index

April 25, 2022

1 Introduction to Time Series

1.1 Introduction

From stock prices to climate data, time series data is found in a wide variety of domains, and being able to effectively work with such data is an increasingly important skill for data scientists.

In this lecture, you will be introduced to some common techniques used to import, clean, and manipulate time series data. Additionally, you'll learn how you can effectively visualize time series data in Python.

1.2 Objectives

You will be able to:

- Load time series data using Pandas and perform time series indexing
- Perform data cleaning operation on time series data
- Change the granularity of a time series
- Describe pandas' Timestamp and Datetime datatypes
- Explore the temporal structure of time series with line plots
- Construct and interpret time series histogram and density plots
- Create a time series heatmap

1.3 Loading Time Series Data

To get a sense of how to manipulate time series data, we'll walk through an example. The dataset we'll load contains daily minimum temperatures in Melbourne, Australia, from 1981-1990. The data is stored in a CSV file – so you can use Pandas to import the available data in `'min_temp.csv'`.

[1]:
```
import numpy as np
import pandas as pd
temp_data = pd.read_csv('min_temp.csv')
temp_data.head(15)
```

[1]:

	Date	Daily_min
0	1/1/81	20.7
1	2/1/81	17.9
2	3/1/81	18.8
3	4/1/81	14.6
4	5/1/81	15.8

```
5    6/1/81       15.8
6    7/1/81       15.8
7    8/1/81       17.4
8    9/1/81       21.8
9   10/1/81       20.0
10  11/1/81       16.2
11  12/1/81       13.3
12  13/1/81       16.7
13  14/1/81       21.5
14  15/1/81       25.0
```

Now, let's look at the information in our dataset:

[2]:
```
temp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3650 entries, 0 to 3649
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       3650 non-null   object
 1   Daily_min  3650 non-null   float64
dtypes: float64(1), object(1)
memory usage: 57.2+ KB
```

While working with time series data in Python, two things can make your life easier:

- Dates are in the index of the DataFrame (helps you with plotting)
- The columns are understood by Python as true "date" classes

To ensure dates are understood correctly, you can use Pandas' `Timestamp` or base Python's `Datetime` types; and they are interchangeable in most cases. It's the type used for the entries that make up a `DatetimeIndex`, and other time series oriented data structures in Pandas. Further details on Timestamp can be found here.

We need to do two things now:

1. Make sure that we change the dates in our dataset from "non-null object" to "non-null datetime" (i.e., change the data type of dates). This can be done using the `to_datetime()` function from Pandas. To make sure Python understands the date correctly, a `format` argument can be passed as specified in the documentation.
2. Ensure the date becomes the index.

[3]:
```
# Convert Date to a datetime column
temp_data['Date'] = pd.to_datetime(temp_data['Date'], format='%d/%m/%y')
```

[4]:
```
temp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3650 entries, 0 to 3649
Data columns (total 2 columns):
```

```
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       3650 non-null   datetime64[ns]
 1   Daily_min  3650 non-null   float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 57.2 KB
```

[5]:
```python
# Make Date the index
temp_data.set_index('Date', inplace=True)
```

[6]:
```python
temp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3650 entries, 1981-01-01 to 1990-12-31
Data columns (total 1 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Daily_min  3650 non-null   float64
dtypes: float64(1)
memory usage: 57.0 KB
```

[7]:
```python
temp_data.head(15)
```

[7]:
```
            Daily_min
Date
1981-01-01       20.7
1981-01-02       17.9
1981-01-03       18.8
1981-01-04       14.6
1981-01-05       15.8
1981-01-06       15.8
1981-01-07       15.8
1981-01-08       17.4
1981-01-09       21.8
1981-01-10       20.0
1981-01-11       16.2
1981-01-12       13.3
1981-01-13       16.7
1981-01-14       21.5
1981-01-15       25.0
```

1.4 Resampling

1.4.1 Downsampling

Did you notice that the date is now in the index!? Having the date as the index has several advantages, among others, easy visualization with dates on the x-axis, and the functionality to resample the data.

Pandas has a simple, powerful, and efficient functionality for performing resampling operations when converting the frequency conversion (e.g., converting monthly data into yearly data). This is very common in financial applications.

In the following cell, we are using the `.resample()` method to change the frequency of the time series to monthly (using the string `'MS'`) and then calculating the monthly mean.

[24]:
```
temp_monthly = temp_data.resample('MS')
month_mean = temp_monthly.mean()
```

[25]:
```
month_mean.head(15)
```

[25]:

	Daily_min
Date	
1981-01-01	17.712903
1981-02-01	17.678571
1981-03-01	13.500000
1981-04-01	12.356667
1981-05-01	9.490323
1981-06-01	7.306667
1981-07-01	7.577419
1981-08-01	7.238710
1981-09-01	10.143333
1981-10-01	10.087097
1981-11-01	11.890000
1981-12-01	13.680645
1982-01-01	16.567742
1982-02-01	15.921429
1982-03-01	14.935484

1.4.2 Upsampling

In some cases, it is useful to create upsampled time series as well, especially if you're trying to merge several time series with different frequencies. You can do this by using the `.resample()` method with an interval that is more frequent than the timestamp from the original time series.

In the following cell, we change the frequency of the data to 12 hours (using thr string `'12H'`) and then chain the `.asfreq()` method to return the resulting data.

[29]:
```
temp_bidaily = temp_data.resample('12H').asfreq()
temp_bidaily.head()
```

[29]:

	Daily_min
Date	
1981-01-01 00:00:00	20.7
1981-01-01 12:00:00	NaN
1981-01-02 00:00:00	17.9
1981-01-02 12:00:00	NaN
1981-01-03 00:00:00	18.8

As you can see, these new timestamps all have missing values in the resulting data. One of the common ways around this is to fill the current missing value with a previous valid one. To do this, you can use the `.ffill()` method as shown below:

[30]:
```
temp_bidaily_fill = temp_bidaily.ffill()
temp_bidaily_fill.head()
```

[30]:
```
                     Daily_min
Date
1981-01-01 00:00:00       20.7
1981-01-01 12:00:00       20.7
1981-01-02 00:00:00       17.9
1981-01-02 12:00:00       17.9
1981-01-03 00:00:00       18.8
```

1.5 Selecting and slicing time series data

Pandas carries the ability to handle date stamp indices allowing quick and handy way of slicing data. For example, we can slice our dataset to only retrieve data points that come after the year 1985:

[36]:
```
temp_1985_onwards = temp_data['1985':]

print(temp_1985_onwards.head())
print(temp_1985_onwards.tail())
```

```
            Daily_min
Date
1985-01-01       13.3
1985-01-02       15.2
1985-01-03       13.1
1985-01-04       12.7
1985-01-05       14.6
            Daily_min
Date
1990-12-27       14.0
1990-12-28       13.6
1990-12-29       13.5
1990-12-30       15.7
1990-12-31       13.0
```

1.6 Missing Data

It's pretty common for a time series dataset to have missing values as real-world data tends to be messy and imperfect. The simplest way to detect missing values is either plotting the data and identifying disjoint areas of time series, or by using a combination of `.isnull()` and `.sum()` methods:

[13]:
```
temp_data.isnull().sum()
```

[13]:
```
Daily_min    0
dtype: int64
```

In this case, there is no missing data. When data are missing, they can be handled in a multitude of ways: * Drop the data elements with missing values (this may result in low accuracy and loss of valuable information) * Fill in the missing values under a defined criteria * Use advanced machine learning methods to predict the missing values

In general, the `.fillna()` method can be used along with methods like `.bfill()` of `.ffill()` as an argument/criterion for filling in missing values . `.bfill()` (backward filling) looks for the next valid entry in the time series and fills the gaps with this value. Similarly, `.ffill()` can be used to copy forward the previous valid entry of the time series (as demonstrated above).

1.7 Visualizing time series data

Visualizations play an important role in time series analysis. Time series data naturally lends itself to visualization techniques for identifying rises, falls, trends, and noise, etc. Plotting raw time series allows data diagnostics to identify certain trends or events.

In what follows, we'll use a dataset downloaded from Qlik DataMarket. The dataset contains information on the average monthly returns of the NYSE between 1961 and 1966.

[37]:
```python
nyse = pd.read_csv('NYSE_monthly.csv')
nyse['Month'] = pd.to_datetime(nyse['Month'])
nyse.set_index('Month', inplace=True)

nyse.head()
```

[37]:

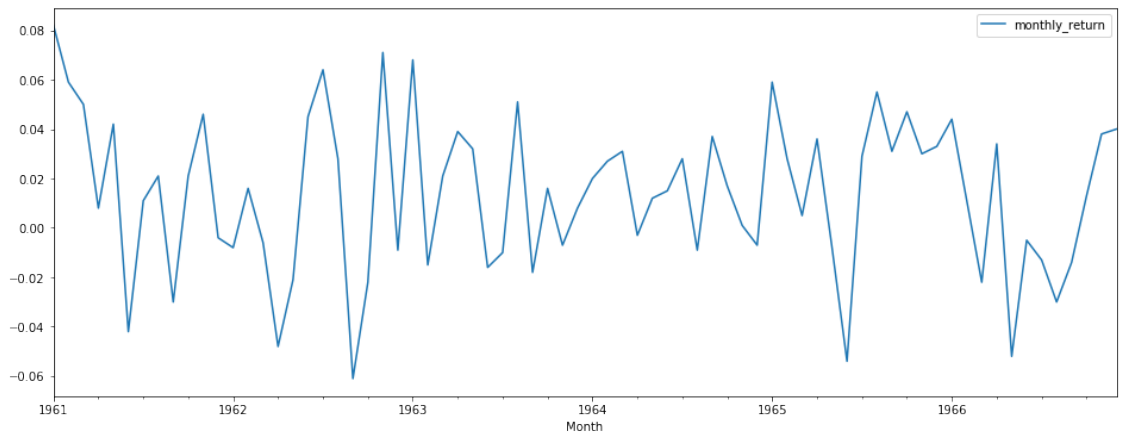
	monthly_return
Month	
1961-01-01	0.082
1961-02-01	0.059
1961-03-01	0.050
1961-04-01	0.008
1961-05-01	0.042

1.8 Time series line plot

Line plots are the most common technique for visualizing time series data as they can clearly show changes over time. Using the convention, time is shown on the x-axis with the observation values along the y-axis.

Let's use the simple `.plot()` method to draw the line graph for the `nyse` series.

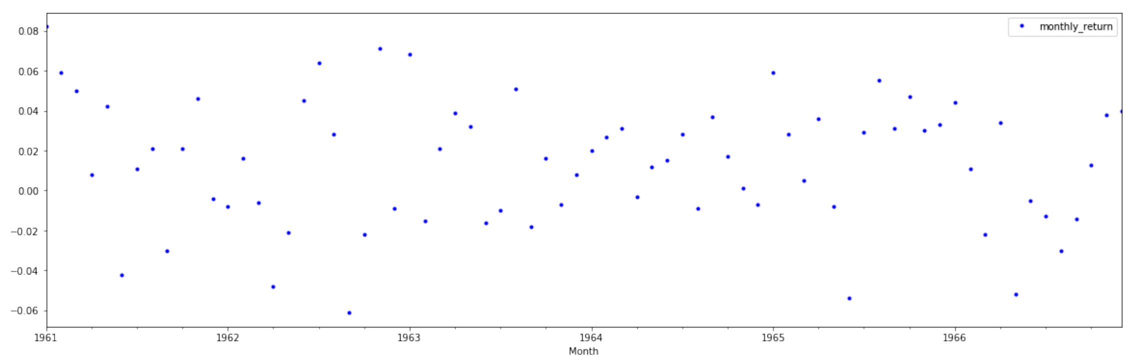
[39]:
```python
# Draw a line plot using nyse and .plot() method
import matplotlib.pyplot as plt
%matplotlib inline

nyse.plot(figsize = (16,6));
```



1.9 Time series dot plot

For some time series, you may want to change the style of a line plot for a more refined visualization with a higher resolution of events. These time series are not very dense so it might be useful to change from a continuous line to dots because this representation might be misleading. You can change the continuous line to dots, each representing one entry in the time series. This can be achieved by changing the `style` parameter of the line plot. Let's pass `style='.b'` as an argument to `.plot()` method.

[44]:
```
# Draw a dot plot using temp and .plot() method
nyse.plot(figsize = (20,6), style = '.b');
```



It's not a surprise to see that the general pattern looks very much similar to the line plot, however, we can identify some outliers that represent very low and very high return months. Dot plots can prove to be very helpful in identifying outliers and very small patterns which may not be so obvious otherwise.

In the dataset, the NYSE returns span 6 years. We can group data by year and create a line plot for each year for direct comparison.

1.10 Grouping and Visualizing Time Series Data

Now, we'll look at how a time series can be regrouped for a given time interval, i.e. weekly/monthly/yearly average values and compare them to identify any changes taking place over time. We'll use the Pandas' `grouper()` function in conjunction with the `.groupby()` method to achieve this.

The list of aliases for time series frequencies can be found here.

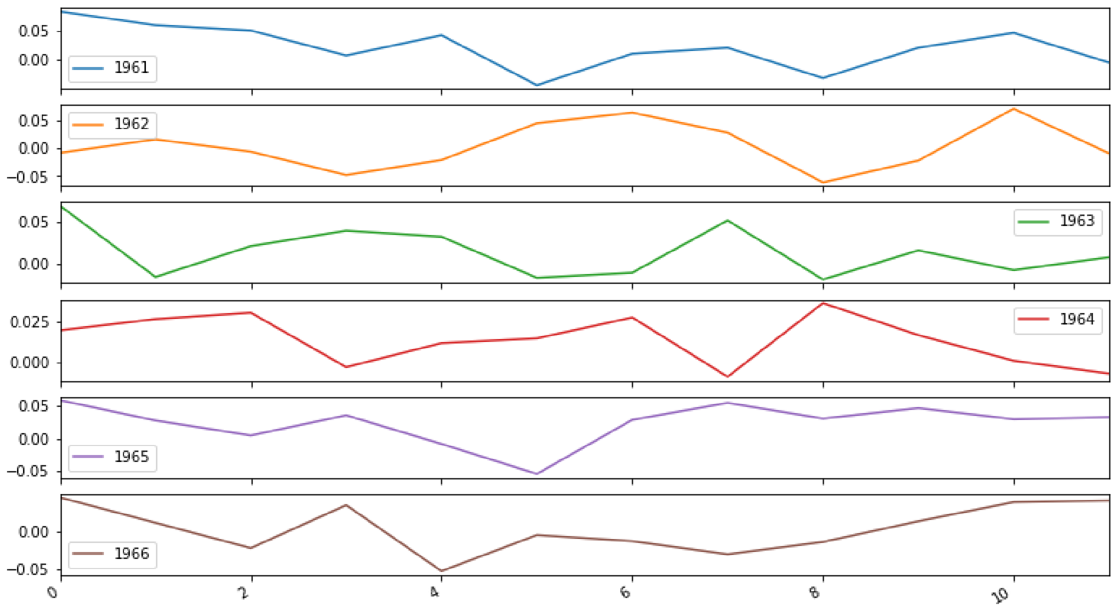
[74]:
```
# Use pandas grouper to group values using annual frequency
year_groups = nyse.groupby(pd.Grouper(freq ='A'))
pd.DataFrame(year_groups)[1].values.ravel()
```

[74]:
```
array([            monthly_return
Month
1961-01-01          0.082
1961-02-01          0.059
1961-03-01          0.050
1961-04-01          0.008
1961-05-01          0.042
1961-06-01         -0.042
1961-07-01          0.011
1961-08-01          0.021
1961-09-01         -0.030
1961-10-01          0.021
1961-11-01          0.046
1961-12-01         -0.004,
                   monthly_return
Month
1962-01-01         -0.008
1962-02-01          0.016
1962-03-01         -0.006
1962-04-01         -0.048
1962-05-01         -0.021
1962-06-01          0.045
1962-07-01          0.064
1962-08-01          0.028
1962-09-01         -0.061
1962-10-01         -0.022
1962-11-01          0.071
1962-12-01         -0.009,
                   monthly_return
Month
1963-01-01          0.068
1963-02-01         -0.015
1963-03-01          0.021
1963-04-01          0.039
1963-05-01          0.032
```

```
1963-06-01          -0.016
1963-07-01          -0.010
1963-08-01           0.051
1963-09-01          -0.018
1963-10-01           0.016
1963-11-01          -0.007
1963-12-01           0.008,
                   monthly_return
Month
1964-01-01           0.020
1964-02-01           0.027
1964-03-01           0.031
1964-04-01          -0.003
1964-05-01           0.012
1964-06-01           0.015
1964-07-01           0.028
1964-08-01          -0.009
1964-09-01           0.037
1964-10-01           0.017
1964-11-01           0.001
1964-12-01          -0.007,
                   monthly_return
Month
1965-01-01           0.059
1965-02-01           0.028
1965-03-01           0.005
1965-04-01           0.036
1965-05-01          -0.008
1965-06-01          -0.054
1965-07-01           0.029
1965-08-01           0.055
1965-09-01           0.031
1965-10-01           0.047
1965-11-01           0.030
1965-12-01           0.033,
                   monthly_return
Month
1966-01-01           0.044
1966-02-01           0.011
1966-03-01          -0.022
1966-04-01           0.034
1966-05-01          -0.052
1966-06-01          -0.005
1966-07-01          -0.013
1966-08-01          -0.030
1966-09-01          -0.014
1966-10-01           0.013
```

```
1966-11-01          0.038
1966-12-01          0.040], dtype=object)
```

[81]:
```python
# Create a new DataFrame and store yearly values in columns
nyse_annual = pd.DataFrame()

for yr, group in year_groups:

    nyse_annual[yr.year] = group.values.ravel()

# Plot the yearly groups as subplots
nyse_annual.plot(figsize = (13,8), subplots=True, legend=True);
```



[82]:
```python
nyse_annual
```

[82]:

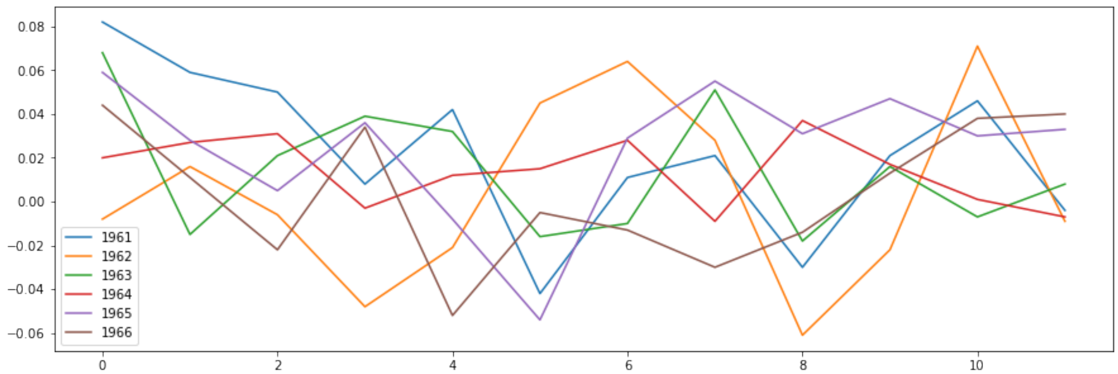
	1961	1962	1963	1964	1965	1966
0	0.082	-0.008	0.068	0.020	0.059	0.044
1	0.059	0.016	-0.015	0.027	0.028	0.011
2	0.050	-0.006	0.021	0.031	0.005	-0.022
3	0.008	-0.048	0.039	-0.003	0.036	0.034
4	0.042	-0.021	0.032	0.012	-0.008	-0.052
5	-0.042	0.045	-0.016	0.015	-0.054	-0.005
6	0.011	0.064	-0.010	0.028	0.029	-0.013
7	0.021	0.028	0.051	-0.009	0.055	-0.030
8	-0.030	-0.061	-0.018	0.037	0.031	-0.014
9	0.021	-0.022	0.016	0.017	0.047	0.013
10	0.046	0.071	-0.007	0.001	0.030	0.038

```
11 -0.004 -0.009  0.008 -0.007  0.033  0.040
```

We can see 6 subplots, one for each year. Each plot is 12 months in length following the annual frequency.

In some cases, it may also be a good idea to plot these groups in an overlapping manner for a direct comparison. Disabling subplots can help us achieve this.

[47]:
```
# Plot overlapping yearly groups
nyse_annual.plot(figsize = (15,5), subplots=False, legend=True);
```



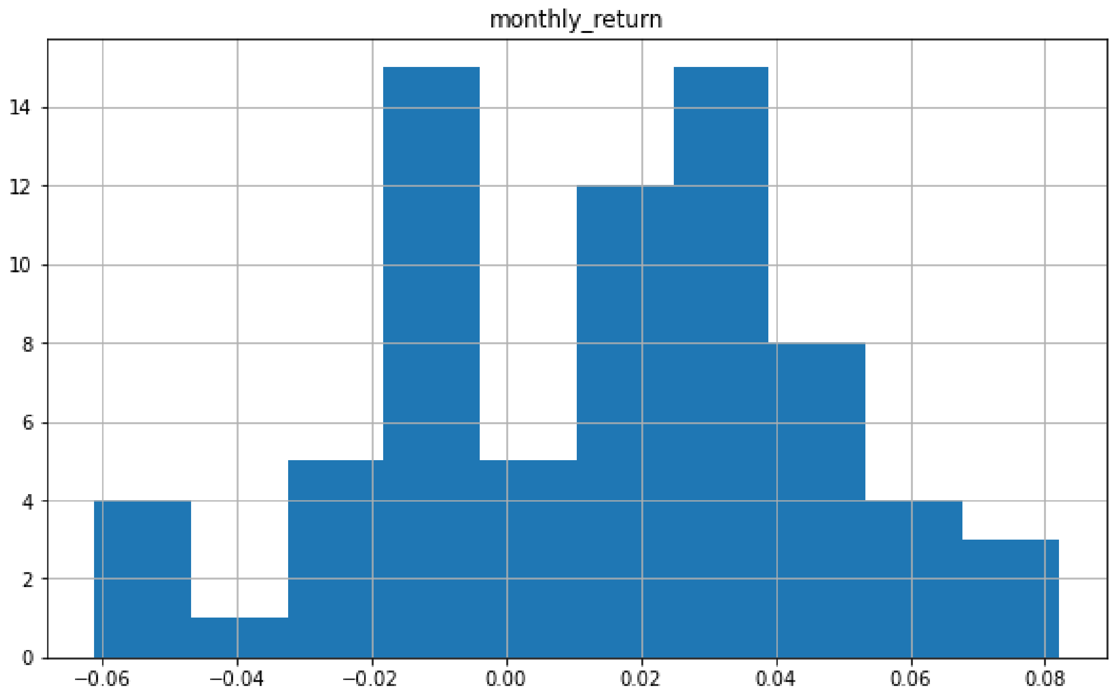
1.11 Time Series Histogram and Density Plots

Most linear time series forecasting methods assume a well-behaved distribution of observations e.g. a normal distribution. This can be explicitly checked using tools like statistical hypothesis tests we saw during hypothesis testing exercises. Visualizing these distributions can provide a useful first check of the distribution of observations both on raw observations and after any type of data transformation has been performed.

We will now create a histogram plot of the observations in the dataset using the `.hist()` method.

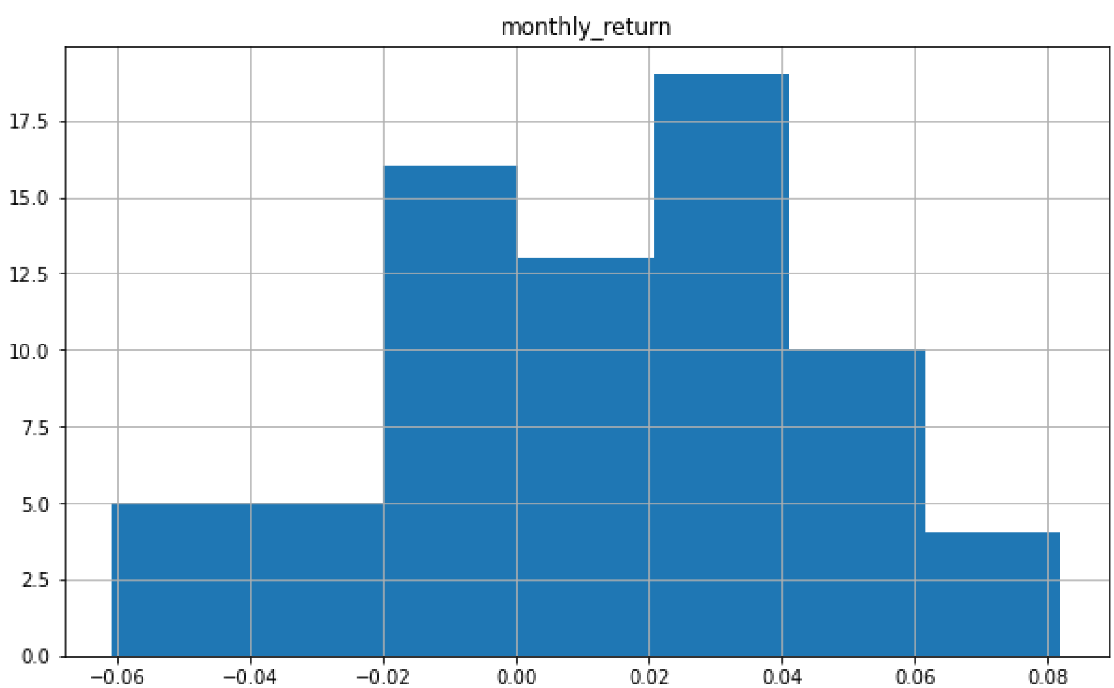
> A histogram groups values into bins, and the frequency or count of observations in each bin can provide insight into the underlying distribution of the observations.

[53]:
```
nyse.hist(figsize = (10,6));
```



The plot shows a distribution that doesn't exactly look Gaussian/Normal. The plotting function automatically selected the size of the bins based on the spread of values in the data here. Let's see what happens if we set the number of bins equal to 7.

[54]:
```
nyse.hist(figsize = (10,6), bins = 7);
```

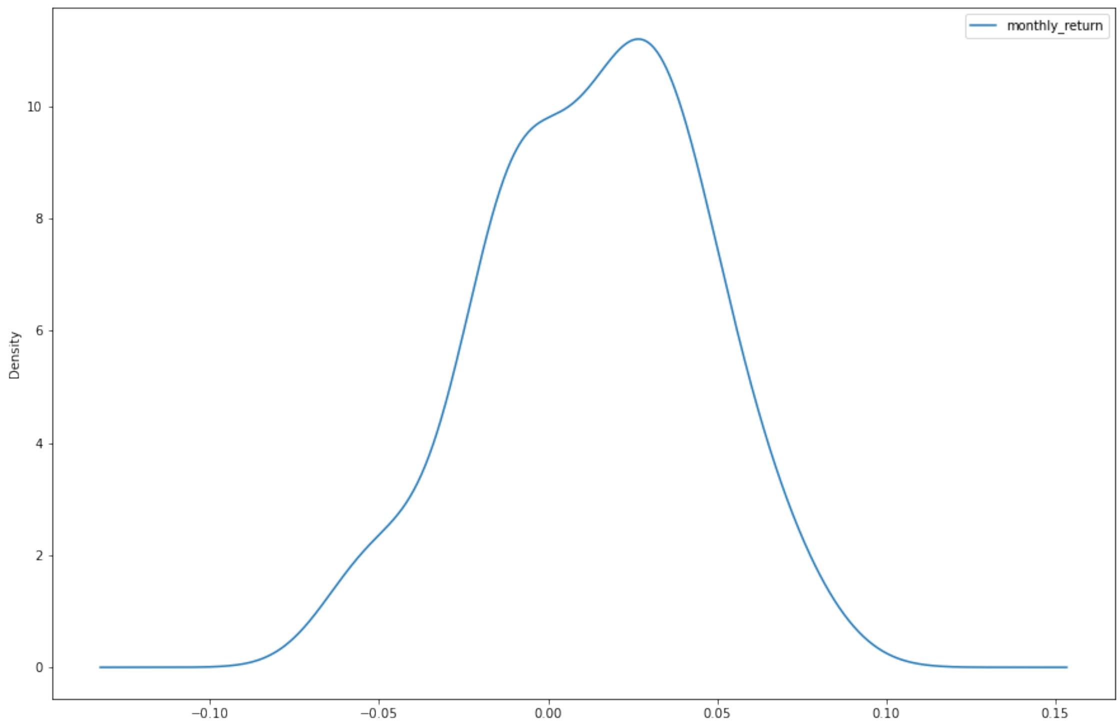


This already looks more normal. With stock exchange returns, it is to be expected that on average, the returns will be 0 and have a Gaussian distribution around that. With only 6 years of monthly data, it is to be expected that the distribution does not exactly look Gaussian.

We can also get a better idea of the shape of the distribution of observations by using a density plot which is like the histogram, except a function is used to fit the distribution of observations with smoothing to summarize this distribution.

Let's plot a density plot of the NYSE stock exchange data. We will achieve this by setting the `kind` parameter of the `plot()` method to `'kde'`, which stands for Kernel Density Estimation.

[55]:
```
# Plot a density plot for nyse dataset
nyse.plot(kind='kde', figsize = (15,10));
```



We can see that the density plot provides a clearer summary of the distribution of observations. We can see that perhaps the distribution is more Gaussian than we were able to see in the histogram.

Seeing a distribution like this may suggest later exploring statistical hypothesis tests to formally check if the distribution is Gaussian and perhaps data preparation techniques to reshape the distribution.

1.12 Time series box and whisker plots by year

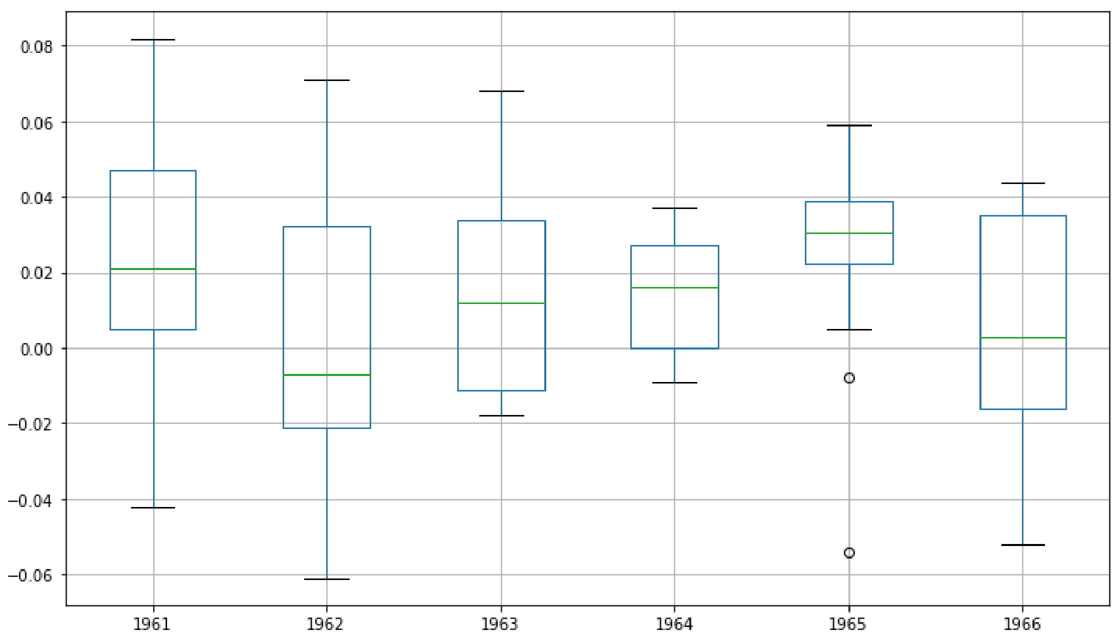
Histograms and density plots provide insight into the distribution of all observations, but we may be interested in the distribution of values by **time interval**.

Another type of plot that is useful to summarize the distribution of observations is the box and whisker plot. This plot draws a box around the 25th and 75th percentiles of the data that captures the middle 50% of observations. A line is drawn at the 50th percentile (the median) and whiskers are drawn above and below the box to summarize the general extent of the observations. Dots are drawn for outliers outside the whiskers or extent of the data.

Box and whisker plots can be created and compared for each interval in a time series, such as years, months, or days.

Let's use our groups by years DataFrame to plot a box and whisker plot for each year, side-by-side, for direct comparison using the `.boxplot()` method.

[23]:
```
# Generate a box and whiskers plot for nyse_annual
nyse_annual.boxplot(figsize = (12,7));
```



Comparing box and whisker plots by consistent intervals is a useful tool. Within an interval, it can help to spot outliers (dots above or below the whiskers).

Across intervals, we can look for multiple year trends, seasonality, and other structural information that could be modeled. Seasonality is generally not a thing in financial data, but in the lab that follows you'll explore visualizing seasonal temperature data!

1.13 Time series heat maps

A matrix of numbers can be plotted as a surface, where the values in each cell of the matrix are assigned a unique color. This is called a heat map, as larger values can be drawn with warmer colors (yellows and reds) and smaller values can be drawn with cooler colors (blues and greens). Like the box and whisker plots, we can compare observations between intervals using a heat map.

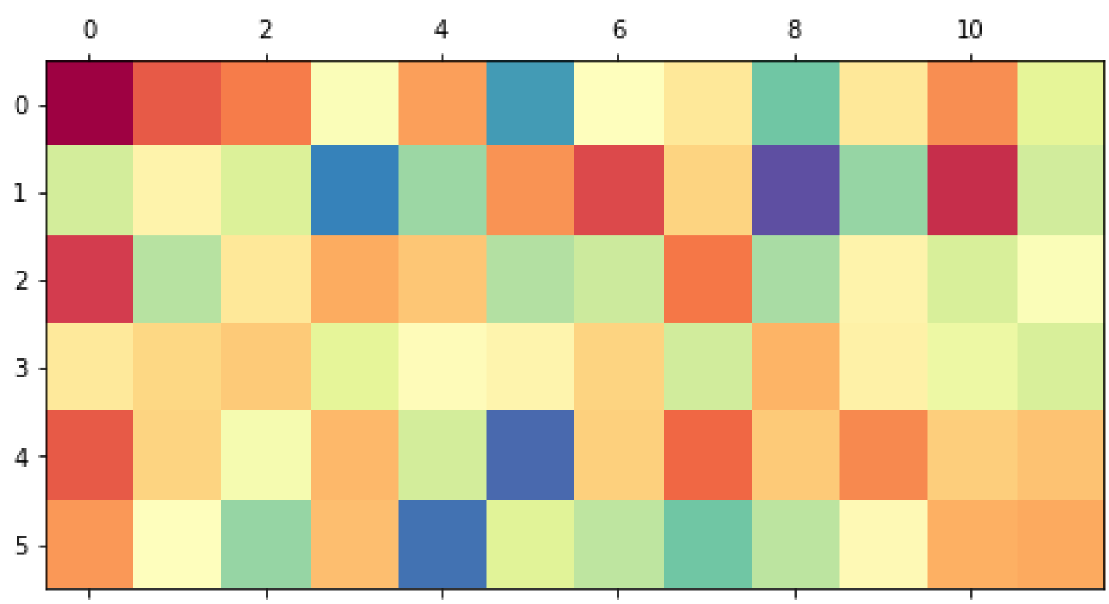
In the case of our NYSE dataset, the observations can be arranged into a matrix of year-columns and month-rows, with monthly returns in the cell for each day. A heat map of this matrix can then be plotted.

We'll now create a heatmap of the minimum daily temperatures data. The `matshow()` function from the matplotlib library is used as no heatmap support is provided directly in Pandas. In the following cell, we will:

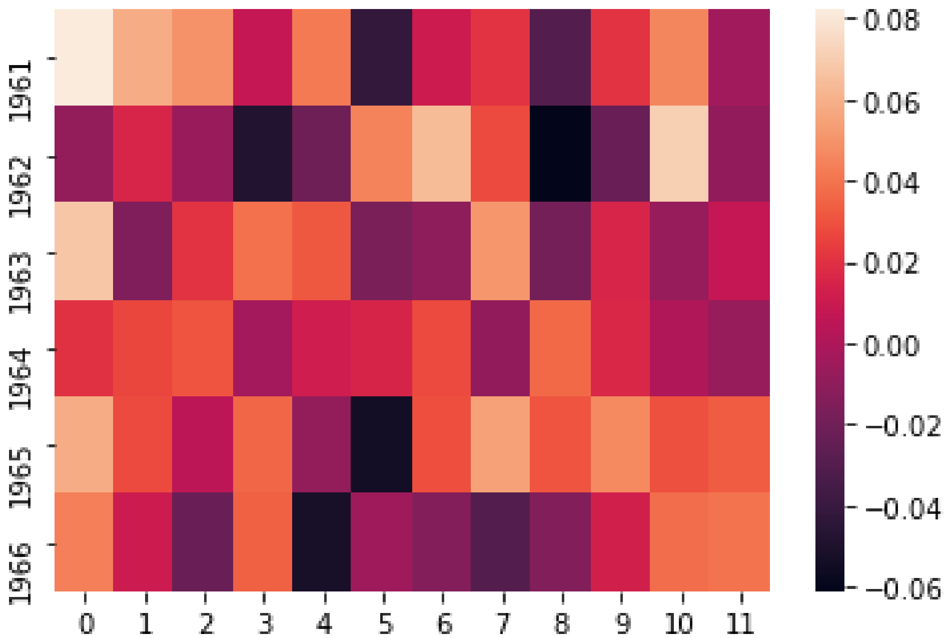
1. Rotate (transpose) the `nyse_annual` DataFrame as a new matrix so that each row represents one year and each column one day. This provides a more intuitive, left-to-right layout of the data.

2. Use `matshow()` function to draw a heatmap for the transposed yearly matrix.

[89]:
```
year_matrix = nyse_annual.T
plt.matshow(year_matrix, interpolation=None, aspect='auto', cmap=plt.cm.
    ↪Spectral_r);
```



[90]:
```
import seaborn as sns
sns.heatmap(year_matrix);
```



Orange/red colors represent higher values, blue represents low values, green represents values in the middle. Heat maps will make more sense after you've gone through the lab which will follow later and examine the Australian temperature data.

1.14 Additional reading

- An overview of Pandas time series functionality can be found here.
- Some more manipulation tricks can be found here.

1.15 Summary

In this introductory lesson, we learned how to import and manipulate time series data in Python using Pandas. We learned how to fulfill all the requirements for a dataset to be classified as a time series by ensuring timestamp values as data index. Basic data handling techniques for getting time series data ready for further analysis were introduced. We also learned how to explore the temporal relationships with line and dot plots. We also explored the distribution of observations with histograms and density plots and change in distribution of observations with box and whisker and heat map plots.