

# **From Introduction to Data Visualization in Python**

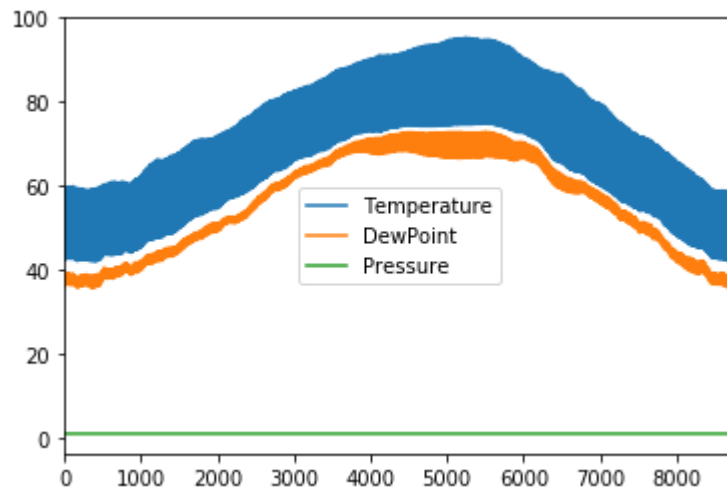
## **Plotting time series, datetime indexing**

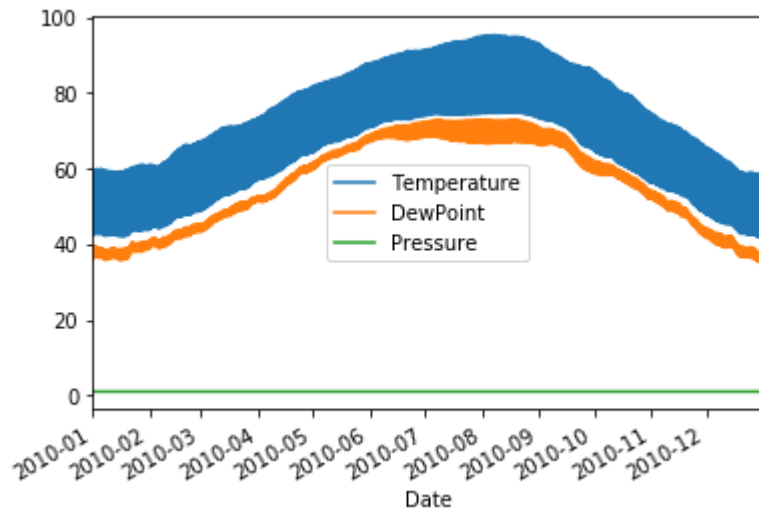
Pandas handles datetimes not only in your data, but also in your plotting.

```
In [2]: import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv('weather_data_austin_2010.csv')
# Plot the raw data before setting the datetime index
df.plot()
plt.show()

df.Date = pd.to_datetime(df.Date)
df.set_index('Date', inplace=True)

df.plot()
plt.show()
```



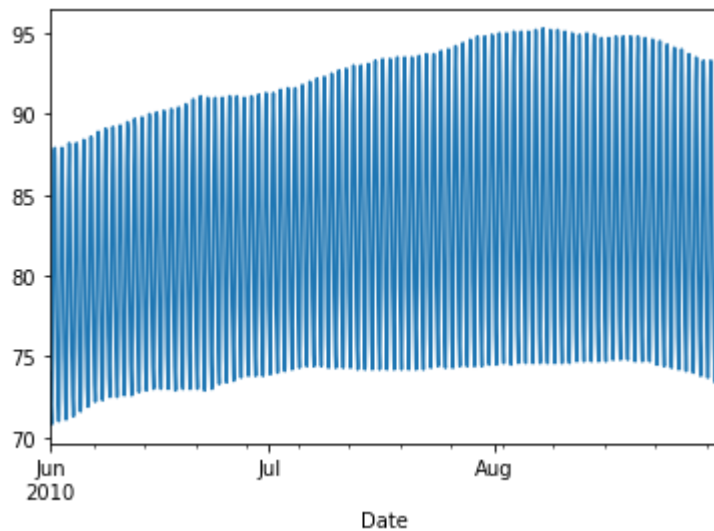


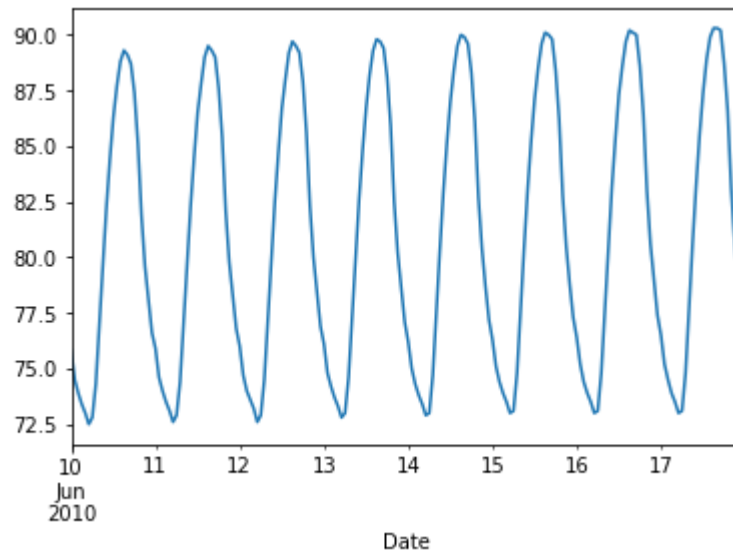
**Plotting date ranges, partial indexing**

```
In [3]: import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv('weather_data_austin_2010.csv', index_col='Date', parse_dates=True )

#df.head()
# Plot the summer data
df.Temperature['2010-Jun':'2010-Aug'].plot()
#unsmoothed = df.loc['2010-Aug-01':'2010-Aug-15', 'Temperature']
plt.show()
plt.clf()

# Plot the one week data
df.Temperature['2010-06-10':'2010-06-17'].plot()
plt.show()
plt.clf()
```





<Figure size 432x288 with 0 Axes>

**Plotting multiple stock prices on same graph**

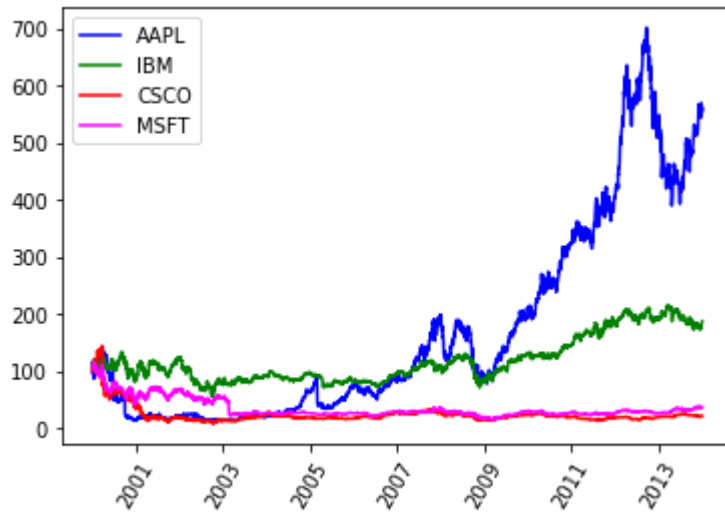
```
In [4]: # A time-series plot with timedate data type as index.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df_stocks = pd.read_csv("stocks.csv", index_col = 0)
aapl = df_stocks['AAPL']
ibm = df_stocks['IBM']
cscs = df_stocks['CSCO']
msft = df_stocks['MSFT']

aapl.index = pd.to_datetime(aapl.index)
ibm.index = pd.to_datetime(ibm.index)
cscs.index = pd.to_datetime(cscs.index)
msft.index = pd.to_datetime(msft.index)

*****In plt.plot(), only Y-coordinates are provided. Therefore, X coordinates will use default: the dataframe index
*****In the index is not read-in as datetime type, then it will give very congested x-axis.

plt.plot(aapl, color = 'blue', label = 'AAPL')
plt.plot(ibm, color = 'green', label = 'IBM')
plt.plot(cscs, color = 'red', label = 'CSCO')
plt.plot(msft, color = 'magenta', label = 'MSFT')
plt.legend(loc='upper left')
plt.xticks(rotation=60)
plt.show()
```



# From Visualizing Time Series Data in Python

## Introduction

Load your time series data

```
In [4]: # Import pandas
import pandas as pd

discoveries = pd.read_csv('ch1_discoveries.csv')

# Display the first five lines of the DataFrame
discoveries.head()
discoveries.info()
discoveries.describe()
print(discoveries.dtypes) #Check data types of each column. Alternative way is using info().
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 2 columns):
date      100 non-null object
Y         100 non-null int64
dtypes: int64(1), object(1)
memory usage: 1.6+ KB
date      object
Y         int64
dtype: object
```

**Test whether your data is of the correct type**



```
In [5]: # Convert the date column to a datestamp type
discoveries['date'] = pd.to_datetime(discoveries['date'])
print(discoveries.dtypes)
print(discoveries.info())
discoveries.head()
```

```
date    datetime64[ns]
Y        int64
dtype: object
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 2 columns):
date    100 non-null datetime64[ns]
Y        100 non-null int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 1.6 KB
None
```

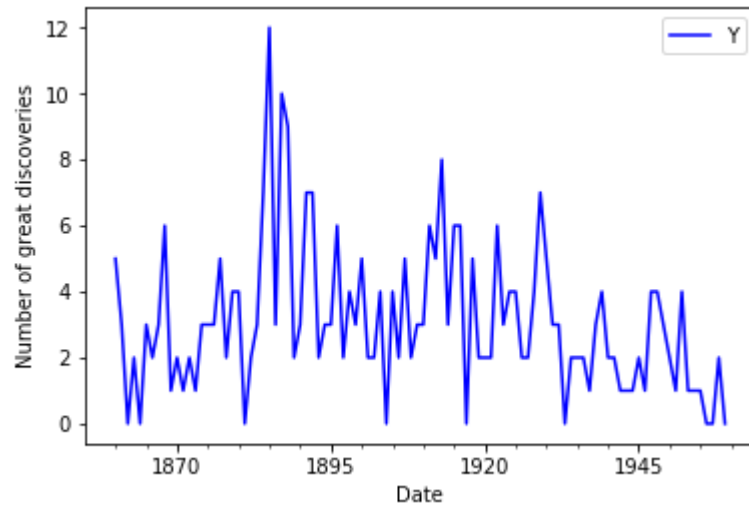
```
Out[5]:
```

	date	Y
0	1860-01-01	5
1	1861-01-01	3
2	1862-01-01	0
3	1863-01-01	2
4	1864-01-01	0

## Your first plot!

matplotlib is the most widely used plotting library in Python, and would be the most appropriate tool for this job. Fortunately for us, the pandas library has implemented a `.plot()` method on Series and DataFrame objects that is a wrapper around `matplotlib.pyplot.plot()`, which makes it easier to produce plots.

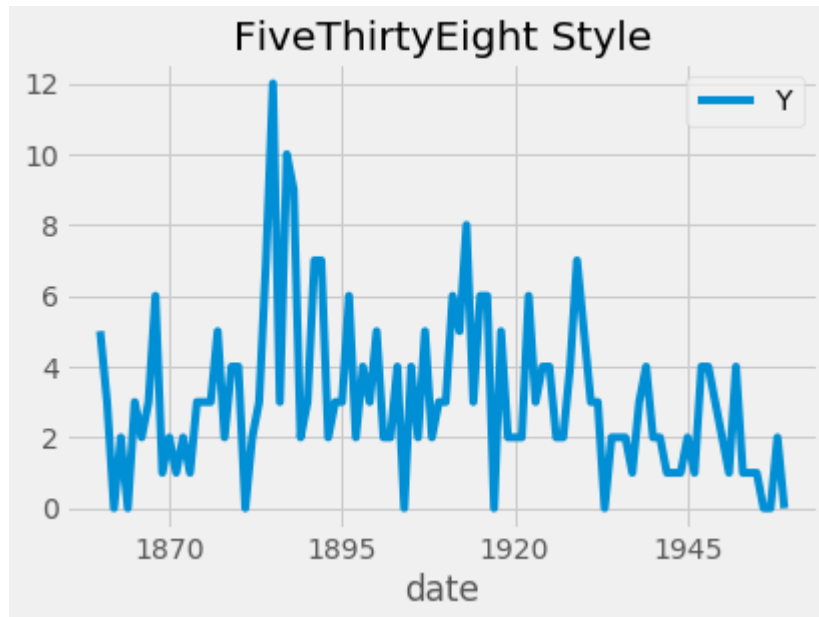
```
In [6]: import matplotlib.pyplot as plt
discoveries = discoveries.set_index('date')
ax = discoveries.plot(color='blue')
ax.set_xlabel('Date')
ax.set_ylabel('Number of great discoveries')
plt.show()
```



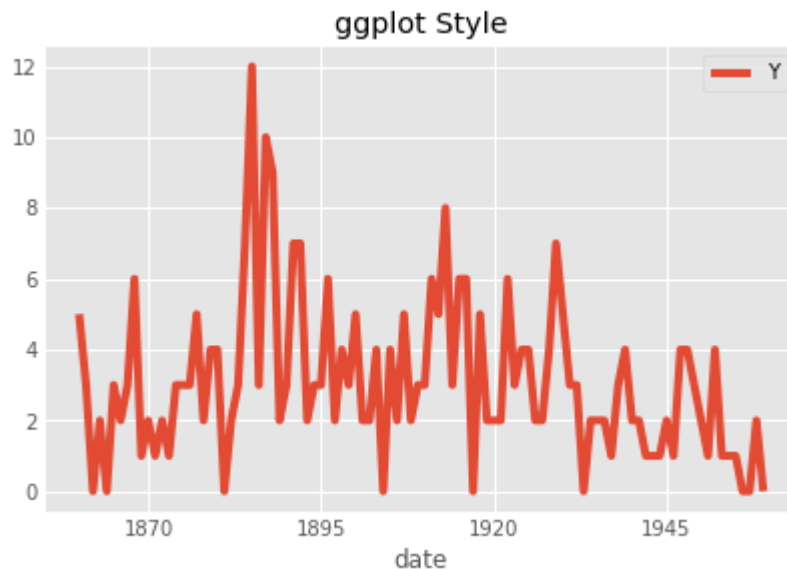
## Specify plot styles

See other types in the general python visualization write up.

```
In [8]: import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
ax1 = discoveries.plot()
ax1.set_title('FiveThirtyEight Style')
plt.show()
```



```
In [9]: import matplotlib.pyplot as plt
plt.style.use('ggplot')
ax2 = discoveries.plot()
ax2.set_title('ggplot Style')
plt.show()
```



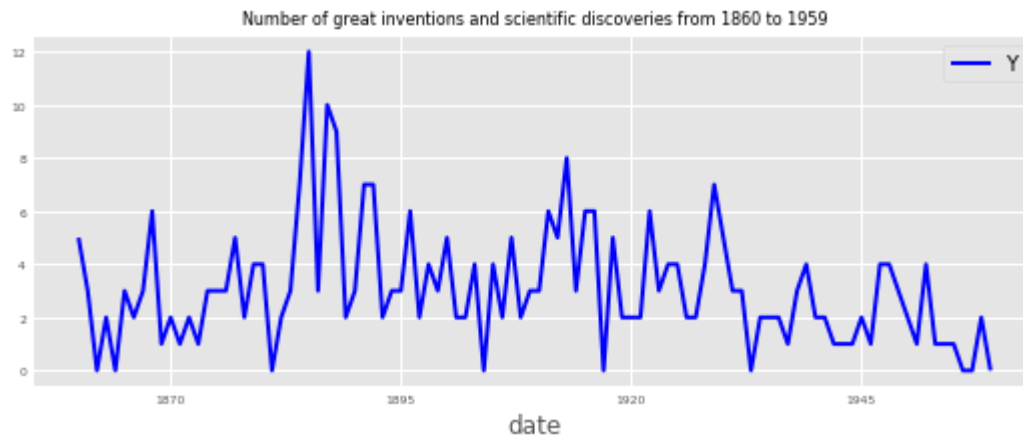
```
In [10]: print(plt.style.available)
```

```
['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark-palette', 'seaborn-dark', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid', 'seaborn', 'Solarize_Light2', '_classic_test']
```

## Display and label plots

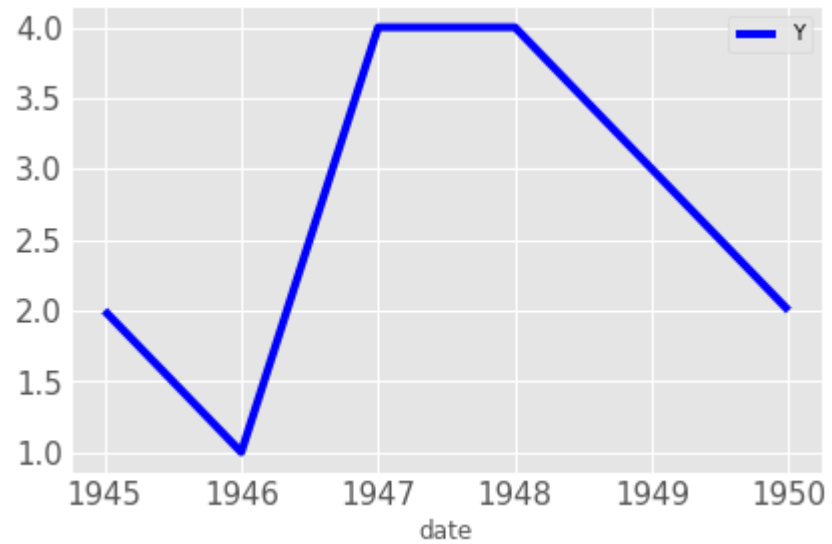
Here using plot() built on data frame object is nice to specify size directly in site.

```
In [11]: ax = discoveries.plot(color='blue', figsize=(8, 3), linewidth=2, fontsize=6)
ax.set_title('Number of great inventions and scientific discoveries from 1860 to 1959', fontsize=8)
plt.show()
```

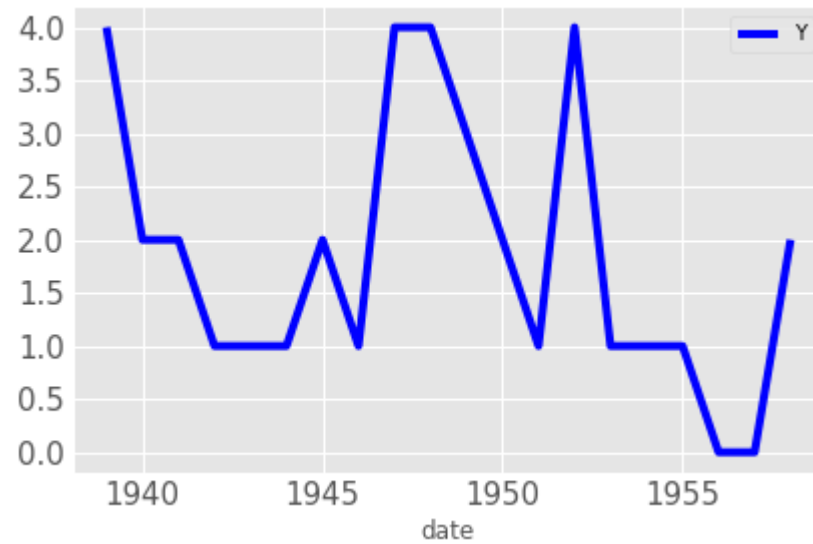


**Subset time series data**

```
In [12]: discoveries_subset_1 = discoveries['1945-01':'1950-01']  
ax = discoveries_subset_1.plot(color='blue', fontsize=15)  
plt.show()
```

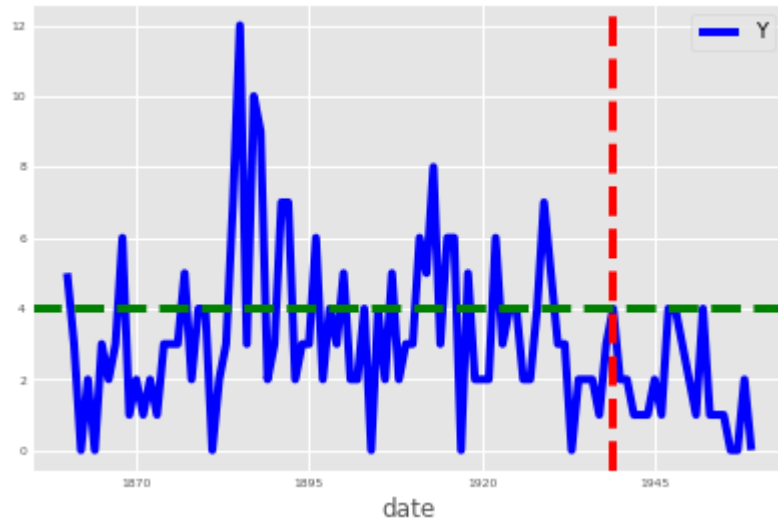


```
In [13]: discoveries_subset_2 = discoveries['1939-01':'1958-01']  
ax = discoveries_subset_2.plot(color='blue', fontsize=15)  
plt.show()
```



**Add vertical and horizontal markers**

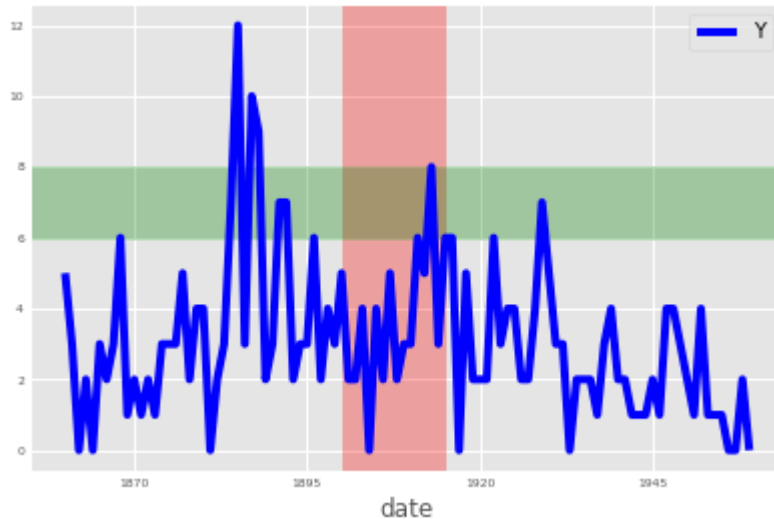
```
In [14]: ax = discoveries.plot(color='blue', fontsize=6)
ax.axvline('1939-01-01', color='red', linestyle='--')
ax.axhline(4, color='green', linestyle='--')
plt.show()
```



**Add shaded regions to your plot**



```
In [16]: ax = discoveries.plot(color='blue', fontsize=6)
ax.axvspan('1900-01-01', '1915-01-01', color='red', alpha=0.3)
ax.axhspan(6, 8, color='green', alpha=0.3)
plt.show()
```



## Summary Statistics and Diagnostics

In this chapter, you will gain a deeper understanding of your time series data by computing summary statistics and plotting aggregated views of your data.

### Find missing values

```
In [21]: import pandas as pd
co2_levels = pd.read_csv('ch2_co2_levels.csv')
co2_levels = co2_levels.set_index('datestamp')
print(co2_levels.isnull().sum())
```

```
co2      59
dtype: int64
```

### Handle missing values

In order to replace missing values in your time series data, you can use the command:

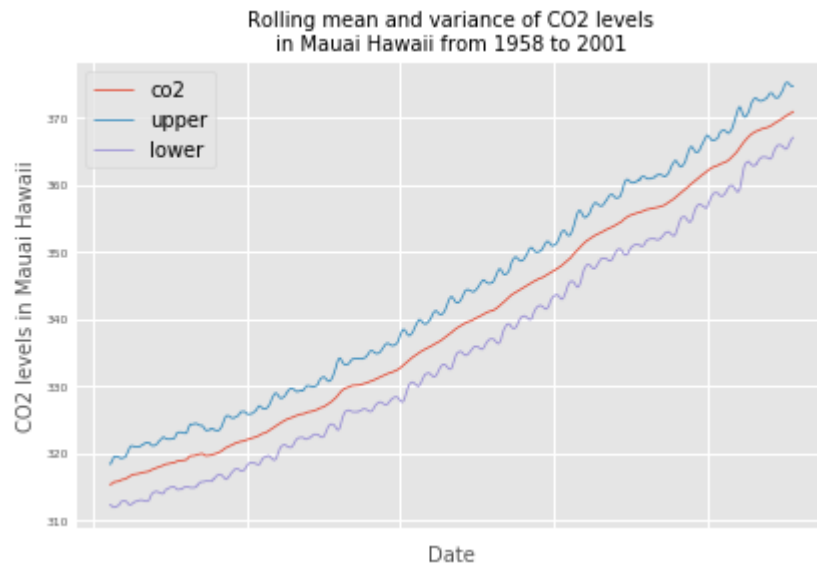
`df = df.fillna(method="ffill")` where the argument specifies the type of method you want to use. For example, specifying `bfill` (i.e backfilling) will ensure that missing values are replaced using the next valid observation, while `ffill` (i.e. forward-filling) ensures that missing values are replaced using the last valid observation.

```
In [22]: co2_levels = co2_levels.fillna(method='bfill')  
         print(co2_levels.isnull().sum())
```

```
co2      0  
dtype: int64
```

## Display rolling averages

```
In [23]: ma = co2_levels.rolling(window=52).mean()
mstd = co2_levels.rolling(window=52).std()
ma['upper'] = ma['co2'] + (mstd['co2'] * 2)
ma['lower'] = ma['co2'] - (mstd['co2'] * 2)
ax = ma.plot(linewidth=0.8, fontsize=6)
ax.set_xlabel('Date', fontsize=10)
ax.set_ylabel('CO2 levels in Maui Hawaii', fontsize=10)
ax.set_title('Rolling mean and variance of CO2 levels\nin Maui Hawaii from 1958 to 2001', fontsize=10)
plt.show()
```



## Display aggregated values

You may sometimes be required to display your data in a more aggregated form. For example, the `co2_levels` data contains weekly data, but you may need to display its values aggregated by month of year. In datasets such as the `co2_levels` DataFrame where the index is a datetime type, you can extract the year of each dates in the index:

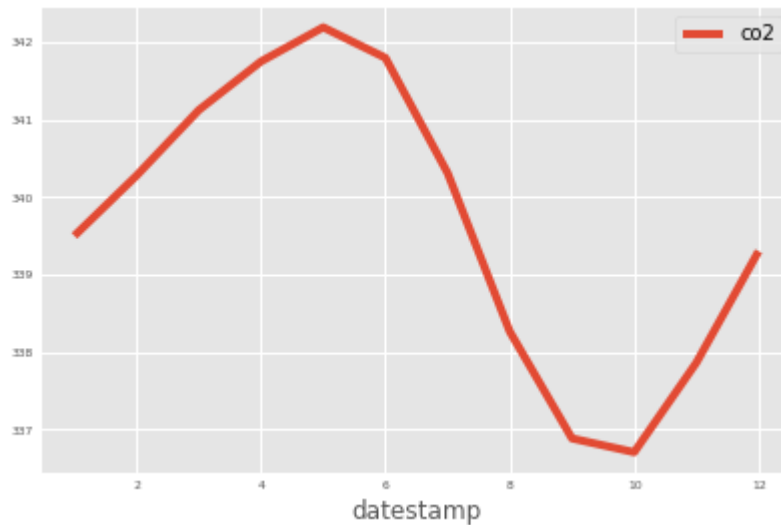
## extract of the year in each dates of the df DataFrame

`index_year = df.index.year` To extract the month or day of the dates in the indices of the `df` DataFrame, you would use `df.index.month` and `df.index.day`, respectively. You can then use the extracted year of each indices in the `co2_levels` DataFrame and the `groupby` function to compute the mean CO2 levels by year:

```
df_by_year = df.groupby(index_year).mean()
```

```
In [33]: co2_levels.index = pd.to_datetime(co2_levels.index)
# This sentence is newly added. Otherwise the code below will not work.

index_month = co2_levels.index.month
mean_co2_levels_by_month = co2_levels.groupby(index_month).mean()
mean_co2_levels_by_month.plot(fontsize=6)
plt.legend(fontsize=10)
plt.show()
```



**Compute numerical summaries**

```
In [35]: print(co2_levels.describe())
```

```
count    2284.000000
mean     339.657750
std       17.100899
min      313.000000
25%      323.975000
50%      337.700000
75%      354.500000
max      373.900000
```

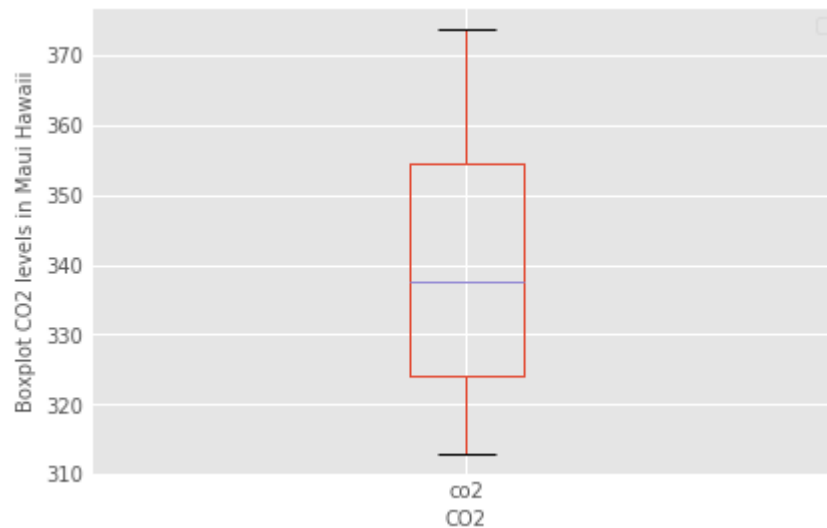
## Boxplots and Histograms

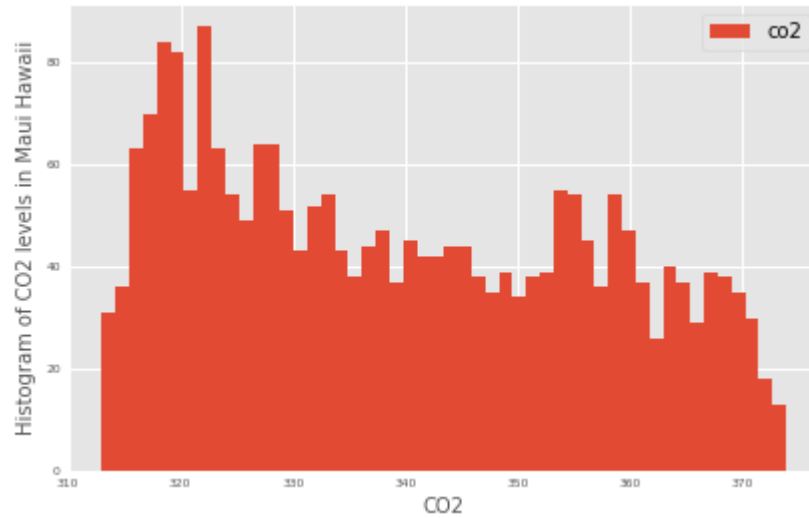
```
In [38]: ax = co2_levels.boxplot()
ax.set_xlabel('CO2', fontsize=10)
ax.set_ylabel('Boxplot CO2 levels in Maui Hawaii', fontsize=10)
plt.legend(fontsize=10)
plt.show()

ax = co2_levels.plot(kind='hist', bins=50, fontsize=6)

ax.set_xlabel('CO2', fontsize=10)
ax.set_ylabel('Histogram of CO2 levels in Maui Hawaii', fontsize=10)
plt.legend(fontsize=10)
plt.show()
```

No handles with labels found to put in legend.

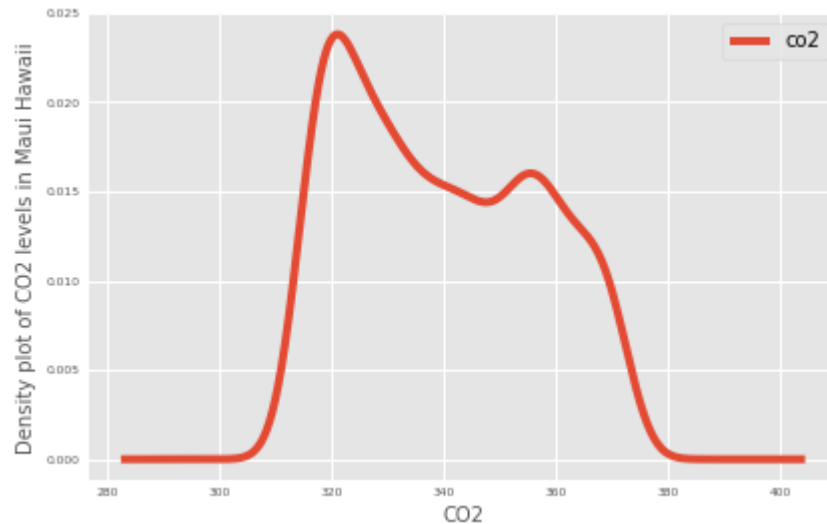




## Density plots

In practice, histograms can be a substandard method for assessing the distribution of your data because they can be strongly affected by the number of bins that have been specified. Instead, kernel density plots represent a more effective way to view the distribution of your data. An example of how to generate a density plot of is shown below:

```
In [39]: ax = co2_levels.plot(kind='density', linewidth=4, fontsize=6)
ax.set_xlabel('CO2', fontsize=10)
ax.set_ylabel('Density plot of CO2 levels in Maui Hawaii', fontsize=10)
plt.show()
```



## Seasonality, Trend and Noise

You will **go beyond summary statistics by learning about autocorrelation and partial autocorrelation plots**. You will also learn how to automatically detect seasonality, trend and noise in your time series data.

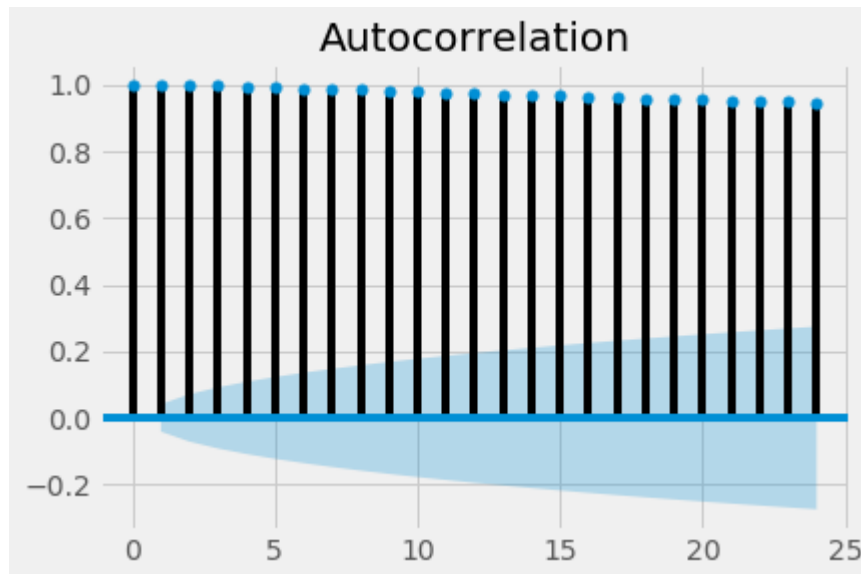
### Autocorrelation in time series data



```
In [40]: import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from statsmodels.graphics import tsaplots
fig = tsaplots.plot_acf(co2_levels['co2'], lags=24)
plt.show()
```

C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\compat\pandas.py:56: FutureWarning: The pandas.core.date tools module is deprecated and will be removed in a future version. Please use the pandas.tseries module instead.

```
from pandas.core import datetools
```



## Interpret autocorrelation plots

If autocorrelation values are close to 0, then values between consecutive observations are not correlated with one another. Inversely, autocorrelations values close to 1 or -1 indicate that there exists strong positive or negative correlations between consecutive observations, respectively. In order to help you assess how trustworthy these autocorrelation values are, the `plot_acf()` function also returns confidence intervals (represented as blue shaded regions). **If an autocorrelation value goes beyond the confidence interval region, you can assume that the observed autocorrelation value is statistically significant.** In other words, this value is not due to the volatility in measurement.

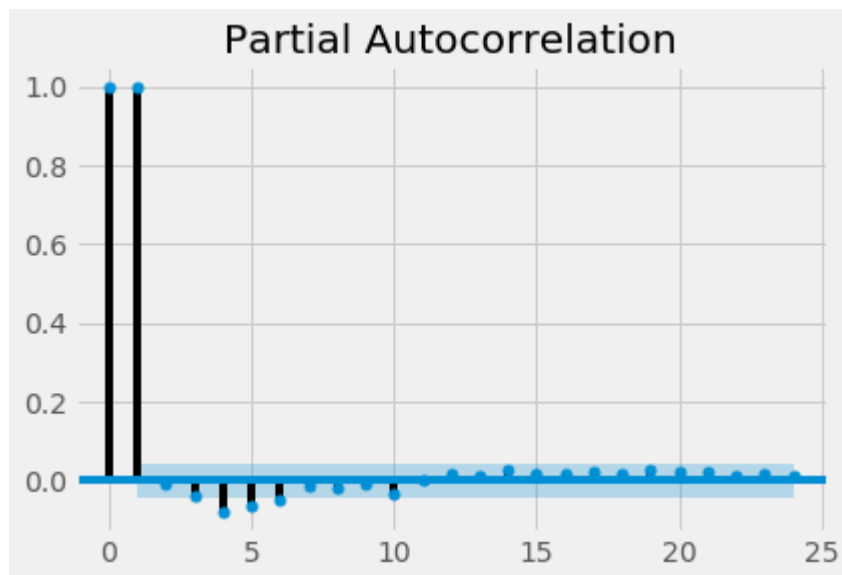
In the autocorrelation plot above, the consecutive observations are highly correlated (i.e. superior to 0.5) and statistically significant.

## Partial autocorrelation in time series data

Like autocorrelation, the partial autocorrelation function (PACF) measures the correlation coefficient between a time-series and lagged versions of itself. However, it extends upon this idea by also removing the effect of previous time points. For example, a partial autocorrelation function of order 3 returns the correlation between our time series ( $t_1, t_2, t_3, \dots$ ) and its own values lagged by 3 time points ( $t_4, t_5, t_6, \dots$ ), but only after removing all effects attributable to lags 1 and 2. **Figure this out in the future**

The `plot_pacf()` function in the `statsmodels` library can be used to measure and plot the partial autocorrelation of a time series.

```
In [41]: import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from statsmodels.graphics import tsaplots
fig = tsaplots.plot_pacf(co2_levels['co2'], lags=24)
plt.show()
```



Just like autocorrelation, partial autocorrelation plots can be tricky to interpret, so let's test your understanding of those!

### Interpret partial autocorrelation plots

If partial autocorrelation values are close to 0, then values between observations and lagged observations are not correlated with one another. Inversely, partial autocorrelations with values close to 1 or -1 indicate that there exists strong positive or negative correlations between the lagged observations of the time series.

The `.plot_pacf()` function also returns confidence intervals, which are represented as blue shaded regions. If partial autocorrelation values are beyond this confidence interval regions, then you can assume that the observed partial autocorrelation values are statistically significant.

In the partial autocorrelation plot below, at which lag values do we have statistically significant partial autocorrelations?

Answer: 0, 1, 4, 5, 6

## **Time series decomposition**

When visualizing time series data, you should look out for some distinguishable patterns:

seasonality: does the data display a clear periodic pattern?

trend: does the data follow a consistent upwards or downward slope?

noise: are there any outlier points or missing values that are not consistent with the rest of the data?

You can rely on a method known as time-series decomposition to automatically extract and quantify the structure of time-series data. The `statsmodels` library provides the `seasonal_decompose()` function to perform time series decomposition out of the box.

```
In [42]: import statsmodels.api as sm
decomposition = sm.tsa.seasonal_decompose(co2_levels)
print(decomposition.seasonal)
```

```
co2
datestamp
1958-03-29  1.028042
1958-04-05  1.235242
1958-04-12  1.412344
1958-04-19  1.701186
1958-04-26  1.950694
1958-05-03  2.032939
1958-05-10  2.445506
1958-05-17  2.535041
1958-05-24  2.662031
1958-05-31  2.837948
1958-06-07  2.786137
1958-06-14  2.897139
1958-06-21  2.700962
1958-06-28  2.637389
1958-07-05  2.499487
1958-07-12  2.328869
1958-07-19  2.016146
1958-07-26  1.696378
1958-08-02  1.320640
1958-08-09  0.900761
1958-08-16  0.515989
1958-08-23  0.086897
1958-08-30 -0.474590
1958-09-06 -0.810900
1958-09-13 -1.287685
1958-09-20 -1.805108
1958-09-27 -2.068716
1958-10-04 -2.560531
1958-10-11 -2.856752
1958-10-18 -3.108765
...
2001-06-09  1.320640
2001-06-16  0.900761
2001-06-23  0.515989
2001-06-30  0.086897
2001-07-07 -0.474590
```

```
2001-07-14 -0.810900
2001-07-21 -1.287685
2001-07-28 -1.805108
2001-08-04 -2.068716
2001-08-11 -2.560531
2001-08-18 -2.856752
2001-08-25 -3.108765
2001-09-01 -3.170460
2001-09-08 -3.267396
2001-09-15 -3.194297
2001-09-22 -3.016323
2001-09-29 -2.812656
2001-10-06 -2.588640
2001-10-13 -2.351296
2001-10-20 -2.072159
2001-10-27 -1.802325
2001-11-03 -1.509391
2001-11-10 -1.284167
2001-11-17 -1.024060
2001-11-24 -0.791949
2001-12-01 -0.525044
2001-12-08 -0.392799
2001-12-15 -0.134838
2001-12-22  0.116056
2001-12-29  0.285354
```

```
[2284 rows x 1 columns]
```

Time series decomposition is a powerful method to reveal the structure of your time series. Now let's visualize these components.

## Plot individual components

It is also possible to extract other inferred quantities from your time-series decomposition object. The following code shows you how to extract the observed, trend and noise (or residual, resid) components.

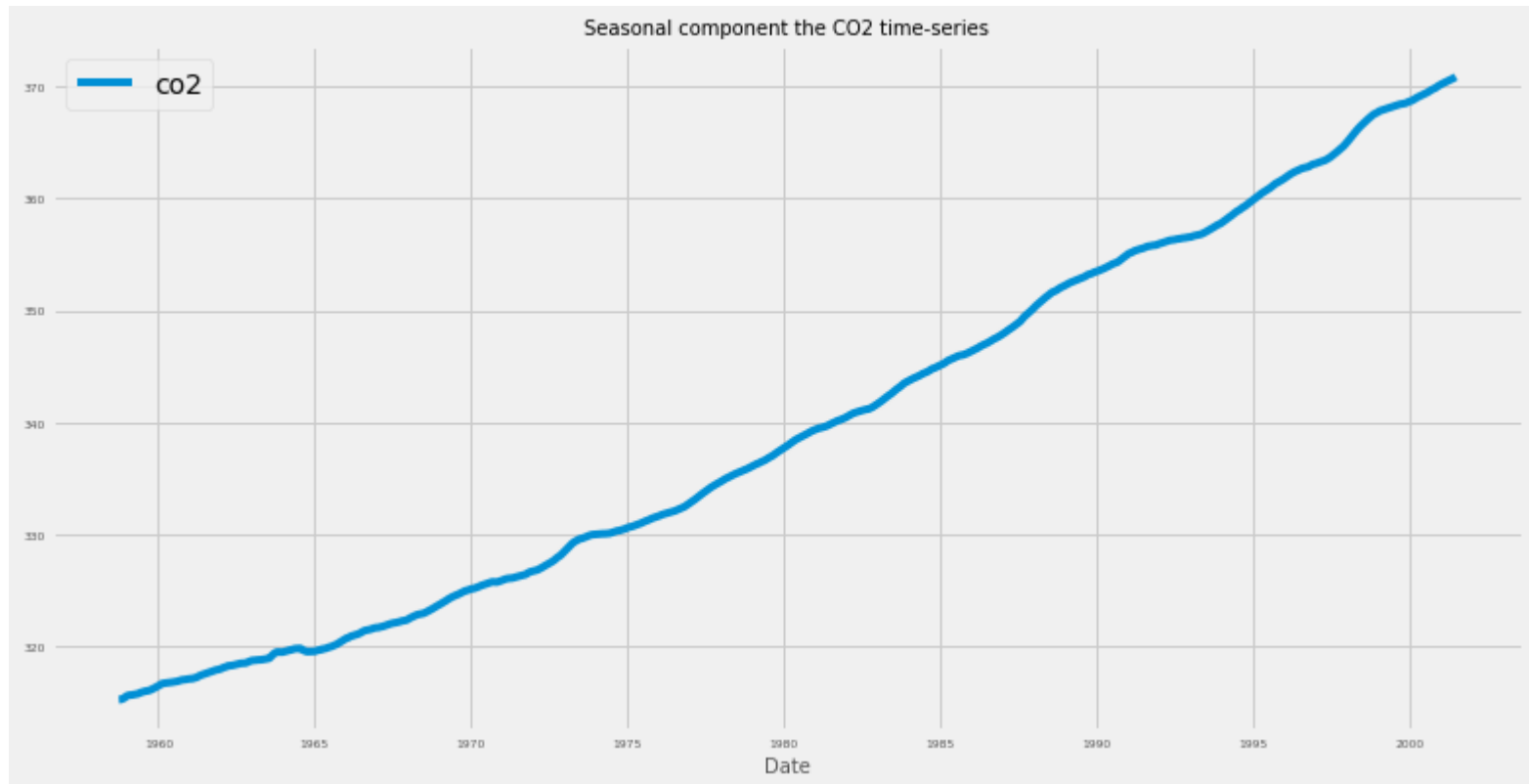
```
observed = decomposition.observed
```

```
trend = decomposition.trend
```

```
residuals = decomposition.resid
```

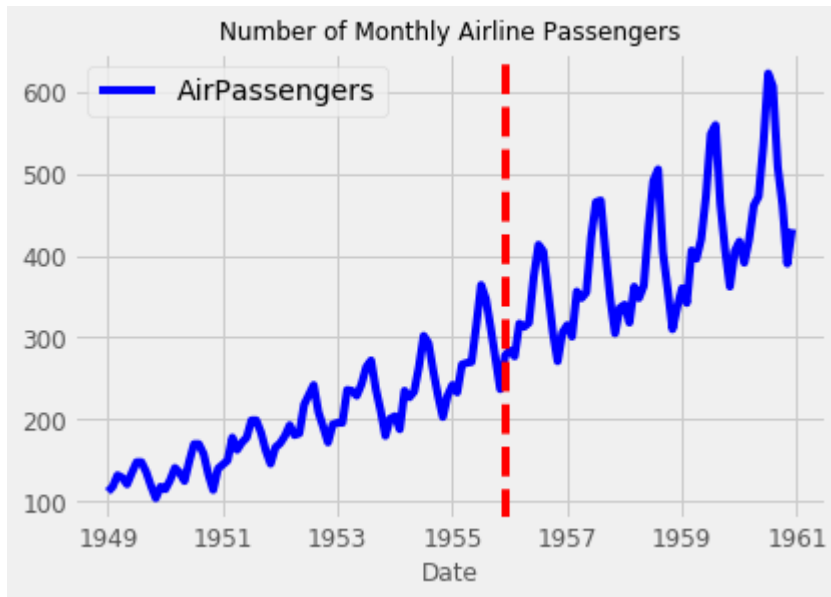
You can then use the extracted components and plot them individually.

```
In [43]: trend = decomposition.trend
ax = trend.plot(figsize=(12, 6), fontsize=6)
ax.set_xlabel('Date', fontsize=10)
ax.set_title('Seasonal component the CO2 time-series', fontsize=10)
plt.show()
```



**Visualize the airline dataset**

```
In [47]: import pandas as pd
airline = pd.read_csv('ch3_airline_passengers.csv', index_col = 'Month', parse_dates = True)
ax = airline.plot(color='blue', fontsize=12)
ax.axvline('1955-12-01', color='red', linestyle='--')
ax.set_xlabel('Date', fontsize=12)
ax.set_title('Number of Monthly Airline Passengers', fontsize=12)
plt.show()
```



## Analyze the airline dataset

- How to check for the presence of missing values, and how to collect summary statistics of time series data contained in a pandas DataFrame.
- To generate boxplots of your data to quickly gain insight in your data.
- Display aggregate statistics of your data using groupby().

```
In [48]: print(airline.isnull().sum())
print(airline.describe())

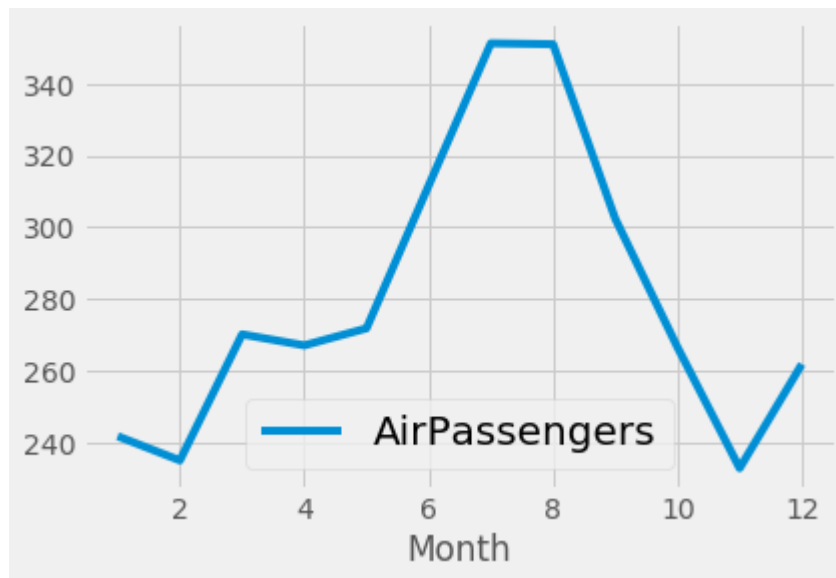
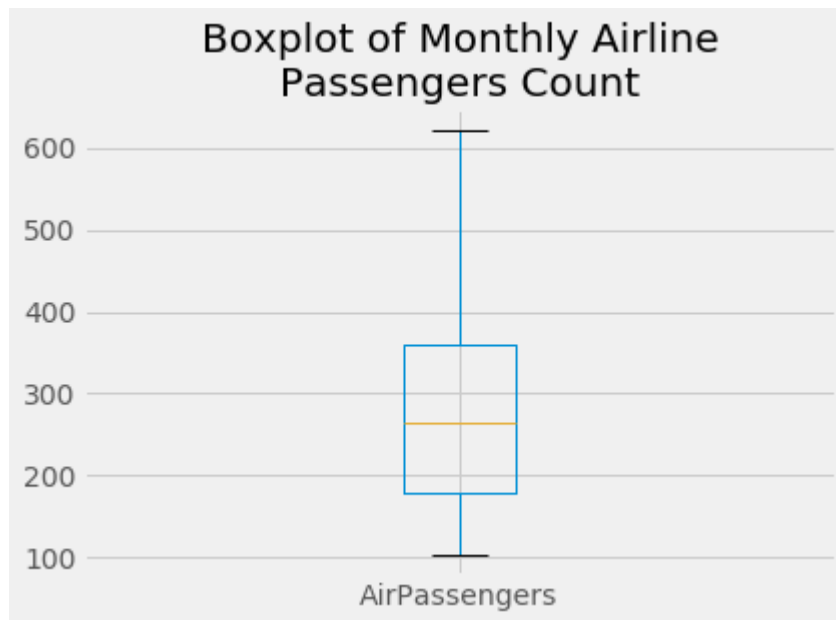
ax = airline.boxplot()
ax.set_title('Boxplot of Monthly Airline\nPassengers Count', fontsize=20)
plt.show()

index_month = airline.index.month
mean_airline_by_month = airline.groupby(index_month).mean()
mean_airline_by_month.plot()
plt.legend(fontsize=20)
plt.show()
```

```
AirPassengers    0
dtype: int64

      AirPassengers
count    144.000000
mean     280.298611
std      119.966317
min      104.000000
25%      180.000000
50%      265.500000
75%      360.500000
max      622.000000
```





**Time series decomposition of the airline dataset**

```
In [ ]: import statsmodels.api as sm
decomposition = sm.tsa.seasonal_decompose(airline)
trend = decomposition.trend
seasonal = decomposition.seasonal

#To print, I need put trend and seasonal above into the airline_decomposed below
print(airline_decomposed.head(5))
ax = airline_decomposed.plot(figsize=(12, 6), fontsize=15)
ax.set_xlabel('Date', fontsize=15)
plt.legend(fontsize=15)
plt.show()
```

## Work with Multiple Time Series

### Load multiple time series

```
In [52]: meat = pd.read_csv('ch4_meat.csv', index_col = 'date', parse_dates = True)
#The above sentence replace the following sentences.
print(meat.head(5))
# meat['date'] = pd.to_datetime(meat['date'])
# meat = meat.set_index('date')
print(meat.describe())
```

	beef	veal	pork	lamb_and_mutton	broilers	other_chicken	\
date							
1944-01-01	751.0	85.0	1280.0	89.0	NaN	NaN	
1944-02-01	713.0	77.0	1169.0	72.0	NaN	NaN	
1944-03-01	741.0	90.0	1128.0	75.0	NaN	NaN	
1944-04-01	650.0	89.0	978.0	66.0	NaN	NaN	
1944-05-01	681.0	106.0	1029.0	78.0	NaN	NaN	

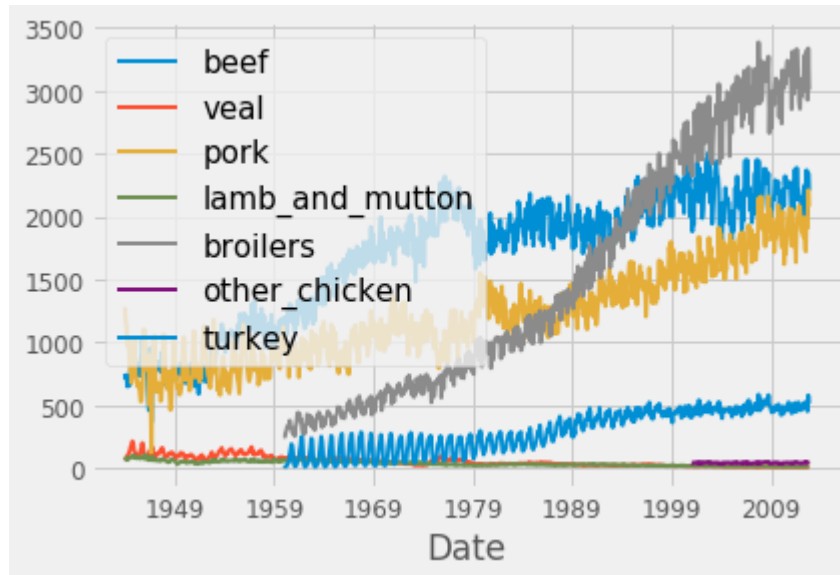
	turkey
date	
1944-01-01	NaN
1944-02-01	NaN
1944-03-01	NaN
1944-04-01	NaN
1944-05-01	NaN

	beef	veal	pork	lamb_and_mutton	broilers	\
count	827.000000	827.000000	827.000000	827.000000	635.000000	
mean	1683.463362	54.198549	1211.683797	38.360701	1516.582520	
std	501.698480	39.062804	371.311802	19.624340	963.012101	
min	366.000000	8.800000	124.000000	10.900000	250.900000	
25%	1231.500000	24.000000	934.500000	23.000000	636.350000	
50%	1853.000000	40.000000	1156.000000	31.000000	1211.300000	
75%	2070.000000	79.000000	1466.000000	55.000000	2426.650000	
max	2512.000000	215.000000	2210.400000	109.000000	3383.800000	

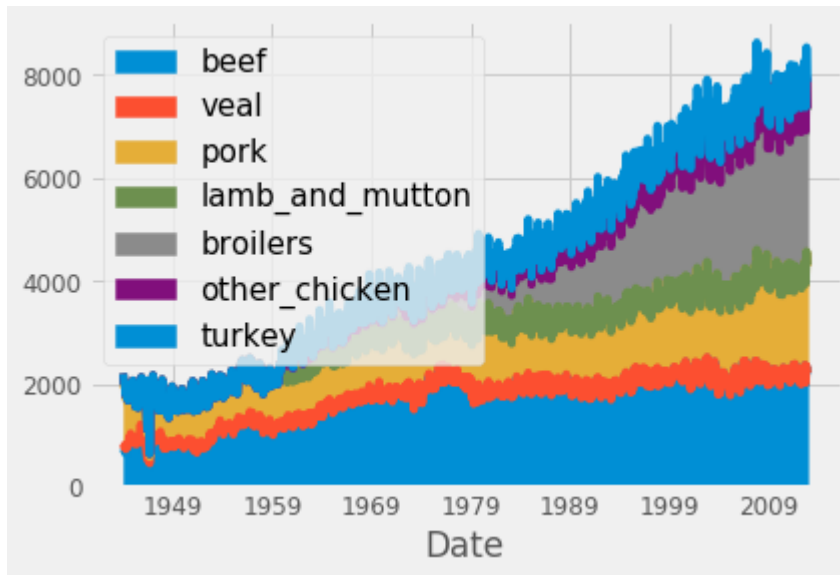
	other_chicken	turkey
count	143.000000	635.000000
mean	43.033566	292.814646
std	3.867141	162.482638
min	32.300000	12.400000
25%	40.200000	154.150000
50%	43.400000	278.300000
75%	45.650000	449.150000
max	51.100000	585.100000

## Visualize multiple time series

```
In [53]: ax = meat.plot(linewidth=2, fontsize=12)
ax.set_xlabel('Date')
ax.legend(fontsize=15)
plt.show()
```



```
In [54]: # Plot an area chart
ax = meat.plot.area(fontsize=12)
ax.set_xlabel('Date')
ax.legend(fontsize=15)
plt.show()
```

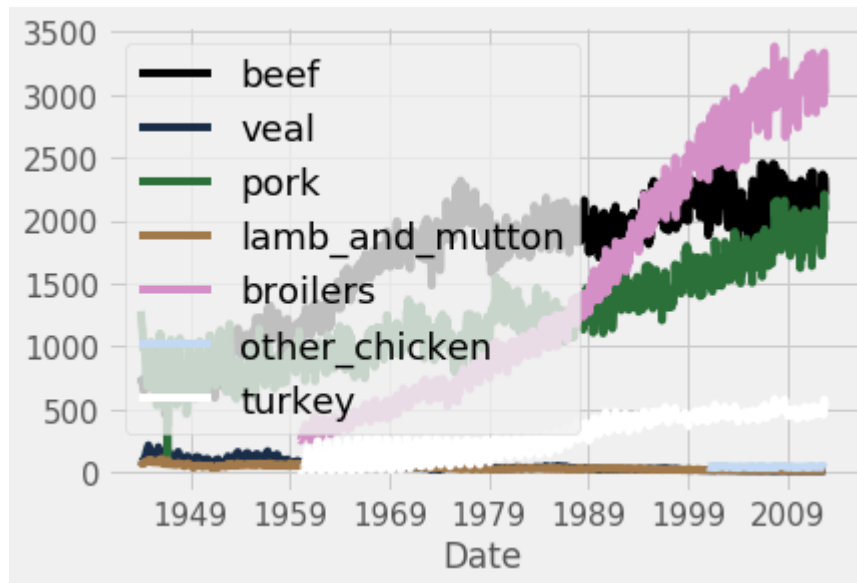


## Define the color palette of your plots

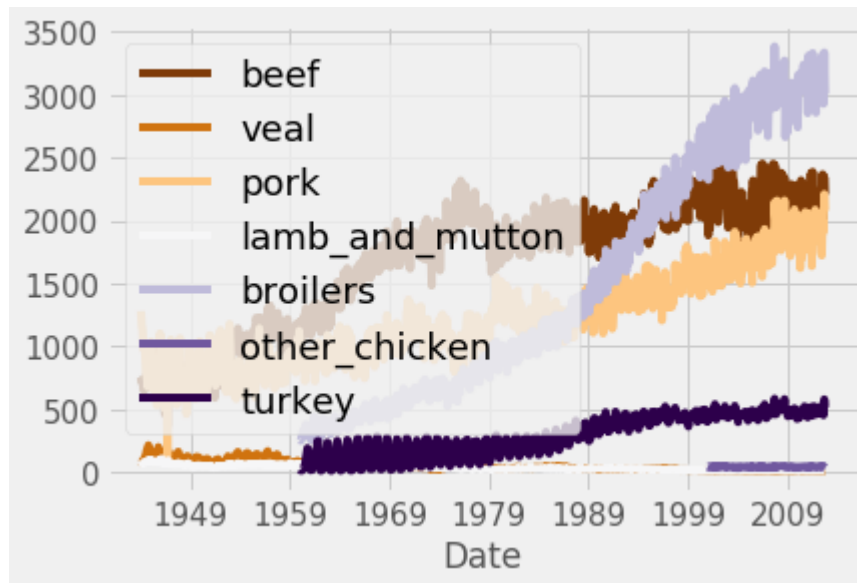
When visualizing multiple time series, it can be difficult to differentiate between various colors in the default color scheme.

To remedy this, you can define each color manually, but this may be time-consuming. Fortunately, it is possible to leverage the `colormap` argument to `.plot()` to automatically assign specific color palettes with varying contrasts. You can either provide a matplotlib colormap as an input to this parameter, or provide one of the default strings that is available in the `colormap()` function available in matplotlib (all of which are available [here](#)).

```
In [55]: ax = meat.plot(colormap='cubehelix', fontsize=15)
ax.set_xlabel('Date')
ax.legend(fontsize=18)
plt.show()
```



```
In [56]: ax = meat.plot(colormap='PuOr', fontsize=15)
ax.set_xlabel('Date')
ax.legend(fontsize=18)
plt.show()
```



## Add summary statistics to your time series plot

It is possible to visualize time series plots and numerical summaries on one single graph by using the pandas API to matplotlib along with the table method:

```
In [ ]: ax = meat.plot(fontsize=6, linewidth=1)
ax.set_xlabel('Date', fontsize=6)

# Add summary table information to the plot
ax.table(cellText=meat_mean.values,
         colWidths = [0.15]*len(meat_mean.columns),
         rowLabels=meat_mean.index,
         colLabels=meat_mean.columns,
         loc='top')

ax.legend(loc='upper center', bbox_to_anchor=(0.5, 0.95), ncol=3, fontsize=6)

plt.show()
```

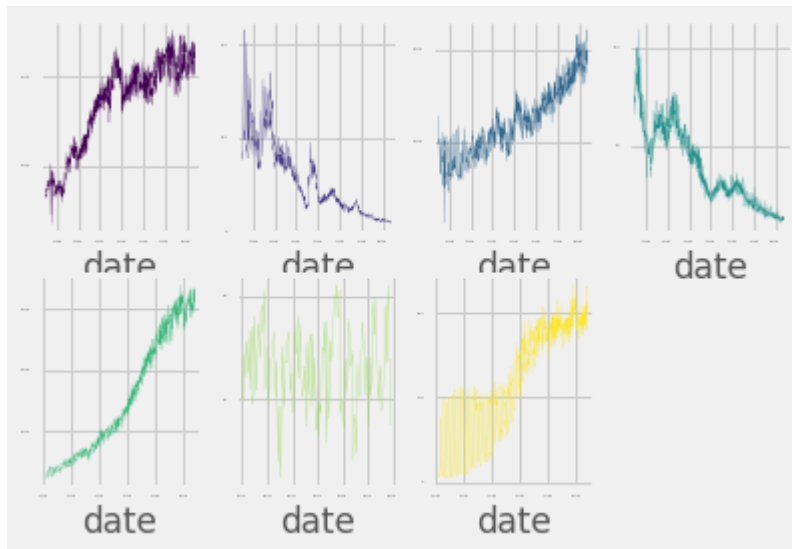
**Plot your time series on individual plots**



In [58]: *# Create a faceted graph with 2 rows and 4 columns*

```
meat.plot(subplots=True,  
          layout=(2, 4),  
          sharex=False,  
          sharey=False,  
          colormap='viridis',  
          fontsize=2,  
          legend=False,  
          linewidth=0.2)
```

```
plt.show()
```



## Compute correlations between time series

Correlation coefficients can be computed with the pearson, kendall and spearman methods. A full discussion of these different methods is outside the scope of this course, but the pearson method should be used when relationships between your variables are thought to be linear, while the kendall and spearman methods should be used when relationships between your variables are **thought to be non-linear**.

```
In [60]: print(meat[['beef', 'pork']].corr(method='spearman'))
```

	beef	pork
beef	1.000000	0.827587
pork	0.827587	1.000000

```
In [61]: print(meat[['pork', 'veal', 'turkey']].corr(method='pearson'))
```

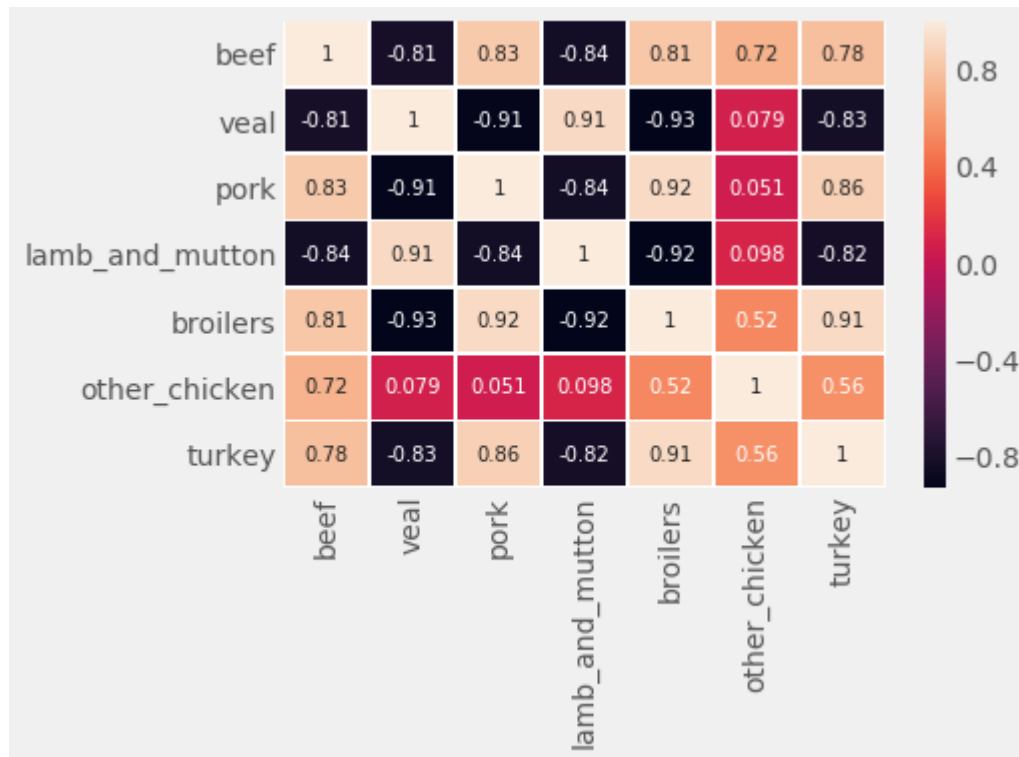
	pork	veal	turkey
pork	1.000000	-0.808834	0.835215
veal	-0.808834	1.000000	-0.768366
turkey	0.835215	-0.768366	1.000000

## Visualize correlation matrices

The correlation matrix generated in the previous exercise can be plotted using a heatmap. To do so, you can leverage the `heatmap()` function from the `seaborn` library which contains several arguments to tailor the look of your heatmap.

```
In [62]: import seaborn as sns
corr_meat = meat.corr(method='spearman')
sns.heatmap(corr_meat,
            annot=True,
            linewidths=0.4,
            annot_kws={"size": 10})

plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.show()
```



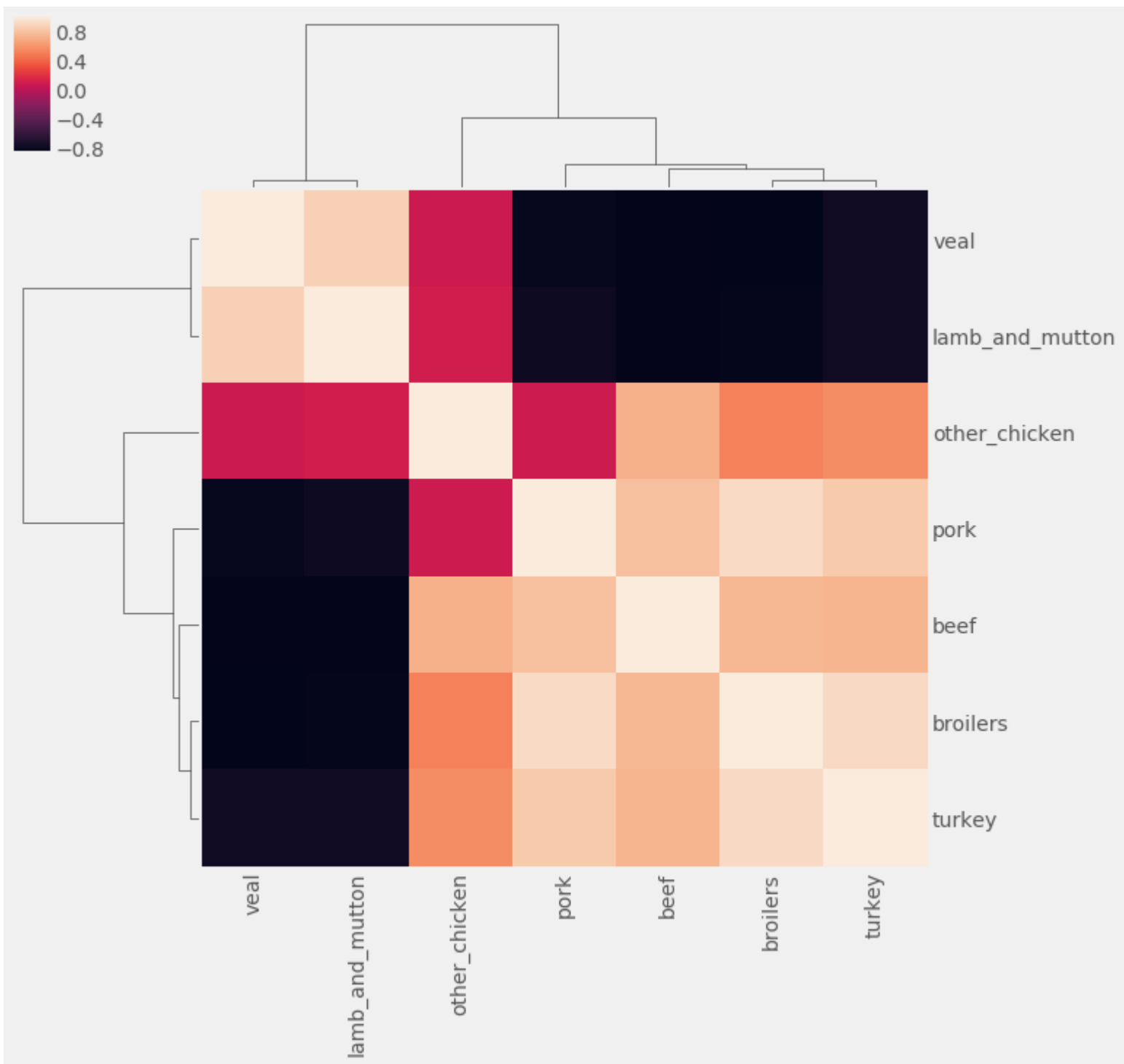
## Clustered heatmaps

Heatmaps are extremely useful to visualize a correlation matrix, but clustermaps are better. A Clustermap allows to uncover structure in a correlation matrix by producing a hierarchically-clustered heatmap:

```
In [63]: import seaborn as sns
corr_meat = meat.corr(method='pearson')

# Customize the heatmap of the corr_meat correlation matrix and rotate the x-axis labels
fig = sns.clustermap(corr_meat,
                     row_cluster=True,
                     col_cluster=True,
                     figsize=(10, 10))

plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90)
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
plt.show()
```



## Case Study

This chapter will give you a chance to practice all the concepts covered in the course. You will visualize the unemployment rate in the US from 2000 to 2010.

```
In [65]: import pandas as pd
jobs = pd.read_csv('ch5_employment.csv')

print(jobs.head(5))
print(jobs.dtypes)
jobs['datestamp'] = pd.to_datetime(jobs['datestamp'])
jobs = jobs.set_index('datestamp')
print(jobs.isnull().sum())
```

	datestamp	Agriculture	Business services	Construction	\
0	2000-01-01	10.3	5.7	9.7	
1	2000-02-01	11.5	5.2	10.6	
2	2000-03-01	10.4	5.4	8.7	
3	2000-04-01	8.9	4.5	5.8	
4	2000-05-01	5.1	4.7	5.0	

	Durable goods manufacturing	Education and Health	Finance	Government	\
0	3.2	2.3	2.7	2.1	
1	2.9	2.2	2.8	2.0	
2	2.8	2.5	2.6	1.5	
3	3.4	2.1	2.3	1.3	
4	3.4	2.7	2.2	1.9	

	Information	Leisure and hospitality	Manufacturing	Mining and Extraction	\
0	3.4	7.5	3.6	3.9	
1	2.9	7.5	3.4	5.5	
2	3.6	7.4	3.6	3.7	
3	2.4	6.1	3.7	4.1	
4	3.5	6.2	3.4	5.3	

	Nondurable goods manufacturing	Other	Self-employed	\
0	4.4	4.9	2.3	
1	4.2	4.1	2.5	
2	5.1	4.3	2.0	
3	4.0	4.2	2.0	
4	3.6	4.5	1.9	

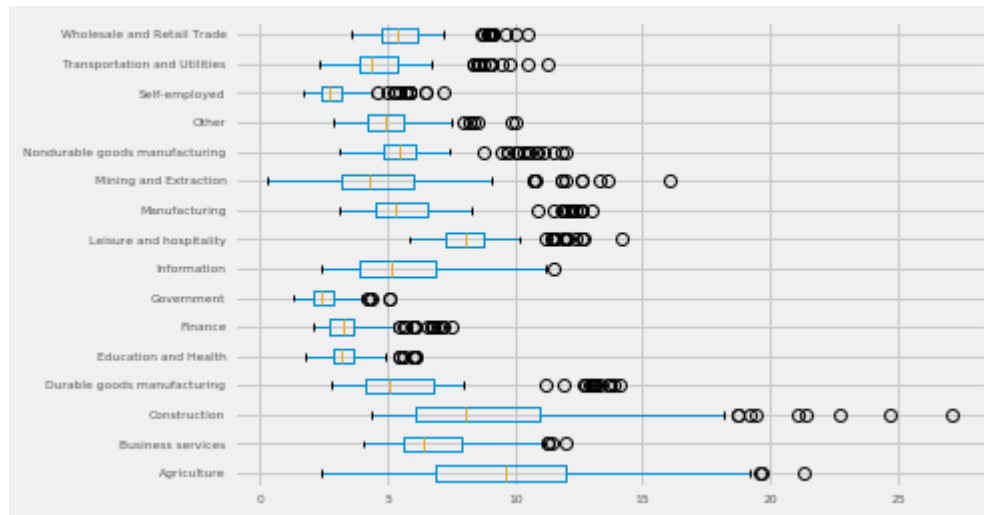
	Transportation and Utilities	Wholesale and Retail Trade
0	4.3	5.0
1	4.0	5.2
2	3.5	5.1
3	3.4	4.1

4	3.4	4.3
datestamp		object
Agriculture		float64
Business services		float64
Construction		float64
Durable goods manufacturing		float64
Education and Health		float64
Finance		float64
Government		float64
Information		float64
Leisure and hospitality		float64
Manufacturing		float64
Mining and Extraction		float64
Nondurable goods manufacturing		float64
Other		float64
Self-employed		float64
Transportation and Utilities		float64
Wholesale and Retail Trade		float64
dtype: object		
Agriculture	0	
Business services	0	
Construction	0	
Durable goods manufacturing	0	
Education and Health	0	
Finance	0	
Government	0	
Information	0	
Leisure and hospitality	0	
Manufacturing	0	
Mining and Extraction	0	
Nondurable goods manufacturing	0	
Other	0	
Self-employed	0	
Transportation and Utilities	0	
Wholesale and Retail Trade	0	
dtype: int64		

**Describe time series data with boxplots**



```
In [66]: jobs.boxplot(fontsize=6, vert=False)
plt.show()
print(jobs.describe())
print('Agriculture')
print('Construction')
```



	Agriculture	Business services	Construction \
count	122.000000	122.000000	122.000000
mean	9.840984	6.919672	9.426230
std	3.962067	1.862534	4.587619
min	2.400000	4.100000	4.400000
25%	6.900000	5.600000	6.100000
50%	9.600000	6.450000	8.100000
75%	11.950000	7.875000	10.975000
max	21.300000	12.000000	27.100000

	Durable goods manufacturing	Education and Health	Finance \
count	122.000000	122.000000	122.000000
mean	6.025410	3.420492	3.540164
std	2.854475	0.877538	1.235405
min	2.800000	1.800000	2.100000
25%	4.125000	2.900000	2.700000
50%	5.100000	3.200000	3.300000
75%	6.775000	3.700000	3.700000
max	14.100000	6.100000	7.500000

	Government	Information	Leisure and hospitality	Manufacturing \
count	122.000000	122.000000	122.000000	122.000000
mean	2.581148	5.486885	8.315574	5.982787
std	0.686750	2.016582	1.605570	2.484221
min	1.300000	2.400000	5.900000	3.100000
25%	2.100000	3.900000	7.300000	4.500000
50%	2.400000	5.150000	8.050000	5.300000
75%	2.875000	6.900000	8.800000	6.600000
max	5.100000	11.500000	14.200000	13.000000

	Mining and Extraction	Nondurable goods manufacturing	Other \
count	122.000000	122.000000	122.000000
mean	5.088525	5.930328	5.096721
std	2.942428	1.922330	1.317457
min	0.300000	3.100000	2.900000
25%	3.200000	4.825000	4.200000
50%	4.300000	5.500000	4.900000
75%	6.050000	6.100000	5.600000
max	16.100000	12.000000	10.000000

	Self-employed	Transportation and Utilities	Wholesale and Retail Trade
count	122.000000	122.000000	122.000000
mean	3.031967	4.935246	5.766393
std	1.124429	1.753340	1.463417
min	1.700000	2.300000	3.600000
25%	2.400000	3.900000	4.800000
50%	2.700000	4.400000	5.400000
75%	3.200000	5.400000	6.200000
max	7.200000	11.300000	10.500000

Agriculture  
Construction

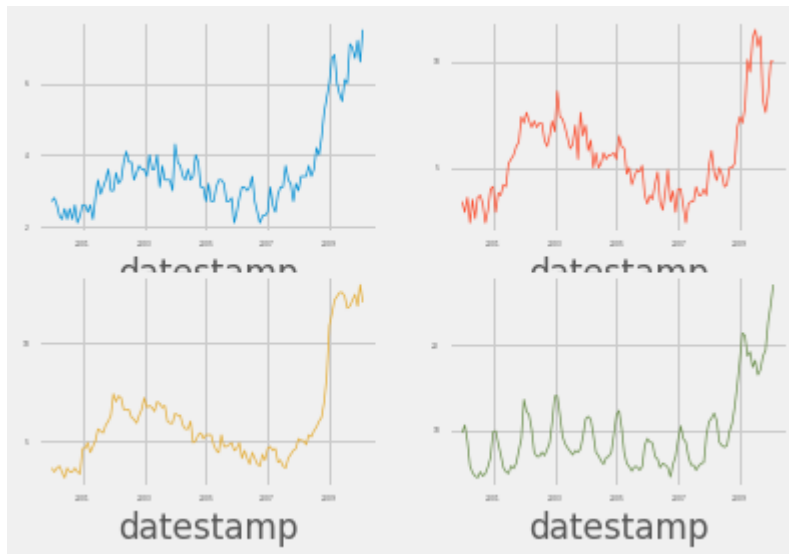
## Plot all the time series in your dataset

The jobs DataFrame contains 16 time series representing the unemployment rate of various industries between 2001 and 2010. This may seem like a large amount of time series to visualize at the same time, but Chapter 4 introduced you to faceted plots. In this exercise, you will explore some of the time series in the jobs DataFrame and look to extract some meaningful information from these plots.

```
In [68]: jobs_subset = jobs[['Finance', 'Information', 'Manufacturing', 'Construction']]
print(jobs_subset.head(5))
ax = jobs_subset.plot(subplots=True,
                      layout=(2, 2),
                      sharex=False,
                      sharey=False,
                      linewidth=0.7,
                      fontsize=3,
                      legend=False)

plt.show()
```

	Finance	Information	Manufacturing	Construction
datestamp				
2000-01-01	2.7	3.4	3.6	9.7
2000-02-01	2.8	2.9	3.4	10.6
2000-03-01	2.6	3.6	3.6	8.7
2000-04-01	2.3	2.4	3.7	5.8
2000-05-01	2.2	3.5	3.4	5.0

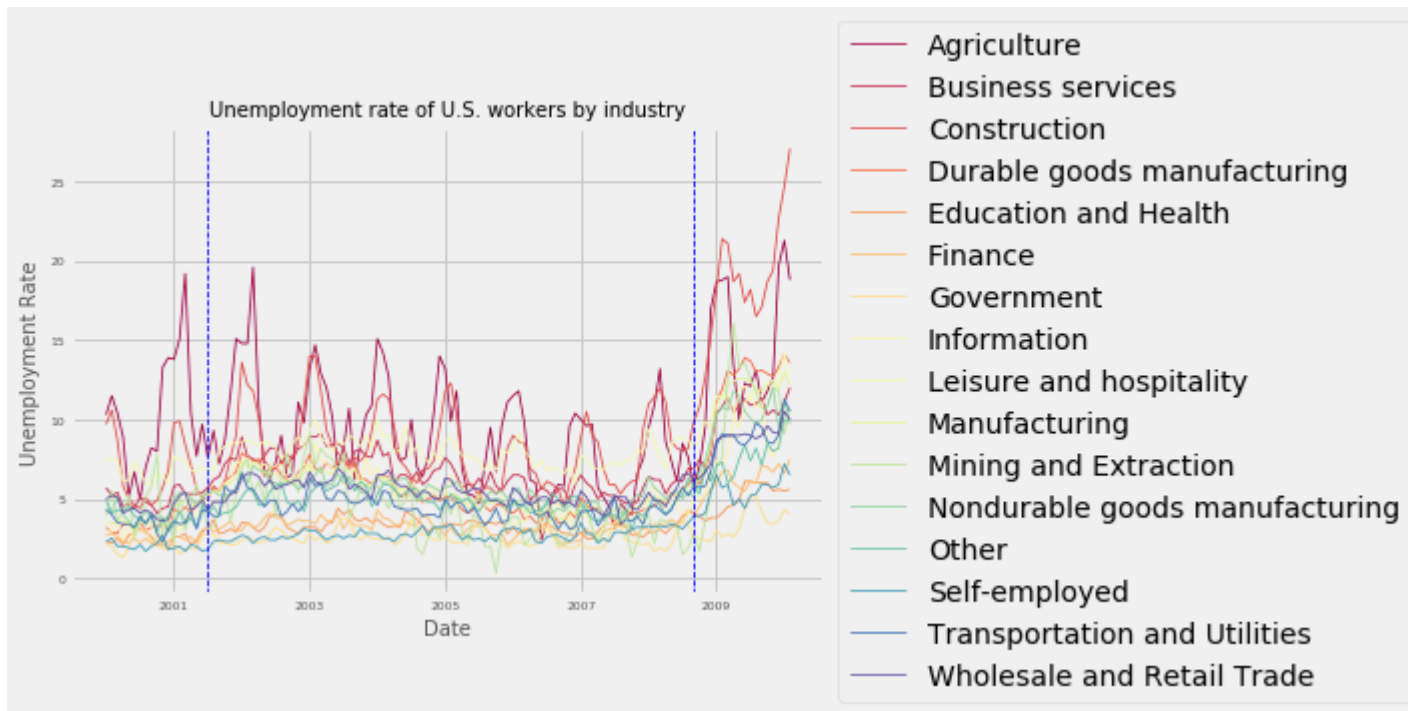


**Annotate significant events in time series data**

```
In [69]: ax = jobs.plot(colormap='Spectral', fontsize=6, linewidth=0.8)
ax.set_xlabel('Date', fontsize=10)
ax.set_ylabel('Unemployment Rate', fontsize=10)
ax.set_title('Unemployment rate of U.S. workers by industry', fontsize=10)
ax.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))

ax.axvline('2001-07-01', color='blue', linestyle='--', linewidth=0.8)
ax.axvline('2008-09-01', color='blue', linestyle='--', linewidth=0.8)

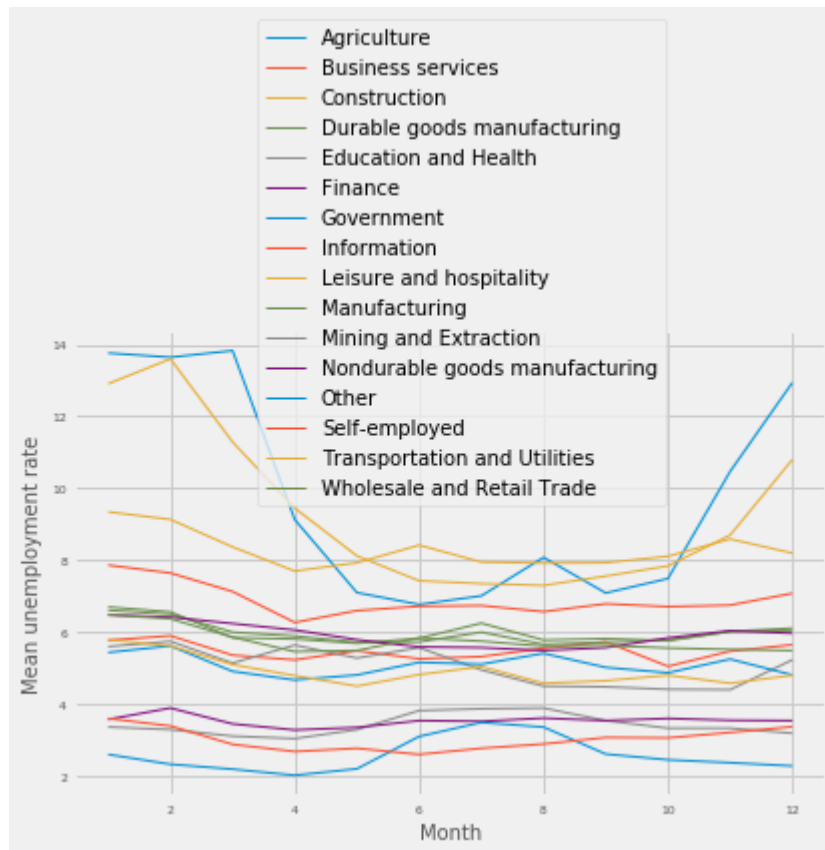
plt.show()
```



**Plot monthly and yearly trends**

```
In [70]: index_month = jobs.index.month
jobs_by_month = jobs.groupby(index_month).mean()

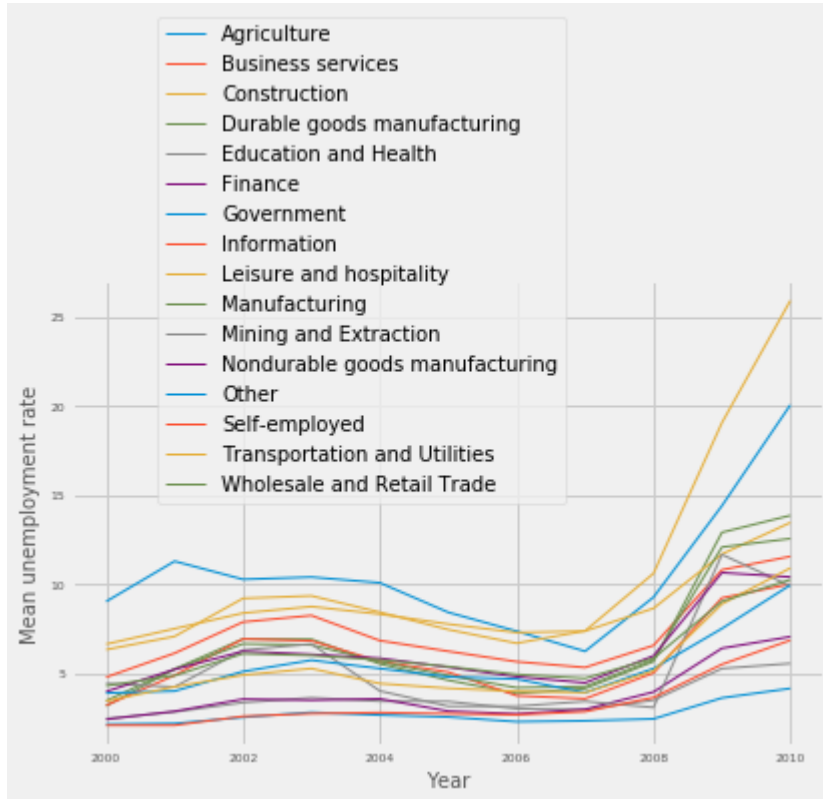
# Plot the mean unemployment rate for each month
ax = jobs_by_month.plot(fontsize=6, linewidth=1)
ax.set_xlabel('Month', fontsize=10)
ax.set_ylabel('Mean unemployment rate', fontsize=10)
ax.legend(bbox_to_anchor=(0.8, 0.6), fontsize=10)
plt.show()
```



**Plot monthly and yearly trends**

```
In [71]: index_year = jobs.index.year
jobs_by_year = jobs.groupby(index_year).mean()

ax = jobs_by_year.plot(fontsize=6, linewidth=1)
ax.set_xlabel('Year', fontsize=10)
ax.set_ylabel('Mean unemployment rate', fontsize=10)
ax.legend(bbox_to_anchor=(0.1, 0.5), fontsize=10)
plt.show()
```



## Apply time series decomposition to your dataset

You will now perform time series decomposition on multiple time series. You can achieve this by leveraging the Python dictionary to store the results of each time series decomposition.

In this exercise, you will initialize an empty dictionary with a set of curly braces, {}, use a for loop to iterate through the columns of the DataFrame and apply time series decomposition to each time series. After each time series decomposition, you place the results in the dictionary by using the command `my_dict[key] = value`, where `my_dict` is your dictionary, `key` is the name of the column/time series, and `value` is the decomposition object of that time series.

```
In [80]: jobs_decomp = {}  
         jobs_names = jobs.columns  
  
         for ts in jobs_names:  
             ts_decomposition = sm.tsa.seasonal_decompose(jobs[ts])  
             jobs_decomp[ts] = ts_decomposition
```

## Visualize the seasonality of multiple time series

You will now extract the seasonality component of `jobs_decomp` to visualize the seasonality in these time series. Note that before plotting, you will have to convert the dictionary of seasonality components into a DataFrame using the `pd.DataFrame.from_dict()` function.

An empty dictionary `jobs_seasonal` and the time series decomposition object `jobs_decomp` from the previous exercise are available in your workspace.

```
In [ ]: # Extract the seasonal values for the decomposition of each time series
        for ts in jobs_names:
            jobs_seasonal[ts] = jobs_decomp[ts].seasonal

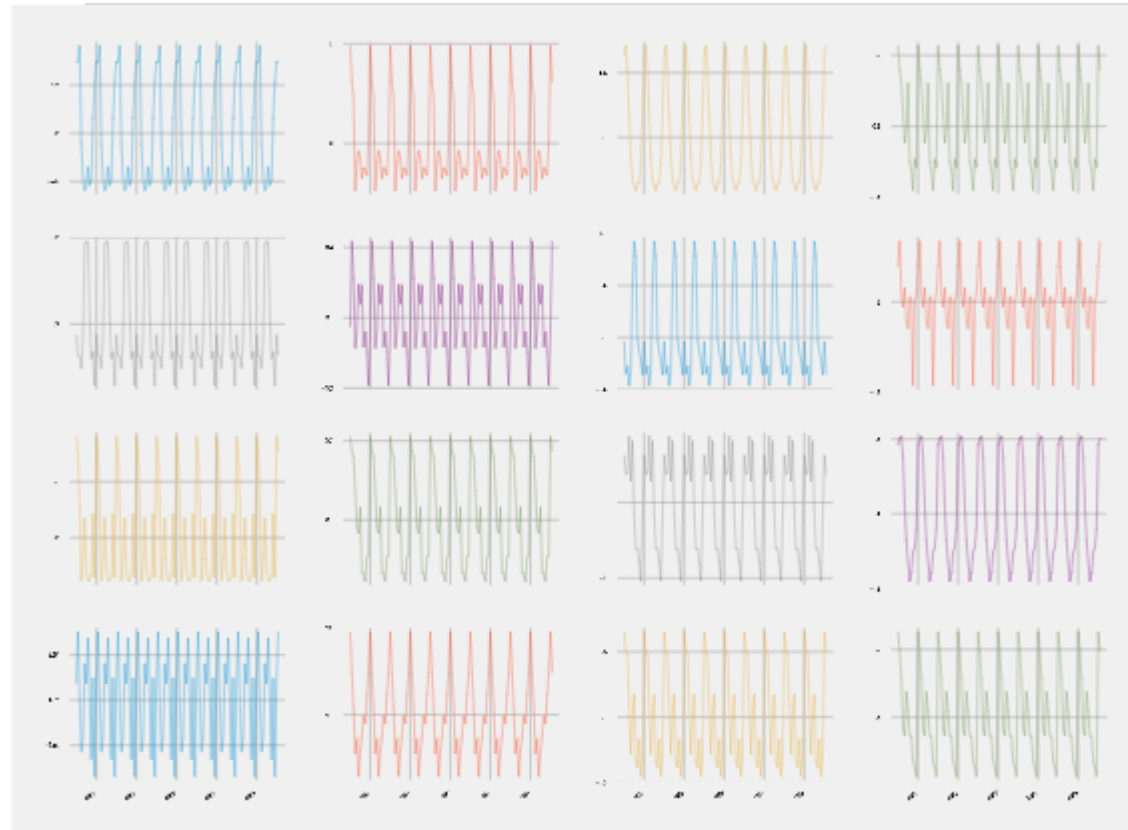
        # Create a DataFrame from the jobs_seasonal dictionary
        seasonality_df = pd.DataFrame.from_dict(jobs_seasonal)

        # Remove the label for the index
        seasonality_df.index.name = None

        # Create a faceted plot of the seasonality_df DataFrame
        seasonality_df.plot(subplots=True,
                             layout=(4, 4),
                             sharey=False,
                             fontsize=2,
                             linewidth=0.3,
                             legend=False)

        # Show plot
        plt.show()
```





```
In [ ]: # Get correlation matrix of the seasonality_df DataFrame
seasonality_corr = seasonality_df.corr(method='spearman')

# Customize the clustermap of the seasonality_corr correlation matrix
fig = sns.clustermap(seasonality_corr, annot=True, annot_kws={"size": 4}, linewidths=.4, figsize=(15, 10))
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90)
plt.show()

# Print the correlation between the seasonalities of the Government and Education & Health industries
print(0.89)
```

