HTML and extraction of the relevant information.
4. **Data Extraction:** After parsing, the necessary data is extracted. This could be anything from product details on an ecommerce site, stock prices, sports statistics, or any other information available on the web.

13 Imports

[ ]:
```python
import pandas as pd
import numpy as np
import requests
import warnings
import matplotlib.pyplot as plt
from bs4 import BeautifulSoup

warnings.filterwarnings("ignore")
```

13.1 BeautifulSoup Methods (Common)

Method	Description
`.find()`	Finds the **first** matching element
`.find_all()`	Finds **all** matching elements
`.find_next()`	Finds the **next** matching element
`.find_previous()`	Finds the **previous** matching element
`.find_next_sibling()`	Finds the **next** sibling element
`.find_previous_sibling()`	Finds the **previous** sibling element
`.find_parents()`	Finds **all** parent elements
`.find_parent()`	Finds the **direct** parent element
`.get_text()`	Extracts text inside a tag
`.decompose()`	Removes an element from the HTML
`.replace_with()`	Replaces an element with new content
`.select()`	Finds **multiple** elements using CSS selectors
`.select_one()`	Finds the **first** element using CSS selectors
`.get()`	Retrieves an **attribute value**
`.has_attr()`	Checks if an element has an **attribute**

13.2 Web Scrape Basic Demo

[ ]:
```
# Link to URL

url = 'https://renatomaaliw3.github.io/scrape_demo.html'
```

[ ]:
```
# Send requests

# When you're making HTTP requests for web scraping, many websites check the␣
 ↪"User-Agent" header
# to see what kind of client is making the request.
# By setting this header, you can mimic a real browser, which can help avoid␣
 ↪blocks
# or serve the correct content.

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36'}
```

[ ]:
```
# Response

# This is a function from the requests library that makes an HTTP GET request.
# GET requests are used to retrieve data from a specified resource.

response = requests.get(url = url, headers = headers)
```

[ ]:
```
# Create a bs4 object to parse the HTML content

# This contains the raw bytes of the HTML content returned by your requests.
 ↪get() call.
# Using .content ensures that you get the raw data, which is useful for parsing.

# This tells BeautifulSoup to parse the HTML using Python's built-in HTML␣
 ↪parser.
# The parser converts the raw HTML into a structured format (a parse tree) that␣
 ↪you
# can easily navigate and search with methods like .find(), .find_all(), and .
 ↪select().

soup = BeautifulSoup(response.content, 'html.parser')
```

[ ]:
```
# .find()

# output = soup.find('table')
# output = soup.find('table', id = 'table-2')
# output = soup.find('td', class_ = 'vip')
output = soup.find('td', attrs = {'data-occupation': 'ceo'})
```

```
output
```

[ ]:
```
<td data-occupation="ceo">Ariel Buckner</td>
```

[ ]:
```
# .find_all()

# output = soup.find_all('th')
# output = soup.find_all('th', class_ = 'name') # or used attrs

# target_ol = soup.find('ol', class_ = 'list level2')
# output = target_ol.find_all('span', recursive = False)

# target_ol = soup.find('ol', class_ = 'list level1')
# output = target_ol.find_all('span', string = ' LEVEL 2 ')

# import re
# target_ol = soup.find('ol', class_ = 'list level1')
# output = target_ol.find_all('span', string = re.compile('LEVEL 2'))

# target_ol = soup.find('ol', class_ = 'list level1')
# output = target_ol.find_all('ol', class_ = 'list level3', limit = 2)

output
```

[ ]:
```
<td data-occupation="ceo">Ariel Buckner</td>
```

[ ]:
```
# .select()

# output = soup.select('ol')
# output = soup.select('li > span')
# output = soup.select('li#item-304')
# output = soup.select('ol#list-837 li > span.red')
# output = soup.select('li', attrs = {'data-code': 'E5F6'})
# output = soup.select('ol.list.level3 > li:first-child')
# output = soup.select('span[class ="link red"]')

output
```

[ ]:
```
<td data-occupation="ceo">Ariel Buckner</td>
```

[ ]:
```
# .select_one()

# output = soup.select_one('ol.list.level3 > li')
# output = soup.select_one('ol span')
# output = soup.select_one('table span')

output
```

[ ]: <td data-occupation="ceo">Ariel Buckner</td>

[ ]:
```
# .children()

ol_list_level3 = soup.find('ol', class_ = 'list level3')

for child in ol_list_level3.children:

  print(child)
```

```
<li class="entry gamma" data-code="C3D4" id="item-867">
<span class="link green"> Third Level A.1 </span>
</li>


<li class="entry delta" data-code="E5F6" id="item-145">
<span class="link purple"> Third Level A.2 </span>
</li>
```

[ ]:
```
# .descendants()

ol_list_level3 = soup.find('ol', class_ = 'list level3')

for child in ol_list_level3.descendants:

  print(child)
```

```
<li class="entry gamma" data-code="C3D4" id="item-867">
<span class="link green"> Third Level A.1 </span>
</li>


<span class="link green"> Third Level A.1 </span>
 Third Level A.1




<li class="entry delta" data-code="E5F6" id="item-145">
<span class="link purple"> Third Level A.2 </span>
</li>
```

```
<span class="link purple"> Third Level A.2 </span>
 Third Level A.2
```

[ ]:
```
# .find_next_sibling()

table_1 = soup.select_one('table#table-1')
th_name = table_1.find('th', class_ = 'name')
next_sib = th_name.find_next_sibling()

next_sib
```

[ ]:
```
<th class="email"> Email </th>
```

[ ]:
```
# .find_previous_sibling()

table_1 = soup.select_one('table#table-1')
th_name = table_1.find('th', class_ = 'email')
next_sib = th_name.find_previous_sibling('th')

next_sib
```

[ ]:
```
<th class="name"> Name </th>
```

[ ]:
```
# .get_text()

table_1 = soup.select_one('table#table-1')
th_name = table_1.find('th', class_ = 'name')
th_name.get_text().strip()
```

[ ]:
```
'Name'
```

[ ]:
```
# .get_text() - long method

data = []
table_rows = soup.select('table#table-1 > tr')[1:]

for row in table_rows:

  cols = []
  tds = row.find_all('td')

  for td in tds:
```

```
    text = td.get_text()
    cols.append(text)

  data.append(cols)

data
```

[ ]:
```
[["Geoffrey O'Donnell", 'vitae.semper@protonmail.org', 'Nigeria', 'Flevoland'],
 ['Justin Patrick', 'orci.ut.sagittis@outlook.com', 'Peru', 'Punjab'],
 ['Ariel Buckner', 'nunc@google.net', 'Vietnam', 'North Chungcheong'],
 ['Imani Faulkner', 'hendrerit@yahoo.com', 'Philippines', 'Bicol'],
 ['Colorado Hampton', 'eros.nam@hotmail.couk', 'Brazil', 'Katsina']]
```

[ ]:
```
# .get_text() - recommended

data = []
table_rows = soup.select('table#table-1 > tr')[1:]

for row in table_rows:

  cols = [col.get_text(strip = True) for col in row.find_all('td')]
  data.append(cols)

data
```

[ ]:
```
[["Geoffrey O'Donnell", 'vitae.semper@protonmail.org', 'Nigeria', 'Flevoland'],
 ['Justin Patrick', 'orci.ut.sagittis@outlook.com', 'Peru', 'Punjab'],
 ['Ariel Buckner', 'nunc@google.net', 'Vietnam', 'North Chungcheong'],
 ['Imani Faulkner', 'hendrerit@yahoo.com', 'Philippines', 'Bicol'],
 ['Colorado Hampton', 'eros.nam@hotmail.couk', 'Brazil', 'Katsina']]
```

[ ]:
```
# .attrs

div_tag = soup.find('li', id = 'item-592')
div_tag.attrs
```

[ ]:
```
{'id': 'item-592', 'class': ['entry', 'alpha'], 'data-code': 'X7Z3'}
```

[ ]:
```
# sample scenario

if (div_tag.attrs.get('class')[0] == 'entry'):

  print(True)

else:

  print(False)
```

True

[ ]:
```
# sample scenario (in)

classes = div_tag.attrs.get('class')

if ('entry' in classes and 'alpha' in classes):

  print(True)

else:

  print(False)
```

True

[ ]:
```
# sample scenario (set)


if (set(div_tag.attrs.get('class')) == {'entry', 'alpha'}):

  print(True)

else:

  print(False)
```

True

[ ]:
```
# .get

div_tag = soup.find('li', id = 'item-592')
div_tag.get('data-code')
```

[ ]:
```
'X7Z3'
```

13.3 Web Scrape Worldometer

[ ]:
```
# URL

url = 'https://www.worldometers.info/world-population/world-population-by-year/
 ↪' # Worldometer

# Send requests

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36'}
```

```
# Response

response = requests.get(url = url, headers = headers)

# Create a bs4 object to parse the HTML content

soup = BeautifulSoup(response.content, 'html.parser')
```

[ ]:
```
## Find the table containing the data

table_rows = soup.select('tbody > tr')
```

[ ]:
```
# Define column names

data = []
column_names = ['Year', 'Population', 'Yearly Change', 'Net Change', 'Density␣
↪(P/km sq)']

for row in table_rows:

  cols = [col.get_text().replace(',', '').replace('%', '').strip() for col in␣
↪row.find_all('td')]
  data.append(cols)
```

[ ]:
```
# Convert to DataFrame

df = pd.DataFrame(data, columns = column_names)
df = df.iloc[:79,:]
df
```

[ ]:
```
    Year  Population Yearly Change Net Change Density (P/km sq)
0   2024  8161972572          0.87   70237642                55
1   2023  8091734930          0.88   70327738                54
2   2022  8021407192          0.84   66958801                54
3   2021  7954448391          0.86   67447099                53
4   2020  7887001292          0.97   75707594                53
..   …          …            …         …                   …
74  1900  1600000000
75  1850  1200000000
76  1804  1000000000
77  1760   770000000
78  1700   610000000

[79 rows x 5 columns]
```

[ ]:
```
# Fill blanks
```

```
# Replace cells that are empty or just whitespace with NaN
df = df.replace(r'^\s*$', 0, regex = True)
df.tail(25)
```

[ ]:

	Year	Population	Yearly Change	Net Change	Density (P/km sq)
54	1970	3694683794	2.08	75192215	25
55	1969	3619491579	2.10	74304328	24
56	1968	3545187251	2.07	71774371	24
57	1967	3473412880	2.04	69371755	23
58	1966	3404041125	2.08	69507422	23
59	1965	3334533703	2.15	70046364	22
60	1964	3264487339	2.25	71679511	22
61	1963	3192807828	2.22	69433513	21
62	1962	3123374315	1.91	58504640	21
63	1961	3064869675	1.64	49398781	21
64	1960	3015470894	1.67	49520543	20
65	1959	2965950351	1.88	54700680	20
66	1958	2911249671	2.06	58631334	20
67	1957	2852618337	2.05	57208343	19
68	1956	2795409994	2.01	55196202	19
69	1955	2740213792	2.02	54318932	18
70	1954	2685894860	1.97	51788625	18
71	1953	2634106235	1.94	50019896	18
72	1952	2584086339	1.86	47159304	17
73	1927	2000000000	0	0	0
74	1900	1600000000	0	0	0
75	1850	1200000000	0	0	0
76	1804	1000000000	0	0	0
77	1760	770000000	0	0	0
78	1700	610000000	0	0	0

```
# Data Cleaning (Numerical Values)

# Convert Numerical Values to float data type

df_float = df.loc[:,'Population':].astype(float)
df_float.tail()
```

[ ]:

	Population	Yearly Change	Net Change	Density (P/km sq)
74	1.600000e+09	0.0	0.0	0.0
75	1.200000e+09	0.0	0.0	0.0
76	1.000000e+09	0.0	0.0	0.0
77	7.700000e+08	0.0	0.0	0.0
78	6.100000e+08	0.0	0.0	0.0

```
# Data Cleaning (Dates)
```

[ ]:

```
# Convert Numerical Values to float data type

df_date = pd.to_datetime(df['Year']).astype('datetime64[ns]')
```

[ ]:
```
# Concatenate

df = pd.concat([df_date, df_float], axis = 1)
df = df.sort_values('Year', ascending = True)
df
```

[ ]:

	Year	Population	Yearly Change	Net Change	Density (P/km sq)
78	1700-01-01	6.100000e+08	0.00	0.0	0.0
77	1760-01-01	7.700000e+08	0.00	0.0	0.0
76	1804-01-01	1.000000e+09	0.00	0.0	0.0
75	1850-01-01	1.200000e+09	0.00	0.0	0.0
74	1900-01-01	1.600000e+09	0.00	0.0	0.0
..	…	…	…	…	…
4	2020-01-01	7.887001e+09	0.97	75707594.0	53.0
3	2021-01-01	7.954448e+09	0.86	67447099.0	53.0
2	2022-01-01	8.021407e+09	0.84	66958801.0	54.0
1	2023-01-01	8.091735e+09	0.88	70327738.0	54.0
0	2024-01-01	8.161973e+09	0.87	70237642.0	55.0

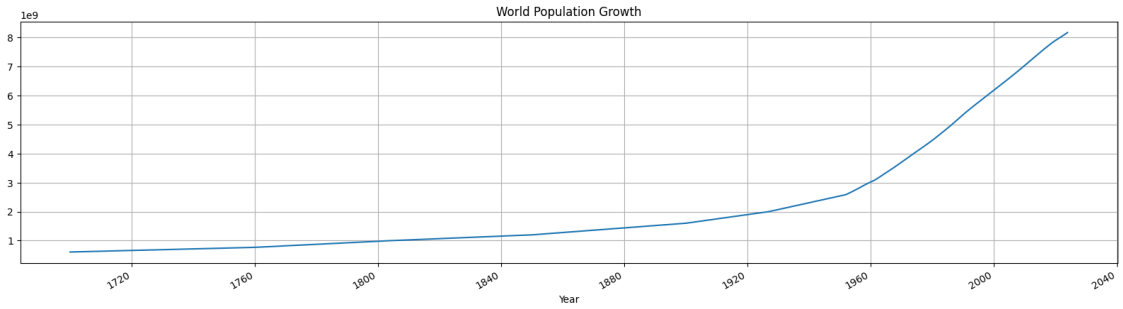
[79 rows x 5 columns]

[ ]:
```
# Set Date to Index

df = df.set_index('Year')
df.tail()
```

[ ]:

Year	Population	Yearly Change	Net Change	Density (P/km sq)
2020-01-01	7.887001e+09	0.97	75707594.0	53.0
2021-01-01	7.954448e+09	0.86	67447099.0	53.0
2022-01-01	8.021407e+09	0.84	66958801.0	54.0
2023-01-01	8.091735e+09	0.88	70327738.0	54.0
2024-01-01	8.161973e+09	0.87	70237642.0	55.0

[ ]:
```
df['Population'].plot(figsize = (20,5))

plt.title('World Population Growth')
plt.grid()
plt.show()
```



13.4 Web Scrape Yahoo Finance

[ ]:
```
# URL

crypto = 'BTC'

url = f'https://finance.yahoo.com/quote/{crypto}-USD/history/?
 ↪period1=1410912000&period2=1740124395' #

# Send requests

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36'}

# Response

response = requests.get(url = url, headers = headers)

# Create a bs4 object to parse the HTML content

soup = BeautifulSoup(response.content, 'html.parser')
```

[ ]:
```
## Find the table containing the data

# Find the table with the stats

table_rows = soup.select('tbody > tr.yf-1jecxey')
```

[ ]:
```
# Define column names

data = []
column_names = ["Date", "Open", "High", "Low", "Close", "Adj Close", "Volume"]

for row in table_rows:

```

```
    cols = [col.get_text().replace(',', '') for col in row.find_all('td')]
    data.append(cols)
```

[ ]:
```
# Convert to DataFrame

df = pd.DataFrame(data, columns = column_names)
df.head()
```

[ ]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	Feb 21 2025	98340.67	99497.97	94852.96	96125.55	96125.55	49608706470
1	Feb 20 2025	96632.68	98767.20	96442.67	98333.94	98333.94	31668022771
2	Feb 19 2025	95532.53	96855.59	95011.97	96635.61	96635.61	28990872862
3	Feb 18 2025	95773.81	96695.38	93388.84	95539.55	95539.55	37325720482
4	Feb 17 2025	96179.01	97032.23	95243.55	95773.38	95773.38	27336550690

[ ]:
```
# Data Cleaning (Numerical Values)

# Convert Numerical Values to float data type

df_float = df.loc[:, 'Open':].astype(float)
df_float.head()
```

[ ]:

	Open	High	Low	Close	Adj Close	Volume
0	98340.67	99497.97	94852.96	96125.55	96125.55	4.960871e+10
1	96632.68	98767.20	96442.67	98333.94	98333.94	3.166802e+10
2	95532.53	96855.59	95011.97	96635.61	96635.61	2.899087e+10
3	95773.81	96695.38	93388.84	95539.55	95539.55	3.732572e+10
4	96179.01	97032.23	95243.55	95773.38	95773.38	2.733655e+10

[ ]:
```
# Data Cleaning (Dates)

# Convert Numerical Values to float data type

df_date = pd.to_datetime(df['Date']).astype('datetime64[ns]')
df_date.head()
```

[ ]:
```
0   2025-02-21
1   2025-02-20
2   2025-02-19
3   2025-02-18
4   2025-02-17
Name: Date, dtype: datetime64[ns]
```

[ ]:
```
# Concatenate

df = pd.concat([df_date, df_float], axis = 1)
df = df.sort_values('Date', ascending = True)
```

```
df
```

[ ]:
```
            Date      Open      High       Low     Close  Adj Close  \
3810 2014-09-17    465.86    468.17    452.42    457.33     457.33
3809 2014-09-18    456.86    456.86    413.10    424.44     424.44
3808 2014-09-19    424.10    427.83    384.53    394.80     394.80
3807 2014-09-20    394.67    423.30    389.88    408.90     408.90
3806 2014-09-21    408.08    412.43    393.18    398.82     398.82
…         …         …         …         …         …
4    2025-02-17  96179.01  97032.23  95243.55  95773.38   95773.38
3    2025-02-18  95773.81  96695.38  93388.84  95539.55   95539.55
2    2025-02-19  95532.53  96855.59  95011.97  96635.61   96635.61
1    2025-02-20  96632.68  98767.20  96442.67  98333.94   98333.94
0    2025-02-21  98340.67  99497.97  94852.96  96125.55   96125.55

            Volume
3810  2.105680e+07
3809  3.448320e+07
3808  3.791970e+07
3807  3.686360e+07
3806  2.658010e+07
…              …
4     2.733655e+10
3     3.732572e+10
2     2.899087e+10
1     3.166802e+10
0     4.960871e+10

[3811 rows x 7 columns]
```

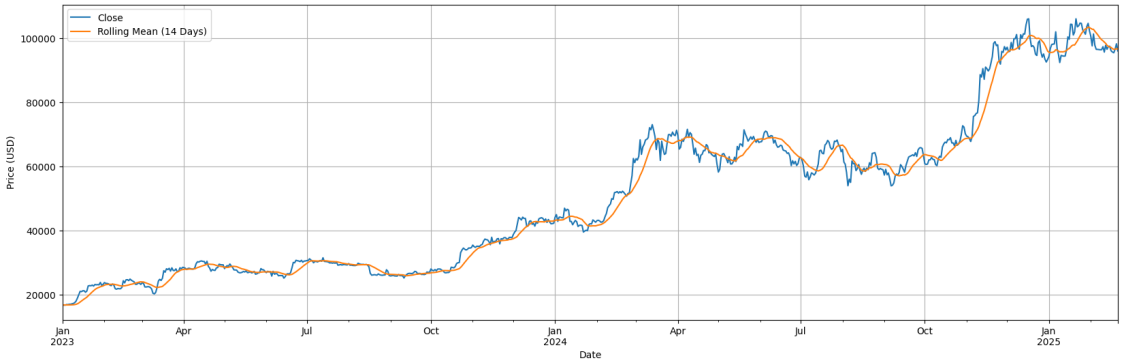
[ ]:
```
# Set Date to Index

df = df.set_index('Date')
df
```

[ ]:
```
                Open      High       Low     Close  Adj Close        Volume
Date
2014-09-17    465.86    468.17    452.42    457.33     457.33  2.105680e+07
2014-09-18    456.86    456.86    413.10    424.44     424.44  3.448320e+07
2014-09-19    424.10    427.83    384.53    394.80     394.80  3.791970e+07
2014-09-20    394.67    423.30    389.88    408.90     408.90  3.686360e+07
2014-09-21    408.08    412.43    393.18    398.82     398.82  2.658010e+07
…                ...       …         …         …          …             …
2025-02-17  96179.01  97032.23  95243.55  95773.38   95773.38  2.733655e+10
2025-02-18  95773.81  96695.38  93388.84  95539.55   95539.55  3.732572e+10
2025-02-19  95532.53  96855.59  95011.97  96635.61   96635.61  2.899087e+10
2025-02-20  96632.68  98767.20  96442.67  98333.94   98333.94  3.166802e+10
```

```
2025-02-21   98340.67   99497.97   94852.96   96125.55    96125.55   4.960871e+10

[3811 rows x 6 columns]
```

[ ]:
```
df['Close']['2023':].plot(figsize = (20,5)).autoscale()
plt.title(f'{crypto} Close Price')
plt.grid()
plt.show()
```

[ ]:
```
# 2023 Data Onwards

df_concise = df.loc['2023':, :]
df_concise['Rolling Mean (14 Days)'] = df['Close'].rolling(window = 14).mean()
df_concise[['Close', 'Rolling Mean (14 Days)']].plot(figsize = (20, 6), ylabel␣
↪= 'Price (USD)')

plt.grid()
plt.show()
```



14 Web Scrape ESPN NBA Data

[ ]:
```
# URL

url = 'https://www.espn.ph/nba/table/_/group/league'

# Send requests

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'}

# Response

response = requests.get(url = url, headers = headers)

# Create a bs4 object to parse the HTML content

soup = BeautifulSoup(response.content, 'html.parser')
```

[ ]:
```
# Find the table containing teams

table_teams = soup.select_one('tbody.Table__TBODY')

# Find the table with the stats

table_data = soup.select_one('div.Table__Scroller tbody.Table__TBODY')
```

[ ]:
```
# Define column names

column_names = ["Wins", "Losses", "WinPCT", "GB", "Home", "Away", "Div",␣
 ↪"Conf", "PPG", "Opp_PPG", "Diff", "Strk", "L10"]
```

[ ]:
```
# Team Names

team = []

for teams in table_teams.find_all('span', class_ = 'hide-mobile'):

  team.append(teams.get_text())

#team
```

[ ]:
```
# Team Stats

stat = []

for tr in table_data.find_all('tr'):
```

```
  row_data = []

  for td in tr.find_all('td'):

    row_data.append(td.get_text())

  stat.append(row_data)

# stat
```

[ ]:
```
# Create Dataframe for Team

df_team = pd.DataFrame(team, columns = ['Team'])
df_team.head()
```

[ ]:
```
                    Team
0    Cleveland Cavaliers
1  Oklahoma City Thunder
2         Boston Celtics
3     Los Angeles Lakers
4         Denver Nuggets
```

[ ]:
```
# Create Dataframe for Stats

df_stats = pd.DataFrame(stat, columns = column_names)
df_stats.head()
```

[ ]:
```
  Wins Losses WinPCT  GB   Home    Away   Div   Conf    PPG Opp_PPG   Diff  \
0   54     10   .844   -   29-4   25-6  11-1   37-7  122.9   111.5  +11.4
1   53     11   .828   1   28-4   24-7  11-3  32-10  119.6   106.6  +13.0
2   46     18   .719   8  22-11   24-7  11-2  32-11  117.0   108.2   +8.8
3   40     22   .645  13   25-7  15-15  11-3  27-12  112.9   111.1   +1.8
4   41     23   .641  13   22-9  19-14   6-5  24-14  121.2   116.5   +4.7

  Strk   L10
0  W14  10-0
1   W7   9-1
2   W4   8-2
3   L1   8-2
4   L1   6-4
```

[ ]:
```
# Concatenate

df = pd.concat([df_team, df_stats], axis = 1)
df
```

[ ]:

	Team	Wins	Losses	WinPCT	GB	Home	Away	Div
0	Cleveland Cavaliers	54	10	.844	-	29-4	25-6	11-1
1	Oklahoma City Thunder	53	11	.828	1	28-4	24-7	11-3
2	Boston Celtics	46	18	.719	8	22-11	24-7	11-2
3	Los Angeles Lakers	40	22	.645	13	25-7	15-15	11-3
4	Denver Nuggets	41	23	.641	13	22-9	19-14	6-5
5	New York Knicks	40	23	.635	13.5	21-11	19-12	10-3
6	Memphis Grizzlies	40	24	.625	14	22-10	18-14	10-5
7	Houston Rockets	39	25	.609	15	21-10	18-14	12-3
8	Milwaukee Bucks	36	27	.571	17.5	21-11	14-16	6-6
9	Indiana Pacers	35	27	.565	18	19-9	15-17	7-4
10	Golden State Warriors	36	28	.563	18	18-13	18-15	2-10
11	Minnesota Timberwolves	37	29	.561	18	18-14	19-15	7-5
12	Detroit Pistons	36	29	.554	18.5	17-14	19-15	4-9
13	LA Clippers	35	29	.547	19	22-10	13-19	7-7
14	Sacramento Kings	33	30	.524	20.5	16-14	17-16	4-8
15	Dallas Mavericks	32	33	.492	22.5	19-15	13-18	7-5
16	Atlanta Hawks	30	34	.469	24	15-16	15-17	7-4
17	Phoenix Suns	30	34	.469	24	18-13	12-21	8-4
18	Orlando Magic	30	35	.462	24.5	18-16	12-19	8-3
19	Miami Heat	29	34	.460	24.5	15-14	13-20	7-4
20	Portland Trail Blazers	28	37	.431	26.5	16-15	12-22	5-9
21	San Antonio Spurs	26	36	.419	27	15-14	10-21	3-10
22	Chicago Bulls	26	38	.406	28	11-22	15-16	3-11
23	Philadelphia 76ers	22	41	.349	31.5	12-21	10-20	3-10
24	Brooklyn Nets	21	42	.333	32.5	9-21	12-21	3-8
25	Toronto Raptors	21	43	.328	33	14-20	7-23	4-8
26	New Orleans Pelicans	17	48	.262	37.5	11-21	6-27	3-12
27	Charlotte Hornets	15	48	.238	38.5	10-24	5-24	0-11
28	Utah Jazz	15	49	.234	39	8-24	7-25	2-9
29	Washington Wizards	13	49	.210	40	7-25	6-23	6-6

	Conf	PPG	Opp_PPG	Diff	Strk	L10
0	37-7	122.9	111.5	+11.4	W14	10-0
1	32-10	119.6	106.6	+13.0	W7	9-1
2	32-11	117.0	108.2	+8.8	W4	8-2
3	27-12	112.9	111.1	+1.8	L1	8-2
4	24-14	121.2	116.5	+4.7	L1	6-4
5	28-13	116.9	112.8	+4.1	L3	5-5
6	23-16	122.7	116.7	+6.0	W2	4-6
7	24-16	113.2	109.3	+3.9	W2	5-5
8	26-18	114.6	112.5	+2.1	L2	7-3
9	20-19	116.6	115.3	+1.3	L2	6-4
10	20-19	113.3	111.2	+2.1	W4	9-1
11	26-18	112.8	109.2	+3.6	W5	6-4
12	24-20	114.7	113.1	+1.6	W1	7-3
13	21-21	110.8	108.7	+2.1	W3	4-6

14	23-21	116.8	114.9	+1.9	L1	6-4
15	22-23	114.5	114.2	+0.3	L5	3-7
16	22-18	117.1	119.7	-2.6	W2	4-6
17	20-22	114.4	116.5	-2.1	W1	4-6
18	24-20	104.6	106.2	-1.6	W1	4-6
19	18-21	110.1	110.5	-0.4	L3	4-6
20	15-29	110.5	114.4	-3.9	L3	5-5
21	18-24	113.4	115.8	-2.4	L2	3-7
22	21-23	116.6	120.6	-4.0	W2	4-6
23	14-25	109.3	114.1	-4.8	W1	2-8
24	12-27	105.1	111.6	-6.5	L7	2-8
25	13-29	110.6	116.0	-5.4	L1	4-6
26	11-32	110.8	119.1	-8.3	L4	4-6
27	8-32	105.3	113.0	-7.7	W1	1-9
28	7-36	112.6	119.8	-7.2	L5	2-8
29	10-26	108.5	120.5	-12.0	W2	4-6

15 Introduction to Statsmodels

Statsmodels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration. An extensive list of result statistics are available for each estimator. The results are tested against existing statistical packages to ensure that they are correct. The package is released under the open source Modified BSD (3-clause) license. The online documentation is hosted at statsmodels.org. The statsmodels version used in the development of this course is 0.9.0.

For Further Reading:

Statsmodels Tutorial: Time Series Analysis

Let's walk through a very simple example of using statsmodels!

15.0.1 Perform standard imports and load the dataset

For these exercises we'll be using a statsmodels built-in macroeconomics dataset:

[17]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('macrodata.csv',index_col = 0, parse_dates=True)
df.head()
```

[17]:

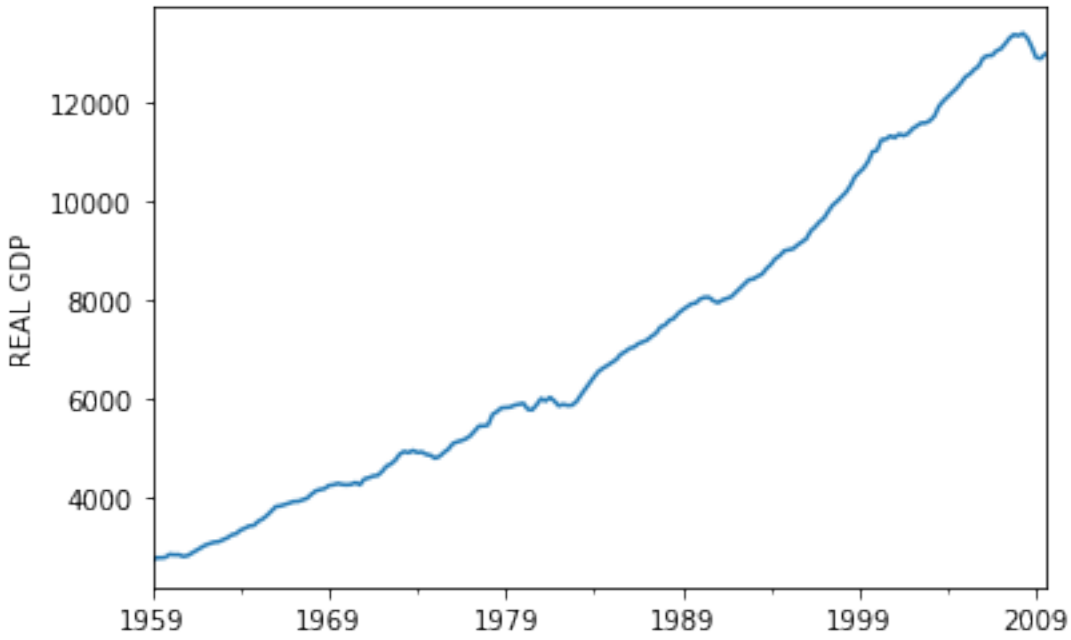
	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	\
1959-03-31	1959	1	2710.349	1707.4	286.898	470.045	1886.9	
1959-06-30	1959	2	2778.801	1733.7	310.859	481.301	1919.7	
1959-09-30	1959	3	2775.488	1751.8	289.226	491.260	1916.4	
1959-12-31	1959	4	2785.204	1753.7	299.356	484.052	1931.3	
1960-03-31	1960	1	2847.699	1770.5	331.722	462.199	1955.5	

	cpi	m1	tbilrate	unemp	pop	infl	realint
1959-03-31	28.98	139.7	2.82	5.8	177.146	0.00	0.00
1959-06-30	29.15	141.7	3.08	5.1	177.830	2.34	0.74
1959-09-30	29.35	140.5	3.82	5.3	178.657	2.74	1.09
1959-12-31	29.37	140.0	4.33	5.6	179.386	0.27	4.06
1960-03-31	29.54	139.6	3.50	5.2	180.007	2.31	1.19

15.0.2 Plot the dataset

[7]:
```
df['realgdp'].plot(ylabel = 'REAL GDP')
```

[7]: <Axes: ylabel='REAL GDP'>



15.1 Using Statsmodels to get the trend

Related Function:

statsmodels.tsa.filters.hp_filter.hpfilter(X, lamb=1600) Hodrick-Prescott filter

The Hodrick-Prescott filter separates a time-series y_t into a trend component τ_t and a cyclical component c_t

$y_t = \tau_t + c_t$

The components are determined by minimizing the following quadratic loss function, where λ is a smoothing parameter:

$\min_{\tau_t} \sum_{t=1}^{T} c_t^2 + \lambda \sum_{t=1}^{T} [(\tau_t - \tau_{t-1}) - (\tau_{t-1} - \tau_{t-2})]^2$

The λ value above handles variations in the growth rate of the trend component.When analyzing quarterly data, the default lambda value of 1600 is recommended. Use 6.25 for annual data, and 129,600 for monthly data.

[8]:
```
from statsmodels.tsa.filters.hp_filter import hpfilter

# Tuple unpacking

gdp_cycle, gdp_trend = hpfilter(df['realgdp'], lamb=1600)
```

[9]:
```
gdp_cycle
```

[9]:
```
1959-03-31     39.511915
1959-06-30     80.088532
1959-09-30     48.875455
1959-12-31     30.591933
1960-03-31     64.882667
                  ...
2008-09-30    102.018455
2008-12-31   -107.269472
2009-03-31   -349.047706
2009-06-30   -397.557073
2009-09-30   -333.115243
Name: realgdp_cycle, Length: 203, dtype: float64
```

We see from these numbers that for the period from 1960-09-30 to 1965-06-30 actual values fall below the trendline.

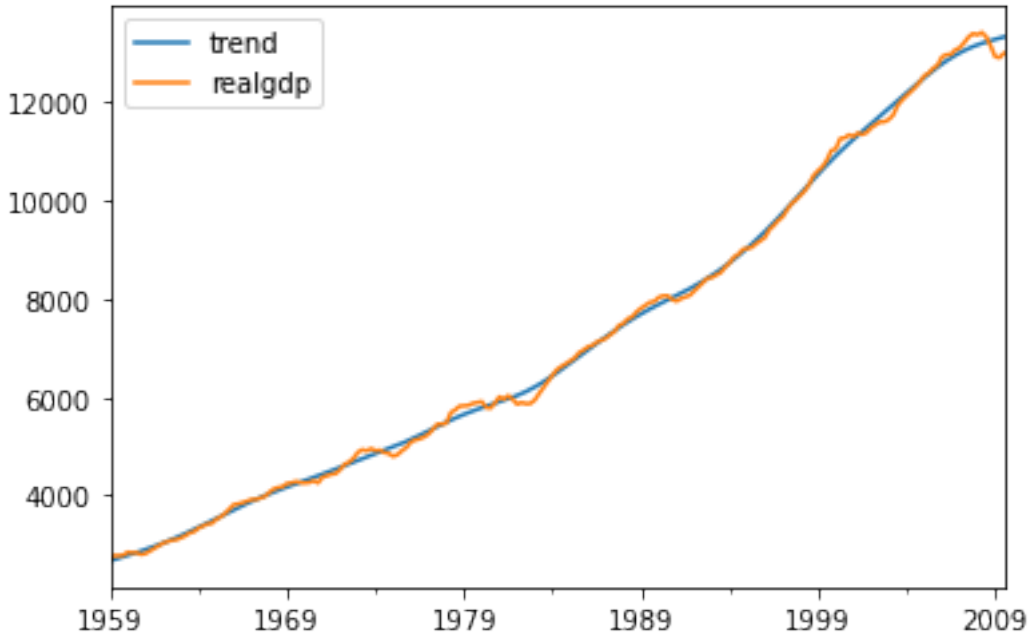
[10]:
```
type(gdp_cycle)
```

[10]:
```
pandas.core.series.Series
```

[11]:
```
df['trend'] = gdp_trend
```

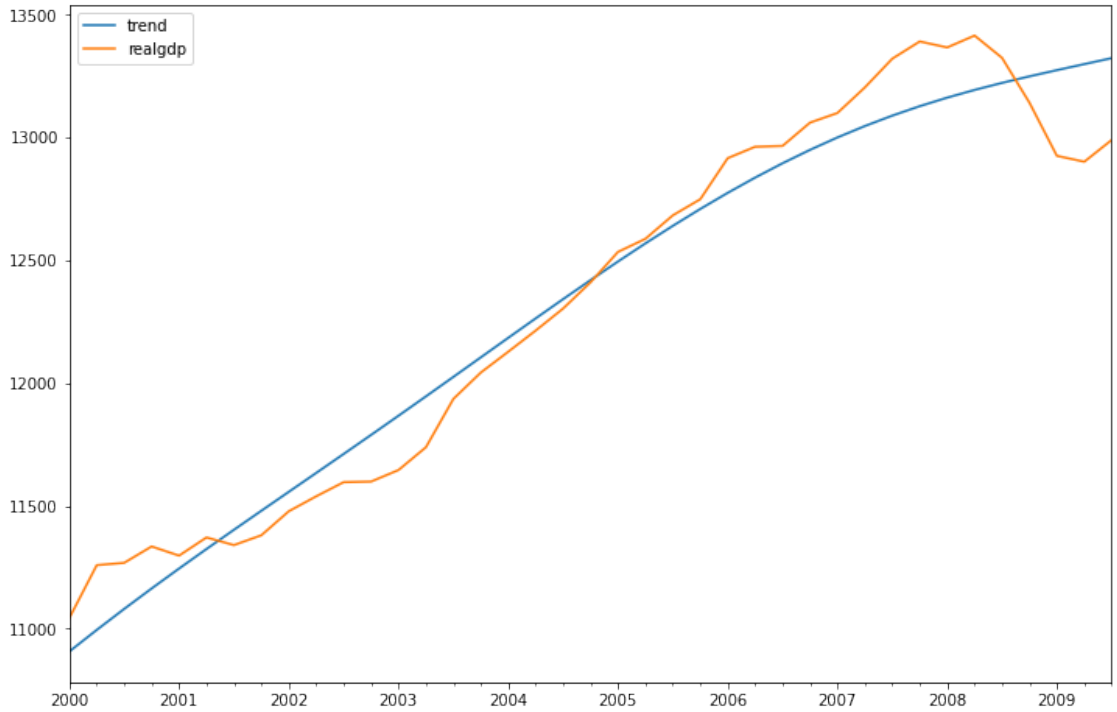
[12]:
```
df[['trend','realgdp']].plot()
```

[12]:
```
<Axes: >
```



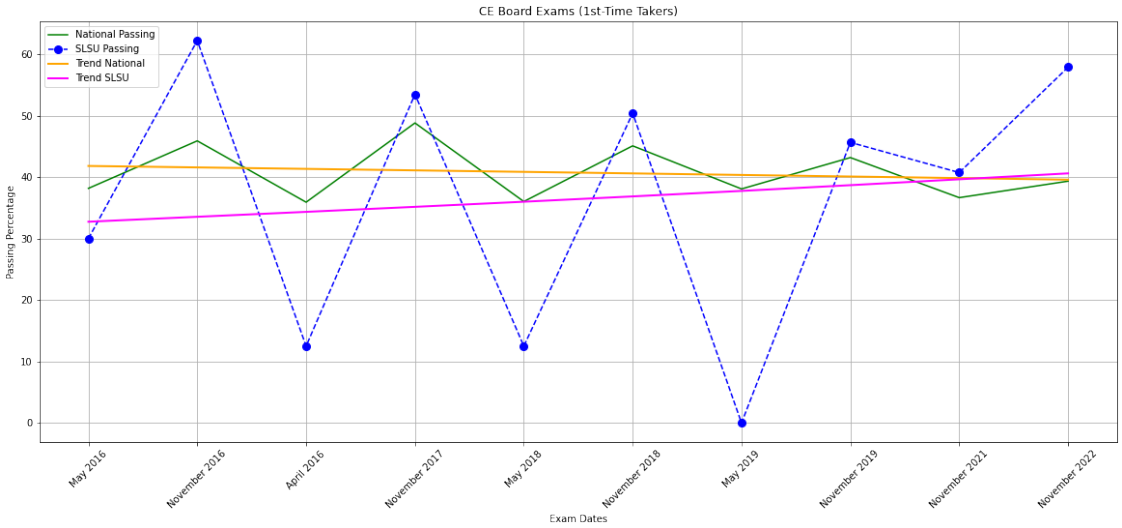
[14]:
```
df[['trend','realgdp']]['2000-03-31':].plot(figsize=(12,8))
```

[14]: <Axes: >



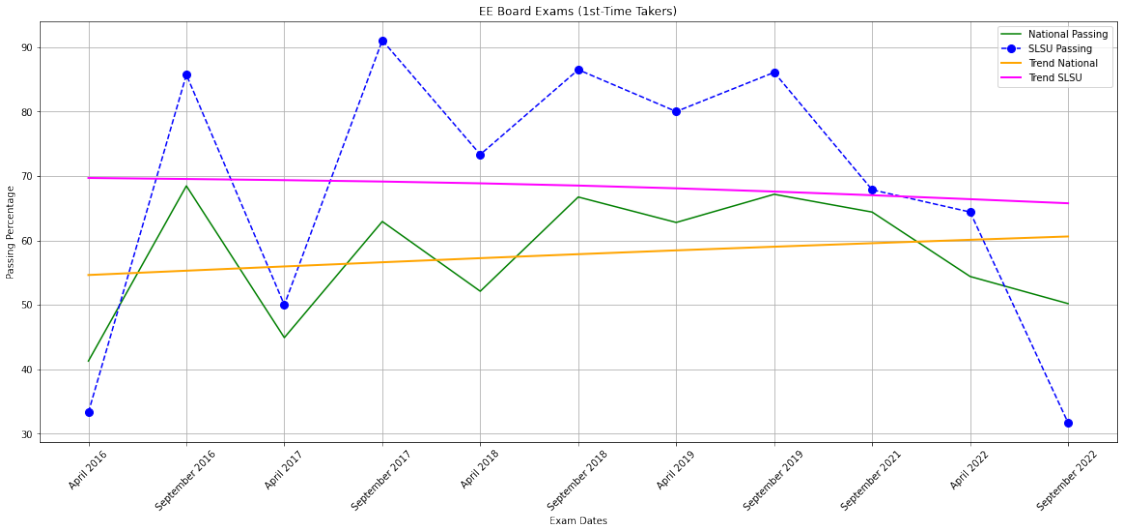
15.2 SLSU Board Exams

[38]:
```
# Civil Engineering

exam_dates = ['May 2016','November 2016','April 2016','November 2017',
              'May 2018','November 2018','May 2019','November 2019', 'November␣
 ↪2021', 'November 2022']

national_passing = [38.17,45.90,35.92,48.81,36.03,45.09,38.08,43.18,36.66,39.34]
slsu_passing     = [30,62.22,12.50,53.47,12.50,50.39,0,45.65,40.74,57.89]

cycle_slsu, trend_slsu = hpfilter(slsu_passing, lamb=1600) # monthly (129600),␣
 ↪quarterly (1600), annual (6.25)
cycle_national, trend_national = hpfilter(national_passing, lamb=1600) #␣
 ↪monthly (129600), quarterly (1600), annual (6.25)

# -----

plt.figure(figsize = (20,8))
plt.title('CE Board Exams (1st-Time Takers)')
plt.ylabel('Passing Percentage')
plt.xlabel('Exam Dates')
plt.xticks(rotation = 45)
plt.grid(True)


plt.plot(exam_dates, national_passing, color = 'green', label = 'National␣
 ↪Passing')
plt.plot(exam_dates, slsu_passing, ls='--', marker='o', markerfacecolor='blue',␣
 ↪markersize='8', color='blue', label = 'SLSU Passing')
plt.plot(exam_dates, trend_national, color='orange', linewidth=2, label =␣
 ↪'Trend National')
plt.plot(exam_dates, trend_slsu, color='magenta', linewidth=2, label = 'Trend␣
 ↪SLSU')

plt.legend()

plt.show()
```



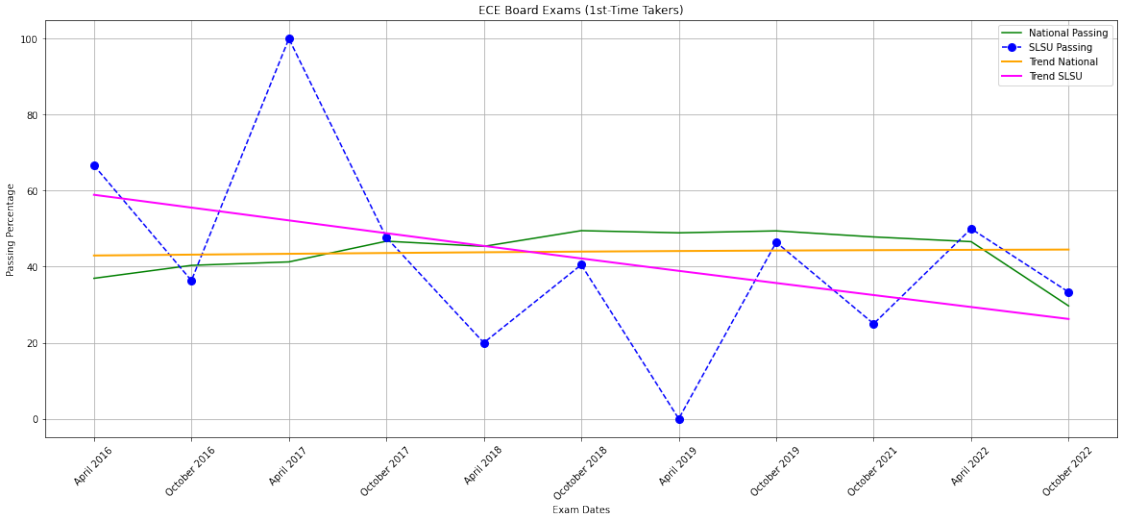
[39]:
```
# Electrical Engineering

exam_dates = ['April 2016','September 2016','April 2017','September 2017',
              'April 2018','September 2018','April 2019','September 2019',␣
 ↪'September 2021','April 2022','September 2022']

national_passing = [41.29,68.46,44.92,62.94,52.12,66.74,62.79,67.16,64.40,54.
 ↪41,50.20]
slsu_passing     = [33.33,85.71,50,91.04,73.33,86.49,80,86.07,67.86,64.41,31.71]

cycle_slsu, trend_slsu = hpfilter(slsu_passing, lamb=1600) # monthly (129600),␣
 ↪quarterly (1600), annual (6.25)
cycle_national, trend_national = hpfilter(national_passing, lamb=1600) #␣
 ↪monthly (129600), quarterly (1600), annual (6.25)

plt.figure(figsize = (20,8))
plt.title('EE Board Exams (1st-Time Takers)')
plt.ylabel('Passing Percentage')
plt.xlabel('Exam Dates')
plt.xticks(rotation = 45)
plt.grid(True)


plt.plot(exam_dates, national_passing, color = 'green', label = 'National␣
 ↪Passing')
plt.plot(exam_dates, slsu_passing, ls='--', marker='o', markerfacecolor='blue',␣
 ↪markersize='8', color='blue', label = 'SLSU Passing')
plt.plot(exam_dates, trend_national, color='orange', linewidth=2, label =␣
 ↪'Trend National')
```

```
plt.plot(exam_dates, trend_slsu, color='magenta', linewidth=2, label = 'Trend␣
 ↪SLSU')

plt.legend()

plt.show()
```



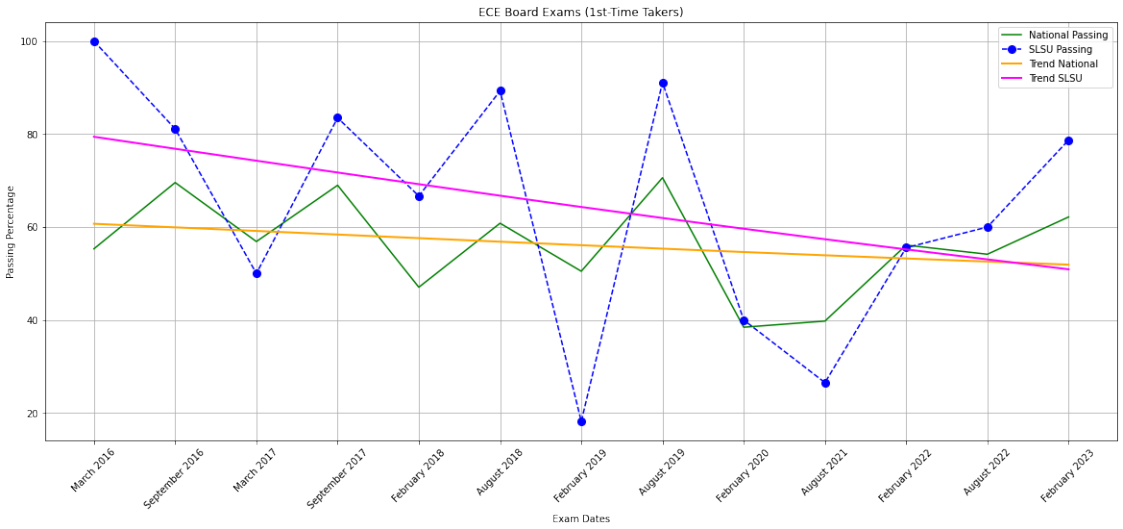
[41]:
```
# Electronics Engineering

exam_dates = ['April 2016','October 2016','April 2017','October 2017',
              'April 2018','Ocotober 2018','April 2019','October 2019','October␣
 ↪2021','April 2022','October 2022']

national_passing = [36.95,40.36,41.27,46.72,45.36,49.49,48.92,49.43,47.84,46.
 ↪60,29.69]
slsu_passing     = [66.67,36.36,100,47.69,20,40.51,0,46.43,25,50,33.33]

cycle_slsu, trend_slsu = hpfilter(slsu_passing, lamb=1600) # monthly (129600),␣
 ↪quarterly (1600), annual (6.25)
cycle_national, trend_national = hpfilter(national_passing, lamb=1600) #␣
 ↪monthly (129600), quarterly (1600), annual (6.25)

# ----

plt.figure(figsize = (20,8))
plt.title('ECE Board Exams (1st-Time Takers)')
plt.ylabel('Passing Percentage')
plt.xlabel('Exam Dates')
```

```
plt.xticks(rotation = 45)
plt.grid(True)


plt.plot(exam_dates, national_passing, color = 'green', label = 'National␣
 ↪Passing')
plt.plot(exam_dates, slsu_passing, ls='--', marker='o', markerfacecolor='blue',␣
 ↪markersize='8', color='blue', label = 'SLSU Passing')
plt.plot(exam_dates, trend_national, color='orange', linewidth=2, label =␣
 ↪'Trend National')
plt.plot(exam_dates, trend_slsu, color='magenta', linewidth=2, label = 'Trend␣
 ↪SLSU')

plt.legend()

plt.show()
```



[42]:
```
# Mechanical Engineering

exam_dates = ['March 2016','September 2016','March 2017','September 2017',
              'February 2018','August 2018','February 2019','August␣
 ↪2019','February 2020','August 2021','February 2022','August 2022',
              'February 2023']

national_passing = [55.32,69.57,56.88,68.99,47.05,60.82,50.49,70.61,38.45,39.
 ↪77,56.11,54.15,62.17]
slsu_passing     = [100,81.11,50,83.50,66.67,89.29,18.18,91.11,40,26.53,55.
 ↪56,60,78.72]
```

```
cycle_slsu, trend_slsu = hpfilter(slsu_passing, lamb=1600) # monthly (129600), quarterly (1600), annual (6.25)
cycle_national, trend_national = hpfilter(national_passing, lamb=1600) # monthly (129600), quarterly (1600), annual (6.25)

# ----

plt.figure(figsize = (20,8))
plt.title('ECE Board Exams (1st-Time Takers)')
plt.ylabel('Passing Percentage')
plt.xlabel('Exam Dates')
plt.xticks(rotation = 45)
plt.grid(True)


plt.plot(exam_dates, national_passing, color = 'green', label = 'National Passing')
plt.plot(exam_dates, slsu_passing, ls='--', marker='o', markerfacecolor='blue', markersize='8', color='blue', label = 'SLSU Passing')
plt.plot(exam_dates, trend_national, color='orange', linewidth=2, label = 'Trend National')
plt.plot(exam_dates, trend_slsu, color='magenta', linewidth=2, label = 'Trend SLSU')

plt.legend()

plt.show()
```



15.3 ETS

15.4 Error/Trend/Seasonality Models

As we begin working with endogenous data ("endog" for short) and start to develop forecasting models, it helps to identify and isolate factors working within the system that influence behavior. Here the name "endogenous" considers internal factors, while "exogenous" would relate to external forces. These fall under the category of state space models, and include decomposition (described below), and exponential smoothing (described in an upcoming section).

The decomposition of a time series attempts to isolate individual components such as error, trend, and seasonality (ETS). We've already seen a simplistic example of this in the Introduction to Statsmodels section with the Hodrick-Prescott filter. There we separated data into a trendline and a cyclical feature that mapped observed data back to the trend.

Related Function:

statsmodels.tsa.seasonal.seasonal_decompose(x, model) Seasonal decomposition using moving averages

For Further Reading:

Forecasting: Principles and Practice Innovations state space models for exponential smoothing Wikipedia Decomposition of time series

15.5 Seasonal Decomposition

Statsmodels provides a seasonal decomposition tool we can use to separate out the different components. This lets us see quickly and visually what each component contributes to the overall behavior.

We apply an additive model when it seems that the trend is more linear and the seasonality and trend components seem to be constant over time (e.g. every year we add 10,000 passengers). A multiplicative model is more appropriate when we are increasing (or decreasing) at a non-linear rate (e.g. each year we double the amount of passengers).

For these examples we'll use the International Airline Passengers dataset, which gives monthly totals in thousands from January 1949 to December 1960.

[ ]:
```
import pandas as pd
import numpy as np
%matplotlib inline
```

[ ]:
```
airline = pd.read_csv('../Data/airline_passengers.
  ↪csv',index_col='Month',parse_dates=True)
```

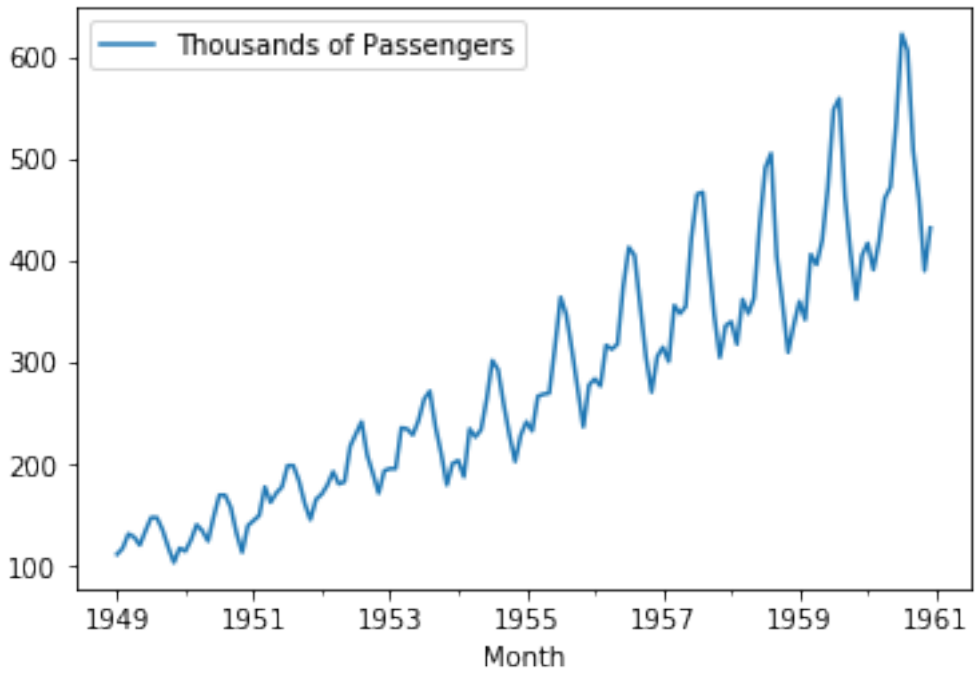
[ ]:
```
airline.dropna(inplace=True)
```

[ ]:
```
airline.head()
```

[ ]:
```
            Thousands of Passengers
Month
```

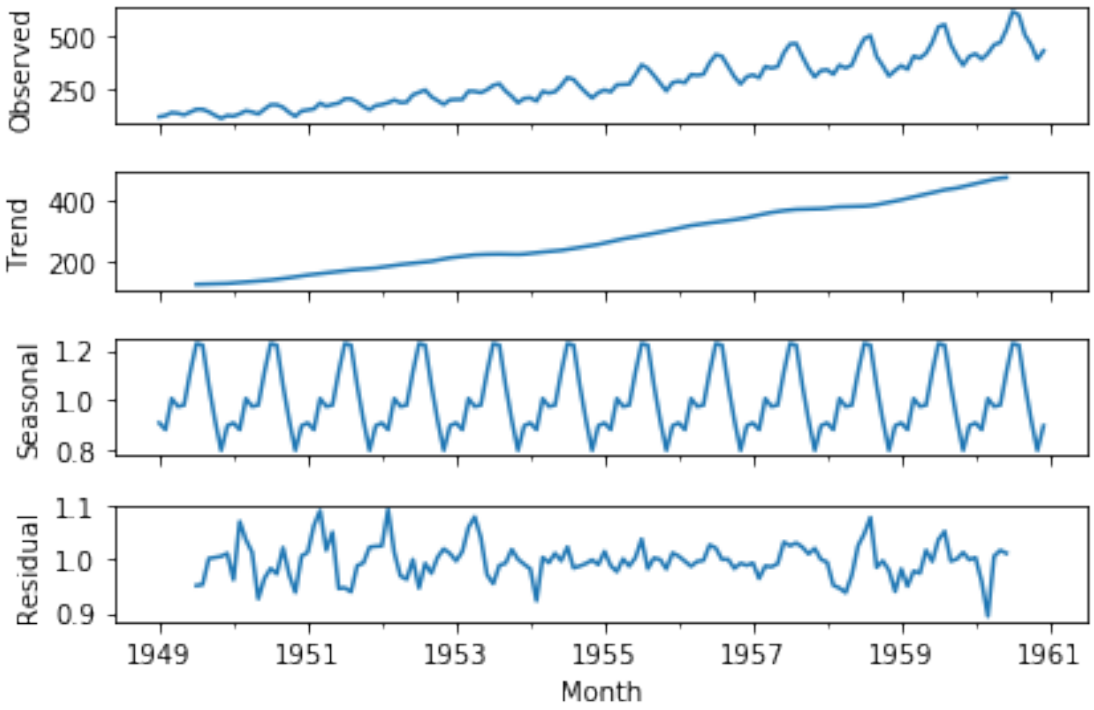
```
1949-01-01                    112
1949-02-01                    118
1949-03-01                    132
1949-04-01                    129
1949-05-01                    121
```

[ ]:
```
airline.plot();
```



Based on this chart, it looks like the trend in the earlier days is increasing at a higher rate than just linear (although it is a bit hard to tell from this one plot).

[ ]:
```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(airline['Thousands of Passengers'],␣
 ↪model='multiplicative')  # model='mul' also works
result.plot();
```



Great! In the next section we'll see how to apply exponential smoothing models to each of these contributing factors.

15.6 ETS

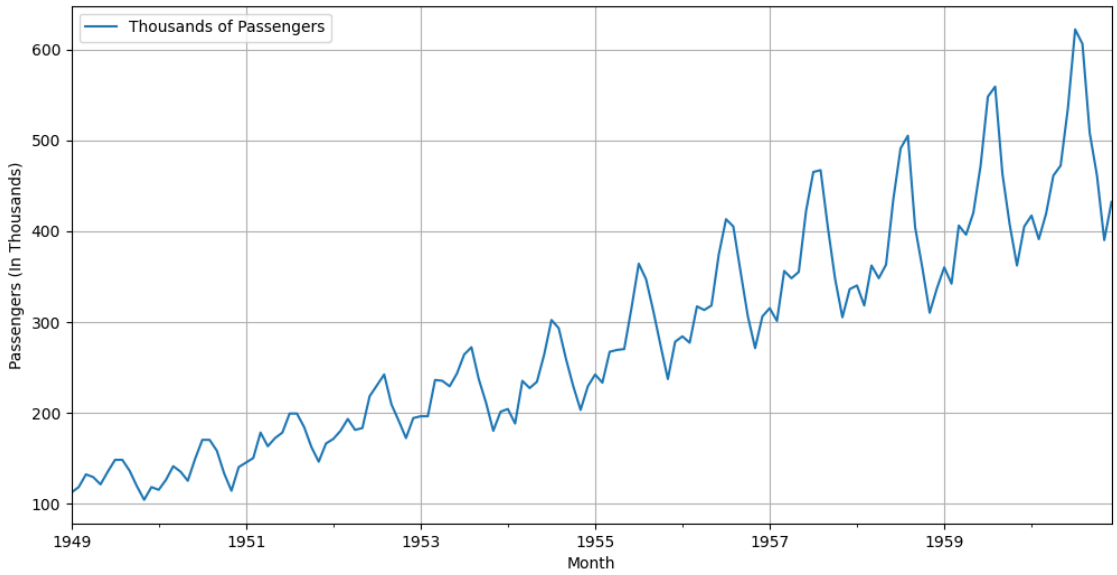
[26]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[27]:
```
airline = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
  ↪public_files/refs/heads/master/Data%20Sets/airline_passengers.
  ↪csv',index_col='Month',parse_dates=True)
airline.head()
```

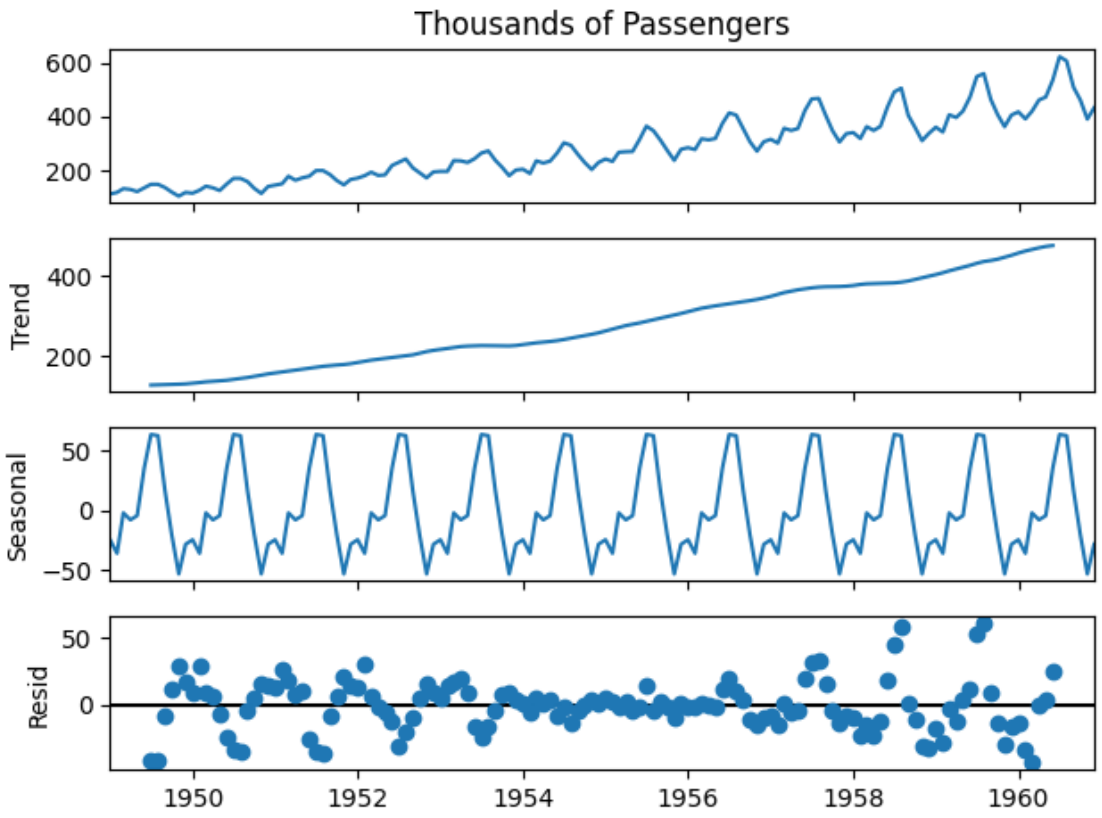
[27]:

Month	Thousands of Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

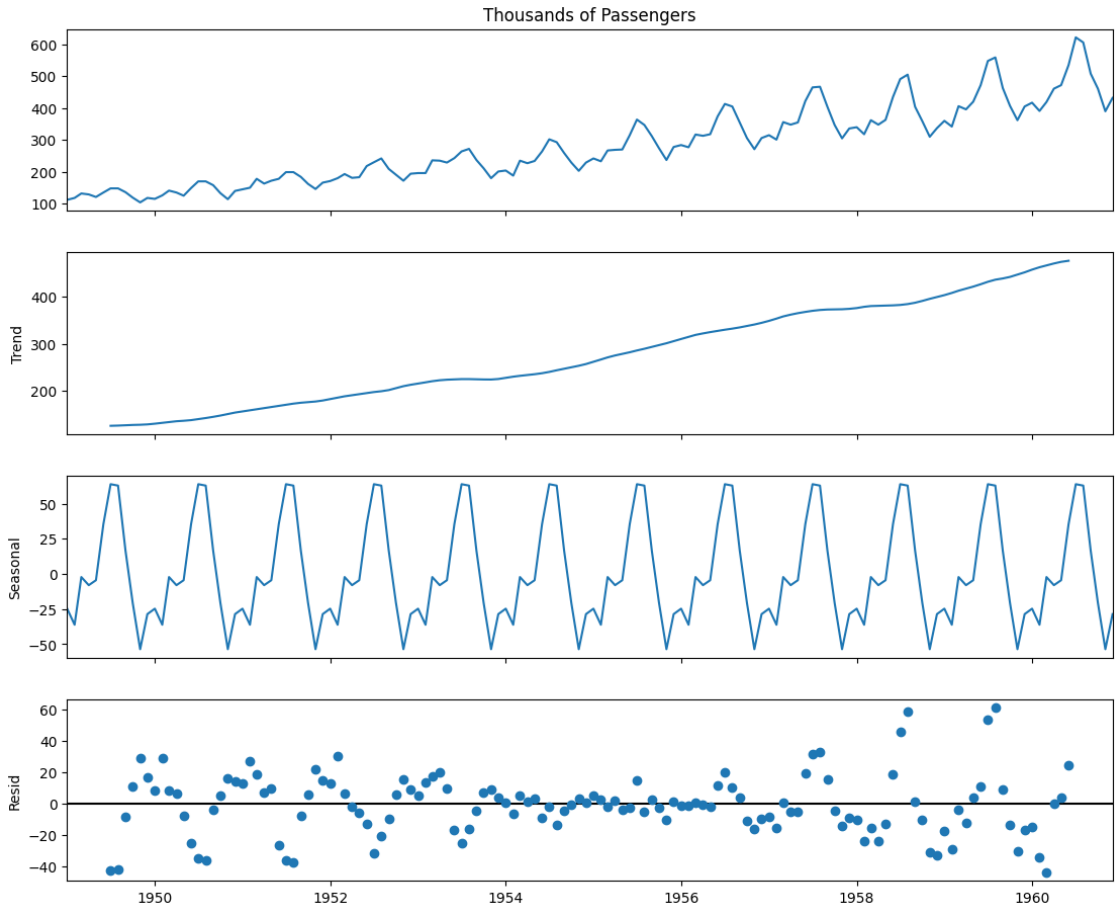
[28]:
```
airline.plot(figsize = (12, 6), ylabel = 'Passengers (In Thousands)');
plt.grid()
```



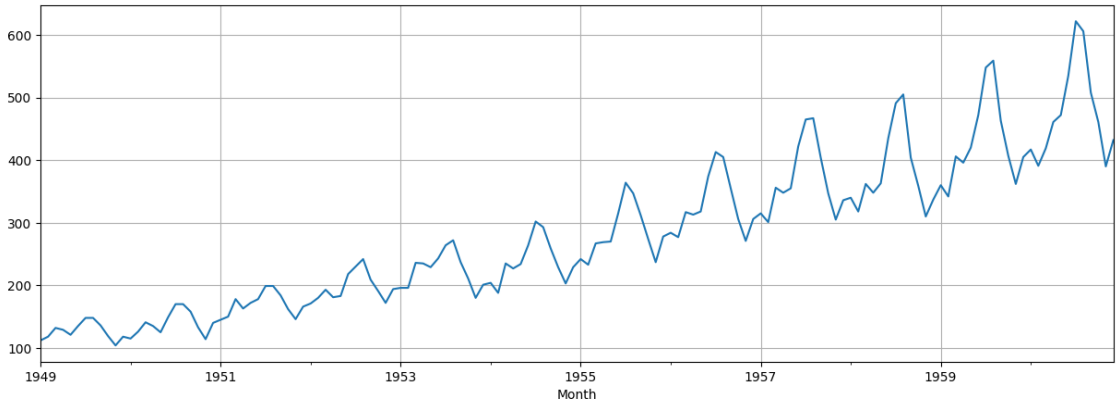
[29]:
```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(airline['Thousands of Passengers'],␣
 ↪model='additive')
result.plot();
```



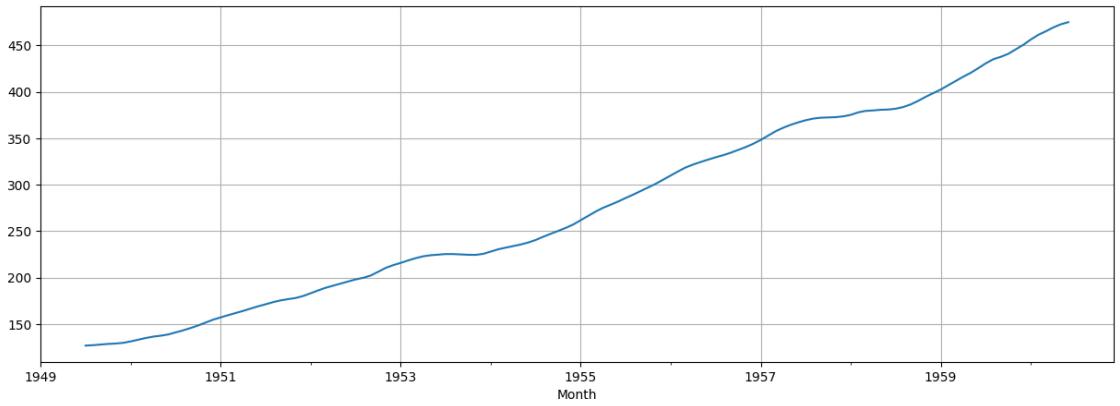
[30]:
```
decomposition = seasonal_decompose(airline['Thousands of Passengers'],␣
↪model='additive')

# Increase figure size
fig = decomposition.plot()
fig.set_size_inches(12, 10)  # Width = 12 inches, Height = 8 inches
plt.show()
```



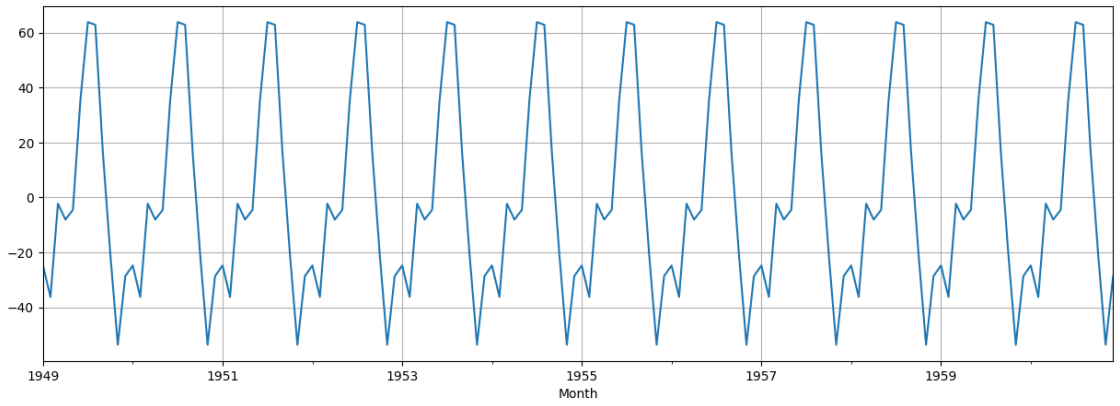
[35]:
```
# Individual Graphs (Observed)

decomposition.observed.plot(figsize = (15,5))
plt.grid()
plt.show()
```



[36]:
```
# Individual Graphs (Trend)

decomposition.trend.plot(figsize = (15,5))
plt.grid()
plt.show()
```

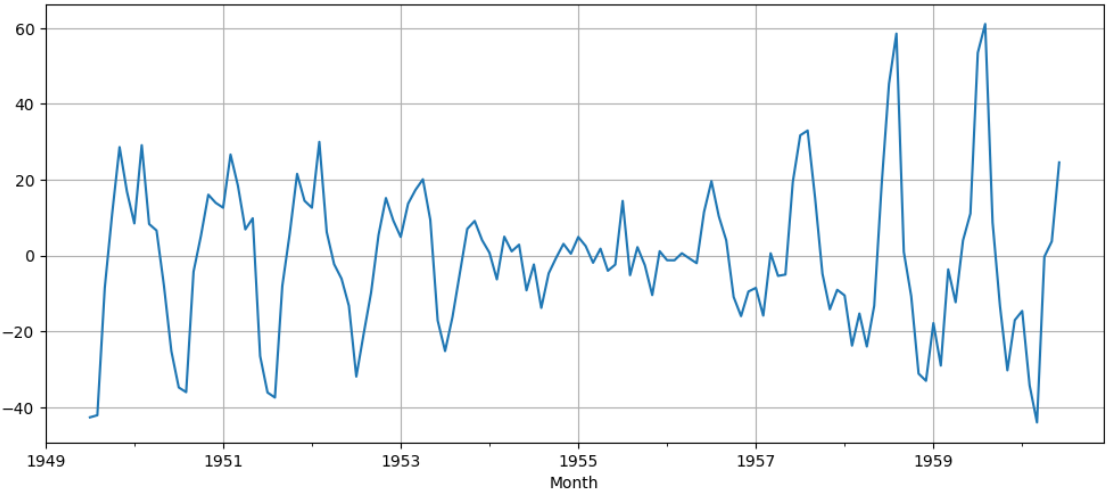


[33]:
```
# Individual Graphs (Seasonal)

decomposition.seasonal.plot(figsize = (15,5))
plt.grid()
plt.show()
```



[34]:
```
# Individual Graphs (Residual/Error)

decomposition.resid.plot(figsize = (12,5))
plt.grid()
```

```
plt.show()
```



15.7 STL

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

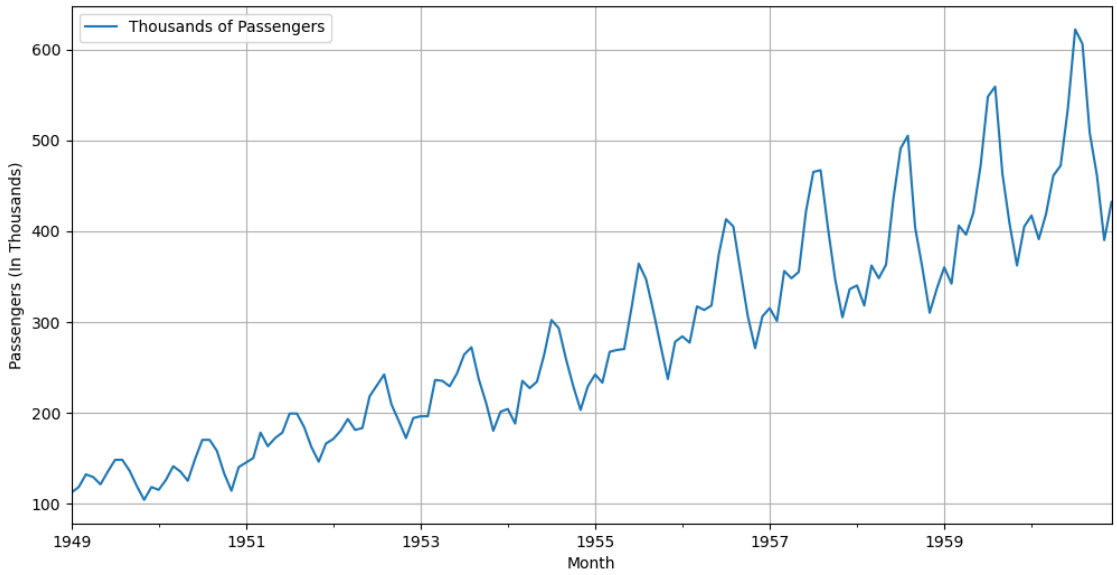
15.7.1 1st Case

[ ]:
```
airline = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
  ↪public_files/refs/heads/master/Data%20Sets/airline_passengers.
  ↪csv',index_col='Month',parse_dates=True)
airline.head()
```

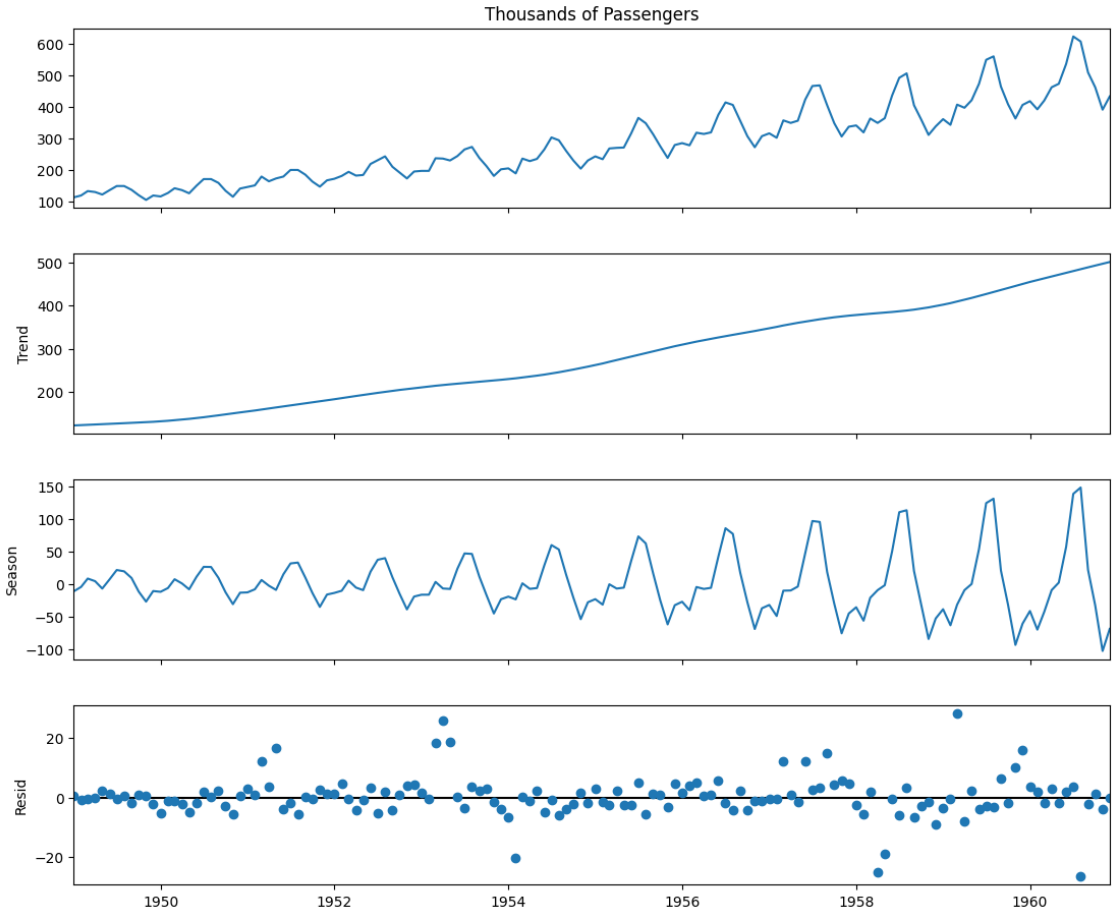
[ ]:

Month	Thousands of Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

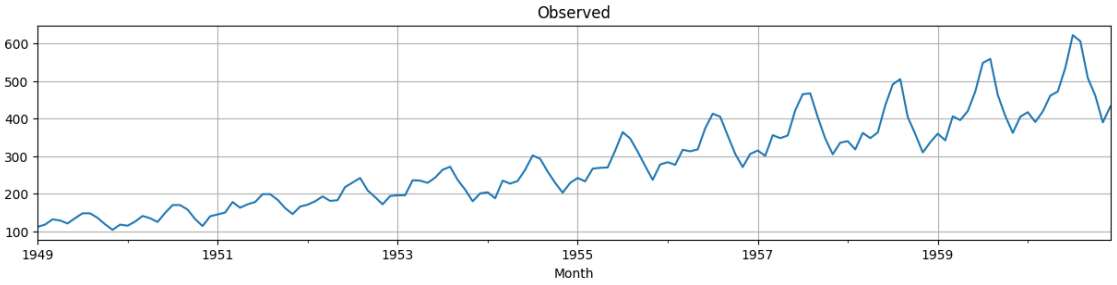
[ ]:
```
airline.plot(figsize = (12, 6), ylabel = 'Passengers (In Thousands)');
plt.grid()
```



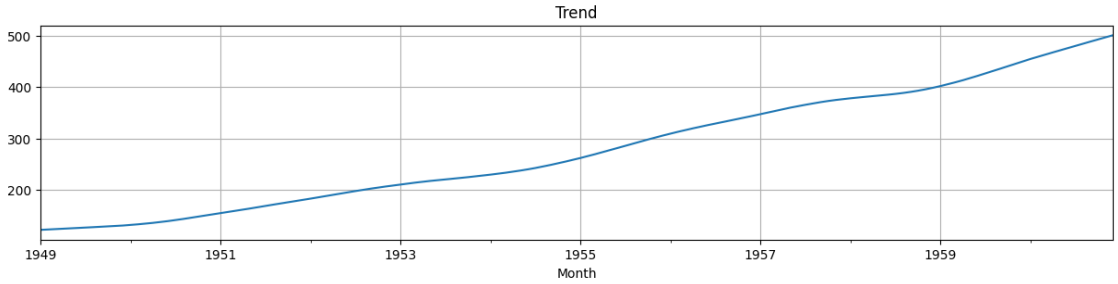
[ ]:
```
from statsmodels.tsa.seasonal import STL

stl = STL(airline["Thousands of Passengers"], period = 12, robust = True)  #␣
 ↪Robust = better handling of outliers
decomp_stl = stl.fit()

# Increase figure size
fig = decomp_stl.plot();
fig.set_size_inches(12, 10)  # Width = 12 inches, Height = 8 inches
plt.show()
```



[ ]:
```
# Individual Graphs (Observed)

decomp_stl.observed.plot(figsize = (15,3))
plt.title('Observed')
plt.grid()
plt.show()
```

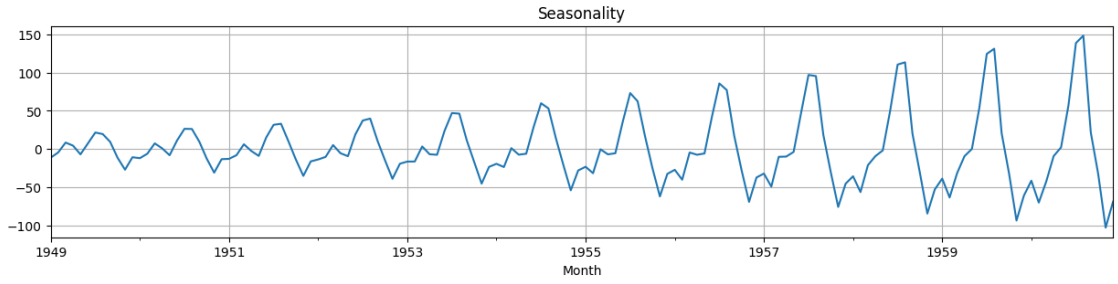


[ ]:
```
# Individual Graphs (Trend)

decomp_stl.trend.plot(figsize = (15,3))
plt.title('Trend')
plt.grid()
plt.show()
```

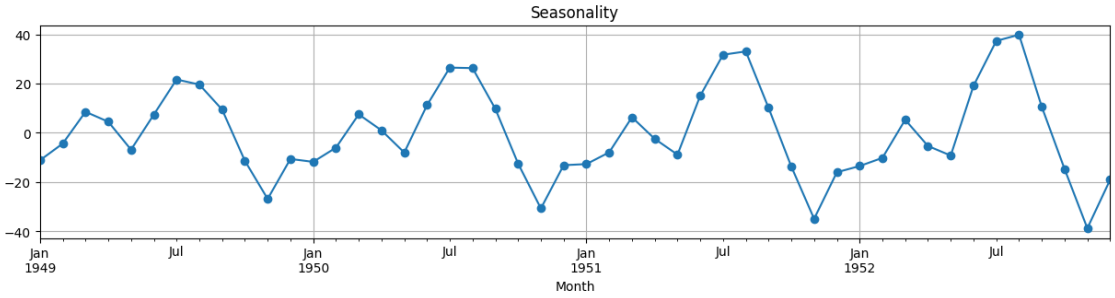


[ ]:
```
# The trend line shows a consistent upward trend, indicating long-term growth␣
↪in airline passenger numbers.
# The trend is smooth, capturing the increasing demand over time.
```

[ ]:
```
# Individual Graphs (Seasonal)

decomp_stl.seasonal.plot(figsize = (15,3))
plt.title('Seasonality')
plt.grid()
plt.show()
```

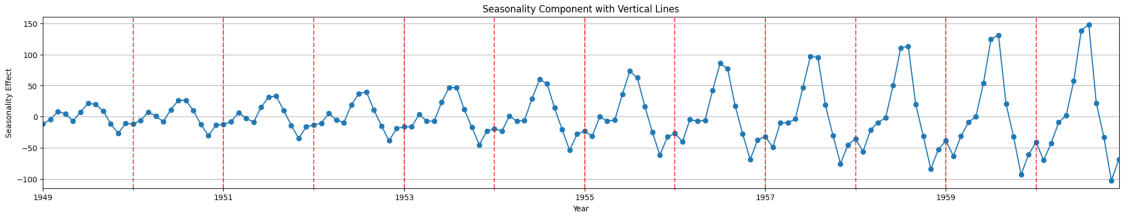


[ ]:
```
# Seasonality (Zoomed In)

decomp_stl.seasonal[:48].plot(figsize = (15,3), marker = 'o')
plt.title('Seasonality')
plt.grid()
plt.show()
```

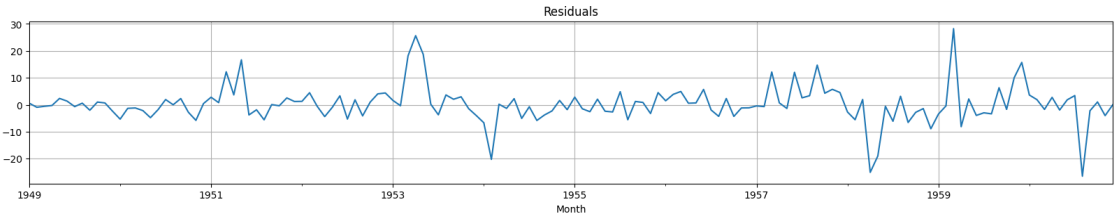


[ ]:
```
# The seasonal component displays a strong repeating pattern every 12 months.
# This confirms that seasonality is well-defined and occurs annually.
```

[ ]:
```
# Plot STL decomposition with vertical lines every 12 months

fig, ax = plt.subplots(figsize=(25, 4))
decomp_stl.seasonal.plot(ax = ax, title='Seasonality Component with Vertical␣
 ↪Lines', marker = 'o')

# Add vertical lines every 12 months
for i in range(0, len(airline), 12):

    ax.axvline(airline.index[i], color='red', linestyle='--', alpha=0.7)

plt.xlabel("Year")
plt.ylabel("Seasonality Effect")
plt.grid(True)
plt.show()
```



[ ]:
```
# Individual Graphs (Residual/Error)

decomp_stl.resid.plot(figsize = (20,3))
plt.title('Residuals')
plt.grid()
plt.show()
```



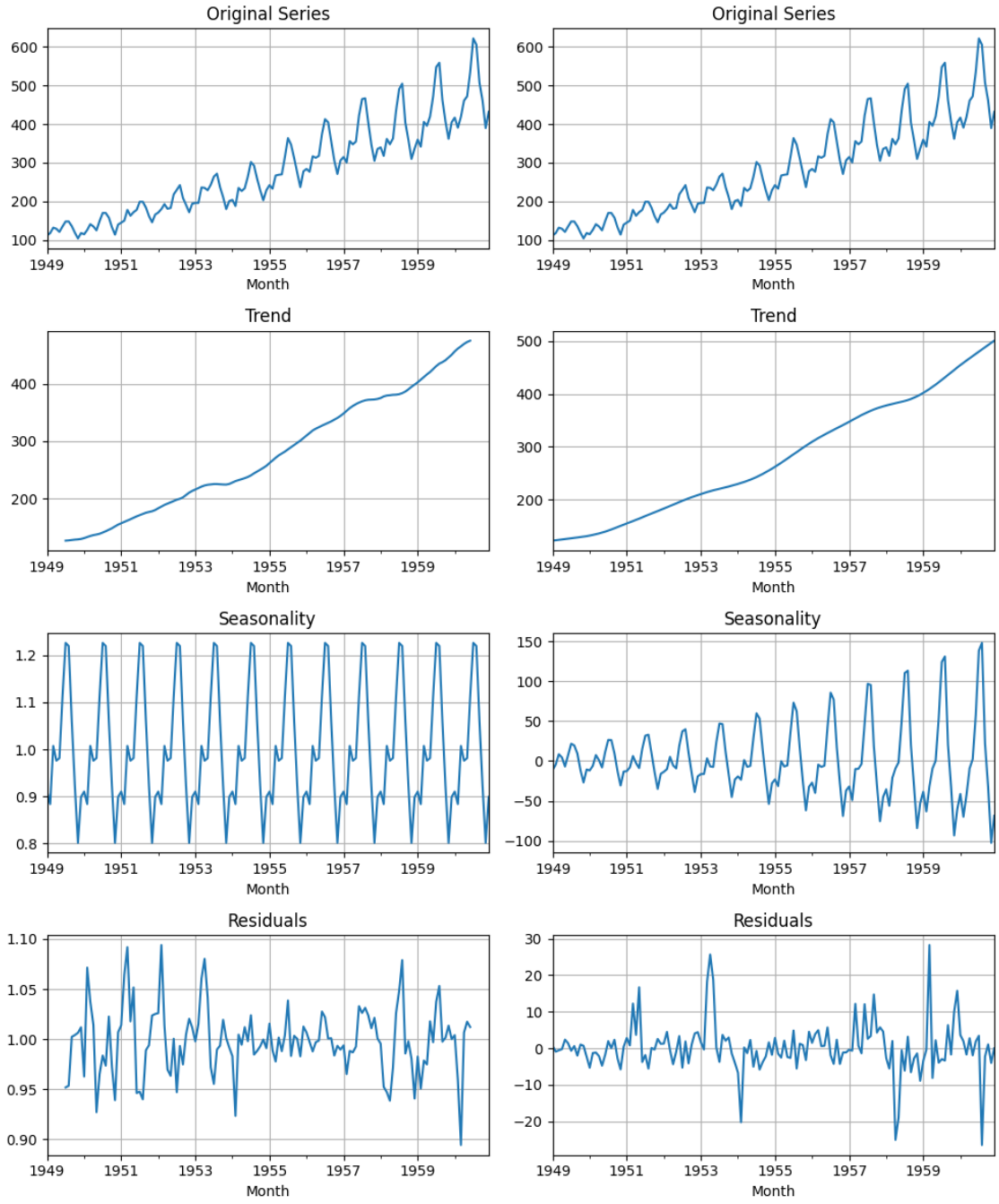
[ ]:
```
# The residuals are relatively small and mostly centered around zero.
# The residuals do not exhibit strong structure (no visible patterns),␣
 ↪confirming that the STL decomposition
# has successfully extracted trend and seasonality.
```

Comparative Graphs

[ ]:
```
# Apply ETS Decomposition (Additive Model)

from statsmodels.tsa.seasonal import seasonal_decompose

# ETS
decomp_ets = seasonal_decompose(airline["Thousands of Passengers"], model =␣
 ↪"multiplicative", period = 12)

# STL
stl = STL(airline["Thousands of Passengers"], period = 12, robust = True)  #␣
 ↪Robust = better handling of outliers
decomp_stl = stl.fit()
```

[ ]:
```
# Side by Side Comparisons

# Plot ETS and STL Decomposition Side-by-Side
fig, axes = plt.subplots(4, 2, figsize=(10, 12))

# Titles for both models
titles = ["Original Series", "Trend", "Seasonality", "Residuals"]

# Plot ETS Decomposition (Left/Side)
components_ets = [airline["Thousands of Passengers"], decomp_ets.trend,␣
 ↪decomp_ets.seasonal, decomp_ets.resid]
for i, component in enumerate(components_ets):

    component.plot(ax = axes[i, 0], title = titles[i])
    axes[i, 0].grid()

# Plot STL Decomposition (Right/Side)
```

```
components_stl = [airline["Thousands of Passengers"], decomp_stl.trend,␣
 ↪decomp_stl.seasonal, decomp_stl.resid]
for i, component in enumerate(components_stl):

    component.plot(ax = axes[i, 1], title = titles[i])
    axes[i, 1].grid()

plt.tight_layout()
plt.show()
```



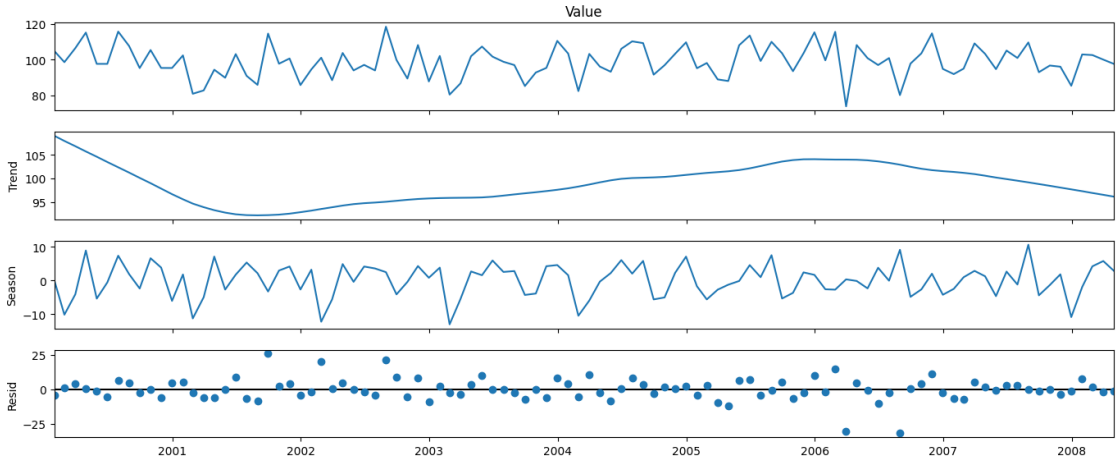
15.7.2 2nd Case

[ ]:
```
ts_sensor = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
↪public_files/refs/heads/master/Data%20Sets/ts_no_pattern.
↪csv',index_col='Date', parse_dates=True)
ts_sensor.head()
```

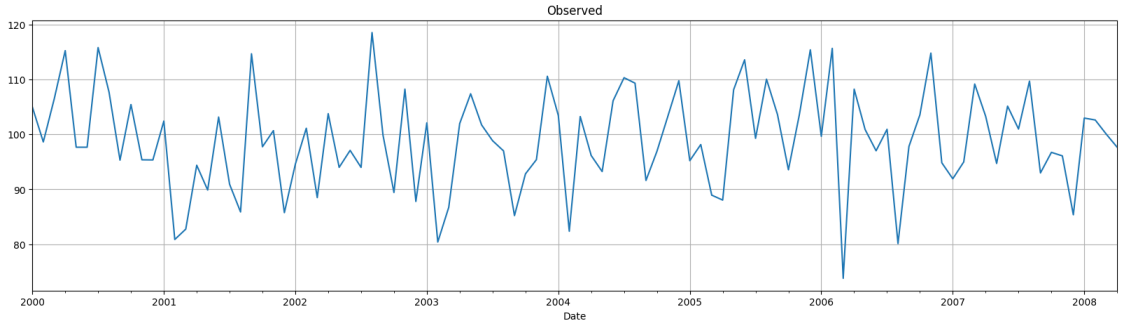
[ ]:
```
                Value
Date
2000-01-31  104.967142
2000-02-29   98.617357
2000-03-31  106.476885
2000-04-30  115.230299
2000-05-31   97.658466
```

[ ]:
```
#  STL decomposition with a 12-month seasonal period (checking for yearly␣
↪seasonality)

stl = STL(ts_sensor['Value'], period = 12, robust = True)
stl_decomposition = stl.fit()
```

[ ]:
```
# STL Graphs

fig = stl_decomposition.plot()
fig.set_size_inches(15, 6)
plt.show()
```

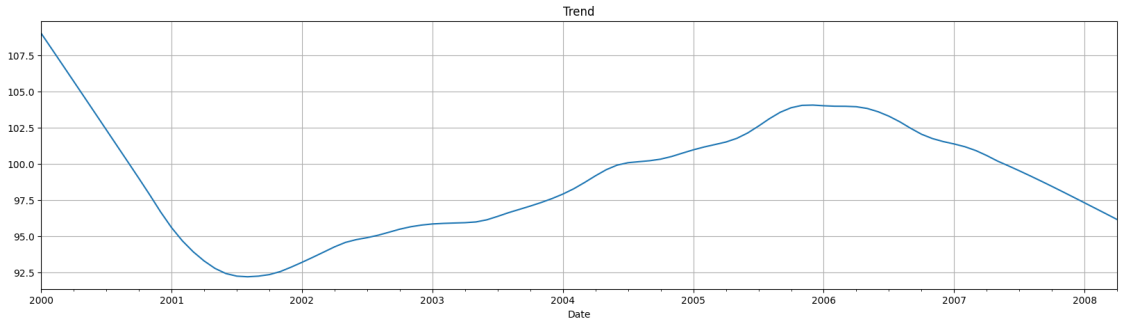


[ ]:
```
# Observed

fig = stl_decomposition.observed.plot(figsize = (20, 5))
plt.title('Observed')
plt.grid()
plt.show()
```



[ ]:
```
# The original time series shows random fluctuations without any clear pattern.
```

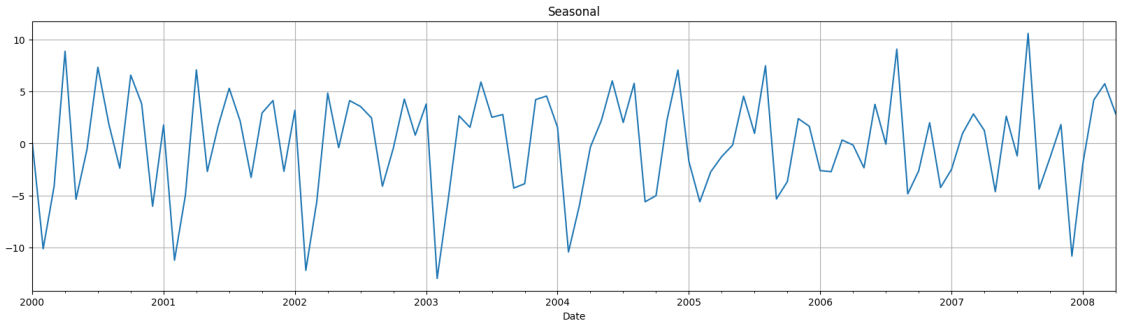
[ ]:
```
# Trend

fig = stl_decomposition.trend.plot(figsize = (20, 5))
plt.title('Trend')
plt.grid()
plt.show()
```



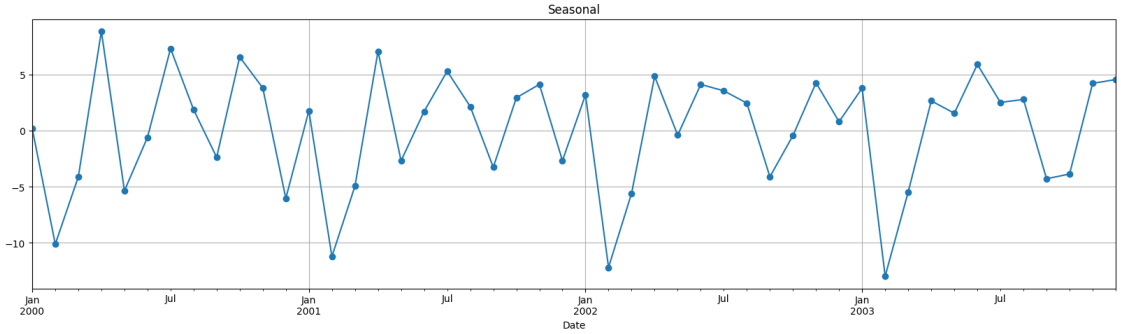
[ ]:
```
# The trend component is unstable and fluctuates without a clear direction.
# This indicates that no consistent upward or downward trend exists.
```

[ ]:
```
# Seasonality

fig = stl_decomposition.seasonal.plot(figsize = (20, 5))
```

```
plt.title('Seasonal')
plt.grid()
plt.show()
```

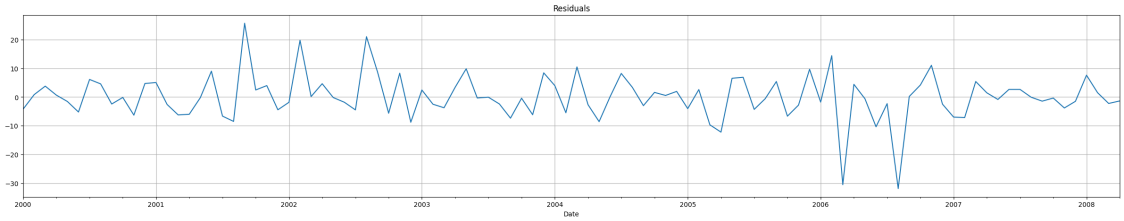


[ ]:
```
# Seasonality (Zoomed In)

fig = stl_decomposition.seasonal[:48].plot(figsize = (20, 5), marker = 'o')
plt.title('Seasonal')
plt.grid()
plt.show()
```



[ ]:
```
# The seasonal component appears random rather than showing a repeating cycle.
# This confirms that no meaningful seasonality exists.
# Unlike ETS decomposition, STL does not try to force a seasonal pattern where␣
↪there is none.
```

[ ]:
```
# Residuals

fig = stl_decomposition.resid.plot(figsize = (30, 5))
plt.title('Residuals')
plt.grid()
plt.show()
```



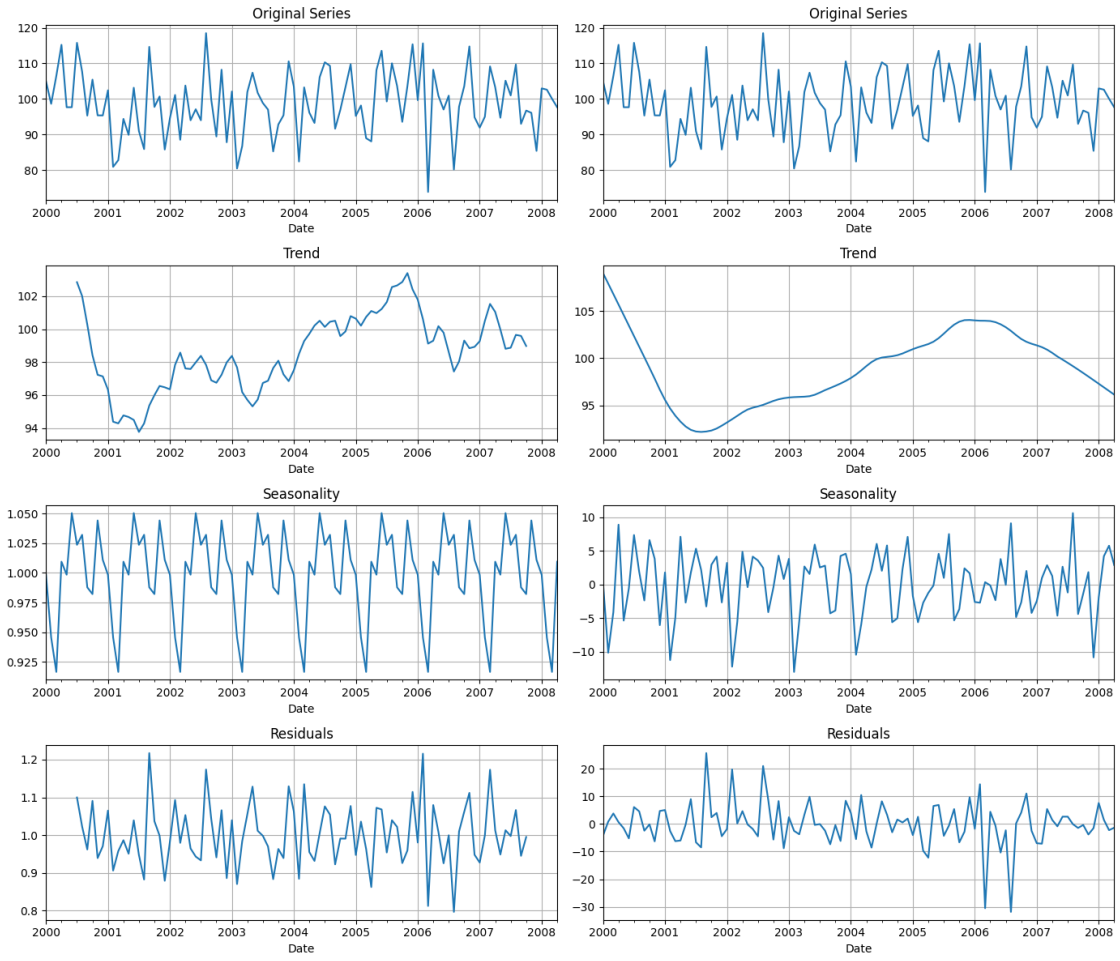
[ ]:
```
# The residuals behave like pure noise, meaning all variation in the dataset is␣
↪random.
# This reinforces that there is no hidden structure in the data.
```

[ ]:
```
# STL decomposition clearly reveals that there is no seasonality in this␣
↪dataset.
```

Side by Side Comparison

[ ]:
```
# Side by Side Comparisons

decomp_ets = seasonal_decompose(ts_sensor["Value"], model="multiplicative",␣
↪period = 12)
stl = STL(ts_sensor["Value"], period = 12, robust = True)  # Robust = better␣
↪handling of outliers
decomp_stl = stl.fit()

# Plot ETS and STL Decomposition Side-by-Side
fig, axes = plt.subplots(4, 2, figsize=(14, 12))

# Titles for both models
titles = ["Original Series", "Trend", "Seasonality", "Residuals"]

# Plot ETS Decomposition (Left/Side)
components_ets = [ts_sensor["Value"], decomp_ets.trend, decomp_ets.seasonal,␣
↪decomp_ets.resid]
for i, component in enumerate(components_ets):

    component.plot(ax = axes[i, 0], title = titles[i])
    axes[i, 0].grid()

# Plot STL Decomposition (Right/Side)
components_stl = [ts_sensor["Value"], decomp_stl.trend, decomp_stl.seasonal,␣
↪decomp_stl.resid]
for i, component in enumerate(components_stl):

    component.plot(ax = axes[i, 1], title = titles[i])
    axes[i, 1].grid()
```

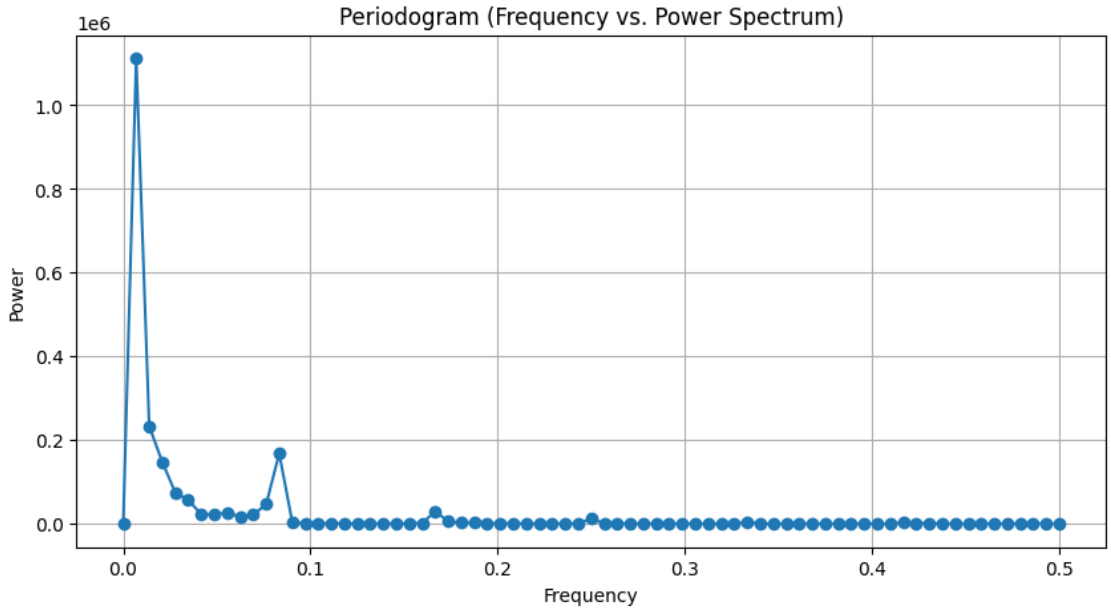
```
plt.tight_layout()
plt.show()
```



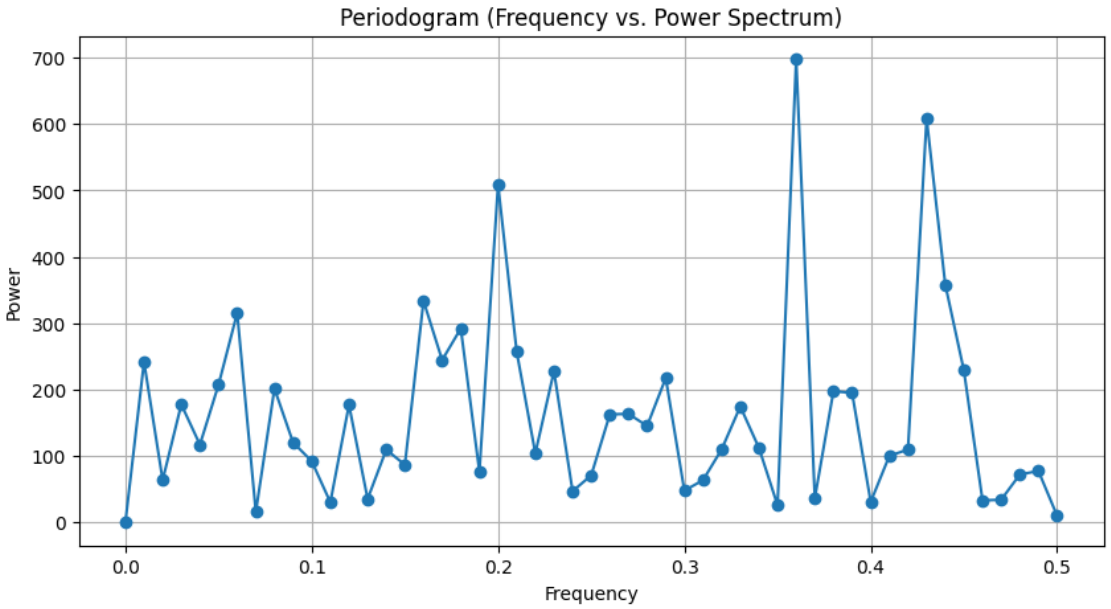
15.7.3 Alternative Method for Seasonality

[ ]:
```
# Other Methods for Seasonality (Periodogram)

from scipy.signal import periodogram

# Compute the periodogram
frequencies, power = periodogram(airline['Thousands of Passengers'])

# Plot the periodogram

plt.figure(figsize=(10, 5))
plt.plot(frequencies, power, marker='o', linestyle='-')
```

```
plt.title("Periodogram (Frequency vs. Power Spectrum)")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.grid(True)
plt.show()
```



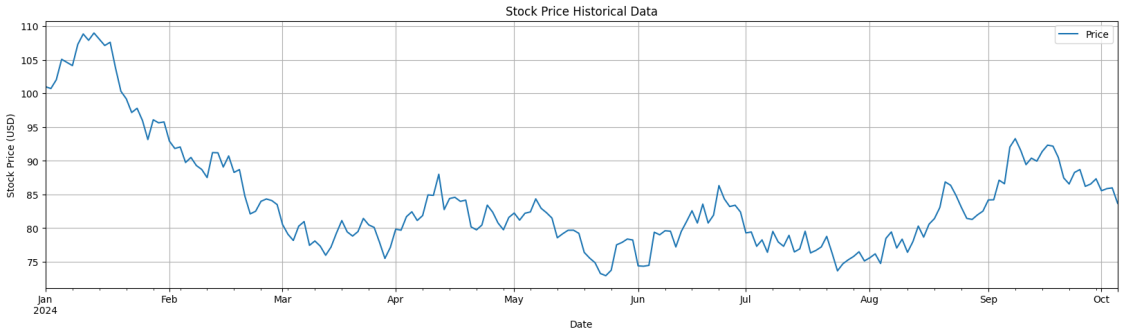
[ ]:
```
# Compute the periodogram
frequencies, power = periodogram(ts_sensor['Value'])

# Plot the periodogram

plt.figure(figsize=(10, 5))
plt.plot(frequencies, power, marker='o', linestyle='-')
plt.title("Periodogram (Frequency vs. Power Spectrum)")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.grid(True)
plt.show()
```



16 Exponential Weighted Moving Average (EWMA)

[30]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[31]:
```
# Load Dataset

stocks = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
 ↪public_files/master/Data%20Sets/stock_price_sample_03.csv',
                     index_col = 'Date', parse_dates = True)

stocks.plot(figsize = (20,5), grid = True, ylabel = 'Stock Price (USD)', title␣
 ↪= 'Stock Price Historical Data')
plt.show()
```



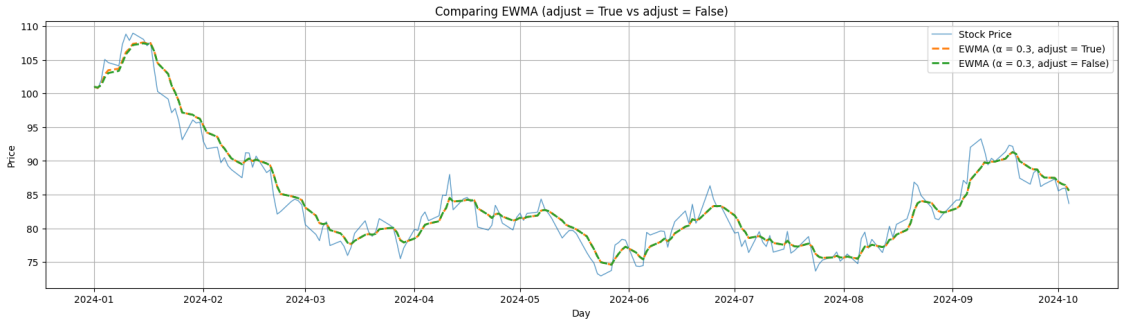
16.1 EWMA Implementation

Parameters

Parameter	Description	Notes:
com (center of Mass)	Alternative way to specify . com = (1/) - 1	Use only one: com, span, halflife, or alpha
span	Controls smoothing: = 2 / (span + 1)	Higher span = slower response
halflife	Defines the time it takes for a weight to reduce by half.	= 1 - e^(-ln(2) / halflife)
alpha	Directly set smoothing factor (0 < 1).	Most explicit way to set smoothing.
min_periods	Minimum number of observationsrequired before returning a value.	Default is 0 (returns values from the start).
adjust	True: Normalize weights (default) Weights are normalized so that they sum to 1. It behaves like weighted moving average,with older values still contributing	False: Uses simpler recursion formula (faster). Uses a simplified recursive formula that does not normalize past values. It treats it as a direct exponential smoothing Puts more emphasis on the most recent values,making it more responsive to changes
ignore_na	True: Ignores NaNs when computing the moving average.	Default is False (NaNs affect calculations).
axis	Specifies axis for calculation.	Default: 0 (row-wise for Series/DataFrame).
times	Uses a time series index for weight computation.	Useful for irregular time intervals.

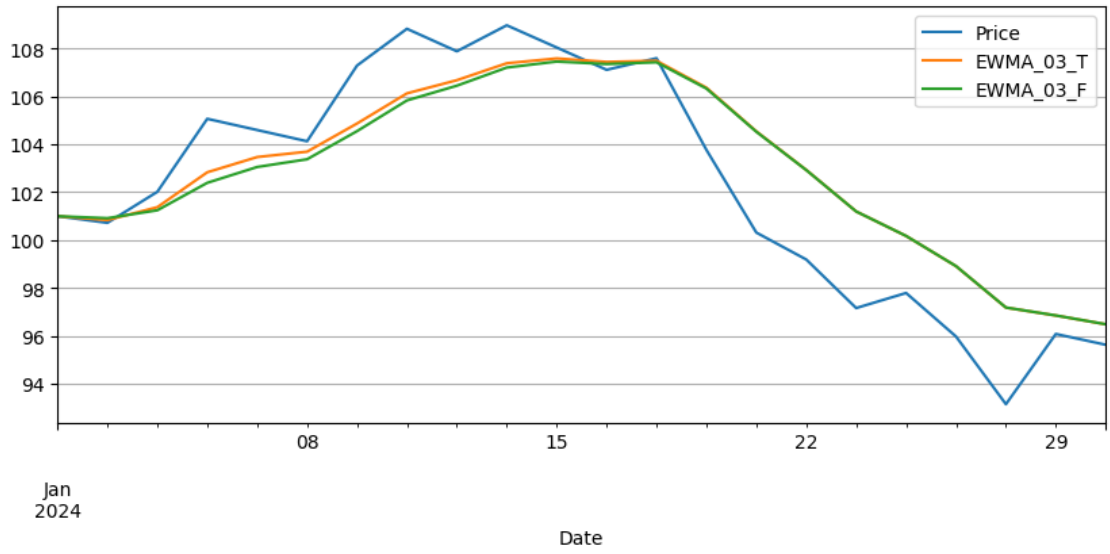
[32]:
```
# Compute EWMA with alpha = 0.3 and adjust = False
```

```
ewma_03T = 'EWMA_03_T'
ewma_03F = 'EWMA_03_F'

stocks[ewma_03T] = stocks['Price'].ewm(alpha = 0.3, adjust = True).mean()
stocks[ewma_03F] = stocks['Price'].ewm(alpha = 0.3, adjust = False).mean()

# Plot comparison between adjust=True and adjust = False

plt.figure(figsize = (20, 5))

plt.plot(stocks.index, stocks['Price'], label='Stock Price', linewidth = 1,␣
 ↪alpha = 0.7)
plt.plot(stocks.index, stocks[ewma_03T], label = 'EWMA (  = 0.3, adjust =␣
 ↪True)', linewidth = 2, linestyle = 'dashed')
plt.plot(stocks.index, stocks[ewma_03F], label = 'EWMA (  = 0.3, adjust =␣
 ↪False)', linewidth = 2, linestyle = 'dashed')

plt.xlabel('Day')
plt.ylabel('Price')
plt.title('Comparing EWMA (adjust = True vs adjust = False)')
plt.legend()
plt.grid()
plt.show()
```

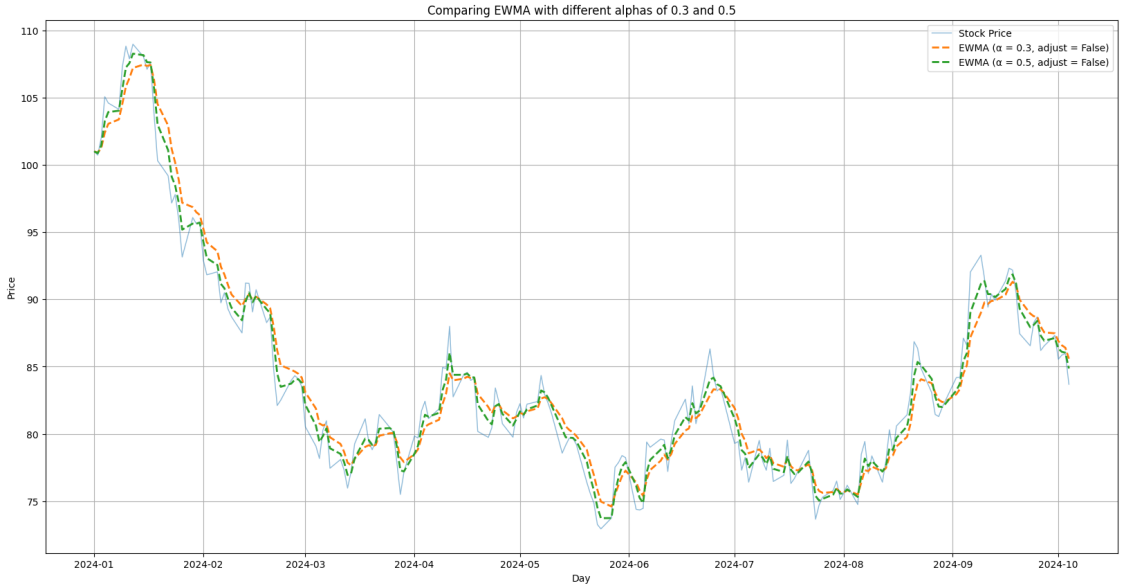


[33]:
```
# Zoomed-In

stocks.loc['2024-01-01': '2024-01-30'].plot(figsize = (10,4), grid = True)

# EWMA_03_T is much closer because the adjust = True considers the old value to␣
 ↪still contribute
```

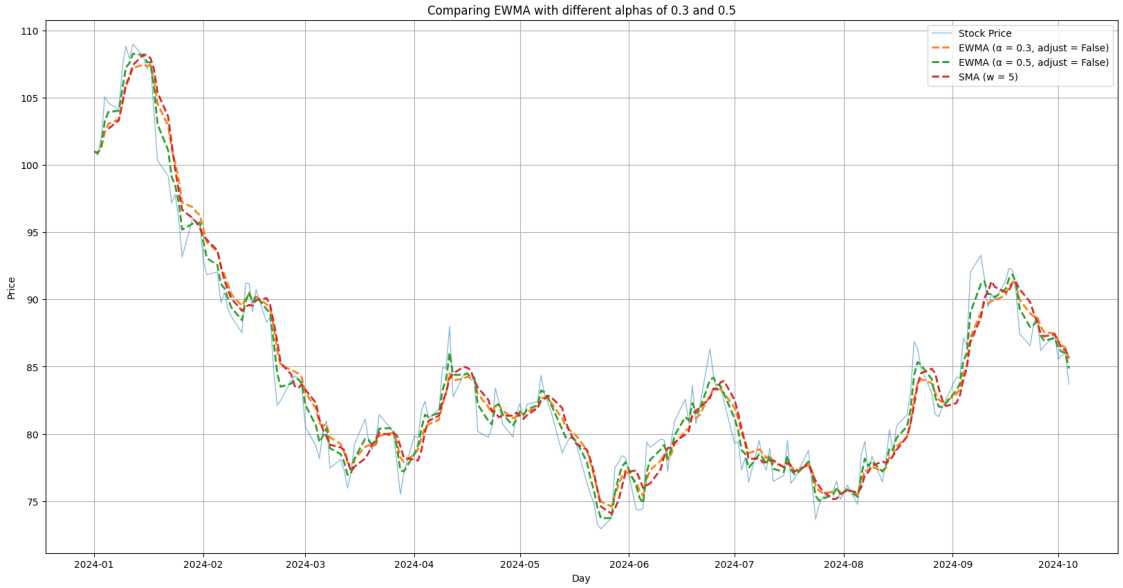
[33]: <Axes: xlabel='Date'>



[34]:
```
# Compute EWMA with alpha = 0.5 and adjust = False

ewma_05F = 'EWMA_05_F'

stocks[ewma_05F] = stocks['Price'].ewm(alpha = 0.5, adjust = False).mean()

# Plot comparison between adjust=True and adjust = False

plt.figure(figsize = (20, 10))

plt.plot(stocks.index, stocks['Price'], label='Stock Price', linewidth = 1,␣
↪alpha = 0.5)
plt.plot(stocks.index, stocks[ewma_03F], label = 'EWMA ( = 0.3, adjust =␣
↪False)', linewidth = 2, linestyle = 'dashed')
plt.plot(stocks.index, stocks[ewma_05F], label = 'EWMA ( = 0.5, adjust =␣
↪False)', linewidth = 2, linestyle = 'dashed')

plt.xlabel('Day')
plt.ylabel('Price')
plt.title('Comparing EWMA with different alphas of 0.3 and 0.5')
plt.legend()
plt.grid()
plt.show()
```

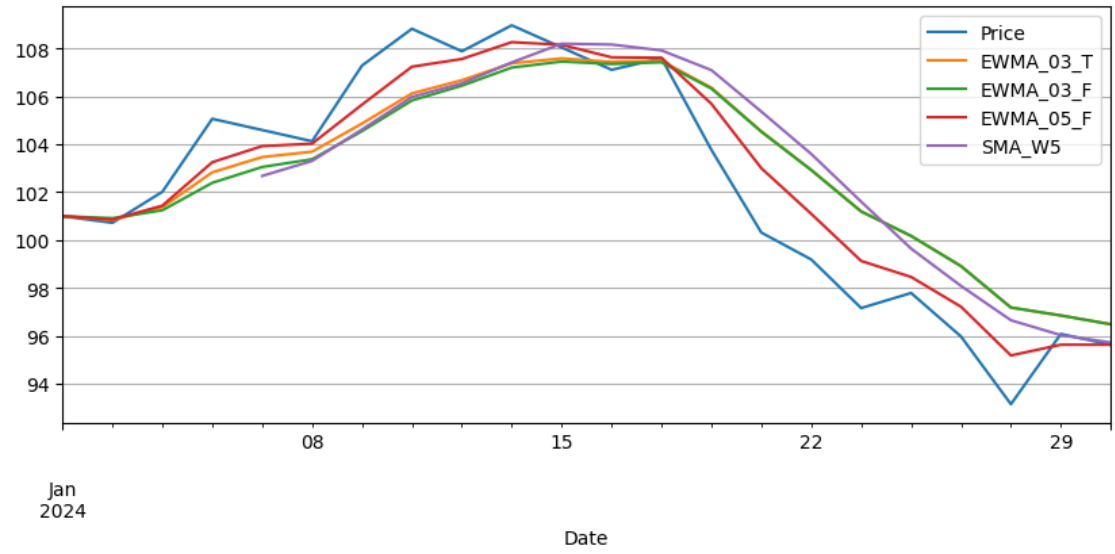


[35]:
```
# Compute EWMAs with SMA

sma_w5 = 'SMA_W5'

stocks[sma_w5] = stocks['Price'].rolling(window = 5).mean()

# Plot comparison between adjust=True and adjust = False

plt.figure(figsize = (20, 10))

plt.plot(stocks.index, stocks['Price'], label='Stock Price', linewidth = 1,␣
 ↪alpha = 0.5)
plt.plot(stocks.index, stocks[ewma_03F], label = 'EWMA ( = 0.3, adjust =␣
 ↪False)', linewidth = 2, linestyle = 'dashed')
plt.plot(stocks.index, stocks[ewma_05F], label = 'EWMA ( = 0.5, adjust =␣
 ↪False)', linewidth = 2, linestyle = 'dashed')
plt.plot(stocks.index, stocks[sma_w5], label = 'SMA (w = 5)', linewidth = 2,␣
 ↪linestyle = 'dashed')


plt.xlabel('Day')
plt.ylabel('Price')
plt.title('Comparing EWMA with different alphas of 0.3 and 0.5')
plt.legend()
plt.grid()
plt.show()
```



[36]:
```
# Zoomed-In

stocks.loc['2024-01-01': '2024-01-30'].plot(figsize = (10,4), grid = True)
```

[36]: <Axes: xlabel='Date'>



[37]:
```
# A. SMA lags Behind Trends
```

```
# Problem: SMA equally weights all past values, meaning it reacts slowly to new␣
 ↪price movements.

# Example:
# If the price suddenly drops, SMA will take longer to reflect this change.
# EWMA, on the other hand, assigns higher weights to recent prices, so it␣
 ↪adjusts much faster.
# Impact: SMA misses turning points in trends, while EWMA quickly adapts.

# ----

# B. SMA Ignores the Most Recent Data's Importance
# Problem: SMA treats old and new data the same, meaning a price from 20 days␣
 ↪ago has the same impact as today's price.

# Example:
# Imagine a 10-day SMA:
# The average is based on 10 past prices, dropping the oldest price each day.
# A sudden price spike today barely affects the SMA because it's diluted by the␣
 ↪past 9 days.
# EWMA gives recent data more weight, making it more responsive.
# Impact: EWMA better reflects conditions right now, while SMA is stale.

# ----

# C. SMA Discards Useful Old Data Too Abruptly

# Example:
# Suppose we have a 10-day SMA.
# On Day 11, the SMA completely removes Day 1's price instead of gradually␣
 ↪reducing its impact.
# EWMA gradually decreases the weight of older values instead of an abrupt␣
 ↪cut-off.
# Impact: EWMA provides a smoother, more natural trend, while SMA introduces␣
 ↪artificial jumps.

# ----

# D. SMA is Bad for Forecasting
# Problem: Since SMA treats all values equally, it cannot predict future trends␣
 ↪well.

# Example:
# In forecasting models, recent trends matter more.
# EWMA helps with momentum tracking, whereas SMA lags too much.
```

```
# Impact: EWMA is more predictive, while SMA just looks at raw history without␣
↪understanding trends.
```

16.2 Why is the Best Practice Range 0.2 - 0.5?

In real-world applications, values between 0.2 and 0.5 work best because they provide a balance between smoothness and responsiveness.

16.2.1 Effects of = 0.2 to 0.5:

1. Smooth Enough to Remove Noise

- These values keep short-term fluctuations under control.
- Small market spikes don't cause overreaction.

2. Fast Enough to Detect Trends

- Unlike SMA, EWMA with = 0.2 - 0.5 reacts to new data faster but without being too aggressive.

16.2.2 Better for Forecasting

- Financial models, weather forecasting, and anomaly detection all prefer a balanced .
- Common industry choices:
- = 0.2 → Used for long-term trend tracking.
- = 0.3 - 0.5 → Used in trading and short-term forecasting.

16.2.3 Conclusion:

If is too low (0.1) → EWMA is too slow, like SMA.

If is too high (0.9) → EWMA is too fast, like noisy raw data. = 0.2 - 0.5 is the sweet spot

17 MA

17.1 Moving Averages

In this section we'll compare Simple Moving Averages to Exponentially Weighted Moving Averages in terms of complexity and performance.

Related Functions:

pandas.DataFrame.rolling(window) Provides rolling window calculations pandas.DataFrame.ewm(span) Provides exponential weighted functions

17.1.1 Perform standard imports and load the dataset

For these examples we'll use the International Airline Passengers dataset, which gives monthly totals in thousands from January 1949 to December 1960.

[ ]:
```
import pandas as pd
import numpy as np
```

```
%matplotlib inline
```

[ ]:
```
airline = pd.read_csv('../Data/airline_passengers.
  ↪csv',index_col='Month',parse_dates=True)
```

[ ]:
```
airline.dropna(inplace=True)
```

[ ]:
```
airline.head()
```

[ ]:
```
            Thousands of Passengers
Month
1949-01-01                      112
1949-02-01                      118
1949-03-01                      132
1949-04-01                      129
1949-05-01                      121
```

---

18 SMA

18.1 Simple Moving Average

We've already shown how to create a simple moving average by applying a mean function to a rolling window.

For a quick review:

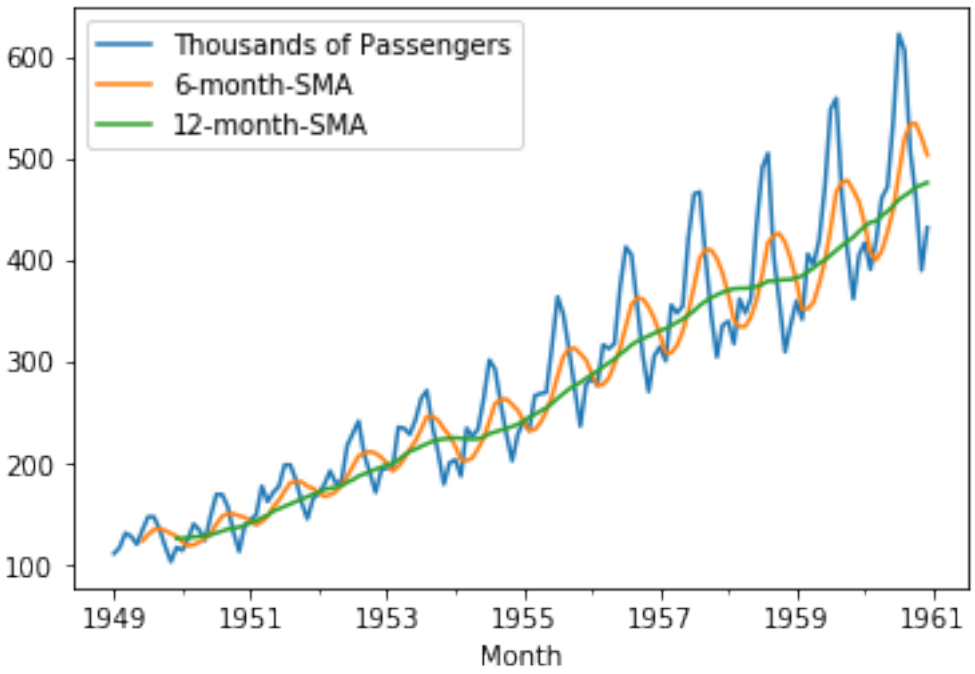
[ ]:
```
airline['6-month-SMA'] = airline['Thousands of Passengers'].rolling(window=6).
  ↪mean()
airline['12-month-SMA'] = airline['Thousands of Passengers'].rolling(window=12).
  ↪mean()
```

[ ]:
```
airline.head(15)
```

[ ]:
```
            Thousands of Passengers  6-month-SMA  12-month-SMA
Month
1949-01-01                      112          NaN           NaN
1949-02-01                      118          NaN           NaN
1949-03-01                      132          NaN           NaN
1949-04-01                      129          NaN           NaN
1949-05-01                      121          NaN           NaN
1949-06-01                      135   124.500000           NaN
1949-07-01                      148   130.500000           NaN
1949-08-01                      148   135.500000           NaN
1949-09-01                      136   136.166667           NaN
1949-10-01                      119   134.500000           NaN
1949-11-01                      104   131.666667           NaN
```

1949-12-01	118	128.833333	126.666667
1950-01-01	115	123.333333	126.916667
1950-02-01	126	119.666667	127.583333
1950-03-01	141	120.500000	128.333333

[ ]:
```
airline.plot();
```



---

19 EWMA

19.1 Exponentially Weighted Moving Average

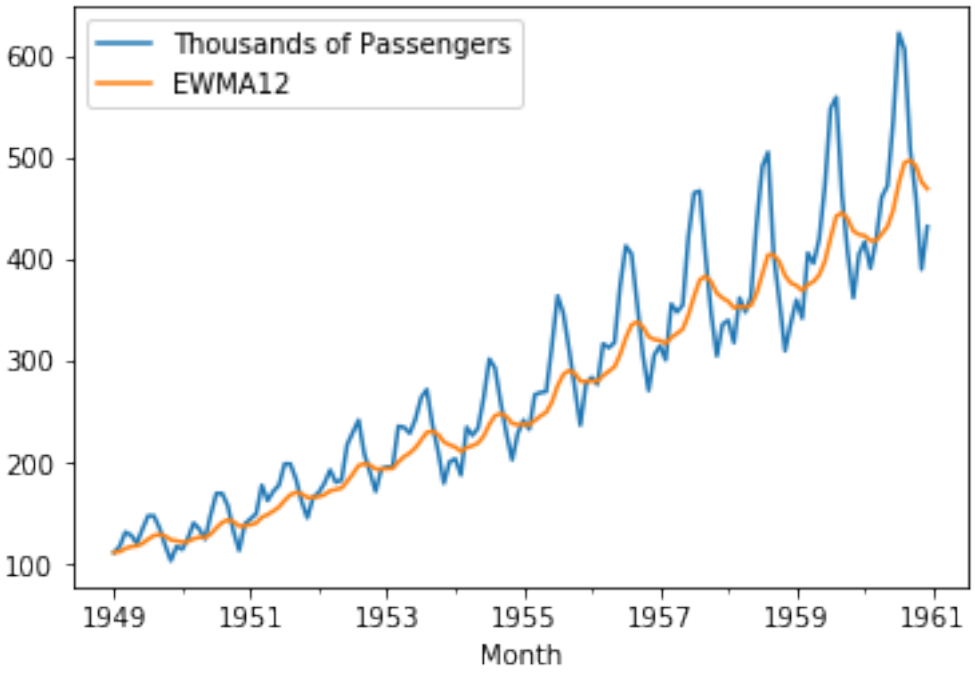
We just showed how to calculate the SMA based on some window. However, basic SMA has some weaknesses: * Smaller windows will lead to more noise, rather than signal * It will always lag by the size of the window * It will never reach to full peak or valley of the data due to the averaging. * Does not really inform you about possible future behavior, all it really does is describe trends in your data. * Extreme historical values can skew your SMA significantly

To help fix some of these issues, we can use an EWMA (Exponentially weighted moving average).

EWMA will allow us to reduce the lag effect from SMA and it will put more weight on values that occured more recently (by applying more weight to the more recent values, thus the name). The amount of weight applied to the most recent values will depend on the actual parameters used in the EWMA and the number of periods given a window size. Full details on Mathematics behind this can be found here. Here is the shorter version of the explanation behind EWMA.

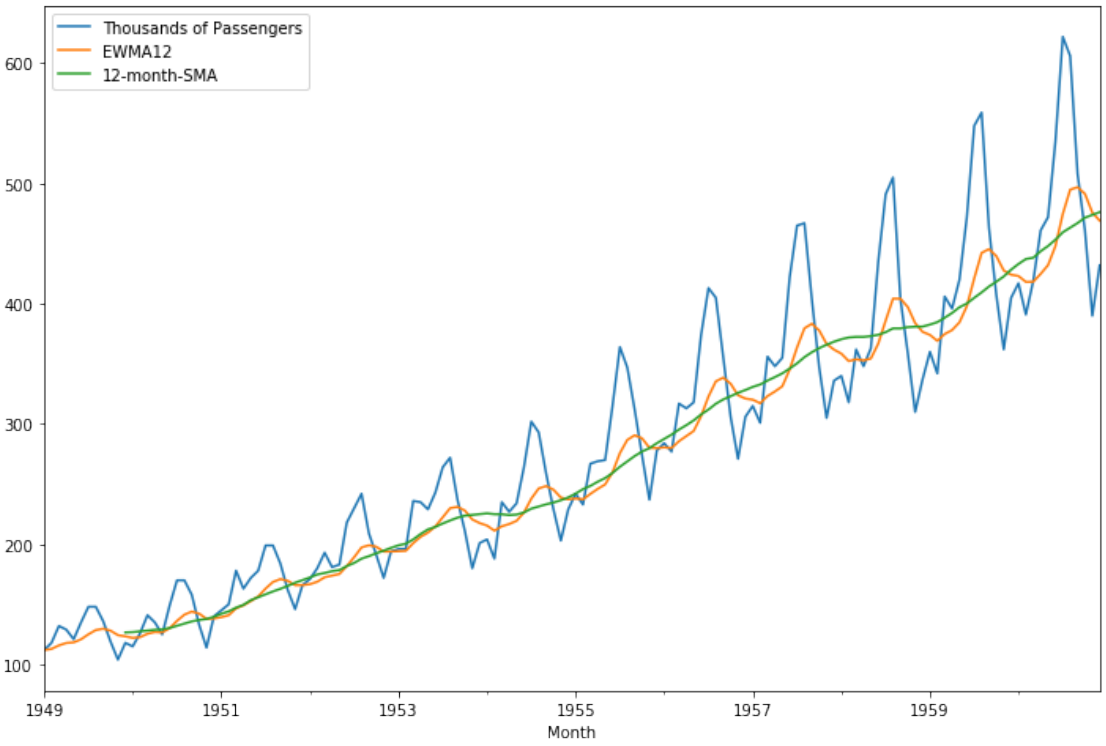
```
[ ]: airline['EWMA12'] = airline['Thousands of Passengers'].
       ↪ewm(span=12,adjust=False).mean()
```

```
[ ]: airline[['Thousands of Passengers','EWMA12']].plot();
```



19.2 Comparing SMA to EWMA

```
[ ]: airline[['Thousands of Passengers','EWMA12','12-month-SMA']].
       ↪plot(figsize=(12,8)).autoscale(axis='x',tight=True);
```



19.3 Simple Exponential Smoothing

The above example employed Simple Exponential Smoothing with one smoothing factor . Unfortunately, this technique does a poor job of forecasting when there is a trend in the data as seen above. In the next section we'll look at Double and Triple Exponential Smoothing with the Holt-Winters Methods.

[9]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

[10]:
```
# Load Dataset

# Load Another Dataset (This has missing dates)

airline = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
 ↪public_files/master/Data%20Sets/airline_passengers.csv',
                      index_col = 'Month', parse_dates = True)

airline.dropna() # Drop all missing data, if there is any
```

```
airline.head(5)
```

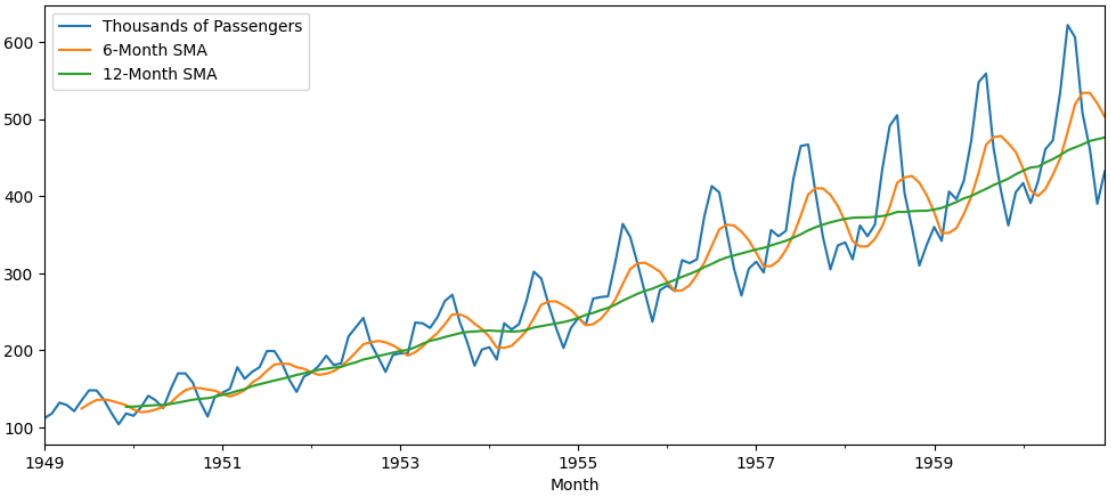
[10]:
```
            Thousands of Passengers
Month
1949-01-01                      112
1949-02-01                      118
1949-03-01                      132
1949-04-01                      129
1949-05-01                      121
```

20 EWMA (Exponentially Weighted Moving Average)

[11]:
```
# The Exponentially Weighted Moving Average (EWMA) is a powerful statistical␣
 ↪tool used in time series analysis,
# particularly for smoothing data and identifying underlying trends. Unlike␣
 ↪simple moving averages that give equal
# weight to all observations in the window, EWMA assigns exponentially␣
 ↪decreasing weights to older observations.
# This approach allows EWMA to be more responsive to recent changes in the␣
 ↪data, making it highly valuable for
# forecasting, and signal processing, among other applications.
```

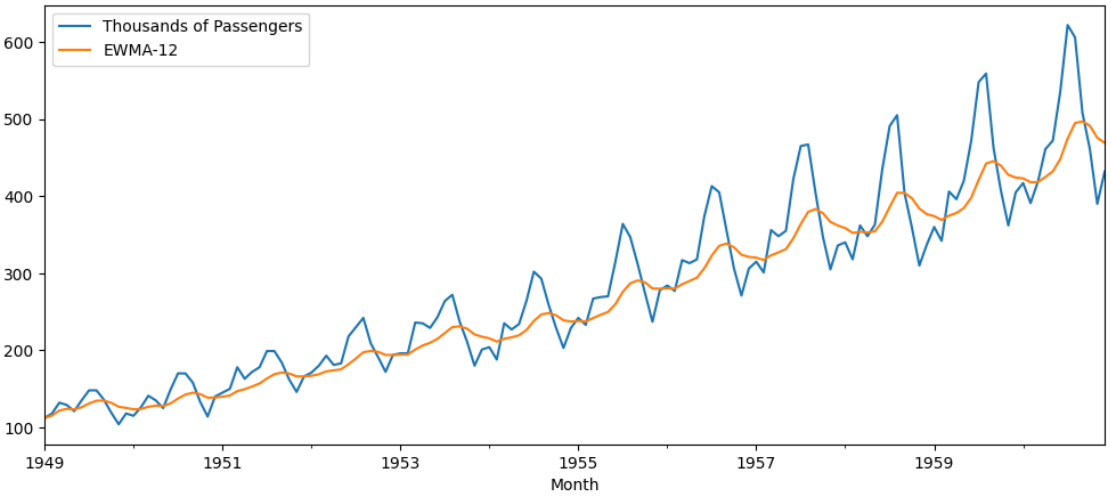
[12]:
```
# Global Graph Sizing

from pylab import rcParams
rcParams['figure.figsize'] = 12, 5


# Comparing SMA (rolling) to EWMA

airline['6-Month SMA'] = airline['Thousands of Passengers'].rolling(window = 6).
 ↪mean() # This is a 6-Month SMA
airline['12-Month SMA'] = airline['Thousands of Passengers'].rolling(window =␣
 ↪12).mean() # This is a 12-Month SMA

airline.plot()

# Responsiveness to Recent Data: By placing more weight on recent observations,␣
 ↪EWMA can quickly adapt to changes,
# making it particularly useful for tracking financial markets or economic␣
 ↪indicators where recent data may be more indicative of current trends.
```

[12]:
```
<Axes: xlabel='Month'>
```



[13]:
```
airline['EWMA-12'] = airline['Thousands of Passengers'].ewm(span = 12).mean() #␣
 ↪span is N EW moving average, 12 months == 1 Year

airline[['Thousands of Passengers', 'EWMA-12']].plot()
```

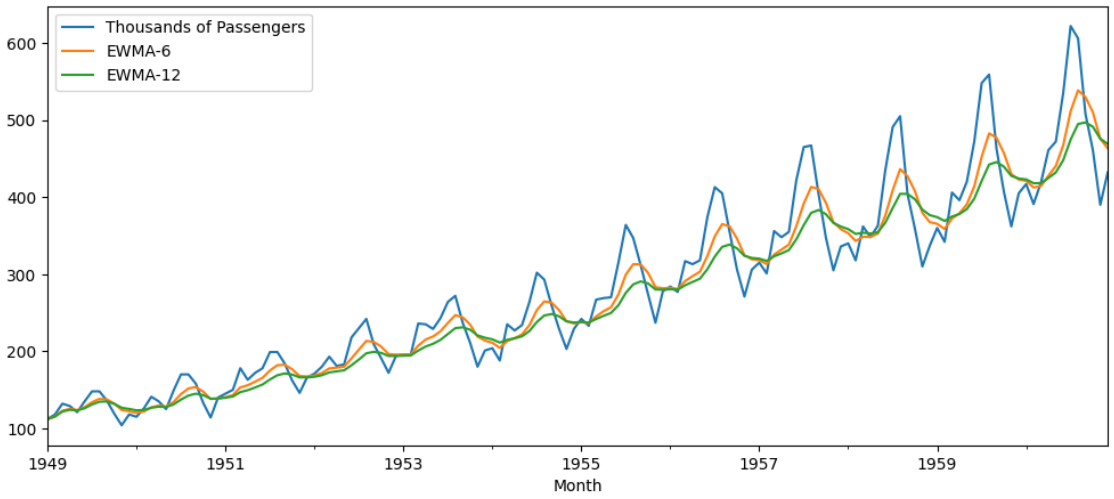
[13]: <Axes: xlabel='Month'>



[14]:
```
# EWMA Comparison 6 months vs 12 months

airline['EWMA-6'] = airline['Thousands of Passengers'].ewm(span = 6).mean() #␣
 ↪span is N EW moving average, 6 months
```

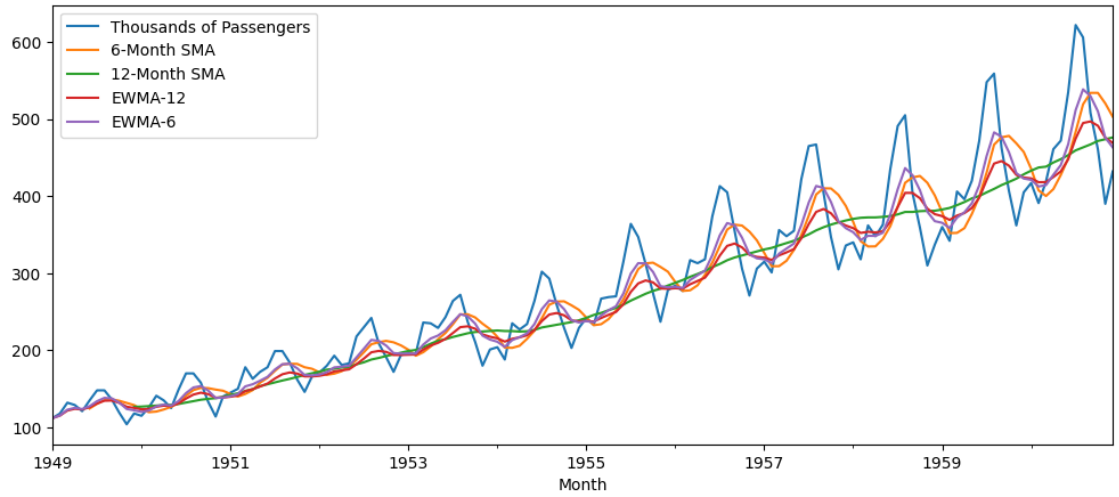
```
airline[['Thousands of Passengers', 'EWMA-6', 'EWMA-12']].plot()

# Minimal Lag: Compared to simple moving averages, EWMA introduces less lag␣
 ↪because it
# diminishes the influence of older data points more rapidly. This␣
 ↪characteristic is crucial for timely decision-making based on the analysis.

# EWMA can handle missing values more gracefully than simple moving averages␣
 ↪since it does not require a
# fixed number of observations to calculate the average. Each EWMA value is␣
 ↪computed using only the available data up to that point.
```

[14]: <Axes: xlabel='Month'>



[15]:
```
# Overall Comparison SMAs vs EWMAs

airline.plot()
```

[15]: <Axes: xlabel='Month'>



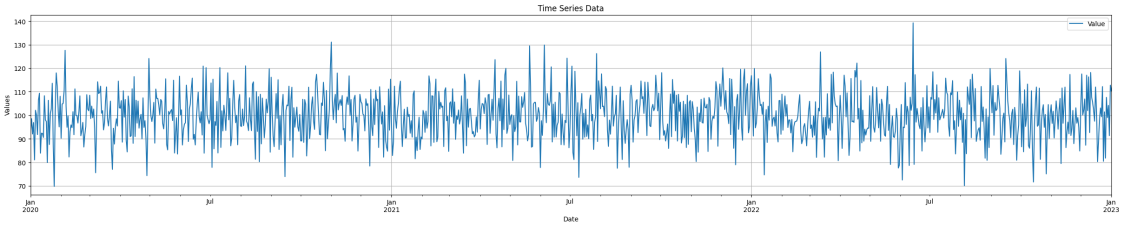
21 Single Exponential Smoothing (SES)

[24]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

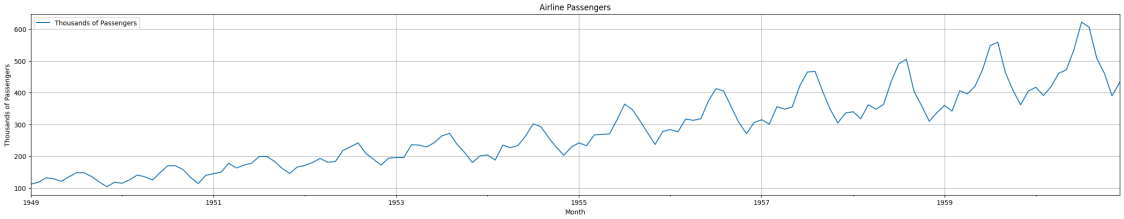
21.1 1st Case

21.1.1 Load datasets

[25]:
```
# Load dataset

ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/ts_ses_data-02.csv',
                      index_col = 'Date', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Values', title = 'Time Series␣
 ↪Data')
plt.show()
```



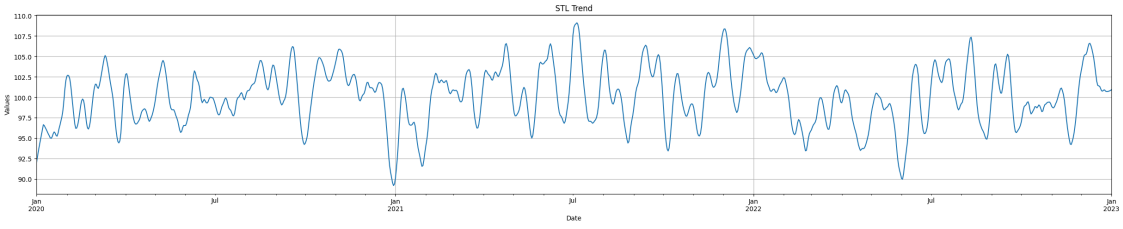
[26]:
```
# Load dataset

airline = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/
 ↪public_files/master/Data%20Sets/airline_passengers.csv',
                      index_col = 'Month', parse_dates = True)

airline.plot(figsize = (30,5), grid = True, ylabel = 'Thousands of Passengers',␣
 ↪title = 'Airline Passengers')
plt.show()
```



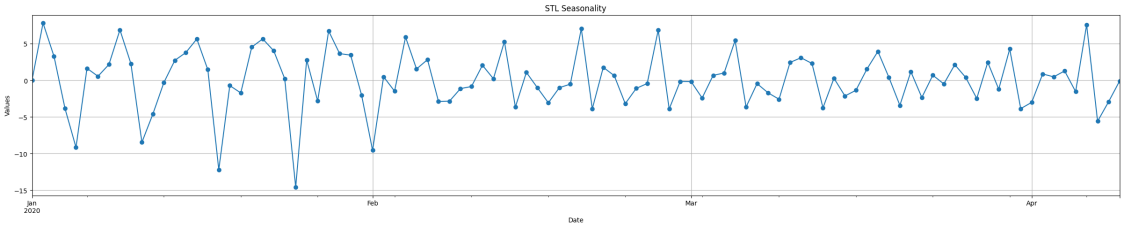
21.1.2 Check for trend and seasonality (1st data set)

[27]:
```
# STL Decomposition

from statsmodels.tsa.seasonal import STL

ts_stl = STL(ts["Value"], robust = True)  # Robust = better handling of outliers
decomp_stl = ts_stl.fit()
```

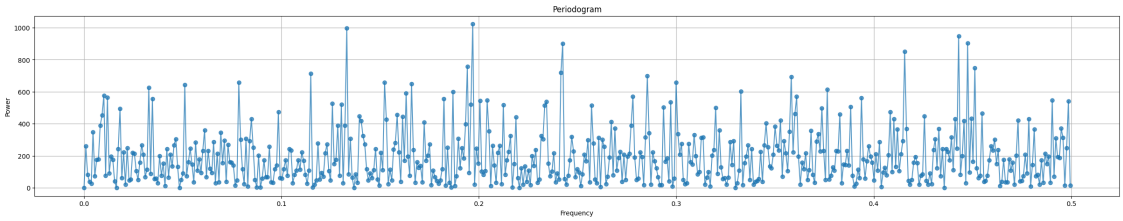
[28]:
```
# STL Trend

decomp_stl.trend.plot(figsize = (30,5), grid = True, ylabel = 'Values', title =␣
 ↪'STL Trend')
plt.show()

# No Clear Trend
```



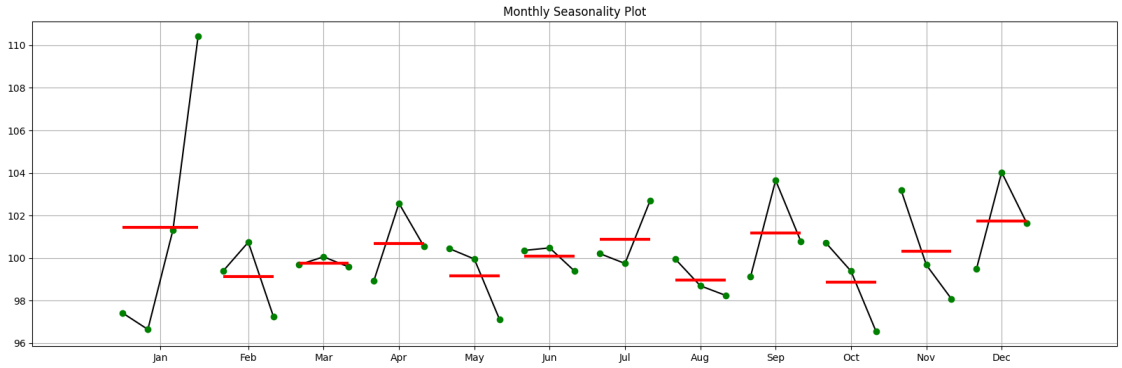
[29]:
```
# STL Seasonality

decomp_stl.seasonal[:100].plot(figsize = (30,5), grid = True, ylabel =␣
 ↪'Values', title = 'STL Seasonality', marker = 'o')
plt.show()

# No Clear Seasonality
```



[30]:
```
# Periodogram (for Seasonality)

from scipy.signal import periodogram

# Compute the periodogram
frequencies, power = periodogram(ts['Value'])

# Plot the periodogram
plt.figure(figsize=(30, 5))
plt.plot(frequencies, power, marker='o', linestyle='-', alpha = 0.75)
plt.title("Periodogram")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.grid(True)
plt.show()

# No Clear Seasonality
```

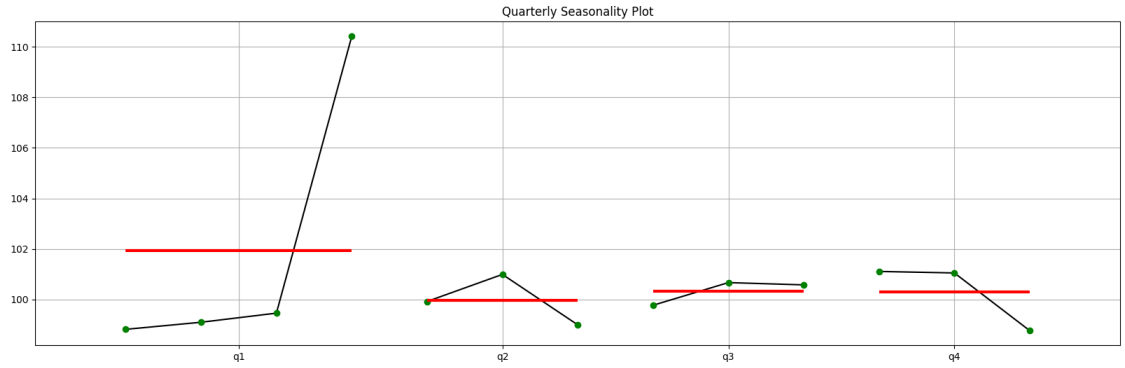


[31]:
```
# Month Plots

from statsmodels.graphics.tsaplots import month_plot

ts_monthly = ts['Value'].resample('ME').mean()

fig, ax = plt.subplots(figsize=(20, 6))
month_plot(ts_monthly, ax = ax)
plt.title("Monthly Seasonality Plot")
plt.grid()

for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent monthly disribution of values
# The red lines is the mean values for each month

# The red line does not show a perfectly smooth repeating pattern, but there␣
 ↪are some slight fluctuations.
# There are higher means in December, September, July, and January while lower␣
 ↪means in March, May, and August.
# However, these fluctuations are not very strong, meaning that the seasonality␣
 ↪is not very pronounced.
```



[32]:
```
# Quarter Plots

from statsmodels.graphics.tsaplots import quarter_plot
```

```
ts_quarter = ts['Value'].resample('QE').mean()

fig, ax = plt.subplots(figsize=(20, 6))
quarter_plot(ts_quarter, ax = ax)
plt.title("Quarterly Seasonality Plot")
plt.grid()

for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent monthly disribution of values
# The red lines is the mean values for each month

# Q1 (Jan-Mar) has a high spread, with some extreme values, making it more␣
 ↪variable than other quarters.
# Q2, Q3, and Q4 show narrower spreads, suggesting more stable values across␣
 ↪those periods
# If seasonality were strong, we would expect consistent patterns where certain␣
 ↪quarters always have higher or lower values.
```



21.1.3 Forecasting Using SES

1. Select a Model

[33]:
```
# We will use Single Exponential Smoothing or Simple Exponential Smoothing

from statsmodels.tsa.holtwinters import SimpleExpSmoothing
```

2. Split data into train & test sets

[34]:
```
# Splitting

forecast_horizon = 14 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 14 days
test_data = ts.iloc[-forecast_horizon:] # select the last 14 days

train_data = train_data.asfreq('D')
test_data = test_data.asfreq('D')
```

3. Fit model on training set

[35]:
```
ses_model = SimpleExpSmoothing(train_data['Value']).fit()

# ses_model.summary()
```

4. Evaluate model on test set (visually first)

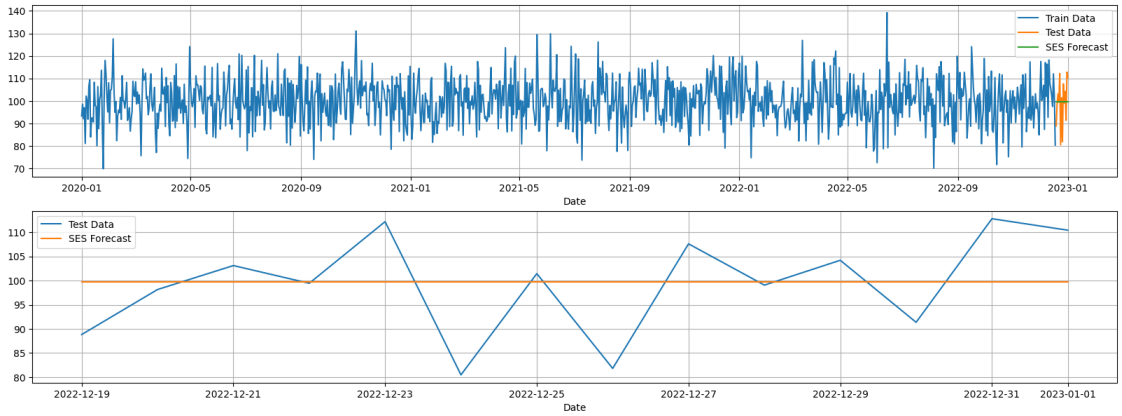
[36]:
```
# Forecast/Predict based on train_data

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[36]:
```
2022-12-19    99.733732
2022-12-20    99.733732
2022-12-21    99.733732
2022-12-22    99.733732
2022-12-23    99.733732
2022-12-24    99.733732
2022-12-25    99.733732
2022-12-26    99.733732
2022-12-27    99.733732
2022-12-28    99.733732
2022-12-29    99.733732
2022-12-30    99.733732
2022-12-31    99.733732
2023-01-01    99.733732
Freq: D, dtype: float64
```

[37]:
```
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data['Value'], label = 'Train Data')
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
```

```
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[37]:
```
Text(0.5, 0, 'Date')
```



5. Re-fit model on entire data set

[38]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('D')

ses_model = SimpleExpSmoothing(ts['Value']).fit()
```

6. Forecast for future data

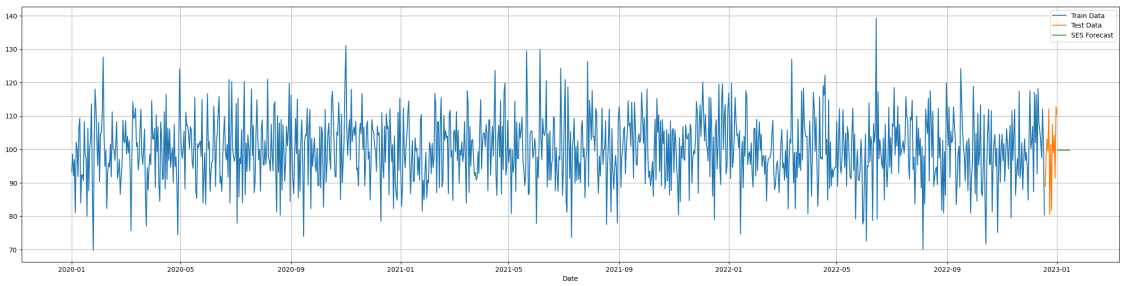
[39]:
```
# We forecast using the entire dataset

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[39]:
```
2023-01-02    99.695687
2023-01-03    99.695687
2023-01-04    99.695687
2023-01-05    99.695687
2023-01-06    99.695687
2023-01-07    99.695687
2023-01-08    99.695687
2023-01-09    99.695687
```

```
2023-01-10    99.695687
2023-01-11    99.695687
2023-01-12    99.695687
2023-01-13    99.695687
2023-01-14    99.695687
2023-01-15    99.695687
Freq: D, dtype: float64
```

[40]:
```
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(1, 1, 1)
plt.plot(train_data.index, train_data['Value'], label = 'Train Data')
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[40]:
```
Text(0.5, 0, 'Date')
```



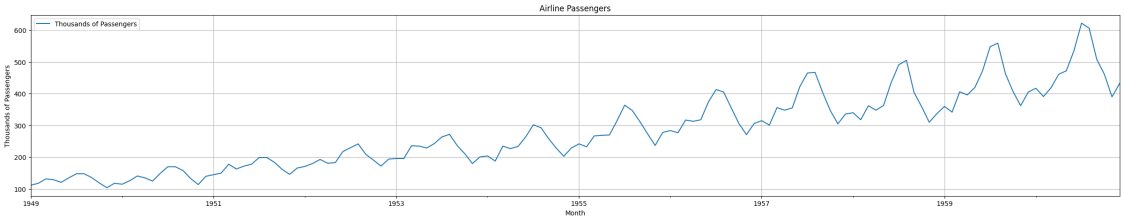
22 Single Exponential Smoothing (SES)

[59]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

22.0.1 Load datasets

[60]:
```
# Load dataset
```

```
ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/airline_passengers.csv',
                      index_col = 'Month', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Thousands of Passengers',␣
 ↪title = 'Airline Passengers')
plt.show()
```



22.0.2 Forecasting Using SES

1. Select a Model

[61]:
```
# We will use Single Exponential Smoothing or Simple Exponential Smoothing

from statsmodels.tsa.holtwinters import SimpleExpSmoothing
```

2. Split data into train & test sets

[62]:
```
# Splitting

forecast_horizon = 12 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 quarters
test_data = ts.iloc[-forecast_horizon:] # select the last 12 quarters

train_data = train_data.asfreq('QS')
test_data = test_data.asfreq('QS')
```

3. Fit model on training set

[63]:
```
ses_model = SimpleExpSmoothing(train_data).fit()

# ses_model.summary()
```

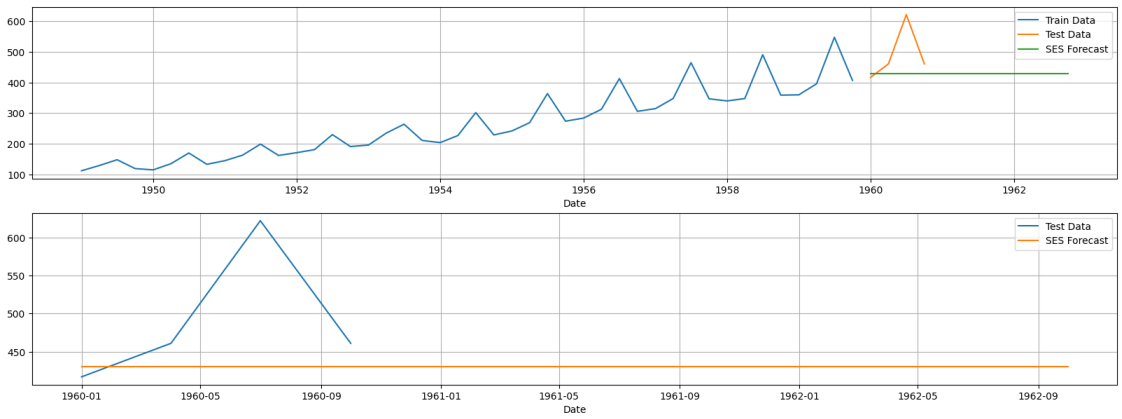
4. Evaluate model on test set (visually first)

[64]:
```
# Forecast/Predict based on train_data

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[64]:
```
1960-01-01    430.379512
1960-04-01    430.379512
1960-07-01    430.379512
1960-10-01    430.379512
1961-01-01    430.379512
1961-04-01    430.379512
1961-07-01    430.379512
1961-10-01    430.379512
1962-01-01    430.379512
1962-04-01    430.379512
1962-07-01    430.379512
1962-10-01    430.379512
Freq: QS-JAN, dtype: float64
```

[65]:
```
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[65]:
```
Text(0.5, 0, 'Date')
```



5. Re-fit model on entire data set

[66]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('QS')

ses_model = SimpleExpSmoothing(ts).fit()
```

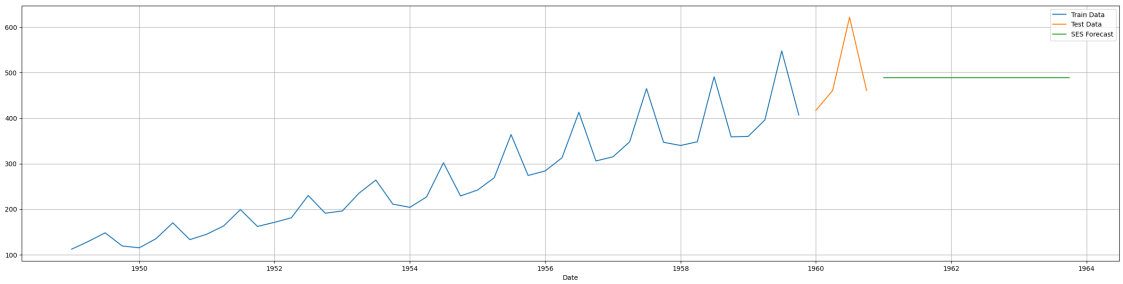
6. Forecast for future data

[67]:
```
# We forecast using the entire dataset

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[67]:
```
1961-01-01    489.248896
1961-04-01    489.248896
1961-07-01    489.248896
1961-10-01    489.248896
1962-01-01    489.248896
1962-04-01    489.248896
1962-07-01    489.248896
1962-10-01    489.248896
1963-01-01    489.248896
1963-04-01    489.248896
1963-07-01    489.248896
1963-10-01    489.248896
Freq: QS-JAN, dtype: float64
```

[68]:
```
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(1, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[68]:
```
Text(0.5, 0, 'Date')
```



22.1 2nd Case (Airline Passengers)

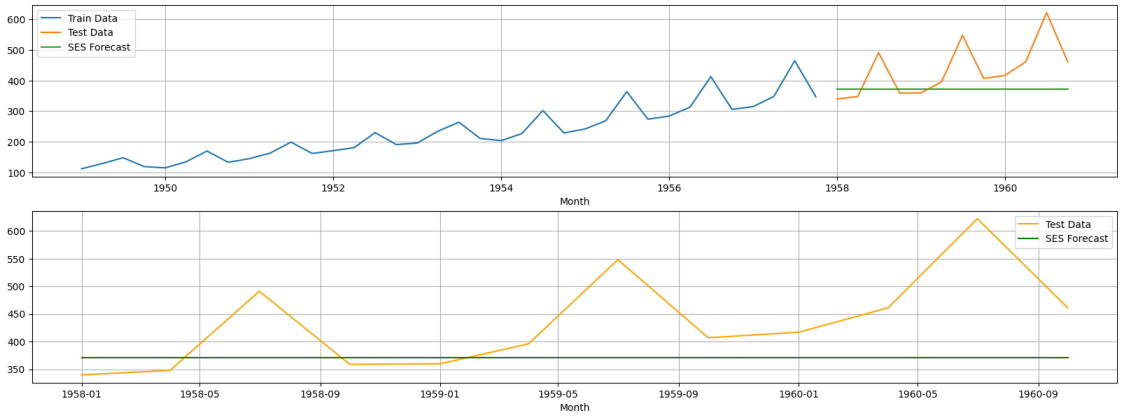
[69]:
```
# 1. We have imported a model already
# 2. Splitting

forecast_horizon = 12 # or quarters

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 quarters
test_data = ts.iloc[-forecast_horizon:] # select the last last 12 quarters

train_data = train_data.asfreq('QS')
test_data = test_data.asfreq('QS')
```

[70]:
```
# Fit the model on training

ses_model = SimpleExpSmoothing(train_data).fit()

# Forecast/Predict based on train_data

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[70]:
```
1958-01-01    371.000445
1958-04-01    371.000445
1958-07-01    371.000445
1958-10-01    371.000445
1959-01-01    371.000445
1959-04-01    371.000445
1959-07-01    371.000445
1959-10-01    371.000445
1960-01-01    371.000445
1960-04-01    371.000445
1960-07-01    371.000445
1960-10-01    371.000445
Freq: QS-JAN, dtype: float64
```

[71]:
```
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Month')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data', color = 'orange')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast', color = 'green')
plt.legend()
plt.grid()
plt.xlabel('Month')
```

[71]:
```
Text(0.5, 0, 'Month')
```

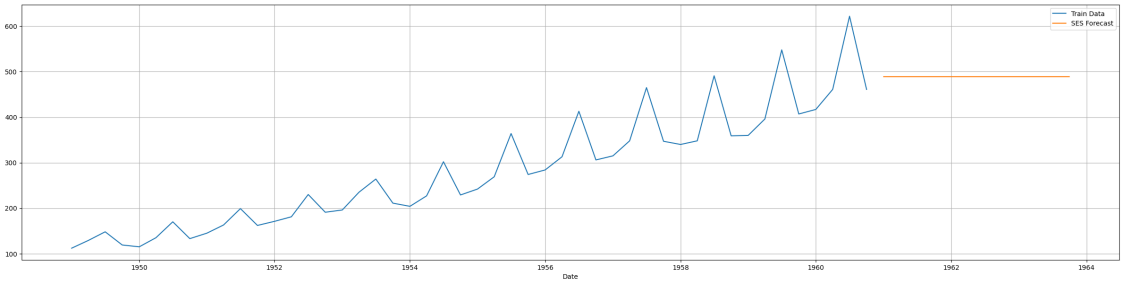


[72]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('QS')

ses_model = SimpleExpSmoothing(ts).fit()

# We forecast using the entire dataset

ses_pred = ses_model.forecast(forecast_horizon)
ses_pred
```

[72]:
```
1961-01-01    489.248896
1961-04-01    489.248896
1961-07-01    489.248896
1961-10-01    489.248896
1962-01-01    489.248896
1962-04-01    489.248896
1962-07-01    489.248896
1962-10-01    489.248896
1963-01-01    489.248896
1963-04-01    489.248896
1963-07-01    489.248896
1963-10-01    489.248896
Freq: QS-JAN, dtype: float64
```

[75]:
```
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(1, 1, 1)
plt.plot(ts.index, ts, label = 'Train Data')
plt.plot(ses_pred.index, ses_pred, label = 'SES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[75]:
```
Text(0.5, 0, 'Date')
```



[74]:
```
# Why the forecast is bad? SES does not account for trend and seasonality
```

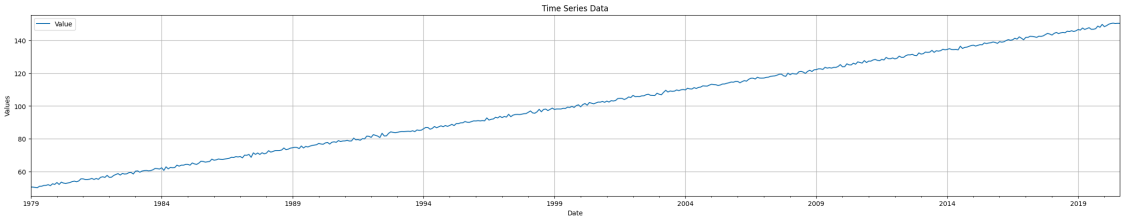
23 Double Exponential Smoothing (DES)

[ ]:
```
# Imports

import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

23.0.1 Load datasets

[ ]:
```
# Load dataset

ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/ts_des_data-01.csv',
                      index_col = 'Date', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Values', title = 'Time Series␣
 ↪Data')
plt.show()
ts
```



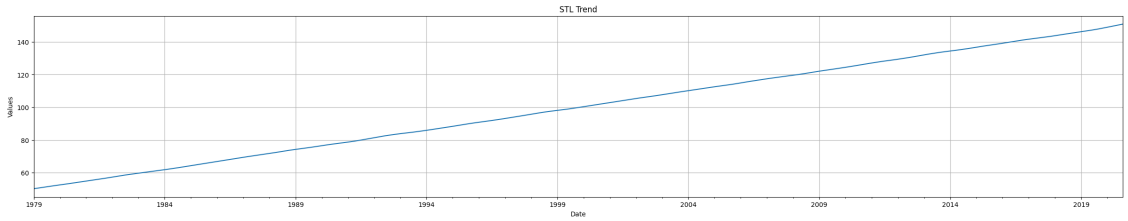
[ ]:
```
                Value
Date
1979-01-31   50.699678
1979-02-28   50.662718
1979-03-31   50.430617
1979-04-30   50.277734
1979-05-31   51.150715
…                  …
2020-04-30  150.201443
2020-05-31  150.429549
2020-06-30  150.203382
2020-07-31  150.311630
2020-08-31  150.296263

[500 rows x 1 columns]
```

23.0.2 Check for trend and seasonality

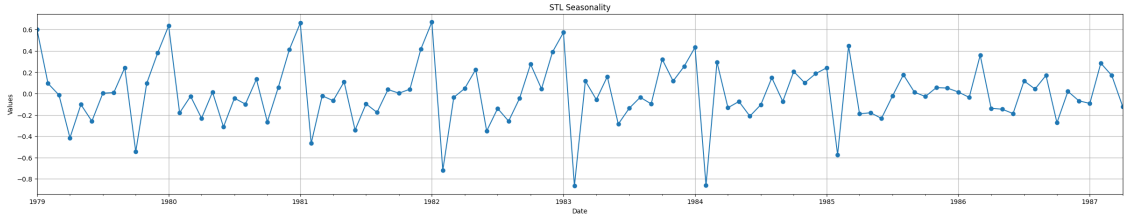
[ ]:
```
# STL Decomposition

from statsmodels.tsa.seasonal import STL

ts_stl = STL(ts["Value"], robust = True)  # Robust = better handling of outliers
```

```
decomp_stl = ts_stl.fit()
```

[ ]:
```
# STL Trend

decomp_stl.trend.plot(figsize = (30,5), grid = True, ylabel = 'Values', title =␣
↪'STL Trend')
plt.show()

# It has a clear trend (upward)
```

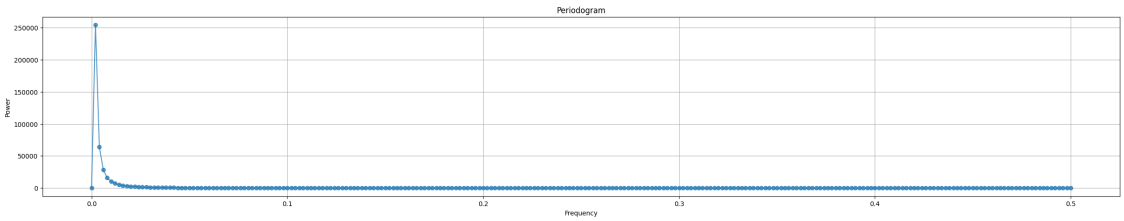


[ ]:
```
# STL Seasonality

decomp_stl.seasonal[:100].plot(figsize = (30,5), grid = True, ylabel =␣
↪'Values', title = 'STL Seasonality', marker = 'o')
plt.show()

# No Clear Seasonality
```

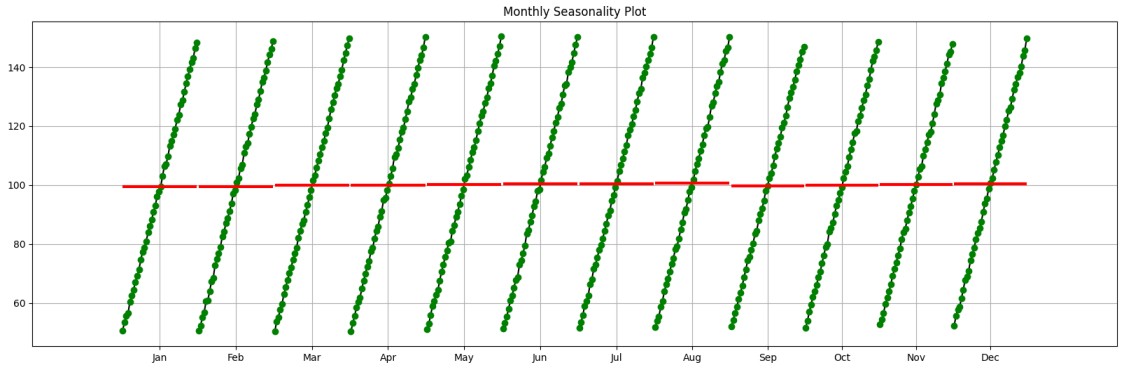


[ ]:
```
# Periodogram (for Seasonality)

from scipy.signal import periodogram

# Compute the periodogram
frequencies, power = periodogram(ts['Value'])

# Plot the periodogram
plt.figure(figsize=(30, 5))
plt.plot(frequencies, power, marker='o', linestyle='-', alpha = 0.75)
```

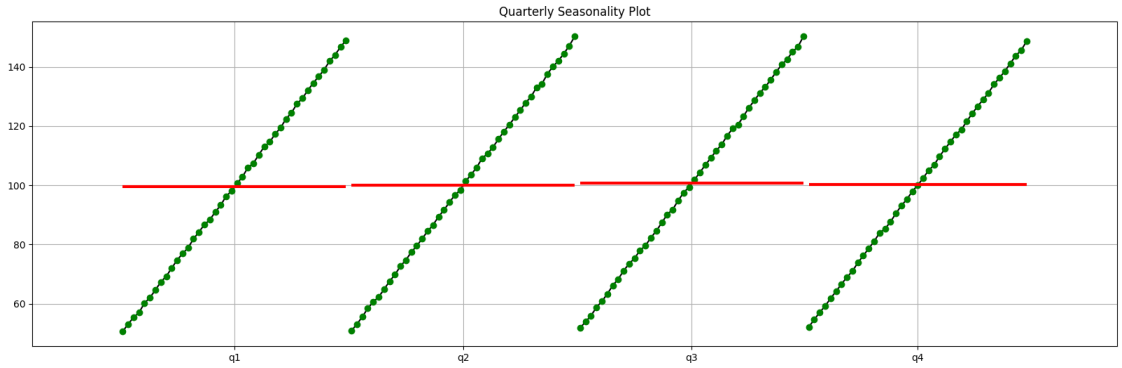
```
plt.title("Periodogram")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.grid(True)
plt.show()

# It exhibits seasonality at the start but does not sustain it on the later␣
 ↪part of the series
```



[ ]:
```
# Month Plots

from statsmodels.graphics.tsaplots import month_plot

ts_monthly = ts['Value'].resample('MS').mean()

fig, ax = plt.subplots(figsize=(20, 6))
month_plot(ts_monthly, ax = ax)
plt.title("Monthly Seasonality Plot")
plt.grid()

for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent monthly disribution of values
# The red lines is the mean values for each month

# The red line does not show a perfectly smooth repeating pattern, no␣
 ↪considerable fluctuations.
# However, these fluctuations are not very strong, meaning that the seasonality␣
 ↪is not very pronounced.
```



[ ]:
```
# Quarter Plots

from statsmodels.graphics.tsaplots import quarter_plot

ts_quarter = ts['Value'].resample('QE').mean()

fig, ax = plt.subplots(figsize=(20, 6))
quarter_plot(ts_quarter, ax = ax)
plt.title("Quarterly Seasonality Plot")
plt.grid()

for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent quarterly disribution of values
# The red lines is the mean values for each quarter

# We would expect consistent patterns where certain quarters always have higher␣
  ↪or lower values. But the graphs shows it doe not have
```



23.0.3 Forecasting Using DES

1. Select a Model

```
[ ]: # We will use Double Exponential Smoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

2. Split data into train & test sets

```
[ ]: # Splitting

forecast_horizon = 12 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 months
test_data = ts.iloc[-forecast_horizon:] # select the last 12 months

train_data = train_data.asfreq('ME')
test_data = test_data.asfreq('ME')
```

3. Fit model on training set

```
[ ]: des_model = ExponentialSmoothing(train_data['Value'], trend = 'additive').fit()

# des_model_.summary()
```

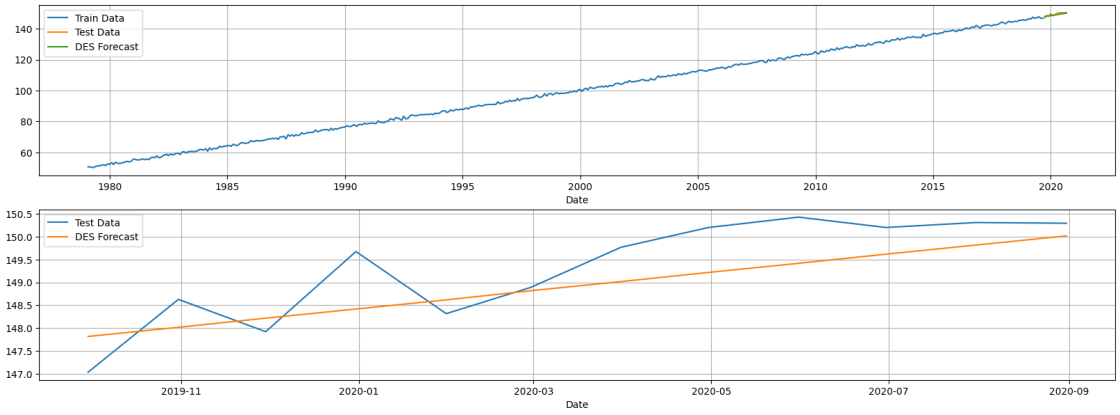
4. Evaluate model on test set (visually first)

```
[ ]: # Forecast/Predict based on train_data

des_pred = des_model.forecast(forecast_horizon)
des_pred
```

```
[ ]: 2019-09-30    147.815586
     2019-10-31    148.015882
     2019-11-30    148.216179
     2019-12-31    148.416475
```

```
2020-01-31    148.616771
2020-02-29    148.817068
2020-03-31    149.017364
2020-04-30    149.217660
2020-05-31    149.417957
2020-06-30    149.618253
2020-07-31    149.818549
2020-08-31    150.018846
Freq: ME, dtype: float64
```

[ ]:
```python
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data['Value'], label = 'Train Data')
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
plt.show()
```



5. Re-fit model on entire data set

[ ]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('ME')

des_model = ExponentialSmoothing(ts['Value'], trend = 'additive').fit()
```

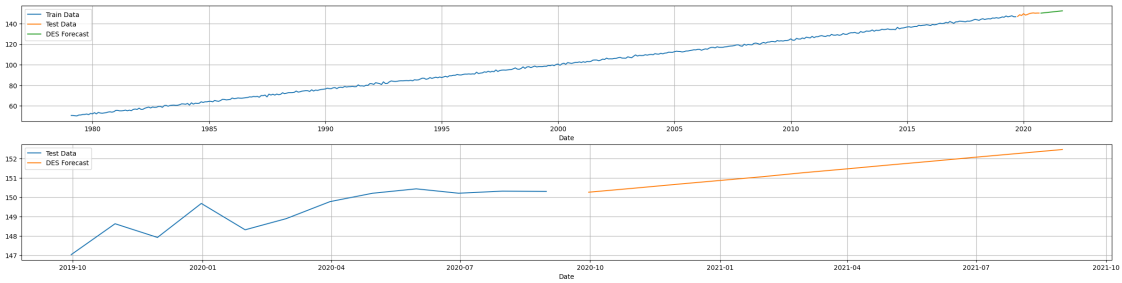
6. Forecast for future data

[ ]:
```
# We forecast using the entire dataset

des_pred = des_model.forecast(forecast_horizon)
des_pred
```

[ ]:
```
2020-09-30    150.256053
2020-10-31    150.456459
2020-11-30    150.656866
2020-12-31    150.857272
2021-01-31    151.057679
2021-02-28    151.258085
2021-03-31    151.458492
2021-04-30    151.658898
2021-05-31    151.859304
2021-06-30    152.059711
2021-07-31    152.260117
2021-08-31    152.460524
Freq: ME, dtype: float64
```

[ ]:
```
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data['Value'], label = 'Train Data')
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[ ]:
```
Text(0.5, 0, 'Date')
```

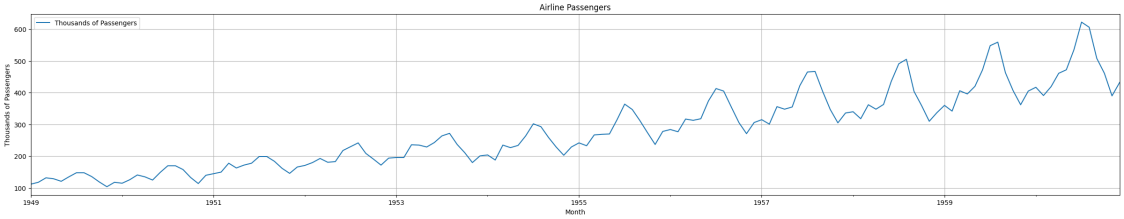


24 Double Exponential Smoothing (DES)

```
[ ]: # Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

24.0.1 Load datasets

```
[ ]: # Load dataset

ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
  ↪master/Data%20Sets/airline_passengers.csv',
                     index_col = 'Month', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Thousands of Passengers',␣
  ↪title = 'Airline Passengers')
plt.show()
```



24.0.2 Forecasting Using SES

1. Select a Model

```
[ ]: # We will use Double Exponential Smoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

2. Split data into train & test sets

```
[ ]: # Splitting

forecast_horizon = 36 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 months
test_data = ts.iloc[-forecast_horizon:] # select the last 12 months

train_data = train_data.asfreq('MS')
test_data = test_data.asfreq('MS')
```

3. Fit model on training set

```
[ ]: des_model = ExponentialSmoothing(train_data, trend = 'multiplicative').fit()

# des_model.summary()
```

4. Evaluate model on test set (visually first)

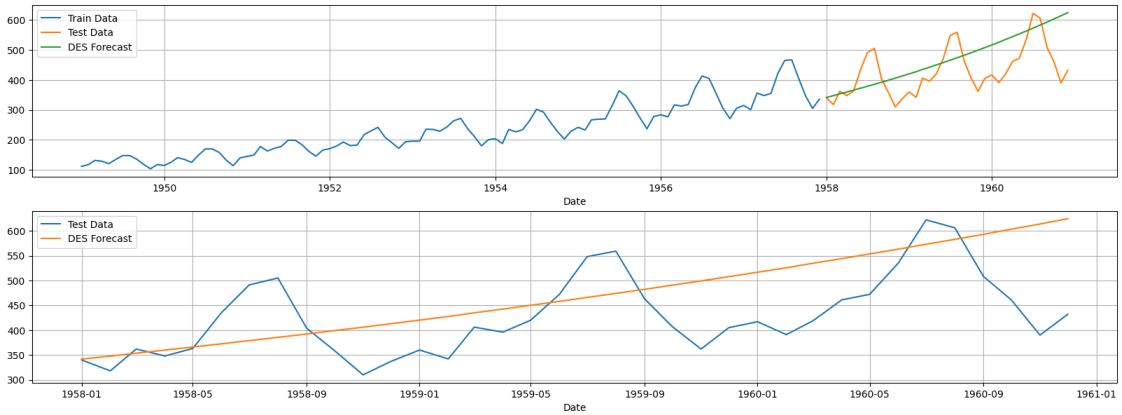
```
[ ]: # Forecast/Predict based on train_data

des_pred = des_model.forecast(forecast_horizon)
des_pred
```

```
[ ]: 1958-01-01    341.706530
     1958-02-01    347.641683
     1958-03-01    353.679925
     1958-04-01    359.823046
     1958-05-01    366.072868
     1958-06-01    372.431244
     1958-07-01    378.900060
     1958-08-01    385.481233
     1958-09-01    392.176716
     1958-10-01    398.988494
     1958-11-01    405.918587
     1958-12-01    412.969049
     1959-01-01    420.141972
     1959-02-01    427.439483
     1959-03-01    434.863745
     1959-04-01    442.416960
     1959-05-01    450.101368
     1959-06-01    457.919248
     1959-07-01    465.872918
     1959-08-01    473.964737
     1959-09-01    482.197103
     1959-10-01    490.572459
     1959-11-01    499.093288
     1959-12-01    507.762116
     1960-01-01    516.581514
```

```
1960-02-01    525.554098
1960-03-01    534.682528
1960-04-01    543.969511
1960-05-01    553.417801
1960-06-01    563.030200
1960-07-01    572.809559
1960-08-01    582.758776
1960-09-01    592.880803
1960-10-01    603.178641
1960-11-01    613.655344
1960-12-01    624.314018
Freq: MS, dtype: float64
```

[ ]:
```
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[ ]:
```
Text(0.5, 0, 'Date')
```



5. Re-fit model on entire data set

[ ]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('MS')

des_model = ExponentialSmoothing(ts, trend = 'multiplicative').fit()
```

6. Forecast for future data

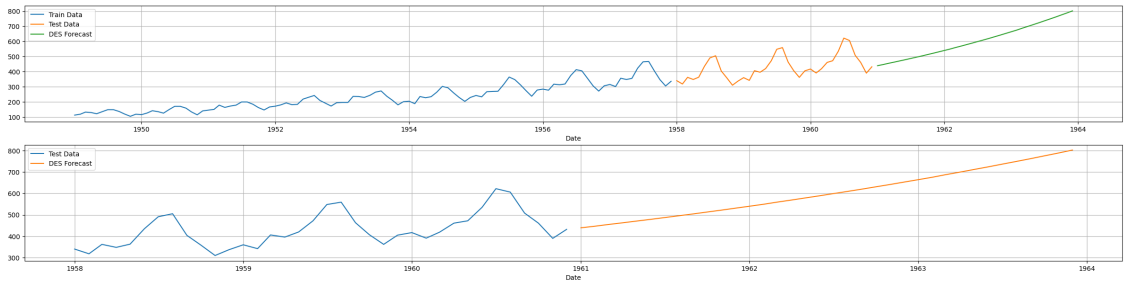
[ ]:
```
# We forecast using the entire dataset

des_pred = des_model.forecast(forecast_horizon)
des_pred
```

[ ]:
```
1961-01-01    439.319912
1961-02-01    446.944038
1961-03-01    454.700476
1961-04-01    462.591523
1961-05-01    470.619513
1961-06-01    478.786825
1961-07-01    487.095875
1961-08-01    495.549124
1961-09-01    504.149073
1961-10-01    512.898270
1961-11-01    521.799303
1961-12-01    530.854808
1962-01-01    540.067466
1962-02-01    549.440004
1962-03-01    558.975197
1962-04-01    568.675867
1962-05-01    578.544886
1962-06-01    588.585176
1962-07-01    598.799708
1962-08-01    609.191508
1962-09-01    619.763651
1962-10-01    630.519267
1962-11-01    641.461541
1962-12-01    652.593710
1963-01-01    663.919072
1963-02-01    675.440978
1963-03-01    687.162840
1963-04-01    699.088127
1963-05-01    711.220371
1963-06-01    723.563162
1963-07-01    736.120155
1963-08-01    748.895066
```

```
1963-09-01    761.891678
1963-10-01    775.113839
1963-11-01    788.565461
1963-12-01    802.250529
Freq: MS, dtype: float64
```

[ ]:
```python
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(des_pred.index, des_pred, label = 'DES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[ ]:
```
Text(0.5, 0, 'Date')
```



[ ]:
```python
# Why the forecast is bad? DES accounts for trend but not seasonality
```

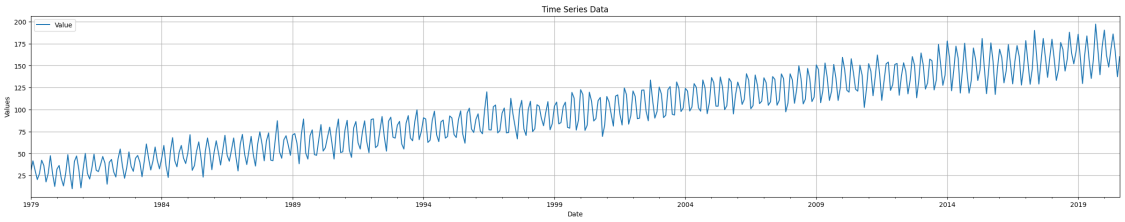
25 Triple Exponential Smoothing (TES)

[43]:
```python
# Imports

import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

25.0.1 Load datasets

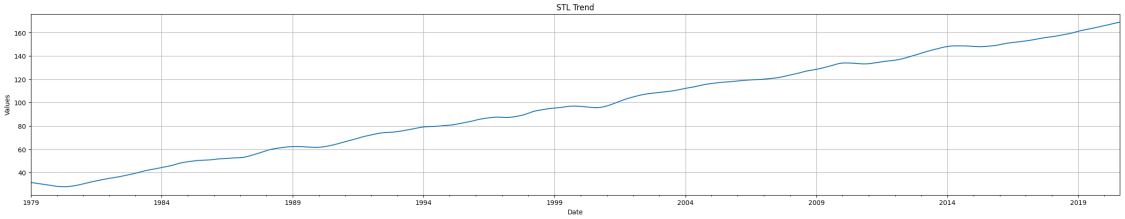
[44]:
```
# Load dataset

ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/ts_tes_data-01.csv',
                      index_col = 'Date', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Values', title = 'Time Series␣
 ↪Data')
plt.show()
```



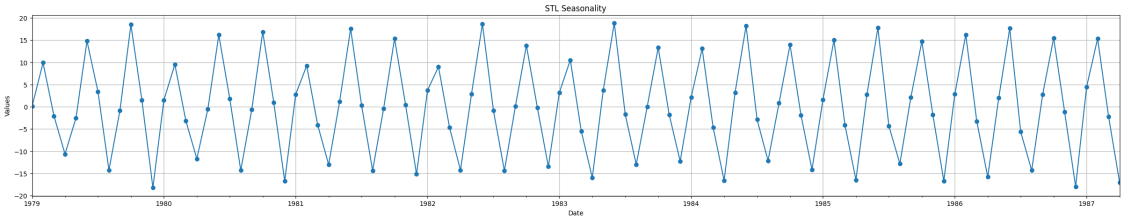
25.0.2 Check for trend and seasonality

[45]:
```
# STL Decomposition

from statsmodels.tsa.seasonal import STL

ts_stl = STL(ts["Value"], robust = True)  # Robust = better handling of outliers
decomp_stl = ts_stl.fit()
```

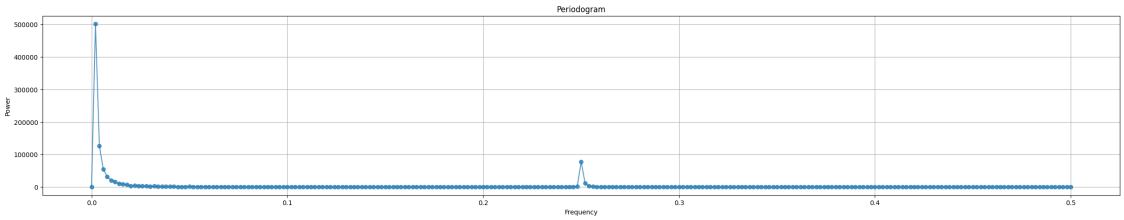
[46]:
```
# STL Trend

decomp_stl.trend.plot(figsize = (30,5), grid = True, ylabel = 'Values', title =␣
 ↪'STL Trend')
plt.show()

# It has a clear trend (upward)
```



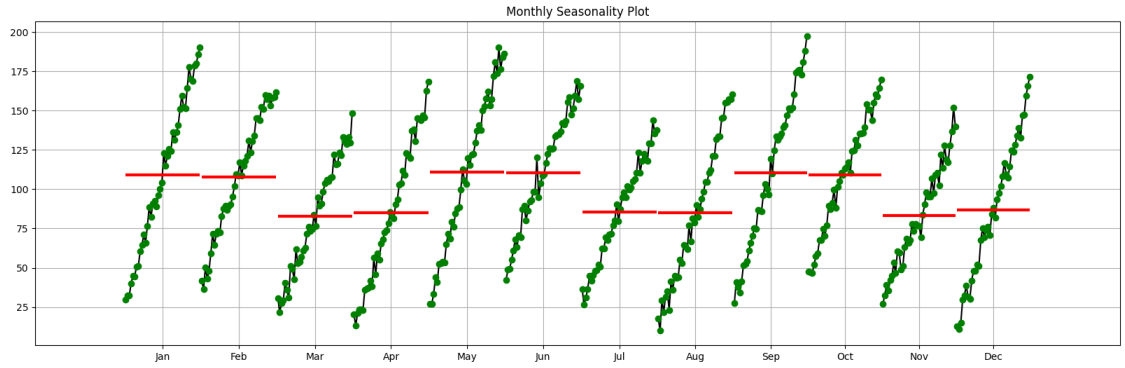
[47]:
```
# STL Seasonality

decomp_stl.seasonal[:100].plot(figsize = (30,5), grid = True, ylabel =␣
 ↪'Values', title = 'STL Seasonality', marker = 'o')
plt.show()

# There is a clear seasonality
```



[48]:
```
# Periodogram (for Seasonality)

from scipy.signal import periodogram

# Compute the periodogram
frequencies, power = periodogram(ts['Value'])

# Plot the periodogram
plt.figure(figsize=(30, 5))
plt.plot(frequencies, power, marker='o', linestyle='-', alpha = 0.75)
plt.title("Periodogram")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.grid(True)
plt.show()

# It exhibits seasonality at the start but does not sustain it on the later␣
 ↪part of the series
```

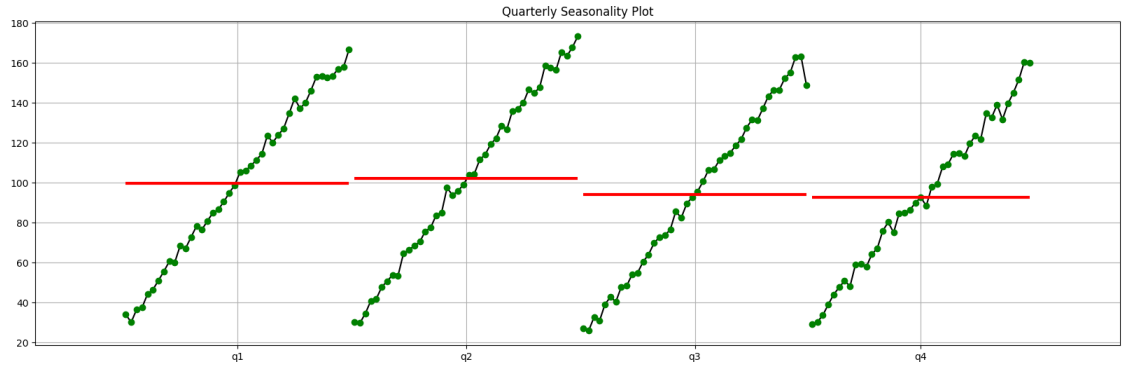


[49]:
```
# Month Plots

from statsmodels.graphics.tsaplots import month_plot

ts_monthly = ts['Value'].resample('MS').mean()

fig, ax = plt.subplots(figsize=(20, 6))
month_plot(ts_monthly, ax = ax)
plt.title("Monthly Seasonality Plot")
plt.grid()

for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent monthly disribution of values
# The red lines is the mean values for each month
```



[50]:
```
# Quarter Plots

from statsmodels.graphics.tsaplots import quarter_plot

ts_quarter = ts['Value'].resample('QE').mean()

fig, ax = plt.subplots(figsize=(20, 6))
quarter_plot(ts_quarter, ax = ax)
plt.title("Quarterly Seasonality Plot")
plt.grid()
```

```
for line in ax.get_lines():

  line.set_marker('o')
  line.set_markerfacecolor('green')
  line.set_markeredgecolor('green')

plt.show()

# The black lines represent quarterly disribution of values
# The red lines is the mean values for each quarter
```



25.0.3 Forecasting Using TES

1. Select a Model

[51]:
```
# We will use Double Exponential Smoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

2. Split data into train & test sets

[52]:
```
# Splitting

forecast_horizon = 12 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 months
test_data = ts.iloc[-forecast_horizon:] # select the last 12 months

train_data = train_data.asfreq('ME')
test_data = test_data.asfreq('ME')
```

3. Fit model on training set

[53]:
```
tes_model = ExponentialSmoothing(train_data['Value'], trend = 'additive',␣
 ↪seasonal = 'additive', seasonal_periods = 12).fit()
```

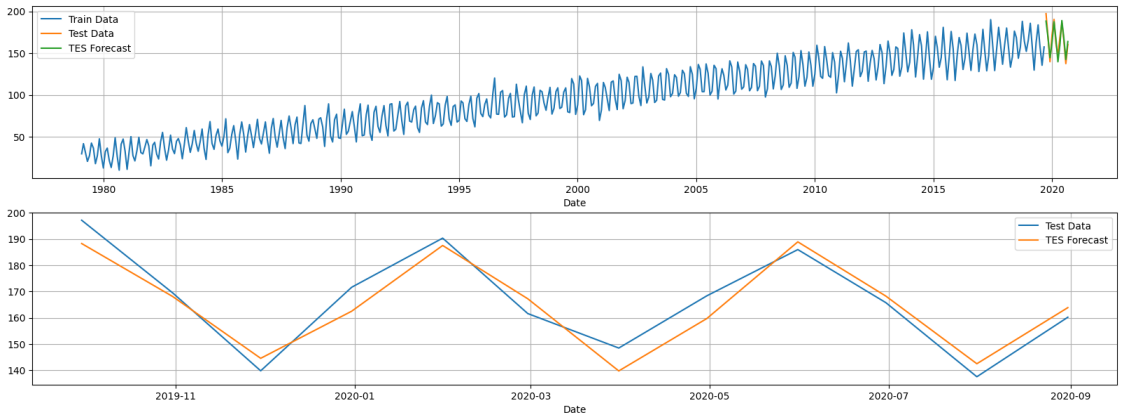
```
# tes_model_.summary()
```

4. Evaluate model on test set (visually first)

[54]:
```
# Forecast/Predict based on train_data

tes_pred = tes_model.forecast(forecast_horizon)
tes_pred
```

[54]:
```
2019-09-30    188.303352
2019-10-31    168.084297
2019-11-30    144.481540
2019-12-31    162.468450
2020-01-31    187.557764
2020-02-29    167.211702
2020-03-31    139.664775
2020-04-30    159.671178
2020-05-31    188.944837
2020-06-30    168.235834
2020-07-31    142.410558
2020-08-31    163.835983
Freq: ME, dtype: float64
```

[55]:
```
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data['Value'], label = 'Train Data')
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data['Value'], label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
plt.show()
```



5. Re-fit model on entire data set

[56]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('ME')

tes_model = ExponentialSmoothing(ts['Value'], trend = 'additive', seasonal =␣
 ↪'additive', seasonal_periods = 12).fit()
```

6. Forecast for future data

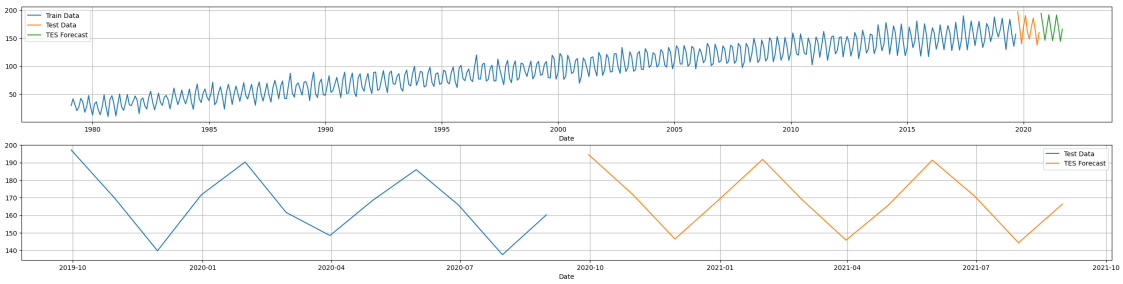
[57]:
```
# We forecast using the entire dataset

tes_pred = tes_model.forecast(forecast_horizon)
tes_pred
```

[57]:
```
2020-09-30    194.509289
2020-10-31    171.899299
2020-11-30    146.435579
2020-12-31    168.788566
2021-01-31    191.847231
2021-02-28    168.784800
2021-03-31    145.748526
2021-04-30    165.796796
2021-05-31    191.415254
2021-06-30    170.813018
2021-07-31    144.246656
2021-08-31    166.307719
Freq: ME, dtype: float64
```

[58]:
```
# Visualization

plt.figure(figsize = (30, 7))
```

```
plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[58]:
```
Text(0.5, 0, 'Date')
```



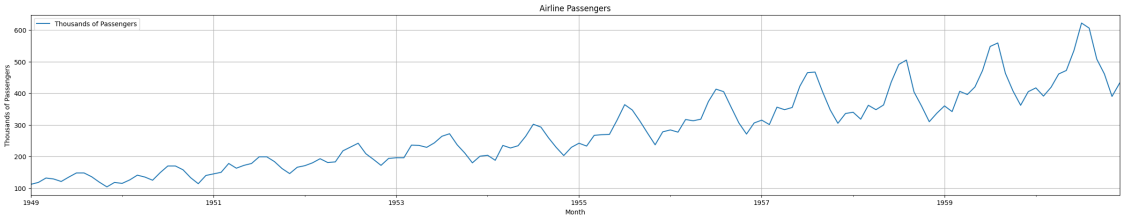
26 Triple Exponential Smoothing (TES)

[ ]:
```
# Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

26.0.1 Load datasets

[ ]:
```
# Load dataset

ts = pd.read_csv('https://raw.githubusercontent.com/renatomaaliw3/public_files/
 ↪master/Data%20Sets/airline_passengers.csv',
                     index_col = 'Month', parse_dates = True)

ts.plot(figsize = (30,5), grid = True, ylabel = 'Thousands of Passengers',␣
 ↪title = 'Airline Passengers')
```

```
plt.show()
```



26.0.2 Forecasting Using SES

1. Select a Model

[ ]:
```
# We will use Double Exponential Smoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

2. Split data into train & test sets

[ ]:
```
# Splitting

forecast_horizon = 36 # or periods

train_data = ts.iloc[:-forecast_horizon] # data except the last 12 months
test_data = ts.iloc[-forecast_horizon:] # select the last 12 months

train_data = train_data.asfreq('MS')
test_data = test_data.asfreq('MS')
```

3. Fit model on training set

[ ]:
```
tes_model = ExponentialSmoothing(train_data, trend = 'additive', seasonal =␣
 ↪'multiplicative', seasonal_periods = 12).fit()

# tes_model.summary()
```

4. Evaluate model on test set (visually first)

[ ]:
```
# Forecast/Predict based on train_data

tes_pred = tes_model.forecast(forecast_horizon)
tes_pred
```

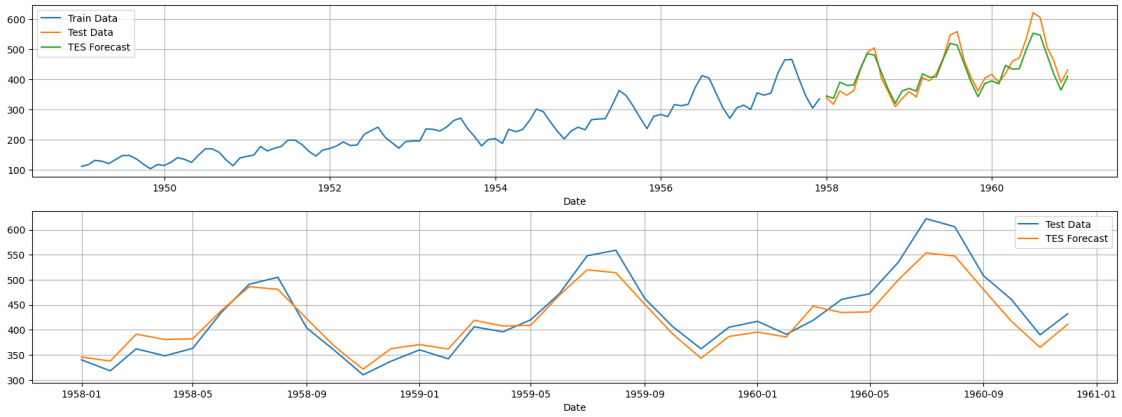
[ ]:
```
1958-01-01    345.593736
1958-02-01    337.453466
1958-03-01    391.270831
1958-04-01    380.630372
1958-05-01    382.065456
```

```
1958-06-01    438.507820
1958-07-01    486.134301
1958-08-01    480.818450
1958-09-01    422.394591
1958-10-01    367.220563
1958-11-01    321.233042
1958-12-01    362.194055
1959-01-01    370.491427
1959-02-01    361.619621
1959-03-01    419.124799
1959-04-01    407.567064
1959-05-01    408.945187
1959-06-01    469.178663
1959-07-01    519.939279
1959-08-01    514.061135
1959-09-01    451.430694
1959-10-01    392.320129
1959-11-01    343.065005
1959-12-01    386.671223
1960-01-01    395.389117
1960-02-01    385.785777
1960-03-01    446.978766
1960-04-01    434.503755
1960-05-01    435.824917
1960-06-01    499.849507
1960-07-01    553.744256
1960-08-01    547.303820
1960-09-01    480.466796
1960-10-01    417.419696
1960-11-01    364.896968
1960-12-01    411.148391
Freq: MS, dtype: float64
```

[ ]:
```python
# Visualization

plt.figure(figsize = (20, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
```

```
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[ ]:
```
Text(0.5, 0, 'Date')
```



5. Re-fit model on entire data set

[ ]:
```
# Notice we are using the entire dataset

ts = ts.asfreq('MS')

tes_model = ExponentialSmoothing(ts, trend = 'additive', seasonal =␣
 ↪'multiplicative', seasonal_periods = 12).fit()
```

6. Forecast for future data

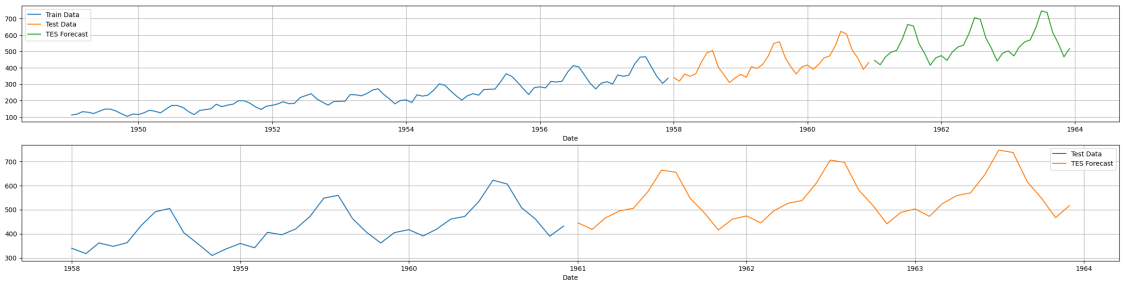
[ ]:
```
# We forecast using the entire dataset

tes_pred = tes_model.forecast(forecast_horizon)
tes_pred
```

[ ]:
```
1961-01-01    445.242361
1961-02-01    418.225341
1961-03-01    465.309832
1961-04-01    494.951245
1961-05-01    505.475873
1961-06-01    573.312666
1961-07-01    663.596376
1961-08-01    654.904046
1961-09-01    546.760994
1961-10-01    488.446831
1961-11-01    415.723528
```

```
1961-12-01    460.377815
1962-01-01    474.071074
1962-02-01    445.159419
1962-03-01    495.116229
1962-04-01    526.488038
1962-05-01    537.513154
1962-06-01    609.458560
1962-07-01    705.215761
1962-08-01    695.764708
1962-09-01    580.697963
1962-10-01    518.608285
1962-11-01    441.262917
1962-12-01    488.516423
1963-01-01    502.899788
1963-02-01    472.093496
1963-03-01    524.922626
1963-04-01    558.024832
1963-05-01    569.550435
1963-06-01    645.604455
1963-07-01    746.835146
1963-08-01    736.625370
1963-09-01    614.634933
1963-10-01    548.769739
1963-11-01    466.802307
1963-12-01    516.655031
Freq: MS, dtype: float64
```

[ ]:
```
# Visualization

plt.figure(figsize = (30, 7))

plt.subplot(2, 1, 1)
plt.plot(train_data.index, train_data, label = 'Train Data')
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')

plt.subplot(2, 1, 2)
plt.plot(test_data.index, test_data, label = 'Test Data')
plt.plot(tes_pred.index, tes_pred, label = 'TES Forecast')
plt.legend()
plt.grid()
plt.xlabel('Date')
```

[ ]:
```
Text(0.5, 0, 'Date')
```



[ ]:
```
# Why the forecast is good? It accounts for trend and seasonality
```