

# Introduction to Python

## Chapter 1 - Python Basics

```
In [ ]: print(4+5)
```

```
# Subtraction  
print(5-5)
```

```
# Multiplication  
print(3*5)
```

```
# Division  
print(10/2)
```

```
9  
0  
15  
5.0
```

```
In [ ]: # Create a variable savings  
savings = 100
```

```
# Print out savings  
print (savings)
```

```
100
```

```
In [ ]: # Create the variables monthly_savings and num_months  
monthly_savings = 10  
num_months = 4
```

```
# Multiply monthly_savings and num_months, save the result as new_savings  
new_savings = monthly_savings * num_months
```

```
# Add new_savings to your savings, save the sum as total_savings  
total_savings = new_savings + savings
```

```
# Print total_savings  
print(total_savings)
```

```
140
```

```
In [ ]: # Create a variable half  
half = 0.5
```

```
# Create a variable intro
intro = "Hello! How are you?"
```

```
# Create a variable is_good
is_good = True
```

```
In [ ]: monthly_savings = 10
num_months = 12
intro = "Hello! How are you?"

# Calculate year_savings using monthly_savings and num_months
year_savings = monthly_savings*num_months

# Print the type of year_savings
print(type(year_savings))

# Assign sum of intro and intro to doubleintro
doubleintro = intro + intro

# Print out doubleintro
print(doubleintro)

<class 'int'>
Hello! How are you?Hello! How are you?
```

```
In [ ]: # Definition of savings and total_savings
savings = 100
total_savings = 150

# Fix the printout
print("I started with $" + str(savings) + " and now have $" + str(total_savings) + ". Awesome!")

# Definition of pi_string
pi_string = "3.1415926"

# Convert pi_string into float: pi_float
pi_float = float(pi_string)
```

```
I started with $100 and now have $150. Awesome!
```

## Chapter 2 - Python Lists

```
In [ ]: # area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50
```

```
# Create list areas
areas = [hall, kit, liv, bed, bath]

# Print areas
print(areas)

[11.25, 18.0, 20.0, 10.75, 9.5]
```

```
In [ ]: # area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50

# Adapt list areas
areas = ["hallway", hall, "kitchen", kit, "living room", liv, "bedroom", bed, "bathroom", bath]

# Print areas
print(areas)

['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0, 'bedroom', 10.75, 'bathroom', 9.5]
```

```
In [ ]: # area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50

# house information as a list of lists
house = [["hallway", hall],
          ["kitchen", kit],
          ["living room", liv],
          ["bedroom", bed],
          ["bathroom", bath]]

# Print out house
print(house)

# Print out the type of house
print(type(house))

[['hallway', 11.25], ['kitchen', 18.0], ['living room', 20.0], ['bedroom', 10.75], ['bathroom', 9.5]]
<class 'list'>
```

```
In [ ]: #Subsetting lists

# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]
```

```
# Print out second element from areas
print(areas[1])

# Print out last element from areas
print(areas[-1])

# Print out the area of the living room
print(areas[5])
```

```
11.25
9.5
20.0
```

```
In [ ]: # Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Sum of kitchen and bedroom area: eat_sleep_area
eat_sleep_area = areas[3]+areas[-3]

# Print the variable eat_sleep_area
print(eat_sleep_area)
```

```
28.75
```

```
In [ ]: # Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Use slicing to create downstairs
downstairs = areas[0:6]
# Alternative slicing to create downstairs
downstairs = areas[:6]

# Use slicing to create upstairs
upstairs = areas[6:10]
# Alternative slicing to create upstairs
upstairs = areas[6:]

# Print out downstairs and upstairs
print(downstairs)
print(upstairs)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0]
['bedroom', 10.75, 'bathroom', 9.5]
```

```
In [ ]: #manipulation Lists

# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Correct the bathroom area
```

```
areas[9] = 10.50

# Change "living room" to "chill zone"
areas[4] = "chill zone"

print(areas)

['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom', 10.75, 'bathroom', 10.5]
```

```
In [ ]: # Create the areas list and make some changes
areas = ["hallway", 11.25, "kitchen", 18.0, "chill zone", 20.0,
         "bedroom", 10.75, "bathroom", 10.50]

# Add poolhouse data to areas, new list is areas_1
areas_1 = areas + ["poolhouse", 24.5]

# Add garage data to areas_1, new list is areas_2
areas_2 = areas_1 + ["garage", 15.45]

print(areas_1)
print(areas_2)

['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom', 10.75, 'bathroom', 10.5, 'poolhouse', 24.5]
['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom', 10.75, 'bathroom', 10.5, 'poolhouse', 24.5, 'garage', 15.45]
```

```
In [ ]: # Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Create areas_copy
areas_copy = list(areas) # areas_copy = areas[:]

# Change areas_copy
areas_copy[0] = 5.0

# Print areas
print(areas)
print(areas_copy)

[11.25, 18.0, 20.0, 10.75, 9.5]
[5.0, 18.0, 20.0, 10.75, 9.5]
```

## Chapter 3 - Functions and Packages

```
In [ ]: # Create variables var1 and var2
var1 = [1, 2, 3, 4]
var2 = True

# Print out type of var1
print(type(var1))
```

```
# Print out length of var1
print(len(var1))

# Convert var2 to an integer: out2
out2 = int(var2)
print("true became an integer: ", out2)

<class 'list'>
4
true became an integer: 1
```

```
In [ ]: # Create lists first and second
first = [11.25, 18.0, 20.0]
second = [10.75, 9.50]

# Paste together first and second: full
full = first+second
print(full)

# Sort full in descending order: full_sorted
full_sorted = sorted(full, reverse = True)

# Print out full_sorted
print(full_sorted)
```

[11.25, 18.0, 20.0, 10.75, 9.5]  
[20.0, 18.0, 11.25, 10.75, 9.5]

```
In [ ]: # string to experiment with: place
place = "poolhouse"

# Use upper() on place: place_up
place_up = place.upper()

# Print out place and place_up
print(place)
print(place_up)

# Print out the number of o's in place
print(place.count("o"))
```

poolhouse  
POOLHOUSE  
3

```
In [ ]: # Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0
print(areas.index(20.0))
```

```
# Print out how often 9.50 appears in areas
print(areas.count(9.50))
```

```
2
1
```

```
In [ ]: # Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Use append twice to add poolhouse and garage size
areas.append(24.5)
areas.append(15.45)

# Print out areas
print(areas)

# Reverse the orders of the elements in areas
areas.reverse()

# Print out areas
print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5, 24.5, 15.45]
[15.45, 24.5, 9.5, 10.75, 20.0, 18.0, 11.25]
```

```
In [ ]: # Import the math package
import math

# Definition of radius
r = 0.43

# Calculate C
C = 2*math.pi*r

# Calculate A
A = math.pi*pow(r,2)

# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

```
Circumference: 2.701769682087222
Area: 0.5808804816487527
```

```
In [ ]: # Import radians function of math package
from math import radians

# Definition of radius
r = 192500

# Travel angle of the Moon over 12 degrees
```

```
angle_degrees = 12

# Convert angle to radians using radians() function
angle_radians = radians(angle_degrees)

# Calculate the travel distance using the formula: distance = radius * angle
dist = r * angle_radians

# Print out dist
print(dist)
```

```
40317.10572106901
```

## Chapter 4 - NumPy

The MLB also offers to let you analyze their weight data. Again, both are available as regular Python lists: height\_in and weight\_lb. height\_in is in inches and weight\_lb is in pounds.

It's now possible to calculate the BMI of each baseball player. Python code to convert height\_in to a numpy array with the correct units is already available in the workspace. Follow the instructions step by step and finish the game! height\_in and weight\_lb are available as regular lists.

```
In [ ]: import pandas as pd
mlb = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Da
height_in = mlb['Height'].tolist()
weight_lb = mlb['Weight'].tolist()
import numpy as np
```

```
In [ ]: # Import the numpy package as np
import numpy as np

# Create list baseball
baseball = [180, 215, 210, 210, 188, 176, 209, 200]

# Create a numpy array from baseball: np_baseball
np_baseball = np.array(baseball)

# Print out type of np_baseball
print(type(np_baseball))

<class 'numpy.ndarray'>
```

```
In [ ]: # Import numpy
import numpy as np

# Create array from height_in with metric units: np_height_m
np_height_m = np.array(height_in) * 0.0254
```

```
# Create array from weight_lb with metric units: np_weight_kg
np_weight_kg = np.array(weight_lb) * 0.453592

# Calculate the BMI: bmi
bmi = np_weight_kg / np_height_m ** 2

# Print out bmi
print(bmi)
```

```
[23.11037639 27.60406069 28.48080465 ... 25.62295933 23.74810865
 25.72686361]
```

```
In [ ]: # Import numpy
import numpy as np
```

```
# Calculate the BMI: bmi
np_height_m = np.array(height_in) * 0.0254
np_weight_kg = np.array(weight_lb) * 0.453592
bmi = np_weight_kg / np_height_m ** 2
```

```
# Create the light array
light = bmi < 21
```

```
# Print out light
print(light)
```

```
# Print out BMIs of all baseball players whose BMI is below 21
print(bmi[light])
```

```
[False False False ... False False False]
[20.54255679 20.54255679 20.69282047 20.69282047 20.34343189 20.34343189
 20.69282047 20.15883472 19.4984471 20.69282047 20.9205219 ]
```

```
In [ ]: #Subsetting NumPy Arrays
```

```
# Import numpy
import numpy as np

# Store weight and height Lists as numpy arrays
np_weight_lb = np.array(weight_lb)
np_height_in = np.array(height_in)
```

```
# Print out the weight at index 50
print(np_weight_lb[50])
```

```
# Print out sub-array of np_height_in: index 100 up to and including index 110
print(np_height_in[100:111])
```

```
200
[73 74 72 73 69 72 73 75 75 73 72]
```

## Your First 2D NumPy Array

```
In [ ]: # Import numpy
import numpy as np

# Create baseball, a list of lists
baseball = [[180, 78.4],
            [215, 102.7],
            [210, 98.5],
            [188, 75.2]]

# Create a 2D numpy array from baseball: np_baseball
np_baseball = np.array(baseball)

# Print out the type of np_baseball
print(type(np_baseball))

# Print out the shape of np_baseball
print(np_baseball.shape)
```

```
<class 'numpy.ndarray'>
(4, 2)
```

## Subsetting 2D NumPy Arrays

If your 2D numpy array has a regular structure, i.e. each row and column has a fixed number of values, complicated ways of subsetting become very easy. Have a look at the code below where the elements "a" and "c" are extracted from a list of lists.

For regular Python lists, this is a real pain. For 2D numpy arrays, however, it's pretty intuitive! The indexes before the comma refer to the rows, while those after the comma refer to the columns. The : is for slicing; in this example, it tells Python to include all rows.

```
In [ ]: #Example how to extract the first element from each nested list in a list

# regular list of lists
x = [["a", "b"], ["c", "d"]]
[x[0][0], x[1][0]]

# numpy
import numpy as np
np_x = np.array(x)
np_x[:, 0]
```

```
Out[ ]: array(['a', 'c'], dtype='<U1')
```

```
In [ ]: np_baseball
```

```
Out[ ]: array([[180. ,  78.4],  
   [215. , 102.7],  
   [210. ,  98.5],  
   [188. ,  75.2]])
```

```
In [ ]: baseball = [[74, 180], [74, 215], [72, 210], [72, 210], [73, 188], [69, 176], [69, 209], [71, 200], [76, 231], [71, 180], [73, 188]]
```

```
In [ ]: # Create np_baseball (2 cols)  
np_baseball = np.array(baseball)  
  
# Print out the 50th row of np_baseball  
print(np_baseball[49,:])  
  
# Select the entire second column of np_baseball: np_weight_lb  
np_weight_lb=np_baseball[:,1]  
  
# Print out height of 124th player  
print(np_baseball[123,0])
```

```
[ 70 195]  
75
```

## 2D Arithmetic

```
In [ ]: # another example  
np_mat = np.array([[1, 2],  
                  [3, 4],  
                  [5, 6]])  
np_mat * 2  
np_mat + np.array([10, 10])  
np_mat + np_mat
```

```
Out[ ]: array([[ 2,  4],  
   [ 6,  8],  
   [10, 12]])
```

```
In [ ]: # Store weight and height lists as numpy arrays  
np_weight_lb = np.array(weight_lb)  
np_height_in = np.array(height_in)  
  
# Print out the weight at index 50  
  
print(np_weight_lb[50])  
# Print out sub-array of np_height: index 100 up to and including index 110  
print(np_height_in[100:111])
```

```
200  
[73 74 72 73 69 72 73 75 75 73 72]
```

## Average versus median

You now know how to use numpy functions to get a better feeling for your data. It basically comes down to importing numpy and then calling several simple functions on the numpy arrays:

```
import numpy as np

x = [1, 4, 8, 10, 12]

np.mean(x)

np.median(x)
```

The baseball data is available as a 2D numpy array with 3 columns (height, weight, age) and 200 rows. The name of this numpy array is np\_baseball. After restructuring the data, however, you notice that some height values are abnormally high. Follow the instructions and discover which summary statistic is best suited if you're dealing with so-called outliers.

```
In [ ]: avg = np.mean(np_baseball[:,0])
print("Average: " + str(avg))

# Print median height. Replace 'None'
med = np.median(np_baseball[:,0])
print("Median: " + str(med))

# Print out the standard deviation on height. Replace 'None'
stddev = np.std(np_baseball[:,0])
print("Standard Deviation: " + str(stddev))

# Print out correlation between first and second column. Replace 'None'
corr = np.corrcoef(np_baseball[:,0], np_baseball[:,1])
print("Correlation: " + str(corr))
```

```
Average: 73.83
Median: 74.0
Standard Deviation: 2.2893448844593074
Correlation: [[1.          0.55676088]
               [0.55676088 1.        ]]
```

## Explore the baseball data

Because the mean and median are so far apart, you decide to complain to the MLB. They find the error and send the corrected data over to you. It's again available as a 2D Numpy array np\_baseball, with three columns.

The Python script on the right already includes code to print out informative messages with the different summary statistics. Can you finish the job?

```
In [ ]: # Create np_height_in from np_baseball
np_height_in=np_baseball[:,0]

# Print out the mean of np_height_in
print(np.mean(np_height_in))

# Print out the median of np_height_in
print(np.median(np_height_in))
```

73.83

74.0

# Chapter 1 - Matplotlib

Data Visualization is a key skill for aspiring data scientists. Matplotlib makes it easy to create meaningful and insightful plots. In this chapter, you will learn to build various types of plots and to customize them to make them more visually appealing and interpretable.

## Line plot (1)

With matplotlib, you can create a bunch of different plots in Python. The most basic plot is the line plot. A general recipe is given here.

```
import matplotlib.pyplot as plt
plt.plot(x,y)
plt.show()
```

In the video, you already saw how much the world population has grown over the past years. Will it continue to do so? The world bank has estimates of the world population for the years 1950 up to 2100. The years are loaded in your workspace as a list called year, and the corresponding populations as a list called pop

```
In [ ]: year = [x for x in range(1950, 2101)]
print(year)

[1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 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, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100]

In [ ]: pop = [2.53, 2.57, 2.62, 2.67, 2.71, 2.76, 2.81, 2.86, 2.92, 2.97, 3.03, 3.08, 3.14, 3.2, 3.26, 3.33, 3.4, 3.47, 3.54, 3.62, 3.69, ...]

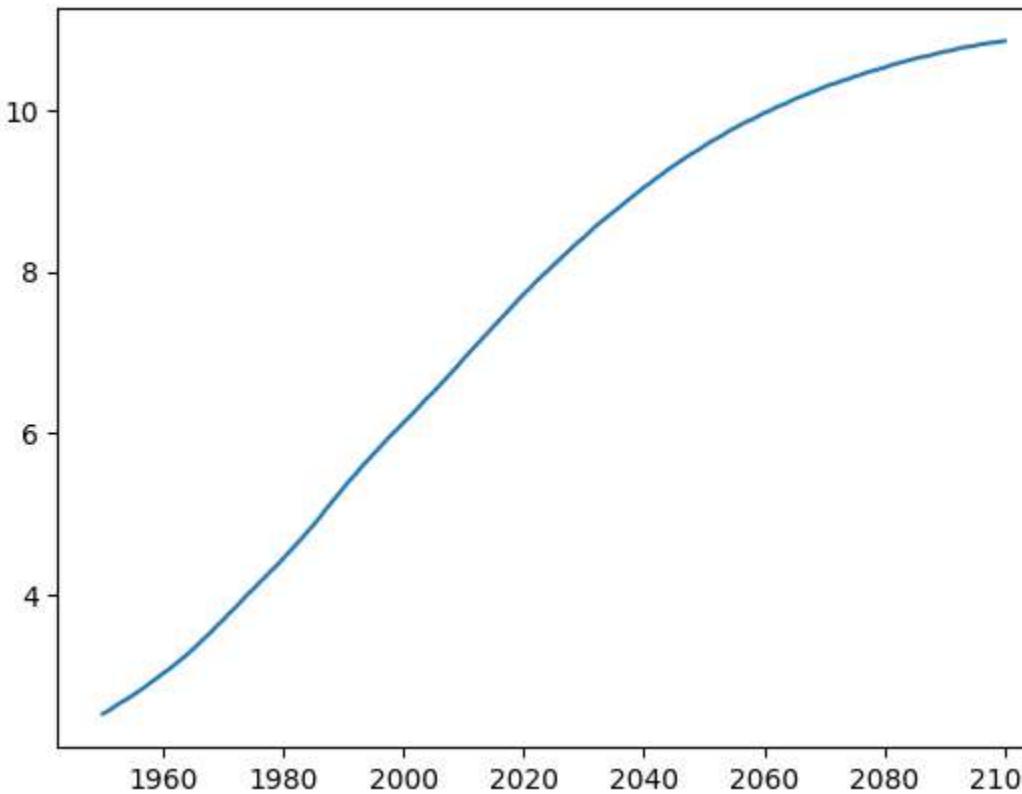
In [ ]: # Print the last item from year and pop
print(year[-1])
print(pop[-1])

# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt

# Make a Line plot: year on the x-axis, pop on the y-axis
plt.plot(year, pop)
```

```
# Display the plot with plt.show()
plt.show() # no need to use plt.show() in jupyter notebook
```

```
2100  
10.85
```



## Line plot (2)

Now that you've built your first line plot, let's start working on the data that professor Hans Rosling used to build his beautiful bubble chart. It was collected in 2007. Two lists are available for you:

- `life_exp` which contains the life expectancy for each country and
- `gdp_cap`, which contains the GDP per capita (i.e. per person) for each country expressed in US Dollars.

GDP stands for Gross Domestic Product. It basically represents the size of the economy of a country. Divide this by the population and you get the GDP per capita.

```
In [ ]: gdp_cap = [974.5803384, 5937.029525999998, 6223.367465, 4797.231267, 12779.37964, 34435.367439999995, 36126.4927, 29796.04834, 139
```

```
In [ ]: life_exp = [43.828, 76.423, 72.301, 42.731, 75.32, 81.235, 79.829, 75.635, 64.062, 79.441, 56.728, 65.554, 74.852, 50.728, 72.39,
```

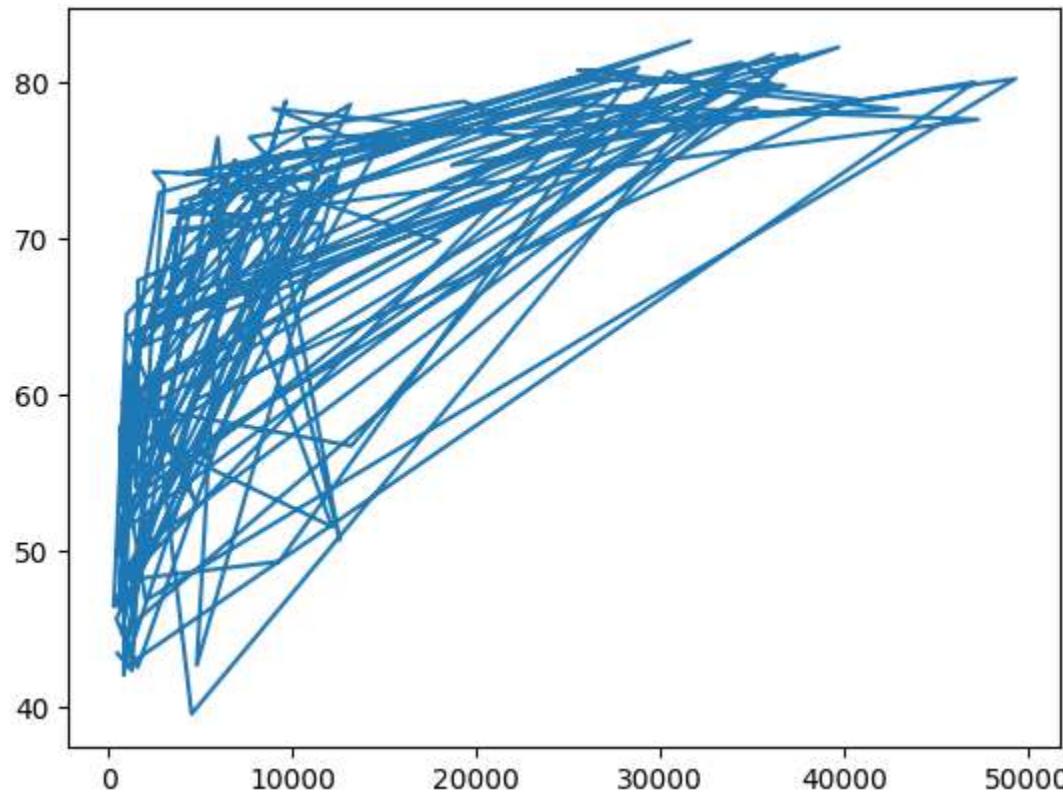
```
In [ ]: # Print the last item of gdp_cap and life_exp
print(gdp_cap[-1])
print(life_exp[-1])
```

```
# Make a Line plot, gdp_cap on the x-axis, life_exp on the y-axis
plt.plot(gdp_cap, life_exp)
```

```
469.70929810000007
```

```
43.487
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1b755ce3830>]
```



Well done, but this doesn't look right. Let's build a plot that makes more sense.

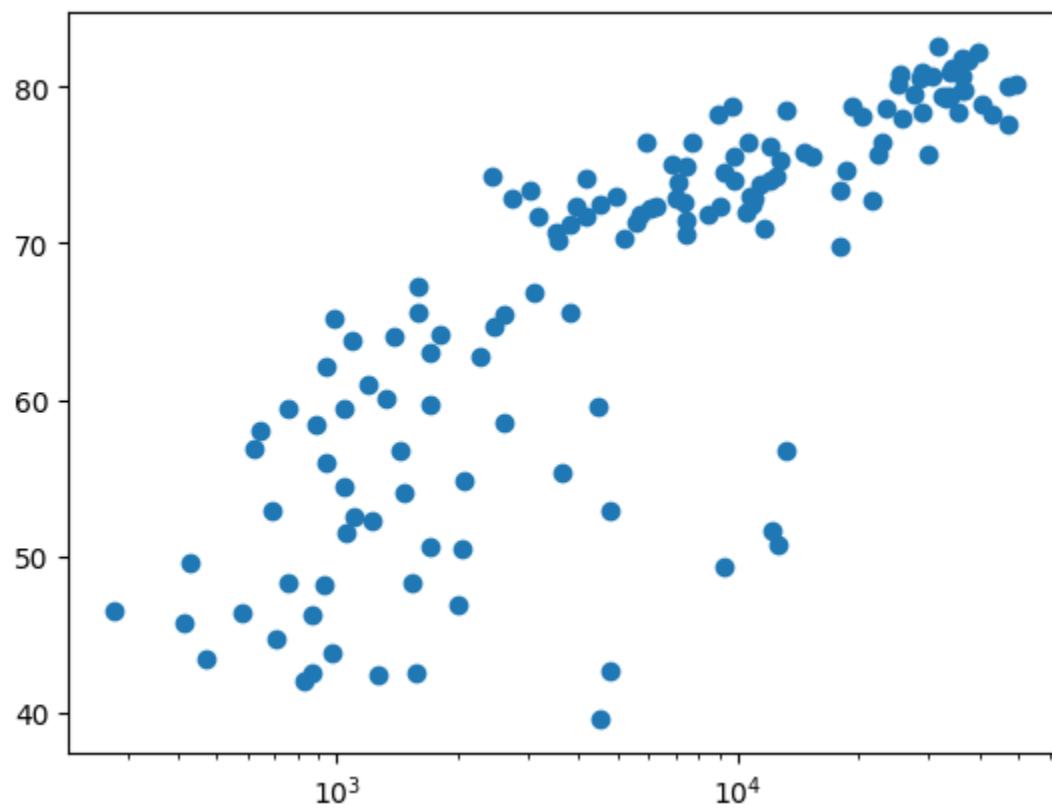
## Scatter Plot (1)

When you have a time scale along the horizontal axis, the line plot is your friend. But in many other cases, when you're trying to assess if there's a correlation between two variables, for example, the scatter plot is the better choice. Below is an example of how to build a scatter plot.

```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.show()
```

Let's continue with the gdp\_cap versus life\_exp plot, the GDP and life expectancy data for different countries in 2007. Maybe a scatter plot will be a better alternative?

```
In [ ]: # Change the line plot below to a scatter plot  
plt.scatter(gdp_cap, life_exp)  
  
# Put the x-axis on a logarithmic scale  
plt.xscale('log')
```



Great! That looks much better!

## Scatter plot (2)

In the previous exercise, you saw that that the higher GDP usually corresponds to a higher life expectancy. In other words, there is a positive correlation.

Do you think there's a relationship between population and life expectancy of a country? The list life\_exp from the previous exercise is already available. In addition, now also pop\_2007 is available, listing the corresponding populations for the countries in 2007. The populations are in millions of people.

```
In [ ]: len(life_exp)
```

```
Out[ ]: 142
```

```
In [ ]: pop_2007 = [31.889923, 3.600523, 33.333216, 12.420476, 40.301927, 20.434176, 8.199783, 0.708573, 150.448339, 10.392226, 8.078314,
```

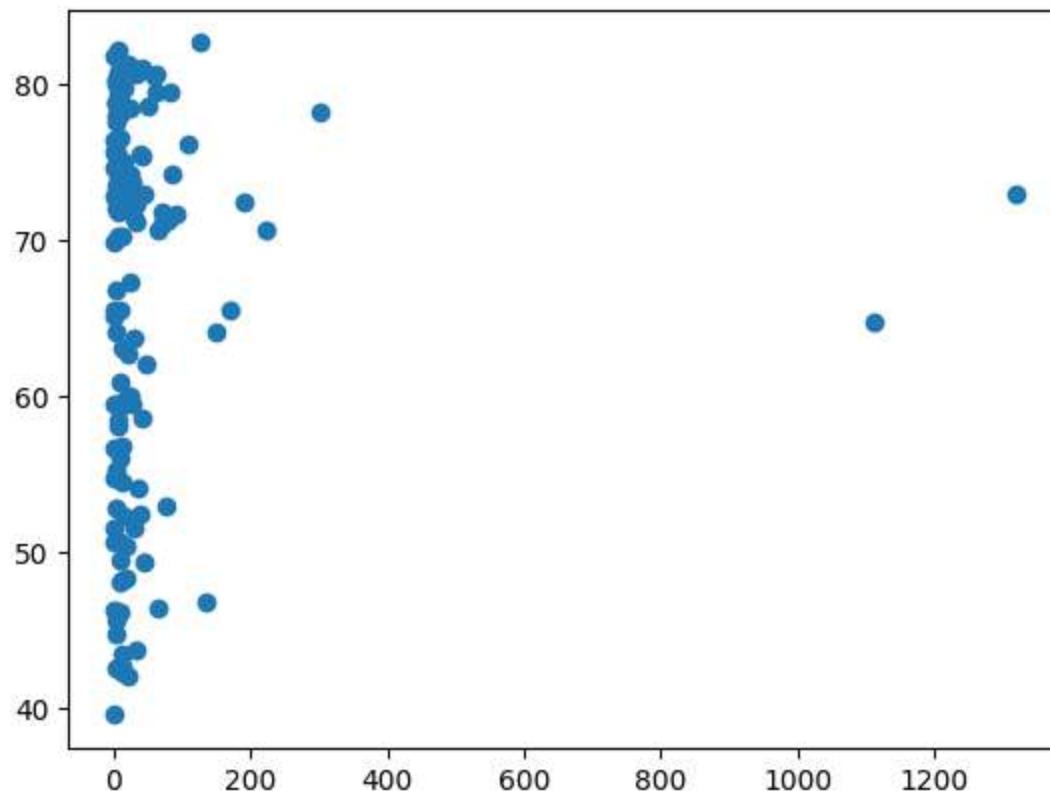
```
In [ ]: len(pop_2007)
```

```
Out[ ]: 142
```

```
In [ ]: # Build Scatter plot
```

```
plt.scatter(pop_2007, life_exp)
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x1b759069220>
```



Nice! There's no clear relationship between population and life expectancy, which makes perfect sense.

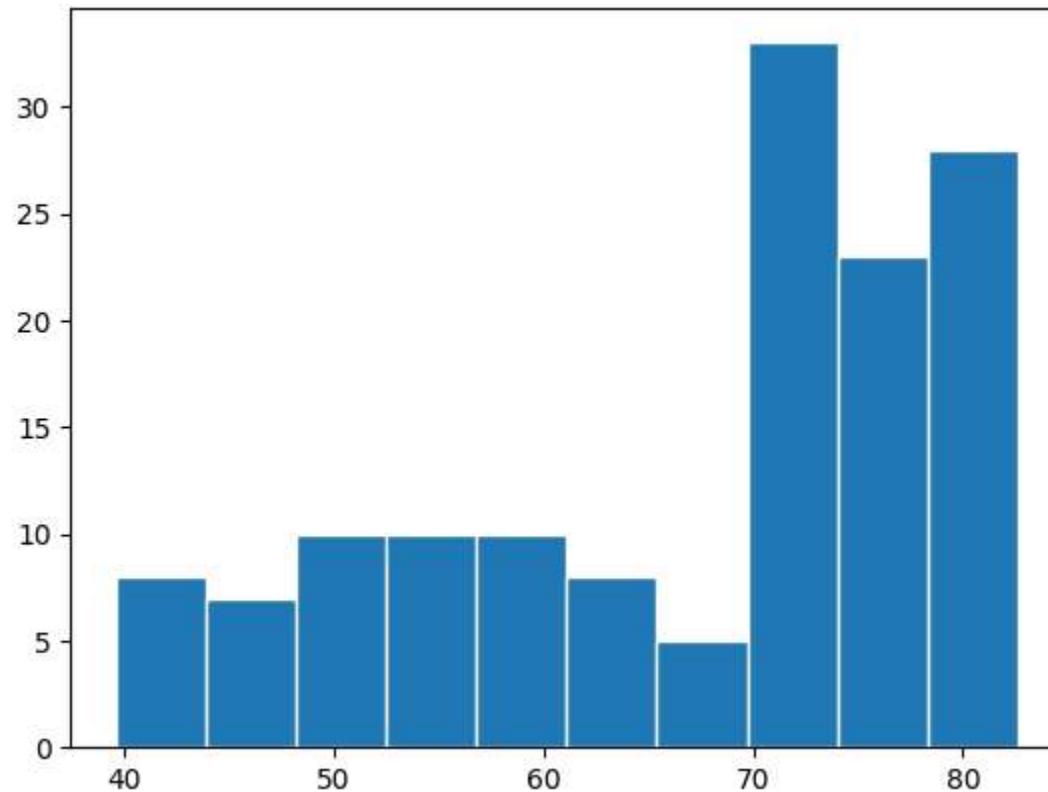
## Build a histogram (1)

life\_exp, the list containing data on the life expectancy for different countries in 2007, is available

To see how life expectancy in different countries is distributed, let's create a histogram of life\_exp.

```
In [ ]: # Create histogram of life_exp data  
plt.hist(life_exp, ec='white')
```

```
Out[ ]: (array([ 8.,  7., 10., 10.,  8.,  5., 33., 23., 28.]),  
 array([39.613, 43.912, 48.211, 52.51 , 56.809, 61.108, 65.407, 69.706,  
       74.005, 78.304, 82.603]),  
<BarContainer object of 10 artists>)
```

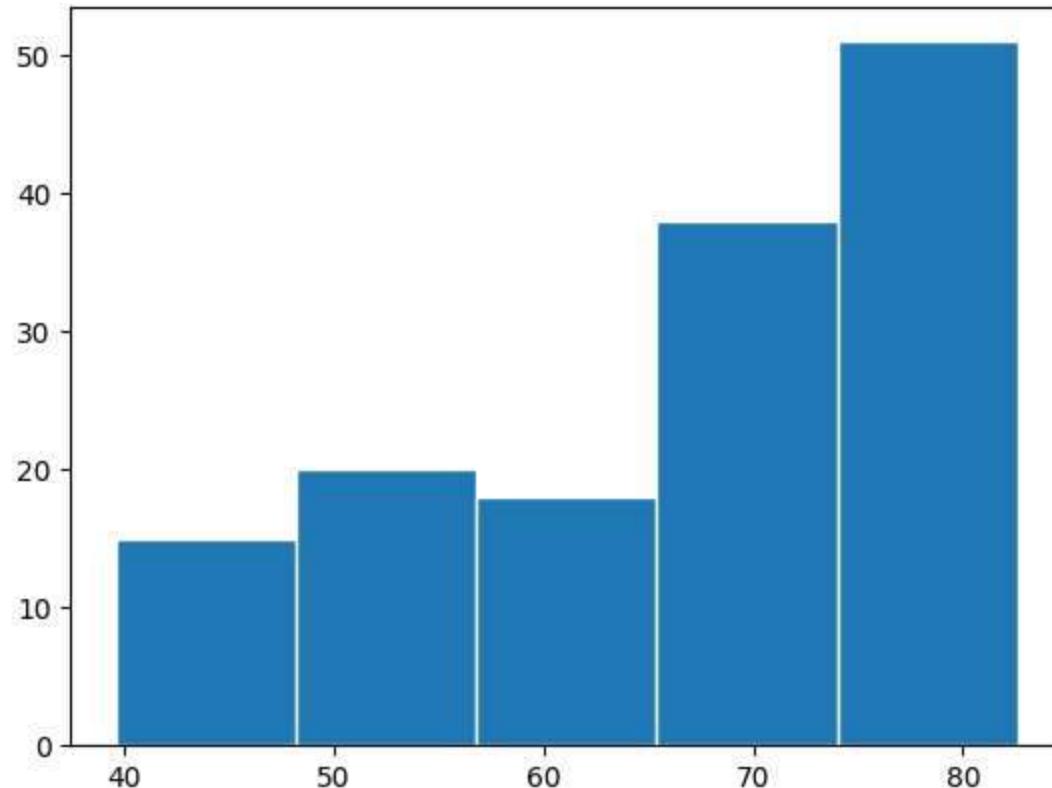


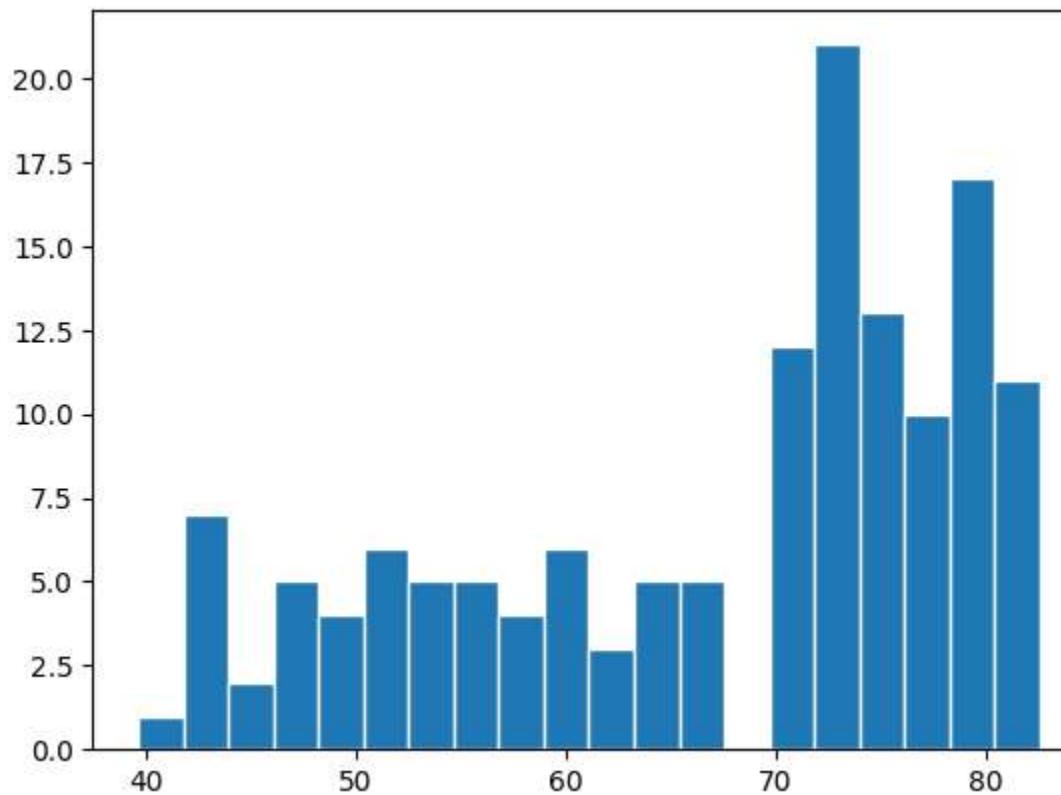
## Build a histogram (2): bins

In the previous exercise, you didn't specify the number of bins. By default, Python sets the number of bins to 10 in that case. The number of bins is pretty important. Too few bins will oversimplify reality and won't show you the details. Too many bins will overcomplicate reality and won't show the bigger picture.

To control the number of bins to divide your data in, you can set the bins argument.

```
In [ ]: # Build histogram with 5 bins  
plt.hist(life_exp, bins = 5, ec='white')  
plt.show()  
  
# Build histogram with 20 bins  
plt.hist(life_exp, bins = 20, ec='white')  
plt.show()
```





### Build a histogram (3): compare

In the video, you saw population pyramids for the present day and for the future. Because we were using a histogram, it was very easy to make a comparison.

Let's do a similar comparison. `life_exp` contains life expectancy data for different countries in 2007. You also have access to a second list now, `life_exp1950`, containing similar data for 1950. Can you make a histogram for both datasets?

You'll again be making two plots. The `plt.show()` and `plt.clf()` commands to render everything nicely are already included.

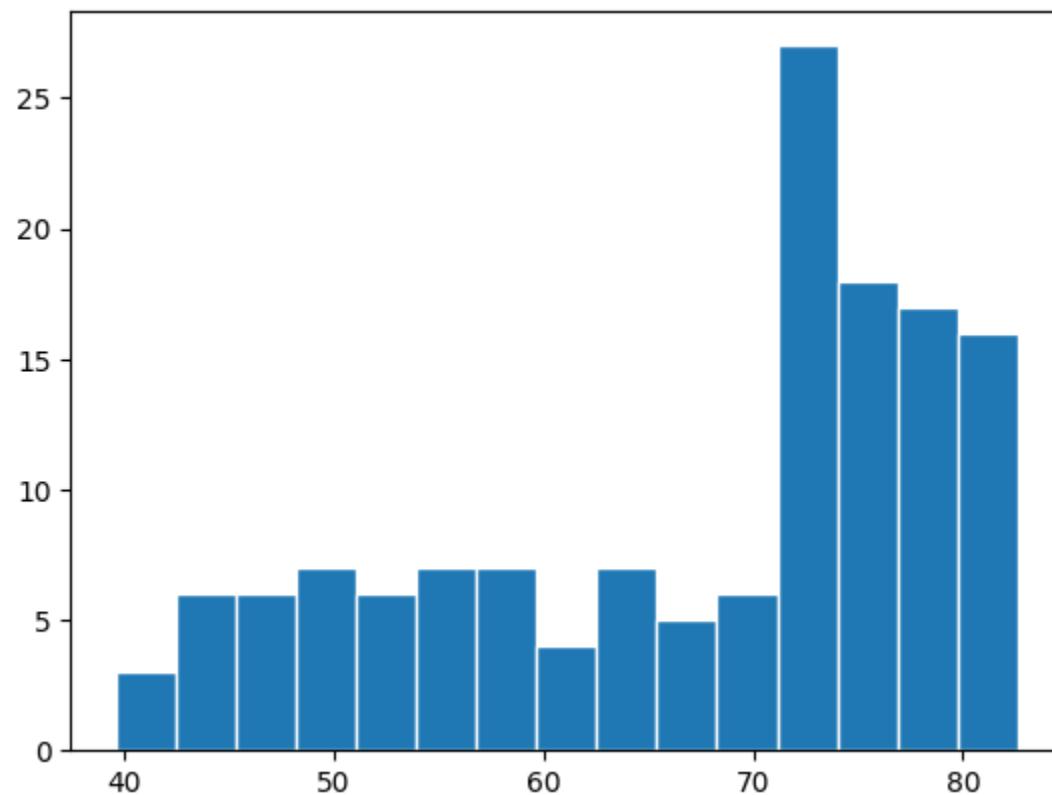
```
In [ ]: life_exp1950 = [28.8, 55.23, 43.08, 30.02, 62.48, 69.12, 66.8, 50.94, 37.48, 68.0, 38.22, 40.41, 53.82, 47.62, 50.92, 59.6, 31.98,
```

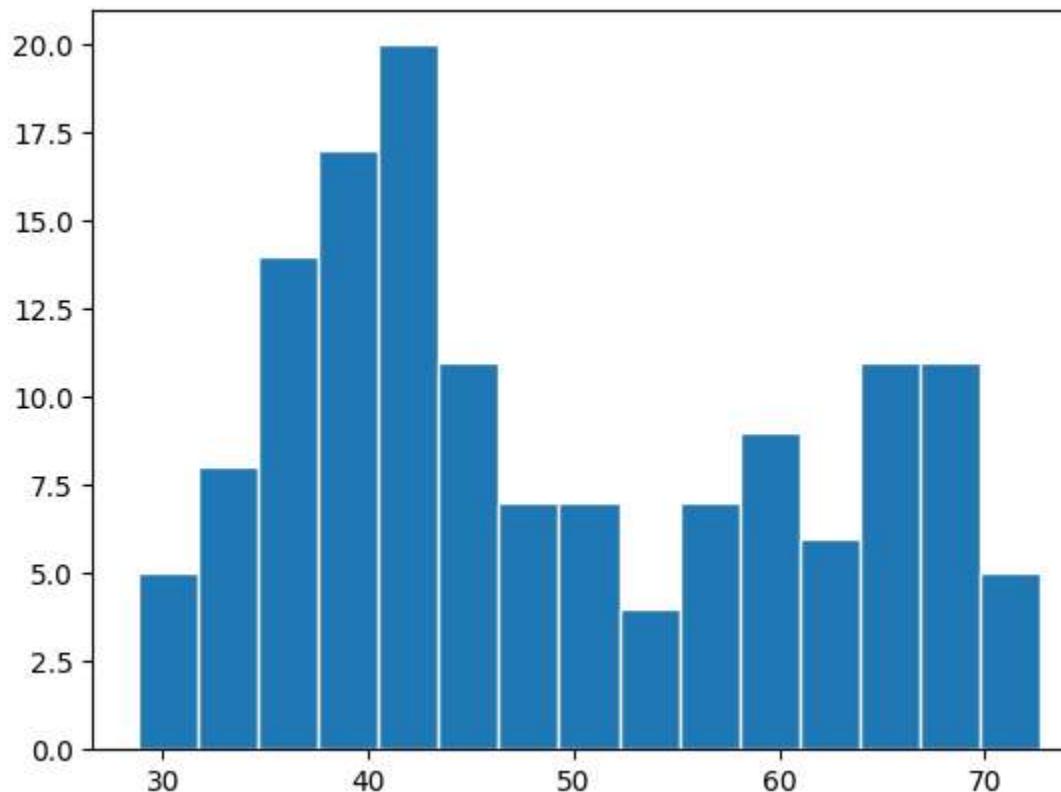
```
In [ ]: # Histogram of life_exp, 15 bins
plt.hist(life_exp, bins = 15, ec='white')

# Show and clear plot
plt.show()
plt.clf()

# Histogram of life_exp1950, 15 bins
```

```
plt.hist(life_exp1950, bins = 15, ec='white')
# Show and clear plot again
plt.show()
plt.clf()
```





<Figure size 640x480 with 0 Axes>

## Labels

It's time to customize your own plot. This is the fun part, you will see your plot come to life!

You're going to work on the scatter plot with world development data: GDP per capita on the x-axis (logarithmic scale), life expectancy on the y-axis.

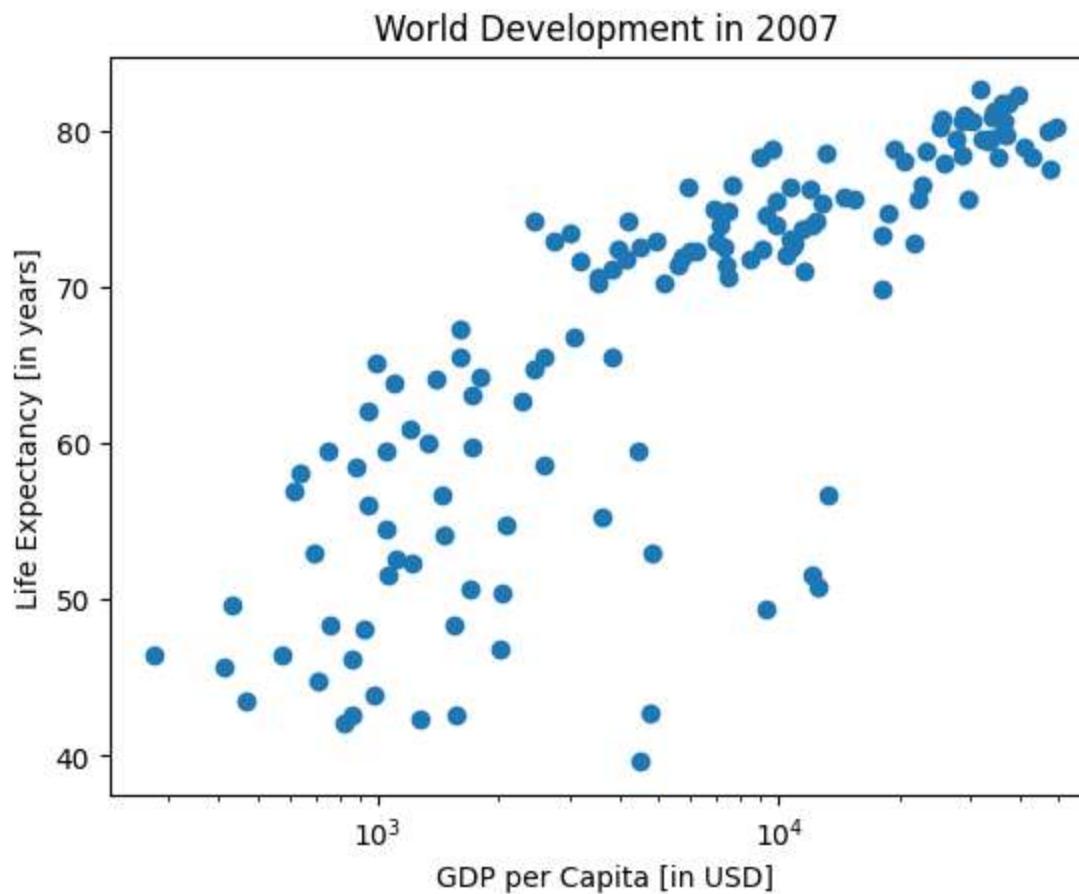
```
In [ ]: # Basic scatter plot, log scale
plt.scatter(gdp_cap, life_exp)
plt.xscale('log')

# Strings
xlab = 'GDP per Capita [in USD]'
ylab = 'Life Expectancy [in years]'
title = 'World Development in 2007'

# Add axis labels
plt.xlabel(xlab)
plt.ylabel(ylab)
```

```
# Add title  
plt.title(title)
```

```
Out[ ]: Text(0.5, 1.0, 'World Development in 2007')
```



## Ticks

You could control the y-ticks by specifying two arguments:

```
plt.yticks([0,1,2], ["one","two","three"])
```

In this example, the ticks corresponding to the numbers 0, 1 and 2 will be replaced by one, two and three, respectively.

Let's do a similar thing for the x-axis of your world development chart, with the `xticks()` function. The tick values 1000, 10000 and 100000 should be replaced by 1k, 10k and 100k.

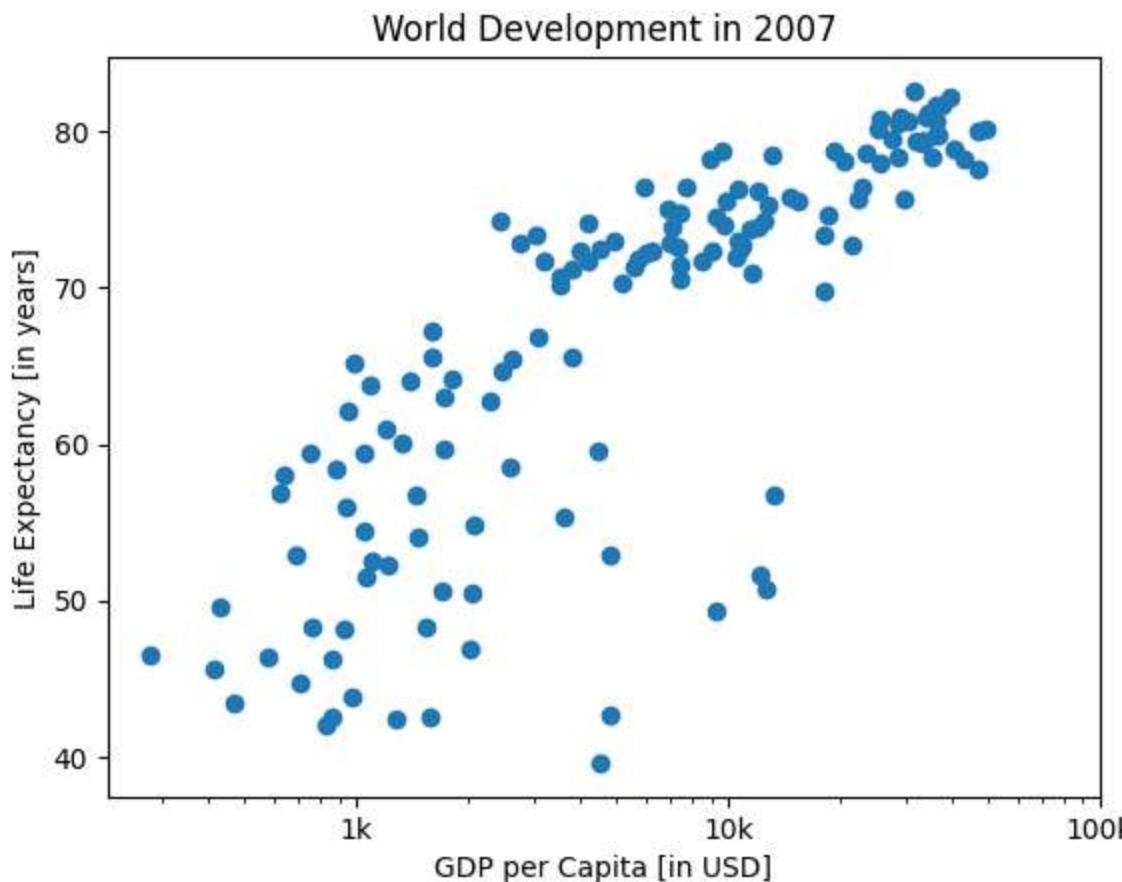
```
In [ ]: # Scatter plot  
plt.scatter(gdp_cap, life_exp)
```

```
# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')

# Definition of tick_val and tick_lab
tick_val = [1000, 10000, 100000]
tick_lab = ['1k', '10k', '100k']

# Adapt the ticks on the x-axis
plt.xticks(tick_val, tick_lab)
```

```
Out[ ]: ([<matplotlib.axis.XTick at 0x1b7591128a0>,
 <matplotlib.axis.XTick at 0x1b75947b740>,
 <matplotlib.axis.XTick at 0x1b75947a960>],
 [Text(1000, 0, '1k'), Text(10000, 0, '10k'), Text(100000, 0, '100k')])
```



Great! Your plot is shaping up nicely!

## Sizes

Right now, the scatter plot is just a cloud of blue dots, indistinguishable from each other. Let's change this. Wouldn't it be nice if the size of the dots corresponds to the population?

In [ ]:

```
# Import numpy as np
import numpy as np
```

```
# Store pop_2017 as a numpy array: np_pop
np_pop = np.array(pop_2007)
```

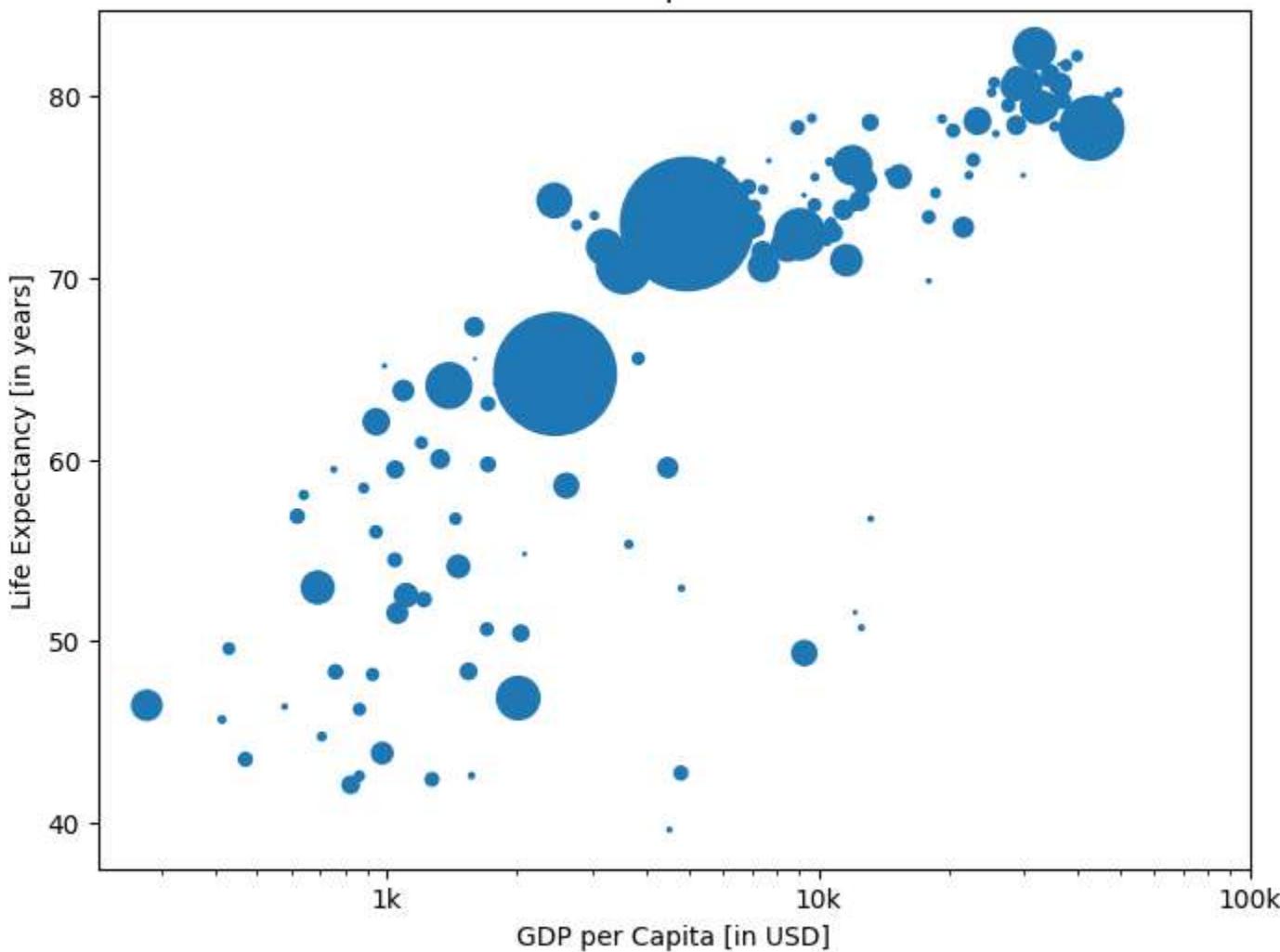
```
# Double np_pop
np_pop = np_pop * 2
```

```
# Update: set s argument to np_pop
plt.figure(figsize = (8, 6))
plt.scatter(gdp_cap, life_exp, s = np_pop)
```

```
# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000, 10000, 100000],['1k', '10k', '100k'])
```

```
# Display the plot
plt.show()
```

## World Development in 2007



Bellissimo! Can you already tell which bubbles correspond to which countries?

### Colors

The next step is making the plot more colorful! To do this, a list col has been created for you. It's a list with a color for each corresponding country, depending on the continent the country is part of.

How did we make the list col you ask? The Gapminder data contains a list continent with the continent each country belongs to. A dictionary is constructed that maps continents onto colors:

```
dict = {
    'Asia': 'red',
    'Europe': 'green',
```

```
'Africa':'blue',
'Americas':'yellow',
'Oceania':'black'
}
```

Nothing to worry about now; you will learn about dictionaries in the next chapter.

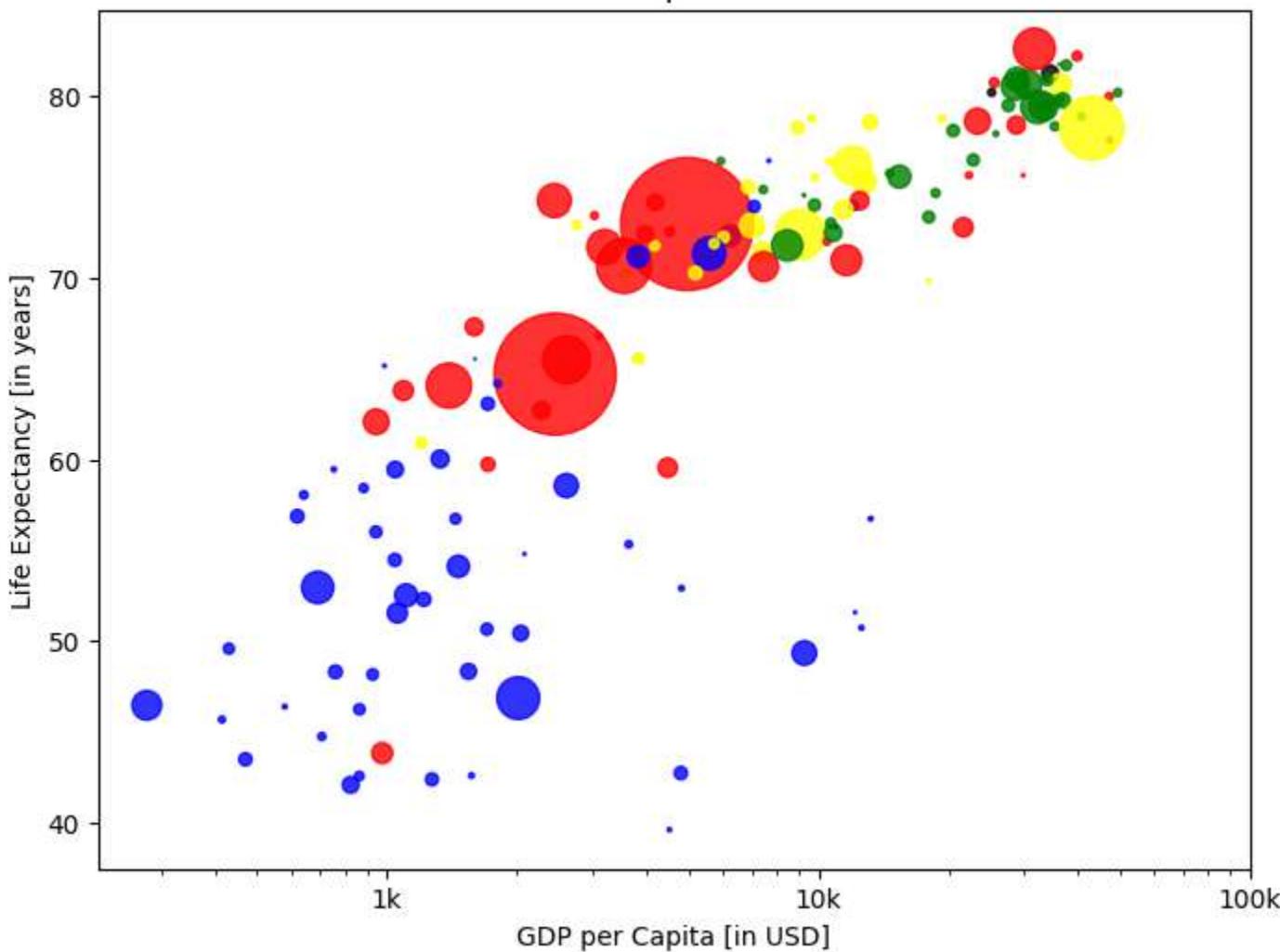
```
In [ ]: col = ['red', 'green', 'blue', 'blue', 'yellow', 'black', 'green', 'red', 'red', 'green', 'blue', 'yellow', 'green', 'blue', 'yellow']
```

```
In [ ]: # Specify c and alpha inside plt.scatter()
plt.figure(figsize = (8, 6))
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop_2007) * 2, c = col, alpha = 0.8)

# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])
```

```
Out[ ]: ([<matplotlib.axis.XTick at 0x1b7594328a0>,
 <matplotlib.axis.XTick at 0x1b758fa3dd0>,
 <matplotlib.axis.XTick at 0x1b755da54f0>],
 [Text(1000, 0, '1k'), Text(10000, 0, '10k'), Text(100000, 0, '100k')])
```

## World Development in 2007



Nice! This is looking more and more like Hans Rosling's plot!

### Additional Customizations

If you have another look at the script, under # Additional Customizations, you'll see that there are two plt.text() functions now. They add the words "India" and "China" in the plot

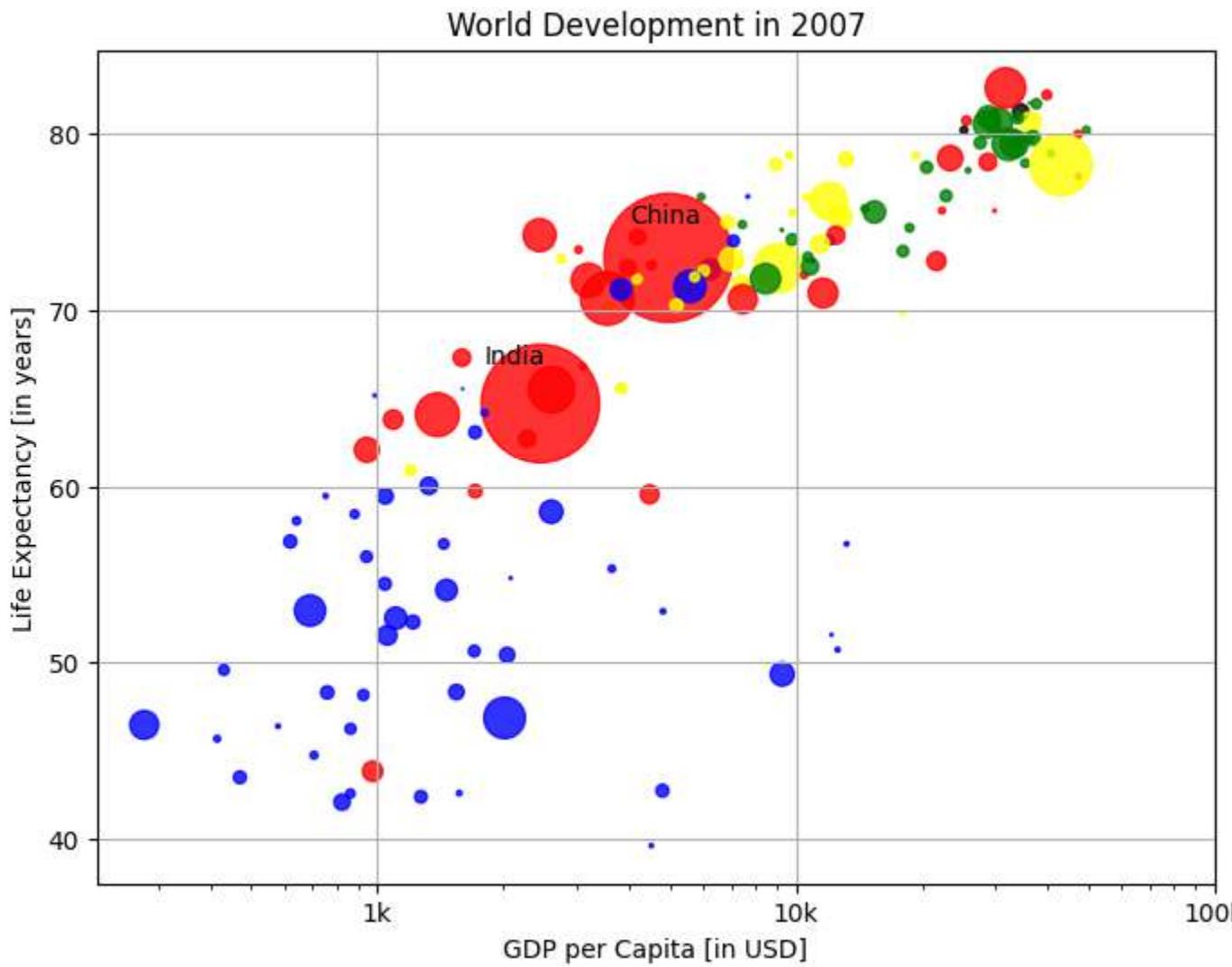
```
In [ ]: # Scatter plot
plt.figure(figsize = (8, 6))
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop_2007) * 2, c = col, alpha = 0.8)

# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
```

```
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Additional customizations
plt.text(1800, 67, 'India')
plt.text(4000, 75, 'China')

# Add grid() call
plt.grid(True)
```



Beautiful! A visualization only makes sense if you can interpret it properly. Let's do that in the next exercise.

## Interpretation

If you have a look at your colorful plot, it's clear that people live longer in countries with a higher GDP per capita. No high income countries have really short life expectancy, and no low income countries have very long life expectancy. Still, there is a huge difference in life expectancy between countries on the same income level. Most people live in middle income countries where difference in lifespan is huge between countries; depending on how income is distributed and how it is used.

## What can you say about the plot?

The countries in blue, corresponding to Africa, have both low life expectancy and a low GDP per capita.

Correct! Up to the next chapter, on dictionaries!

# Chapter 2 - Dictionaries & Pandas

Learn about the dictionary, an alternative to the Python list, and the Pandas DataFrame, the de facto standard to work with tabular data in Python. You will get hands-on practice with creating, manipulating and accessing the information you need from these data structures.

## Motivation for dictionaries

To see why dictionaries are useful, have a look at the two lists defined below. `countries` contains the names of some European countries. `capitals` lists the corresponding names of their capital.

```
In [ ]: # Definition of countries and capital
countries = ['spain', 'france', 'germany', 'norway']
capitals = ['madrid', 'paris', 'berlin', 'oslo']

# Get index of 'germany': ind_ger
ind_ger = countries.index('germany')

# Use ind_ger to print out capital of Germany
print(capitals[ind_ger])
```

berlin

it's not very convenient. Head over to the next exercise to create a dictionary of this data.

## Create dictionary

The countries and capitals lists are again available. It's your job to convert this data to a dictionary where the country names are the keys and the capitals are the corresponding values. As a refresher, here is a recipe for creating a dictionary:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
}
```

In this recipe, both the keys and the values are strings. This will also be the case for this exercise.

```
In [ ]: # From string in countries and capitals, create dictionary europe  
europe = { 'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo' }  
  
# Print europe  
print(europe)  
  
{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo'}
```

Great! Now that you've built your first dictionaries, let's get serious!

## Access dictionary

If the keys of a dictionary are chosen wisely, accessing the values in a dictionary is easy and intuitive. For example, to get the capital for France from europe you can use:

```
europe['france']
```

Here, 'france' is the key and 'paris' the value is returned.

```
In [ ]: # Print out the keys in europe  
print(europe.keys())  
  
# Print out value that belongs to key 'norway'  
print(europe['norway'])  
  
dict_keys(['spain', 'france', 'germany', 'norway'])  
oslo
```

Good job, now you're warmed up for some more.

## Dictionary Manipulation (1)

If you know how to access a dictionary, you can also assign a new value to it. To add a new key-value pair to europe you can use something like this:

```
europe['iceland'] = 'reykjavik'
```

```
In [ ]: # Add italy to europe  
europe['italy'] = 'rome'
```

```

# Print out italy in europe
print('italy' in europe)
# Add poland to europe
europe['poland'] = 'warsaw'

# Print europe
print(europe)

```

```

True
{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo', 'italy': 'rome', 'poland': 'warsaw'}

```

Well done! Europe is growing by the minute! Did you notice that the order of the printout is not the same as the order in the dictionary's definition? That's because dictionaries are inherently unordered.

## Dictionary Manipulation (2)

Somebody thought it would be funny to mess with your accurately generated dictionary. An adapted version of the europe dictionary is available

Can you clean up? Do not do this by adapting the definition of europe, but by adding Python commands to update and remove key:value pairs.

```

In [ ]:
# Definition of dictionary
europe = {'spain':'madrid', 'france':'paris', 'germany':'bonn',
          'norway':'oslo', 'italy':'rome', 'poland':'warsaw',
          'australia':'vienna' }

# Update capital of germany
europe['germany'] = 'berlin'

# Remove australia
del(europe['australia'])

# Print europe
print(europe)

```

```

{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo', 'italy': 'rome', 'poland': 'warsaw'}

```

Great job! That's much better!

## Dictionarception

Remember lists? They could contain anything, even other lists. Well, for dictionaries the same holds. Dictionaries can contain key:value pairs where the values are again dictionaries.

As an example, have a look at the script where another version of europe - the dictionary you've been working with all along - is coded. The keys are still the country names, but the values are dictionaries that contain more information than just the capital.

It's perfectly possible to chain square brackets to select elements. To fetch the population for Spain from europe, for example, you need:

```
europe['spain']['population']
```

```
In [ ]: # Dictionary of dictionaries
europe = { 'spain': { 'capital':'madrid', 'population':46.77 },
           'france': { 'capital':'paris', 'population':66.03 },
           'germany': { 'capital':'berlin', 'population':80.62 },
           'norway': { 'capital':'oslo', 'population':5.084 } }

# Print out the capital of France
print(europe['france']['capital'])

# Create sub-dictionary data
data = {'capital':'rome', 'population':59.83}

# Add data to europe under key 'italy'
europe['italy'] = data

# Print europe
print(europe)

paris
{'spain': {'capital': 'madrid', 'population': 46.77}, 'france': {'capital': 'paris', 'population': 66.03}, 'germany': {'capital': 'berlin', 'population': 80.62}, 'norway': {'capital': 'oslo', 'population': 5.084}, 'italy': {'capital': 'rome', 'population': 59.83}}
```

Great! It's time to learn about a new data structure!

## Dictionary to DataFrame (1)

Pandas is an open source library, providing high-performance, easy-to-use data structures and data analysis tools for Python. Sounds promising!

The DataFrame is one of Pandas' most important data structures. It's basically a way to store tabular data where you can label the rows and the columns. One way to build a DataFrame is from a dictionary.

In the exercises that follow you will be working with vehicle data from different countries. Each observation corresponds to a country and the columns give information about the number of vehicles per capita, whether people drive left or right, and so on.

Three lists are defined in the script:

- names, containing the country names for which data is available.
- dr, a list with booleans that tells whether people drive left or right in the corresponding country.
- cpc, the number of motor vehicles per 1000 people in the corresponding country.

Each dictionary key is a column label and each value is a list which contains the column elements.

In [ ]:

```
# Pre-defined lists
names = ['United States', 'Australia', 'Japan', 'India', 'Russia', 'Morocco', 'Egypt']
dr = [True, False, False, False, True, True, True]
cpc = [809, 731, 588, 18, 200, 70, 45]

# Import pandas as pd
import pandas as pd

# Create dictionary my_dict with three key:value pairs: my_dict
my_dict = {'country': names, 'drives_right': dr, 'cars_per_cap': cpc}

# Build a DataFrame cars from my_dict: cars
cars = pd.DataFrame(my_dict)

# Print cars
cars
```

Out[ ]:

	country	drives_right	cars_per_cap
0	United States	True	809
1	Australia	False	731
2	Japan	False	588
3	India	False	18
4	Russia	True	200
5	Morocco	True	70
6	Egypt	True	45

Good job! Notice that the columns of cars can be of different types. This was not possible with 2D Numpy arrays!

## Dictionary to DataFrame (2)

Have you noticed above that the row labels (i.e. the labels for the different observations) were automatically set to integers from 0 up to 6?

To solve this a list `row_labels` has been created. You can use it to specify the row labels of the `cars` DataFrame. You do this by setting the `index` attribute of `cars`, that you can access as `cars.index`.

In [ ]:

```
# Definition of row_labels
row_labels = ['US', 'AUS', 'JAP', 'IN', 'RU', 'MOR', 'EG']

# Specify row labels of cars
cars.index = row_labels
```

```
# Print cars  
cars
```

```
Out[ ]:      country  drives_right  cars_per_cap  
    US  United States       True        809  
  AUS   Australia     False        731  
  JAP      Japan     False        588  
   IN      India     False         18  
  RU      Russia       True        200  
MOR   Morocco       True         70  
  EG      Egypt       True         45
```

Nice! That looks much better already!

## CSV to DataFrame (1)

Putting data in a dictionary and then building a DataFrame works, but it's not very efficient. What if you're dealing with millions of observations? In those cases, the data is typically available as files with a regular structure. One of those file types is the CSV file, which is short for "comma-separated values".

To import CSV data into Python as a Pandas DataFrame you can use `read_csv()`.

Let's explore this function with the same cars data from the previous exercises. This time, however, the data is available in a CSV file, named `cars.csv`. It is available in your current working directory, so the path to the file is simply '`cars.csv`'.

```
In [ ]: # Import the cars.csv data: cars  
cars = pd.read_csv('cars.csv')  
  
# Print out cars  
cars
```

```
-----
```

```
FileNotFoundException
```

```
Traceback (most recent call last)
```

```
Cell In[35], line 2
```

```
 1 # Import the cars.csv data: cars
----> 2 cars = pd.read_csv('cars.csv')
  4 # Print out cars
  5 cars
```

```
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:948, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision, storage_options, dtype_backend)
```

```
 935 kwds_defaults = _refine_defaults_read(
 936     dialect,
 937     delimiter,
(...),
 944     dtype_backend=dtype_backend,
 945 )
 946 kwds.update(kwds_defaults)
--> 948 return _read(filepath_or_buffer, kwds)
```

```
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:611, in _read(filepath_or_buffer, kwds)
 608 _validate_names(kwds.get("names", None))
 610 # Create the parser.
--> 611 parser = TextFileReader(filepath_or_buffer, **kwds)
 613 if chunksize or iterator:
 614     return parser
```

```
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1448, in TextFileReader.__init__(self, f, engine, **kwds)
 1445     self.options["has_index_names"] = kwds["has_index_names"]
 1447 self.handles: IOHandles | None = None
-> 1448 self._engine = self._make_engine(f, self.engine)
```

```
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1705, in TextFileReader._make_engine(self, f, engine)
 1703     if "b" not in mode:
 1704         mode += "b"
--> 1705 self.handles = get_handle(
 1706     f,
 1707     mode,
 1708     encoding=self.options.get("encoding", None),
 1709     compression=self.options.get("compression", None),
 1710     memory_map=self.options.get("memory_map", False),
 1711     is_text=is_text,
 1712     errors=self.options.get("encoding_errors", "strict"),
```

```
1713     storage_options=self.options.get("storage_options", None),
1714 )
1715 assert self.handles is not None
1716 f = self.handles.handle

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\common.py:863, in get_handle(path_or_buf,
mode, encoding, compression, memory_map, is_text, errors, storage_options)
858 elif isinstance(handle, str):
859     # Check whether the filename is to be opened in binary mode.
860     # Binary mode does not support 'encoding' and 'newline'.
861     if ioargs.encoding and "b" not in ioargs.mode:
862         # Encoding
--> 863         handle = open(
864             handle,
865             ioargs.mode,
866             encoding=ioargs.encoding,
867             errors=errors,
868             newline="",
869         )
870     else:
871         # Binary mode
872         handle = open(handle, ioargs.mode)

FileNotFoundException: [Errno 2] No such file or directory: 'cars.csv'
```

Nice job! Looks nice, but not exactly what we expected. Let's fix this in the next exercise.

## CSV to DataFrame (2)

Your `read_csv()` call to import the CSV data didn't generate an error, but the output is not entirely what we wanted. The row labels were imported as another column without a name.

Remember `index_col`, an argument of `read_csv()`, that you can use to specify which column in the CSV file should be used as a row label? Well, that's exactly what you need here!

```
In [ ]: # Fix import by including index_col
cars = pd.read_csv('cars.csv', index_col = 0)

# Print out cars
cars
```

	<code>cars_per_cap</code>	<code>country</code>	<code>drives_right</code>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False
<b>IN</b>	18	India	False
<b>RU</b>	200	Russia	True
<b>MOR</b>	70	Morocco	True
<b>EG</b>	45	Egypt	True

That's much better!

## Square Brackets (1)

You saw that you can index and select Pandas DataFrames in many different ways. The simplest, but not the most powerful way, is to use square brackets.

To select only the `cars_per_cap` column from `cars`, you can use:

```
cars['cars_per_cap']  
cars[['cars_per_cap']]
```

The single bracket version gives a Pandas Series, the double bracket version gives a Pandas DataFrame.

```
In [ ]: from IPython import InteractiveShell  
InteractiveShell.ast_node_interactivity = 'all'
```

```
In [ ]: # Print out country column as Pandas Series  
cars['country']  
  
# Print out country column as Pandas DataFrame  
cars[['country']]  
  
# Print out DataFrame with country and drives_right columns  
cars[['country', 'drives_right']]
```

```
US      United States
AUS     Australia
JAP     Japan
IN      India
RU      Russia
MOR     Morocco
EG      Egypt
Name: country, dtype: object
```

### country

country
US United States
AUS Australia
JAP Japan
IN India
RU Russia
MOR Morocco
EG Egypt

### country drives\_right

country	drives_right
US United States	True
AUS Australia	False
JAP Japan	False
IN India	False
RU Russia	True
MOR Morocco	True
EG Egypt	True

## Square Brackets (2)

Square brackets can do more than just selecting columns. You can also use them to get rows, or observations, from a DataFrame. The following call selects the first five rows from the cars DataFrame:

```
cars[0:5]
```

The result is another DataFrame containing only the rows you specified.

Pay attention: You can only select rows using square brackets if you specify a slice, like 0:4. Also, you're using the integer indexes of the rows here, not the row labels!

```
In [ ]: # Print out first 3 observations  
cars[0:3]
```

```
# Print out fourth, fifth and sixth observation  
cars[3:6]
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>IN</b>	18	India	False
<b>RU</b>	200	Russia	True
<b>MOR</b>	70	Morocco	True

Good job. You can get interesting information, but using square brackets to do indexing is rather limited. Experiment with more advanced techniques in the following exercises.

## loc and iloc (1)

With loc and iloc you can do practically any data selection operation on DataFrames you can think of. loc is label-based, which means that you have to specify rows and columns based on their row and column labels. iloc is integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.

Try out the following commands in the IPython Shell to experiment with loc and iloc to select observations. Each pair of commands here gives the same result.

```
cars.loc['RU']  
cars.iloc[4]
```

```
cars.loc[['RU']]  
cars.iloc[[4]]
```

```
cars.loc[['RU', 'AUS']]  
cars.iloc[[4, 1]]
```

```
In [ ]: cars
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False
<b>IN</b>	18	India	False
<b>RU</b>	200	Russia	True
<b>MOR</b>	70	Morocco	True
<b>EG</b>	45	Egypt	True

```
In [ ]: # Print out observation for Japan
```

```
cars.loc['JAP']
```

```
# Print out observations for Australia and Egypt
```

```
cars.loc[['AUS', 'EG']]
```

```
cars_per_cap      588
country          Japan
drives_right     False
Name: JAP, dtype: object
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>AUS</b>	731	Australia	False
<b>EG</b>	45	Egypt	True

You aced selecting observations from DataFrames; over to selecting both rows and columns!

## loc and iloc (2)

loc and iloc also allow you to select both rows and columns from a DataFrame. To experiment, try out the following commands. Again, paired commands produce the same result.

```
cars.loc['IN', 'cars_per_cap']
cars.iloc[3, 0]
```

```
cars.loc[['IN', 'RU'], 'cars_per_cap']
cars.iloc[[3, 4], 0]
```

```
cars.loc[['IN', 'RU'], ['cars_per_cap', 'country']]  
cars.iloc[[3, 4], [0, 1]]
```

```
In [ ]: # Print out drives_right value of Morocco  
cars.loc['MOR', 'drives_right']
```

```
# Print sub-DataFrame  
cars.loc[['RU', 'MOR'], ['country', 'drives_right']]
```

```
True
```

	country	drives_right
<b>RU</b>	Russia	True
<b>MOR</b>	Morocco	True

Great! You might wonder if you can also combine label-based selection the loc way and index-based selection the iloc way. You can! It's done with ix, but we won't go into that here.

## loc and iloc (3)

It's also possible to select only columns with loc and iloc. In both cases, you simply put a slice going from beginning to end in front of the comma:

```
cars.loc[:, 'country']  
cars.iloc[:, 1]
```

```
cars.loc[:, ['country', 'drives_right']]  
cars.iloc[:, [1, 2]]
```

```
In [ ]: # Print out drives_right column as Series  
cars.loc[:, 'drives_right']
```

```
# Print out drives_right column as DataFrame  
cars.loc[:, ['drives_right']]
```

```
# Print out cars_per_cap and drives_right as DataFrame  
cars.loc[:, ['cars_per_cap', 'drives_right']]
```

```
US      True  
AUS     False  
JAP     False  
IN      False  
RU      True  
MOR     True  
EG      True  
Name: drives_right, dtype: bool
```

drives_right		
		drives_right
US	True	
AUS	False	
JAP	False	
IN	False	
RU	True	
MOR	True	
EG	True	

	cars_per_cap	drives_right
US	809	True
AUS	731	False
JAP	588	False
IN	18	False
RU	200	True
MOR	70	True
EG	45	True

What a drill on indexing and selecting data from Pandas DataFrames! You've done great! It's time to head over to chapter 3 to learn all about logic, control flow, and filtering!

## Chapter 3 - Logic, Control Flow and Filtering

Boolean logic is the foundation of decision-making in your Python programs. Learn about different comparison operators, how you can combine them with boolean operators and how to use the boolean outcomes in control structures. You'll also learn to filter data from Pandas DataFrames using logic.

### Equality

To check if two Python values, or variables, are equal you can use `==`. To check for inequality, you need `!=`. As a refresher, have a look at the following examples that all result in True.

```
2 == (1 + 1)
"intermediate" != "python"
True != False
"Python" != "python" When you write these comparisons in a script, you will need to wrap a print() function around them to see the output.
```

```
In [ ]: # Comparison of booleans
print(True == False)

# Comparison of integers
print(-5*15 != 75)

# Comparison of strings
print("pyscript" == "PyScript")

# Compare a boolean with an integer
print(True == 1)
```

```
False
True
False
True
```

The last comparison worked fine because actually, a boolean is a special kind of integer: True corresponds to 1, False corresponds to 0

## Greater and less than

You know about the less than and greater than signs, `<` and `>` in Python. You can combine them with an equals sign: `<=` and `>=`. Pay attention: `<=` is valid syntax, but `=<` is not.

All Python expressions in the following code chunk evaluate to True:

```
3 < 4
3 <= 4
"alpha" <= "beta"
```

Remember that for string comparison, Python determines the relationship based on alphabetical order

```
In [ ]: # Comparison of integers
x = -3 * 6
print(x >= -10)

# Comparison of strings
y = "test"
```

```
print("test" <= y)

# Comparison of booleans
print(True > False)
```

```
False
True
True
```

## Compare arrays

Out of the box, you can also use comparison operators with Numpy arrays.

Remember areas, the list of area measurements for different rooms in your house from the previous course? This time there's two Numpy arrays: my\_house and your\_house. They both contain the areas for the kitchen, living room, bedroom and bathroom in the same order, so you can compare them.

```
In [ ]: # Create arrays
import numpy as np
my_house = np.array([18.0, 20.0, 10.75, 9.50])
your_house = np.array([14.0, 24.0, 14.25, 9.0])

# my_house greater than or equal to 18
print(my_house >= 18)

# my_house less than your_house
print(my_house < your_house)
```

```
[ True  True False False]
[False  True  True False]
```

Good job. It appears that the living room and bedroom in my\_house are smaller than the corresponding areas in your\_house.

## and, or, not (1)

A boolean is either 1 or 0, True or False. With boolean operators such as and, or and not, you can combine these booleans to perform more advanced queries on your data.

```
In [ ]: # Define variables
my_kitchen = 18.0
your_kitchen = 14.0

# my_kitchen bigger than 10 and smaller than 18?
print(my_kitchen > 10 and my_kitchen < 18)

# my_kitchen smaller than 14 or bigger than 17?
print(my_kitchen < 14 or my_kitchen > 17)
```

```
# Double my_kitchen smaller than triple your_kitchen?  
print(my_kitchen*2 < your_kitchen*3)
```

```
False  
True  
True
```

## and, or, not (2)

To see if you completely understood the boolean operators, have a look at the following piece of Python code:

```
x = 8  
y = 9  
not(not(x < 3) and not(y > 14 or y > 10))
```

What will the result be if you execute these three commands in the IPython Shell?

NB: Notice that not has a higher priority than and and or, it is executed first.

**False**

Correct!  $x < 3$  is False.  $y > 14$  or  $y > 10$  is False as well. If you continue working like this, simplifying from inside outwards, you'll end up with False.

## Boolean operators with Numpy

Before, the operational operators like `<` and `>=` worked with Numpy arrays out of the box. Unfortunately, this is not true for the boolean operators `and`, `or`, and `not`.

To use these operators with Numpy, you will need `np.logical_and()`, `np.logical_or()` and `np.logical_not()`. Here's an example on the `my_house` and `your_house` arrays from before to give you an idea:

```
np.logical_and(your_house > 13,  
               your_house < 15)
```

```
In [ ]: # my_house greater than 18.5 or smaller than 10  
print(np.logical_or(my_house > 18.5, my_house < 10))  
  
# Both my_house and your_house smaller than 11  
print(np.logical_and(my_house < 11, your_house < 11))
```

```
[False  True False  True]  
[False False False  True]
```

Correcto perfecto!

**if**

It's time to take a closer look around in your house.

Two variables are defined in the sample code: room, a string that tells you which room of the house we're looking at, and area, the area of that room.

In [ ]:

```
# Define variables
room = "kit"
area = 14.0

# if statement for room
if room == "kit" :
    print("looking around in the kitchen.")

# if statement for area
if area > 15:
    print('big place!')
```

looking around in the kitchen.

Great! big place! wasn't printed, because area > 15 is not True. Experiment with other values of room and area to see how the printouts change.

## Add else

The if construct for room has been extended with an else statement so that "looking around elsewhere." is printed if the condition room == "kit" evaluates to False.

Can you do a similar thing to add more functionality to the if construct for area?

In [ ]:

```
# if-else construct for room
if room == "kit" :
    print("looking around in the kitchen.")
else :
    print("looking around elsewhere.")

# if-else construct for area
if area > 15 :
    print("big place!")
else:
    print("pretty small.")
```

looking around in the kitchen.  
pretty small.

Nice! Again, feel free to play around with different values of room and area some more. After, head over to the next exercise where you'll take this customization one step further!

## Customize further: elif

It's also possible to have a look around in the bedroom. The sample code contains an `elif` part that checks if `room` equals "bed". In that case, "looking around in the bedroom." is printed out.

It's up to you now! Make a similar addition to the second control structure to further customize the messages for different values of `area`.

```
In [ ]: # if-elif-else construct for room
if room == "kit" :
    print("looking around in the kitchen.")
elif room == "bed":
    print("looking around in the bedroom.")
else :
    print("looking around elsewhere.")

# if-elif-else construct for area
if area > 15 :
    print("big place!")
elif area > 10:
    print("medium size, nice!")
else :
    print("pretty small.")
```

```
looking around in the kitchen.
medium size, nice!
```

## Driving right (1)

Remember that `cars` dataset, containing the cars per 1000 people (`cars_per_cap`) and whether people drive right (`drives_right`) for different countries (`country`)?

Let's start simple and try to find all observations in `cars` where `drives_right` is True.

`drives_right` is a boolean column, so you'll have to extract it as a Series and then use this boolean Series to select observations from `cars`.

```
In [ ]: # Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

# Extract drives_right column as Series: dr
dr = cars['drives_right']

# Use dr to subset cars: sel
sel = cars[dr]

# Print sel
print(sel)
```

	cars_per_cap	country	drives_right
US	809	United States	True
RU	200	Russia	True
MOR	70	Morocco	True
EG	45	Egypt	True

## Driving right (2)

The code in the previous example worked fine, but you actually unnecessarily created a new variable dr. You can achieve the same result without this intermediate variable. Put the code that computes dr straight into the square brackets that select observations from cars

```
In [ ]: cars[cars['drives_right']]
```

	cars_per_cap	country	drives_right
US	809	United States	True
RU	200	Russia	True
MOR	70	Morocco	True
EG	45	Egypt	True

## Cars per capita (1)

Let's stick to the cars data some more. This time you want to find out which countries have a high cars per capita figure. In other words, in which countries do many people have a car, or maybe multiple cars.

Similar to the previous example, you'll want to build up a boolean Series, that you can then use to subset the cars DataFrame to select certain observations. If you want to do this in a one-liner, that's perfectly fine!

```
In [ ]: # Create car_maniac: observations that have a cars_per_cap over 500
cpc = cars['cars_per_cap']
#many_cars = cars[cpc > 500]

car_maniac = cars[cars['cars_per_cap'] > 500]

# Print car_maniac
car_maniac
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False

```
In [ ]: cars[cars['cars_per_cap'] > 500]
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False

Good job! The output shows that the US, Australia and Japan have a cars\_per\_cap of over 500.

```
In [ ]: cars
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>
<b>US</b>	809	United States	True
<b>AUS</b>	731	Australia	False
<b>JAP</b>	588	Japan	False
<b>IN</b>	18	India	False
<b>RU</b>	200	Russia	True
<b>MOR</b>	70	Morocco	True
<b>EG</b>	45	Egypt	True

## Cars per capita (2)

Remember about np.logical\_and(), np.logical\_or() and np.logical\_not(), the Numpy variants of the and, or and not operators? You can also use them on Pandas Series to do more advanced filtering operations.

Take this example that selects the observations that have a cars\_per\_cap between 10 and 80. Try out these lines of code step by step to see what's happening.

```
cpc = cars['cars_per_cap']
between = np.logical_and(cpc > 10, cpc < 80)
```

```
medium = cars[between]
```

```
In [ ]: # Import numpy, you'll need this
```

```
import numpy as np
```

```
# Create medium: observations with cars_per_cap between 100 and 500
```

```
cpc = cars['cars_per_cap']
```

```
between = np.logical_and(cpc > 100, cpc < 500)
```

```
medium = cars[between]
```

```
# Print medium
```

```
medium
```

	<code>cars_per_cap</code>	<code>country</code>	<code>drives_right</code>
<b>RU</b>	200	Russia	True

Great work!

## Chapter 4 - Loops

There are several techniques to repeatedly execute Python code. While loops are like repeated if statements; the for loop is there to iterate over all kinds of data structures. Learn all about them in this chapter.

### while: warming up

The while loop is like a repeated if statement. The code is executed over and over again, as long as the condition is True. Have another look at its recipe.

```
while condition :  
    expression
```

Can you tell how many printouts the following while loop will do?

```
x = 1  
while x < 4 :  
    print(x)  
    x = x + 1
```

**Answer:** 3

Correct! After 3 runs, x will be equal to 4, causing  $x < 4$  to evaluate to False. This means that the while loop is executed 3 times, giving three printouts.

## Basic while loop

Below you can find the example where the error variable, initially equal to 50.0, is divided by 4 and printed out on every run:

```
error = 50.0
while error > 1 :
    error = error / 4
    print(error)
```

This example will come in handy, because it's time to build a while loop yourself! We're going to code a while loop that implements a very basic control system for an inverted pendulum. If there's an offset from standing perfectly straight, the while loop will incrementally fix this offset

```
In [ ]: # Initialize offset
offset = 8

# Code the while Loop
while offset !=0:
    print('correcting...')
    offset -= 1
    print(offset)

correcting...
7
correcting...
6
correcting...
5
correcting...
4
correcting...
3
correcting...
2
correcting...
1
correcting...
0
```

## Add conditionals

The while loop that corrects the offset is a good start, but what if offset is negative? You can try to run the following code where offset is initialized to -6:

```
# Initialize offset
offset = -6
```

```
# Code the while Loop
while offset != 0 :
    print("correcting...")
    offset = offset - 1
    print(offset)
```

but your session will be disconnected. The while loop will never stop running, because offset will be further decreased on every run. offset != 0 will never become False and the while loop continues forever.

Fix things by putting an if-else statement inside the while loop.

```
In [ ]: # Initialize offset
offset = -6

# Code the while Loop
while offset != 0 :
    print("correcting...")
    if offset > 0:
        offset -= 1
    else:
        offset += 1
    print(offset)

correcting...
-5
correcting...
-4
correcting...
-3
correcting...
-2
correcting...
-1
correcting...
0
```

Good work! The while loop is not that often used in Data Science, so let's head over to the for loop.

## Loop over a list

```
In [ ]: # areas list
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Code the for loop
for elements in areas:
    print(elements)
```

```
11.25  
18.0  
20.0  
10.75  
9.5
```

Great! That wasn't too hard, was it?

## Indexes and values (1)

Using a for loop to iterate over a list only gives you access to every list element in each run, one after the other. If you also want to access the index information, so where the list element you're iterating over is located, you can use enumerate().

As an example, have a look:

```
fam = [1.73, 1.68, 1.71, 1.89]  
for index, height in enumerate(fam) :  
    print("person " + str(index) + ":" + str(height))
```

```
In [ ]: # areas list  
areas = [11.25, 18.0, 20.0, 10.75, 9.50]
```

```
# Change for Loop to use enumerate() and update print()  
for index, value in enumerate(areas) :  
    print("room", index,":",value)
```

```
room 0 : 11.25  
room 1 : 18.0  
room 2 : 20.0  
room 3 : 10.75  
room 4 : 9.5
```

## Indexes and values (2)

For non-programmer folks, room 0: 11.25 is strange. Wouldn't it be better if the count started at 1?

```
In [ ]: # Code the for loop  
for index, area in enumerate(areas) :  
    print("room " + str(index+1) + ":" + str(area))
```

```
room 1: 11.25  
room 2: 18.0  
room 3: 20.0  
room 4: 10.75  
room 5: 9.5
```

Much better!

## Loop over list of lists

Remember the house variable from the Intro to Python course? . It's basically a list of lists, where each sublist contains the name and area of a room in your house.

It's up to you to build a for loop from scratch this time!

```
In [ ]: # house list of lists
house = [["hallway", 11.25],
          ["kitchen", 18.0],
          ["living room", 20.0],
          ["bedroom", 10.75],
          ["bathroom", 9.50]]

# Build a for loop from scratch
for name in house:
    print("the ",name[0]," is", name[1], "sqm")

the hallway is 11.25 sqm
the kitchen is 18.0 sqm
the living room is 20.0 sqm
the bedroom is 10.75 sqm
the bathroom is 9.5 sqm
```

## Loop over dictionary

In Python 3, you need the items() method to loop over a dictionary:

```
world = { "afghanistan":30.55,
          "albania":2.77,
          "algeria":39.21 }

for key, value in world.items() :
    print(key + " -- " + str(value))
```

Remember the europe dictionary that contained the names of some European countries as key and their capitals as corresponding value? Go ahead and write a loop to iterate over it!

```
In [ ]: # Definition of dictionary
europe = {'spain':'madrid', 'france':'paris', 'germany':'berlin',
          'norway':'oslo', 'italy':'rome', 'poland':'warsaw', 'austria':'vienna'}

# Iterate over europe
for key, value in europe.items():
    print("the capital of ",key,"is", value)
```

```
the capital of spain is madrid  
the capital of france is paris  
the capital of germany is berlin  
the capital of norway is oslo  
the capital of italy is rome  
the capital of poland is warsaw  
the capital of austria is vienna
```

Great! Notice that the order of the printouts doesn't necessarily correspond with the order used when defining europe. Remember: dictionaries are inherently unordered!

## Loop over Numpy array

If you're dealing with a 1D Numpy array, looping over all elements can be as simple as:

```
for x in my_array :  
    ...
```

If you're dealing with a 2D Numpy array, it's more complicated. A 2D array is built up of multiple 1D arrays. To explicitly iterate over all separate elements of a multi-dimensional array, you'll need this syntax:

```
for x in np.nditer(my_array) :  
    ...
```

Two Numpy arrays that you might recognize from the intro course are available in your Python session: np\_height, a Numpy array containing the heights of Major League Baseball players, and np\_baseball, a 2D Numpy array that contains both the heights (first column) and weights (second column) of those players.

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

```
In [ ]: np_height = np.array([74,74,72,72,73,69,69,71,71,76,71,73,73,74,74,69,70,73,75,78,79,76,74,76,72,71,75,77,74,73,74,78,73,75,73,75,  
    ,74,70,73,75,76,76,78,74,74,76,77,81,78,75,77,75,76,74,72,72,75,73,73,73,70,70,70,70,76,68,71,72,75,75,75,75,68,74,78,71,73,76,74,  
    ,74,73,72,74,73,74,72,73,69,72,73,75,75,73,72,72,76,74,72,77,74,77,75,76,80,74,74,75,78,73,73,74,75,76,71,73,74,76,76,74,73,74,70,  
    ,71,74,74,72,74,71,74,73,75,75,79,73,75,76,74,76,78,74,76,72,74,76,74,75,78,75,72,74,74,70,71,70,75,71,71,73,72,71,73,75,74,  
    ,76,75,74,76,75,73,71,76])
```

```
In [ ]: np_baseball = np.array([[74,180,74,215,72,210,72,210,73,188,69,176,69,209,71,200,76,231  
    ,71,180,73,188,73,180,74,185,74,160,69,180,70,185,73,189,75,185  
    ,78,219,79,230,76,205,74,230,76,195,72,180,71,192,75,225,77,203  
    ,74,195,73,182,74,188,78,200,73,180,75,200,73,200,75,245,75,240  
    ,74,215,69,185,71,175,74,199,73,200,73,215,76,200,74,205,74,206  
    ,70,186,72,188,77,220,74,210,70,195,73,200,75,200,76,212,76,224  
    ,78,210,74,205,74,220,76,195,77,200,81,260,78,228,75,270,77,200  
    ,75,210,76,190,74,220,72,180,72,205,75,210,73,220,73,211,73,200  
    ,70,180,70,190,70,170,76,230,68,155,71,185,72,185,75,200,75,225  
    ,75,225,75,220,68,160,74,205,78,235,71,250,73,210,76,190,74,160  
    ,74,200,79,205,75,222,73,195,76,205,74,220,74,220,73,170,72,185  
    ,74,195,73,220,74,230,72,180,73,220,69,180,72,180,73,170,75,210])
```

```
,75,215,73,200,72,213,72,180,76,192,74,235,72,185,77,235,74,210  
,77,222,75,210,76,230,80,220,74,180,74,190,75,200,78,210,73,194  
,73,180,74,190,75,240,76,200,71,198,73,200,74,195,76,210,76,220  
,74,190,73,210,74,225,70,180,72,185,73,170,73,185,73,185,73,180  
,71,178,74,175,74,200,72,204,74,211,71,190,74,210,73,190,75,190  
,75,185,79,290,73,175,75,185,76,200,74,220,76,170,78,220,74,190  
,76,220,72,205,74,200,76,250,74,225,75,215,78,210,75,215,72,195  
,74,200,72,194,74,220,70,180,71,180,70,170,75,195,71,180,71,170  
,73,206,72,205,71,200,73,225,72,201,75,225,74,233,74,180,75,225  
,73,180,77,220,73,180,76,237,75,215,74,190,76,235,75,190,73,180  
,71,165,76,195]]).reshape(200, 2)
```

In [ ]:

```
# Import numpy as np
import numpy as np

# For Loop over np_height
for x in np_height:
    print(x,"inches")

# For Loop over np_baseball
for x in np.nditer(np_baseball):
    print(x)
```

74 inches  
74 inches  
72 inches  
72 inches  
73 inches  
69 inches  
69 inches  
71 inches  
76 inches  
71 inches  
73 inches  
73 inches  
74 inches  
74 inches  
69 inches  
70 inches  
73 inches  
75 inches  
78 inches  
79 inches  
76 inches  
74 inches  
76 inches  
72 inches  
71 inches  
75 inches  
77 inches  
74 inches  
73 inches  
74 inches  
78 inches  
73 inches  
75 inches  
73 inches  
75 inches  
75 inches  
74 inches  
69 inches  
71 inches  
74 inches  
73 inches  
73 inches  
76 inches  
74 inches  
74 inches  
70 inches  
72 inches  
77 inches  
74 inches  
70 inches

73 inches  
75 inches  
76 inches  
76 inches  
78 inches  
74 inches  
74 inches  
76 inches  
77 inches  
81 inches  
78 inches  
75 inches  
77 inches  
75 inches  
76 inches  
74 inches  
72 inches  
72 inches  
75 inches  
73 inches  
73 inches  
73 inches  
70 inches  
70 inches  
70 inches  
76 inches  
68 inches  
71 inches  
72 inches  
75 inches  
75 inches  
75 inches  
75 inches  
68 inches  
74 inches  
78 inches  
71 inches  
73 inches  
76 inches  
74 inches  
74 inches  
79 inches  
75 inches  
73 inches  
76 inches  
74 inches  
74 inches  
73 inches  
72 inches  
74 inches

73 inches  
74 inches  
72 inches  
73 inches  
69 inches  
72 inches  
73 inches  
75 inches  
75 inches  
73 inches  
72 inches  
72 inches  
76 inches  
74 inches  
72 inches  
77 inches  
74 inches  
77 inches  
75 inches  
76 inches  
80 inches  
74 inches  
74 inches  
75 inches  
78 inches  
73 inches  
73 inches  
74 inches  
75 inches  
76 inches  
71 inches  
73 inches  
74 inches  
76 inches  
76 inches  
74 inches  
73 inches  
74 inches  
70 inches  
72 inches  
73 inches  
73 inches  
73 inches  
71 inches  
74 inches  
74 inches  
72 inches  
74 inches  
71 inches

74 inches  
73 inches  
75 inches  
75 inches  
79 inches  
73 inches  
75 inches  
76 inches  
74 inches  
76 inches  
78 inches  
74 inches  
76 inches  
72 inches  
74 inches  
76 inches  
74 inches  
75 inches  
78 inches  
75 inches  
72 inches  
74 inches  
72 inches  
74 inches  
70 inches  
71 inches  
70 inches  
75 inches  
71 inches  
71 inches  
73 inches  
72 inches  
71 inches  
73 inches  
72 inches  
75 inches  
74 inches  
74 inches  
75 inches  
73 inches  
77 inches  
73 inches  
76 inches  
75 inches  
74 inches  
76 inches  
75 inches  
73 inches  
71 inches  
76 inches

74

180

74

215

72

210

72

210

73

188

69

176

69

209

71

200

76

231

71

180

73

188

73

180

74

185

74

160

69

180

70

185

73

189

75

185

78

219

79

230

76

205

74

230

76

195

72

180

71

192

75

225

77

203

74

195

73

182

74

188

78

200

73

180

75

200

73

200

75

245

75

240

74

215

69

185

71

175

74

199

73

200

73

215

76

200

74

205

74

206

70

186

72

188

77

220

74

210

70

195

73

200

75

200

76

212

76

224

78

210

74

205

74

220

76

195

77

200

81

260

78

228

75

270

77

200

75

210

76

190

74

220

72

180

72

205

75

210

73

220

73

211

73

200

70

180

70

190

70

170

76

230

68

155

71

185

72

185

75

200

75

225

75

225

75

220

68

160

74

205

78

235

71

250

73

210

76

190

74

160

74

200

79

205

75

222

73

195

76

205

74

220

74

220

73

170

72

185

74

195

73

220

74

230

72

180

73

220

69

180

72

180

73

170

75

210

75

215

73

200

72

213

72

180

76

192

74

235

72

185

77

235

74

210

77

222

75

210

76

230

80

220

74

180

74

190

75

200

78

210

73

194

73

180

74

190

75

240

76

200

71

198

73

200

74

195

76

210

76

220

74

190

73

210

74

225

70

180

72

185

73

170

73

185

73

185

73

180

71

178

74

175

74

200

72

204

74

211

71

190

74

210

73

190

75

190

75

185

79

290

73

175

75

185

76

200

74

220

76

170

78

220

74

190

76

220

72

205

74

200

76

250

74

225

75

215

78

210

75

215

72

195

74

200

72

194

74

220

70

180

71

180

70

170

75

195

71

180

71

170

73

206

72

205

71

200

73

225

72

201

75

225

74

233

74

180

75

225

73

180

77

220

73

180

76

237

75

215

74

190

76

235

75

190

73

180

71

165

76

195

## Loop over DataFrame (1)

Iterating over a Pandas DataFrame is typically done with the `iterrows()` method. Used in a for loop, every observation is iterated over and on every iteration the row label and actual row contents are available:

```
for lab, row in brics.iterrows() :  
    ...
```

In this and the following exercises you will be working on the cars DataFrame. It contains information on the cars per capita and whether people drive right or left for seven countries in the world.

```
In [ ]:  
# Import cars data  
import pandas as pd  
cars = pd.read_csv('cars.csv', index_col = 0)  
  
# Iterate over rows of cars  
for row, value in cars.iterrows():  
    print(row)  
    print(value)
```

```
US
cars_per_cap      809
country          United States
drives_right     True
Name: US, dtype: object
AUS
cars_per_cap      731
country          Australia
drives_right    False
Name: AUS, dtype: object
JAP
cars_per_cap      588
country          Japan
drives_right    False
Name: JAP, dtype: object
IN
cars_per_cap      18
country          India
drives_right    False
Name: IN, dtype: object
RU
cars_per_cap      200
country          Russia
drives_right     True
Name: RU, dtype: object
MOR
cars_per_cap      70
country          Morocco
drives_right     True
Name: MOR, dtype: object
EG
cars_per_cap      45
country          Egypt
drives_right     True
Name: EG, dtype: object
```

## Loop over DataFrame (2)

The row data that's generated by iterrows() on every run is a Pandas Series. This format is not very convenient to print out. Luckily, you can easily select variables from the Pandas Series using square brackets:

```
for lab, row in brics.iterrows() :
    print(row['country'])
```

```
In [ ]: # Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)
```

```
# Adapt for Loop
for lab, row in cars.iterrows() :
    print(lab,": ",row['cars_per_cap'],sep = "")
```

```
US: 809
AUS: 731
JAP: 588
IN: 18
RU: 200
MOR: 70
EG: 45
```

## Add column (1)

You can add the length of the country names of the brics DataFrame in a new column:

```
for lab, row in brics.iterrows() :
    brics.loc[lab, "name_length"] = len(row["country"])
```

You can do similar things on the cars DataFrame.

```
In [ ]: # Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

# Code for loop that adds COUNTRY column
for lab, row in cars.iterrows():
    cars.loc[lab, "COUNTRY"] = row['country'].upper()

# Print cars
cars
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>	<b>COUNTRY</b>
<b>US</b>	809	United States	True	UNITED STATES
<b>AUS</b>	731	Australia	False	AUSTRALIA
<b>JAP</b>	588	Japan	False	JAPAN
<b>IN</b>	18	India	False	INDIA
<b>RU</b>	200	Russia	True	RUSSIA
<b>MOR</b>	70	Morocco	True	MOROCCO
<b>EG</b>	45	Egypt	True	EGYPT

Great, but you might remember that there is also an easier way to do this.

## Add column (2)

Using `iterrows()` to iterate over every observation of a Pandas DataFrame is easy to understand, but not very efficient. On every iteration, you're creating a new Pandas Series.

If you want to add a column to a DataFrame by calling a function on another column, the `iterrows()` method in combination with a for loop is not the preferred way to go. Instead, you'll want to use `apply()`.

Compare the `iterrows()` version with the `apply()` version to get the same result in the `brics` DataFrame:

```
for lab, row in brics.iterrows() :  
    brics.loc[lab, "name_length"] = len(row["country"])  
  
brics["name_length"] = brics["country"].apply(len)
```

We can do a similar thing to call the `upper()` method on every name in the country column. However, `upper()` is a method, so we'll need a slightly different approach:

```
In [ ]: cars = pd.read_csv('cars.csv', index_col = 0)  
  
# Use .apply(str.upper)  
cars['COUNTRY'] = cars['country'].apply(str.upper)  
cars
```

	<b>cars_per_cap</b>	<b>country</b>	<b>drives_right</b>	<b>COUNTRY</b>
<b>US</b>	809	United States	True	UNITED STATES
<b>AUS</b>	731	Australia	False	AUSTRALIA
<b>JAP</b>	588	Japan	False	JAPAN
<b>IN</b>	18	India	False	INDIA
<b>RU</b>	200	Russia	True	RUSSIA
<b>MOR</b>	70	Morocco	True	MOROCCO
<b>EG</b>	45	Egypt	True	EGYPT

Great job! It's time to blend everything you've learned together in a case-study. Head over to the next chapter!

## Chapter 5 - Case Study: Hacker Statistics

This chapter blends together everything you've learned up to now. You will use hacker statistics to calculate your chances of winning a bet. Use random number generators, loops and matplotlib to get the competitive edge!

## Random float

Randomness has many uses in science, art, statistics, cryptography, gaming, gambling, and other fields. You're going to use randomness to simulate a game.

All the functionality you need is contained in the random package, a sub-package of numpy. In this exercise, you'll be using two functions from this package:

- `seed()`: sets the random seed, so that your results are the reproducible between simulations. As an argument, it takes an integer of your choosing. If you call the function, no output will be generated.
- `rand()`: if you don't specify any arguments, it generates a random float between zero and one.

```
In [ ]: # Import numpy as np
import numpy as np

# Set the seed
np.random.seed(123)

# Generate and print random float
print(np.random.rand())
```

0.6964691855978616

Great! Now let's simulate a dice.

## Roll the dice

In the previous exercise, you used `rand()`, that generates a random float between 0 and 1.

You can just as well use `randint()`, also a function of the random package, to generate integers randomly. The following call generates the integer 4, 5, 6 or 7 randomly. 8 is not included.

```
np.random.randint(4, 8)
```

```
In [ ]: np.random.seed(123)

# Use randint() to simulate a dice
print(np.random.randint(1,7))
```

```
# Use randint() again
print(np.random.randint(1,7))

6
3
```

Alright! Time to actually start coding things up!

## Determine your next move

In the Empire State Building bet, your next move depends on the number of eyes you throw with the dice. We can perfectly code this with an if-elif-else construct!

The sample code assumes that you're currently at step 50. Can you fill in the missing pieces to finish the script?

```
In [ ]: np.random.seed(123)
# Starting step
step = 50

# Roll the dice
dice = np.random.randint(1,7)

# Finish the control construct
if dice <= 2 :
    step = step - 1
elif dice < 6:
    step += 1
else :
    step = step + np.random.randint(1,7)

# Print out dice and step
print(dice, step)
```

```
6 53
```

Cool! You threw a 6, so the code for the else statement was executed. You threw again, and apparently you threw 3, causing you to take three steps up: you're currently at step 53.

## The next step

Before, you have already written Python code that determines the next step based on the previous step. Now it's time to put this code inside a for loop so that we can simulate a random walk.

```
In [ ]: np.random.seed(123)
# Initialize random_walk
random_walk = [0]

# Complete the __
```

```

for x in range(100) :
    # Set step: last element in random_walk
    step = random_walk[-1]

    # Roll the dice
    dice = np.random.randint(1,7)

    # Determine next step
    if dice <= 2:
        step = step - 1
    elif dice <= 5:
        step = step + 1
    else:
        step = step + np.random.randint(1,7)

    # append next_step to random_walk
    random_walk.append(step)

# Print random_walk
print(random_walk)

```

```
[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, -1, 0, 5, 4, 3, 4, 3, 4, 5, 6, 7, 8, 7, 8, 7, 8, 9, 10, 11, 10, 14, 15, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25, 26, 27, 32, 33, 37, 38, 37, 38, 39, 38, 39, 40, 42, 43, 44, 43, 42, 43, 44, 43, 42, 43, 44, 4, 6, 45, 44, 45, 44, 45, 46, 47, 49, 48, 49, 50, 51, 52, 53, 52, 51, 52, 53, 52, 55, 56, 57, 58, 57, 58, 59]
```

Good job! There's still something wrong: the level at index 15 is negative!

## How low can you go?

Things are shaping up nicely! You already have code that calculates your location in the Empire State Building after 100 dice throws. However, there's something we haven't thought about - you can't go below 0!

A typical way to solve problems like this is by using `max()`. If you pass `max()` two arguments, the biggest one gets returned. For example, to make sure that a variable `x` never goes below 10 when you decrease it, you can use:

```
x = max(10, x - 1)
```

```
In [ ]: np.random.seed(123)
# Initialize random_walk
random_walk = [0]

for x in range(100) :
    step = random_walk[-1]
    dice = np.random.randint(1,7)

    if dice <= 2:
        # Replace below: use max to make sure step can't go below 0
        step = max(0, step - 1)
```

```
elif dice <= 5:  
    step = step + 1  
else:  
    step = step + np.random.randint(1,7)  
  
random_walk.append(step)  
  
print(random_walk)  
  
[0, 3, 4, 5, 4, 5, 6, 5, 4, 3, 2, 1, 0, 0, 1, 6, 5, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 8, 9, 10, 11, 12, 11, 15, 16, 15,  
16, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 33, 34, 38, 39, 38, 39, 40, 39, 40, 41, 43, 44, 45, 44, 43, 44, 45, 44, 43, 44, 45, 4  
7, 46, 45, 46, 45, 46, 47, 48, 50, 49, 50, 51, 52, 53, 54, 53, 52, 53, 54, 53, 56, 57, 58, 59, 58, 59, 60]
```

If you look closely at the output, you'll see that around index 15 the step stays at 0. You're not going below zero anymore. Great!

## Visualize the walk

Let's visualize this random walk! Remember how you could use matplotlib to build a line plot?

```
import matplotlib.pyplot as plt  
plt.plot(x, y)  
plt.show()
```

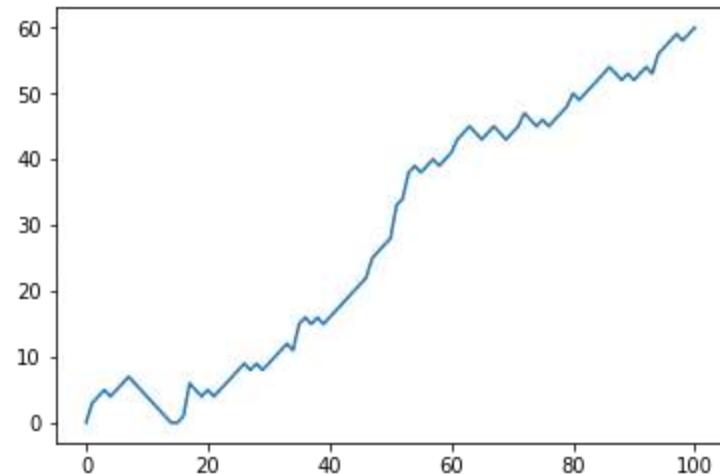
The first list you pass is mapped onto the x axis and the second list is mapped onto the y axis.

If you pass only one argument, Python will know what to do and will use the index of the list to map onto the x axis, and the values in the list onto the y axis.

```
In [ ]: np.random.seed(123)  
# Initialization  
random_walk = [0]  
  
for x in range(100) :  
    step = random_walk[-1]  
    dice = np.random.randint(1,7)  
  
    if dice <= 2:  
        step = max(0, step - 1)  
    elif dice <= 5:  
        step = step + 1  
    else:  
        step = step + np.random.randint(1,7)  
  
    random_walk.append(step)  
  
# Import matplotlib.pyplot as plt  
import matplotlib.pyplot as plt
```

```
# Plot random_walk
plt.plot(random_walk)
```

```
[<matplotlib.lines.Line2D at 0xc879d10>]
```



This is pretty cool! You can clearly see how your random walk progressed.

## Simulate multiple walks

A single random walk is one thing, but that doesn't tell you if you have a good chance at winning the bet.

To get an idea about how big your chances are of reaching 60 steps, you can repeatedly simulate the random walk and collect the results. That's exactly what you'll do in this exercise.

The sample code already sets you off in the right direction. Another for loop is wrapped around the code you already wrote. It's up to you to add some bits and pieces to make sure all of the results are recorded correctly.

```
In [ ]: np.random.seed(123)
# Initialize all_walks (don't change this line)
all_walks = []

# Simulate random walk 10 times
for i in range(10) :

    # Code from before
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)

        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
```

```

        step = step + 1
    else:
        step = step + np.random.randint(1,7)
    random_walk.append(step)

# Append random_walk to all_walks
all_walks.append(random_walk)

# Print all_walks
print(all_walks)

```

```

[[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, 0, 1, 6, 5, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 8, 9, 10, 11, 12, 11, 15, 16, 15, 16, 1
5, 16, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 33, 34, 38, 39, 38, 39, 40, 39, 40, 41, 43, 44, 45, 44, 43, 44, 45, 44, 43, 44, 4
5, 47, 46, 45, 46, 45, 46, 47, 48, 50, 49, 50, 51, 52, 53, 54, 53, 52, 53, 52, 53, 54, 53, 56, 57, 58, 59, 58, 59, 60], [0, 4, 3,
2, 4, 3, 4, 6, 7, 8, 13, 12, 13, 14, 15, 16, 17, 16, 21, 22, 23, 24, 23, 22, 21, 20, 19, 20, 21, 22, 28, 27, 26, 25, 26, 27, 28,
27, 28, 29, 28, 33, 34, 33, 32, 31, 30, 29, 31, 32, 35, 36, 38, 39, 40, 41, 40, 39, 40, 41, 42, 43, 42, 43, 44, 45, 48, 4
9, 50, 49, 50, 49, 50, 51, 52, 56, 55, 54, 55, 56, 57, 56, 57, 59, 64, 63, 64, 65, 66, 67, 68, 69, 68, 69, 70, 71, 73],
[0, 2, 1, 2, 3, 6, 5, 6, 5, 6, 7, 8, 7, 8, 9, 11, 10, 9, 10, 11, 10, 12, 13, 14, 15, 16, 17, 18, 17, 18, 19, 24, 25, 24, 2
3, 22, 21, 22, 23, 24, 29, 30, 29, 30, 31, 32, 33, 34, 35, 34, 33, 34, 33, 39, 38, 39, 38, 39, 38, 39, 43, 47, 49, 51, 50, 51, 5
3, 52, 58, 59, 61, 62, 61, 62, 63, 64, 63, 64, 65, 66, 68, 67, 66, 67, 73, 78, 77, 76, 80, 81, 82, 83, 85, 84, 85, 84, 8
3], [0, 6, 5, 6, 7, 8, 9, 10, 11, 12, 13, 12, 13, 12, 11, 12, 11, 12, 13, 17, 18, 17, 23, 22, 21, 22, 21, 20, 21, 20, 24,
23, 24, 23, 24, 23, 24, 26, 25, 24, 23, 24, 28, 29, 30, 29, 28, 29, 28, 29, 28, 33, 34, 33, 32, 31, 30, 31, 32, 36, 42, 43, 4
4, 45, 46, 45, 46, 48, 49, 50, 51, 50, 49, 50, 51, 52, 51, 52, 53, 54, 53, 52, 53, 54, 59, 60, 61, 66, 65, 66, 65, 66, 6
7, 68, 69, 68], [0, 6, 5, 6, 5, 4, 5, 9, 10, 11, 12, 13, 12, 11, 10, 9, 8, 9, 10, 11, 12, 13, 14, 13, 14, 15, 14, 15, 16, 19,
18, 19, 19, 22, 23, 24, 25, 24, 23, 26, 27, 28, 29, 28, 27, 28, 31, 32, 37, 38, 37, 38, 37, 38, 37, 43, 42, 41, 42, 44, 43, 42, 4
1, 42, 43, 44, 45, 49, 54, 55, 56, 57, 60, 61, 62, 63, 64, 65, 66, 65, 64, 65, 66, 65, 71, 70, 71, 72, 71, 70, 71, 70, 69, 75, 7
4, 73, 74, 75, 74, 73], [0, 0, 0, 1, 7, 8, 11, 12, 18, 19, 20, 26, 25, 31, 30, 31, 32, 33, 32, 38, 39, 38, 39, 38, 39, 38, 39, 3
8, 39, 43, 44, 46, 45, 46, 45, 44, 45, 44, 48, 52, 51, 50, 49, 50, 51, 55, 56, 57, 61, 60, 59, 58, 59, 60, 62, 61, 60, 6
1, 62, 64, 67, 72, 73, 72, 73, 74, 75, 76, 77, 76, 77, 78, 84, 83, 88, 87, 91, 90, 94, 93, 96, 97, 96, 97, 103, 102, 101, 100, 10
4, 103, 102, 103, 104, 103, 104, 105, 106, 107, 106], [0, 0, 0, 1, 0, 0, 4, 5, 7, 11, 17, 16, 15, 16, 17, 18, 17, 18, 17, 18, 19,
18, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 33, 32, 35, 36, 35, 34, 35, 36, 37, 36, 35, 34, 33, 34, 35, 36, 37, 38, 39, 40, 3
9, 40, 41, 43, 42, 43, 44, 47, 49, 50, 49, 48, 47, 46, 45, 46, 45, 46, 48, 49, 50, 49, 50, 49, 48, 49, 48, 47, 46, 47, 46, 45, 4
6, 47, 48, 50, 51, 52, 51, 50, 51, 57, 56, 57, 58, 63, 62, 63], [0, 0, 1, 2, 1, 2, 3, 9, 10, 11, 12, 11, 13, 14, 15, 16, 15,
17, 18, 19, 18, 19, 18, 19, 20, 19, 20, 24, 25, 28, 29, 33, 34, 33, 34, 35, 34, 33, 38, 39, 40, 39, 38, 39, 40, 41, 40, 44, 43, 4
4, 45, 46, 47, 48, 49, 50, 49, 48, 47, 48, 49, 53, 54, 53, 54, 55, 54, 60, 61, 62, 63, 62, 63, 64, 67, 66, 67, 66, 65, 64, 65, 6
6, 68, 69, 70, 74, 75, 74, 73, 74, 75, 76, 75, 74, 75, 76], [0, 1, 0, 1, 2, 1, 0, 0, 1, 2, 3, 4, 5, 10, 14, 13, 1
4, 13, 12, 11, 12, 11, 12, 13, 12, 16, 17, 16, 17, 16, 15, 16, 15, 19, 20, 21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 32, 33, 3
4, 33, 34, 33, 34, 35, 40, 41, 42, 41, 42, 43, 44, 43, 44, 44, 45, 44, 43, 42, 43, 44, 43, 42, 41, 42, 46, 47, 48, 4
9, 50, 51, 50, 51, 52, 51, 57, 58, 57, 56, 57, 56, 55, 54, 58, 59, 60, 61, 60], [0, 1, 2, 3, 4, 5, 4, 3, 6, 5, 4, 3, 2, 3, 9,
10, 9, 10, 11, 10, 11, 12, 11, 15, 16, 15, 17, 18, 17, 18, 19, 20, 21, 22, 23, 22, 21, 22, 23, 22, 23, 24, 23, 22, 21, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 33, 34, 35, 36, 37, 38, 37, 36, 37, 38, 37, 36, 42, 43, 44, 43, 42, 41, 45, 46, 50, 49, 55, 56,
57, 61, 62, 6, 1, 60, 61, 62, 63, 64, 63, 69, 70, 69, 73, 74, 73, 74, 73, 79, 85, 86, 85, 86, 87]]

```

## Visualize all walks

all\_walks is a list of lists: every sub-list represents a single random walk. If you convert this list of lists to a Numpy array, you can start making interesting plots!

In [ ]:

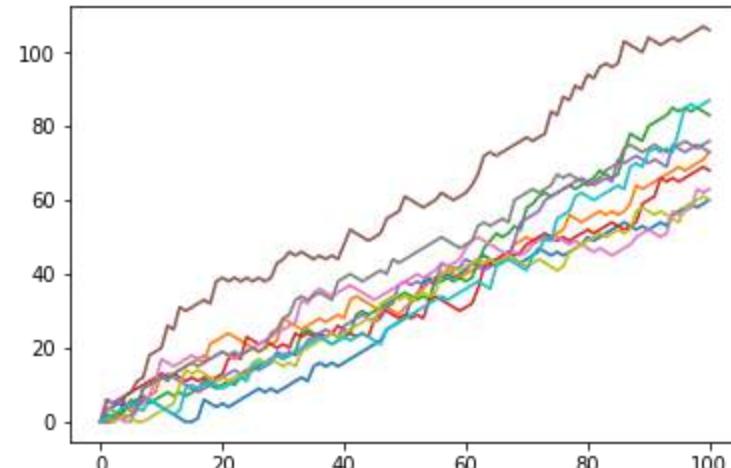
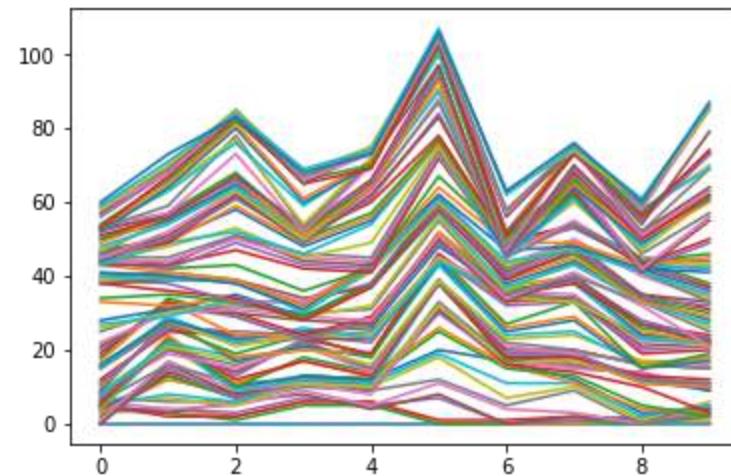
```
# Convert all_walks to Numpy array: np_aw
np_aw = np.array(all_walks)

# Plot np_aw and show
plt.plot(np_aw)
plt.show()

# Clear the figure
plt.clf()

# Transpose np_aw: np_aw_t
np_aw_t = np_aw.transpose()

# Plot np_aw_t and show
plt.plot(np_aw_t)
plt.show()
```



## Implement clumsiness

With this neatly written code of yours, changing the number of times the random walk should be simulated is super-easy. You simply update the `range()` function in the top-level for loop.

There's still something we forgot! You're a bit clumsy and you have a 0.1% chance of falling down. That calls for another random number generation. Basically, you can generate a random float between 0 and 1. If this value is less than or equal to 0.001, you should reset step to 0.

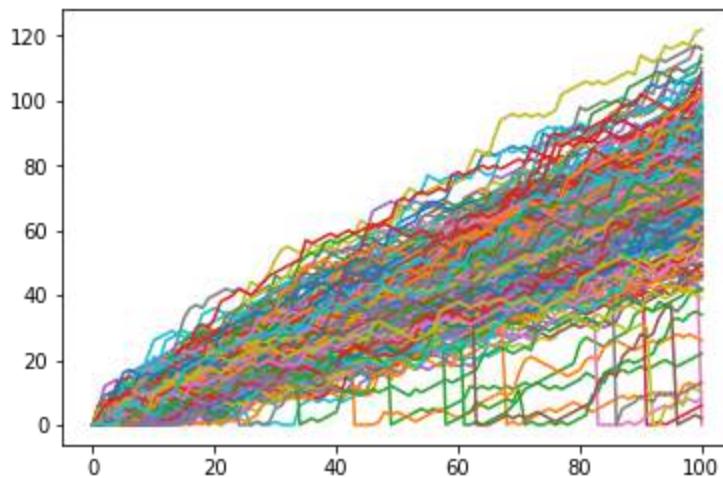
In [ ]:

```
np.random.seed(123)
# Simulate random walk 250 times
all_walks = []
for i in range(250) :
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)
        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)

        # Implement clumsiness
        if np.random.rand() <= 0.001:
            step = 0

        random_walk.append(step)
    all_walks.append(random_walk)

# Create and plot np_aw_t
np_aw_t = np.transpose(np.array(all_walks))
plt.plot(np_aw_t)
plt.show()
```



Superb! Look at the plot. In some of the 250 simulations you're indeed taking a deep dive down!

## Plot the distribution

All these fancy visualizations have put us on a sidetrack. We still have to solve the million-dollar problem: What are the odds that you'll reach 60 steps high on the Empire State Building?

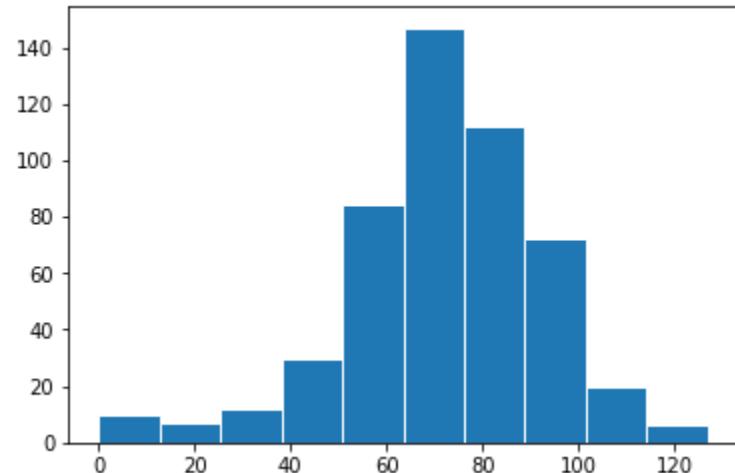
Basically, you want to know about the end points of all the random walks you've simulated. These end points have a certain distribution that you can visualize with a histogram.

```
In [ ]: np.random.seed(123)
# Simulate random walk 500 times
all_walks = []
for i in range(500) :
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)
        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)
        if np.random.rand() <= 0.001 :
            step = 0
        random_walk.append(step)
    all_walks.append(random_walk)

# Create and plot np_aw_t
np_aw_t = np.transpose(np.array(all_walks))
```

```
# Select last row from np_aw_t: ends
ends = np_aw_t[-1, :]

# Plot histogram of ends, display plot
plt.hist(ends, ec='white')
plt.show()
```



Great job! Have a look at a histogram; what do you think your chances are?

## Calculate the odds

The histogram of the previous exercise was created from a Numpy array `ends`, that contains 500 integers. Each integer represents the end point of a random walk. To calculate the chance that this end point is greater than or equal to 60, you can count the number of integers in `ends` that are greater than or equal to 60 and divide that number by 500, the total number of simulations.

Well then, what's the estimated chance that you'll reach 60 steps high if you play this Empire State Building game? The `ends` array is everything you need.

```
In [ ]: chance = np.sum(ends >= 60)/500 * 100
chance
```

78.4

Correct! Seems like you have a pretty high chance of winning the bet!

# Python Data Science Toolbox (Part 1)

## Chapter 1 - Writing your own functions

Here you will learn how to write your very own functions. In this Chapter, you'll learn how to write simple functions, as well as functions that accept multiple arguments and return multiple values. You'll also have the opportunity to apply these newfound skills to questions that commonly arise in Data Science contexts.

### Strings in Python

Strings represent textual data. To assign the string 'DataCamp' to a variable company, you execute:

company = 'DataCamp' You've also learned to use the operations + and \* with strings. Unlike with numeric types such as ints and floats, the + operator concatenates strings together, while the \* concatenates multiple copies of a string together. In this exercise, you will use the + and \* operations on strings to answer the question below. Execute the following code in the shell:

```
object1 = "data" + "analysis" + "visualization"  
object2 = 1*3  
object3 = "1"*3
```

What are the values in object1, object2, and object3, respectively?

```
In [ ]:  
object1 = 'data' + 'analysis' + 'visualization'  
object2 = 1*3  
object3 = "1"*3
```

```
print(object1)  
print(object2)  
print(object3)
```

```
dataanalysisvisualization  
3  
111
```

### Recapping built-in functions

In the video, Hugo briefly examined the return behavior of the built-in functions `print()` and `str()`. Here, you will use both functions and examine their return values. A variable `x` has been preloaded for this exercise. Run the code below in the console. Pay close attention to the results to answer the question that follows.

- Assign `str(x)` to a variable `y1`: `y1 = str(x)`
- Assign `print(x)` to a variable `y2`: `y2 = print(x)`
- Check the types of the variables `x`, `y1`, and `y2`.

What are the types of `x`, `y1`, and `y2`?

```
In [ ]: x = 4.89
```

```
In [ ]: y1 = str(x)
y2 = print(x)
```

```
4.89
```

```
In [ ]: print(type(x))
print(type(y1))
print(type(y2))
```

```
<class 'float'>
<class 'str'>
<class 'NoneType'>
```

Correct! It is important to remember that assigning a variable `y2` to a function that prints a value but does not return a value will result in that variable `y2` being of type `NoneType`.

## Write a simple function

You will now write your own function!

Define a function, `shout()`, which simply prints out a string with three exclamation marks '!!!' at the end. The code for the `square()` function is found below. You can use it as a pattern to define `shout()`.

```
def square():
    new_value = 4 ** 2
    return new_value
```

Function bodies need to be indented by a consistent number of spaces and the choice of 4 is common

```
In [ ]: # Define the function shout
def shout():
    """Print a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = 'congratulations' + '!!!'
```

```
# Print shout_word
print(shout_word)
```

```
# Call shout
shout()
```

```
congratulations!!!
```

## Single-parameter functions

Congratulations! You have successfully defined and called your own function! That's pretty cool.

In the previous exercise, you defined and called the function shout(), which printed out a string concatenated with '!!!'. You will now update shout() by adding a parameter so that it can accept and process any string argument passed to it. Also note that shout(word), the part of the header that specifies the function name and parameter(s), is known as the signature of the function. You may encounter this term in the wild!

```
In [ ]: # Define shout with the parameter, word
def shout(word):
    """Print a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = word + '!!!'

    # Print shout_word
    print(shout_word)

# Call shout with the string 'congratulations'
shout('congratulations')

congratulations!!!
```

## Functions that return single values

You're getting very good at this! Try your hand at another modification to the shout() function so that it now returns a single value instead of printing within the function. Recall that the return keyword lets you return values from functions. Returning values is generally more desirable than printing them out because, as you saw earlier, a print() call assigned to a variable has type NoneType

```
In [ ]: # Define shout with the parameter, word
def shout(word):
    """Return a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = word + '!!!'

    # Replace print with return
    return shout_word
```

```
# Pass 'congratulations' to shout: yell
yell = shout('congratulations')

# Print yell
print(yell)

congratulations!!!
```

Great work! Here it made sense to assign the output of shout('congratulations') to a variable yell because the function shout actually returns a value, it does not merely print one.

## Functions with multiple parameters

You are now going to use what you've learned to modify the shout() function further. Here, you will modify shout() to accept two arguments.

```
In [ ]: # Define shout with parameters word1 and word2
def shout(word1, word2):
    """Concatenate strings with three exclamation marks"""
    # Concatenate word1 with '!!!!': shout1
    shout1 = word1 + '!!!!'

    # Concatenate word2 with '!!!!': shout2
    shout2 = word2 + '!!!!'

    # Concatenate shout1 with shout2: new_shout
    new_shout = shout1 + shout2

    # Return new_shout
    return new_shout

# Pass 'congratulations' and 'you' to shout(): yell
yell = shout('congratulations', 'you')

# Print yell
print(yell)

congratulations!!!you!!!
```

## A brief introduction to tuples

Alongside learning about functions, you've also learned about tuples! Here, you will practice what you've learned about tuples: how to construct, unpack, and access tuple elements.

```
In [ ]: nums = (3, 4, 6)

In [ ]: type(nums)
```

```
Out[ ]: tuple
```

```
In [ ]: # Unpack nums into num1, num2, and num3
num1, num2, num3 = nums
```

```
In [ ]: print(num1)
print(num2)
print(num3)
```

```
3
4
6
```

## Functions that return multiple values

In the previous exercise, you constructed tuples, assigned tuples to variables, and unpacked tuples. Here you will return multiple values from a function using tuples. Let's now update our shout() function to return multiple values. Instead of returning just one string, we will return two strings with the string !!! concatenated to each.

Note that the return statement return x, y has the same result as return (x, y): the former actually packs x and y into a tuple under the hood!

```
In [ ]: # Define shout_all with parameters word1 and word2
def shout_all(word1, word2):

    # Concatenate word1 with '!!!': shout1
    shout1 = word1 + '!!!'

    # Concatenate word2 with '!!!': shout2
    shout2 = word2 + '!!!'

    # Construct a tuple with shout1 and shout2: shout_words
    shout_words = (shout1, shout2)

    # Return shout_words
    return shout_words

# Pass 'congratulations' and 'you' to shout_all(): yell1, yell2
yell1, yell2 = shout_all('congratulations', 'you')

# Print yell1 and yell2
print(yell1)
print(yell2)

congratulations!!!
you!!!
```

## Bringing it all together (1)

You've got your first taste of writing your own functions in the previous exercises. You've learned how to add parameters to your own function definitions, return a value or multiple values with tuples, and how to call the functions you've defined.

In this and the following exercise, you will bring together all these concepts and apply them to a simple data science problem. You will load a dataset and develop functionalities to extract simple insights from the data.

For this exercise, your goal is to recall how to load a dataset into a DataFrame. The dataset contains Twitter data and you will iterate over entries in a column to build a dictionary in which the keys are the names of languages and the values are the number of tweets in the given language. The file tweets.csv is available in your current directory.

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

# Import Twitter data as DataFrame: df
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
#df = pd.read_csv('tweets.csv')

# Initialize an empty dictionary: langs_count
langs_count = {}

# Extract column from DataFrame: col
col = df['lang']

# Iterate over lang column in DataFrame
for entry in col:

    # If the language is in langs_count, add 1
    if entry in langs_count.keys():
        langs_count[entry] +=1
    # Else add the language to langs_count, set the value to 1
    else:
        langs_count[entry] = 1

# Print the populated dictionary
print(langs_count)

{'en': 97, 'et': 1, 'und': 2}
```

## ringing it all together (2)

Great job! You've now defined the functionality for iterating over entries in a column and building a dictionary with keys the names of languages and values the number of tweets in the given language.

In this exercise, you will define a function with the functionality you developed in the previous exercise, return the resulting dictionary from within the function, and call the function with the appropriate arguments.

In [ ]:

```
# Define count_entries()
def count_entries(df, col_name):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: langs_count
    langs_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over lang column in DataFrame
    for entry in col:

        # If the language is in langs_count, add 1
        if entry in langs_count.keys():
            langs_count[entry] += 1
        # Else add the language to langs_count, set the value to 1
        else:
            langs_count[entry] = 1

    # Return the langs_count dictionary
    return langs_count

# Call count_entries(): result
result = count_entries(df, 'lang')

# Print the result
print(result)

{'en': 97, 'et': 1, 'und': 2}
```

---

## Chapter 2 - Default arguments, variable-length arguments and scope

In this chapter, you'll learn to write functions with default arguments, so that the user doesn't always need to specify them, and variable-length arguments, so that they can pass to your functions an arbitrary number of arguments. These are both incredibly useful tools! You'll also learn about the essential concept of scope. Enjoy!

## Pop quiz on understanding scope

In this exercise, you will practice what you've learned about scope in functions. The variable num has been predefined as 5, alongside the following function definitions:

```
def func1():
    num = 3
    print(num)

def func2():
    global num
    double_num = num * 2
    num = 6
    print(double_num)
```

Try calling func1() and func2() in the shell, then answer the following questions:

What are the values printed out when you call func1() and func2()? What is the value of num in the global scope after calling func1() and func2()?

```
In [ ]: num = 6
```

```
In [ ]: def func1():
    num = 3
    print(num)
```

```
In [ ]: func1()
```

```
3
```

```
In [ ]: def func2():
    global num
    double_num = num * 2
    num = 6
    print(double_num)
```

```
In [ ]: func2()
```

```
12
```

# The keyword global

Let's work more on your mastery of scope. In this exercise, you will use the keyword `global` within a function to alter the value of a variable defined in the global scope.

```
In [ ]: # Create a string: team
team = "teen titans"

# Define change_team()
def change_team():
    """Change the value of the global variable team."""

    # Use team in global scope
    global team

    # Change the value of team in global: team
    team = 'justice league'
# Print team
print(team)

# Call change_team()
change_team()

# Print team
print(team)
```

teen titans  
justice league

## Nested Functions I

One reason why you'd like to do this is to avoid writing out the same computations within functions repeatedly. There's nothing new about defining nested functions: you simply define it as you would a regular function with `def` and embed it inside another function!

In this exercise, inside a function `three_shouts()`, you will define a nested function `inner()` that concatenates a string object with `!!!`. `three_shouts()` then returns a tuple of three elements, each a string concatenated with `!!!` using `inner()`. Go for it!

```
In [ ]: # Define three_shouts
def three_shouts(word1, word2, word3):
    """Returns a tuple of strings
    concatenated with '!!!'."""

    # Define inner
```

```

def inner(word):
    """Returns a string concatenated with '!!!'."""
    return word + '!!!'

# Return a tuple of strings
return (inner(word1), inner(word2), inner(word3))

# Call three_shouts() and print
print(three_shouts('a', 'b', 'c'))

('a!!!', 'b!!!', 'c!!!')

```

## Nested Functions II

Great job, you've just nested a function within another function. One other pretty cool reason for nesting functions is the idea of a closure. This means that the nested or inner function remembers the state of its enclosing scope when called. Thus, anything defined locally in the enclosing scope is available to the inner function even when the outer function has finished execution.

Let's move forward then! In this exercise, you will complete the definition of the inner function `inner_echo()` and then call `echo()` a couple of times, each with a different argument. Complete the exercise and see what the output will be!

```

In [ ]: # Define echo
def echo(n):
    """Return the inner_echo function."""

    # Define inner_echo
    def inner_echo(word1):
        """Concatenate n copies of word1."""
        echo_word = word1 * n
        return echo_word

    # Return inner_echo
    return inner_echo

# Call echo: twice
twice = echo(2)

# Call echo: thrice
thrice = echo(3)

# Call twice() and thrice() then print
print(twice('hello'), thrice('hello'))

```

hellohello hellohellohello

## The keyword nonlocal and nested functions

Let's once again work further on your mastery of scope! In this exercise, you will use the keyword nonlocal within a nested function to alter the value of a variable defined in the enclosing scope.

In [ ]:

```
# Define echo_shout()
def echo_shout(word):
    """Change the value of a nonlocal variable"""

    # Concatenate word with itself: echo_word
    echo_word = word + word

    # Print echo_word
    print(echo_word)

    # Define inner function shout()
    def shout():
        """Alter a variable in the enclosing scope"""
        # Use echo_word in nonlocal scope
        nonlocal echo_word

        # Change echo_word to echo_word concatenated with '!!!'
        echo_word = echo_word + '!!!'

    # Call function shout()
    shout()

    # Print echo_word
    print(echo_word)

# Call function echo_shout() with argument 'hello'
echo_shout('hello')
```

```
hellohello
hellohello!!!
```

Quite something, that nonlocal keyword!

## Functions with one default argument

In the previous chapter, you've learned to define functions with more than one parameter and then calling those functions by passing the required number of arguments. In the last video, Hugo built on this idea by showing you how to define functions with default arguments. You will practice that skill in this exercise by writing a function that uses a default argument and then calling the function a couple of times.

In [ ]:

```
# Define shout_echo
def shout_echo(word1, echo = 1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Concatenate echo copies of word1 using *: echo_word
```

```

echo_word = echo*word1

# Concatenate '!!!!' to echo_word: shout_word
shout_word = echo_word + '!!!!'

# Return shout_word
return shout_word

# Call shout_echo() with "Hey": no_echo
no_echo = shout_echo('Hey')

# Call shout_echo() with "Hey" and echo=5: with_echo
with_echo = shout_echo('Hey', echo=5)

# Print no_echo and with_echo
print(no_echo)
print(with_echo)

```

Hey!!!  
HeyHeyHeyHeyHey!!!

## Functions with multiple default arguments

You've now defined a function that uses a default argument - don't stop there just yet! You will now try your hand at defining a function with more than one default argument and then calling this function in various ways.

After defining the function, you will call it by supplying values to all the default arguments of the function. Additionally, you will call the function by not passing a value to one of the default arguments - see how that changes the output of your function!

```
In [ ]: # Define shout_echo
def shout_echo(word1, echo = 1, intense = False):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Capitalize echo_word if intense is True
    if intense is True:
        # Capitalize and concatenate '!!!!': echo_word_new
        echo_word_new = echo_word.upper() + '!!!!'
    else:
        # Concatenate '!!!!' to echo_word: echo_word_new
        echo_word_new = echo_word + '!!!!'

    # Return echo_word_new
    return echo_word_new
```

```
# Call shout_echo() with "Hey", echo=5 and intense=True: with_big_echo
with_big_echo = shout_echo("Hey", 5, True)

# Call shout_echo() with "Hey" and intense=True: big_no_echo
big_no_echo = shout_echo("Hey", intense = True)

# Print values
print(with_big_echo)
print(big_no_echo)
```

```
HEYHEYHEYHEYHEY!!!
HEY!!!
```

## Functions with variable-length arguments (\*args)

Flexible arguments enable you to pass a variable number of arguments to a function. In this exercise, you will practice defining a function that accepts a variable number of string arguments.

The function you will define is gibberish() which can accept a variable number of string values. Its return value is a single string composed of all the string arguments concatenated together in the order they were passed to the function call. You will call the function with a single string argument and see how the output changes with another call using more than one string argument. Within the function definition, args is a tuple.

```
In [ ]: # Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""

    # Initialize an empty string: hodgepodge
    hodgepodge = ""

    # Concatenate the strings in args
    for word in args:
        hodgepodge += word

    # Return hodgepodge
    return hodgepodge

# Call gibberish() with one string: one_word
one_word = gibberish("luke")

# Call gibberish() with five strings: many_words
many_words = gibberish("luke", "leia", "han", "obi", "darth")

# Print one_word and many_words
print(one_word)
print(many_words)
```

luke  
lukeleiahobidarth

## Functions with variable-length keyword arguments (\*\*kwargs)

Let's push further on what you've learned about flexible arguments - you've used \*args, you're now going to use **kwargs!** What makes kwargs different is that it allows you to pass a variable number of keyword arguments to functions. Within the function definition, kwargs is a dictionary.

To understand this idea better, you're going to use \*\*kwargs in this exercise to define a function that accepts a variable number of keyword arguments. The function simulates a simple status report system that prints out the status of a character in a movie.

```
In [ ]: # Define report_status
def report_status(**kwargs):
    """Print out the status of a movie character."""

    print("\nBEGIN: REPORT\n")

    # Iterate over the key-value pairs of kwargs
    for key, value in kwargs.items():
        # Print out the keys and values, separated by a colon ':'
        print(key + ": " + value)

    print("\nEND REPORT")

# First call to report_status()
report_status(name='luke', affiliation='jedi', status='missing')

# Second call to report_status()
report_status(name='anakin', affiliation='sith lord', status='deceased')
```

BEGIN: REPORT

name: luke  
affiliation: jedi  
status: missing

END REPORT

BEGIN: REPORT

name: anakin  
affiliation: sith lord  
status: deceased

END REPORT

**Bringing it all together (1)**

Recall the Bringing it all together exercise in the previous chapter where you did a simple Twitter analysis by developing a function that counts how many tweets are in certain languages. The output of your function was a dictionary that had the language as the keys and the counts of tweets in that language as the value.

In this exercise, we will generalize the Twitter language analysis that you did in the previous chapter. You will do that by including a default argument that takes a column name.

```
In [ ]: # Define count_entries()
def count_entries(df, col_name = 'lang'):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over the column in DataFrame
    for entry in col:

        # If entry is in cols_count, add 1
        if entry in cols_count.keys():
            cols_count[entry] += 1

        # Else add the entry to cols_count, set the value to 1
        else:
            cols_count[entry] = 1

    # Return the cols_count dictionary
    return cols_count

# Call count_entries(): result1
result1 = count_entries(df, 'lang')

# Call count_entries(): result2
result2 = count_entries(df, 'source')

# Print result1 and result2
print(result1)
print(result2)
```

```
{'en': 97, 'et': 1, 'und': 2}
{'<a href="http://twitter.com" rel="nofollow">Twitter Web Client</a>': 24, '<a href="http://www.facebook.com/twitter" rel="nofollow">Facebook</a>': 1, '<a href="http://twitter.com/download/android" rel="nofollow">Twitter for Android</a>': 26, '<a href="http://twitter.com/download/iphone" rel="nofollow">Twitter for iPhone</a>': 33, '<a href="http://www.twitter.com" rel="nofollow">Twitter for BlackBerry</a>': 2, '<a href="http://www.google.com/" rel="nofollow">Google</a>': 2, '<a href="http://twitter.com/#/download/ipad" rel="nofollow">Twitter for iPad</a>': 6, '<a href="http://linkis.com" rel="nofollow">Linkis.com</a>': 2, '<a href="http://rutracker.org/forum/viewforum.php?f=93" rel="nofollow">newzlasz</a>': 2, '<a href="http://ifttt.com" rel="nofollow">IFTTT</a>': 1, '<a href="http://www.myplume.com/" rel="nofollow">Plume for Android</a>': 1}
```

## Bringing it all together (2)

Wow, you've just generalized your Twitter language analysis that you did in the previous chapter to include a default argument for the column name. You're now going to generalize this function one step further by allowing the user to pass it a flexible argument, that is, as many column names as the user would like!

In [ ]:

```
# Define count_entries()
def count_entries(df, *args):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Iterate over column names in args
    for col_name in args:

        # Extract column from DataFrame: col
        col = df[col_name]

        # Iterate over the column in DataFrame
        for entry in col:

            # If entry is in cols_count, add 1
            if entry in cols_count.keys():
                cols_count[entry] += 1

            # Else add the entry to cols_count, set the value to 1
            else:
                cols_count[entry] = 1

    # Return the cols_count dictionary
    return cols_count

# Call count_entries(): result1
result1 = count_entries(df, 'lang')

# Call count_entries(): result2
result2 = count_entries(df, 'lang', 'source')
```

```
# Print result1 and result2
```

```
print(result1)
```

```
print(result2)
```

```
{'en': 97, 'et': 1, 'und': 2}
{'en': 97, 'et': 1, 'und': 2, '<a href="http://twitter.com" rel="nofollow">Twitter Web Client</a>': 24, '<a href="http://www.facebook.com/twitter" rel="nofollow">Facebook</a>': 1, '<a href="http://twitter.com/download/android" rel="nofollow">Twitter for Android</a>': 26, '<a href="http://twitter.com/download/iphone" rel="nofollow">Twitter for iPhone</a>': 33, '<a href="http://www.twitter.com" rel="nofollow">Twitter for BlackBerry</a>': 2, '<a href="http://www.google.com/" rel="nofollow">Google</a>': 2, '<a href="http://twitter.com/#!/download/ipad" rel="nofollow">Twitter for iPad</a>': 6, '<a href="http://linkis.com" rel="nofollow">Linkis.com</a>': 2, '<a href="http://rutracker.org/forum/viewforum.php?f=93" rel="nofollow">newzlasz</a>': 2, '<a href="http://ifttt.com" rel="nofollow">IFTTT</a>': 1, '<a href="http://www.myplume.com/" rel="nofollow">PlumeÃ  forÃ  Android</a>': 1}
```

---

## Chapter 3 - Lambda functions and error-handling

Herein, you'll learn about lambda functions, which allow you to write functions quickly and on-the-fly. You'll also get practice at handling errors that your functions, at some point, will inevitably throw. You'll wrap up once again applying these skills to Data Science questions.

### Pop quiz on lambda functions

In this exercise, you will practice writing a simple lambda function and calling this function. Recall what you know about lambda functions and answer the following questions:

How would you write a lambda function add\_bangs that adds three exclamation points '!!!' to the end of a string a? How would you call add\_bangs with the argument 'hello'?

```
In [ ]: add_bangs = lambda a : a + '!!!'
```

```
In [ ]: add_bangs('hello')
```

```
Out[ ]: 'hello!!!'
```

### Writing a lambda function you already know

Some function definitions are simple enough that they can be converted to a lambda function. By doing this, you write less lines of code, which is pretty awesome and will come in handy, especially when you're writing and maintaining big programs. In this exercise, you will use what you know about lambda functions to convert a function that does a simple task into a lambda function. Take a look at this function definition:

```
def echo_word(word1, echo):
    """Concatenate echo copies of word1."""
    words = word1 * echo
    return words
```

The function `echo_word` takes 2 parameters: a string value, `word1` and an integer value, `echo`. It returns a string that is a concatenation of `echo` copies of `word1`. Your task is to convert this simple function into a lambda function.

```
In [ ]: # Define echo_word as a Lambda function: echo_word
echo_word = (lambda word1, echo : echo*word1)

# Call echo_word: result
result = echo_word('hey', 5)

# Print result
print(result)
```

heyheyheyheyhey

## Map() and lambda functions

So far, you've used lambda functions to write short, simple functions as well as to redefine functions with simple functionality. The best use case for lambda functions, however, are for when you want these simple functionalities to be anonymously embedded within larger expressions. What that means is that the functionality is not stored in the environment, unlike a function defined with `def`. To understand this idea better, you will use a lambda function in the context of the `map()` function.

`map()` applies a function over an object, such as a list. Here, you can use lambda functions to define the function that `map()` will use to process the object. For example:

```
nums = [2, 4, 6, 8, 10]

result = map(lambda a: a ** 2, nums)
```

You can see here that a lambda function, which raises a value `a` to the power of 2, is passed to `map()` alongside a list of numbers, `nums`. The map object that results from the call to `map()` is stored in `result`. You will now practice the use of lambda functions with `map()`. For this exercise, you will map the functionality of the `add_bangs()` function you defined in previous exercises over a list of strings.

```
In [ ]: # Create a list of strings: spells
spells = ["protego", "accio", "expecto patronum", "legilimens"]

# Use map() to apply a Lambda function over spells: shout_spells
shout_spells = map(lambda item : item + '!!!!', spells)

# Convert shout_spells to a list: shout_spells_list
shout_spells_list = list(shout_spells)
```

```
# Convert shout_spells into a list and print it
print(shout_spells_list)

['protego!!!!', 'accio!!!!', 'expecto patronum!!!!', 'legilimens!!!!']
```

## Filter() and lambda functions

In the previous exercise, you used lambda functions to anonymously embed an operation within map(). You will practice this again in this exercise by using a lambda function with filter(), which may be new to you! The function filter() offers a way to filter out elements from a list that don't satisfy certain criteria.

Your goal in this exercise is to use filter() to create, from an input list of strings, a new list that contains only strings that have more than 6 characters.

```
In [ ]: # Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas', 'gimli', 'gandalf']

# Use filter() to apply a Lambda function over fellowship: result
result = filter(lambda member: len(member) > 6, fellowship)

# Convert result to a list: result_list
result_list = list(result)

# Convert result into a list and print it
print(result_list)

['samwise', 'aragorn', 'boromir', 'legolas', 'gandalf']
```

## Reduce() and lambda functions

You're getting very good at using lambda functions! Here's one more function to add to your repertoire of skills. The reduce() function is useful for performing some computation on a list and, unlike map() and filter(), returns a single value as a result. To use reduce(), you must import it from the functools module.

Remember gibberish() from a few exercises back?

```
# Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""
    hodgepodge = ''
    for word in args:
        hodgepodge += word
    return hodgepodge
```

gibberish() simply takes a list of strings as an argument and returns, as a single-value result, the concatenation of all of these strings. In this exercise, you will replicate this functionality by using reduce() and a lambda function that concatenates strings together.

```
In [ ]: # Import reduce from functools
from functools import reduce

# Create a list of strings: stark
stark = ['robb', 'sansa', 'arya', 'brandon', 'rickon']

# Use reduce() to apply a Lambda function over stark: result
result = reduce(lambda item1, item2 : item1 + item2, stark)

# Print the result
print(result)
```

robbsansaaryabrandonrickon

## Error handling with try-except

A good practice in writing your own functions is also anticipating the ways in which other people (or yourself, if you accidentally misuse your own function) might use the function you defined.

As in the previous exercise, you saw that the len() function is able to handle input arguments such as strings, lists, and tuples, but not int type ones and raises an appropriate error and error message when it encounters invalid input arguments. One way of doing this is through exception handling with the try-except block.

In this exercise, you will define a function as well as use a try-except block for handling cases when incorrect input arguments are passed to the function.

Recall the shout\_echo() function you defined in previous exercises; parts of the function definition are provided in the sample code. Your goal is to complete the exception handling code in the function definition and provide an appropriate error message when raising an error.

```
In [ ]: # Define shout_echo
def shout_echo(word1, echo=1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""
    # Initialize empty strings: echo_word, shout_words
    echo_word = ""
    shout_words = ""

    # Add exception handling with try-except
    try:
        # Concatenate echo copies of word1 using *: echo_word
        ...
```

```
echo_word = echo * word1

    # Concatenate '!!!!' to echo_word: shout_words
    shout_words = echo_word + '!!!!'
except:
    # Print error message
    print("word1 must be a string and echo must be an integer.")

# Return shout_words
return shout_words

# Call shout_echo
shout_echo("particle", echo="accelerator")
```

word1 must be a string and echo must be an integer.

Out[ ]:

## Error handling by raising an error

Another way to raise an error is by using `raise`. In this exercise, you will add a `raise` statement to the `shout_echo()` function you defined before to raise an error message when the value supplied by the user to the `echo` argument is less than 0.

The call to `shout_echo()` uses valid argument values. To test and see how the `raise` statement works, simply change the value for the `echo` argument to a negative value. Don't forget to change it back to valid values to move on to the next exercise!

In [ ]:

```
# Define shout_echo
def shout_echo(word1, echo=1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Raise an error with raise
    if echo < 0:
        raise ValueError('echo must be greater than 0')

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Concatenate '!!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!!'

    # Return shout_word
    return shout_word

# Call shout_echo
shout_echo("particle", echo=5)
```

```
Out[ ]: 'particleparticleparticleparticleparticle!!!'
```

## Bringing it all together (1)

This is awesome! You have now learned how to write anonymous functions using lambda, how to pass lambda functions as arguments to other functions such as map(), filter(), and reduce(), as well as how to write errors and output custom error messages within your functions. You will now put together these learnings to good use by working with a Twitter dataset. Before practicing your new error handling skills,in this exercise, you will write a lambda function and use filter() to select retweets, that is, tweets that begin with the string 'RT'.

```
In [ ]: # Select retweets from the Twitter DataFrame: result
result = filter(lambda x: x[0:2] == 'RT', df['text'])

# Create list from filter object result: res_list
res_list = list(result)

# Print all retweets in res_list
for tweet in res_list:
    print(tweet)
```

RT @bpolitics: .@krollbondrating's Christopher Whalen says Clinton is the weakest Dem candidate in 50 years <https://t.co/pLk7rvoR>  
Sn <https://t.co/7QCFz9ehNe>

RT @HeidiAlpine: @dmartosko Cruz video found.....racing from the scene.... #cruzsexscandal <https://t.co/zuAPZfQDk3>

RT @AlanLohner: The anti-American D.C. elites despise Trump for his America-first foreign policy. Trump threatens their gravy train. <https://t.co/7QCFz9ehNe>

RT @BIackPplTweets: Young Donald trump meets his neighbor <https://t.co/RFlu17Z1eE>

RT @trumpresearch: @WaitingInBagdad @thehill Trump supporters have selective amnesia.

RT @HouseCracka: 29,000+ PEOPLE WATCHING TRUMP LIVE ON ONE STREAM!!!

<https://t.co/7QCFz9ehNe>

RT @urfavandtrump: RT for Brendon Urie  
Fav for Donald Trump <https://t.co/PZ5vS94l0g>

RT @trapgrampa: This is how I see #Trump every time he speaks. <https://t.co/fYSiHNS0nT>

RT @trumpresearch: @WaitingInBagdad @thehill Trump supporters have selective amnesia.

RT @Pjw20161951: NO KIDDING: #SleazyDonald just attacked Scott Walker for NOT RAISING TAXES in WI! #LyinTrump  
#NeverTrump #CruzCrew <https://t.co/7QCFz9ehNe>

RT @urfavandtrump: RT for Brendon Urie  
Fav for Donald Trump <https://t.co/PZ5vS94l0g>

RT @ggreenwald: The media spent all day claiming @SusanSarandon said she might vote for Trump. A total fabrication, but whatever... <https://t.co/7QCFz9ehNe>

RT @Pjw20161951: NO KIDDING: #SleazyDonald just attacked Scott Walker for NOT RAISING TAXES in WI! #LyinTrump  
#NeverTrump #CruzCrew <https://t.co/7QCFz9ehNe>

RT @trapgrampa: This is how I see #Trump every time he speaks. <https://t.co/fYSiHNS0nT>

RT @mitchellvii: So let me get this straight. Any reporter can assault Mr Trump at any time and Corey can do nothing? Michelle is clearly!

RT @paulbenedict7: How #Trump Sacks RINO Strongholds by Hitting Positions Held by Dems and GOP <https://t.co/D7ulnAJhis> #tcot #PJNET <https://t.co/7QCFz9ehNe>

RT @DRUDGE\_REPORT: VIDEO: Trump emotional moment with Former Miss Wisconsin who has terminal illness... <https://t.co/qt06aG9int>

RT @ggreenwald: The media spent all day claiming @SusanSarandon said she might vote for Trump. A total fabrication, but whatever... <https://t.co/7QCFz9ehNe>

RT @DennisApgar: Thank God I seen Trump at first stop in Wisconsin media doesn't know how great he is, advice watch live streaming <https://t.co/7QCFz9ehNe>

RT @paulbenedict7: How #Trump Sacks RINO Strongholds by Hitting Positions Held by Dems and GOP <https://t.co/D7ulnAJhis> #tcot #PJNET <https://t.co/7QCFz9ehNe>

RT @DRUDGE\_REPORT: VIDEO: Trump emotional moment with Former Miss Wisconsin who has terminal illness... <https://t.co/qt06aG9int>

RT @DennisApgar: Thank God I seen Trump at first stop in Wisconsin media doesn't know how great he is, advice watch live streaming <https://t.co/7QCFz9ehNe>

RT @mitchellvii: So let me get this straight. Any reporter can assault Mr Trump at any time and Corey can do nothing? Michelle is clearly!

RT @sciam: Trump's idiosyncratic patterns of speech are why people tend either to love or hate him <https://t.co/QXwquVgs3c> <https://t.co/P9N>

RT @Norsu2: Nightmare WI poll for Ted Cruz has Kasich surging: Trump 29, Kasich 27, Cruz 25. <https://t.co/lJsgbLYY1P> #NeverTrump

RT @thehill: WATCH: Protester pepper-sprayed point blank at Trump rally <https://t.co/B5f65A19ld> <https://t.co/skAfByXuQc>

RT @sciam: Trump's idiosyncratic patterns of speech are why people tend either to love or hate him <https://t.co/QXwquVgs3c> <https://t.co/P9N>

RT @ggreenwald: The media spent all day claiming @SusanSarandon said she might vote for Trump. A total fabrication, but whatever... <https://t.co/7QCFz9ehNe>

RT @DebbieStout5: Wow! Last I checked it was just 12 points & that wasn't more than a day ago. Oh boy Trump ppl might want to rethinkðŸ” httpâ€

RT @tyleroakley: i'm a messy bitch, but at least i'm not voting for trump

RT @vandives: Trump supporters r tired of justice NOT being served. There's no justice anymore. Hardworking Americans get screwed. That's nã€!

RT @AP: BREAKING: Trump vows to stand by campaign manager charged with battery, says he does not discard people.

RT @AP: BREAKING: Trump vows to stand by campaign manager charged with battery, says he does not discard people.

RT @urfavandtrump: RT for Jerrie (Little Mix)

Fav for Donald Trump <https://t.co/nEVxEIw6iG>

RT @urfavandtrump: RT for Jerrie (Little Mix)

Fav for Donald Trump <https://t.co/nEVxEIw6iG>

RT @NoahCRothman: When Walker was fighting for reforms, Trump was defending unions and collective bargaining privileges <https://t.co/e1UWNN>

RT @RedheadAndRight: Report: Secret Service Says Michelle Fields Touched Trump <https://t.co/c5c2sD8V02>

This is the only article you will nã€!

RT @AIIAmericanGirI: VIDEO=&gt; Anti-Trump Protester SLUGS Elderly Trump Supporter in the Face  
<https://t.co/GeEryMDuDY>

RT @NoahCRothman: When Walker was fighting for reforms, Trump was defending unions and collective bargaining privileges <https://t.co/e1UWNN>

RT @JusticeRanger1: @realDonaldTrump @Pudingtane @DanScavino @GOP @infowars @EricTrump

URGENT PUBLIC TRUMP ALERT:

COVERT KILL MEANS https:â€

RT @AIIAmericanGirI: VIDEO=&gt; Anti-Trump Protester SLUGS Elderly Trump Supporter in the Face  
<https://t.co/GeEryMDuDY>

RT @RedheadAndRight: Report: Secret Service Says Michelle Fields Touched Trump <https://t.co/c5c2sD8V02>

This is the only article you will nã€!

RT @JusticeRanger1: @realDonaldTrump @Pudingtane @DanScavino @GOP @infowars @EricTrump

URGENT PUBLIC TRUMP ALERT:

COVERT KILL MEANS https:â€

RT @Schneider\_CM: Trump says nobody had ever heard of executive orders before Obama started signing them. Never heard of the Emancipation Pâ€

RT @RonBasler1: @DavidWhitDennis @realDonaldTrump @tedcruz

CRUZ SCREWS HOOKERS

CRUZ / CLINTON

RT @DonaldsAngel: Former Ms. WI just said that she is terminally ill but because of Trump pageant, her 7 yr. old son has his college education

RT @Schneider\_CM: Trump says nobody had ever heard of executive orders before Obama started signing them. Never heard of the Emancipation Pâ€

RT @DonaldsAngel: Former Ms. WI just said that she is terminally ill but because of Trump pageant, her 7 yr. old son has his college education

RT @Dodarey: @DR8801 @SykesCharlie Charlie, let's see you get a straight "yes" or "no" answer from Cruz a/b being unfaithful to his wife @Tâ€

RT @RonBasler1: @DavidWhitDennis @realDonaldTrump @tedcruz

CRUZ SCREWS HOOKERS

CRUZ / CLINTON

RT @RockCliffOne: Remember when the idea of a diabolical moron holding the world hostage was an idea for a funny movie? #Trump #GOP <https://t.co/â€>

RT @HillaryClinton: "Every day, another Republican bemoans the rise of Donald Trump... but [he] didn't come out of nowhere." à €"Hillary  
https://t.co/1qfjwvZC

RT @Dodarey: @DR8801 @SykesCharlie Charlie, let's see you get a straight "yes" or "no" answer from Cruz a/b being unfaithful to his wife @Tâ€!

RT @HillaryClinton: "Every day, another Republican bemoans the rise of Donald Trump... but [he] didn't come out of nowhere." à €"Hillary  
https://t.co/1qfjwvZC

RT @RockCliffOne: Remember when the idea of a diabolical moron holding the world hostage was an idea for a funny movie? #Trump #GOP https://t.co/1qfjwvZC

RT @immigrant4trump: @immigrant4trump msm, cable news attacking trump all day, from 8am to 10pm today, then the reruns come on, repeating tâ€!

RT @immigrant4trump: @immigrant4trump msm, cable news attacking trump all day, from 8am to 10pm today, then the reruns come on, repeating tâ€!

RT @GlendaJazzey: Donald Trumpâ€™s Campaign Financing Dodge, @rrotunda https://t.co/L8fI14lswG via @VerdictJustia

RT @TUSK81: LOUDER FOR THE PEOPLE IN THE BACK https://t.co/h1PVyNLXzx

RT @loopzoop: Well...put it back https://t.co/8Yb7BDT5VM

RT @claytoncubitt: Stop asking Bernie supporters if theyâ€™ll vote for Hillary against Trump. We got a plan to beat Trump already. Called Berâ€!

RT @akaMaude13: Seriously can't make this up. What a joke. #NeverTrump https://t.co/JkTx6mdRgC

## Bringing it all together (2)

Sometimes, we make mistakes when calling functions - even ones you made yourself. But don't fret! In this exercise, you will improve on your previous work with the count\_entries() function in the last chapter by adding a try-except block to it. This will allow your function to provide a helpful message when the user calls your count\_entries() function but provides a column name that isn't in the DataFrame.

```
In [ ]: # Define count_entries()
def count_entries(df, col_name='lang'):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Add try block
    try:
        # Extract column from DataFrame: col
        col = df[col_name]

        # Iterate over the column in dataframe
        for entry in col:

            # If entry is in cols_count, add 1
            if entry in cols_count.keys():
                cols_count[entry] += 1
            # Else add the entry to cols_count, set the value to 1
            else:
```

```

cols_count[entry] = 1

# Return the cols_count dictionary
return cols_count

# Add except block
except:
    print("The DataFrame does not have a " + col_name + 'column')

# Call count_entries(): result1
result1 = count_entries(df, 'lang')

# Print result1
print(result1)

{'en': 97, 'et': 1, 'und': 2}

```

## Bringing it all together (3)

In the previous exercise, you built on your function `count_entries()` to add a try-except block. This was so that users would get helpful messages when calling your `count_entries()` function and providing a column name that isn't in the DataFrame. In this exercise, you'll instead raise a `ValueError` in the case that the user provides a column name that isn't in the DataFrame.

Once again, for your convenience, pandas has been imported as `pd` and the `'tweets.csv'` file has been imported into the DataFrame `tweets_df`. Parts of the code from your previous work are also provided.

```

In [ ]: # Define count_entries()
def count_entries(df, col_name='lang'):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Raise a ValueError if col_name is NOT in DataFrame
    if col_name not in df.columns:
        raise ValueError('The DataFrame does not have a '+col_name+' column.')

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over the column in DataFrame
    for entry in col:

        # If entry is in cols_count, add 1
        if entry in cols_count.keys():
            cols_count[entry] += 1

```

```
# Else add the entry to cols_count, set the value to 1
else:
    cols_count[entry] = 1

# Return the cols_count dictionary
return cols_count

# Call count_entries(): result1
result1 = count_entries(df)

# Print result1
print(result1)

{'en': 97, 'et': 1, 'und': 2}
```

# Chapter 1 - Using iterators in PythonLand

Here, you'll learn all about iterators and iterables, which you have already worked with before when writing for loops! You'll learn about some very useful functions that will allow you to effectively work with iterators and finish the chapter with a use case that is pertinent to the world of Data Science - dealing with large amounts of data - in this case, data from Twitter that you will load in chunks using iterators!

## Iterators vs Iterables

An iterable is an object that can return an iterator, while an iterator is an object that keeps state and produces the next value when you call next() on it.

## Iterating over iterables (1)

Great, you're familiar with what iterables and iterators are! In this exercise, you will reinforce your knowledge about these by iterating over and printing from iterables and iterators.

You are provided with a list of strings `flash`. You will practice iterating over the list by using a for loop. You will also create an iterator for the list and access the values from the iterator.

```
In [ ]: # Create a list of strings: flash
flash = ['jay garrick', 'barry allen', 'wally west', 'bart allen']

# Print each list item in flash using a for loop
for person in flash:
    print(person)

# Create an iterator for flash: superspeed
superspeed = iter(flash)

# Print each item from the iterator
print(next(superspeed))
print(next(superspeed))
print(next(superspeed))
print(next(superspeed))
```

```
jay garrick  
barry allen  
wally west  
bart allen  
jay garrick  
barry allen  
wally west  
bart allen
```

```
In [ ]: superspeed
```

```
Out[ ]: <list_iterator at 0x2b36b146560>
```

## Iterating over iterables (2)

One of the things you learned about in this chapter is that not all iterables are actual lists. A couple of examples that we looked at are strings and the use of the range() function. In this exercise, we will focus on the range() function.

You can use range() in a for loop as if it's a list to be iterated over:

```
for i in range(5):  
    print(i)
```

Recall that range() doesn't actually create the list; instead, it creates a range object with an iterator that produces the values until it reaches the limit (in the example, until the value 4). If range() created the actual list, calling it with a value of 10100 may not work, especially since a number as big as that may go over a regular computer's memory. The value 10100 is actually what's called a **Googol** which is a 1 followed by a hundred 0s. That's a huge number!

Your task for this exercise is to show that calling range() with 10100 won't actually pre-create the list.

```
In [ ]: range(3)
```

```
Out[ ]: range(0, 3)
```

```
In [ ]: # Create an iterator for range(3): small_value  
small_value = iter(range(3))
```

```
# Print the values in small_value  
print(next(small_value))  
print(next(small_value))  
print(next(small_value))
```

```
# Loop over range(3) and print the values  
for num in range(3):  
    print(num)
```

```
# Create an iterator for range(10 ** 100): googol
googol = iter(range(10**100))

# Print the first 5 values from googol
print(next(googol))
print(next(googol))
print(next(googol))
print(next(googol))
print(next(googol))
```

```
0
1
2
0
1
2
0
1
2
3
4
```

## Iterators as function arguments

You've been using the `iter()` function to get an iterator object, as well as the `next()` function to retrieve the values one by one from the iterator object.

There are also functions that take iterators and iterables as arguments. For example, the `list()` and `sum()` functions return a list and the sum of elements, respectively.

In this exercise, you will use these functions by passing an iterable from `range()` and then printing the results of the function calls.

```
In [ ]: # Create a range object: values
values = range(10,21)

# Print the range object
print(values)

# Create a list of integers: values_list
values_list = list(values)

# Print values_list
print(values_list)

# Get the sum of values: values_sum
values_sum = sum(values)
```

```
# Print values_sum
print(values_sum)

range(10, 21)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
165
```

## Using enumerate

You're really getting the hang of using iterators, great job!

You've just gained several new ideas on iterators and one of them is the `enumerate()` function. `enumerate()` returns an enumerate object that produces a sequence of tuples, and each of the tuples is an index-value pair.

In this exercise, you are given a list of strings `mutants` and you will practice using `enumerate()` on it by printing out a list of tuples and unpacking the tuples using a for loop.

```
In [ ]: # Create a list of strings: mutants
mutants = ['charles xavier', 'bobby drake', 'kurt wagner', 'max eisenhardt', 'kitty pryde']

# Create a list of tuples: mutant_list
mutant_list = list(enumerate(mutants))

# Print the list of tuples
print(mutant_list)

# Unpack and print the tuple pairs
for index1, value1 in enumerate(mutants):
    print(index1, value1)

# Change the start index
for index2, value2 in enumerate(mutants, start = 1):
    print(index2, value2)
```

[(0, 'charles xavier'), (1, 'bobby drake'), (2, 'kurt wagner'), (3, 'max eisenhardt'), (4, 'kitty pryde')]

0 charles xavier  
1 bobby drake  
2 kurt wagner  
3 max eisenhardt  
4 kitty pryde  
1 charles xavier  
2 bobby drake  
3 kurt wagner  
4 max eisenhardt  
5 kitty pryde

## Using zip

Another interesting function is `zip()`, which takes any number of iterables and returns a zip object that is an iterator of tuples. If you wanted to print the values of a zip object, you can convert it into a list and then print it. Printing just a zip object will not return the values unless you unpack it first. In this exercise, you will explore this for yourself.

Three lists of strings are pre-loaded: mutants, aliases, and powers. First, you will use `list()` and `zip()` on these lists to generate a list of tuples. Then, you will create a zip object using `zip()`. Finally, you will unpack this zip object in a for loop to print the values in each tuple. Observe the different output generated by printing the list of tuples, then the zip object, and finally, the tuple values in the for loop.

```
In [ ]: aliases = ['prof x', 'iceman', 'nightcrawler', 'magneto', 'shadowcat']
powers = ['telepathy', 'thermokinesis', 'teleportation', 'magnetokinesis', 'intangibility']
```

```
In [ ]: # Create a list of tuples: mutant_data
mutant_data = list(zip(mutants, aliases, powers))

# Print the list of tuples
print(mutant_data)

# Create a zip object using the three lists: mutant_zip
mutant_zip = zip(mutants, aliases, powers)

# Print the zip object
print(mutant_zip)

# Unpack the zip object and print the tuple values
for value1, value2, value3 in mutant_zip:
    print(value1, value2, value3)
```

```
[('charles xavier', 'prof x', 'telepathy'), ('bobby drake', 'iceman', 'thermokinesis'), ('kurt wagner', 'nightcrawler', 'teleportation'), ('max eisenhardt', 'magneto', 'magnetokinesis'), ('kitty pryde', 'shadowcat', 'intangibility')]
<zip object at 0x000002B36D0E14C0>
charles xavier prof x telepathy
bobby drake iceman thermokinesis
kurt wagner nightcrawler teleportation
max eisenhardt magneto magnetokinesis
kitty pryde shadowcat intangibility
```

## Using \* and zip to 'unzip'

You know how to use `zip()` as well as how to print out values from a zip object. Excellent!

Let's play around with `zip()` a little more. There is no `unzip` function for doing the reverse of what `zip()` does. We can, however, reverse what has been zipped together by using `zip()` with a little help from `*`. `*` unpacks an iterable such as a list or a tuple into positional arguments in a

function call.

In this exercise, you will use \* in a call to zip() to unpack the tuples produced by zip().

Two tuples of strings, mutants and powers have been pre-loaded.

```
In [ ]: mutants = tuple(mutants)
powers = tuple(powers)
```

```
In [ ]: # Create a zip object from mutants and powers: z1
z1 = zip(mutants, powers)

# Print the tuples in z1 by unpacking with *
print(*z1)

# Re-create a zip object from mutants and powers: z1
z1 = zip(mutants, powers)

# 'Unzip' the tuples in z1 by unpacking with * and zip(): result1, result2
result1, result2 = zip(*z1)

# Check if unpacked tuples are equivalent to original tuples
print(result1 == mutants)
print(result2 == powers)

('charles xavier', 'telepathy') ('bobby drake', 'thermokinesis') ('kurt wagner', 'teleportation') ('max eisenhardt', 'magnetokinesis') ('kitty pryde', 'intangibility')
True
True
```

## Processing large amounts of Twitter data

Sometimes, the data we have to process reaches a size that is too much for a computer's memory to handle. This is a common problem faced by data scientists. A solution to this is to process an entire data source chunk by chunk, instead of a single go all at once.

In this exercise, you will do just that. You will process a large csv file of Twitter data in the same way that you processed 'tweets.csv' in Bringing it all together exercises of the prequel course, but this time, working on it in chunks of 10 entries at a time.

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

```
In [ ]: # Initialize an empty dictionary: counts_dict
counts_dict = {}

# Iterate over the file chunk by chunk
```

```
file_path = 'C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp'
for chunk in pd.read_csv(file_path, chunksize=10):

    # Iterate over the column in DataFrame
    for entry in chunk['lang']:
        if entry in counts_dict.keys():
            counts_dict[entry] += 1
        else:
            counts_dict[entry] = 1

    # Print the populated dictionary
    print(counts_dict)

{'en': 97, 'et': 1, 'und': 2}
```

## Extracting information for large amounts of Twitter data

Great job chunking out that file in the previous exercise. You now know how to deal with situations where you need to process a very large file and that's a very useful skill to have!

It's good to know how to process a file in smaller, more manageable chunks, but it can become very tedious having to write and rewrite the same code for the same task each time. In this exercise, you will be making your code more reusable by putting your work in the last exercise in a function definition.

```
In [ ]: # Define count_entries()
def count_entries(csv_file, c_size, colname):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: counts_dict
    counts_dict = {}

    # Iterate over the file chunk by chunk
    for chunk in pd.read_csv(csv_file, chunksize = c_size):

        # Iterate over the column in DataFrame
        for entry in chunk[colname]:
            if entry in counts_dict.keys():
                counts_dict[entry] += 1
            else:
                counts_dict[entry] = 1

    # Return counts_dict
    return counts_dict

# Call count_entries(): result_counts
result_counts = count_entries('tweets.csv', 10, 'lang')
```

```
# Print result_counts  
print(result_counts)
```

```
-----  
FileNotFoundError                                     Traceback (most recent call last)  
Cell In[27], line 23  
  20     return counts_dict  
  21 # Call count_entries(): result_counts  
--> 23 result_counts = count_entries('tweets.csv', 10, 'lang')  
  24 # Print result_counts  
  25 print(result_counts)  
  
Cell In[27], line 10, in count_entries(csv_file, c_size, colname)  
    7 counts_dict = {}  
    8 # Iterate over the file chunk by chunk  
--> 10 for chunk in pd.read_csv(csv_file, chunksize = c_size):  
    11  
    12     # Iterate over the column in DataFrame  
    13     for entry in chunk[colname]:  
    14         if entry in counts_dict.keys():  
  
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:948, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision, storage_options, dtype_backend)  
    935 kwds_defaults = _refine_defaults_read(  
    936     dialect,  
    937     delimiter,  
(...)  
    944     dtype_backend=dtype_backend,  
    945 )  
    946 kwds.update(kwds_defaults)  
--> 948 return _read(filepath_or_buffer, kwds)  
  
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:611, in _read(filepath_or_buffer, kwds)  
   608 _validate_names(kwds.get("names", None))  
   610 # Create the parser.  
--> 611 parser = TextFileReader(filepath_or_buffer, **kwds)  
   613 if chunksize or iterator:  
   614     return parser  
  
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1448, in TextFileReader.__init__(self, f, engine, **kwds)  
  1445     self.options["has_index_names"] = kwds["has_index_names"]  
  1447 self.handles: IOHandles | None = None  
-> 1448 self._engine = self._make_engine(f, self.engine)  
  
File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1705, in TextFileReader._make_engine(self, f, engine)
```

```

1703     if "b" not in mode:
1704         mode += "b"
-> 1705 self.handles = get_handle(
1706     f,
1707     mode,
1708     encoding=self.options.get("encoding", None),
1709     compression=self.options.get("compression", None),
1710     memory_map=self.options.get("memory_map", False),
1711     is_text=is_text,
1712     errors=self.options.get("encoding_errors", "strict"),
1713     storage_options=self.options.get("storage_options", None),
1714 )
1715 assert self.handles is not None
1716 f = self.handles.handle

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\common.py:863, in get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
858 elif isinstance(handle, str):
859     # Check whether the filename is to be opened in binary mode.
860     # Binary mode does not support 'encoding' and 'newline'.
861     if ioargs.encoding and "b" not in ioargs.mode:
862         # Encoding
-> 863         handle = open(
864             handle,
865             ioargs.mode,
866             encoding=ioargs.encoding,
867             errors=errors,
868             newline="",
869         )
870     else:
871         # Binary mode
872         handle = open(handle, ioargs.mode)

FileNotFoundException: [Errno 2] No such file or directory: 'tweets.csv'

```

## Chapter - List comprehensions and generators

In this chapter, you'll build on your knowledge of iterators and be introduced to list comprehensions, which allow you to create complicated lists and lists of lists in one line of code! List comprehensions can dramatically simplify your code and make it more efficient, and will become a vital part of your Python Data Science toolbox. You'll then learn about generators, which are extremely helpful when working with large sequences of data that you may not want to store in memory but instead generate on the fly.

## Writing list comprehensions

Your job in this exercise is to write a list comprehension that produces a list of the squares of the numbers ranging from 0 to 9.

```
In [ ]: # Create List comprehension: squares  
squares = [x**2 for x in range(0,10)]  
squares
```

```
Out[ ]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

## Nested list comprehensions

Great! At this point, you have a good grasp of the basic syntax of list comprehensions. Let's push your code-writing skills a little further. In this exercise, you will be writing a list comprehension within another list comprehension, or nested list comprehensions. It sounds a little tricky, but you can do it!

Let's step aside for a while from strings. One of the ways in which lists can be used are in representing multi-dimension objects such as matrices. Matrices can be represented as a list of lists in Python. For example a  $5 \times 5$  matrix with values 0 to 4 in each row can be written as:

```
matrix = [[0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4]]
```

Your task is to recreate this matrix by using nested listed comprehensions. Recall that you can create one of the rows of the matrix with a single list comprehension. To create the list of lists, you simply have to supply the list comprehension as the output expression of the overall list comprehension:

```
[[output expression] for iterator variable in iterable]
```

Note that here, the output expression is itself a list comprehension.

```
In [ ]: matrix = [[x for x in range(5)] for x in range(5)]
```

```
In [ ]: matrix
```

```
Out[ ]: [[0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4],  
          [0, 1, 2, 3, 4]]
```

```
In [ ]: # Create a 5 x 5 matrix using a list of lists: matrix
matrix = [[col for col in range(0,5)] for row in range(0,5) ]

# Print the matrix
for row in matrix:
    print(row)
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

## Using conditionals in comprehensions (1)

You've been using list comprehensions to build lists of values, sometimes using operations to create these values.

An interesting mechanism in list comprehensions is that you can also create lists with values that meet only a certain condition. One way of doing this is by using conditionals on iterator variables. In this exercise, you will do exactly that!

You can apply a conditional statement to test the iterator variable by adding an if statement in the optional predicate expression part after the for statement in the comprehension:

```
[ output expression for iterator variable in iterable if predicate expression ].
```

You will use this recipe to write a list comprehension for this exercise. You are given a list of strings fellowship and, using a list comprehension, you will create a list that only includes the members of fellowship that have 7 characters or more.

```
In [ ]: # Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

# Create list comprehension: new_fellowship
new_fellowship = [member for member in fellowship if len(member) >= 7]

# Print the new list
print(new_fellowship)

['samwise', 'aragorn', 'legolas', 'boromir']
```

## Using conditionals in comprehensions (2)

In the previous exercise, you used an if conditional statement in the predicate expression part of a list comprehension to evaluate an iterator variable. In this exercise, you will use an if-else statement on the output expression of the list.

You will work on the same list, fellowship and, using a list comprehension and an if-else conditional statement in the output expression, create a list that keeps members of fellowship with 7 or more characters and replaces others with an empty string. Use member as the iterator variable in the list comprehension.

```
In [ ]: # Create List comprehension: new_fellowship
new_fellowship = [member if len(member) >= 7 else "" for member in fellowship]

# Print the new list
print(new_fellowship)

['', 'samwise', '', 'aragorn', 'legolas', 'boromir', '']
```

## Dict comprehensions

Comprehensions aren't relegated merely to the world of lists. There are many other objects you can build using comprehensions, such as dictionaries, pervasive objects in Data Science. You will create a dictionary using the comprehension syntax for this exercise. In this case, the comprehension is called a `dict comprehension`.

The main difference between a list comprehension and a dict comprehension is the use of curly braces {} instead of []. Additionally, members of the dictionary are created using a colon : as in : .

You are given a list of strings fellowship and, using a dict comprehension, create a dictionary with the members of the list as the keys and the length of each string as the corresponding values.

```
In [ ]: # Create dict comprehension: new_fellowship
new_fellowship = {member : len(member) for member in fellowship}

# Print the new list
print(new_fellowship)

{'frodo': 5, 'samwise': 7, 'merry': 5, 'aragorn': 7, 'legolas': 7, 'boromir': 7, 'gimli': 5}
```

## List comprehensions vs generators

list comprehensions and generator expressions look very similar in their syntax, except for the use of parentheses () in generator expressions and brackets [] in list comprehensions.

```
# Generator expression
fellow2 = (member for member in fellowship if len(member) >= 7)
```

A list comprehension produces a list as output, a generator produces a generator object.

## Write your own generator expressions

You are familiar with what generators and generator expressions are, as well as its difference from list comprehensions. In this exercise, you will practice building generator expressions on your own.

Recall that generator expressions basically have the same syntax as list comprehensions, except that it uses parentheses () instead of brackets []; this should make things feel familiar! Furthermore, if you have ever iterated over a dictionary with .items(), or used the range() function, for example, you have already encountered and used generators before, without knowing it! When you use these functions, Python creates generators for you behind the scenes.

Now, you will start simple by creating a generator object that produces numeric values.

```
In [ ]: # Create generator object: result
result = ( num for num in range(0,31))

# Print the first 5 values
print(next(result))
print(next(result))
print(next(result))
print(next(result))
print(next(result))

# Print the rest of the values
for value in result:
    print(value)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30
```

```
In [ ]: result
```

```
Out[ ]: <generator object <genexpr> at 0x01354E70>
```

## Changing the output in generator expressions

Great! At this point, you already know how to write a basic generator expression. In this exercise, you will push this idea a little further by adding to the output expression of a generator expression. Because generator expressions and list comprehensions are so alike in syntax, this should be a familiar task for you!

You are given a list of strings `lannister` and, using a generator expression, create a generator object that you will iterate over to print its values.

```
In [ ]: # Create a list of strings: Lannister  
lannister = ['cersei', 'jaime', 'tywin', 'tyrion', 'joffrey']
```

```
# Create a generator object: lengths
lengths = (len(person) for person in lannister)

# Iterate over and print the values in lengths
for value in lengths:
    print(value)
```

```
6
5
5
6
7
```

## Build a generator

In previous exercises, you've dealt mainly with writing generator expressions, which uses comprehension syntax. Being able to use comprehension syntax for generator expressions made your work so much easier!

Not only are there generator expressions, there are generator functions as well. Generator functions are functions that, like generator expressions, yield a series of values, instead of returning a single value. A generator function is defined as you do a regular function, but whenever it generates a value, it uses the keyword `yield` instead of `return`.

In this exercise, you will create a generator function with a similar mechanism as the generator expression you defined in the previous exercise

```
In [ ]: # Define generator function get_lengths
def get_lengths(input_list):
    """Generator function that yields the
    length of the strings in input_list."""

    # Yield the length of a string
    for person in input_list:
        yield len(person)

    # Print the values generated by get_lengths()
    for value in get_lengths(lannister):
        print(value)
```

```
6
5
5
6
7
```

## List comprehensions for time-stamped data

You will now make use of what you've learned from this chapter to solve a simple data extraction problem. You will also be introduced to a data structure, the pandas Series, in this exercise. We won't elaborate on it much here, but what you should know is that it is a data structure that you will be working with a lot of times when analyzing data from pandas DataFrames. You can think of DataFrame columns as single-dimension arrays called Series.

In this exercise, you will be using a list comprehension to extract the time from time-stamped Twitter data.

```
In [ ]: df = pd.read_csv('tweets.csv')
```

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

				contributors	coordinates	created_at	entities	extended_entities	favorite_count	favorited	filter_level	geo	id	...	quoted_st
0						Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [{'screen_na...']}	{'media': [{sizes': 'large': {'w': 1024, 'h':...}}	0	False	low	NaN	714960401759387648	...	
1						Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [{'text': 'cruzsexscandal', 'indi...']}	{'media': [{sizes': 'large': {'w': 500, 'h':...}}	0	False	low	NaN	714960401977319424	...	
2						Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [], 'symbols...']}	NaN	0	False	low	NaN	714960402426236928	...	
3						Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [], 'symbols...')}	NaN	0	False	low	NaN	714960402367561730	...	
4						Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [{'screen_na...']}	NaN	0	False	low	NaN	714960402149416960	...	

5 rows × 31 columns

```
In [ ]: # Extract the created_at column from df: tweet_time  
tweet_time = df['created_at']  
  
# Extract the clock time: tweet_clock_time  
tweet_clock_time = [entry[11:19] for entry in tweet_time]  
  
# Print the extracted times  
print(tweet_clock_time)
```

## Conditional list comprehensions for time-stamped data

Great, you've successfully extracted the data of interest, the time, from a pandas DataFrame! Let's tweak your work further by adding a conditional that further specifies which entries to select.

In this exercise, you will be using a list comprehension to extract the time from time-stamped Twitter data. You will add a conditional expression to the list comprehension so that you only select the times in which entry[17:19] is equal to '19'.

```
In [ ]: # Extract the clock time: tweet_clock_time  
tweet_clock_time = [entry[11:19] for entry in tweet_time if entry[17:19] == '19'
```

```
# Print the extracted time  
print(tweet_clock_time)
```

# Chapter 3 - Bringing it all together!

This chapter will allow you to apply your newly acquired skills towards wrangling and extracting meaningful information from a real-world dataset - the World Bank's World Development Indicators dataset! You'll have the chance to write your own functions and list comprehensions as you work with iterators and generators and solidify your Python Data Science chops. Enjoy!

## Dictionaries for data science

For this exercise, you'll use what you've learned about the `zip()` function and combine two lists into a dictionary.

These lists are actually extracted from a bigger dataset file of world development indicators from the World Bank. For pedagogical purposes, we have pre-processed this dataset into the lists that you'll be working with.

The first list `feature_names` contains header names of the dataset and the second list `row_vals` contains actual values of a row from the dataset, corresponding to each of the header names.

```
In [ ]: row_vals = ['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56090740552298']
feature_names = ['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56090740552298']
```

```
In [ ]: # Zip lists: zipped_lists
zipped_lists = zip(feature_names, row_vals)

# Create a dictionary: rs_dict
rs_dict = dict(zipped_lists)

# Print the dictionary
print(rs_dict)
```

```
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility
rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.56090740552298'}
```

## Writing a function to help you

Suppose you needed to repeat the same process done in the previous exercise to many, many rows of data. Rewriting your code again and again could become very tedious, repetitive, and unmaintainable.

In this exercise, you will create a function to house the code you wrote earlier to make things easier and much more concise. Why? This way, you only need to call the function and supply the appropriate lists to create your dictionaries!

```
In [ ]: # Define lists2dict()
def lists2dict(list1, list2):
    """Return a dictionary where list1 provides
    keys and list2 provides the values"""
```

```
the keys and list2 provides the values."""
```

```
# Zip lists: zipped_lists
zipped_lists = zip(list1, list2)

# Create a dictionary: rs_dict
rs_dict = dict(zipped_lists)

# Return the dictionary
return rs_dict

# Call lists2dict: rs_fxn
rs_fxn = lists2dict(feature_names, row_vals)

# Print rs_fxn
print(rs_fxn)
```

```
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.56090740552298'}
```

## Using a list comprehension

This time, you're going to use the lists2dict() function you defined in the last exercise to turn a bunch of lists into a list of dictionaries with the help of a list comprehension.

The lists2dict() function has already been preloaded, together with a couple of lists, feature\_names and row\_lists. feature\_names contains the header names of the World Bank dataset and row\_lists is a list of lists, where each sublist is a list of actual values of a row from the dataset.

Your goal is to use a list comprehension to generate a list of dicts, where the keys are the header names and the values are the row entries.

```
In [ ]: row_lists = [['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56090740552298']]
```

```
In [ ]: # Print the first two lists in row_lists
print(row_lists[0])
print(row_lists[1])

# Turn list of lists into list of dicts: list_of_dicts
list_of_dicts = [lists2dict(feature_names, sublist) for sublist in row_lists]

# Print the first two dictionaries in list_of_dicts
print(list_of_dicts[0])
print(list_of_dicts[1])
```

```
['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56090740552298']
['Arab World', 'ARB', 'Age dependency ratio (% of working-age population)', 'SP.POP.DPND', '1960', '87.7976011532547']
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility
rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.56090740552298'}
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Age dependency ratio
(% of working-age population)', 'SP.ADO.TFRT': 'SP.POP.DPND', '1960': '1960', '133.56090740552298': '87.7976011532547'}
```

## Turning this all into a DataFrame

You've zipped lists together, created a function to house your code, and even used the function in a list comprehension to generate a list of dictionaries. That was a lot of work and you did a great job!

You will now use of all these to convert the list of dictionaries into a pandas DataFrame. You will see how convenient it is to generate a DataFrame from dictionaries with the `DataFrame()` function from the pandas package.

```
In [ ]: # Turn list of lists into list of dicts: list_of_dicts
list_of_dicts = [lists2dict(feature_names, sublist) for sublist in row_lists]

# Turn list of dicts into a DataFrame: df
df = pd.DataFrame(list_of_dicts)

# Print the head of the DataFrame
df.head()
```

```
Out[ ]:   133.56090740552298  1960  ARB  Adolescent fertility rate (births per 1,000 women ages 15-19)  Arab World  SP.ADO.TFRT
0    133.56090740552298  1960  ARB                               Adolescent fertility rate (births per 1,000 wo...  Arab World  SP.ADO.TFRT
1    87.7976011532547   1960  ARB                         Age dependency ratio (% of working-age populat...  Arab World  SP.POP.DPND
2    6.634579191565161   1960  ARB          Age dependency ratio, old (% of working-age po...  Arab World  SP.POP.DPND.OL
3    81.02332950839141   1960  ARB          Age dependency ratio, young (% of working-age ...  Arab World  SP.POP.DPND.YG
4      3000000.0     1960  ARB                           Arms exports (SIPRI trend indicator values)  Arab World  MS.MIL.XPRT.KD
```

## Processing data in chunks (1)

Sometimes, data sources can be so large in size that storing the entire dataset in memory becomes too resource-intensive. In this exercise, you will process the first 1000 rows of a file line by line, to create a dictionary of the counts of how many times each country appears in a column in the dataset.

The csv file 'world\_ind\_pop\_data.csv' is in your current directory for your use. To begin, you need to open a connection to this file using what is known as a context manager. For example, the command with open('datacamp.csv') as datacamp binds the csv file 'datacamp.csv' as datacamp in the context manager. Here, the with statement is the context manager, and its purpose is to ensure that resources are efficiently allocated when opening a connection to a file.

```
In [ ]: # Open a connection to the file
with open('world_ind_pop_data.csv') as file:

    # Skip the column names
    file.readline()

    # Initialize an empty dictionary: counts_dict
    counts_dict = {}

    # Process only the first 1000 rows
    for j in range(0,1000):

        # Split the current line into a list: line
        line = file.readline().split(',')

        # Get the value for the first column: first_col
        first_col = line[0]

        # If the column value is in the dict, increment its value
        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1

        # Else, add to the dict and set value to 1
        else:
            counts_dict[first_col] = 1

    # Print the resulting dictionary
    print(counts_dict)
```

{'Arab World': 5, 'Caribbean small states': 5, 'Central Europe and the Baltics': 5, 'East Asia & Pacific (all income levels)': 5, 'East Asia & Pacific (developing only)': 5, 'Euro area': 5, 'Europe & Central Asia (all income levels)': 5, 'Europe & Central Asia (developing only)': 5, 'European Union': 5, 'Fragile and conflict affected situations': 5, 'Heavily indebted poor countries (HIPC)': 5, 'High income': 5, 'High income: nonOECD': 5, 'High income: OECD': 5, 'Latin America & Caribbean (all income levels)': 5, 'Latin America & Caribbean (developing only)': 5, 'Least developed countries: UN classification': 5, 'Low & middle income': 5, 'Low income': 5, 'Lower middle income': 5, 'Middle East & North Africa (all income levels)': 5, 'Middle East & North Africa (developing only)': 5, 'Middle income': 5, 'North America': 5, 'OECD members': 5, 'Other small states': 5, 'Pacific island small states': 5, 'Small states': 5, 'South Asia': 5, 'Sub-Saharan Africa (all income levels)': 5, 'Sub-Saharan Africa (developing only)': 5, 'Upper middle income': 5, 'World': 4, 'Afghanistan': 4, 'Albania': 4, 'Algeria': 4, 'American Samoa': 4, 'Andorra': 4, 'Angola': 4, 'Antigua and Barbuda': 4, 'Argentina': 4, 'Armenia': 4, 'Aruba': 4, 'Australia': 4, 'Austria': 4, 'Azerbaijan': 4, 'Bahamas': 4, 'Bahrain': 4, 'Bangladesh': 4, 'Barbados': 4, 'Belarus': 4, 'Belgium': 4, 'Belize': 4, 'Benin': 4, 'Bermuda': 4, 'Bhutan': 4, 'Bolivia': 4, 'Bosnia and Herzegovina': 4, 'Botswana': 4, 'Brazil': 4, 'Brunei Darussalam': 4, 'Bulgaria': 4, 'Burkina Faso': 4, 'Burundi': 4, 'Cabo Verde': 4, 'Cambodia': 4, 'Cameroon': 4, 'Canada': 4, 'Cayman Islands': 4, 'Central African Republic': 4, 'Chad': 4, 'Channel Islands': 4, 'Chile': 4, 'China': 4, 'Colombia': 4, 'Comoros': 4, '"Congo": 8, 'Costa Rica': 4, "Cote d'Ivoire": 4, 'Croatia': 4, 'Cuba': 4, 'Curacao': 4, 'Czech Republic': 4, 'Denmark': 4, 'Djibouti': 4, 'Dominica': 4, 'Dominican Republic': 4, 'Ecuador': 4, '"Egypt": 4, 'El Salvador': 4, 'Equatorial Guinea': 4, 'Eritrea': 4, 'Estonia': 4, 'Ethiopia': 4, 'Faeroe Islands': 4, 'Fiji': 4, 'Finland': 4, 'France': 4, 'French Polynesia': 4, 'Gabon': 4, '"Gambia": 4, 'Georgia': 4, 'Germany': 4, 'Ghana': 4, 'Greece': 4, 'Greenland': 4, 'Grenada': 4, 'Guam': 4, 'Guatemala': 4, 'Guinea': 4, 'Guinea-Bissau': 4, 'Guyana': 4, 'Haiti': 4, 'Honduras': 4, '"Hong Kong SAR": 4, 'Hungary': 4, 'Iceland': 4, 'India': 4, 'Indonesia': 4, '"Iran": 4, 'Iraq': 4, 'Ireland': 4, 'Isle of Man': 4, 'Israel': 4, 'Italy': 4, 'Jamaica': 4, 'Japan': 4, 'Jordan': 4, 'Kazakhstan': 4, 'Kenya': 4, 'Kiribati': 4, '"Korea": 8, 'Kuwait': 4, 'Kyrgyz Republic': 4, 'Lao PDR': 4, 'Latvia': 4, 'Lebanon': 4, 'Lesotho': 4, 'Liberia': 4, 'Libya': 4, 'Liechtenstein': 4, 'Lithuania': 4, 'Luxembourg': 4, '"Macao SAR": 4, '"Macedonia": 4, 'Madagascar': 4, 'Malawi': 4, 'Malaysia': 4, 'Maldives': 4, 'Mali': 4, 'Malta': 4, 'Marshall Islands': 4, 'Mauritania': 4, 'Mauritius': 4, 'Mexico': 4, '"Micronesia": 4, 'Moldova': 4, 'Monaco': 4, 'Mongolia': 4, 'Montenegro': 4, 'Morocco': 4, 'Mozambique': 4, 'Myanmar': 4, 'Namibia': 4, 'Nepal': 4, 'Netherlands': 4, 'New Caledonia': 4, 'New Zealand': 4, 'Nicaragua': 4, 'Niger': 4, 'Nigeria': 4, 'Northern Mariana Islands': 4, 'Norway': 4, 'Oman': 4, 'Pakistan': 4, 'Palau': 4, 'Panama': 4, 'Papua New Guinea': 4, 'Paraguay': 4, 'Peru': 4, 'Philippines': 4, 'Poland': 4, 'Portugal': 4, 'Puerto Rico': 4, 'Qatar': 4, 'Romania': 4, 'Russian Federation': 4, 'Rwanda': 4, 'Samoa': 4, 'San Marino': 4, 'Sao Tome and Principe': 4, 'Saudi Arabia': 4, 'Senegal': 4, 'Seychelles': 4, 'Sierra Leone': 4, 'Singapore': 4, 'Slovak Republic': 4, 'Slovenia': 4, 'Solomon Islands': 4, 'Somalia': 4, 'South Africa': 4, 'South Sudan': 4, 'Spain': 4, 'Sri Lanka': 4, 'St. Kitts and Nevis': 4, 'St. Lucia': 4, 'St. Vincent and the Grenadines': 4, 'Sudan': 4, 'Suriname': 4, 'Swaziland': 4, 'Sweden': 4, 'Switzerland': 4, 'Syrian Arab Republic': 4, 'Tajikistan': 4, 'Tanzania': 4, 'Thailand': 4, 'Timor-Leste': 4, 'Togo': 4, 'Tonga': 4, 'Trinidad and Tobago': 4, 'Tunisia': 4, 'Turkey': 4, 'Turkmenistan': 4, 'Turks and Caicos Islands': 4, 'Tuvalu': 4, 'Uganda': 4, 'Ukraine': 4, 'United Arab Emirates': 4, 'United Kingdom': 4, 'United States': 4, 'Uruguay': 4, 'Uzbekistan': 4, 'Vanuatu': 4, '"Venezuela": 4, 'Vietnam': 4, 'Virgin Islands (U.S.)': 4, '"Yemen": 4, 'Zambia': 4, 'Zimbabwe': 4}

## Writing a generator to load data in chunks (2)

In the previous exercise, you processed a file line by line for a given number of lines. What if, however, you want to do this for the entire file?

In this case, it would be useful to use generators. Generators allow users to lazily evaluate data. This concept of lazy evaluation is useful when you have to deal with very large datasets because it lets you generate values in an efficient manner by yielding only chunks of data at a time instead of the whole thing at once.

In this exercise, you will define a generator function `read_large_file()` that produces a generator object which yields a single line from a file each time `next()` is called on it. The csv file '`world_ind_pop_data.csv`' is in your current directory for your use.

Note that when you open a connection to a file, the resulting file object is already a generator! So out in the wild, you won't have to explicitly create generator objects in cases such as this. However, for pedagogical reasons, we are having you practice how to do this here with the `read_large_file()` function. Go for it!

```
In [ ]: # Define read_large_file()
def read_large_file(file_object):
    """A generator function to read a large file lazily."""

    # Loop indefinitely until the end of the file
    while True:

        # Read a line from the file: data
        data = file_object.readline()

        # Break if this is the end of the file
        if not data:
            break

        # Yield the line of data
        yield data

# Open a connection to the file
with open('world_ind_pop_data.csv') as file:

    # Create a generator object for the file: gen_file
    gen_file = read_large_file(file)

    # Print the first three lines of the file
    print(next(gen_file))
    print(next(gen_file))
    print(next(gen_file))
```

CountryName,CountryCode,Year,Total Population,Urban population (% of total)

Arab World,ARB,1960,92495902.0,31.285384211605397

Caribbean small states,CSS,1960,4190810.0,31.5974898513652

Wonderful work! Note that since a file object is already a generator, you don't have to explicitly create a generator object with your `read_large_file()` function. However, it is still good to practice how to create generators - well done!

## Writing a generator to load data in chunks (3)

Great! You've just created a generator function that you can use to help you process large files.

Now let's use your generator function to process the World Bank dataset like you did previously. You will process the file line by line, to create a dictionary of the counts of how many times each country appears in a column in the dataset. For this exercise, however, you won't process just

1000 rows of data, you'll process the entire dataset!

```
In [ ]: # Initialize an empty dictionary: counts_dict
counts_dict = {}

# Open a connection to the file
with open('world_ind_pop_data.csv') as file:

    # Iterate over the generator from read_large_file()
    for line in read_large_file(file):

        row = line.split(',')
        first_col = row[0]

        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1
        else:
            counts_dict[first_col] = 1

# Print
print(counts_dict)
```

```
{'CountryName': 1, 'Arab World': 55, 'Caribbean small states': 55, 'Central Europe and the Baltics': 55, 'East Asia & Pacific (all income levels)': 55, 'East Asia & Pacific (developing only)': 55, 'Euro area': 55, 'Europe & Central Asia (all income levels)': 55, 'Europe & Central Asia (developing only)': 55, 'European Union': 55, 'Fragile and conflict affected situations': 55, 'Heavily indebted poor countries (HIPC)': 55, 'High income': 55, 'High income: nonOECD': 55, 'High income: OECD': 55, 'Latin America & Caribbean (all income levels)': 55, 'Latin America & Caribbean (developing only)': 55, 'Least developed countries: UN classification': 55, 'Low & middle income': 55, 'Low income': 55, 'Lower middle income': 55, 'Middle East & North Africa (all income levels)': 55, 'Middle East & North Africa (developing only)': 55, 'Middle income': 55, 'North America': 55, 'OECD members': 55, 'Other small states': 55, 'Pacific island small states': 55, 'Small states': 55, 'South Asia': 55, 'Sub-Saharan Africa (all income levels)': 55, 'Sub-Saharan Africa (developing only)': 55, 'Upper middle income': 55, 'World': 55, 'Afghanistan': 55, 'Albania': 55, 'Algeria': 55, 'American Samoa': 55, 'Andorra': 55, 'Angola': 55, 'Antigua and Barbuda': 55, 'Argentina': 55, 'Armenia': 55, 'Aruba': 55, 'Australia': 55, 'Austria': 55, 'Azerbaijan': 55, 'Bahamas': 55, 'Bahrain': 55, 'Bangladesh': 55, 'Barbados': 55, 'Belarus': 55, 'Belgium': 55, 'Belize': 55, 'Benin': 55, 'Bermuda': 55, 'Bhutan': 55, 'Bolivia': 55, 'Bosnia and Herzegovina': 55, 'Botswana': 55, 'Brazil': 55, 'Brunei Darussalam': 55, 'Bulgaria': 55, 'Burkina Faso': 55, 'Burundi': 55, 'Cabo Verde': 55, 'Cambodia': 55, 'Cameroon': 55, 'Canada': 55, 'Cayman Islands': 55, 'Central African Republic': 55, 'Chad': 55, 'Channel Islands': 55, 'China': 55, 'Colombia': 55, 'Comoros': 55, 'Congo': 110, 'Costa Rica': 55, 'Cote d'Ivoire': 55, 'Croatia': 55, 'Cuba': 55, 'Curacao': 55, 'Cyprus': 55, 'Czech Republic': 55, 'Denmark': 55, 'Djibouti': 55, 'Dominica': 55, 'Dominican Republic': 55, 'Ecuador': 55, 'Egypt': 55, 'El Salvador': 55, 'Equatorial Guinea': 55, 'Eritrea': 55, 'Estonia': 55, 'Ethiopia': 55, 'Faeroe Islands': 55, 'Fiji': 55, 'Finland': 55, 'France': 55, 'French Polynesia': 55, 'Gabon': 55, 'Gambia': 55, 'Georgia': 55, 'Germany': 55, 'Ghana': 55, 'Greece': 55, 'Greenland': 55, 'Grenada': 55, 'Guam': 55, 'Guatemala': 55, 'Guinea': 55, 'Guinea-Bissau': 55, 'Guyana': 55, 'Haiti': 55, 'Honduras': 55, 'Hong Kong SAR': 55, 'Hungary': 55, 'Iceland': 55, 'India': 55, 'Indonesia': 55, 'Iran': 55, 'Iraq': 55, 'Ireland': 55, 'Isle of Man': 55, 'Israel': 55, 'Italy': 55, 'Jamaica': 55, 'Japan': 55, 'Jordan': 55, 'Kazakhstan': 55, 'Kenya': 55, 'Kiribati': 55, 'Korea': 110, 'Kuwait': 52, 'Kyrgyz Republic': 55, 'Lao PDR': 55, 'Latvia': 55, 'Lebanon': 55, 'Lesotho': 55, 'Liberia': 55, 'Libya': 55, 'Liechtenstein': 55, 'Lithuania': 55, 'Luxembourg': 55, 'Macao SAR': 55, 'Macedonia': 55, 'Madagascar': 55, 'Malawi': 55, 'Malaysia': 55, 'Maldives': 55, 'Mali': 55, 'Malta': 55, 'Marshall Islands': 55, 'Mauritania': 55, 'Mauritius': 55, 'Mexico': 55, 'Micronesia': 55, 'Moldova': 55, 'Monaco': 55, 'Mongolia': 55, 'Montenegro': 55, 'Morocco': 55, 'Mozambique': 55, 'Myanmar': 55, 'Namibia': 55, 'Nepal': 55, 'Netherlands': 55, 'New Caledonia': 55, 'New Zealand': 55, 'Nicaragua': 55, 'Niger': 55, 'Nigeria': 55, 'Northern Mariana Islands': 55, 'Norway': 55, 'Oman': 55, 'Pakistan': 55, 'Palau': 55, 'Panama': 55, 'Papua New Guinea': 55, 'Paraguay': 55, 'Peru': 55, 'Philippines': 55, 'Poland': 55, 'Portugal': 55, 'Puerto Rico': 55, 'Qatar': 55, 'Romania': 55, 'Russian Federation': 55, 'Rwanda': 55, 'Samoa': 55, 'San Marino': 55, 'Sao Tome and Principe': 55, 'Saudi Arabia': 55, 'Senegal': 55, 'Seychelles': 55, 'Sierra Leone': 55, 'Singapore': 55, 'Slovak Republic': 55, 'Slovenia': 55, 'Solomon Islands': 55, 'Somalia': 55, 'South Africa': 55, 'South Sudan': 55, 'Spain': 55, 'Sri Lanka': 55, 'St. Kitts and Nevis': 55, 'St. Lucia': 55, 'St. Vincent and the Grenadines': 55, 'Sudan': 55, 'Suriname': 55, 'Swaziland': 55, 'Sweden': 55, 'Switzerland': 55, 'Syrian Arab Republic': 55, 'Tajikistan': 55, 'Tanzania': 55, 'Thailand': 55, 'Timor-Leste': 55, 'Togo': 55, 'Tonga': 55, 'Trinidad and Tobago': 55, 'Tunisia': 55, 'Turkey': 55, 'Turkmenistan': 55, 'Turks and Caicos Islands': 55, 'Tuvalu': 55, 'Uganda': 55, 'Ukraine': 55, 'United Arab Emirates': 55, 'United Kingdom': 55, 'United States': 55, 'Uruguay': 55, 'Uzbekistan': 55, 'Vanuatu': 55, 'Venezuela': 55, 'Vietnam': 55, 'Virgin Islands (U.S.)': 55, 'Yemen': 55, 'Zambia': 55, 'Zimbabwe': 55, 'Serbia': 25, 'West Bank and Gaza': 25, 'Sint Maarten (Dutch part)': 17}
```

## Writing an iterator to load data in chunks (1)

Another way to read data too large to store in memory in chunks is to read the file in as DataFrames of a certain length, say, 100. For example, with the pandas package (imported as pd), you can do pd.read\_csv(filename, chunksize=100). This creates an iterable reader object, which means that you can use next() on it.

In this exercise, you will read a file in small DataFrame chunks with read\_csv(). You're going to use the World Bank Indicators data 'ind\_pop.csv', available in your current directory, to look at the urban population indicator for numerous countries and years.

In [ ]:

```
# Initialize reader object: df_reader
df_reader = pd.read_csv('world_ind_pop_data.csv', chunksize = 10)

# Print two chunks
print(next(df_reader))
print(next(df_reader))
```

	CountryName	CountryCode	Year	\
0	Arab World	ARB	1960	
1	Caribbean small states	CSS	1960	
2	Central Europe and the Baltics	CEB	1960	
3	East Asia & Pacific (all income levels)	EAS	1960	
4	East Asia & Pacific (developing only)	EAP	1960	
5	Euro area	EMU	1960	
6	Europe & Central Asia (all income levels)	ECS	1960	
7	Europe & Central Asia (developing only)	ECA	1960	
8	European Union	EUU	1960	
9	Fragile and conflict affected situations	FCS	1960	
Total Population Urban population (% of total)				
0	9.249590e+07	31.285384		
1	4.190810e+06	31.597490		
2	9.140158e+07	44.507921		
3	1.042475e+09	22.471132		
4	8.964930e+08	16.917679		
5	2.653965e+08	62.096947		
6	6.674890e+08	55.378977		
7	1.553174e+08	38.066129		
8	4.094985e+08	61.212898		
9	1.203546e+08	17.891972		
CountryName CountryCode Year \				
10	Heavily indebted poor countries (HIPC)	HPC	1960	
11	High income	HIC	1960	
12	High income: nonOECD	NOC	1960	
13	High income: OECD	OEC	1960	
14	Latin America & Caribbean (all income levels)	LCN	1960	
15	Latin America & Caribbean (developing only)	LAC	1960	
16	Least developed countries: UN classification	LDC	1960	
17	Low & middle income	LMY	1960	
18	Low income	LIC	1960	
19	Lower middle income	LMC	1960	
Total Population Urban population (% of total)				
10	1.624912e+08	12.236046		
11	9.075975e+08	62.680332		
12	1.866767e+08	56.107863		
13	7.209208e+08	64.285435		
14	2.205642e+08	49.284688		
15	1.776822e+08	44.863308		
16	2.410728e+08	9.616261		
17	2.127373e+09	21.272894		
18	1.571884e+08	11.498396		
19	9.429116e+08	19.810513		

## Writing an iterator to load data in chunks (2)

In the previous exercise, you used `read_csv()` to read in DataFrame chunks from a large dataset. In this exercise, you will read in a file using a bigger DataFrame chunk size and then process the data from the first chunk.

To process the data, you will create another DataFrame composed of only the rows from a specific country. You will then zip together two of the columns from the new DataFrame, 'Total Population' and 'Urban population (% of total)'. Finally, you will create a list of tuples from the zip object, where each tuple is composed of a value from each of the two columns mentioned.

In [ ]:

```
# Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize=1000)

# Get the first DataFrame chunk: df_urb_pop
df_urb_pop = next(urb_pop_reader)

# Check out the head of the DataFrame
print(df_urb_pop.head())

# Check out specific country: df_pop_ceb
df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']

# Zip DataFrame columns of interest: pops
pops = zip(df_pop_ceb['Total Population'], df_pop_ceb['Urban population (% of total)'])

# Turn zip object into list: pops_list
pops_list = list(pops)

# Print pops_list
print(pops_list)
```

```
CountryName CountryCode Year \
0 Arab World ARB 1960
1 Caribbean small states CSS 1960
2 Central Europe and the Baltics CEB 1960
3 East Asia & Pacific (all income levels) EAS 1960
4 East Asia & Pacific (developing only) EAP 1960
```

```
Total Population Urban population (% of total)
0 9.249590e+07 31.285384
1 4.190810e+06 31.597490
2 9.140158e+07 44.507921
3 1.042475e+09 22.471132
4 8.964930e+08 16.917679
[(91401583.0, 44.5079211390026), (92237118.0, 45.206665319194), (93014890.0, 45.866564696018), (93845749.0, 46.5340927663649), (94722599.0, 47.2087429803526)]
```

## Writing an iterator to load data in chunks (3)

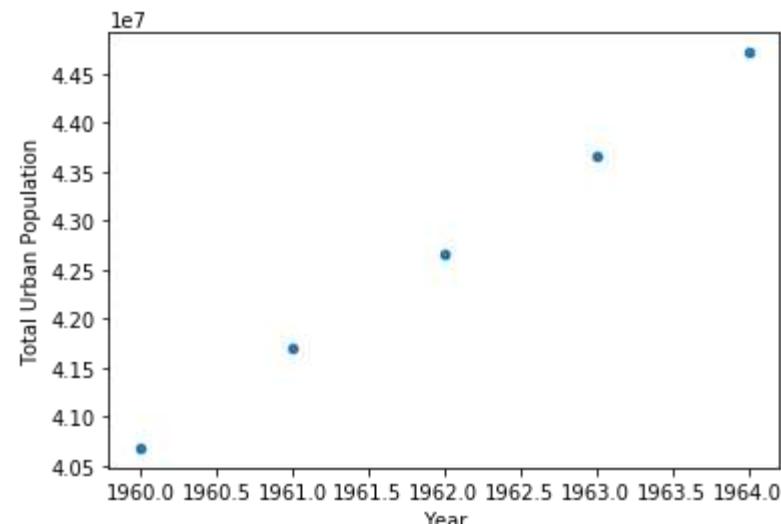
You're getting used to reading and processing data in chunks by now. Let's push your skills a little further by adding a column to a DataFrame.

Starting from the code of the previous exercise, you will be using a list comprehension to create the values for a new column 'Total Urban Population' from the list of tuples that you generated earlier. Recall from the previous exercise that the first and second elements of each tuple consist of, respectively, values from the columns 'Total Population' and 'Urban population (% of total)'. The values in this new column 'Total Urban Population', therefore, are the product of the first and second element in each tuple. Furthermore, because the 2nd element is a percentage, you need to divide the entire result by 100, or alternatively, multiply it by 0.01.

You will also plot the data from this new column to create a visualization of the urban population data.

```
In [ ]: import warnings  
warnings.filterwarnings('ignore')
```

```
In [ ]: # Code from previous exercise  
urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize=1000)  
df_urb_pop = next(urb_pop_reader)  
df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']  
pops = zip(df_pop_ceb['Total Population'],  
           df_pop_ceb['Urban population (% of total)'])  
pops_list = list(pops)  
  
# Use List comprehension to create new DataFrame column 'Total Urban Population'  
df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in pops_list]  
  
# Plot urban population data  
df_pop_ceb.plot(kind='scatter', x='Year', y='Total Urban Population')  
plt.show()
```



## Writing an iterator to load data in chunks (4)

In the previous exercises, you've only processed the data from the first DataFrame chunk. This time, you will aggregate the results over all the DataFrame chunks in the dataset. This basically means you will be processing the entire dataset now. This is neat because you're going to be able to process the entire large dataset by just working on smaller pieces of it!

```
In [ ]: # Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize=1000)

# Initialize empty DataFrame: data
data = pd.DataFrame()

# Iterate over each DataFrame chunk
for df_urb_pop in urb_pop_reader:

    # Check out specific country: df_pop_ceb
    df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']

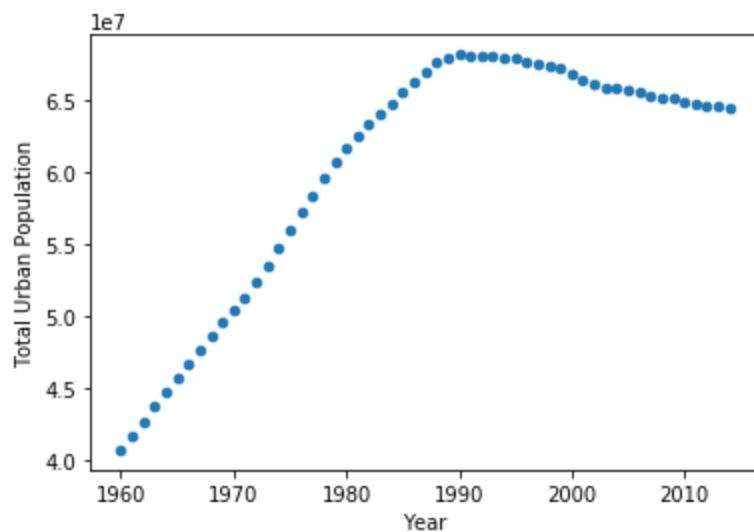
    # Zip DataFrame columns of interest: pops
    pops = zip(df_pop_ceb['Total Population'],
               df_pop_ceb['Urban population (% of total)'])

    # Turn zip object into list: pops_list
    pops_list = list(pops)

    # Use list comprehension to create new DataFrame column 'Total Urban Population'
    df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in pops_list]

    # Append DataFrame chunk to data: data
    data = data.append(df_pop_ceb)

# Plot urban population data
data.plot(kind='scatter', x='Year', y='Total Urban Population')
plt.show()
```



## Writing an iterator to load data in chunks (5)

This is the last leg. You've learned a lot about processing a large dataset in chunks. In this last exercise, you will put all the code for processing the data into a single function so that you can reuse the code without having to rewrite the same things all over again.

You're going to define the function `plot_pop()` which takes two arguments: the filename of the file to be processed, and the country code of the rows you want to process in the dataset.

Because all of the previous code you've written in the previous exercises will be housed in `plot_pop()`, calling the function already does the following:

- Loading of the file chunk by chunk,
- Creating the new column of urban population values, and
- Plotting the urban population data.

That's a lot of work, but the function now makes it convenient to repeat the same process for whatever file and country code you want to process and visualize!

After you are done, take a moment to look at the plots and reflect on the new skills you have acquired. The journey doesn't end here! If you have enjoyed working with this data, you can continue exploring it using the pre-processed version available on [Kaggle](#).

In [ ]:

```
# Define plot_pop()
def plot_pop(filename, country_code):

    # Initialize reader object: urb_pop_reader
    urb_pop_reader = pd.read_csv(filename, chunksize=1000)
```

```
# Initialize empty DataFrame: data
data = pd.DataFrame()

# Iterate over each DataFrame chunk
for df_urb_pop in urb_pop_reader:
    # Check out specific country: df_pop_ceb
    df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == country_code]

    # Zip DataFrame columns of interest: pops
    pops = zip(df_pop_ceb['Total Population'],
               df_pop_ceb['Urban population (% of total)'])

    # Turn zip object into list: pops_list
    pops_list = list(pops)

    # Use list comprehension to create new DataFrame column 'Total Urban Population'
    df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in pops_list]

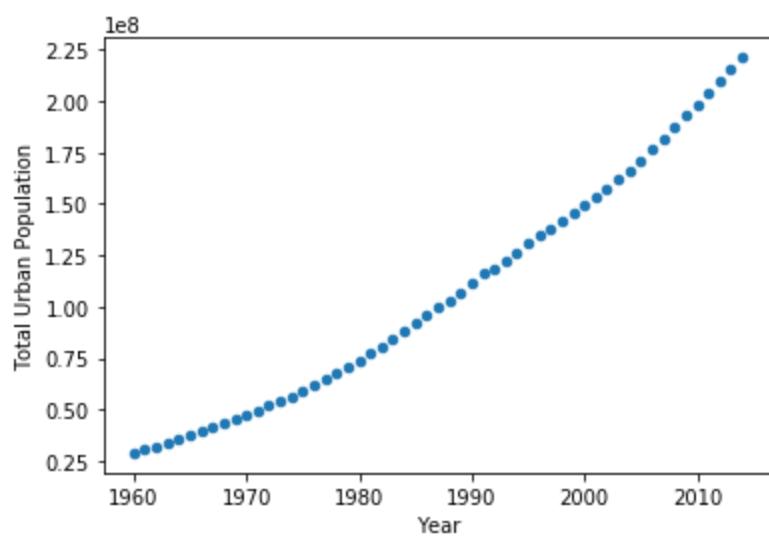
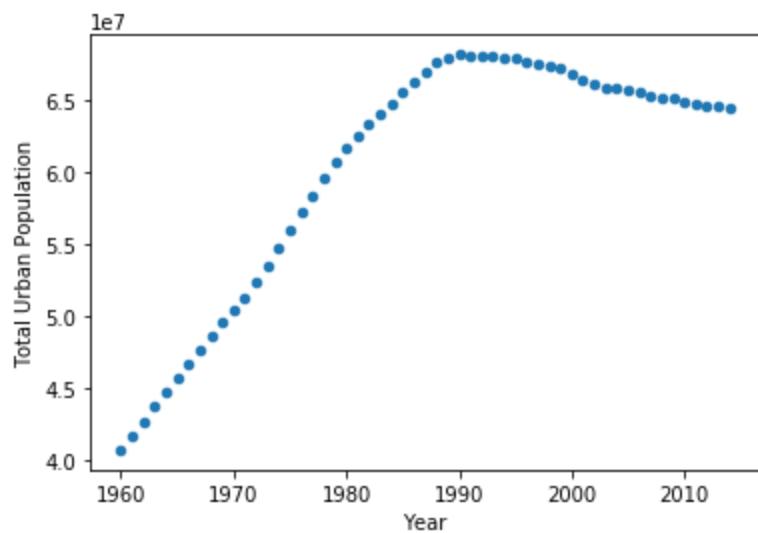
    # Append DataFrame chunk to data: data
    data = data.append(df_pop_ceb)

# Plot urban population data
data.plot(kind='scatter', x='Year', y='Total Urban Population')
plt.show()

# Set the filename: fn
fn = 'world_ind_pop_data.csv'

# Call plot_pop for country code 'CEB'
plot_pop(fn, 'CEB')

# Call plot_pop for country code 'ARB'
plot_pop(fn, 'ARB')
```



# Data Manipulation with pandas

## Transforming Data

Let's master the pandas basics. Learn how to inspect DataFrames and perform fundamental manipulations, including sorting rows, subsetting, and adding new columns.

[Link for reference](#)

## Inspecting a DataFrame

```
In [ ]: # Import pandas using the alias pd
import pandas as pd

#pathway of the file
homelessness = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python

In [ ]: # Print the head of the homelessness data
print(homelessness.head())

      Unnamed: 0         region       state  individuals  family_members \
0            0  East South Central    Alabama        2570.0           864.0
1            1             Pacific     Alaska        1434.0           582.0
2            2            Mountain   Arizona        7259.0          2606.0
3            3  West South Central  Arkansas        2280.0           432.0
4            4             Pacific  California      109008.0          20964.0

      state_pop
0    4887681
1    735139
2    7158024
3    3009733
4    39461588

In [ ]: # Print information about homelessness
print(homelessness.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 51 entries, 0 to 50
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   Unnamed: 0    51 non-null    int64  
 1   region       51 non-null    object  
 2   state        51 non-null    object  
 3   individuals  51 non-null    float64 
 4   family_members 51 non-null    float64 
 5   state_pop    51 non-null    int64  
dtypes: float64(2), int64(2), object(2)
memory usage: 2.5+ KB
None
```

```
In [ ]: # Print the shape of homelessness
print(homelessness.shape)
```

```
(51, 6)
```

```
In [ ]: # Print a description of homelessness
print(homelessness.describe())
```

	Unnamed: 0	individuals	family_members	state_pop
count	51.000000	51.000000	51.000000	5.100000e+01
mean	25.000000	7225.784314	3504.882353	6.405637e+06
std	14.866069	15991.025083	7805.411811	7.327258e+06
min	0.000000	434.000000	75.000000	5.776010e+05
25%	12.500000	1446.500000	592.000000	1.777414e+06
50%	25.000000	3082.000000	1482.000000	4.461153e+06
75%	37.500000	6781.500000	3196.000000	7.340946e+06
max	50.000000	109008.000000	52070.000000	3.946159e+07

## Parts of a DataFrame

```
In [ ]: # Print the values of homelessness
print(homelessness.values)
```

```
[0 'East South Central' 'Alabama' 2570.0 864.0 4887681]
[1 'Pacific' 'Alaska' 1434.0 582.0 735139]
[2 'Mountain' 'Arizona' 7259.0 2606.0 7158024]
[3 'West South Central' 'Arkansas' 2280.0 432.0 3009733]
[4 'Pacific' 'California' 109008.0 20964.0 39461588]
[5 'Mountain' 'Colorado' 7607.0 3250.0 5691287]
[6 'New England' 'Connecticut' 2280.0 1696.0 3571520]
[7 'South Atlantic' 'Delaware' 708.0 374.0 965479]
[8 'South Atlantic' 'District of Columbia' 3770.0 3134.0 701547]
[9 'South Atlantic' 'Florida' 21443.0 9587.0 21244317]
[10 'South Atlantic' 'Georgia' 6943.0 2556.0 10511131]
[11 'Pacific' 'Hawaii' 4131.0 2399.0 1420593]
[12 'Mountain' 'Idaho' 1297.0 715.0 1750536]
[13 'East North Central' 'Illinois' 6752.0 3891.0 12723071]
[14 'East North Central' 'Indiana' 3776.0 1482.0 6695497]
[15 'West North Central' 'Iowa' 1711.0 1038.0 3148618]
[16 'West North Central' 'Kansas' 1443.0 773.0 2911359]
[17 'East South Central' 'Kentucky' 2735.0 953.0 4461153]
[18 'West South Central' 'Louisiana' 2540.0 519.0 4659690]
[19 'New England' 'Maine' 1450.0 1066.0 1339057]
[20 'South Atlantic' 'Maryland' 4914.0 2230.0 6035802]
[21 'New England' 'Massachusetts' 6811.0 13257.0 6882635]
[22 'East North Central' 'Michigan' 5209.0 3142.0 9984072]
[23 'West North Central' 'Minnesota' 3993.0 3250.0 5606249]
[24 'East South Central' 'Mississippi' 1024.0 328.0 2981020]
[25 'West North Central' 'Missouri' 3776.0 2107.0 6121623]
[26 'Mountain' 'Montana' 983.0 422.0 1060665]
[27 'West North Central' 'Nebraska' 1745.0 676.0 1925614]
[28 'Mountain' 'Nevada' 7058.0 486.0 3027341]
[29 'New England' 'New Hampshire' 835.0 615.0 1353465]
[30 'Mid-Atlantic' 'New Jersey' 6048.0 3350.0 8886025]
[31 'Mountain' 'New Mexico' 1949.0 602.0 2092741]
[32 'Mid-Atlantic' 'New York' 39827.0 52070.0 19530351]
[33 'South Atlantic' 'North Carolina' 6451.0 2817.0 10381615]
[34 'West North Central' 'North Dakota' 467.0 75.0 758080]
[35 'East North Central' 'Ohio' 6929.0 3320.0 11676341]
[36 'West South Central' 'Oklahoma' 2823.0 1048.0 3940235]
[37 'Pacific' 'Oregon' 11139.0 3337.0 4181886]
[38 'Mid-Atlantic' 'Pennsylvania' 8163.0 5349.0 12800922]
[39 'New England' 'Rhode Island' 747.0 354.0 1058287]
[40 'South Atlantic' 'South Carolina' 3082.0 851.0 5084156]
[41 'West North Central' 'South Dakota' 836.0 323.0 878698]
[42 'East South Central' 'Tennessee' 6139.0 1744.0 6771631]
[43 'West South Central' 'Texas' 19199.0 6111.0 28628666]
[44 'Mountain' 'Utah' 1904.0 972.0 3153550]
[45 'New England' 'Vermont' 780.0 511.0 624358]
[46 'South Atlantic' 'Virginia' 3928.0 2047.0 8501286]
[47 'Pacific' 'Washington' 16424.0 5880.0 7523869]
[48 'South Atlantic' 'West Virginia' 1021.0 222.0 1804291]
```

```
[49 'East North Central' 'Wisconsin' 2740.0 2167.0 5807406]
[50 'Mountain' 'Wyoming' 434.0 205.0 577601]]
```

```
In [ ]: # Print the column index of homelessness
print(homelessness.columns)
```

```
Index(['Unnamed: 0', 'region', 'state', 'individuals', 'family_members',
       'state_pop'],
      dtype='object')
```

```
In [ ]: # Print the row index of homelessness
print(homelessness.index)
```

```
RangeIndex(start=0, stop=51, step=1)
```

## Sorting rows

```
In [ ]: # Sort homelessness by individuals
homelessness_ind = homelessness.sort_values(["individuals"])

# Print the top few rows
print(homelessness_ind.head())
```

	Unnamed: 0	region	state	individuals	family_members	\
50	50	Mountain	Wyoming	434.0	205.0	
34	34	West North Central	North Dakota	467.0	75.0	
7	7	South Atlantic	Delaware	708.0	374.0	
39	39	New England	Rhode Island	747.0	354.0	
45	45	New England	Vermont	780.0	511.0	

	state_pop
50	577601
34	758080
7	965479
39	1058287
45	624358

```
In [ ]: # Sort homelessness by descending family members
homelessness_fam = homelessness.sort_values(["family_members"], ascending=[False])

# Print the top few rows
print(homelessness_fam.head())
```

```
      Unnamed: 0          region       state  individuals \
32            32    Mid-Atlantic     New York    39827.0
4              4           Pacific    California  109008.0
21            21      New England  Massachusetts   6811.0
9              9  South Atlantic        Florida  21443.0
43            43  West South Central        Texas  19199.0

family_members  state_pop
32      52070.0  19530351
4       20964.0  39461588
21     13257.0   6882635
9       9587.0  21244317
43      6111.0  28628666
```

```
In [ ]: # Sort homelessness by region, then descending family members
homelessness_reg_fam = homelessness.sort_values(["region", "family_members"], ascending=[True, False])

# Print the top few rows
print(homelessness_reg_fam.head())
```

```
      Unnamed: 0          region       state  individuals  family_members \
13            13  East North Central  Illinois    6752.0      3891.0
35            35  East North Central      Ohio    6929.0      3320.0
22            22  East North Central  Michigan    5209.0      3142.0
49            49  East North Central  Wisconsin   2740.0      2167.0
14            14  East North Central  Indiana    3776.0      1482.0

state_pop
13  12723071
35  11676341
22  9984072
49  5807406
14  6695497
```

## Subsetting columns

```
In [ ]: # Select the individuals column
individuals = homelessness["individuals"]

# Print the head of the result
print(individuals.head())
```

```
0      2570.0
1      1434.0
2      7259.0
3      2280.0
4     109008.0
Name: individuals, dtype: float64
```

```
In [ ]: # Select the state and family_members columns  
state_fam = homelessness[["state", "family_members"]]  
  
# Print the head of the result  
print(state_fam.head())
```

```
      state  family_members  
0    Alabama        864.0  
1     Alaska        582.0  
2   Arizona       2606.0  
3  Arkansas        432.0  
4 California      20964.0
```

```
In [ ]: # Select only the individuals and state columns, in that order  
ind_state = homelessness[["individuals", "state"]]  
  
# Print the head of the result  
print(ind_state.head())
```

```
  individuals      state  
0      2570.0    Alabama  
1      1434.0    Alaska  
2      7259.0   Arizona  
3      2280.0  Arkansas  
4    109008.0  California
```

## Subsetting rows

```
In [ ]: # Filter for rows where individuals is greater than 10000  
ind_gt_10k = homelessness[homelessness["individuals"] > 10000]  
  
# See the result  
print(ind_gt_10k)
```

```
      Unnamed: 0      region      state  individuals  family_members  \  
4            4        Pacific  California      109008.0      20964.0  
9            9  South Atlantic    Florida      21443.0      9587.0  
32           32    Mid-Atlantic   New York      39827.0      52070.0  
37           37        Pacific    Oregon      11139.0      3337.0  
43           43  West South Central    Texas      19199.0      6111.0  
47           47        Pacific  Washington      16424.0      5880.0
```

```
      state_pop  
4      39461588  
9      21244317  
32     19530351  
37     4181886  
43     28628666  
47     7523869
```

```
In [ ]: # Filter for rows where region is Mountain  
mountain_reg = homelessness[homelessness["region"] == "Mountain"]  
  
# See the result  
print(mountain_reg)
```

	Unnamed: 0	region	state	individuals	family_members	state_pop
2	2	Mountain	Arizona	7259.0	2606.0	7158024
5	5	Mountain	Colorado	7607.0	3250.0	5691287
12	12	Mountain	Idaho	1297.0	715.0	1750536
26	26	Mountain	Montana	983.0	422.0	1060665
28	28	Mountain	Nevada	7058.0	486.0	3027341
31	31	Mountain	New Mexico	1949.0	602.0	2092741
44	44	Mountain	Utah	1904.0	972.0	3153550
50	50	Mountain	Wyoming	434.0	205.0	577601

```
In [ ]: # Filter for rows where family_members is less than 1000  
# and region is Pacific  
fam_lt_1k_pac = homelessness[(homelessness["family_members"] < 1000) & (homelessness["region"]=="Pacific")]  
  
# See the result  
print(fam_lt_1k_pac)
```

	Unnamed: 0	region	state	individuals	family_members	state_pop
1	1	Pacific	Alaska	1434.0	582.0	735139

## Subsetting rows by categorical variables

```
In [ ]: # Subset for rows in South Atlantic or Mid-Atlantic regions  
south_mid_atlantic = homelessness[homelessness["region"].isin(["South Atlantic", "Mid-Atlantic"])]  
  
# See the result  
print(south_mid_atlantic)
```

	region	state	individuals	\
7	South Atlantic	Delaware	708.0	
8	South Atlantic	District of Columbia	3770.0	
9	South Atlantic	Florida	21443.0	
10	South Atlantic	Georgia	6943.0	
20	South Atlantic	Maryland	4914.0	
30	Mid-Atlantic	New Jersey	6048.0	
32	Mid-Atlantic	New York	39827.0	
33	South Atlantic	North Carolina	6451.0	
38	Mid-Atlantic	Pennsylvania	8163.0	
40	South Atlantic	South Carolina	3082.0	
46	South Atlantic	Virginia	3928.0	
48	South Atlantic	West Virginia	1021.0	
	family_members	state_pop		
7	374.0	965479		
8	3134.0	701547		
9	9587.0	21244317		
10	2556.0	10511131		
20	2230.0	6035802		
30	3350.0	8886025		
32	52070.0	19530351		
33	2817.0	10381615		
38	5349.0	12800922		
40	851.0	5084156		
46	2047.0	8501286		
48	222.0	1804291		

```
In [ ]: # The Mojave Desert states
canu = ["California", "Arizona", "Nevada", "Utah"]

# Filter for rows in the Mojave Desert states
mojave_homelessness = homelessness[homelessness["state"].isin(canu)]

# See the result
print(mojave_homelessness.head())
```

	region	state	individuals	family_members	state_pop
2	Mountain	Arizona	7259.0	2606.0	7158024
4	Pacific	California	109008.0	20964.0	39461588
28	Mountain	Nevada	7058.0	486.0	3027341
44	Mountain	Utah	1904.0	972.0	3153550

## Adding new columns

```
In [ ]: # Add total col as sum of individuals and family_members
homelessness["total"] = homelessness["individuals"] + homelessness["family_members"]
# Add p_individuals col as proportion of individuals
homelessness["p_individuals"] = homelessness["individuals"] / homelessness["total"]
```

```
# See the result
print(homelessness.head())

      Unnamed: 0          region     state  individuals  family_members \
0            0   East South Central    Alabama      2570.0        864.0
1            1             Pacific    Alaska      1434.0        582.0
2            2           Mountain   Arizona      7259.0       2606.0
3            3   West South Central  Arkansas      2280.0        432.0
4            4             Pacific  California  109008.0      20964.0

  state_pop  total  p_individuals
0  4887681  3434.0      0.748398
1  735139   2016.0      0.711310
2  7158024  9865.0      0.735834
3  3009733  2712.0      0.840708
4  39461588 129972.0      0.838704
```

## Combo-attack!

```
In [ ]: # Create indiv_per_10k col as homeless individuals per 10k state pop
homelessness["indiv_per_10k"] = 10000 * homelessness["individuals"] / homelessness["state_pop"]

# Subset rows for indiv_per_10k greater than 20
high_homelessness = homelessness[homelessness["indiv_per_10k"] > 20]

# Sort high_homelessness by descending indiv_per_10k
high_homelessness_srt = high_homelessness.sort_values("indiv_per_10k", ascending=False)

# From high_homelessness_srt, select the state and indiv_per_10k cols
result = high_homelessness_srt[["state", "indiv_per_10k"]]

# See the result
print(result)
```

	state	indiv_per_10k
8	District of Columbia	53.738381
11	Hawaii	29.079406
4	California	27.623825
37	Oregon	26.636307
28	Nevada	23.314189
47	Washington	21.829195
32	New York	20.392363

## otros codigos

```
In [ ]: #example using group by
homelessness.groupby("family_members")["individuals"].mean()
print(homelessness.head())
```

```

      Unnamed: 0          region      state  individuals  family_members \
0            0   East South Central    Alabama      2570.0        864.0
1            1             Pacific     Alaska      1434.0        582.0
2            2           Mountain   Arizona      7259.0       2606.0
3            3  West South Central  Arkansas      2280.0        432.0
4            4             Pacific  California     109008.0      20964.0

  state_pop    total  p_individuals  indiv_per_10k
0  4887681  3434.0      0.748398      5.258117
1  735139   2016.0      0.711310     19.506515
2  7158024  9865.0      0.735834     10.141067
3  3009733  2712.0      0.840708      7.575423
4  39461588 129972.0      0.838704     27.623825

```

## Ch2 Aggregating DataFrames

### Aggregating Data

In this chapter, you'll calculate summary statistics on DataFrame columns, and master grouped summary statistics and pivot tables.

```
In [ ]: # Import pandas using the alias pd
import pandas as pd

sales = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
```

### Mean and median

```
In [ ]: # Print the head of the sales DataFrame
print(sales.head())

# Print the info about the sales DataFrame
print(sales.info())

# Print the mean of weekly_sales
print(sales["weekly_sales"].mean())

# Print the median of weekly_sales
print(sales["weekly_sales"].median())
```

```
    Unnamed: 0  store type  department      date weekly_sales  is_holiday \
0          0     1    A           1  2010-02-05     24924.50    False
1          1     1    A           1  2010-03-05     21827.90    False
2          2     1    A           1  2010-04-02     57258.43    False
3          3     1    A           1  2010-05-07     17413.94    False
4          4     1    A           1  2010-06-04     17558.09    False

   temperature_c  fuel_price_usd_per_l  unemployment
0      5.727778            0.679451       8.106
1      8.055556            0.693452       8.106
2     16.816667            0.718284       7.808
3     22.527778            0.748928       7.808
4     27.050000            0.714586       7.808
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10774 entries, 0 to 10773
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Unnamed: 0        10774 non-null   int64  
 1   store             10774 non-null   int64  
 2   type              10774 non-null   object  
 3   department         10774 non-null   int64  
 4   date              10774 non-null   object  
 5   weekly_sales      10774 non-null   float64
 6   is_holiday         10774 non-null   bool   
 7   temperature_c     10774 non-null   float64
 8   fuel_price_usd_per_l  10774 non-null   float64
 9   unemployment       10774 non-null   float64
dtypes: bool(1), float64(4), int64(3), object(2)
memory usage: 768.2+ KB
None
23843.95014850566
12049.064999999999
```

Summarizing dates

```
In [ ]: # Print the maximum of the date column
print(sales["date"].max())

# Print the minimum of the date column
print(sales["date"].min())
```

```
2012-10-26
2010-02-05
```

Efficient summaries

```
In [ ]: # A custom IQR function
def iqr(column):
```

```
    return column.quantile(0.75) - column.quantile(0.25)

# Print IQR of the temperature_c column
print(sales["temperature_c"].agg(iqr))
```

16.58333333333336

Cumulative statistics

In [ ]:

```
import pandas as pd

# Sample data
data = {
    'store': [5, 1, 4, 9, 8, 7, 10, 3, 1, 6, 11, 2],
    'type': ['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A'],
    'department': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    'date': ['2010-07-02', '2010-02-05', '2010-06-04', '2010-11-05', '2010-10-01', '2010-09-03', '2010-12-03', '2010-05-07', '2010-12-27', '2010-08-28', '2010-11-24'],
    'weekly_sales': [16333.14, 24924.5, 17558.09, 34238.88, 20094.19, 16241.78, 22517.56, 17413.94, 21827.9, 17508.41, 15984.24, 15984.24],
    'is_holiday': [False, False, False, False, False, False, False, False, False, False, False],
    'temperature_c': [27.172, 5.728, 27.05, 14.856, 22.161, 27.339, 9.594, 22.528, 8.056, 30.644, 9.039, 16.817],
    'fuel_price_usd_per_l': [0.705, 0.679, 0.715, 0.71, 0.688, 0.681, 0.715, 0.749, 0.693, 0.694, 0.786, 0.718],
    'unemployment': [7.787, 8.106, 7.808, 7.838, 7.838, 7.787, 7.838, 7.808, 8.106, 7.787, 7.742, 7.808]
}

# Create a Pandas DataFrame
sales_1_1 = pd.DataFrame(data)

# Display the DataFrame as a table
sales_1_1
```

Out[ ]:

	store	type	department	date	weekly_sales	is_holiday	temperature_c	fuel_price_usd_per_l	unemployment
0	5	A	1	2010-07-02	16333.14	False	27.172	0.705	7.787
1	1	A	1	2010-02-05	24924.50	False	5.728	0.679	8.106
2	4	A	1	2010-06-04	17558.09	False	27.050	0.715	7.808
3	9	A	1	2010-11-05	34238.88	False	14.856	0.710	7.838
4	8	A	1	2010-10-01	20094.19	False	22.161	0.688	7.838
5	7	A	1	2010-09-03	16241.78	False	27.339	0.681	7.787
6	10	A	1	2010-12-03	22517.56	False	9.594	0.715	7.838
7	3	A	1	2010-05-07	17413.94	False	22.528	0.749	7.808
8	1	A	1	2010-03-05	21827.90	False	8.056	0.693	8.106
9	6	A	1	2010-08-06	17508.41	False	30.644	0.694	7.787
10	11	A	1	2011-01-07	15984.24	False	9.039	0.786	7.742
11	2	A	1	2010-04-02	57258.43	False	16.817	0.718	7.808

In [ ]:

```
# Sort sales_1_1 by date
sales_1_1 = sales_1_1.sort_values(by='date')

# Get the cumulative sum of weekly_sales, add as cum_weekly_sales col
sales_1_1['cum_weekly_sales'] = sales_1_1['weekly_sales'].cumsum()

# Get the cumulative max of weekly_sales, add as cum_max_sales col
sales_1_1['cum_max_sales'] = sales_1_1['weekly_sales'].cummax()

# See the columns you calculated
print(sales_1_1[['date', "weekly_sales", "cum_weekly_sales", "cum_max_sales"]])
```

	date	weekly_sales	cum_weekly_sales	cum_max_sales
1	2010-02-05	24924.50	24924.50	24924.50
8	2010-03-05	21827.90	46752.40	24924.50
11	2010-04-02	57258.43	104010.83	57258.43
7	2010-05-07	17413.94	121424.77	57258.43
2	2010-06-04	17558.09	138982.86	57258.43
0	2010-07-02	16333.14	155316.00	57258.43
9	2010-08-06	17508.41	172824.41	57258.43
5	2010-09-03	16241.78	189066.19	57258.43
4	2010-10-01	20094.19	209160.38	57258.43
3	2010-11-05	34238.88	243399.26	57258.43
6	2010-12-03	22517.56	265916.82	57258.43
10	2011-01-07	15984.24	281901.06	57258.43

## Dropping duplicates

In [ ]:

```
# Drop duplicate store/type combinations
store_types = sales.drop_duplicates(subset=["store", "type"])
print(store_types.head())

# Drop duplicate store/department combinations
store_depts = sales.drop_duplicates(subset=["store", "department"])
print(store_depts.head())

# Subset the rows where is_holiday is True and drop duplicate dates
holiday_dates = sales[sales["is_holiday"]].drop_duplicates("date")

# Print date col of holiday_dates
print(holiday_dates.head())
```

	Unnamed: 0	store	type	department	date	weekly_sales	\
0	0	1	A	1	2010-02-05	24924.50	
901	901	2	A	1	2010-02-05	35034.06	
1798	1798	4	A	1	2010-02-05	38724.42	
2699	2699	6	A	1	2010-02-05	25619.00	
3593	3593	10	B	1	2010-02-05	40212.84	

	is_holiday	temperature_c	fuel_price_usd_per_l	unemployment
0	False	5.727778	0.679451	8.106
901	False	4.550000	0.679451	8.324
1798	False	6.533333	0.686319	8.623
2699	False	4.683333	0.679451	7.259
3593	False	12.411111	0.782478	9.765

	Unnamed: 0	store	type	department	date	weekly_sales	is_holiday	\
0	0	1	A	1	2010-02-05	24924.50	False	
12	12	1	A	2	2010-02-05	50605.27	False	
24	24	1	A	3	2010-02-05	13740.12	False	
36	36	1	A	4	2010-02-05	39954.04	False	
48	48	1	A	5	2010-02-05	32229.38	False	

	temperature_c	fuel_price_usd_per_l	unemployment
0	5.727778	0.679451	8.106
12	5.727778	0.679451	8.106
24	5.727778	0.679451	8.106
36	5.727778	0.679451	8.106
48	5.727778	0.679451	8.106

	Unnamed: 0	store	type	department	date	weekly_sales	\
498	498	1	A	45	2010-09-10	11.47	
691	691	1	A	77	2011-11-25	1431.00	
2315	2315	4	A	47	2010-02-12	498.00	
6735	6735	19	A	39	2012-09-07	13.41	
6810	6810	19	A	47	2010-12-31	-449.00	

	is_holiday	temperature_c	fuel_price_usd_per_l	unemployment
498	True	25.938889	0.677602	7.787
691	True	15.633333	0.854861	7.866
2315	True	-1.755556	0.679715	8.623
6735	True	22.333333	1.076766	8.193
6810	True	-1.861111	0.881278	8.067

Counting categorical variables

```
In [ ]: # Count the number of stores of each type
store_counts = store_types["type"].value_counts()
print(store_counts)

# Get the proportion of stores of each type
store_props = store_types["type"].value_counts(normalize=True)
print(store_props)
```

```
# Count the number of each department number and sort
dept_counts_sorted = store_depts["department"].value_counts(sort="department", ascending=False)
print(dept_counts_sorted)

# Get the proportion of departments of each number and sort
dept_props_sorted = store_depts["department"].value_counts(sort="department", normalize=True)
print(dept_props_sorted)

type
A    11
B     1
Name: count, dtype: int64
type
A    0.916667
B    0.083333
Name: proportion, dtype: float64
department
1    12
55   12
72   12
71   12
67   12
...
37   10
48    8
50    6
39    4
43    2
Name: count, Length: 80, dtype: int64
department
1    0.012917
55   0.012917
72   0.012917
71   0.012917
67   0.012917
...
37   0.010764
48   0.008611
50   0.006459
39   0.004306
43   0.002153
Name: proportion, Length: 80, dtype: float64
```

What percent of sales occurred at each store type?

```
In [ ]: # Calc total weekly sales
sales_all = sales["weekly_sales"].sum()

# Subset for type A stores, calc total weekly sales
```

```
sales_A = sales[sales["type"] == "A"]["weekly_sales"].sum()

# Subset for type B stores, calc total weekly sales
sales_B = sales[sales["type"] == "B"]["weekly_sales"].sum()

# Subset for type C stores, calc total weekly sales
sales_C = sales[sales["type"] == "C"]["weekly_sales"].sum()

# Get proportion for each type
sales_propn_by_type = [sales_A, sales_B, sales_C] / sales_all
print(sales_propn_by_type)
```

```
[0.9097747 0.0902253 0.]
```

Calculations with .groupby()

```
In [ ]: # Group by type; calc total weekly sales
sales_by_type = sales.groupby("type")["weekly_sales"].sum()

# Get proportion for each type
sales_propn_by_type = sales_by_type / sum(sales_by_type)
print(sales_propn_by_type)

# Group by type and is_holiday; calc total weekly sales
sales_by_type_is_holiday = sales.groupby(["type", "is_holiday"])["weekly_sales"].sum()
print(sales_by_type_is_holiday)
```

```
type
A    0.909775
B    0.090225
Name: weekly_sales, dtype: float64
type  is_holiday
A    False        2.336927e+08
      True         2.360181e+04
B    False        2.317678e+07
      True         1.621410e+03
Name: weekly_sales, dtype: float64
```

Multiple grouped summaries

```
In [ ]: # Import numpy with the alias np
import numpy as np

# For each store type, aggregate weekly_sales: get min, max, mean, and median
sales_stats = sales.groupby("type")["weekly_sales"].agg([min, max, np.mean, np.median])

# Print sales_stats
print(sales_stats)
```

```
# For each store type, aggregate unemployment and fuel_price_usd_per_l: get min, max, mean, and median  
unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
```

```
# Print unemp_fuel_stats
```

```
print(unemp_fuel_stats)
```

	min	max	mean	median	
type					
A	-1098.0	293966.05	23674.667242	11943.92	
B	-798.0	232558.51	25696.678370	13336.08	
	unemployment				fuel_price_usd_per_l \
	min	max	mean	median	min
type					max
A	3.879	8.992	7.972611	8.067	0.664129 1.107410
B	7.170	9.765	9.279323	9.199	0.760023 1.107674
	mean	median			
type					
A	0.744619	0.735455			
B	0.805858	0.803348			

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:5: FutureWarning: The provided callable <built-in function min> is currently using SeriesGroupBy.min. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "min" instead.
    sales_stats = sales.groupby("type")["weekly_sales"].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:5: FutureWarning: The provided callable <built-in function max> is currently using SeriesGroupBy.max. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "max" instead.
    sales_stats = sales.groupby("type")["weekly_sales"].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:5: FutureWarning: The provided callable <function mean at 0x000001EC3C73B100> is currently using SeriesGroupBy.mean. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "mean" instead.
    sales_stats = sales.groupby("type")["weekly_sales"].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:5: FutureWarning: The provided callable <function median at 0x0000001EC3C862340> is currently using SeriesGroupBy.median. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "median" instead.
    sales_stats = sales.groupby("type")["weekly_sales"].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:11: FutureWarning: The provided callable <built-in function min> is currently using SeriesGroupBy.min. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "min" instead.
    unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:11: FutureWarning: The provided callable <built-in function max> is currently using SeriesGroupBy.max. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "max" instead.
    unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:11: FutureWarning: The provided callable <function mean at 0x000001EC3C73B100> is currently using SeriesGroupBy.mean. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "mean" instead.
    unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:11: FutureWarning: The provided callable <function median at 0x0000001EC3C862340> is currently using SeriesGroupBy.median. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "median" instead.
    unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\1976951964.py:11: FutureWarning: The provided callable <built-in function min> is currently using SeriesGroupBy.min. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "min" instead.
    unemp_fuel_stats = sales.groupby("type")[["unemployment", "fuel_price_usd_per_l"]].agg([min, max, np.mean, np.median])
```

Pivoting on one variable

```
In [ ]: # Pivot for mean weekly_sales for each store type
mean_sales_by_type = sales.pivot_table(values="weekly_sales", index="type")

# Print mean_sales_by_type
print(mean_sales_by_type)
```

	weekly_sales
type	
A	23674.667242
B	25696.678370

Fill in missing values and sum values with pivot tables

```
In [ ]: # Print mean weekly_sales by department and type; fill missing values with 0
import numpy as np
print(sales.pivot_table(values="weekly_sales", index="department", columns="type", aggfunc=np.mean, fill_value=0))

type              A            B
department
1                30961.725379  44050.626667
2                67600.158788  112958.526667
3                17160.002955  30580.655000
4                44285.399091  51219.654167
5                34821.011364  63236.875000
...
95               123933.787121  77082.102500
96               21367.042857   9528.538333
97               28471.266970  5828.873333
98               12875.423182  217.428333
99               379.123659    0.000000

[80 rows x 2 columns]
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_33652\321967142.py:3: FutureWarning: The provided callable <function mean at 0x000001
EC3C73B100> is currently using DataFrameGroupBy.mean. In a future version of pandas, the provided callable will be used directly.
To keep current behavior pass the string "mean" instead.
```

```
print(sales.pivot_table(values="weekly_sales", index="department", columns="type", aggfunc=np.mean, fill_value=0))
```

## Chapter 3 - Slicing and Indexing DataFrames

Indexes are supercharged row and column names. Learn how they can be combined with slicing for powerful DataFrame subsetting.

```
In [ ]: # Import pandas using the alias pd
import pandas as pd

temperatures = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python
```

Setting and removing indexes

```
In [ ]: # Look at temperatures
print(temperatures.head())

# Index temperatures by city
temperatures_ind = temperatures.set_index("city")

# Look at temperatures_ind
print(temperatures_ind.head())
```

```
# Reset the index, keeping its contents
print(temperatures_ind.reset_index())
```

```
# Reset the index, dropping its contents
print(temperatures_ind.reset_index(drop=True))
```

```
      Unnamed: 0        date       city      country  avg_temp_c
0            0  2000-01-01  Abidjan  Côte D'Ivoire      27.293
1            1  2000-02-01  Abidjan  Côte D'Ivoire      27.685
2            2  2000-03-01  Abidjan  Côte D'Ivoire      29.061
3            3  2000-04-01  Abidjan  Côte D'Ivoire      28.162
4            4  2000-05-01  Abidjan  Côte D'Ivoire      27.547
      Unnamed: 0        date      country  avg_temp_c
city
Abidjan            0  2000-01-01  Côte D'Ivoire      27.293
Abidjan            1  2000-02-01  Côte D'Ivoire      27.685
Abidjan            2  2000-03-01  Côte D'Ivoire      29.061
Abidjan            3  2000-04-01  Côte D'Ivoire      28.162
Abidjan            4  2000-05-01  Côte D'Ivoire      27.547
      city  Unnamed: 0        date      country  avg_temp_c
0    Abidjan            0  2000-01-01  Côte D'Ivoire      27.293
1    Abidjan            1  2000-02-01  Côte D'Ivoire      27.685
2    Abidjan            2  2000-03-01  Côte D'Ivoire      29.061
3    Abidjan            3  2000-04-01  Côte D'Ivoire      28.162
4    Abidjan            4  2000-05-01  Côte D'Ivoire      27.547
...
...
...
16495    Xian          16495  2013-05-01      China      18.979
16496    Xian          16496  2013-06-01      China      23.522
16497    Xian          16497  2013-07-01      China      25.251
16498    Xian          16498  2013-08-01      China      24.528
16499    Xian          16499  2013-09-01      China        NaN
```

[16500 rows x 5 columns]

```
      Unnamed: 0        date      country  avg_temp_c
0            0  2000-01-01  Côte D'Ivoire      27.293
1            1  2000-02-01  Côte D'Ivoire      27.685
2            2  2000-03-01  Côte D'Ivoire      29.061
3            3  2000-04-01  Côte D'Ivoire      28.162
4            4  2000-05-01  Côte D'Ivoire      27.547
...
...
...
16495          16495  2013-05-01      China      18.979
16496          16496  2013-06-01      China      23.522
16497          16497  2013-07-01      China      25.251
16498          16498  2013-08-01      China      24.528
16499          16499  2013-09-01      China        NaN
```

[16500 rows x 4 columns]

## Subsetting with .loc[]

In [ ]:

```
# Make a list of cities to subset on
cities = ["Moscow", "Saint Petersburg"]

# Subset temperatures using square brackets
print(temperatures[temperatures["city"].isin(cities)])

# Subset temperatures_ind using .loc[]
print(temperatures_ind.loc[cities])
```

```
      Unnamed: 0        date          city country  avg_temp_c
10725    10725  2000-01-01      Moscow   Russia     -7.313
10726    10726  2000-02-01      Moscow   Russia     -3.551
10727    10727  2000-03-01      Moscow   Russia     -1.661
10728    10728  2000-04-01      Moscow   Russia     10.096
10729    10729  2000-05-01      Moscow   Russia     10.357
...
...
...
13360    13360  2013-05-01  Saint Petersburg  Russia     12.355
13361    13361  2013-06-01  Saint Petersburg  Russia     17.185
13362    13362  2013-07-01  Saint Petersburg  Russia     17.234
13363    13363  2013-08-01  Saint Petersburg  Russia     17.153
13364    13364  2013-09-01  Saint Petersburg  Russia       NaN
```

[330 rows x 5 columns]

```
      Unnamed: 0        date country  avg_temp_c
city
Moscow           10725  2000-01-01  Russia     -7.313
Moscow           10726  2000-02-01  Russia     -3.551
Moscow           10727  2000-03-01  Russia     -1.661
Moscow           10728  2000-04-01  Russia     10.096
Moscow           10729  2000-05-01  Russia     10.357
...
...
...
Saint Petersburg 13360  2013-05-01  Russia     12.355
Saint Petersburg 13361  2013-06-01  Russia     17.185
Saint Petersburg 13362  2013-07-01  Russia     17.234
Saint Petersburg 13363  2013-08-01  Russia     17.153
Saint Petersburg 13364  2013-09-01  Russia       NaN
```

[330 rows x 4 columns]

Setting multi-level indexes

In [ ]:

```
# Index temperatures by country & city
temperatures_ind = temperatures.set_index(["country", "city"])

# List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore
rows_to_keep = [("Brazil", "Rio De Janeiro"), ("Pakistan", "Lahore")]
```

```
# Subset for rows to keep
print(temperatures_ind.loc[rows_to_keep])
```

		Unnamed: 0	date	avg_temp_c
country	city			
Brazil	Rio De Janeiro	12540	2000-01-01	25.974
	Rio De Janeiro	12541	2000-02-01	26.699
	Rio De Janeiro	12542	2000-03-01	26.270
	Rio De Janeiro	12543	2000-04-01	25.750
	Rio De Janeiro	12544	2000-05-01	24.356
...	...	...	...	...
Pakistan	Lahore	8575	2013-05-01	33.457
	Lahore	8576	2013-06-01	34.456
	Lahore	8577	2013-07-01	33.279
	Lahore	8578	2013-08-01	31.511
	Lahore	8579	2013-09-01	NaN

[330 rows x 3 columns]

Sorting by index values

```
In [ ]: # Sort temperatures_ind by index values
print(temperatures_ind.sort_index())

# Sort temperatures_ind by index values at the city Level
print(temperatures_ind.sort_index(level=["city", "country"]))

# Sort temperatures_ind by country then descending city
print(temperatures_ind.sort_index(level=["country", "city"], ascending=[True, False]))
```

		Unnamed: 0	date	avg_temp_c
country	city			
Afghanistan	Kabul	7260	2000-01-01	3.326
	Kabul	7261	2000-02-01	3.454
	Kabul	7262	2000-03-01	9.612
	Kabul	7263	2000-04-01	17.925
	Kabul	7264	2000-05-01	24.658
...		...	...	...
Zimbabwe	Harare	5605	2013-05-01	18.298
	Harare	5606	2013-06-01	17.020
	Harare	5607	2013-07-01	16.299
	Harare	5608	2013-08-01	19.232
	Harare	5609	2013-09-01	NaN

[16500 rows x 3 columns]

		Unnamed: 0	date	avg_temp_c
country	city			
Côte D'Ivoire	Abidjan	0	2000-01-01	27.293
	Abidjan	1	2000-02-01	27.685
	Abidjan	2	2000-03-01	29.061
	Abidjan	3	2000-04-01	28.162
	Abidjan	4	2000-05-01	27.547
...		...	...	...
China	Xian	16495	2013-05-01	18.979
	Xian	16496	2013-06-01	23.522
	Xian	16497	2013-07-01	25.251
	Xian	16498	2013-08-01	24.528
	Xian	16499	2013-09-01	NaN

[16500 rows x 3 columns]

		Unnamed: 0	date	avg_temp_c
country	city			
Afghanistan	Kabul	7260	2000-01-01	3.326
	Kabul	7261	2000-02-01	3.454
	Kabul	7262	2000-03-01	9.612
	Kabul	7263	2000-04-01	17.925
	Kabul	7264	2000-05-01	24.658
...		...	...	...
Zimbabwe	Harare	5605	2013-05-01	18.298
	Harare	5606	2013-06-01	17.020
	Harare	5607	2013-07-01	16.299
	Harare	5608	2013-08-01	19.232
	Harare	5609	2013-09-01	NaN

[16500 rows x 3 columns]

Slicing index values

In [ ]:

```
# Sort the index of temperatures_ind
temperatures_srt = temperatures_ind.sort_index()

# Subset rows from Pakistan to Russia
print(temperatures_srt.loc["Pakistan":"Russia"])

# Try to subset rows from Lahore to Moscow
print(temperatures_srt.loc["Lahore":"Moscow"])

# Subset rows from Pakistan, Lahore to Russia, Moscow
print(temperatures_srt.loc[("Pakistan","Lahore"):(("Russia","Moscow"))])
```

		Unnamed: 0	date	avg_temp_c
country	city			
Pakistan	Faisalabad	4785	2000-01-01	12.792
	Faisalabad	4786	2000-02-01	14.339
	Faisalabad	4787	2000-03-01	20.309
	Faisalabad	4788	2000-04-01	29.072
	Faisalabad	4789	2000-05-01	34.845
...	...	...	...	...
Russia	Saint Petersburg	13360	2013-05-01	12.355
	Saint Petersburg	13361	2013-06-01	17.185
	Saint Petersburg	13362	2013-07-01	17.234
	Saint Petersburg	13363	2013-08-01	17.153
	Saint Petersburg	13364	2013-09-01	NaN

[1155 rows x 3 columns]

		Unnamed: 0	date	avg_temp_c
country	city			
Mexico	Mexico	10230	2000-01-01	12.694
	Mexico	10231	2000-02-01	14.677
	Mexico	10232	2000-03-01	17.376
	Mexico	10233	2000-04-01	18.294
	Mexico	10234	2000-05-01	18.562
...	...	...	...	...
Morocco	Casablanca	3130	2013-05-01	19.217
	Casablanca	3131	2013-06-01	23.649
	Casablanca	3132	2013-07-01	27.488
	Casablanca	3133	2013-08-01	27.952
	Casablanca	3134	2013-09-01	NaN

[330 rows x 3 columns]

		Unnamed: 0	date	avg_temp_c
country	city			
Pakistan	Lahore	8415	2000-01-01	12.792
	Lahore	8416	2000-02-01	14.339
	Lahore	8417	2000-03-01	20.309
	Lahore	8418	2000-04-01	29.072
	Lahore	8419	2000-05-01	34.845
...	...	...	...	...
Russia	Moscow	10885	2013-05-01	16.152
	Moscow	10886	2013-06-01	18.718
	Moscow	10887	2013-07-01	18.136
	Moscow	10888	2013-08-01	17.485
	Moscow	10889	2013-09-01	NaN

[660 rows x 3 columns]

Slicing in both directions

In [ ]:

```
# Subset rows from India, Hyderabad to Iraq, Baghdad
print(temperatures_srt.loc[("India","Hyderabad"):(("Iraq","Baghdad"))])

# Subset columns from date to avg_temp_c
print(temperatures_srt.loc[:, "date":"avg_temp_c"])

# Subset in both directions at once
print(temperatures_srt.loc[("India","Hyderabad"):(("Iraq","Baghdad")), "date":"avg_temp_c"])
```

		Unnamed: 0	date	avg_temp_c
country	city			
India	Hyderabad	5940	2000-01-01	23.779
	Hyderabad	5941	2000-02-01	25.826
	Hyderabad	5942	2000-03-01	28.821
	Hyderabad	5943	2000-04-01	32.698
	Hyderabad	5944	2000-05-01	32.438
...	...	...	...	...
Iraq	Baghdad	1150	2013-05-01	28.673
	Baghdad	1151	2013-06-01	33.803
	Baghdad	1152	2013-07-01	36.392
	Baghdad	1153	2013-08-01	35.463
	Baghdad	1154	2013-09-01	NaN

[2145 rows x 3 columns]

		date	avg_temp_c
country	city		
Afghanistan	Kabul	2000-01-01	3.326
	Kabul	2000-02-01	3.454
	Kabul	2000-03-01	9.612
	Kabul	2000-04-01	17.925
	Kabul	2000-05-01	24.658
...	...	...	...
Zimbabwe	Harare	2013-05-01	18.298
	Harare	2013-06-01	17.020
	Harare	2013-07-01	16.299
	Harare	2013-08-01	19.232
	Harare	2013-09-01	NaN

[16500 rows x 2 columns]

		date	avg_temp_c
country	city		
India	Hyderabad	2000-01-01	23.779
	Hyderabad	2000-02-01	25.826
	Hyderabad	2000-03-01	28.821
	Hyderabad	2000-04-01	32.698
	Hyderabad	2000-05-01	32.438
...	...	...	...
Iraq	Baghdad	2013-05-01	28.673
	Baghdad	2013-06-01	33.803
	Baghdad	2013-07-01	36.392
	Baghdad	2013-08-01	35.463
	Baghdad	2013-09-01	NaN

[2145 rows x 2 columns]

Slicing time series

```
In [ ]: # Use Boolean conditions to subset temperatures for rows in 2010 and 2011
temperatures_bool = temperatures[(temperatures["date"] >= "2010-01-01") & (temperatures["date"] <= "2011-12-31")]
print(temperatures_bool)

# Set date as an index and sort the index
temperatures_ind = temperatures.set_index("date").sort_index()

# Use .loc[] to subset temperatures_ind for rows in 2010 and 2011
print(temperatures_ind.loc["2010":"2011"])

# Use .loc[] to subset temperatures_ind for rows from Aug 2010 to Feb 2011
print(temperatures_ind.loc["2010-08":"2011-02"])
```

	Unnamed: 0	date	city	country	avg_temp_c
120	120	2010-01-01	Abidjan	Côte D'Ivoire	28.270
121	121	2010-02-01	Abidjan	Côte D'Ivoire	29.262
122	122	2010-03-01	Abidjan	Côte D'Ivoire	29.596
123	123	2010-04-01	Abidjan	Côte D'Ivoire	29.068
124	124	2010-05-01	Abidjan	Côte D'Ivoire	28.258
...	...	...	...	...	...
16474	16474	2011-08-01	Xian	China	23.069
16475	16475	2011-09-01	Xian	China	16.775
16476	16476	2011-10-01	Xian	China	12.587
16477	16477	2011-11-01	Xian	China	7.543
16478	16478	2011-12-01	Xian	China	-0.490

[2400 rows x 5 columns]

	Unnamed: 0	city	country	avg_temp_c
date				
2010-01-01	4905	Faisalabad	Pakistan	11.810
2010-01-01	10185	Melbourne	Australia	20.016
2010-01-01	3750	Chongqing	China	7.921
2010-01-01	13155	São Paulo	Brazil	23.738
2010-01-01	5400	Guangzhou	China	14.136
...	...	...	...	...
2010-12-01	6896	Jakarta	Indonesia	26.602
2010-12-01	5246	Gizeh	Egypt	16.530
2010-12-01	11186	Nagpur	India	19.120
2010-12-01	14981	Sydney	Australia	19.559
2010-12-01	13496	Salvador	Brazil	26.265

[1200 rows x 4 columns]

	Unnamed: 0	city	country	avg_temp_c
date				
2010-08-01	2602	Calcutta	India	30.226
2010-08-01	12337	Pune	India	24.941
2010-08-01	6562	Izmir	Turkey	28.352
2010-08-01	15637	Tianjin	China	25.543
2010-08-01	9862	Manila	Philippines	27.101
...	...	...	...	...
2011-01-01	4257	Dar Es Salaam	Tanzania	28.541
2011-01-01	11352	Nairobi	Kenya	17.768
2011-01-01	297	Addis Abeba	Ethiopia	17.708
2011-01-01	11517	Nanjing	China	0.144
2011-01-01	11847	New York	United States	-4.463

[600 rows x 4 columns]

Subsetting by row/column number

```
In [ ]: # Get 23rd row, 2nd column (index 22, 1)
print(temperatures.iloc[22,1])
```

```
# Use slicing to get the first 5 rows
print(temperatures.iloc[:5])

# Use slicing to get columns 3 to 4
print(temperatures.iloc[:,2:4])

# Use slicing in both directions at once
print(temperatures.iloc[5:2:4])
```

```
2001-11-01
      Unnamed: 0      date      city      country  avg_temp_c
0            0  2000-01-01  Abidjan  Côte D'Ivoire      27.293
1            1  2000-02-01  Abidjan  Côte D'Ivoire      27.685
2            2  2000-03-01  Abidjan  Côte D'Ivoire      29.061
3            3  2000-04-01  Abidjan  Côte D'Ivoire      28.162
4            4  2000-05-01  Abidjan  Côte D'Ivoire      27.547
      city      country
0  Abidjan  Côte D'Ivoire
1  Abidjan  Côte D'Ivoire
2  Abidjan  Côte D'Ivoire
3  Abidjan  Côte D'Ivoire
4  Abidjan  Côte D'Ivoire
...
16495    ...    ...
16495     Xian      China
16496     Xian      China
16497     Xian      China
16498     Xian      China
16499     Xian      China

[16500 rows x 2 columns]
      city      country
0  Abidjan  Côte D'Ivoire
1  Abidjan  Côte D'Ivoire
2  Abidjan  Côte D'Ivoire
3  Abidjan  Côte D'Ivoire
4  Abidjan  Côte D'Ivoire
```

Pivot temperature by city and year

```
In [ ]: import pandas as pd

# Assuming you have a DataFrame named temperatures

# Convert "date" column to datetime format
temperatures["date"] = pd.to_datetime(temperatures["date"])

# Add a year column to temperatures
temperatures["year"] = temperatures["date"].dt.year
```

```
# Pivot avg_temp_c by country and city vs year
temp_by_country_city_vs_year = temperatures.pivot_table(values="avg_temp_c", index=["country", "city"], columns="year")

# See the result
print(temp_by_country_city_vs_year)
```

year		2000	2001	2002	2003	\
country	city					
Afghanistan	Kabul	15.822667	15.847917	15.714583	15.132583	
Angola	Luanda	24.410333	24.427083	24.790917	24.867167	
Australia	Melbourne	14.320083	14.180000	14.075833	13.985583	
	Sydney	17.567417	17.854500	17.733833	17.592333	
Bangladesh	Dhaka	25.905250	25.931250	26.095000	25.927417	
...	...	...	...	...	...	
United States	Chicago	11.089667	11.703083	11.532083	10.481583	
	Los Angeles	16.643333	16.466250	16.430250	16.944667	
	New York	9.969083	10.931000	11.252167	9.836000	
Vietnam	Ho Chi Minh City	27.588917	27.831750	28.064750	27.827667	
Zimbabwe	Harare	20.283667	20.861000	21.079333	20.889167	

year		2004	2005	2006	2007	\
country	city					
Afghanistan	Kabul	16.128417	14.847500	15.798500	15.518000	
Angola	Luanda	24.216167	24.414583	24.138417	24.241583	
Australia	Melbourne	13.742083	14.378500	13.991083	14.991833	
	Sydney	17.869667	18.028083	17.749500	18.020833	
Bangladesh	Dhaka	26.136083	26.193333	26.440417	25.951333	
...	...	...	...	...	...	
United States	Chicago	10.943417	11.583833	11.870500	11.448333	
	Los Angeles	16.552833	16.431417	16.623083	16.699917	
	New York	10.389500	10.681417	11.519250	10.627333	
Vietnam	Ho Chi Minh City	27.686583	27.884000	28.044000	27.866667	
Zimbabwe	Harare	20.307667	21.487417	20.699750	20.746250	

year		2008	2009	2010	2011	\
country	city					
Afghanistan	Kabul	15.479250	15.093333	15.676000	15.812167	
Angola	Luanda	24.266333	24.325083	24.440250	24.150750	
Australia	Melbourne	14.110583	14.647417	14.231667	14.190917	
	Sydney	17.321083	18.175833	17.999000	17.713333	
Bangladesh	Dhaka	26.004500	26.535583	26.648167	25.803250	
...	...	...	...	...	...	
United States	Chicago	10.242417	10.298333	11.815917	11.214250	
	Los Angeles	17.014750	16.677000	15.887000	15.874833	
	New York	10.641667	10.141833	11.357583	11.272250	
Vietnam	Ho Chi Minh City	27.611417	27.853333	28.281750	27.675417	
Zimbabwe	Harare	20.680500	20.523833	21.165833	20.781750	

year		2012	2013
country	city		
Afghanistan	Kabul	14.510333	16.206125
Angola	Luanda	24.240083	24.553875
Australia	Melbourne	14.268667	14.741500
	Sydney	17.474333	18.089750
Bangladesh	Dhaka	26.283583	26.587000
...	...	...	

```

United States Chicago      12.821250 11.586889
                  Los Angeles    17.089583 18.120667
                  New York       11.971500 12.163889
Vietnam        Ho Chi Minh City 28.248750 28.455000
Zimbabwe       Harare        20.523333 19.756500

```

[100 rows x 14 columns]

Subsetting pivot tables

```

In [ ]: # Subset for Egypt to India
temp_by_country_city_vs_year.loc["Egypt":"India"]

# Subset for Egypt, Cairo to India, Delhi
temp_by_country_city_vs_year.loc[("Egypt","Cairo"):(("India","Delhi"))]

# Subset in both directions at once
temp_by_country_city_vs_year.loc[("Egypt","Cairo"):(("India","Delhi"), "2005":"2010")]

```

Out[ ]:

	<b>year</b>	<b>2005</b>	<b>2006</b>	<b>2007</b>	<b>2008</b>	<b>2009</b>	<b>2010</b>
<b>country</b>	<b>city</b>						
<b>Egypt</b>	<b>Cairo</b>	22.006500	22.050000	22.361000	22.644500	22.625000	23.718250
	<b>Gizeh</b>	22.006500	22.050000	22.361000	22.644500	22.625000	23.718250
<b>Ethiopia</b>	<b>Addis Abeba</b>	18.312833	18.427083	18.142583	18.165000	18.765333	18.298250
<b>France</b>	<b>Paris</b>	11.552917	11.788500	11.750833	11.278250	11.464083	10.409833
<b>Germany</b>	<b>Berlin</b>	9.919083	10.545333	10.883167	10.657750	10.062500	8.606833
<b>India</b>	<b>Ahmadabad</b>	26.828083	27.282833	27.511167	27.048500	28.095833	28.017833
	<b>Bangalore</b>	25.476500	25.418250	25.464333	25.352583	25.725750	25.705250
	<b>Bombay</b>	27.035750	27.381500	27.634667	27.177750	27.844500	27.765417
	<b>Calcutta</b>	26.729167	26.986250	26.584583	26.522333	27.153250	27.288833
	<b>Delhi</b>	25.716083	26.365917	26.145667	25.675000	26.554250	26.520250

Calculating on a pivot table

```

In [ ]: # Get the worldwide mean temp by year
mean_temp_by_year = temp_by_country_city_vs_year.mean(axis="index")

# Filter for the year that had the highest mean temp
print(mean_temp_by_year[mean_temp_by_year == max(mean_temp_by_year)])

```

```
# Get the mean temp by city
mean_temp_by_city = temp_by_country_city_vs_year.mean(axis="columns")

# Filter for the city that had the lowest mean temp
print(mean_temp_by_city[mean_temp_by_city == min(mean_temp_by_city)])
```

```
year
2013    20.312285
dtype: float64
country   city
China     Harbin    4.876551
dtype: float64
```

## Chapter 4 - Creating and Visualizing DataFrames

Learn to visualize the contents of your DataFrames, handle missing data values, and import data from and export data to CSV files.

```
In [ ]: # Import pandas using the alias pd
import pandas as pd

avocados = pd.read_pickle('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python C
print(avocados.head(3))
```

	date	type	year	avg_price	size	nb_sold
0	2015-12-27	conventional	2015	0.95	small	9626901.09
1	2015-12-20	conventional	2015	0.98	small	8710021.76
2	2015-12-13	conventional	2015	0.93	small	9855053.66

Which avocado size is most popular?

```
In [ ]: # Import matplotlib.pyplot with alias plt
import matplotlib.pyplot as plt

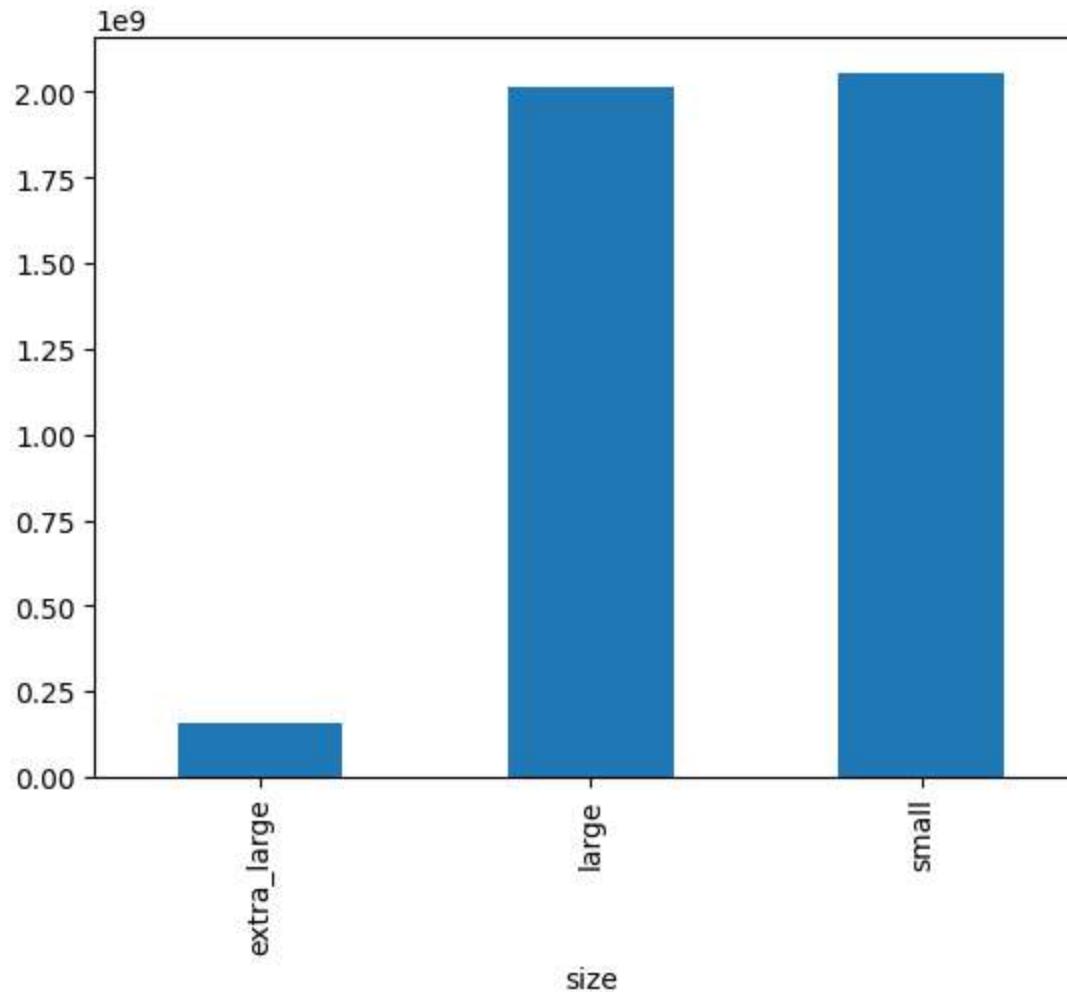
# Look at the first few rows of data
print(avocados.head())

# Get the total number of avocados sold of each size
nb_sold_by_size = avocados.groupby("size")["nb_sold"].sum()

# Create a bar plot of the number of avocados sold by size
nb_sold_by_size.plot(kind="bar")

# Show the plot
plt.show()
```

```
date      type  year  avg_price   size   nb_sold
0  2015-12-27 conventional 2015      0.95 small  9626901.09
1  2015-12-20 conventional 2015      0.98 small  8710021.76
2  2015-12-13 conventional 2015      0.93 small  9855053.66
3  2015-12-06 conventional 2015      0.89 small  9405464.36
4  2015-11-29 conventional 2015      0.99 small  8094803.56
```



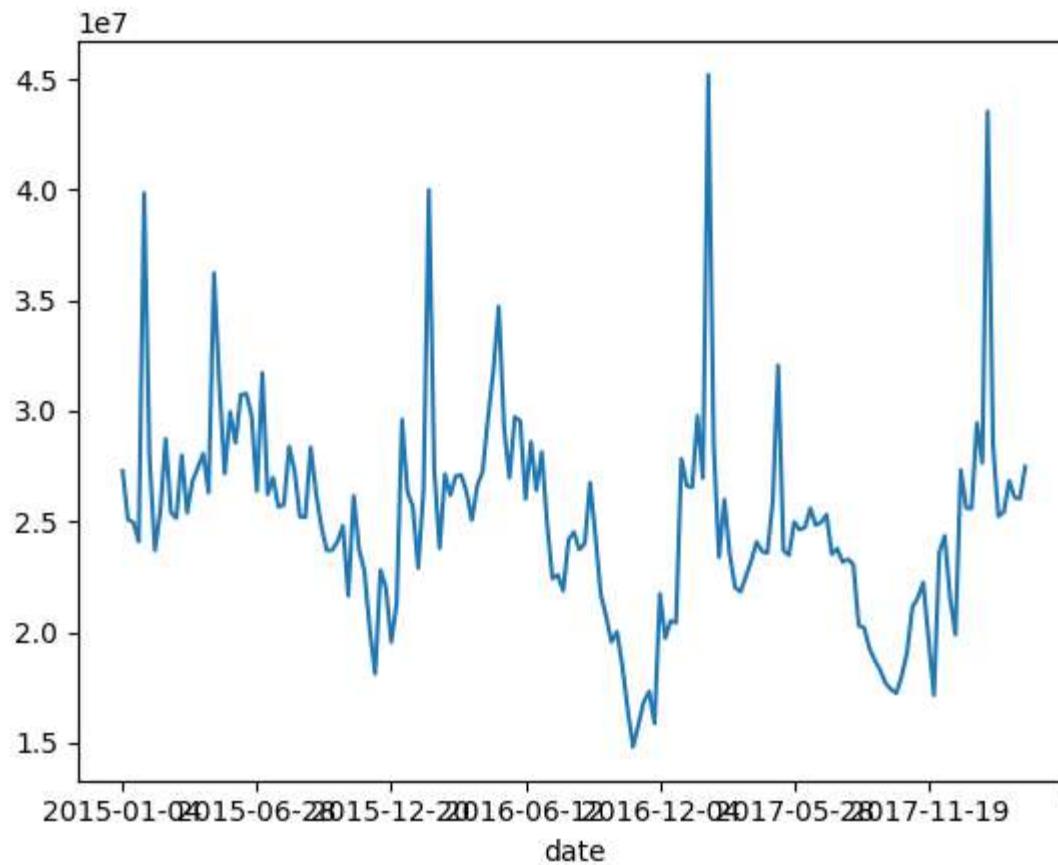
Changes in sales over time

```
In [ ]: # Import matplotlib.pyplot with alias plt
import matplotlib.pyplot as plt

# Get the total number of avocados sold on each date
nb_sold_by_date = avocados.groupby("date")["nb_sold"].sum()

# Create a line plot of the number of avocados sold by date
nb_sold_by_date.plot(kind="line")
```

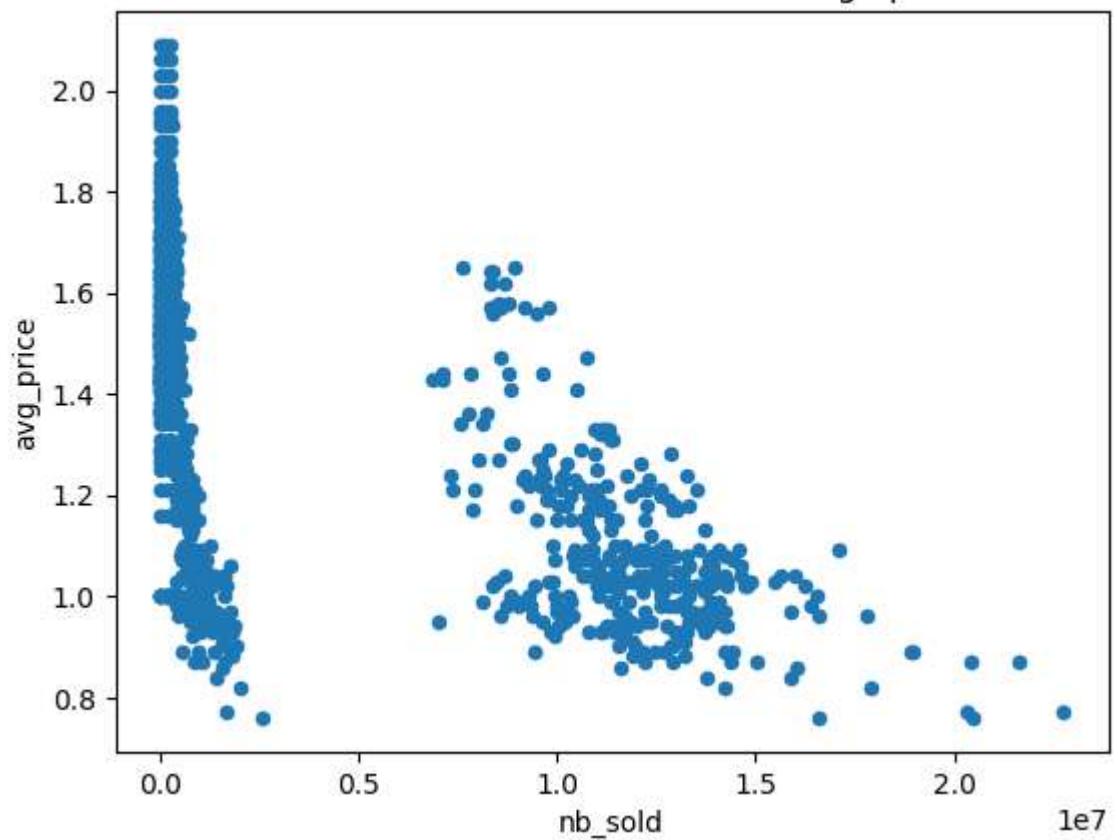
```
# Show the plot  
plt.show()
```



Avocado supply and demand

```
In [ ]: # Scatter plot of nb_sold vs avg_price with title  
avocados.plot(x="nb_sold", y="avg_price", kind="scatter", title="Number of avocados sold vs. average price")  
  
# Show the plot  
plt.show()
```

## Number of avocados sold vs. average price



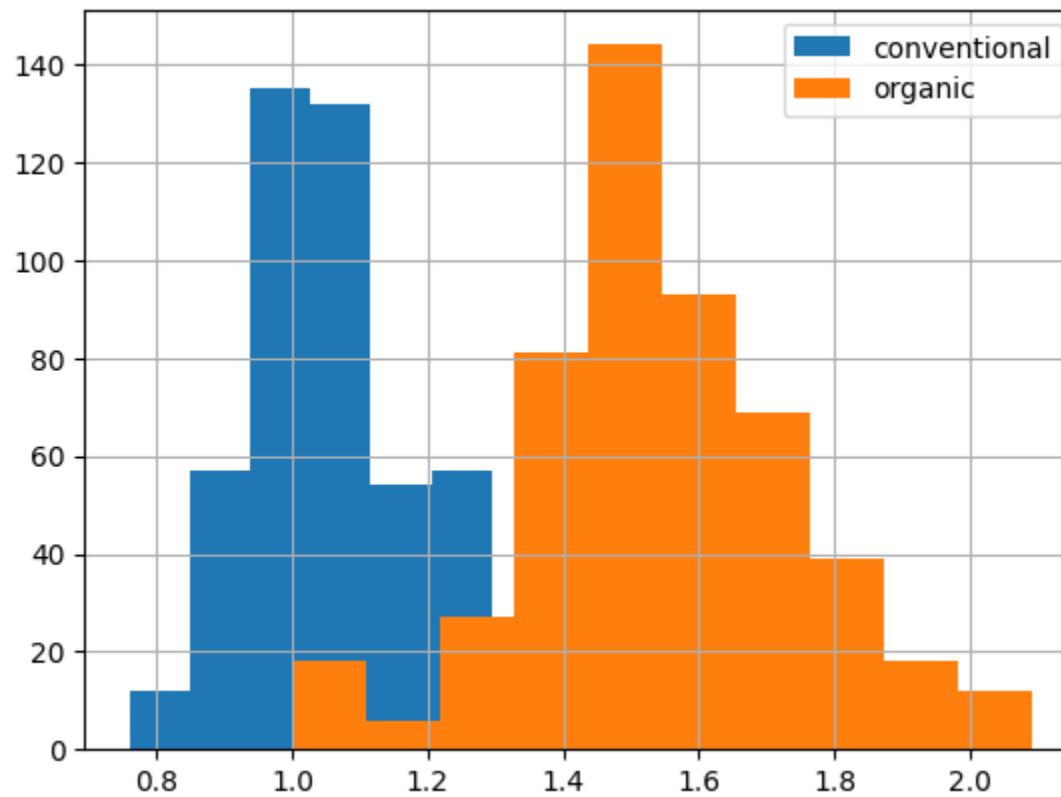
Price of conventional vs. organic avocados

```
In [ ]: # Histogram of conventional avg_price
avocados[avocados["type"] == "conventional"]["avg_price"].hist()

# Histogram of organic avg_price
avocados[avocados["type"] == "organic"]["avg_price"].hist()

# Add a legend
plt.legend(["conventional", "organic"])

# Show the plot
plt.show()
```



Finding missing values

```
In [ ]: # Import matplotlib.pyplot with alias plt
import matplotlib.pyplot as plt

# Check individual values for missing values
print(avocados_2016.isna())

# Check each column for missing values
print(avocados_2016.isna().any())

# Bar plot of missing values by variable
avocados_2016.isna().sum().plot(kind="bar")

# Show plot
plt.show()
```

Removing missing values

```
In [ ]: # Remove rows with missing values
avocados_complete = avocados_2016.dropna()
```

```
# Check if any columns contain missing values
print(avocados_complete.isna().any())
```

Replacing missing values

```
In [ ]: # List the columns with missing values
cols_with_missing = ["small_sold", "large_sold", "xl_sold"]

# Create histograms showing the distributions cols_with_missing
avocados_2016[cols_with_missing].plot(kind="hist")

# Fill in missing values with 0
avocados_filled = avocados_2016.fillna(0)

# Create histograms of the filled columns
avocados_filled[cols_with_missing].hist()

# Show the plot
plt.show()
```

List of dictionaries

```
In [ ]: # Create a list of dictionaries with new data
avocados_list = [
    {"date": "2019-11-03", "small_sold": 10376832, "large_sold": 7835071},
    {"date": "2019-11-10", "small_sold": 10717154, "large_sold": 8561348},
]

# Convert list into DataFrame
avocados_2019 = pd.DataFrame(avocados_list)

# Print the new DataFrame
print(avocados_2019)
```

	date	small_sold	large_sold
0	2019-11-03	10376832	7835071
1	2019-11-10	10717154	8561348

Dictionary of lists

```
In [ ]: # Create a dictionary of lists with new data
avocados_dict = {
    "date": ["2019-11-17", "2019-12-01"],
    "small_sold": [10859987, 9291631],
    "large_sold": [7674135, 6238096]
}

# Convert dictionary into DataFrame
```

```
avocados_2019 = pd.DataFrame(avocados_dict)
```

```
# Print the new DataFrame
print(avocados_2019)
```

```
    date  small_sold  large_sold
0  2019-11-17     10859987      7674135
1  2019-12-01      9291631      6238096
```

CSV to DataFrame

```
In [ ]: # Read CSV as DataFrame called airline_bumping
airline_bumping = pd.read_csv("airline_bumping.csv")

# Take a look at the DataFrame
print(airline_bumping.head())

# For each airline, select nb_bumped and total_passengers and sum
airline_totals = airline_bumping.groupby("airline")[["nb_bumped", "total_passenger"]].sum()

# Create new col, bumps_per_10k: no. of bumps per 10k passengers for each airline
airline_totals["bumps_per_10k"] = airline_totals["nb_bumped"] / airline_totals["total_passenger"] * 10000

# Print airline_totals
print(airline_totals)
```

DataFrame to CSV

```
In [ ]: # Create airline_totals_sorted
airline_totals_sorted = airline_totals.sort_values("bumps_per_10k", ascending=False)

# Print airline_totals_sorted
print(airline_totals_sorted)

# Save as airline_totals_sorted.csv
airline_totals_sorted.to_csv("airline_totals_sorted.csv")
```

# Chapter 1 Data Merging Basics

Learn how you can merge disparate data using inner joins. By combining information from multiple sources you'll uncover compelling insights that may have previously been hidden. You'll also learn how the relationship between those sources, such as one-to-one or one-to-many, can affect your result.

[Link for reference](#)

```
In [ ]: import pandas as pd

#assign name a your file and paste the pathway of the file
taxi_owners = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
taxi_veh = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
wards = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
census = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
licenses = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
biz_owners = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
ridership = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
cal = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
stations = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
land_use = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
```

Your first inner join

```
In [ ]: # Merge the taxi_owners and taxi_veh tables
taxi_own_veh = taxi_owners.merge(taxi_veh, on='vid')

# Print the column names of the taxi_own_veh
print(taxi_own_veh.columns)

Index(['rid', 'vid', 'owner_x', 'address', 'zip', 'make', 'model', 'year',
       'fuel_type', 'owner_y'],
      dtype='object')
```

```
In [ ]: # Merge the taxi_owners and taxi_veh tables setting a suffix
taxi_own_veh = taxi_owners.merge(taxi_veh, on='vid', suffixes=('_own', '_veh'))

# Print the column names of taxi_own_veh
print(taxi_own_veh.columns)

Index(['rid', 'vid', 'owner_own', 'address', 'zip', 'make', 'model', 'year',
       'fuel_type', 'owner_veh'],
      dtype='object')
```

```
In [ ]: # Print the value_counts to find the most popular fuel_type  
print(taxi_own_veh['fuel_type'].value_counts())
```

```
fuel_type  
HYBRID           2792  
GASOLINE          611  
FLEX FUEL          89  
COMPRESSED NATURAL GAS    27  
Name: count, dtype: int64
```

Inner joins and number of rows returned

```
In [ ]: # Merge the wards and census tables on the ward column  
wards_census = wards.merge(census, on='ward')  
  
# Print the shape of wards_census  
print('wards_census table shape:', wards_census.shape)
```

wards\_census table shape: (50, 9)

Inner joins and number of rows returned 2

```
In [ ]: # Print the first few rows of the census_altered table to view the change  
print(census_altered[['ward']].head())  
  
# Merge the wards and census_altered tables on the ward column  
wards_census_altered = wards.merge(census_altered, on='ward')  
  
# Print the shape of wards_census_altered  
print('wards_census_altered table shape:', wards_census_altered.shape)
```

```
-----  
NameError                                                 Traceback (most recent call last)  
Cell In[52], line 2  
      1 # Print the first few rows of the census_altered table to view the change  
----> 2 print(census_altered[['ward']].head())  
      4 # Merge the wards and census_altered tables on the ward column  
      5 wards_census_altered = wards.merge(census_altered, on='ward')  
  
NameError: name 'census_altered' is not defined
```

```
In [ ]: # Print the first few rows of the census_altered table to view the change  
print(census_altered[['ward']].head())  
  
# Merge the wards and census_altered tables on the ward column  
wards_census_altered = wards.merge(census_altered, on='ward')  
  
# Print the shape of wards_census_altered  
print('wards_census_altered table shape:', wards_census_altered.shape)
```

Drag the items into the correct bucket

Drop items here

### One-to-one

The relationship between `customer` and `cust_tax_info`.

The relationship between `products` and `inventory`.

### One-to-many

The relationship between the `customers` and `orders`.

The relationship between the `products` and `orders`.

```
In [ ]: # Merge the licenses and biz_owners table on account
licenses_owners = licenses.merge(biz_owners, on='account')

# Group the results by title then count the number of accounts
counted_df = licenses_owners.groupby('title').agg({'account':'count'})

# Sort the counted_df in descending order
sorted_df = counted_df.sort_values(by='account', ascending=False)

# Use .head() method to print the first few rows of sorted_df
print(sorted_df.head())
```

	account
title	
PRESIDENT	6259
SECRETARY	5205
SOLE PROPRIETOR	1658
OTHER	1200
VICE PRESIDENT	970

Total riders in a month

```
In [ ]: # Merge the ridership and cal tables
ridership_cal = ridership.merge(cal, on=['year', 'month', 'day'])
```

```
In [ ]: # Merge the ridership, cal, and stations tables
ridership_cal_stations = ridership.merge(cal, on=['year','month','day']) \
    .merge(stations, on='station_id')
```

```
In [ ]: # Merge the ridership, cal, and stations tables
ridership_cal_stations = ridership.merge(cal, on=['year','month','day']) \
    .merge(stations, on='station_id')

# Create a filter to filter ridership_cal_stations
filter_criteria = ((ridership_cal_stations['month'] == 7) \
    & (ridership_cal_stations['day_type'] == 'Weekday') \
    & (ridership_cal_stations['station_name'] == 'Wilson'))

# Use .loc and the filter to select for rides
print(ridership_cal_stations.loc[filter_criteria, 'rides'].sum())
```

140005

Three table merge

```
In [ ]: # Merge licenses and zip_demo, on zip; and merge the wards on ward
licenses_zip_ward = licenses.merge(zip_demo, on='zip').merge(wards, on='ward')
```

```
# Print the results by alderman and show median income
print(licenses_zip_ward.groupby('alderman').agg({'income':'median'}))
```

One-to-many merge with multiple tables

```
In [ ]: # Merge Land_use and census and merge result with Licenses including suffixes
land_cen_lic = land_use.merge(census, on='ward').merge(licenses, on='ward', suffixes=('_cen', '_lic'))
```

```
In [ ]: # Merge Land_use and census and merge result with Licenses including suffixes
land_cen_lic = land_use.merge(census, on='ward') \
    .merge(licenses, on='ward', suffixes=('_cen','_lic'))

# Group by ward, pop_2010, and vacant, then count the # of accounts
pop_vac_lic = land_cen_lic.groupby(['ward','pop_2010','vacant'],
                                   as_index=False).agg({'account':'count'})
```

```
In [ ]: # Merge Land_use and census and merge result with Licenses including suffixes
land_cen_lic = land_use.merge(census, on='ward') \
    .merge(licenses, on='ward', suffixes=('_cen','_lic'))

# Group by ward, pop_2010, and vacant, then count the # of accounts
pop_vac_lic = land_cen_lic.groupby(['ward','pop_2010','vacant'],
                                   as_index=False).agg({'account':'count'})

# Sort pop_vac_lic and print the results
sorted_pop_vac_lic = pop_vac_lic.sort_values(['vacant', 'account', 'pop_2010'], ascending=[False, True, True])

# Print the top few rows of sorted_pop_vac_lic
print(sorted_pop_vac_lic.head())
```

	ward	pop_2010	vacant	account
47	7	51581	19	80
12	20	52372	15	123
1	10	51535	14	130
16	24	54909	13	98
7	16	51954	13	156

```
In [ ]:
```

# Chapter 2 Merging Tables With Different Join Types

Take your knowledge of joins to the next level. In this chapter, you'll work with TMDb movie data as you learn about left, right, and outer joins. You'll also discover how to merge a table to itself and merge on a DataFrame index.

[Link for reference](#)

```
In [ ]: import pandas as pd

#assign name a your file and paste the pathway of the file
movies = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
taglines = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
financials = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
movie_to_genres = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
crews = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
ratings = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
sequels = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cour
```

Counting missing rows with left join

```
In [ ]: movies_taglines = movies.merge(taglines, on='id', how='left')
print(movies_taglines.head())
```

```
      id          title  popularity release_date \
0    257        Oliver Twist    20.415572  2005-09-23
1  14290  Better Luck Tomorrow    3.877036  2002-01-12
2   38365         Grown Ups    38.864027  2010-06-24
3    9672        Infamous     3.680896  2006-11-16
4   12819  Alpha and Omega    12.300789  2010-09-17
```

```
                    tagline
0                      NaN
1  Never underestimate an overachiever.
2  Boys will be boys. . . some longer than others.
3  There's more to the story than you know
4  A Pawsome 3D Adventure
```

```
In [ ]: # Merge movies and financials with a left join
movies_financials = movies.merge(financials, how='left', on='id')
```

```
In [ ]: # Merge the movies table with the financials table with a left join
movies_financials = movies.merge(financials, on='id', how='left')

# Count the number of rows in the budget column that are missing
number_of_missing_fin = movies_financials['budget'].isnull().sum()
```

```
# Print the number of movies missing financials
print(number_of_missing_fin)
```

1574

Right join to find unique movies

```
In [ ]: # Merge action_movies to scifi_movies with right join
action_scifi = action_movies.merge(scifi_movies, how='right', on='movie_id')
```

```
In [ ]: # Merge action_movies to scifi_movies with right join
action_scifi = action_movies.merge(scifi_movies, on='movie_id', how='right', suffixes=['_act', '_sci'])

# Print the first few rows of action_scifi to see the structure
print(action_scifi.head())
```

```
In [ ]: # Merge action_movies to scifi_movies with right join
action_scifi = action_movies.merge(scifi_movies, on='movie_id', how='right', suffixes=['_act', '_sci'])

# Print the first few rows of action_scifi to see the structure
print(action_scifi.head())
```

```
In [ ]: # Merge action_movies to scifi_movies with right join
action_scifi = action_movies.merge(scifi_movies, on='movie_id', how='right', suffixes=['_act', '_sci'])

# Print the first few rows of action_scifi to see the structure
print(action_scifi.head())
```

```
In [ ]: # Merge action_movies to the scifi_movies with right join
action_scifi = action_movies.merge(scifi_movies, on='movie_id', how='right',
                                   suffixes=('_act','_sci'))

# From action_scifi, select only the rows where the genre_act column is null
scifi_only = action_scifi[action_scifi['genre_act'].isnull()]

# Merge the movies and scifi_only tables with an inner join
movies_and_scifi_only = movies.merge(scifi_only, how='inner',
                                      left_on='id', right_on='movie_id')

# Print the first few rows and shape of movies_and_scifi_only
print(movies_and_scifi_only.head())
print(movies_and_scifi_only.shape)
```

Popular genres with right join

```
In [ ]: print(movie_to_genres.head())
print(movies.head())
```

```
      movie_id      genre
0            5      Crime
1            5     Comedy
2           11  Science Fiction
3           11       Action
4           11    Adventure
      id      title  popularity release_date
0   257   Oliver Twist    20.415572  2005-09-23
1  14290  Better Luck Tomorrow    3.877036  2002-01-12
2  38365        Grown Ups    38.864027  2010-06-24
3  9672        Infamous    3.680896  2006-11-16
4  12819  Alpha and Omega    12.300789  2010-09-17
```

```
In [ ]: # Use right join to merge the movie_to_genres and pop_movies tables
genres_movies = movie_to_genres.merge(pop_movies, how='right', left_on='movie_id', right_on='id')

# Count the number of genres
genre_count = genres_movies.groupby('genre').agg({'id':'count'})

# Plot a bar chart of the genre_count
genre_count.plot(kind='bar')
plt.show()
```

Using outer join to select actors

```
In [ ]: # Merge iron_1_actors to iron_2_actors on id with outer join using suffixes
iron_1_and_2 = iron_1_actors.merge(iron_2_actors,
                                    how='outer',
                                    on='id',
                                    suffixes=['_1', '_2'])

# Create an index that returns true if name_1 or name_2 are null
m = ((iron_1_and_2['name_1'].isnull() |
      (iron_1_and_2['name_2'].isnull())))

# Print the first few rows of iron_1_and_2
print(iron_1_and_2[m].head())
```

Self join

```
In [ ]: # Merge the crews table to itself
crews_self_merged = crews.merge(crews, on='id', suffixes=('_dir', '_crew'))
```

```
In [ ]: # Merge the crews table to itself
crews_self_merged = crews.merge(crews, on='id', how='inner',
                               suffixes=('_dir','_crew'))

# Create a Boolean index to select the appropriate
boolean_filter = ((crews_self_merged['job_dir'] == 'Director') &
                  (crews_self_merged['job_crew'] != 'Director'))
direct_crews = crews_self_merged[boolean_filter]
```

```
In [ ]: # Merge the crews table to itself
crews_self_merged = crews.merge(crews, on='id', how='inner',
                               suffixes=('_dir','_crew'))

# Create a boolean index to select the appropriate rows
boolean_filter = ((crews_self_merged['job_dir'] == 'Director') &
                  (crews_self_merged['job_crew'] != 'Director'))
direct_crews = crews_self_merged[boolean_filter]

# Print the first few rows of direct_crews
print(direct_crews.head())
```

	id	department_dir	job_dir	name_dir	department_crew	name_crew
156	19995	Directing	Director	James Cameron	Editing	Editor Stephen E. Rivkin
157	19995	Directing	Director	James Cameron	Sound	Sound Designer Christopher Boyes
158	19995	Directing	Director	James Cameron	Production	Casting Mali Finn
160	19995	Directing	Director	James Cameron	Writing	Writer James Cameron
161	19995	Directing	Director	James Cameron	Art	Set Designer Richard F. Mays

Index merge for movie ratings

```
In [ ]: # Merge to the movies table the ratings table on the index
movies_ratings = movies.merge(ratings, how='left', on='id')

# Print the first few rows of movies_ratings
print(movies_ratings.head())
```

```

      id          title  popularity release_date  vote_average \
0   257       Oliver Twist  20.415572  2005-09-23        6.7
1  14290  Better Luck Tomorrow  3.877036  2002-01-12        6.5
2  38365        Grown Ups  38.864027  2010-06-24        6.0
3   9672        Infamous  3.680896  2006-11-16        6.4
4  12819  Alpha and Omega  12.300789  2010-09-17        5.3

      vote_count
0      274.0
1      27.0
2     1705.0
3      60.0
4     124.0

```

Do sequels earn more?

```
In [ ]: # Merge sequels and financials on index id
sequels_fin = sequels.merge(financials, on='id', how='left')
```

```
In [ ]: # Merge sequels and financials on index id
sequels_fin = sequels.merge(financials, on='id', how='left')

# Self merge with suffixes as inner join with left on sequel and right on id
orig_seq = sequels_fin.merge(sequels_fin, how='inner', left_on='sequel',
                             right_on='id', right_index=True,
                             suffixes=('_org', '_seq'))

# Add calculation to subtract revenue_org from revenue_seq
orig_seq['diff'] = orig_seq['revenue_seq'] - orig_seq['revenue_org']
```

```
In [ ]: # Merge sequels and financials on index id
sequels_fin = sequels.merge(financials, on='id', how='left')

# Self merge with suffixes as inner join with left on sequel and right on id
orig_seq = sequels_fin.merge(sequels_fin, how='inner', left_on='sequel',
                             right_on='id', right_index=True,
                             suffixes=('_org', '_seq'))

# Add calculation to subtract revenue_org from revenue_seq
orig_seq['diff'] = orig_seq['revenue_seq'] - orig_seq['revenue_org']

# Select the title_org, title_seq, and diff
titles_diff = orig_seq[['title_org', 'title_seq', 'diff']]
```

```
In [ ]: # Merge sequels and financials on index id
sequels_fin = sequels.merge(financials, on='id', how='left')

# Self merge with suffixes as inner join with left on sequel and right on id
```

```

orig_seq = sequels_fin.merge(sequels_fin, how='inner', left_on='sequel',
                             right_on='id', right_index=True,
                             suffixes=('_org','_seq'))

# Add calculation to subtract revenue_org from revenue_seq
orig_seq['diff'] = orig_seq['revenue_seq'] - orig_seq['revenue_org']

# Select the title_org, title_seq, and diff
titles_diff = orig_seq[['title_org','title_seq','diff']]

# Print the first rows of the sorted titles_diff
print(titles_diff.sort_values(by='diff', ascending=False).head())

```

	title_org	title_seq	\
2929	Before Sunrise	The Amazing Spider-Man 2	
1256	Star Trek III: The Search for Spock	The Matrix	
293	Indiana Jones and the Temple of Doom	Man of Steel	
1084	Saw	Superman Returns	
1334	The Terminator	Star Trek Beyond	

	diff
2929	700182027.0
1256	376517383.0
293	329845518.0
1084	287169523.0
1334	265100616.0

In [ ]:

## Chapter 3 Advanced Merging and Concatenating

In this chapter, you'll leverage powerful filtering techniques, including semi-joins and anti-joins. You'll also learn how to glue DataFrames by vertically combining and using the pandas.concat function to create new datasets. Finally, because data is rarely clean, you'll also learn how to validate your newly combined data structures.

[Link for reference](#)

In [ ]:

```

import pandas as pd

#assign name a your file and paste the pathway of the file

#for ".csv" files
gdp = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Da
sp500 = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course

```

```
# for ".p" files
stations = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python C
```

## Performing an anti join

```
In [ ]: # Merge employees and top_cust  
empl_cust = employees.merge(top_cust, on='srid',  
                           how='left', indicator=True)
```

```
In [ ]: # Merge employees and top_cust
empl_cust = employees.merge(top_cust, on='srid',
                            how='left', indicator=True)

# Select the srid column where _merge is left_only
srid_list = empl_cust.loc[empl_cust['_merge']=='left_only', 'srid']
```

```
In [ ]: # Merge employees and top_cust
empl_cust = employees.merge(top_cust, on='srid',
                            how='left', indicator=True)

# Select the srid column where _merge is left_only
srid_list = empl_cust.loc[empl_cust['_merge'] == 'left_only', 'srid']

# Get employees not working with top customers
print(employees[employees['srid'].isin(srid_list)])
```

## Performing a semi join

```
In [ ]: # Merge the non_mus_tcks and top_invoices tables on tid
tracks_invoices = non_mus_tcks.merge(top_invoices, on='tid', how='inner')

# Use .isin() to subset non_mus_tcks to rows with tid in tracks_invoices
top_tracks = non_mus_tcks[non_mus_tcks['tid'].isin(tracks_invoices['tid'])]

# Group the top_tracks by gid and count the tid rows
cnt_by_gid = top_tracks.groupby(['gid'], as_index=False).agg({'tid':'count'})

# Merge the genres table to cnt_by_gid on gid and print
print(cnt_by_gid.merge(genres, on='gid'))
```

## Concatenation basics

```
In [ ]: # Concatenate the tracks so the index goes from 0 to n-1
tracks_from_albums = pd.concat([tracks_master, tracks_ride, tracks_st],
                               ignore_index=True,
                               sort=True)
print(tracks_from_albums)
```

```
In [ ]: # Concatenate the tracks, show only columns names that are in all tables
tracks_from_albums = pd.concat([tracks_master, tracks_ride, tracks_st],
                               join='inner',
                               sort=True)
print(tracks_from_albums)
```

Concatenating with keys

```
In [ ]: # Concatenate the tables and add keys
inv_jul_thr_sep = pd.concat([inv_jul, inv_aug, inv_sep],
                            keys=['7Jul', '8Aug', '9Sep'])

# Group the invoices by the index keys and find avg of the total column
avg_inv_by_month = inv_jul_thr_sep.groupby(level=0).agg({'total':'mean'})

# Bar plot of avg_inv_by_month
avg_inv_by_month.plot(kind='bar')
plt.show()
```

Concatenate and merge to find common songs

```
In [ ]: # Concatenate the classic tables vertically
classic_18_19 = pd.concat([classic_18, classic_19], ignore_index=True)

# Concatenate the pop tables vertically
pop_18_19 = pd.concat([pop_18, pop_19], ignore_index=True)
```

```
In [ ]: # Concatenate the classic tables vertically
classic_18_19 = pd.concat([classic_18, classic_19], ignore_index=True)

# Concatenate the pop tables vertically
pop_18_19 = pd.concat([pop_18, pop_19], ignore_index=True)

# Merge classic_18_19 with pop_18_19
classic_pop = classic_18_19.merge(pop_18_19, on='tid', how='inner')

# Using .isin(), filter classic_18_19 rows where tid is in classic_pop
popular_classic = classic_18_19[classic_18_19['tid'].isin(classic_pop['tid'])]

# Print popular chart
print(popular_classic)
```

# Chapter 4 - Merging Ordered and Time-Series Data

In this final chapter, you'll step up a gear and learn to apply pandas' specialized methods for merging time-series and ordered data together with real-world financial and economic data from the city of Chicago. You'll also learn how to query resulting tables using a SQL-style format, and unpivot data using the melt method.

[Link for reference](#)

In [ ]:

```
import pandas as pd

# for ".csv" files
gdp = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Da
sp500 = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
pop = pd.read_csv("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Da

# for ".p" files
stations = pd.read_pickle("C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python C

print(gdp.head())
print(sp500.head())
print(stations.head())
```

	Country	Name	Country	Code	Indicator	Name	Year	GDP
0	China		CHN	GDP (current US\$)	(current US\$)	2010	6.087160e+12	
1	Germany		DEU	GDP (current US\$)	(current US\$)	2010	3.417090e+12	
2	Japan		JPN	GDP (current US\$)	(current US\$)	2010	5.700100e+12	
3	United States		USA	GDP (current US\$)	(current US\$)	2010	1.499210e+13	
4	China		CHN	GDP (current US\$)	(current US\$)	2011	7.551500e+12	

	Date	Returns
0	2008	-38.49
1	2009	23.45
2	2010	12.78
3	2011	0.00
4	2012	13.41

	station_id	station_name	location
0	40010	Austin-Forest Park	(41.870851, -87.776812)
1	40020	Harlem-Lake	(41.886848, -87.803176)
2	40030	Pulaski-Lake	(41.885412, -87.725404)
3	40040	Quincy/Wells	(41.878723, -87.63374)
4	40050	Davis	(42.04771, -87.683543)

Correlation between GDP and S&P500

In [ ]:

```
# Use merge_ordered() to merge gdp and sp500 on year and date
gdp_sp500 = pd.merge_ordered(gdp, sp500, left_on='Year', right_on='Date',
```

```
    how='left')

# Print gdp_sp500
print(gdp_sp500)
```

	Country Name	Country Code	Indicator Name	Year	GDP	Date	\
0	China	CHN	GDP (current US\$)	2010	6.087160e+12	2010.0	
1	Germany	DEU	GDP (current US\$)	2010	3.417090e+12	2010.0	
2	Japan	JPN	GDP (current US\$)	2010	5.700100e+12	2010.0	
3	United States	USA	GDP (current US\$)	2010	1.499210e+13	2010.0	
4	China	CHN	GDP (current US\$)	2011	7.551500e+12	2011.0	
5	Germany	DEU	GDP (current US\$)	2011	3.757700e+12	2011.0	
6	Japan	JPN	GDP (current US\$)	2011	6.157460e+12	2011.0	
7	United States	USA	GDP (current US\$)	2011	1.554260e+13	2011.0	
8	China	CHN	GDP (current US\$)	2012	8.532230e+12	2012.0	
9	Germany	DEU	GDP (current US\$)	2012	3.543980e+12	2012.0	
10	Japan	JPN	GDP (current US\$)	2012	6.203210e+12	2012.0	
11	United States	USA	GDP (current US\$)	2012	1.619700e+13	2012.0	
12	China	CHN	GDP (current US\$)	2012	8.532230e+12	2012.0	
13	Germany	DEU	GDP (current US\$)	2012	3.543980e+12	2012.0	
14	Japan	JPN	GDP (current US\$)	2012	6.203210e+12	2012.0	
15	United States	USA	GDP (current US\$)	2012	1.619700e+13	2012.0	
16	China	CHN	GDP (current US\$)	2013	9.570410e+12	2013.0	
17	Germany	DEU	GDP (current US\$)	2013	3.752510e+12	2013.0	
18	Japan	JPN	GDP (current US\$)	2013	5.155720e+12	2013.0	
19	United States	USA	GDP (current US\$)	2013	1.678480e+13	2013.0	
20	China	CHN	GDP (current US\$)	2014	1.043850e+13	2014.0	
21	Germany	DEU	GDP (current US\$)	2014	3.898730e+12	2014.0	
22	Japan	JPN	GDP (current US\$)	2014	4.850410e+12	2014.0	
23	United States	USA	GDP (current US\$)	2014	1.752170e+13	2014.0	
24	China	CHN	GDP (current US\$)	2015	1.101550e+13	2015.0	
25	Germany	DEU	GDP (current US\$)	2015	3.381390e+12	2015.0	
26	Japan	JPN	GDP (current US\$)	2015	4.389480e+12	2015.0	
27	United States	USA	GDP (current US\$)	2015	1.821930e+13	2015.0	
28	China	CHN	GDP (current US\$)	2016	1.113790e+13	2016.0	
29	Germany	DEU	GDP (current US\$)	2016	3.495160e+12	2016.0	
30	Japan	JPN	GDP (current US\$)	2016	4.926670e+12	2016.0	
31	United States	USA	GDP (current US\$)	2016	1.870720e+13	2016.0	
32	China	CHN	GDP (current US\$)	2017	1.214350e+13	2017.0	
33	Germany	DEU	GDP (current US\$)	2017	3.693200e+12	2017.0	
34	Japan	JPN	GDP (current US\$)	2017	4.859950e+12	2017.0	
35	United States	USA	GDP (current US\$)	2017	1.948540e+13	2017.0	
36	China	CHN	GDP (current US\$)	2018	1.360820e+13	NaN	
37	Germany	DEU	GDP (current US\$)	2018	3.996760e+12	NaN	
38	Japan	JPN	GDP (current US\$)	2018	4.970920e+12	NaN	
39	United States	USA	GDP (current US\$)	2018	2.049410e+13	NaN	

### Returns

0	12.78
1	12.78
2	12.78
3	12.78
4	0.00
5	0.00
6	0.00

```
7    0.00
8    13.41
9    13.41
10   13.41
11   13.41
12   13.41
13   13.41
14   13.41
15   13.41
16   29.60
17   29.60
18   29.60
19   29.60
20   11.39
21   11.39
22   11.39
23   11.39
24   -0.73
25   -0.73
26   -0.73
27   -0.73
28   9.54
29   9.54
30   9.54
31   9.54
32   19.42
33   19.42
34   19.42
35   19.42
36   NaN
37   NaN
38   NaN
39   NaN
```

```
In [ ]: # Use merge_ordered() to merge gdp and sp500, interpolate missing value
gdp_sp500 = pd.merge_ordered(gdp, sp500, left_on='Year', right_on='Date', how='left', fill_method='ffill')

# Print gdp_sp500
print (gdp_sp500.head(10))
```

	Country Name	Country Code	Indicator Name	Year	GDP	Date	\
0	China	CHN	GDP (current US\$)	2010	6.087160e+12	2010	
1	Germany	DEU	GDP (current US\$)	2010	3.417090e+12	2010	
2	Japan	JPN	GDP (current US\$)	2010	5.700100e+12	2010	
3	United States	USA	GDP (current US\$)	2010	1.499210e+13	2010	
4	China	CHN	GDP (current US\$)	2011	7.551500e+12	2011	
5	Germany	DEU	GDP (current US\$)	2011	3.757700e+12	2011	
6	Japan	JPN	GDP (current US\$)	2011	6.157460e+12	2011	
7	United States	USA	GDP (current US\$)	2011	1.554260e+13	2011	
8	China	CHN	GDP (current US\$)	2012	8.532230e+12	2012	
9	Germany	DEU	GDP (current US\$)	2012	3.543980e+12	2012	

Returns

0	12.78
1	12.78
2	12.78
3	12.78
4	0.00
5	0.00
6	0.00
7	0.00
8	13.41
9	13.41

```
In [ ]: # Use merge_ordered() to merge gdp and sp500, interpolate missing value
gdp_sp500 = pd.merge_ordered(gdp, sp500, left_on='Year', right_on='Date',
                             how='left', fill_method='ffill')
```

```
# Subset the gdp and returns columns
gdp_returns = gdp_sp500[['GDP', 'Returns']]

# Print gdp_returns correlation
print (gdp_returns.corr())
```

	GDP	Returns
GDP	1.000000	0.040669
Returns	0.040669	1.000000

Phillips curve using merge\_ordered()

```
In [ ]: # Use merge_ordered() to merge inflation, unemployment with inner join
inflation_unemploy = pd.merge_ordered(inflation, unemployment, on='Date', how='inner')

# Print inflation_unemploy
print(inflation_unemploy)

# Plot a scatter plot of unemployment_rate vs cpi of inflation_unemploy
inflation_unemploy.plot(kind='scatter', x='unemployment_rate', y='cpi')
plt.show()
```

## merge\_ordered() caution, multiple columns

In [ ]:

```
print(gdp.head())
print(pop.head())

# Merge gdp and pop on date and country with fill and notice rows 2 and 3
ctry_date = pd.merge_ordered(gdp, pop, on=['Year', 'Country Name'],
                             fill_method='ffill')

# Print ctry_date
print(ctry_date)
```

	Country Name	Country Code	Indicator Name	Year	GDP
0	China	CHN	GDP (current US\$)	2010	6.087160e+12
1	Germany	DEU	GDP (current US\$)	2010	3.417090e+12
2	Japan	JPN	GDP (current US\$)	2010	5.700100e+12
3	United States	USA	GDP (current US\$)	2010	1.499210e+13
4	China	CHN	GDP (current US\$)	2011	7.551500e+12
	Country Name	Country Code	Indicator Name	Year	Pop
0	Aruba	ABW	Population, total	2010	101669.0
1	Afghanistan	AFG	Population, total	2010	29185507.0
2	Angola	AGO	Population, total	2010	23356246.0
3	Albania	ALB	Population, total	2010	2913021.0
4	Andorra	AND	Population, total	2010	84449.0
	Country Name	Country Code_x	Indicator Name_x	Year	\
0	Afghanistan	NaN	NaN	2010	
1	Albania	NaN	NaN	2010	
2	Algeria	NaN	NaN	2010	
3	American Samoa	NaN	NaN	2010	
4	Andorra	NaN	NaN	2010	
...	...	...	...	...	...
2643	West Bank and Gaza	USA	GDP (current US\$)	2018	
2644	World	USA	GDP (current US\$)	2018	
2645	Yemen, Rep.	USA	GDP (current US\$)	2018	
2646	Zambia	USA	GDP (current US\$)	2018	
2647	Zimbabwe	USA	GDP (current US\$)	2018	
	GDP	Country Code_y	Indicator Name_y	Pop	
0	Nan	AFG	Population, total	2.918551e+07	
1	Nan	ALB	Population, total	2.913021e+06	
2	Nan	DZA	Population, total	3.597746e+07	
3	Nan	ASM	Population, total	5.607900e+04	
4	Nan	AND	Population, total	8.444900e+04	
...	...	...	...	...	...
2643	2.049410e+13	PSE	Population, total	4.569087e+06	
2644	2.049410e+13	WLD	Population, total	7.594270e+09	
2645	2.049410e+13	YEM	Population, total	2.849869e+07	
2646	2.049410e+13	ZMB	Population, total	1.735182e+07	
2647	2.049410e+13	ZWE	Population, total	1.443902e+07	

[2648 rows x 8 columns]

```
In [ ]: # Merge gdp and pop on country and date with fill
date_ctry = pd.merge_ordered(gdp, pop, on=['Country Name', 'Year'],
                             fill_method='ffill')

# Print date_ctry
print(date_ctry)
```

	Country Name	Country Code_x	Indicator	Name_x	Year	GDP	\
0	Afghanistan	NaN		NaN	2010	NaN	
1	Afghanistan	NaN		NaN	2011	NaN	
2	Afghanistan	NaN		NaN	2012	NaN	
3	Afghanistan	NaN		NaN	2012	NaN	
4	Afghanistan	NaN		NaN	2013	NaN	
...	...	...		...	...	...	
2643	Zimbabwe	USA	GDP (current US\$)	2014	2.049410e+13		
2644	Zimbabwe	USA	GDP (current US\$)	2015	2.049410e+13		
2645	Zimbabwe	USA	GDP (current US\$)	2016	2.049410e+13		
2646	Zimbabwe	USA	GDP (current US\$)	2017	2.049410e+13		
2647	Zimbabwe	USA	GDP (current US\$)	2018	2.049410e+13		

	Country Code_y	Indicator	Name_y	Pop
0	AFG	Population, total	29185507.0	
1	AFG	Population, total	30117413.0	
2	AFG	Population, total	31161376.0	
3	AFG	Population, total	31161376.0	
4	AFG	Population, total	32269589.0	
...	...	...	...	
2643	ZWE	Population, total	13586681.0	
2644	ZWE	Population, total	13814629.0	
2645	ZWE	Population, total	14030390.0	
2646	ZWE	Population, total	14236745.0	
2647	ZWE	Population, total	14439018.0	

[2648 rows x 8 columns]

Using merge\_asof() to study stocks

```
In [ ]: # Use merge_asof() to merge jpm and wells
jpm_wells = pd.merge_asof(jpm, wells, on='date_time',
                           suffixes=('', '_wells'), direction='nearest')

# Use merge_asof() to merge jpm_wells and bac
jpm_wells_bac = pd.merge_asof(jpm_wells, bac, on='date_time',
                           suffixes=('_jpm', '_bac'), direction='nearest')

# Compute price diff
price_diffs = jpm_wells_bac.diff()

# Plot the price diff of the close of jpm, wells and bac only
price_diffs.plot(y=['close_jpm','close_wells','close_bac'])
plt.show()
```

Using merge\_asof() to create dataset

```
In [ ]: # Merge gdp and recession on date using merge_asof()
gdp_recession = pd.merge_asof(gdp, recession, on='date')

# Create a list based on the row value of gdp_recession['econ_status']
is_recession = ['r' if s=='recession' else 'g' for s in gdp_recession['econ_status']]

# Plot a bar chart of gdp_recession
gdp_recession.plot(kind='bar', y='gdp', x='date', color=is_recession, rot=90)
plt.show()
```

Subsetting rows with .query()

```
In [ ]: # Merge gdp and pop on date and country with fill
gdp_pop = pd.merge_ordered(gdp, pop, on=['Country Name', 'Year'], fill_method='ffill')
```

```
In [ ]: # Merge gdp and pop on date and country with fill
gdp_pop = pd.merge_ordered(gdp, pop, on=['Country Name', 'Year'], fill_method='ffill')

# Add a column named gdp_per_capita to gdp_pop that divides the gdp by pop
gdp_pop['gdp_per_capita'] = gdp_pop['GDP']/gdp_pop['Pop']
```

```
In [ ]: # Merge gdp and pop on date and country with fill
gdp_pop = pd.merge_ordered(gdp, pop, on=['Country Name', 'Year'], fill_method='ffill')

# Add a column named gdp_per_capita to gdp_pop that divides the gdp by pop
gdp_pop['gdp_per_capita'] = gdp_pop['GDP']/gdp_pop['Pop']

# Pivot table of gdp_per_capita, where index is date and columns is country
gdp_pivot = gdp_pop.pivot_table('gdp_per_capita', 'Year', 'Country Name')
```

```
In [ ]: # Merge gdp and pop on date and country with fill
gdp_pop = pd.merge_ordered(gdp, pop, on=['Country Name', 'Year'], fill_method='ffill')

# Add a column named gdp_per_capita to gdp_pop that divides the gdp by pop
gdp_pop['gdp_per_capita'] = gdp_pop['GDP'] / gdp_pop['Pop']

# Pivot table of gdp_per_capita, where index is date and columns are country
gdp_pivot = gdp_pop.pivot_table('gdp_per_capita', 'Year', 'Country Name')

# Convert 'Year' to string to avoid the TypeError
recent_gdp_pop = gdp_pivot.query('Year >= 1991')

# Plot recent_gdp_pop with proper labels
recent_gdp_pop.plot(rot=90)
plt.xlabel('Year')
plt.ylabel('GDP per Capita')
```

```
plt.title('GDP per Capita Over Time')
plt.show()
```

```
-----
NameError                                 Traceback (most recent call last)
Cell In[117], line 15
    13 # Plot recent_gdp_pop with proper labels
    14 recent_gdp_pop.plot(rot=90)
--> 15 plt.xlabel('Year')
    16 plt.ylabel('GDP per Capita')
    17 plt.title('GDP per Capita Over Time')

NameError: name 'plt' is not defined
```

Country Name

- China
- Colombia
- Comoros
- Congo, Dem. Rep.
- Congo, Rep.
- Costa Rica
- Cote d'Ivoire
- Croatia
- Cuba
- Curacao
- Cyprus
- Czech Republic
- Denmark
- Djibouti
- Dominica
- Dominican Republic
- Early-demographic dividend
- East Asia & Pacific
- East Asia & Pacific (IDA & IBRD countries)
- East Asia & Pacific (excluding high income)
- Ecuador
- Egypt, Arab Rep.
- El Salvador
- Equatorial Guinea
- Eritrea
- Estonia
- Eswatini
- Ethiopia
- Euro area
- Europe & Central Asia
- Europe & Central Asia (IDA & IBRD countries)
- Europe & Central Asia (excluding high income)
- European Union
- Faroe Islands
- Fiji
- Finland
- Fragile and conflict affected situations
- France

- France
- French Polynesia
- Gabon
- Gambia, The
- Georgia
- Germany
- Ghana
- Gibraltar
- Greece
- Greenland
- Grenada
- Guam
- Guatemala
- Guinea
- Guinea-Bissau
- Guyana
- Haiti
- Heavily indebted poor countries (HIPC)
- High income
- Honduras
- Hong Kong SAR, China
- Hungary
- IBRD only
- IDA & IBRD total
- IDA blend
- IDA only
- IDA total
- Iceland
- India
- Indonesia
- Iran, Islamic Rep.
- Iraq
- Ireland
- Isle of Man
- Israel
- Italy
- Jamaica
- Japan
- Jordan
- Kazakhstan

Kazakhstan

Kenya

Kiribati

In [ ]:

```
# Unpivot everything besides the year column
ur_tall = ur_wide.melt(id_vars=['year'], var_name='month',
                        value_name='unempl_rate')

# Create a date column using the month and year columns of ur_tall
ur_tall['date'] = pd.to_datetime(ur_tall['month'] + '-' + ur_tall['year'])

# Sort ur_tall by date in ascending order
ur_sorted = ur_tall.sort_values('date')

# Plot the unempl_rate by date
ur_sorted.plot(x='date', y='unempl_rate')
plt.show()
```

Using .melt() for stocks vs bond performance

Lebanon

In [ ]:

```
# Use melt on ten_yr, unpivot everything besides the metric column
bond_perc = ten_yr.melt(id_vars='metric', var_name='date', value_name='close')

# Use query on bond_perc to select only the rows where metric=close
bond_perc_close = bond_perc.query('metric == "close"')

# Merge (ordered) dji and bond_perc_close on date with an inner join
dow_bond = pd.merge_ordered(dji, bond_perc_close, on='date',
                            suffixes=('_dow', '_bond'), how='inner')

# Plot only the close_dow and close_bond columns
dow_bond.plot(y=['close_dow', 'close_bond'], x='date', rot=90)
plt.show()
```

Malawi

Malaysia

Maldives

Mali

Malta

Marshall Islands

Mauritania

Mauritius

Mexico

Micronesia, Fed. Sts.

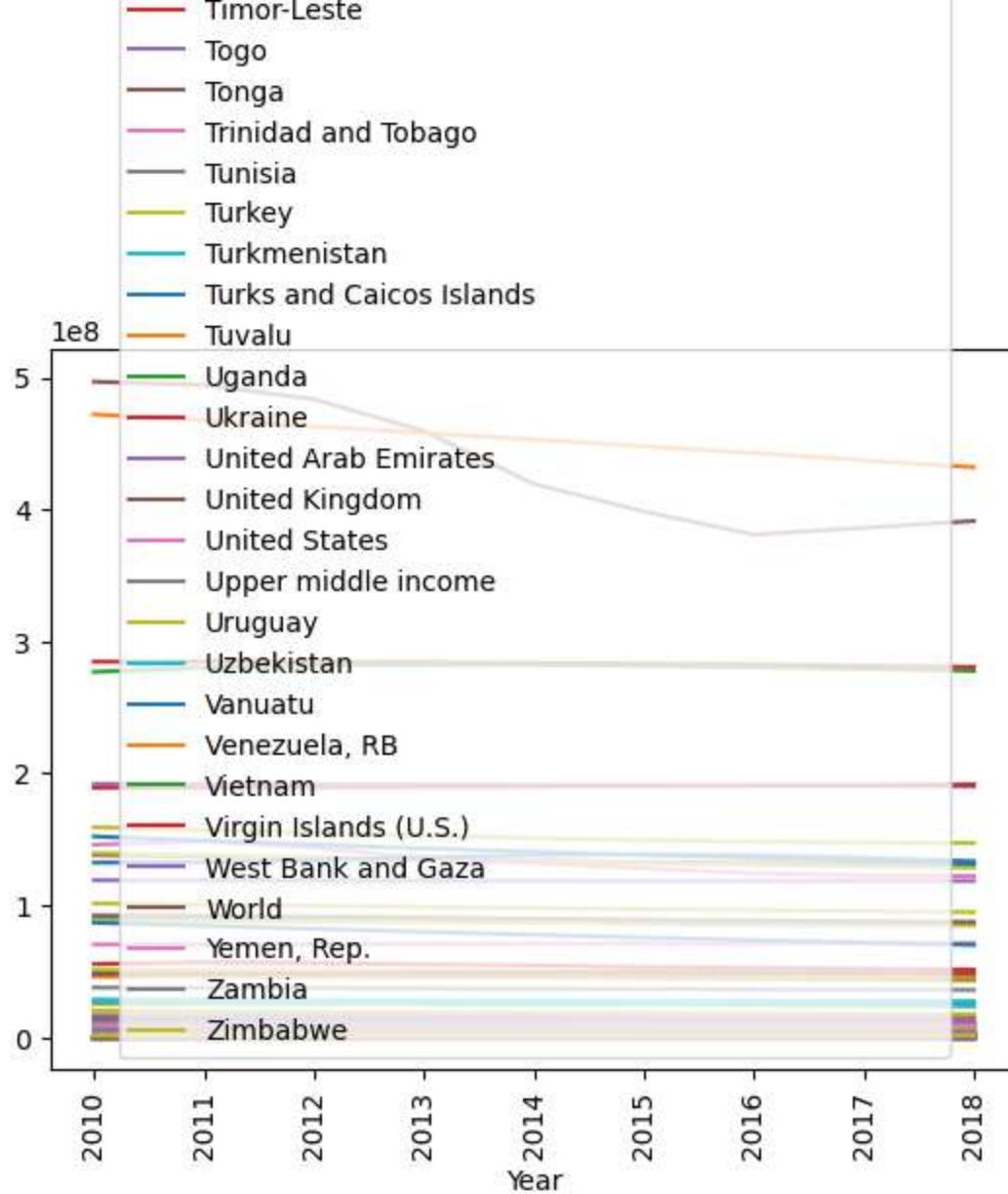
Middle East & North Africa

Middle East & North Africa (IDA & IBRD countries)

Middle East & North Africa (excluding high income)

- Middle East & North Africa (excluding high income)
- Middle income
- Moldova
- Monaco
- Mongolia
- Montenegro
- Morocco
- Mozambique
- Myanmar
- Namibia
- Nauru
- Nepal
- Netherlands
- New Caledonia
- New Zealand
- Nicaragua
- Niger
- Nigeria
- North America
- North Macedonia
- Northern Mariana Islands
- Norway
- OECD members
- Oman
- Other small states
- Pacific island small states
- Pakistan
- Palau
- Panama
- Papua New Guinea
- Paraguay
- Peru
- Philippines
- Poland
- Portugal
- Post-demographic dividend
- Pre-demographic dividend
- Puerto Rico
- Qatar

- Romania
- Russian Federation
- Rwanda
- Samoa
- San Marino
- Sao Tome and Principe
- Saudi Arabia
- Senegal
- Serbia
- Seychelles
- Sierra Leone
- Singapore
- Sint Maarten (Dutch part)
- Slovak Republic
- Slovenia
- Small states
- Solomon Islands
- Somalia
- South Africa
- South Asia
- South Asia (IDA & IBRD)
- South Sudan
- Spain
- Sri Lanka
- St. Kitts and Nevis
- St. Lucia
- St. Martin (French part)
- St. Vincent and the Grenadines
- Sub-Saharan Africa
- Sub-Saharan Africa (IDA & IBRD countries)
- Sub-Saharan Africa (excluding high income)
- Sudan
- Suriname
- Sweden
- Switzerland
- Syrian Arab Republic
- Tajikistan
- Tanzania
- Thailand



# 7. Cleaning Data in Python

## Chapter 1 - Common data problems

Categorical and text data can often be some of the messiest parts of a dataset due to their unstructured nature. In this chapter, you'll learn how to fix whitespace and capitalization inconsistencies in category labels, collapse multiple categories into one, and reformat strings for consistency.

[Link for reference](#)

```
In [ ]: #importing libraries
import pandas as pd
import datetime as dt

# Common path prefix
common_path = "C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCa

# Paths for the variables with double backslashes
airlines = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\airlines_final.csv')
banking = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\banking_dirty.csv')
restaurant = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2.csv')
restaurant_dirty = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2_dirty.csv')
ride_sharing = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\ride_sharing_new.csv')

print(airlines.head(6))
```

	Unnamed: 0	id	day	airline	destination	dest_region
0	0	1351	Tuesday	UNITED INTL	KANSAI	Asia
1	1	373	Friday	ALASKA	SAN JOSE DEL CABO	Canada/Mexico
2	2	2820	Thursday	DELTA	LOS ANGELES	West US
3	3	1157	Tuesday	SOUTHWEST	LOS ANGELES	West US
4	4	2992	Wednesday	AMERICAN	MIAMI	East US
5	5	634	Thursday	ALASKA	NEWARK	East US

	dest_size	boarding_area	dept_time	wait_min	cleanliness
0	Hub	Gates 91-102	2018-12-31	115.0	Clean
1	Small	Gates 50-59	2018-12-31	135.0	Clean
2	Hub	Gates 40-48	2018-12-31	70.0	Average
3	Hub	Gates 20-39	2018-12-31	190.0	Clean
4	Hub	Gates 50-59	2018-12-31	559.0	Somewhat clean
5	Hub	Gates 50-59	2018-12-31	140.0	Somewhat clean

	safety	satisfaction
0	Neutral	Very satisfied
1	Very safe	Very satisfied
2	Somewhat safe	Neutral
3	Very safe	Somewhat satsified
4	Very safe	Somewhat satsified
5	Very safe	Very satisfied

Numeric data or ... ?

```
In [ ]: # Print the information of ride_sharing
print(ride_sharing.info())

# Print summary statistics of user_type column
print(ride_sharing['user_type'].describe())

# Convert user_type from integer to category
ride_sharing['user_type_cat'] = ride_sharing['user_type'].astype('category')

# Write an assert statement confirming the change
assert ride_sharing['user_type_cat'].dtype == 'category'

# Print new summary statistics
print(ride_sharing['user_type_cat'].describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25760 entries, 0 to 25759
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Unnamed: 0        25760 non-null   int64  
 1   duration         25760 non-null   object  
 2   station_A_id    25760 non-null   int64  
 3   station_A_name  25760 non-null   object  
 4   station_B_id    25760 non-null   int64  
 5   station_B_name  25760 non-null   object  
 6   bike_id          25760 non-null   int64  
 7   user_type        25760 non-null   int64  
 8   user_birth_year  25760 non-null   int64  
 9   user_gender      25760 non-null   object  
dtypes: int64(6), object(4)
memory usage: 2.0+ MB
None
count    25760.000000
mean     2.008385
std      0.704541
min      1.000000
25%      2.000000
50%      2.000000
75%      3.000000
max      3.000000
Name: user_type, dtype: float64
count    25760
unique   3
top      2
freq     12972
Name: user_type_cat, dtype: int64
```

Summing strings and concatenating numbers

```
In [ ]: # Strip duration of minutes
ride_sharing['duration_trim'] = ride_sharing['duration'].str.strip('minutes')

# Convert duration to integer
ride_sharing['duration_time'] = ride_sharing['duration_trim'].astype('int')

# Write an assert statement making sure of conversion
assert ride_sharing['duration_time'].dtype == 'int'

# Print formed columns and calculate average ride duration
print(ride_sharing[['duration','duration_trim','duration_time']])
print(ride_sharing['duration_time'].mean())
```

```
duration duration_trim duration_time
0    12 minutes      12      12
1    24 minutes      24      24
2     8 minutes       8       8
3     4 minutes       4       4
4   11 minutes      11      11
...
25755 11 minutes      11      11
25756 10 minutes      10      10
25757 14 minutes      14      14
25758 14 minutes      14      14
25759 29 minutes      29      29
```

[25760 rows x 3 columns]

11.389052795031056

Tire size constraints

```
In [ ]: # Convert tire_sizes to integer
ride_sharing['tire_sizes'] = ride_sharing['tire_sizes'].astype('int')

# Set all values above 27 to 27
ride_sharing.loc[ride_sharing['tire_sizes'] > 27, 'tire_sizes'] = 27

# Reconvert tire_sizes back to categorical
ride_sharing['tire_sizes'] = ride_sharing['tire_sizes'].astype('category')

# Print tire size description
print(ride_sharing['tire_sizes'].head())
```

Back to the future

```
In [ ]: #import
import pandas as pd
import datetime as dt

import pandas as pd
import datetime as dt

# Convert ride_date to date
ride_sharing['ride_dt'] = pd.to_datetime(ride_sharing['ride_date'])

# Save today's date
today = pd.to_datetime(dt.date.today())

# Set all in the future to today's date
ride_sharing.loc[ride_sharing['ride_dt'] > today, 'ride_dt'] = today
```

```
# Print maximum of ride_dt column
print(ride_sharing['ride_dt'].max())
```

Finding duplicates

```
In [ ]: # Find duplicates
duplicates = ride_sharing.duplicated(subset='ride_id', keep=False)

# Sort your duplicated rides
duplicated_rides = ride_sharing[duplicates].sort_values('ride_id')

# Print relevant columns of duplicated_rides
print(duplicated_rides[['ride_id','duration','user_birth_year']])
```

Treating duplicates

```
In [ ]: # Drop complete duplicates from ride_sharing
ride_dup = ride_sharing.drop_duplicates()

# Create statistics dictionary for aggregation function
statistics = {'user_birth_year': 'min', 'duration': 'mean'}

# Group by ride_id and compute new statistics
ride_unique = ride_dup.groupby('ride_id').agg(statistics).reset_index()

# Find duplicated values again
duplicates = ride_unique.duplicated(subset = 'ride_id', keep = False)
duplicated_rides = ride_unique[duplicates == True]

# Assert duplicates are processed
assert duplicated_rides.shape[0] == 0
```

```
In [ ]:
```

## Chapter 2 - Text and categorical data problems

Categorical and text data can often be some of the messiest parts of a dataset due to their unstructured nature. In this chapter, you'll learn how to fix whitespace and capitalization inconsistencies in category labels, collapse multiple categories into one, and reformat strings for consistency.

[Link for reference](#)

```
In [ ]: #importing Libraries
import pandas as pd
import datetime as dt
```

```
# Common path prefix
common_path = "C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCa

# Paths for the variables with double backslashes
airlines = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\airlines_final.csv')
banking = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\banking_dirty.csv')
restaurant = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2.csv')
restaurant_dirty = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2_dirty.csv')
ride_sharing = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\ride_sharing_new.csv')

print(airlines.head(3))
```

	Unnamed: 0	id	day	airline	destination	dest_region	\
0	0	1351	Tuesday	UNITED INTL	KANSAI	Asia	
1	1	373	Friday	ALASKA	SAN JOSE DEL CABO	Canada/Mexico	
2	2	2820	Thursday	DELTA	LOS ANGELES	West US	

	dest_size	boarding_area	dept_time	wait_min	cleanliness	safety	\
0	Hub	Gates 91-102	2018-12-31	115.0	Clean	Neutral	
1	Small	Gates 50-59	2018-12-31	135.0	Clean	Very safe	
2	Hub	Gates 40-48	2018-12-31	70.0	Average	Somewhat safe	

	satisfaction
0	Very satisfied
1	Very satisfied
2	Neutral

Finding consistency

```
In [ ]: # Print categories DataFrame
print("categories: ")
print("*****")
```

```
# Print unique values of survey columns in airlines
print('Cleanliness: ', airlines['cleanliness'].unique(), "\n")
print('Safety: ', airlines['safety'].unique(), "\n")
print('Satisfaction: ', airlines['satisfaction'].unique(), "\n")
```

```
categories:
*****
*****
```

```
Cleanliness: ['Clean' 'Average' 'Somewhat clean' 'Somewhat dirty' 'Dirty']
```

```
Safety: ['Neutral' 'Very safe' 'Somewhat safe' 'Very unsafe' 'Somewhat unsafe']
```

```
Satisfaction: ['Very satisfied' 'Neutral' 'Somewhat satisfied' 'Somewhat unsatisfied'
 'Very unsatisfied']
```

Finding consistency 2

```
In [ ]: # Find the cleanliness category in airlines not in categories
cat_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category
cat_clean_rows = airlines['cleanliness'].isin(cat_clean)

# Print rows with inconsistent category
print(airlines[cat_clean_rows])
```

Finding consistency 3

```
In [ ]: # Find the cleanliness category in airlines not in categories
cat_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category
cat_clean_rows = airlines['cleanliness'].isin(cat_clean)

# Print rows with inconsistent category
print(airlines[cat_clean_rows])

# Print rows with consistent categories only
print(airlines[~cat_clean_rows])
```

Inconsistent categories

```
In [ ]: # Print unique values of both columns
print(airlines['dest_region'].unique())
print(airlines['dest_size'].unique())

# Lower dest_region column and then replace "eur" with "europe"
airlines['dest_region'] = airlines['dest_region'].str.lower()
airlines['dest_region'] = airlines['dest_region'].replace({'eur': 'europe'})

# Remove white spaces from `dest_size`
airlines['dest_size'] = airlines['dest_size'].str.strip()

# Verify changes have been effected
print(airlines['dest_region'].unique())
print(airlines['dest_size'].unique())
```

Remapping categories

```
In [ ]: # Create ranges for categories
label_ranges = [0, 60, 180, np.inf]
label_names = ['short', 'medium', 'long']
```

```

# Create wait_type column
airlines['wait_type'] = pd.cut(airlines['wait_min'], bins = label_ranges,
                               labels = label_names)

# Create mappings and replace
mappings = {'Monday': 'weekday', 'Tuesday': 'weekday', 'Wednesday': 'weekday', 'Thursday': 'weekday', 'Friday': 'weekday',
            'Saturday': 'weekend', 'Sunday': 'weekend'}

airlines['day_week'] = airlines['day'].replace(mappings)

```

Removing titles and taking names

```

In [ ]: # Replace "Dr." with empty string ""
airlines['full_name'] = airlines['full_name'].str.replace("Dr.", "")

# Replace "Mr." with empty string ""
airlines['full_name'] = airlines['full_name'].str.replace("Mr.", "")

# Replace "Miss" with empty string ""
airlines['full_name'] = airlines['full_name'].str.replace("Miss", "")

# Replace "Ms." with empty string ""
airlines['full_name'] = airlines['full_name'].str.replace("Ms.", "")

# Assert that full_name has no honorifics
assert airlines['full_name'].str.contains('Ms.|Mr.|Miss|Dr.').any() == False

```

Keeping it descriptive

```

In [ ]: # Store length of each row in survey_response column
resp_length = airlines['survey_response'].str.len()

# Find rows in airlines where resp_length > 40
airlines_survey = airlines[resp_length > 40]

# Assert minimum survey_response length is > 40
assert airlines_survey['survey_response'].str.len().min() > 40

# Print new survey_response column
print(airlines_survey['survey_response'])

```

## Chapter 3 - Advanced data problems

In this chapter, you'll dive into more advanced data cleaning problems, such as ensuring that weights are all written in kilograms instead of pounds. You'll also gain invaluable skills that will help you verify that values have been added correctly and that missing values don't negatively impact your

analyses.

[Link for reference](#)

```
In [ ]: import pandas as pd

# Common path prefix
common_path = "C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCa

# Paths for the variables with double backslashes
airlines = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\airlines_final.csv')
banking = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\banking_dirty.csv')
restaurant = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2.csv')
restaurant_dirty = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2_dirty.csv')
ride_sharing = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\ride_sharing_new.csv')

print(banking.head(2))

    Unnamed: 0    cust_id  birth_date  Age  acct_amount  inv_amount  fund_A \
0            0  870A9281  1962-06-09   58      63523.31      51295  30105.0
1            1  166B05B0  1962-12-16   58      38175.46     15050  4995.0

    fund_B  fund_C  fund_D account_opened last_transaction
0  4138.0  1420.0  15632.0        02-09-18        22-02-19
1  938.0  6696.0  2421.0        28-02-19        31-10-18
```

Uniform currencies

```
In [ ]: # Find values of acct_cur that are equal to 'euro'
acct_eu = banking['acct_cur'] == 'euro'

# Convert acct_amount where it is in euro to dollars
#banking.loc[banking['acct_cur']=='euro', 'acct_amount'] = banking.loc[banking['acct_cur']=='euro', 'acct_amount'] * 1.1
banking.loc[acct_eu, 'acct_amount'] = banking.loc[acct_eu, 'acct_amount'] * 1.1
# Unify acct_cur column by changing 'euro' values to 'dollar'
banking.loc[acct_eu, 'acct_cur'] = 'dollar'

# Assert that only dollar currency remains
assert banking['acct_cur'].unique() == 'dollar'
```

Uniform dates

```
In [ ]: # Print the header of account_opened
print(banking['account_opened'].head())
```

```
In [ ]: # Print the header of account_opened
print(banking['account_opened'].head())
```

```
# Convert account_opened to datetime
banking['account_opened'] = pd.to_datetime(banking['account_opened'],
                                             # Infer datetime format
                                             infer_datetime_format = True,
                                             # Return missing value for error
                                             errors = 'coerce')
```

In [ ]:

```
# Print the header of account_opened
print(banking['account_opened'].head())

# Convert account_opened to datetime
banking['account_opened'] = pd.to_datetime(banking['account_opened'],
                                             # Infer datetime format
                                             infer_datetime_format = True,
                                             # Return missing value for error
                                             errors = 'coerce')

# Get year of account opened
banking['acct_year'] = banking['account_opened'].dt.strftime('%Y')

# Print acct_year
print(banking['acct_year'])
```

How's our data integrity?

In [ ]:

```
# Store fund columns to sum against
fund_columns = ['fund_A', 'fund_B', 'fund_C', 'fund_D']

# Find rows where fund_columns row sum == inv_amount
inv_equ = banking[fund_columns].sum(axis=1) == banking['inv_amount']

# Store consistent and inconsistent data
consistent_inv = banking[inv_equ]
inconsistent_inv = banking[~inv_equ]

# Store consistent and inconsistent data
print("Number of inconsistent investments: ", inconsistent_inv.shape[0])
```

How's our data integrity? 2

In [ ]:

```
# Store today's date and find ages
today = dt.date.today()
ages_manual = today.year - banking['birth_date'].dt.year

# Find rows where age column == ages_manual
age_equ = ages_manual == banking['age']
```

```
# Store consistent and inconsistent data
consistent_ages = banking[age_equ]
inconsistent_ages = banking[~age_equ]

# Store consistent and inconsistent data
print("Number of inconsistent ages: ", inconsistent_ages.shape[0])
```

Missing investors

```
In [ ]: #import
import pandas as pd
import missingno as msno
import matplotlib.pyplot as plt

# Print number of missing values in banking
print(banking.isna().sum())

# Visualize missingness matrix
msno.matrix(banking)
plt.show()

# Isolate missing and non missing values of inv_amount
missing_investors = banking[banking['inv_amount'].isna()]
investors = banking[~banking['inv_amount'].isna()]

# Sort banking by age and visualize
banking_sorted = banking.sort_values('age')
msno.matrix(banking_sorted)
plt.show()
```

Follow the money

```
In [ ]: # Drop missing values of cust_id
banking_fullid = banking.dropna(subset = ['cust_id'])

# Compute estimated acct_amount
acct_imp = banking_fullid['inv_amount']*5

# Impute missing acct_amount with corresponding acct_imp
banking_imputed = banking_fullid.fillna({'acct_amount':acct_imp})

# Print number of missing values
print(banking_imputed.isna().sum())
```

```
In [ ]:
```

# Chapter 4 - Record linkage

Record linkage is a powerful technique used to merge multiple datasets together, used when values have typos or different spellings. In this chapter, you'll learn how to link records by calculating the similarity between strings—you'll then use your new skills to join two restaurant review datasets into one clean master dataset. Other references.

[Link for reference](#)

In [ ]:

```
#import last modification
import pandas as pd
import missingno as msno
import matplotlib.pyplot as plt

# Common path prefix
common_path = "C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCa

# Paths for the variables with double backslashes
airlines = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\airlines_final.csv')
banking = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\banking_dirty.csv')
restaurant = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2.csv')
restaurant_dirty = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\restaurants_L2_dirty.csv')
ride_sharing = pd.read_csv(f'{common_path}\\\\7. Cleaning Data in Python\\\\datasets\\\\ride_sharing_new.csv')

print(restaurant_dirty.head(8))
```

	Unnamed: 0	name	addr	city	\
0	0	kokomo	6333 w. third st.	la	
1	1	feenix	8358 sunset blvd. west	hollywood	
2	2	parkway	510 s. arroyo pkwy .	pasadena	
3	3	r-23	923 e. third st.	los angeles	
4	4	gumbo	6333 w. third st.	la	
5	5	pink's	709 n. la brea ave.	la	
6	6	original	875 s. figueroa st. downtown	la	
7	7	21 clubs	21 w. 52nd st.	new york	

	phone	type
0	2139330773	american
1	2138486677	american
2	8187951001	californian
3	2136877178	japanese
4	2139330358	cajun/creole
5	2139314223	hot dogs
6	2136276879	diners
7	2125827200	american

The cutoff point

In [ ]:

```
#install
#pip install python-Levenshtein thefuzz
#import
#from thefuzz import process

# Import process from thefuzz
from thefuzz import process #this name "the fuzz" was given randomly in the course!!!!!

# Store the unique values of cuisine_type in unique_types
unique_types = restaurant_dirty['type'].unique()

# Calculate similarity of 'asian' to all values of unique_types
print(process.extract('asian', unique_types, limit = len(unique_types)))

# Calculate similarity of 'american' to all values of unique_types
print(process.extract('american', unique_types, limit = len(unique_types)))

# Calculate similarity of 'italian' to all values of unique_types
print(process.extract('italian', unique_types, limit = len(unique_types)))
```

```
[('asian', 100), ('indonesian', 80), ('californian', 68), ('italian', 67), ('russian', 67), ('american', 62), ('japanese', 54), ('mexican/tex-mex', 54), ('american ( new )', 54), ('mexican', 50), ('fast food', 45), ('middle eastern', 43), ('steakhouses', 40), ('pacific new wave', 40), ('pizza', 40), ('diners', 36), ('cajun/creole', 36), ('vietnamese', 36), ('continental', 36), ('sea food', 33), ('chicken', 33), ('chinese', 33), ('hot dogs', 30), ('hamburgers', 30), ('coffee shops', 30), ('noodle shops', 30), ('southern/soul', 30), ('desserts', 30), ('eclectic', 26), ('coffeebar', 26), ('health food', 22), ('french ( new )', 22), ('delis', 20)]
[('american', 100), ('american ( new )', 90), ('mexican', 80), ('mexican/tex-mex', 72), ('asian', 62), ('italian', 53), ('russia n', 53), ('californian', 53), ('middle eastern', 51), ('southern/soul', 47), ('pacific new wave', 45), ('hamburgers', 44), ('indo nesian', 44), ('cajun/creole', 42), ('chicken', 40), ('pizza', 40), ('japanese', 38), ('eclectic', 38), ('delis', 36), ('french ( new )', 34), ('vietnamese', 33), ('diners', 29), ('seafood', 27), ('chinese', 27), ('desserts', 25), ('coffeebar', 24), ('steakho uses', 21), ('health food', 21), ('continental', 21), ('coffee shops', 20), ('noodle shops', 20), ('fast food', 12), ('hot dogs', 0)]
[('italian', 100), ('asian', 67), ('californian', 56), ('continental', 54), ('american', 53), ('indonesian', 47), ('russian', 43), ('mexican', 43), ('japanese', 40), ('mexican/tex-mex', 39), ('american ( new )', 39), ('pacific new wave', 39), ('middle east ern', 38), ('vietnamese', 35), ('delis', 33), ('pizza', 33), ('steakhouses', 33), ('health food', 33), ('diners', 31), ('cajun/cr eole', 30), ('chicken', 29), ('chinese', 29), ('southern/soul', 28), ('eclectic', 27), ('noodle shops', 22), ('french ( new )', 18), ('seafood', 14), ('hot dogs', 13), ('desserts', 13), ('fast food', 12), ('coffeebar', 12), ('hamburgers', 12), ('coffee shop s', 0)]
```

Remapping categories II

In [ ]:

```
# Inspect the unique values of the cuisine_type column
print(restaurant_dirty['type'].unique())
```

```
['american' 'californian' 'japanese' 'cajun/creole' 'hot dogs' 'diners'  
 'delis' 'hamburgers' 'seafood' 'italian' 'coffee shops' 'russian'  
 'steakhouses' 'mexican/tex-mex' 'noodle shops' 'mexican' 'middle eastern'  
 'asian' 'vietnamese' 'health food' 'american ( new )' 'pacific new wave'  
 'indonesian' 'eclectic' 'chicken' 'fast food' 'southern/soul' 'coffeebar'  
 'continental' 'french ( new )' 'desserts' 'chinese' 'pizza']
```

```
In [ ]: # Create a list of matches, comparing 'italian' with the cuisine_type column  
matches = process.extract('italian', restaurant_dirty['type'], limit = len(restaurant_dirty))  
  
# Inspect the first 5 matches  
print(matches[0:5])
```

```
[('italian', 100, 14), ('italian', 100, 21), ('italian', 100, 47), ('italian', 100, 57), ('italian', 100, 73)]
```

```
In [ ]: # Create a list of matches, comparing 'italian' with the cuisine_type column  
matches = process.extract('italian', restaurant_dirty['type'], limit=len(restaurant_dirty))  
  
# Iterate through the list of matches to italian  
for match in matches:  
    # Check whether the similarity score is greater than or equal to 80  
    if match[1]>=80:  
        # Select all rows where the cuisine_type is spelled this way, and set them to the correct cuisine  
        restaurant_dirty.loc[restaurant_dirty['type'] == match[0], 'type'] = 'italian'
```

```
In [ ]: # List of predefined categories  
categories = ['Asian', 'American', 'Italian', 'Mexican', 'French']  
  
# Iterate through categories  
for cuisine in categories:  
    # Create a list of matches, comparing cuisine with the cuisine_type column  
    matches = process.extract(cuisine, restaurant_dirty['type'], limit=len(restaurant_dirty.type))  
  
    # Iterate through the list of matches  
    for match in matches:  
        # Check whether the similarity score is greater than or equal to 80  
        if match[1] >= 80:  
            # If it is, select all rows where the cuisine_type is spelled this way, and set them to the correct cuisine  
            restaurant_dirty.loc[restaurant_dirty['type'] == match[0]] = cuisine  
  
    # Inspect the final result  
    restaurant_dirty['type'].unique()
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_38696\3801345375.py:15: FutureWarning: Setting an item of incompatible dtype is deprecated and will raise in a future error of pandas. Value 'Asian' has dtype incompatible with int64, please explicitly cast to a compatible dtype first.  
    restaurant_dirty.loc[restaurant_dirty['type'] == match[0]] = cuisine
```

```
Out[ ]: array(['Mexican', 'californian', 'japanese', 'cajun/creole', 'hot dogs',
   'diners', 'delis', 'hamburgers', 'seafood', 'Italian',
   'coffee shops', 'russian', 'steakhouses', 'noodle shops',
   'middle eastern', 'Asian', 'vietnamese', 'health food',
   'pacific new wave', 'eclectic', 'chicken', 'fast food',
   'southern/soul', 'coffeebar', 'continental', 'French', 'desserts',
   'chinese', 'pizza'], dtype=object)
```

Pairs of restaurants

```
In [ ]: # first install this package
#pip install recordlinkage

# Import the required library
import recordlinkage

# Create an indexer and object and find possible pairs
indexer = recordlinkage.Index()

# Block pairing on cuisine_type
indexer.block('type')

# Generate pairs
pairs = indexer.index(restaurant_dirty, restaurant)
```

Similar restaurants

```
In [ ]: # Create a comparison object
comp_cl = recordlinkage.Compare()
```

```
In [ ]: # Find exact matches on city, cuisine_types -
comp_cl.exact('city', 'city', label='city')
comp_cl.exact('cuisine_type', 'cuisine_type', label='cuisine_type')

# Find similar matches of rest_name
comp_cl.string('rest_name', 'rest_name', label='name', threshold = 0.8)
```

```
Out[ ]: <Compare>
```

```
In [ ]: # Get potential matches and print
potential_matches = comp_cl.compute(pairs, restaurant_dirty, restaurant)
print(potential_matches)
```

Linking them together!

```
In [ ]: # Isolate potential matches with row sum >= 3
matches = potential_matches[potential_matches.sum(axis=1) >= 3]
```

```
# Get values of second column index of matches
matching_indices = matches.index.get_level_values(1)

# Subset restaurants_new based on non-duplicate values
non_dup = restaurant_dirty[~restaurant_dirty.index.isin(matching_indices)]

# Append non_dup to restaurants
full_restaurants = restaurant.append(non_dup)
print(full_restaurants)
```

# Introduction to Statistics in Python

## Chapter 1 - Summary Statistics

Summary statistics gives you the tools you need to boil down massive datasets to reveal the highlights. In this chapter, you'll explore summary statistics including mean, median, and standard deviation, and learn how to accurately interpret them. You'll also develop your critical thinking skills, allowing you to choose the best summary statistics for your data.

```
In [ ]: # Import Libraries required
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

food_consumption = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Pyt
```

```
In [ ]: # Import numpy with alias np
import numpy as np

# Filter for Belgium
be_consumption = food_consumption[food_consumption['country'] == 'Belgium']

# Filter for USA
usa_consumption = food_consumption[food_consumption['country'] == 'USA']

# Calculate mean and median consumption in Belgium
print(np.mean(be_consumption['consumption']))
print(np.median(be_consumption['consumption']))

# Calculate mean and median consumption in USA
print(np.mean(usa_consumption['consumption']))
print(np.median(usa_consumption['consumption']))
```

```
42.13272727272727
12.59
44.650000000000006
14.58
```

Mean and median

```
In [ ]: # Import numpy with alias np
import numpy as np
```

```
# Filter for Belgium
be_consumption = food_consumption[food_consumption['country'] == 'Belgium']

# Filter for USA
usa_consumption = food_consumption[food_consumption['country'] == 'USA']

# Calculate mean and median consumption in Belgium
print(np.mean(be_consumption['consumption']))
print(np.median(be_consumption['consumption']))

# Calculate mean and median consumption in USA
print(np.mean(usa_consumption['consumption']))
print(np.median(usa_consumption['consumption']))
```

```
42.13272727272727
12.59
44.650000000000006
14.58
```

```
In [ ]: # Subset for Belgium and USA only
be_and_usa = food_consumption[(food_consumption['country'] == 'Belgium') | (food_consumption['country'] == 'USA')]

# Group by country, select consumption column, and compute mean and median
print(be_and_usa.groupby('country')['consumption'].agg(['mean', 'median']))
```

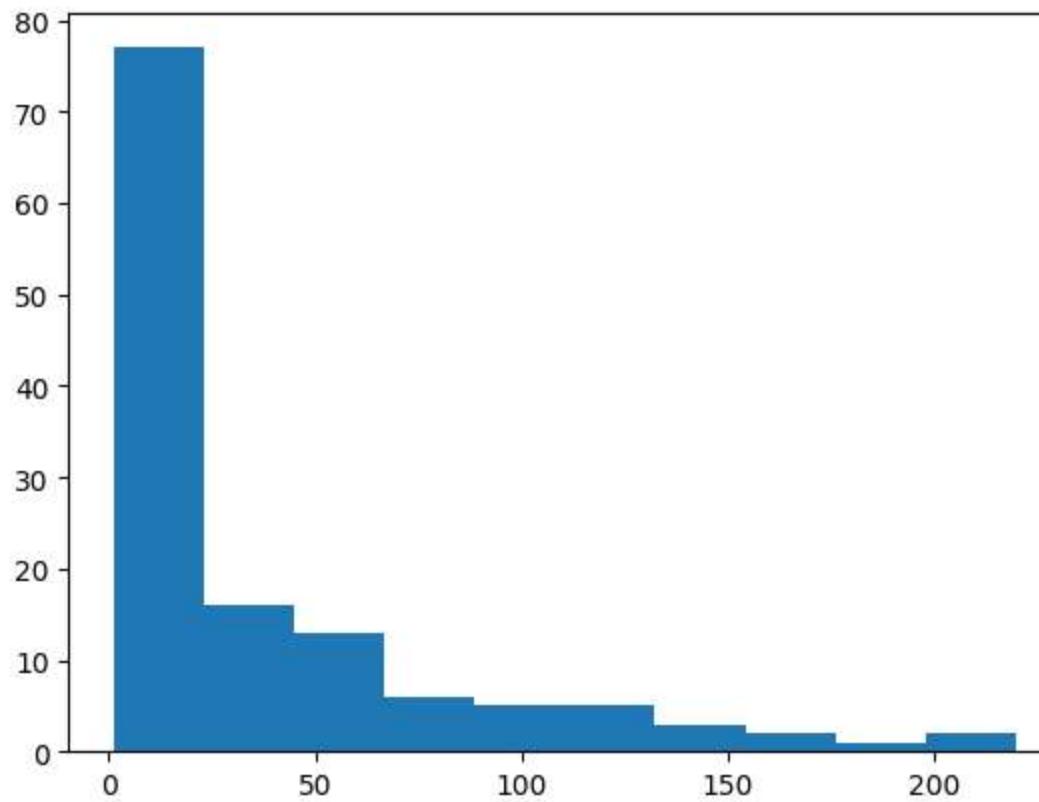
country	mean	median
Belgium	42.132727	12.59
USA	44.650000	14.58

Mean vs. median

```
In [ ]: # Import matplotlib.pyplot with alias plt
import matplotlib.pyplot as plt

# Subset for food_category equals rice
rice_consumption = food_consumption[food_consumption['food_category'] == 'rice']

# Histogram of co2_emission for rice and show plot
plt.hist(rice_consumption['co2_emission'])
plt.show()
```



```
In [ ]: # Subset for food_category equals rice
rice_consumption = food_consumption[food_consumption['food_category'] == 'rice']

# Calculate mean and median of co2_emission with .agg()
print(rice_consumption['co2_emission'].agg(['mean', 'median']))
```

```
mean    37.591615
median   15.200000
Name: co2_emission, dtype: float64
```

Quartiles, quantiles, and quintiles

```
In [ ]: # Calculate the quartiles of co2_emission
print(np.quantile(food_consumption['co2_emission'], [0, 0.25, 0.5, 0.75, 1]))
```

```
[ 0.      5.21    16.53   62.5975 1712. ]
```

```
In [ ]: # Calculate the quintiles of co2_emission
print(np.quantile(food_consumption['co2_emission'], [0, 0.2, 0.4, 0.6, 0.8, 1]))
```

```
[ 0.      3.54    11.026   25.59   99.978 1712. ]
```

```
In [ ]: # Calculate the deciles of co2_emission
print(np.quantile(food_consumption['co2_emission'], np.linspace(0, 1, 10)))
```

```
[0.00000000e+00 9.05555556e-01 4.1911111e+00 8.05333333e+00  
1.32000000e+01 2.10944444e+01 3.58666667e+01 7.90622222e+01  
1.86115556e+02 1.71200000e+03]
```

```
In [ ]: # Calculate the eleven quantiles of co2_emission  
print(np.quantile(food_consumption['co2_emission'], np.linspace(0, 1, 11)))
```

```
[0.00000e+00 6.68000e-01 3.54000e+00 7.04000e+00 1.10260e+01 1.65300e+01  
2.55900e+01 4.42710e+01 9.99780e+01 2.03629e+02 1.71200e+03]
```

Variance and standard deviation

```
In [ ]: # Print variance and sd of co2_emission for each food_category  
print(food_consumption.groupby('food_category')['co2_emission'].agg([np.var, np.std]))  
  
# Import matplotlib.pyplot with alias plt  
import matplotlib.pyplot as plt  
  
# Create histogram of co2_emission for food_category 'beef'  
food_consumption[food_consumption['food_category'] == 'beef']['co2_emission'].hist()  
# Show plot  
plt.show()  
  
# Create histogram of co2_emission for food_category 'eggs'  
food_consumption[food_consumption['food_category'] == 'eggs']['co2_emission'].hist()  
# Show plot  
plt.show()
```

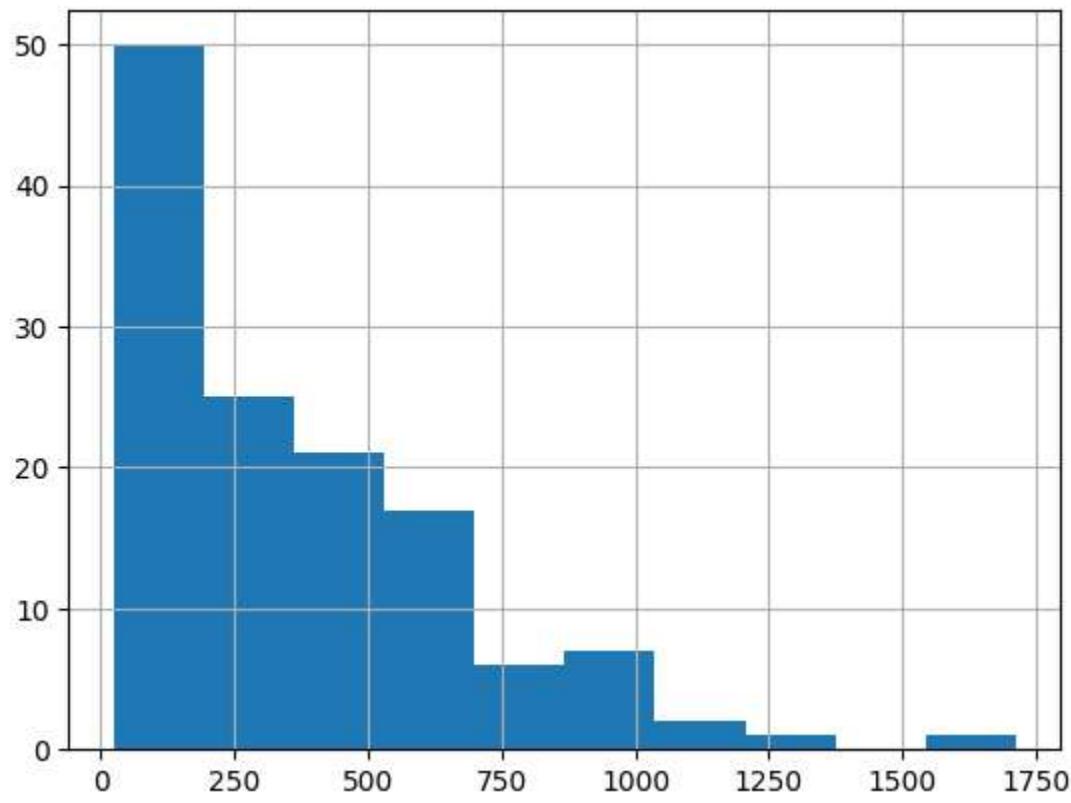
food_category	var	std
beef	88748.408132	297.906710
dairy	17671.891985	132.935669
eggs	21.371819	4.622966
fish	921.637349	30.358481
lamb_goat	16475.518363	128.356996
nuts	35.639652	5.969895
pork	3094.963537	55.632396
poultry	245.026801	15.653332
rice	2281.376243	47.763754
soybeans	0.879882	0.938020
wheat	71.023937	8.427570

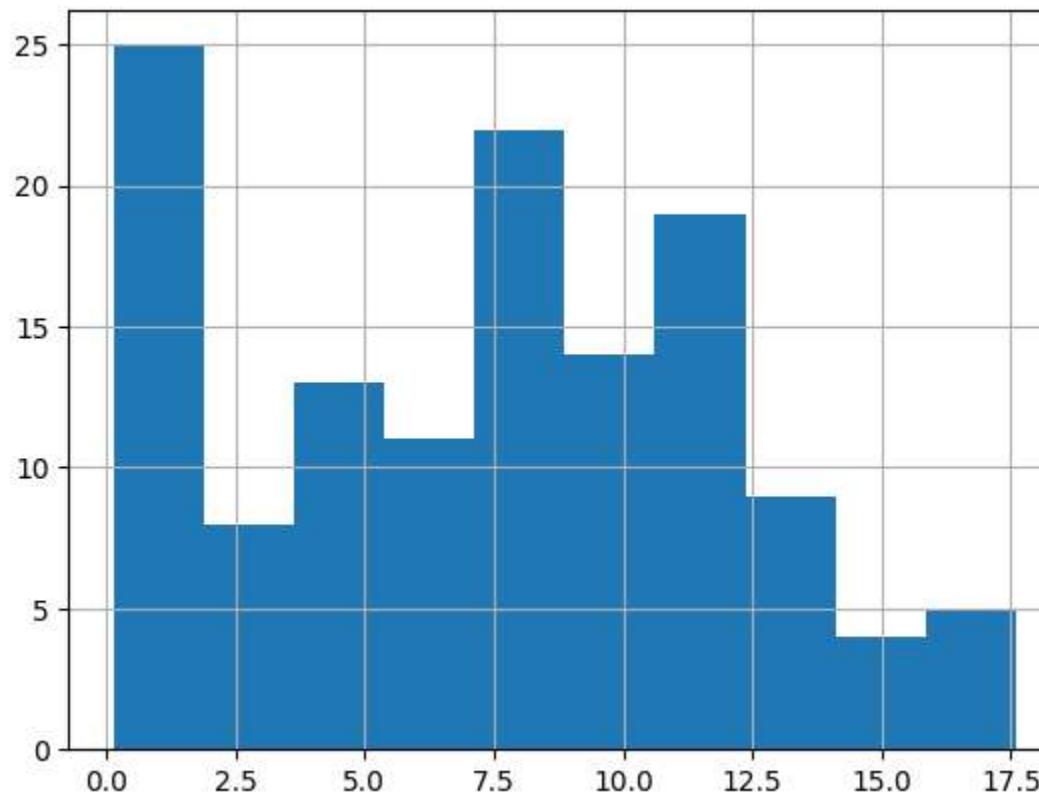
```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_24784\1222949537.py:2: FutureWarning: The provided callable <function var at 0x00000177C1D3B100> is currently using SeriesGroupBy.var. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "var" instead.
```

```
    print(food_consumption.groupby('food_category')['co2_emission'].agg([np.var, np.std]))
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_24784\1222949537.py:2: FutureWarning: The provided callable <function std at 0x00000177C1D3AFC0> is currently using SeriesGroupBy.std. In a future version of pandas, the provided callable will be used directly. To keep current behavior pass the string "std" instead.
```

```
    print(food_consumption.groupby('food_category')['co2_emission'].agg([np.var, np.std]))
```





Finding outliers using IQR

```
In [ ]: # Calculate total co2_emission per country: emissions_by_country
emissions_by_country = food_consumption.groupby('country')['co2_emission'].sum()

# Compute the first and third quantiles and IQR of emissions_by_country
q1 = np.quantile(emissions_by_country, 0.25)
q3 = np.quantile(emissions_by_country, 0.75)
iqr = q3 - q1

# Calculate the lower and upper cutoffs for outliers
lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr

# Subset emissions_by_country to find outliers
outliers = emissions_by_country[(emissions_by_country < lower) | (emissions_by_country > upper)]
print(outliers)

country
Argentina    2172.4
Name: co2_emission, dtype: float64
```

# Chapter 2 - Random Numbers and Probability

In this chapter, you'll learn how to generate random samples and measure chance using probability. You'll work with real-world sales data to calculate the probability of a salesperson being successful. Finally, you'll use the binomial distribution to model events with binary outcomes.

Calculating probabilities

```
In [ ]: # Import Libraries required
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

amir_deals = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Co
```

```
In [ ]: # Count the deals for each product
counts = amir_deals['product'].value_counts()

# Calculate probability of picking a deal with each product
probs = counts / amir_deals['product'].value_counts().sum()
print(probs)
```

```
product
Product B    0.348315
Product D    0.224719
Product A    0.129213
Product C    0.084270
Product F    0.061798
Product H    0.044944
Product I    0.039326
Product E    0.028090
Product N    0.016854
Product G    0.011236
Product J    0.011236
Name: count, dtype: float64
```

Sampling deals

```
In [ ]: # Set random seed
np.random.seed(24)

# Sample 5 deals without replacement
sample_without_replacement = amir_deals.sample(5)
print(sample_without_replacement)
```

```
Unnamed: 0    product  client status   amount  num_users
127         128  Product B  Current    Won  2070.25      7
148         149  Product D  Current    Won  3485.48     52
77          78  Product B  Current    Won  6252.30     27
104        105  Product D  Current    Won  4110.98     39
166        167  Product C     New    Lost  3779.86     11
```

```
In [ ]: # Set random seed
np.random.seed(24)
```

```
# Sample 5 deals with replacement
sample_with_replacement = amir_deals.sample(5, replace=True)
print(sample_with_replacement)
```

```
Unnamed: 0    product  client status   amount  num_users
162         163  Product D  Current    Won  6755.66     59
131         132  Product B  Current    Won  6872.29     25
87          88  Product C  Current    Won  3579.63      3
145        146  Product A  Current    Won  4682.94     63
145        146  Product A  Current    Won  4682.94     63
```

Creating a probability distribution

```
# this part of the code was created randomly by me!
import pandas as pd

# Creating the DataFrame
data = {'group_id': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
        'group_size': [2, 4, 6, 2, 2, 2, 3, 2, 4, 2]}

restaurant_groups = pd.DataFrame(data)

# Displaying the table
restaurant_groups
```

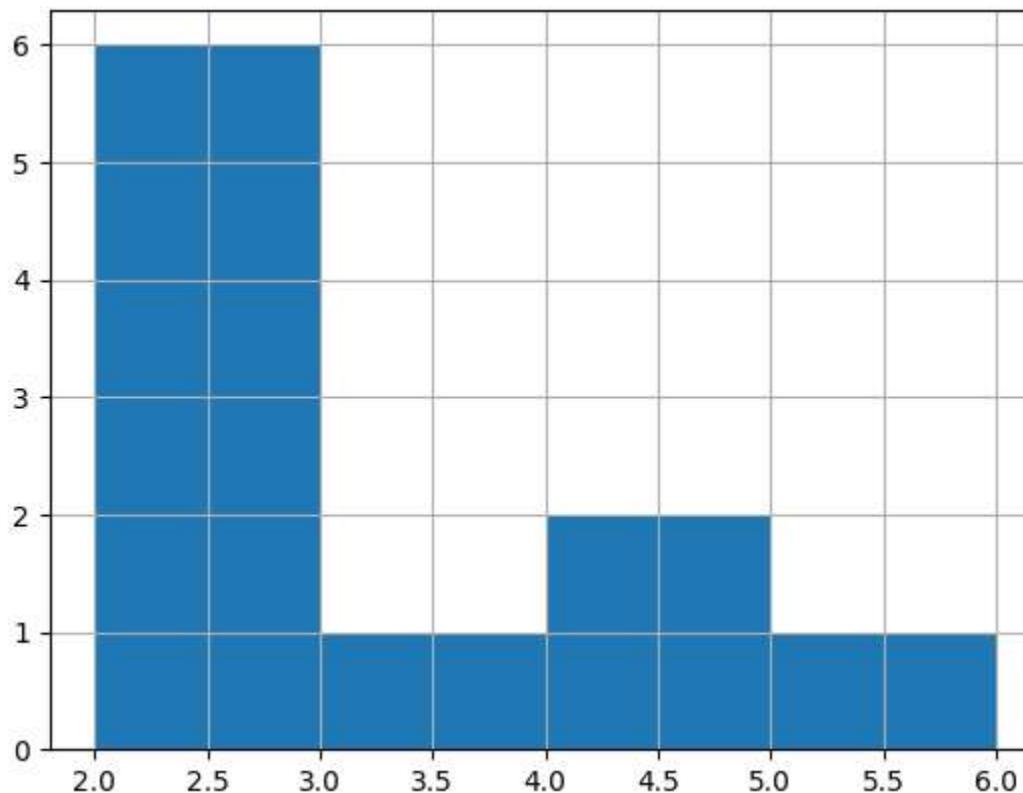
Out[ ]:

	group_id	group_size
0	A	2
1	B	4
2	C	6
3	D	2
4	E	2
5	F	2
6	G	3
7	H	2
8	I	4
9	J	2

In [ ]:

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

# Create a histogram of restaurant_groups and show plot
restaurant_groups['group_size'].hist(bins=[2, 3, 4, 5, 6])
plt.show()
```



```
In [ ]: # Create probability distribution
size_dist = restaurant_groups['group_size'].value_counts() / restaurant_groups.shape[0]
# Reset index and rename columns
size_dist = size_dist.reset_index()
size_dist.columns = ['group_size', 'prob']
```

```
In [ ]: # Expected value
expected_value = np.sum(size_dist['group_size'] * size_dist['prob'])
```

```
In [ ]: # Subset groups of size 4 or more
groups_4_or_more = size_dist[size_dist['group_size'] >= 4]
```

```
In [ ]: # Sum the probabilities of groups_4_or_more
prob_4_or_more = np.sum(groups_4_or_more['prob'])
print(prob_4_or_more)
```

0.3000000000000004

Data back-ups

Calculate probability of waiting less than 5 mins

```
In [ ]: # Min and max wait times for back-up that happens every 30 min  
min_time = 0  
max_time = 30  
  
# Import uniform from scipy.stats  
from scipy.stats import uniform  
  
# Calculate probability of waiting less than 5 mins  
prob_less_than_5 = uniform.cdf(5, min_time, max_time)  
print(prob_less_than_5)
```

0.1666666666666666

Calculate probability of waiting more than 5 mins

```
In [ ]: # Min and max wait times for back-up that happens every 30 min  
min_time = 0  
max_time = 30  
  
# Import uniform from scipy.stats  
from scipy.stats import uniform  
  
# Calculate probability of waiting more than 5 mins  
prob_greater_than_5 = 1 - uniform.cdf(5, min_time, max_time)  
print(prob_greater_than_5)
```

0.8333333333333334

Calculate probability of waiting 10-20 mins

```
In [ ]: # Min and max wait times for back-up that happens every 30 min  
min_time = 0  
max_time = 30  
  
# Import uniform from scipy.stats  
from scipy.stats import uniform  
  
# Calculate probability of waiting 10-20 mins  
prob_between_10_and_20 = uniform.cdf(20, min_time, max_time) - uniform.cdf(10, min_time, max_time)  
print(prob_between_10_and_20)
```

0.3333333333333333

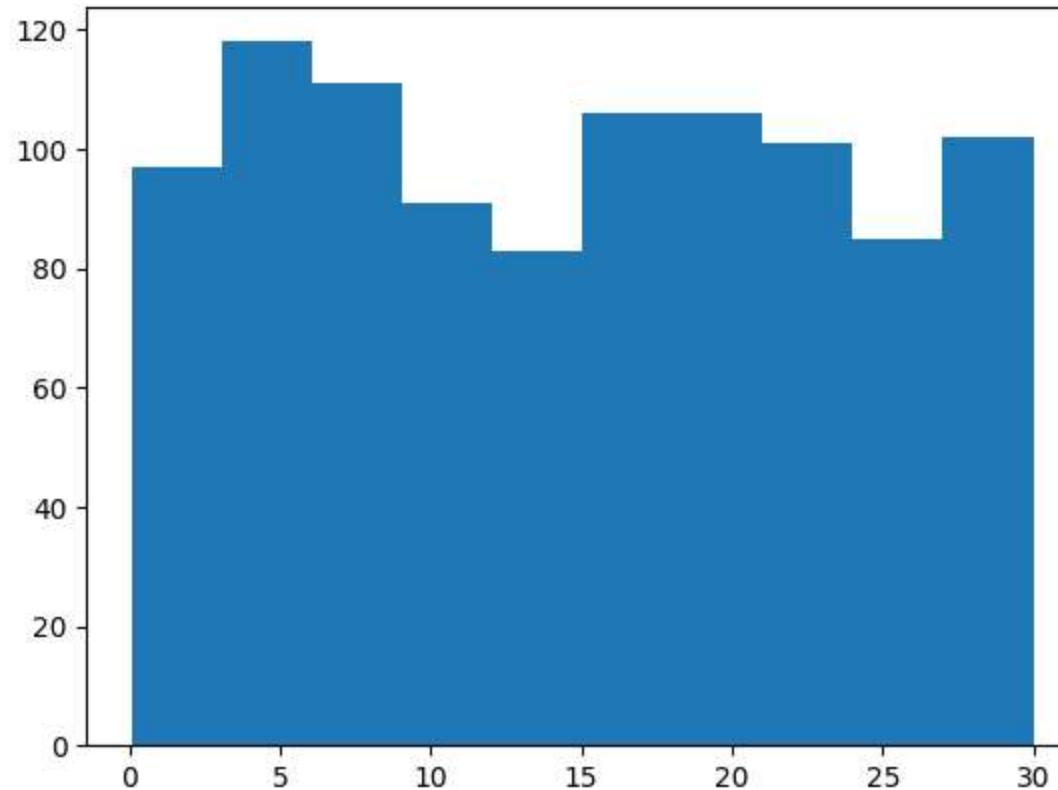
Simulating wait times

```
In [ ]: # Set random seed to 334  
np.random.seed(334)  
  
# Import uniform
```

```
from scipy.stats import uniform

# Generate 1000 wait times between 0 and 30 mins
wait_times = uniform.rvs(0, 30, size=1000)

# Create a histogram of simulated times and show plot
plt.hist(wait_times)
plt.show()
```



Simulating sales deals

```
In [ ]: # Import binom from scipy.stats
from scipy.stats import binom

# Set random seed to 10
np.random.seed(10)

# Simulate a single deal
print(binom.rvs(1, 0.3, size=1))

# Simulate 1 week of 3 deals
print(binom.rvs(3, 0.3, size=1))
```

```
# Simulate 52 weeks of 3 deals
deals = binom.rvs(3, 0.3, size=52)

# Print mean deals won per week
print(np.mean(deals))
```

```
[1]
[0]
0.8461538461538461
```

Calculating binomial probabilities

In [ ]:

```
# Probability of closing 3 out of 3 deals
prob_3 = binom.pmf(3, 3, 0.3)

print(prob_3)
# Probability of closing <= 1 deal out of 3 deals
prob_less_than_or_equal_1 = binom.cdf(1, 3, 0.3)

print(prob_less_than_or_equal_1)

# Probability of closing > 1 deal out of 3 deals
prob_greater_than_1 = 1 - binom.cdf(1, 3, 0.3)

print(prob_greater_than_1)
```

```
0.027
0.784
0.2159999999999997
```

How many sales will be won?

Calculate the expected number of sales out of the 3 he works on that Amir will win each week if he maintains his 30% win rate.

Calculate the expected number of sales out of the 3 he works on that he'll win if his win rate drops to 25%.

Calculate the expected number of sales out of the 3 he works on that he'll win if his win rate rises to 35%.

In [ ]:

```
# Expected number won with 30% win rate
won_30pct = 3 * 0.3
print(won_30pct)

# Expected number won with 25% win rate
won_25pct = 3 * 0.25
print(won_25pct)

# Expected number won with 35% win rate
```

```
won_35pct = 3 * .35  
print(won_35pct)
```

```
0.8999999999999999  
0.75  
1.0499999999999998
```

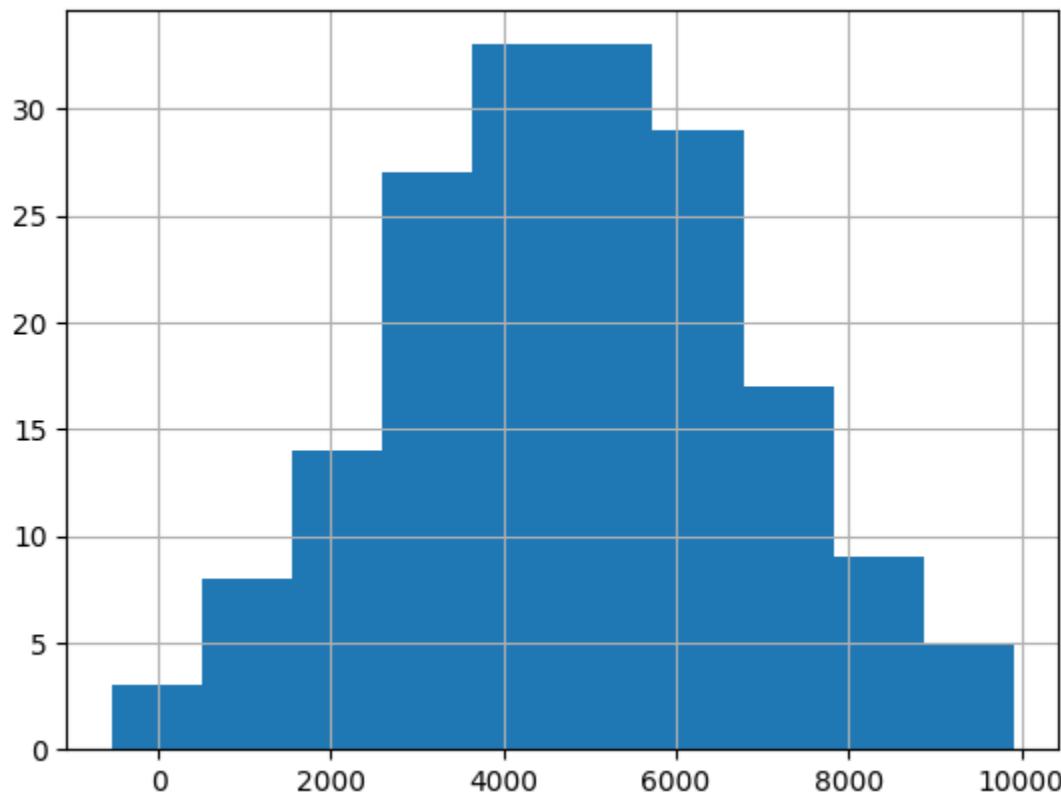
## Chapter 3 - More Distributions and the Central Limit Theorem

It's time to explore one of the most important probability distributions in statistics, normal distribution. You'll create histograms to plot normal distributions and gain an understanding of the central limit theorem, before expanding your knowledge of statistical functions by adding the Poisson, exponential, and t-distributions to your repertoire.

Distribution of Amir's sales

```
In [ ]: # Import Libraries requiered  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
  
amir_deals = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Co
```

```
In [ ]: # Histogram of amount with 10 bins and show plot  
amir_deals['amount'].hist(bins=10)  
plt.show()
```



Probabilities from the normal distribution

```
In [ ]: # Importing Libraries
from scipy.stats import norm
import pandas as pd

# Probability of deal < 7500
prob_less_7500 = norm.cdf(7500, 5000, 2000)

print(prob_less_7500)

# Probability of deal > 1000
prob_over_1000 = 1 - norm.cdf(1000, 5000, 2000)

print(prob_over_1000)

# Probability of deal between 3000 and 7000
prob_3000_to_7000 = norm.cdf(7000, 5000, 2000) - norm.cdf(3000, 5000, 2000)

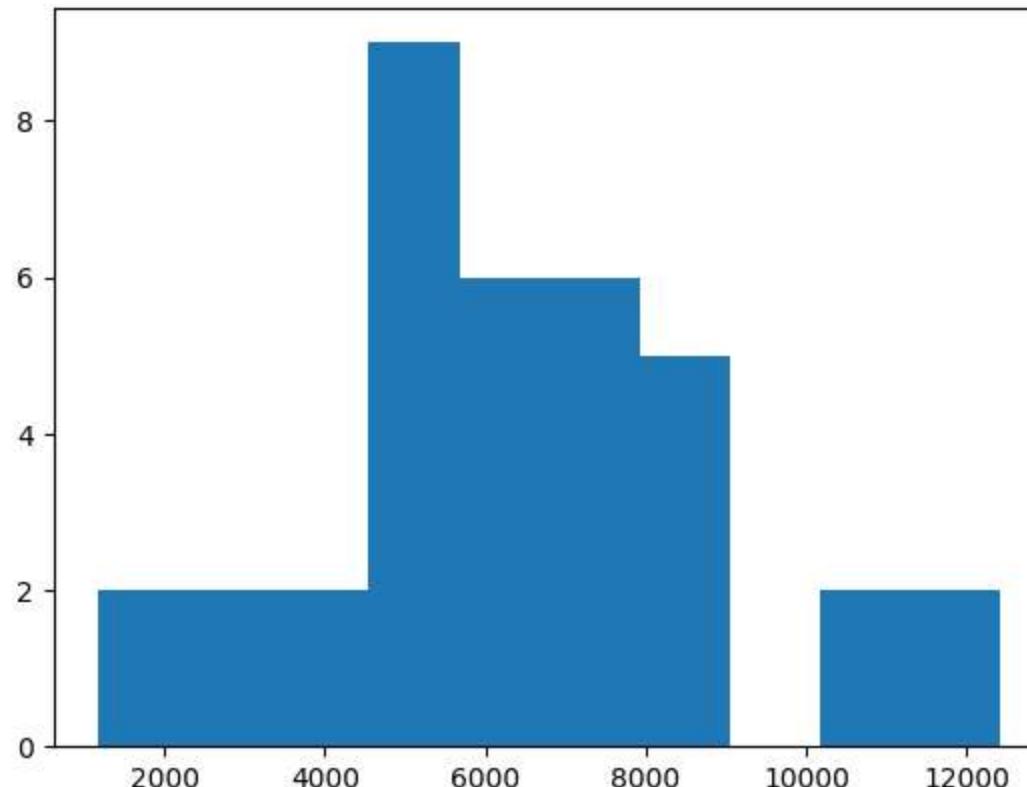
print(prob_3000_to_7000)

# Calculate amount that 25% of deals will be less than
```

```
pct_25 = norm.ppf(0.25, 5000, 2000)  
print(pct_25)  
  
0.8943502263331446  
0.9772498680518208  
0.6826894921370859  
3651.0204996078364
```

Simulating sales under new market conditions

```
In [ ]:  
# Calculate new average amount  
new_mean = 5000 * 1.2  
  
# Calculate new standard deviation  
new_sd = 2000 * 1.3  
  
# Simulate 36 new sales  
new_sales = norm.rvs(6000, 2600, 36)  
  
# Create histogram and show  
plt.hist(new_sales)  
plt.show()
```



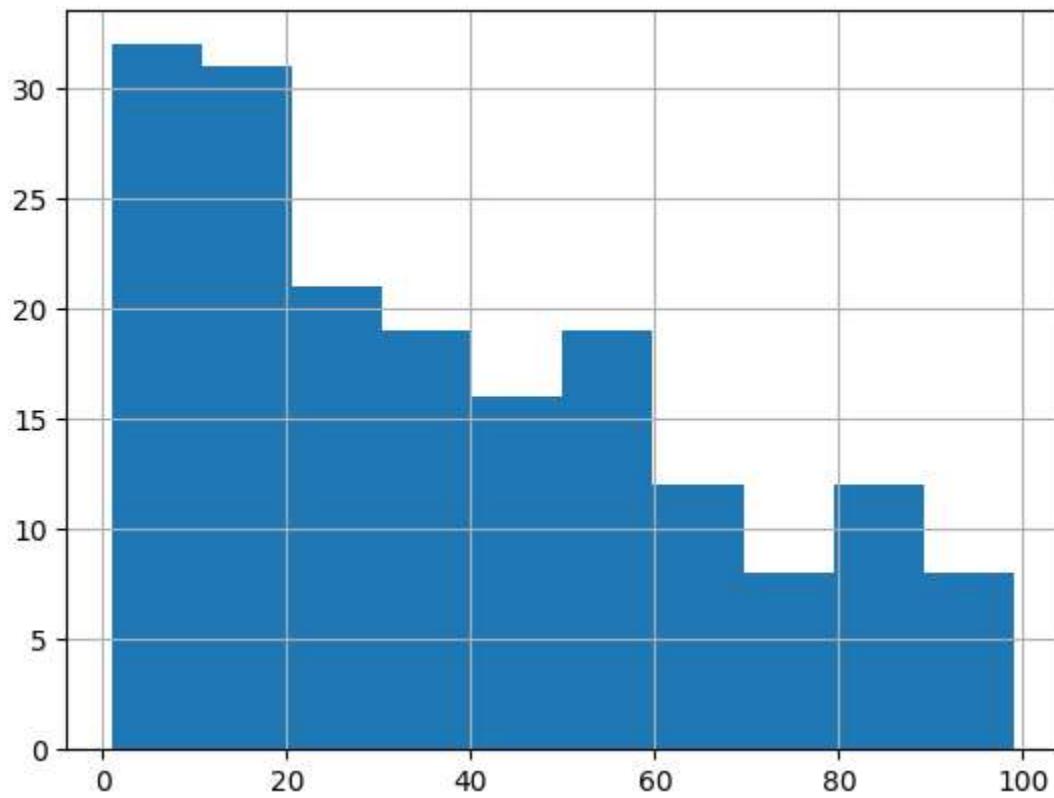
## The CLT in action

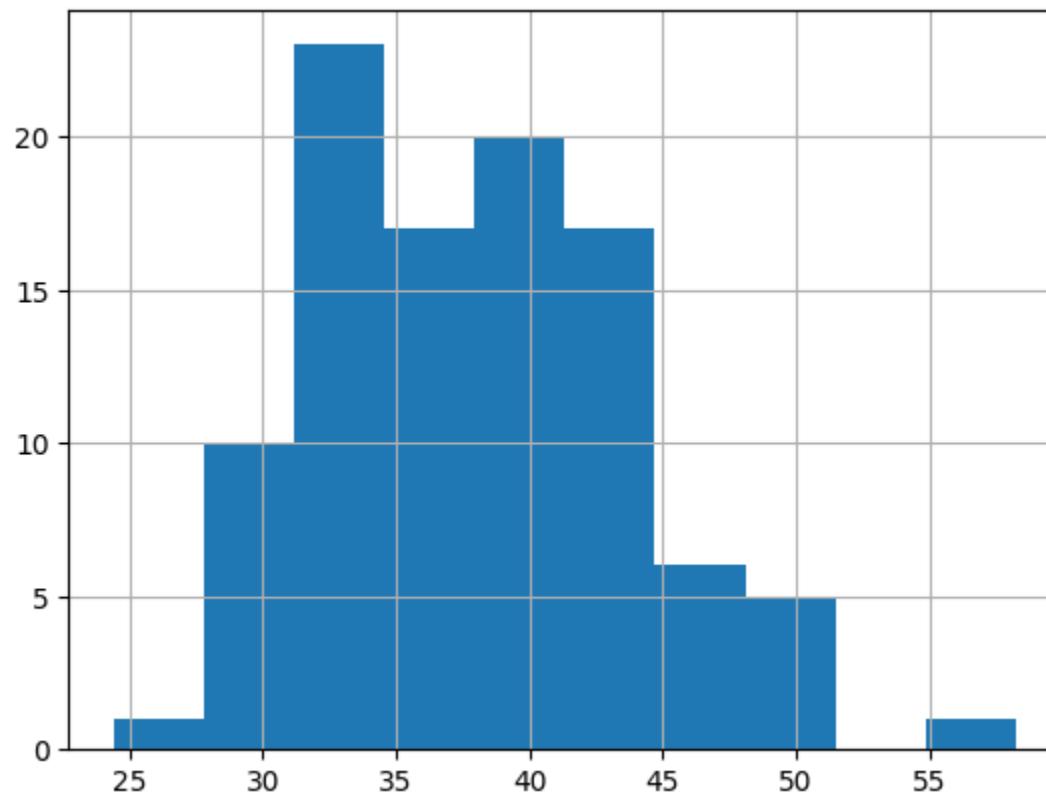
```
In [ ]: # Create a histogram of num_users and show
amir_deals['num_users'].hist()
plt.show()

# Set seed to 104
np.random.seed(104)

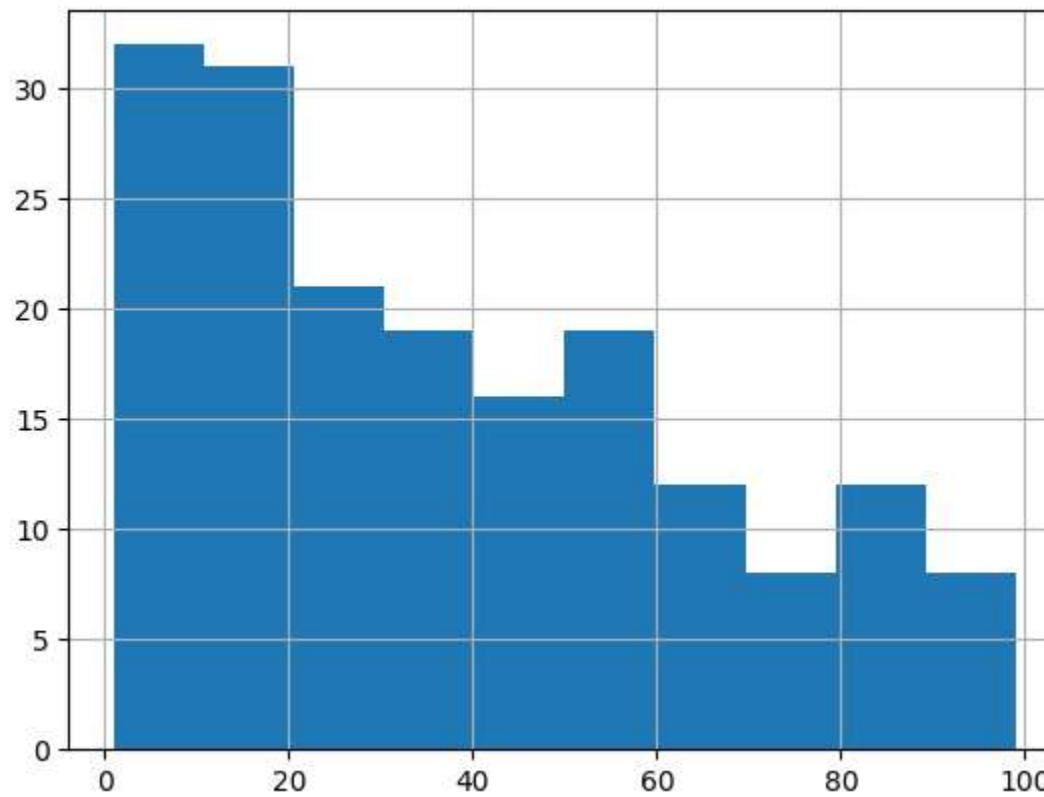
sample_means = []
# Loop 100 times
for i in range(100):
    # Take sample of 20 num_users
    samp_20 = amir_deals['num_users'].sample(20, replace=True)
    # Calculate mean of samp_20
    samp_20_mean = np.mean(samp_20)
    # Append samp_20_mean to sample_means
    sample_means.append(samp_20_mean)

# Convert to Series and plot histogram
sample_means_series = pd.Series(sample_means)
sample_means_series.hist()
# Show plot
plt.show()
```





```
In [ ]: # Create a histogram of num_users and show
amir_deals['num_users'].hist()
plt.show()
```



```
In [ ]: # Set seed to 104
np.random.seed(104)

# Sample 20 num_users with replacement from amir_deals
samp_20 = amir_deals['num_users'].sample(20, replace=True)

# Take mean of samp_20
samp_20_mean = np.mean(samp_20)
print(samp_20_mean)
```

32.0

```
In [ ]: # Set seed to 104
np.random.seed(104)

# Sample 20 num_users with replacement from amir_deals and take mean
samp_20 = amir_deals['num_users'].sample(20, replace=True)
np.mean(samp_20)

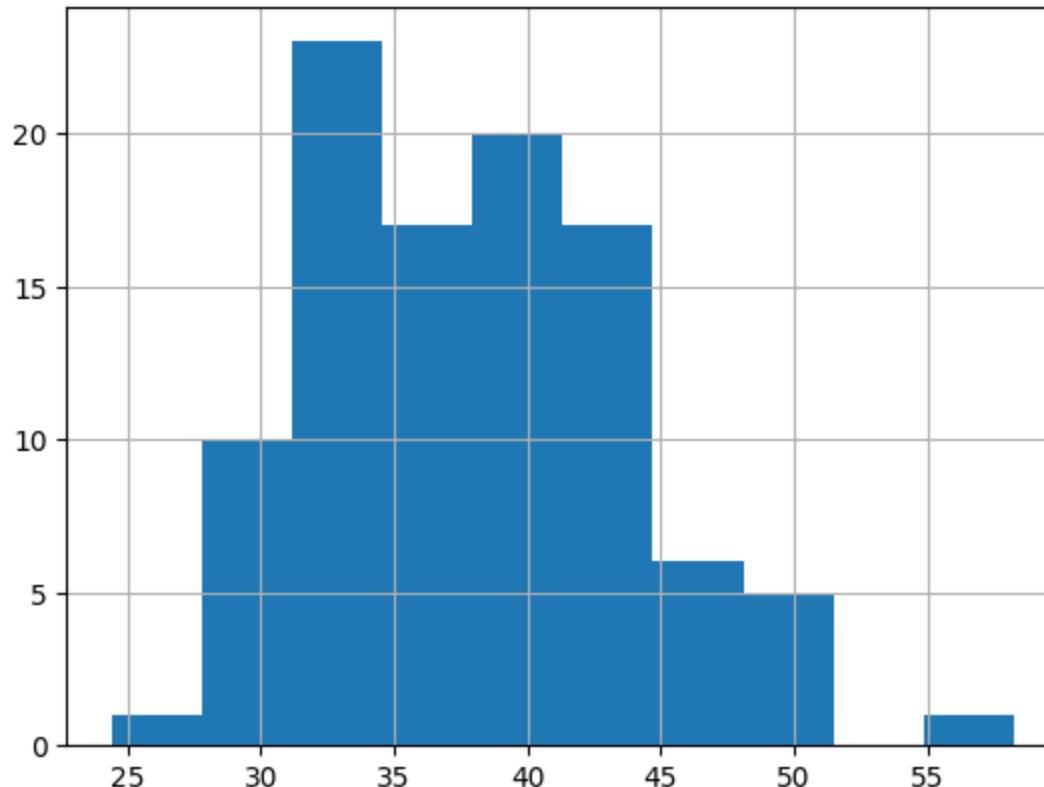
sample_means = []
# Loop 100 times
for i in range(100):
    # Take sample of 20 num_users
```

```
samp_20 = amir_deals['num_users'].sample(20, replace=True)
# Calculate mean of samp_20
samp_20_mean = np.mean(samp_20)
# Append samp_20_mean to sample_means
sample_means.append(samp_20_mean)

print(sample_means)
```

```
[31.35, 45.05, 33.55, 38.15, 50.85, 31.85, 34.65, 36.25, 38.9, 44.05, 35.45, 37.6, 37.95, 28.85, 33.3, 31.65, 45.5, 43.2, 24.4, 41.05, 37.2, 39.3, 29.45, 33.55, 45.3, 45.1, 30.95, 36.25, 37.65, 42.55, 34.55, 41.1, 36.9, 42.45, 38.45, 45.9, 42.7, 38.4, 32.55, 30.25, 38.0, 38.75, 49.3, 39.55, 49.05, 42.05, 41.0, 40.6, 58.25, 34.55, 51.2, 34.15, 36.95, 42.45, 41.85, 33.2, 36.15, 37.55, 34.2, 29.75, 42.35, 43.75, 29.0, 32.05, 31.65, 44.6, 30.85, 29.6, 37.7, 33.1, 36.35, 40.65, 45.7, 33.8, 40.1, 39.9, 33.5, 32.65, 32.85, 42.85, 35.4, 31.7, 32.0, 33.85, 36.6, 44.35, 39.9, 37.0, 37.3, 42.5, 38.35, 42.8, 44.55, 30.3, 50.45, 42.35, 40.65, 29.85, 39.3, 33.1]
```

```
In [ ]: # Convert to Series and plot histogram
sample_means_series = pd.Series(sample_means)
sample_means_series.hist()
# Show plot
plt.show()
```



The mean of means

In [ ]:

```
# Set seed to 321
np.random.seed(321)

sample_means = []
# Loop 30 times to take 30 means
for i in range(30):
    # Take sample of size 20 from num_users col of all_deals with replacement
    cur_sample = all_deals['num_users'].sample(20, replace=True)
    # Take mean of cur_sample
    cur_mean = np.mean(cur_sample)
    # Append cur_mean to sample_means
    sample_means.append(cur_mean)

# Print mean of sample_means
print(np.mean(sample_means))

# Print mean of num_users in amir_deals
print(np.mean(amir_deals['num_users']))
```

Tracking lead responses

In [ ]:

```
# Import poisson from scipy.stats
from scipy.stats import poisson

# Probability of 5 responses
prob_5 = poisson.pmf(5, 4)

print(prob_5)

# Probability of 5 responses
prob_coworker = poisson.pmf(5, 5.5)

print(prob_coworker)

# Probability of 2 or fewer responses
prob_2_or_less = poisson.cdf(2, 4)

print(prob_2_or_less)

# Probability of > 10 responses
prob_over_10 = 1 - poisson.cdf(10, 4)

print(prob_over_10)
```

```
0.1562934518505317  
0.17140068409793663  
0.23810330555354436  
0.0028397661205137315
```

Modeling time between leads

```
In [ ]: # Import expon from scipy.stats  
from scipy.stats import expon
```

```
# Print probability response takes < 1 hour  
print(expon.cdf(1, scale=2.5))
```

```
# Print probability response takes > 4 hours  
print(1 - expon.cdf(4, scale=2.5))
```

```
# Print probability response takes 3-4 hours  
print(expon.cdf(4, scale=2.5) - expon.cdf(3, scale=2.5))
```

```
0.3296799539643607  
0.20189651799465536  
0.09929769391754684
```

```
In [ ]:
```

## Chapter 4 - Correlation and Experimental Design

In this chapter, you'll learn how to quantify the strength of a linear relationship between two variables, and explore how confounding variables can affect the relationship between two other variables. You'll also see how a study's design can influence its results, change how the data should be analyzed, and potentially affect the reliability of your conclusions.

```
In [ ]: # Import Libraries required  
import seaborn as sns  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

```
world_happiness = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Pyth  
print(world_happiness.head(4))
```

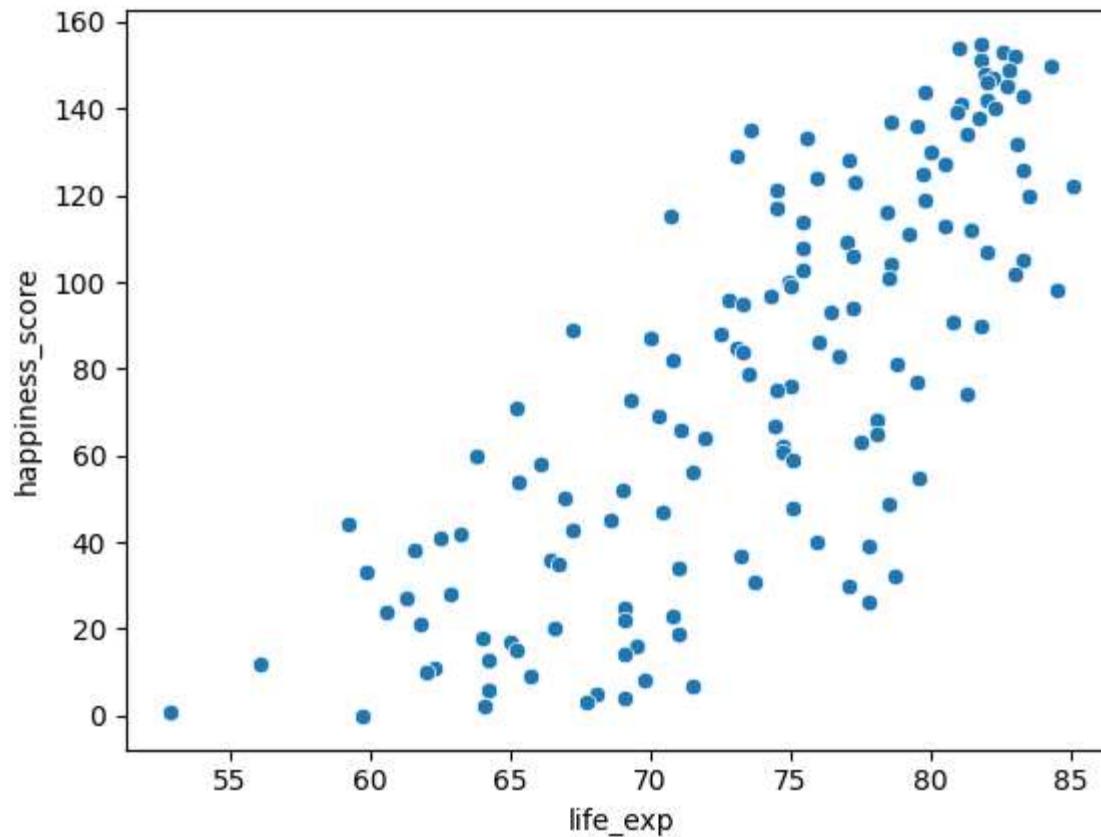
```
Unnamed: 0 country social_support freedom corruption generosity \
0 1 Finland 2.0 5.0 4.0 47.0
1 2 Denmark 4.0 6.0 3.0 22.0
2 3 Norway 3.0 3.0 8.0 11.0
3 4 Iceland 1.0 7.0 45.0 3.0

gdp_per_cap life_exp happiness_score
0 42400 81.8 155
1 48300 81.0 154
2 66300 82.6 153
3 47900 83.0 152
```

Relationships between variables

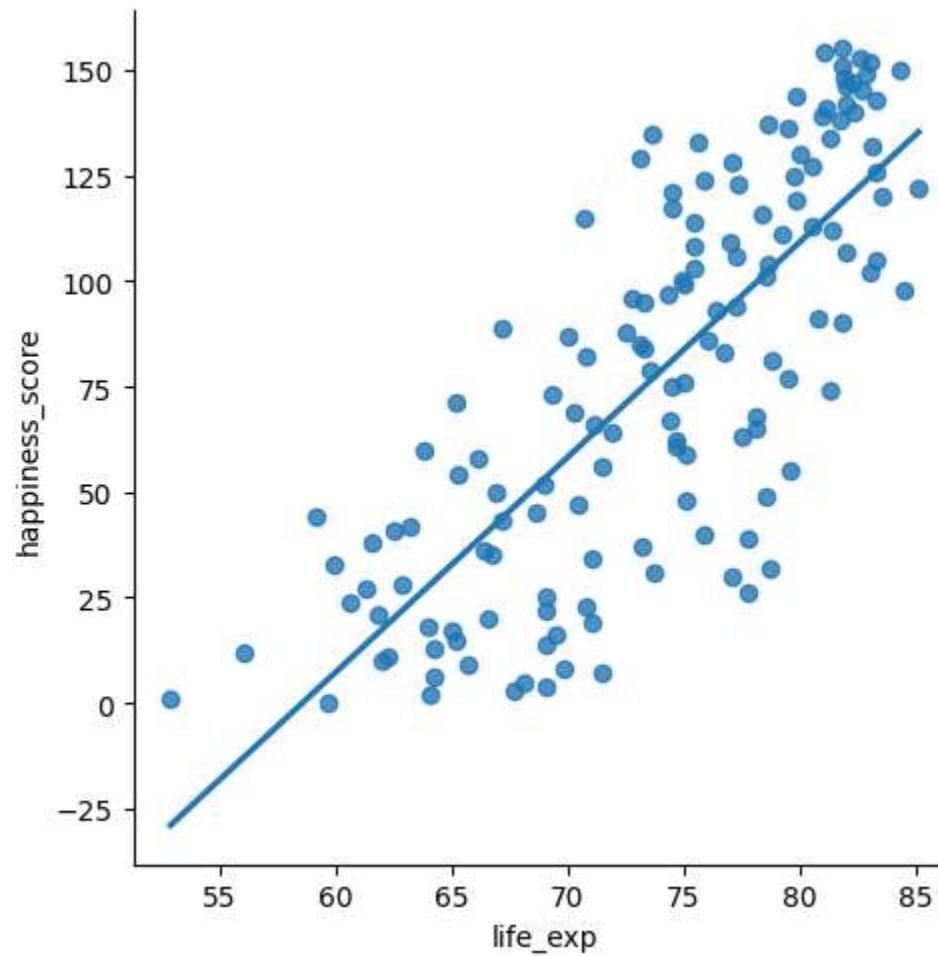
```
In [ ]: # Create a scatterplot of happiness_score vs. Life_exp and show
sns.scatterplot(y='happiness_score', x='life_exp', data=world_happiness)

# Show plot
plt.show()
```



```
In [ ]: # Create scatterplot of happiness_score vs Life_exp with trendline
sns.lmplot(x='life_exp', y='happiness_score', data=world_happiness, ci=None)
```

```
# Show plot  
plt.show()
```



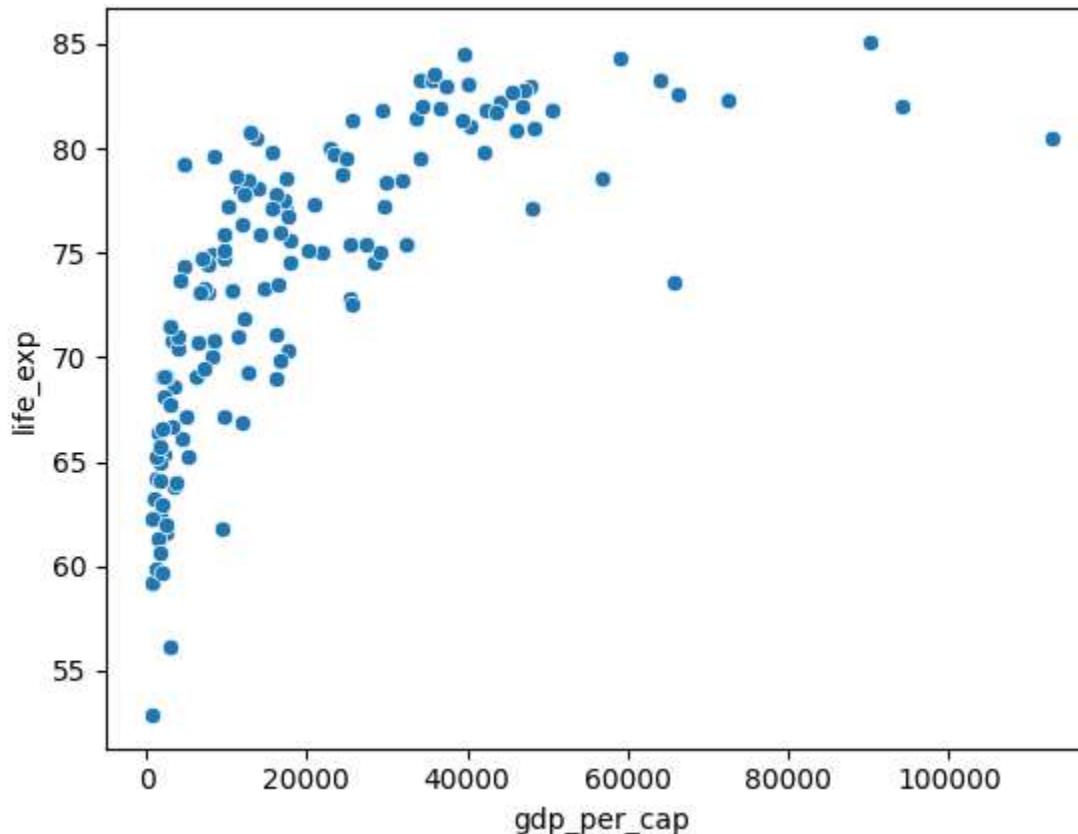
```
In [ ]: # Correlation between life_exp and happiness_score  
cor = world_happiness['life_exp'].corr(world_happiness['happiness_score'])  
  
print(cor)
```

0.7802249053272062

What can't correlation measure?

```
In [ ]: # Scatterplot of gdp_per_cap and life_exp  
sns.scatterplot(x='gdp_per_cap', y='life_exp', data=world_happiness)  
  
# Show plot  
plt.show()
```

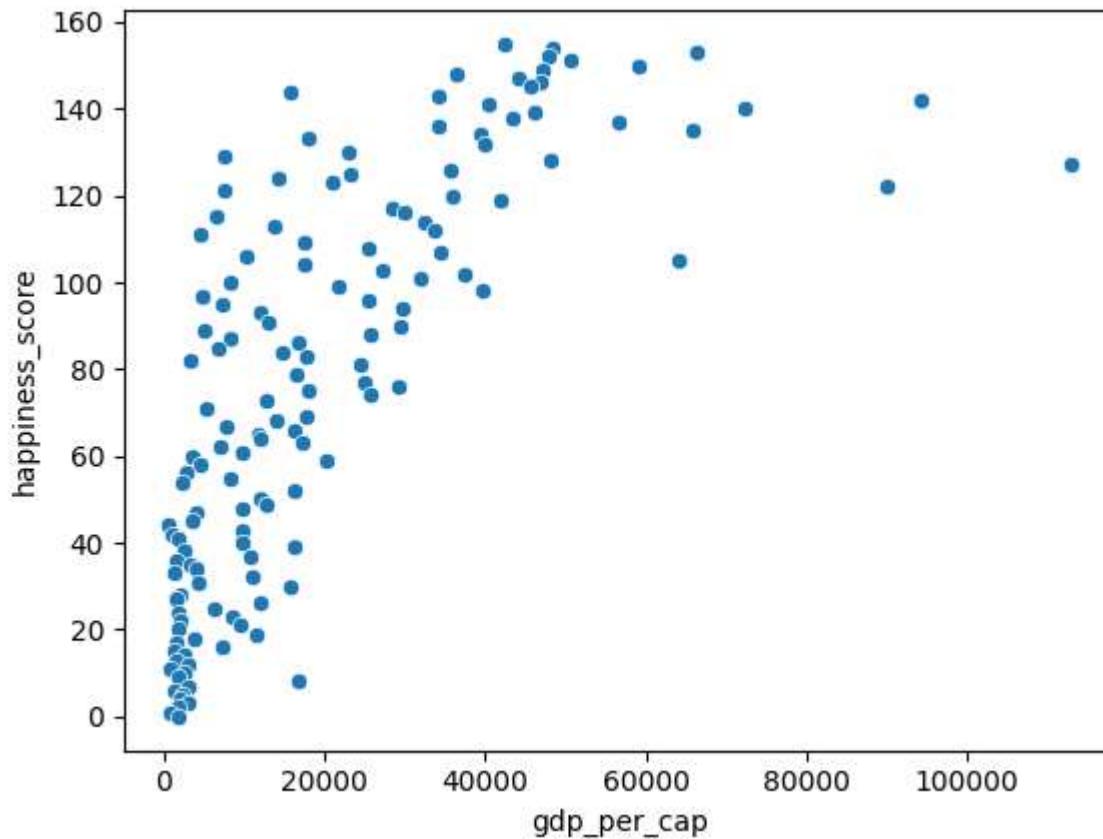
```
# Correlation between gdp_per_cap and life_exp  
cor = world_happiness.gdp_per_cap.corr(world_happiness.life_exp)  
  
print(cor)
```



0.7019547642148015

Transforming variables

```
In [ ]:  
# Scatterplot of happiness_score vs. gdp_per_cap  
sns.scatterplot(y='happiness_score', x='gdp_per_cap', data=world_happiness)  
plt.show()  
  
# Calculate correlation  
cor = world_happiness.happiness_score.corr(world_happiness.gdp_per_cap)  
print(cor)
```

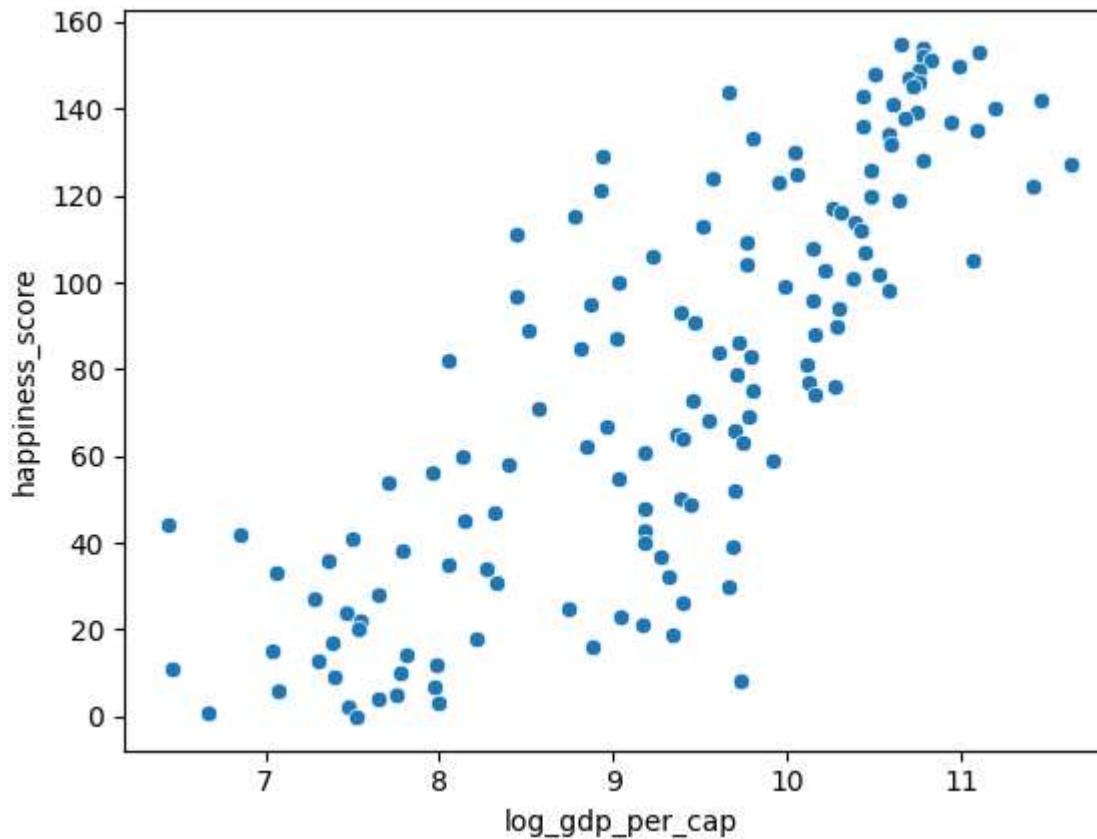


0.7279733012222976

```
In [ ]: # Create log_gdp_per_cap column
world_happiness['log_gdp_per_cap'] = np.log(world_happiness.gdp_per_cap)

# Scatterplot of log_gdp_per_cap and happiness_score
sns.scatterplot(x='log_gdp_per_cap', y='happiness_score', data=world_happiness)
plt.show()

# Calculate correlation
cor = world_happiness['log_gdp_per_cap'].corr(world_happiness.happiness_score)
print(cor)
```



0.8043146004918288

Does sugar improve happiness?

```
In [ ]: # Scatterplot of grams_sugar_per_day and happiness_score
sns.scatterplot(x='grams_sugar_per_day', y='happiness_score', data=world_happiness)
plt.show()

# Correlation between grams_sugar_per_day and happiness_score
cor = world_happiness.grams_sugar_per_day.corr(world_happiness.happiness_score)
print(cor)
```

# 9. Manipulating Time Series Data in Python

## Chapter 1 - Working with Time Series in Pandas

In this course you'll learn the basics of manipulating time series data. Time series data are data that are indexed by a sequence of dates or times. You'll learn how to use methods built into Pandas to work with this index. You'll also learn how resample time series to change the frequency. This course will also show you how to calculate rolling and cumulative values for times series. Finally, you'll use all your new skills to build a value-weighted stock index from actual stock data.

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

plt.rcParams['figure.figsize'] = (10, 5)
```

Your first time series

```
In [ ]: seven_days = pd.date_range(start='2017-1-1', periods=7)

# Iterate over the dates and print the number and name of the weekday
for day in seven_days:
    print(day.dayofweek, day.day_name())
```

```
6 Sunday
0 Monday
1 Tuesday
2 Wednesday
3 Thursday
4 Friday
5 Saturday
```

Create a time series of air quality data

```
In [ ]: #reading file "NYC"
data = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course D

# Inspect data
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6317 entries, 0 to 6316
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   date     6317 non-null   object  
 1   ozone    6317 non-null   float64 
 2   pm25     6317 non-null   float64 
 3   co       6317 non-null   float64 
dtypes: float64(3), object(1)
memory usage: 197.5+ KB
None
```

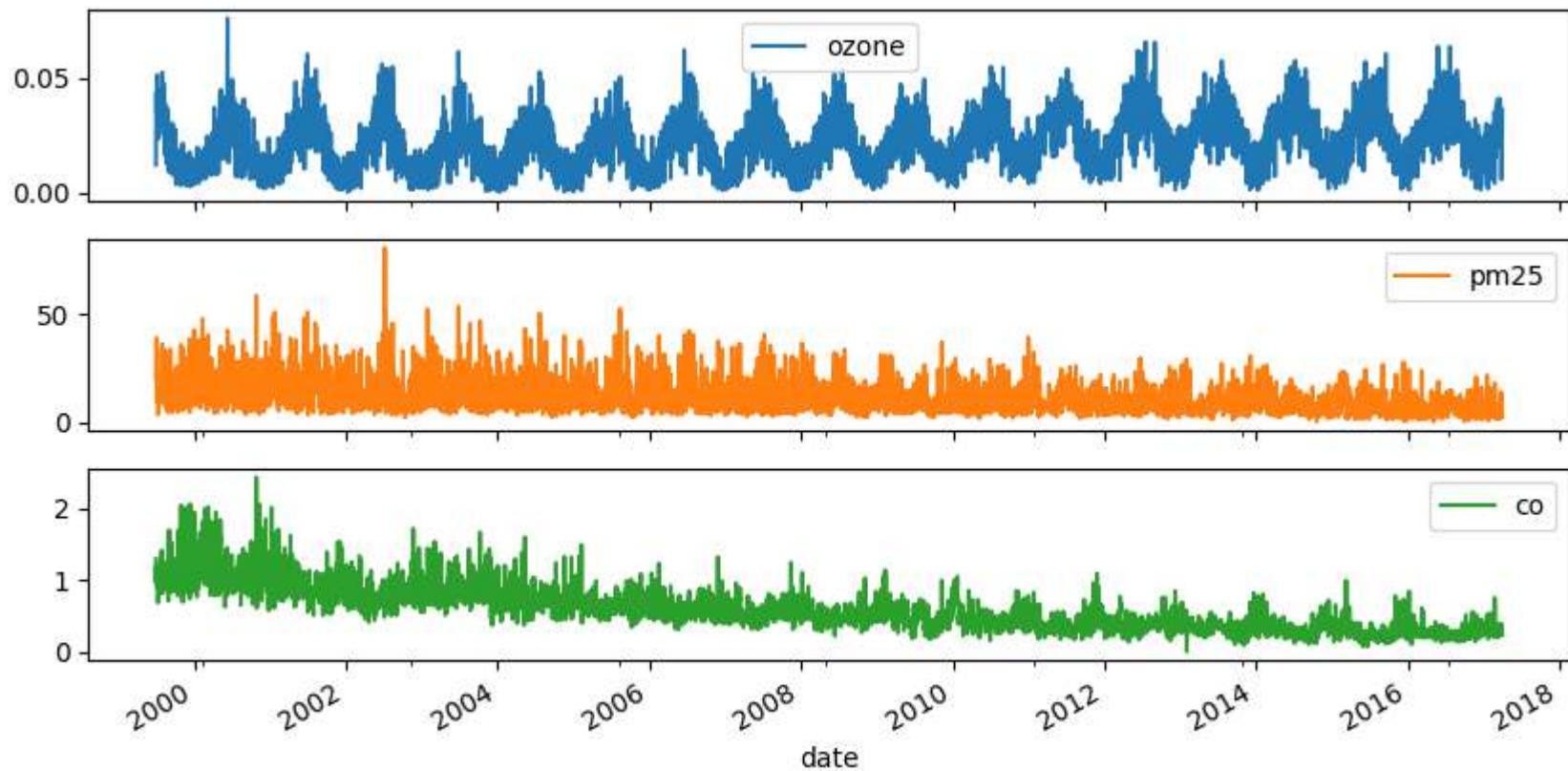
```
In [ ]: # Convert the date column to datetime64
data['date'] = pd.to_datetime(data['date'])
```

```
In [ ]: # Set date column as index
data.set_index('date', inplace=True)
```

```
In [ ]: # Inspect data
print(data.info())

# Plot data
data.plot(subplots=True);
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6317 entries, 1999-07-01 to 2017-03-31
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   ozone    6317 non-null   float64 
 1   pm25     6317 non-null   float64 
 2   co       6317 non-null   float64 
dtypes: float64(3)
memory usage: 197.4 KB
None
```



Compare annual stock price trends

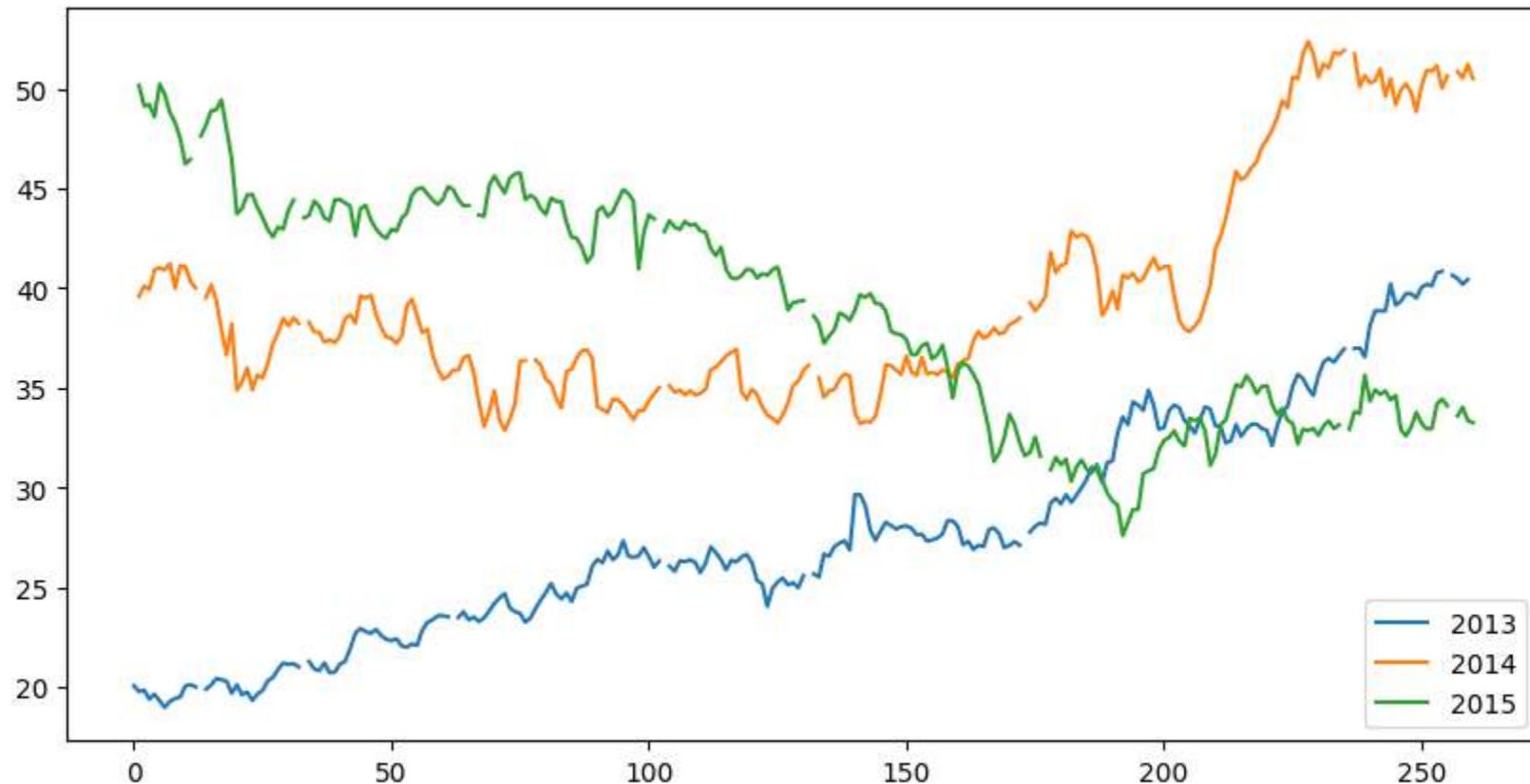
```
In [ ]: df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
#df.drop(['Unnamed: 0'], axis=1, inplace=True)
df.head()
```

```
Out[ ]:      price
date
2013-01-02  20.08
2013-01-03  19.78
2013-01-04  19.86
2013-01-07  19.40
2013-01-08  19.66
```

```
In [ ]: yahoo = df
# Create dataframe prices here
prices = pd.DataFrame()
```

```
# Select data for each year and concatenate with prices here
for year in ['2013', '2014', '2015']:
    price_per_year = yahoo.loc[year, ['price']].reset_index(drop=True)
    price_per_year.rename(columns={'price': year}, inplace=True)
    prices = pd.concat([prices, price_per_year], axis=1)

prices.plot()
plt.show()
```



In [ ]: `print(prices.head())`

	2013	2014	2015
0	20.08	NaN	NaN
1	19.78	39.59	50.17
2	19.86	40.12	49.13
3	19.40	39.93	49.21
4	19.66	40.92	48.59

Set and change time series frequency

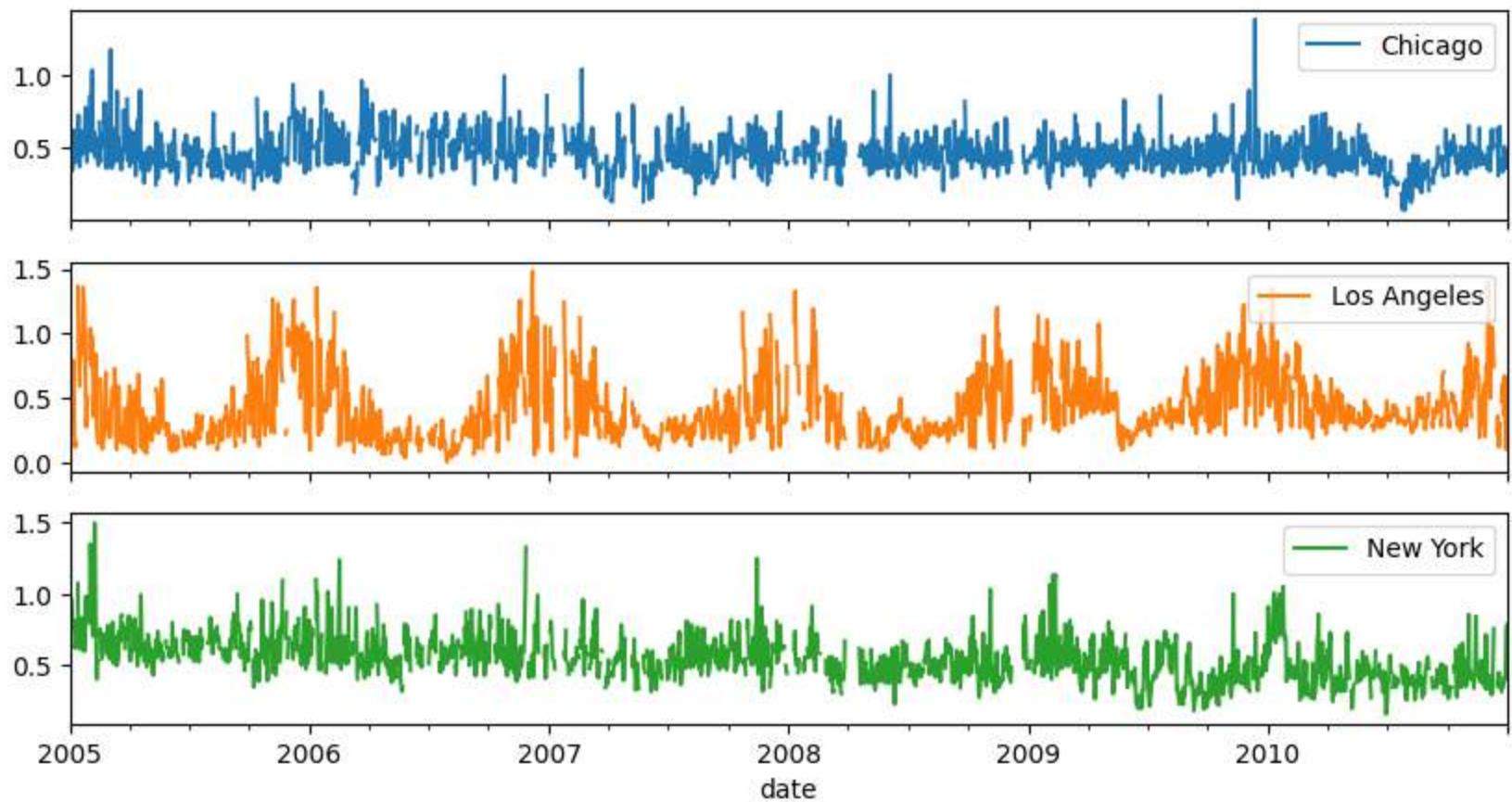
```
In [ ]: co = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat  
co['date'] = pd.to_datetime(co['date'])  
co.set_index('date', inplace=True)  
co.head()
```

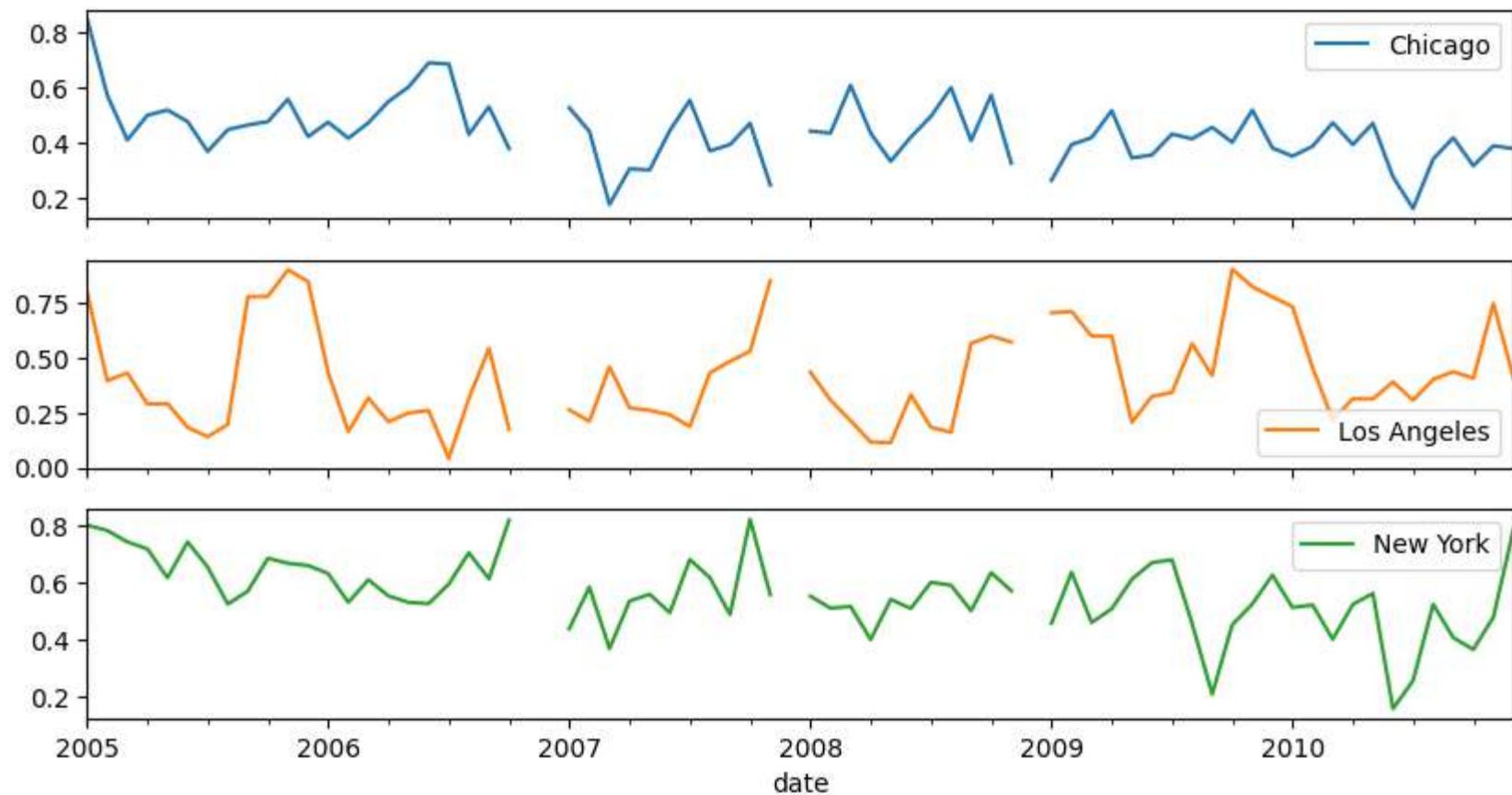
```
Out[ ]: Chicago  Los Angeles  New York
```

	date		
2005-01-01	0.317763	0.777657	0.639830
2005-01-03	0.520833	0.349547	0.969572
2005-01-04	0.477083	0.626630	0.905208
2005-01-05	0.348822	0.613814	0.769176
2005-01-06	0.572917	0.792596	0.815761

```
In [ ]: print(co.info())  
  
# Set the frequency to calendar daily  
co = co.asfreq('D')  
  
# Plot the data  
co.plot(subplots=True);  
plt.show()  
  
# Set Frequency to monthly  
co = co.asfreq('M')  
  
# Plot the data  
co.plot(subplots=True)  
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1898 entries, 2005-01-01 to 2010-12-31  
Data columns (total 3 columns):  
 #   Column      Non-Null Count  Dtype     
---  --    
 0   Chicago     1898 non-null   float64  
 1   Los Angeles  1898 non-null   float64  
 2   New York    1898 non-null   float64  
dtypes: float64(3)  
memory usage: 59.3 KB  
None
```





Shifting stock prices across time

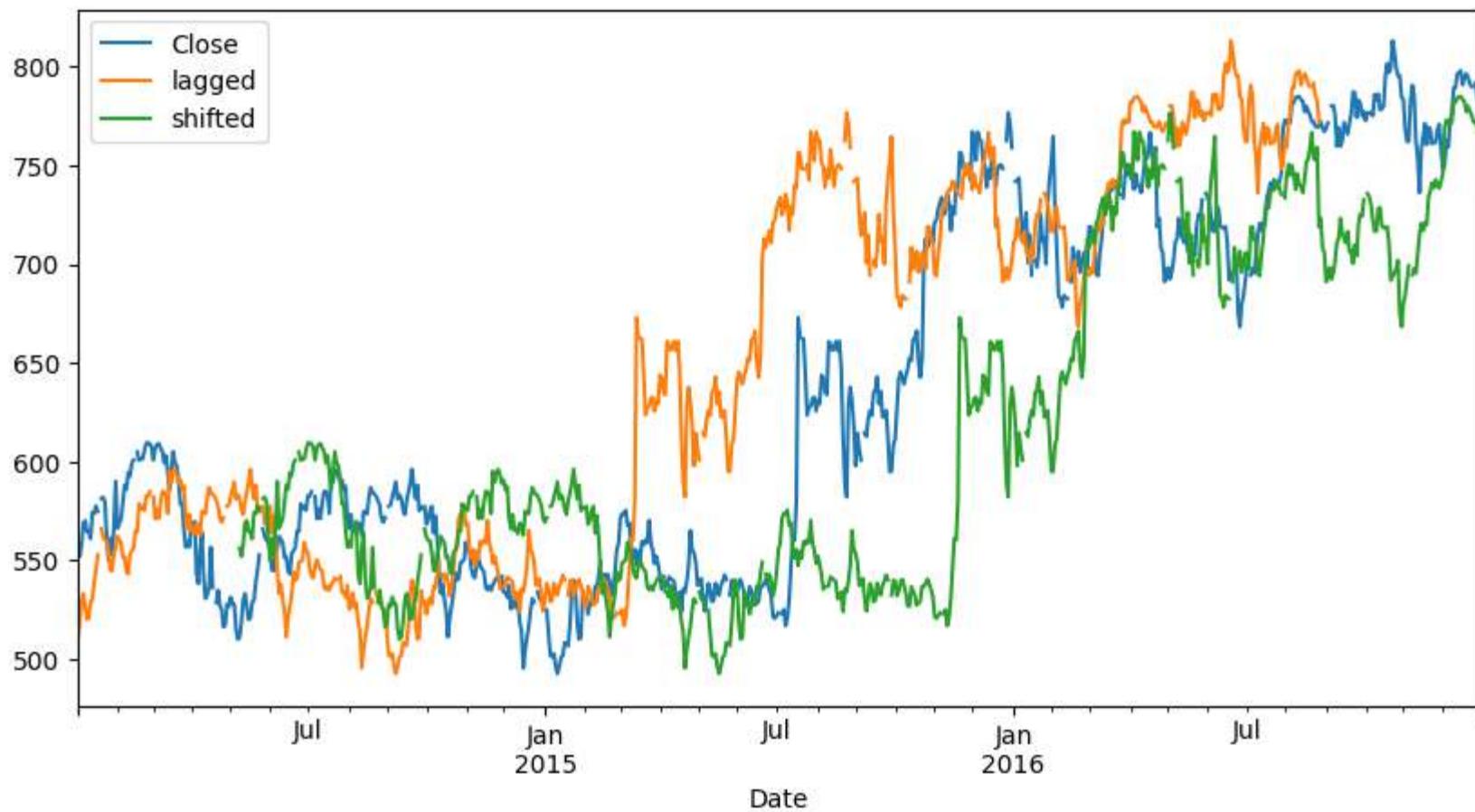
```
In [ ]: google = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\01. Time Series Analysis\\\\01. Shifting Stock Prices\\\\Google Stock Prices.csv')

# Set data frequency to business daily
google = google.asfreq('B')

# Create 'lagged' and 'shifted'
google['lagged'] = google['Close'].shift(periods=-90)
google['shifted'] = google['Close'].shift(periods=90)

# Plot the google price series
google.plot();
plt.savefig('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\01. Time Series Analysis\\\\01. Shifting Stock Prices\\\\Google Price Series Plot.png')
```

```
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1769: FutureWarning: Period with BDay  
freq is deprecated and will be removed in a future version. Use a DatetimeIndex with BDay freq instead.  
    ret = self.converter.convert(x, self.units, self)  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1769: FutureWarning: PeriodDtype[B] i  
s deprecated and will be removed in a future version. Use a DatetimeIndex with freq='B' instead  
    ret = self.converter.convert(x, self.units, self)  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1769: FutureWarning: PeriodDtype[B] i  
s deprecated and will be removed in a future version. Use a DatetimeIndex with freq='B' instead  
    ret = self.converter.convert(x, self.units, self)  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1769: FutureWarning: PeriodDtype[B] i  
s deprecated and will be removed in a future version. Use a DatetimeIndex with freq='B' instead  
    ret = self.converter.convert(x, self.units, self)  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1495: FutureWarning: Period with BDay  
freq is deprecated and will be removed in a future version. Use a DatetimeIndex with BDay freq instead.  
    return self.major.locator()  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\axis.py:1495: FutureWarning: PeriodDtype[B] i  
s deprecated and will be removed in a future version. Use a DatetimeIndex with freq='B' instead  
    return self.major.locator()  
c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplotlib\ticker.py:216: FutureWarning: Period with BDa  
y freq is deprecated and will be removed in a future version. Use a DatetimeIndex with BDay freq instead.  
    return [self(value, i) for i, value in enumerate(values)]
```



Calculating stock price changes

```
In [ ]: yahoo = yahoo.asfreq('B')
```

```
In [ ]: yahoo['shifted_30'] = yahoo['price'].shift(periods=30)

# Subtract shifted_30 from price
yahoo['change_30'] = yahoo['price'] - yahoo['shifted_30']

# Get the 30-day price difference
yahoo['diff_30'] = yahoo['price'].diff(periods=30)

# Inspect the last five rows of price
print(yahoo['price'].tail(5))

# Show the value_counts of the difference between change_30 and diff_30
print(yahoo['diff_30'].sub(yahoo['change_30']).value_counts())
```

```
date
2015-12-25      NaN
2015-12-28    33.60
2015-12-29    34.04
2015-12-30    33.37
2015-12-31    33.26
Freq: B, Name: price, dtype: float64
0.0    703
Name: count, dtype: int64
```

Plotting multi-period returns

```
In [ ]: google = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\Data\\\\Google Stock Data.csv')

# Set data frequency to business daily
google = google.asfreq('D')
```

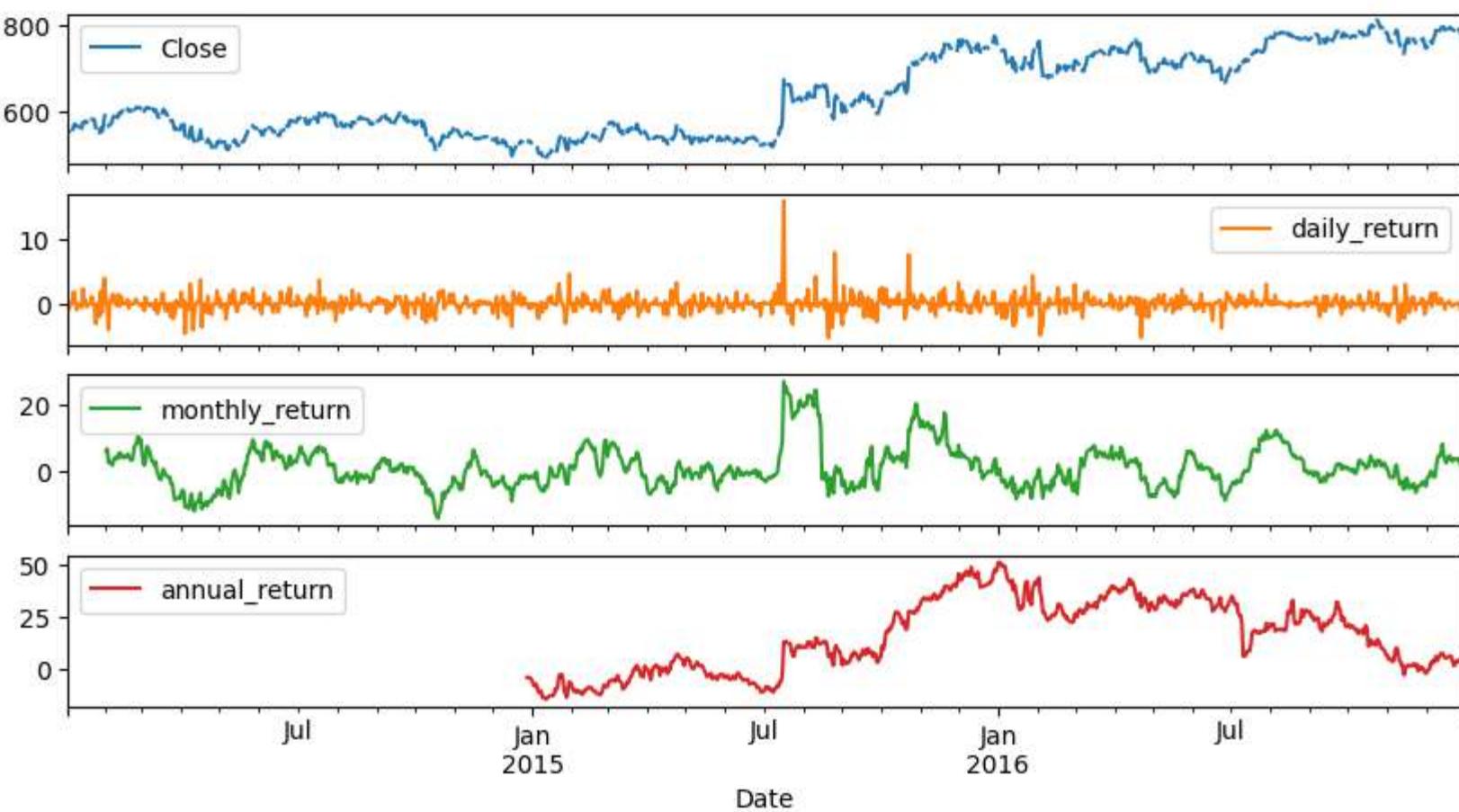
```
In [ ]: google['daily_return'] = google['Close'].pct_change(periods=1) * 100

# Create monthly return
google['monthly_return'] = google['Close'].pct_change(periods=30) * 100

# Create annual return
google['annual_return'] = google['Close'].pct_change(periods=360) * 100

# Plot the result
google.plot(subplots=True);
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_19956\207313826.py:1: FutureWarning: The default fill_method='pad' in Series.pct_change is deprecated and will be removed in a future version. Call ffill before calling pct_change to retain current behavior and silence this warning.
  google['daily_return'] = google['Close'].pct_change(periods=1) * 100
C:\Users\yeiso\AppData\Local\Temp\ipykernel_19956\207313826.py:4: FutureWarning: The default fill_method='pad' in Series.pct_change is deprecated and will be removed in a future version. Call ffill before calling pct_change to retain current behavior and silence this warning.
  google['monthly_return'] = google['Close'].pct_change(periods=30) * 100
C:\Users\yeiso\AppData\Local\Temp\ipykernel_19956\207313826.py:7: FutureWarning: The default fill_method='pad' in Series.pct_change is deprecated and will be removed in a future version. Call ffill before calling pct_change to retain current behavior and silence this warning.
  google['annual_return'] = google['Close'].pct_change(periods=360) * 100
```



## Chapter 2 - Basic Time Series Metrics & Resampling

This chapter dives deeper into the essential time series functionality made available through the pandas `DataTimeIndex`. It introduces resampling and how to compare different time series by normalizing their start points.

Compare the performance of several asset classes

```
In [ ]: # Import data here
# Import 'asset_classes.csv', using .read_csv() to parse dates in the 'DATE'
# column and set this column as the index, then assign the result to prices.
path = 'C:/Users/yeiso/OneDrive - Douglas College/0. DOUGLAS COLLEGE/3. Fund Machine Learning/0. Python Course DataCamp/Course-fun
prices.head()
```

```
Out[ ]: 2013 2014 2015
```

	2013	2014	2015
0	20.08	NaN	NaN
1	19.78	39.59	50.17
2	19.86	40.12	49.13
3	19.40	39.93	49.21
4	19.66	40.92	48.59

```
In [ ]: # Import data here
```

```
prices = pd.read_csv(path+'asset_classes.csv', parse_dates=['DATE'], index_col='DATE')
```

```
# Inspect prices here
```

```
print(prices.info())
```

```
# Select first prices
```

```
first_prices = prices.iloc[0]
```

```
# Create normalized
```

```
normalized = prices.div(first_prices).mul(100)
```

```
# Plot normalized
```

```
normalized.plot()
```

```
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
DatetimeIndex: 2469 entries, 2007-06-29 to 2017-06-26
```

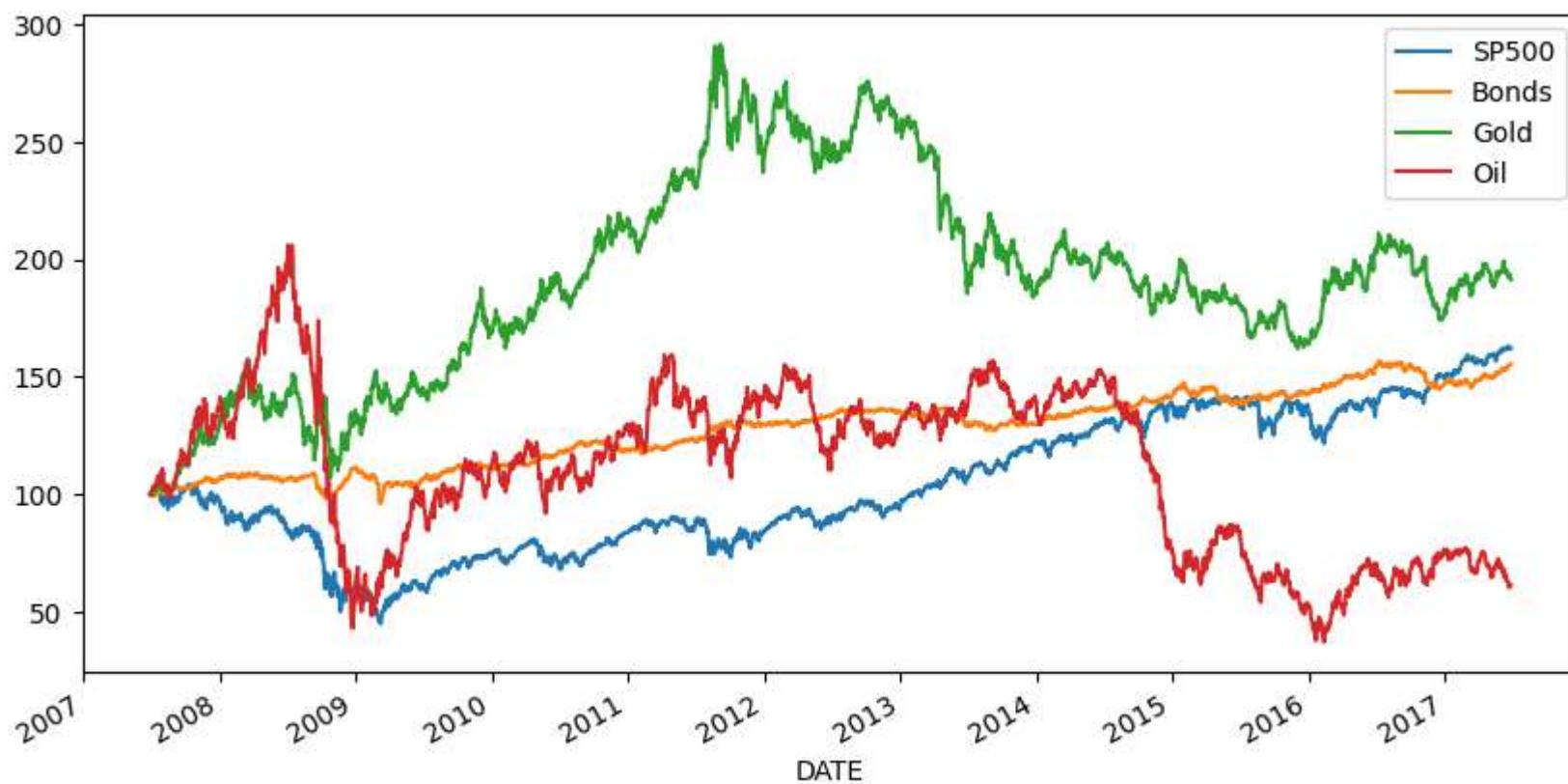
```
Data columns (total 4 columns):
```

#	Column	Non-Null Count	Dtype
0	SP500	2469 non-null	float64
1	Bonds	2469 non-null	float64
2	Gold	2469 non-null	float64
3	Oil	2469 non-null	float64

```
dtypes: float64(4)
```

```
memory usage: 96.4 KB
```

```
None
```



### Comparing stock prices with a benchmark

Compare the performance of various stocks against a benchmark. Learn more about the stock market by comparing the three largest stocks on the NYSE to the Dow Jones Industrial Average, which contains the 30 largest US companies.

The three largest companies on the NYSE are:

Company (Stock Ticker):

Johnson & Johnson (JNJ) Exxon Mobil (XOM) JP Morgan Chase (JPM)

```
In [ ]: # Import stock prices and index here
stocks = pd.read_csv(path+'nyse.csv', parse_dates=['date'], index_col='date')
dow_jones = pd.read_csv(path+'dow_jones.csv', parse_dates=['date'], index_col='date')
```

```
In [ ]: stocks.head()
```

```
Out[ ]:
```

JNJ JPM XOM

**date**

2010-01-04	64.68	42.85	69.15
2010-01-05	63.93	43.68	69.42
2010-01-06	64.45	43.92	70.02
2010-01-07	63.99	44.79	69.80
2010-01-08	64.21	44.68	69.52

```
In [ ]:
```

```
dow_jones.head()
```

```
Out[ ]:
```

DJIA

**date**

2010-01-04	10583.96
2010-01-05	10572.02
2010-01-06	10573.68
2010-01-07	10606.86
2010-01-08	10618.19

```
In [ ]:
```

```
# Concatenate data and inspect result here
# Use pd.concat() along axis=1 to combine stocks and dow_jones
# and assign the result to data. Inspect the .info() of data.
data = pd.concat([stocks, dow_jones], axis=1)
print(data.info())
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   JNJ      1762 non-null   float64
 1   JPM      1762 non-null   float64
 2   XOM      1762 non-null   float64
 3   DJIA     1762 non-null   float64
dtypes: float64(4)
memory usage: 68.8 KB
None
```

```
Out[ ]:
```

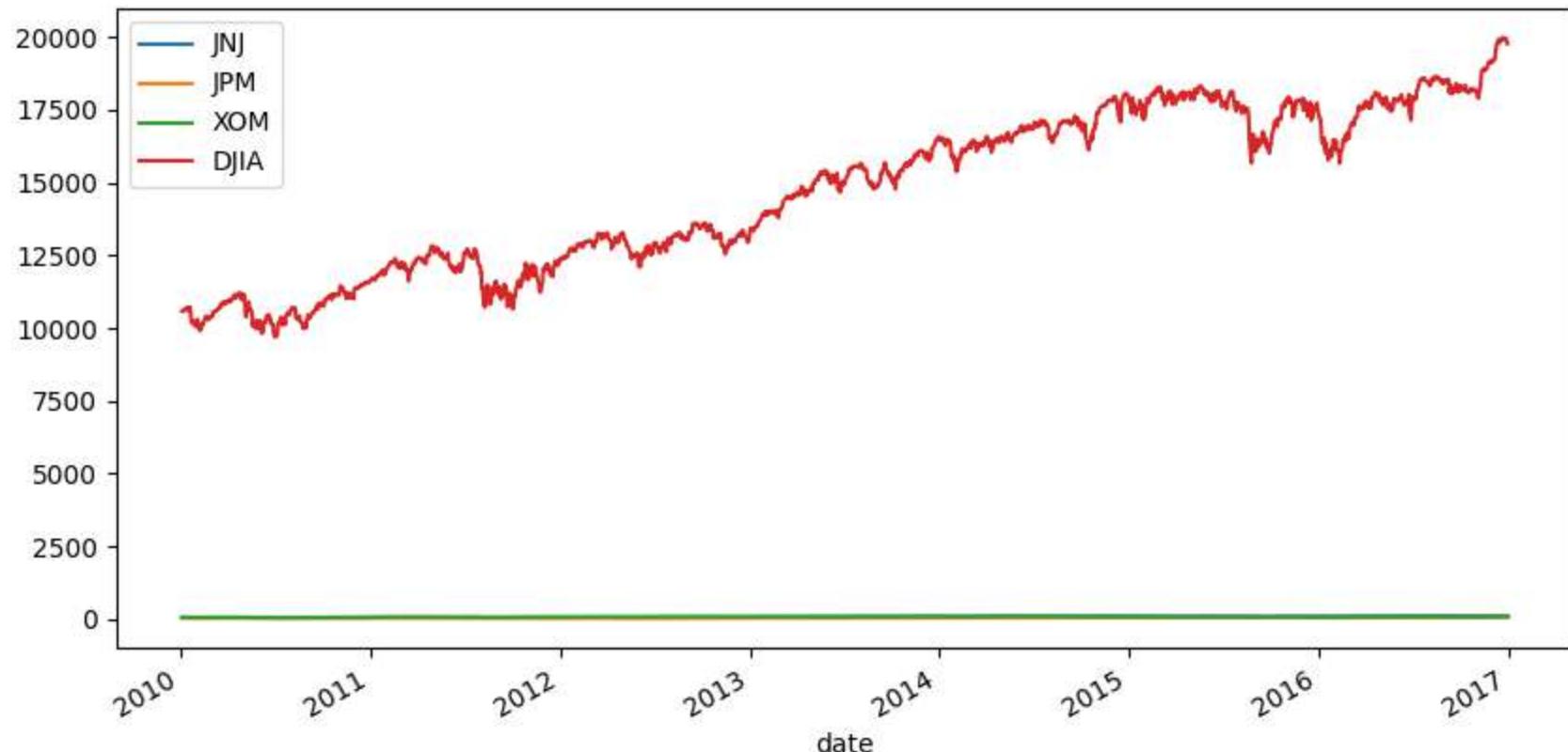
	JNJ	JPM	XOM	DJIA
--	-----	-----	-----	------

date	JNJ	JPM	XOM	DJIA
------	-----	-----	-----	------

2010-01-04	64.68	42.85	69.15	10583.96
2010-01-05	63.93	43.68	69.42	10572.02
2010-01-06	64.45	43.92	70.02	10573.68
2010-01-07	63.99	44.79	69.80	10606.86
2010-01-08	64.21	44.68	69.52	10618.19

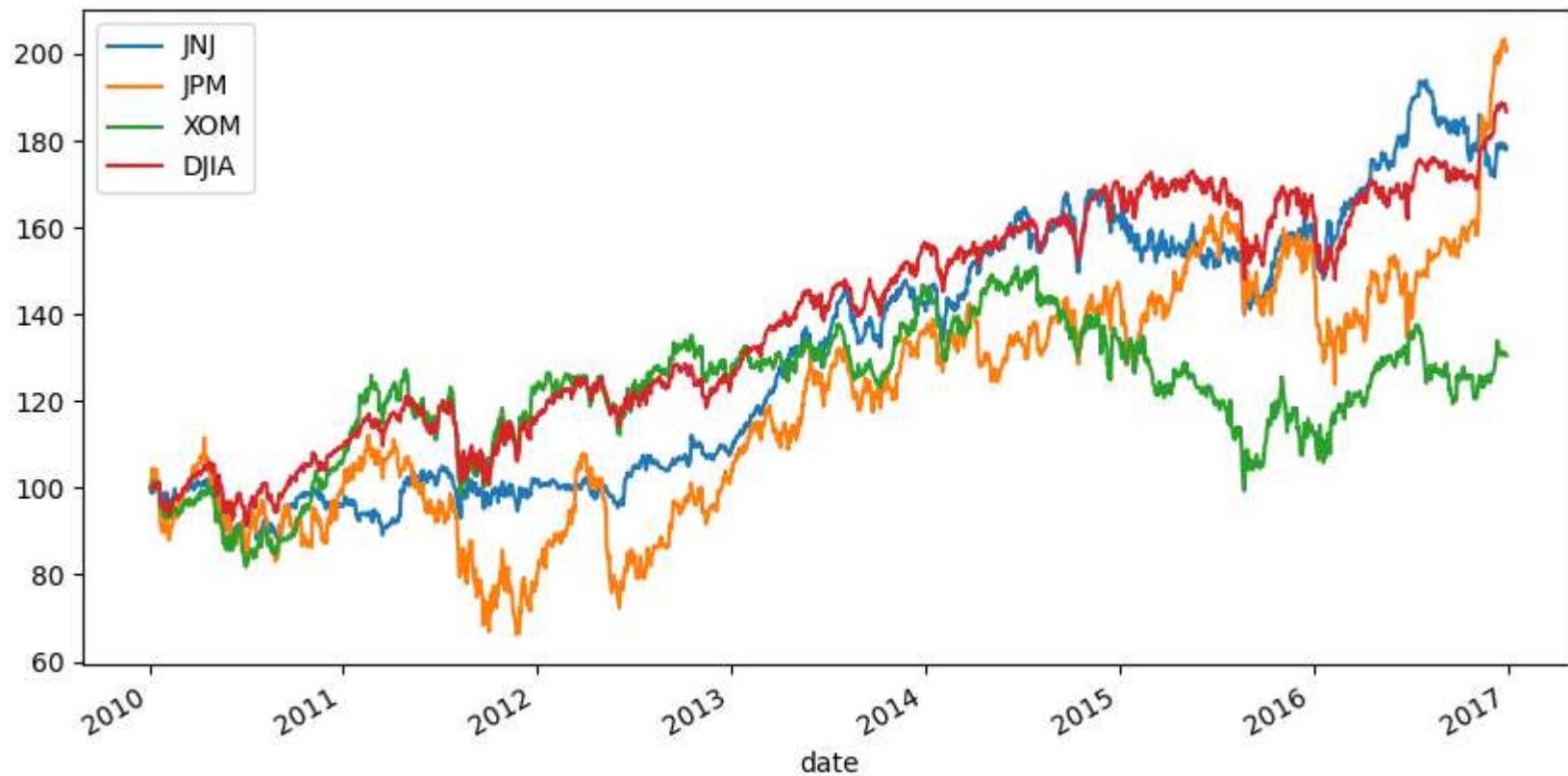
```
In [ ]:
```

```
data.plot()  
plt.show();
```



```
In [ ]:
```

```
# Normalize and plot your data here  
data.div(data.iloc[0]).mul(100).plot()  
plt.show();
```



Plot performance difference vs benchmark index

```
In [ ]: # Import stock data here
stocks = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
stocks.head()
```

```
Out[ ]: AAPL MSFT
```

	AAPL	MSFT
date		
2007-01-03	11.97	29.86
2007-01-04	12.24	29.81
2007-01-05	12.15	29.64
2007-01-08	12.21	29.93
2007-01-09	13.22	29.96

```
In [ ]: # Import index here
sp500 = pd.read_csv(path+'sp500.csv', parse_dates=['date'], index_col='date')
sp500.head()
```

```
Out[ ]:
```

SP500

**date**

2007-06-29	1503.35
2007-07-02	1519.43
2007-07-03	1524.87
2007-07-05	1525.40
2007-07-06	1530.44

```
In [ ]:
```

```
# Concatenate stocks and index here
# Use pd.concat() to concatenate stocks and sp500 along axis=1,
# apply .dropna() to drop all missing values, and assign the result to data.
data = pd.concat([stocks, sp500], axis=1).dropna()
data.head()
```

```
Out[ ]:
```

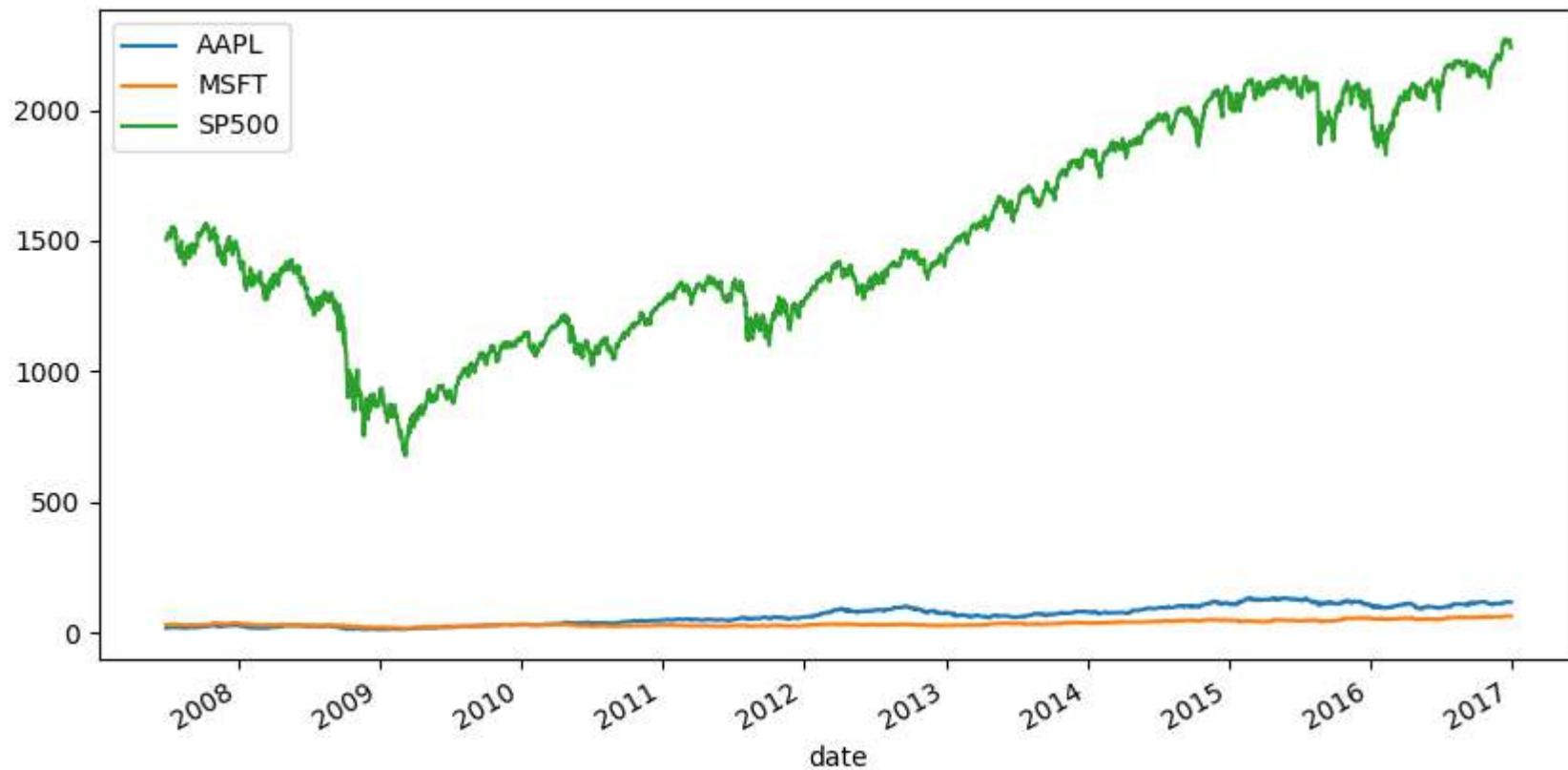
AAPL MSFT SP500

**date**

2007-06-29	17.43	29.47	1503.35
2007-07-02	17.32	29.74	1519.43
2007-07-03	18.17	30.02	1524.87
2007-07-05	18.96	29.99	1525.40
2007-07-06	18.90	29.97	1530.44

```
In [ ]:
```

```
data.plot()
plt.show();
```



```
In [ ]: # Normalize data
normalized = data.div(data.iloc[0]).mul(100)
normalized.head()
```

```
Out[ ]:          AAPL      MSFT      SP500
date
2007-06-29  100.000000  100.000000  100.000000
2007-07-02   99.368904  100.916186  101.069611
2007-07-03  104.245554  101.866305  101.431470
2007-07-05  108.777969  101.764506  101.466724
2007-07-06  108.433735  101.696641  101.801976
```

```
In [ ]: normalized.plot()
plt.show();
```



```
In [ ]: # Create tickers
tickers = ['MSFT', 'AAPL']

# Subtract the normalized index from the normalized stock prices, and plot the result
# Select tickers from normalized, and subtract normalized['SP500']
# with keyword axis=0 to align the indexes, then plot the result.
normalized[tickers].sub(normalized['SP500'], axis=0).plot()
plt.show();
```



Now you can compare these stocks to the overall market so you can more easily spot trends and outliers.

Convert monthly to weekly data

```
In [ ]: import pandas as pd  
  
# Set start and end dates  
start = '2016-1-1'  
end = '2016-2-29'
```

```
In [ ]: # Create monthly_dates here  
# Create monthly_dates using pd.date_range with start,  
# end and frequency alias 'M'.  
monthly_dates = pd.date_range(start=start, end=end, freq='M')  
print(monthly_dates)
```

```
DatetimeIndex(['2016-01-31', '2016-02-29'], dtype='datetime64[ns]', freq='M')
```

```
In [ ]: # Create and print monthly here  
# Create and print the pd.Series monthly, passing the list [1, 2]  
# as the data argument, and using monthly_dates as index.
```

```
monthly = pd.Series(data=[1, 2], index=monthly_dates)
print(monthly)
```

```
2016-01-31    1
2016-02-29    2
Freq: M, dtype: int64
```

```
In [ ]: # Create weekly_dates here
```

```
weekly_dates = pd.date_range(start=start, end=end, freq='W')
print(weekly_dates)
```

```
DatetimeIndex(['2016-01-03', '2016-01-10', '2016-01-17', '2016-01-24',
                 '2016-01-31', '2016-02-07', '2016-02-14', '2016-02-21',
                 '2016-02-28'],
                dtype='datetime64[ns]', freq='W-SUN')
```

```
In [ ]: # Print monthly, reindexed using weekly_dates
```

```
# Apply .reindex() to monthly three times: first without additional options,
# then with ffill and then with bfill, print()-ing each result.
print(monthly.reindex(weekly_dates))
```

```
2016-01-03    NaN
2016-01-10    NaN
2016-01-17    NaN
2016-01-24    NaN
2016-01-31    1.0
2016-02-07    NaN
2016-02-14    NaN
2016-02-21    NaN
2016-02-28    NaN
Freq: W-SUN, dtype: float64
```

```
In [ ]: print(monthly.reindex(weekly_dates, method='bfill'))
```

```
2016-01-03    1
2016-01-10    1
2016-01-17    1
2016-01-24    1
2016-01-31    1
2016-02-07    2
2016-02-14    2
2016-02-21    2
2016-02-28    2
Freq: W-SUN, dtype: int64
```

```
In [ ]: print(monthly.reindex(weekly_dates, method='ffill'))
```

```
2016-01-03    NaN
2016-01-10    NaN
2016-01-17    NaN
2016-01-24    NaN
2016-01-31    1.0
2016-02-07    1.0
2016-02-14    1.0
2016-02-21    1.0
2016-02-28    1.0
Freq: W-SUN, dtype: float64
```

### Create weekly from monthly unemployment data

```
In [ ]: # Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
#df.drop(['Debt/GDP'], axis=1, inplace=True)
df.head()
```

```
Out[ ]:          Debt/GDP  Unemployment
```

	date	
2010-01-01	87.00386	9.8
2010-02-01	NaN	9.8
2010-03-01	NaN	9.9
2010-04-01	88.67047	9.9
2010-05-01	NaN	9.6

all example complete!!!

```
In [ ]: # Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
data=df

# Show first five rows of weekly series
print(data.asfreq('W').head(5))

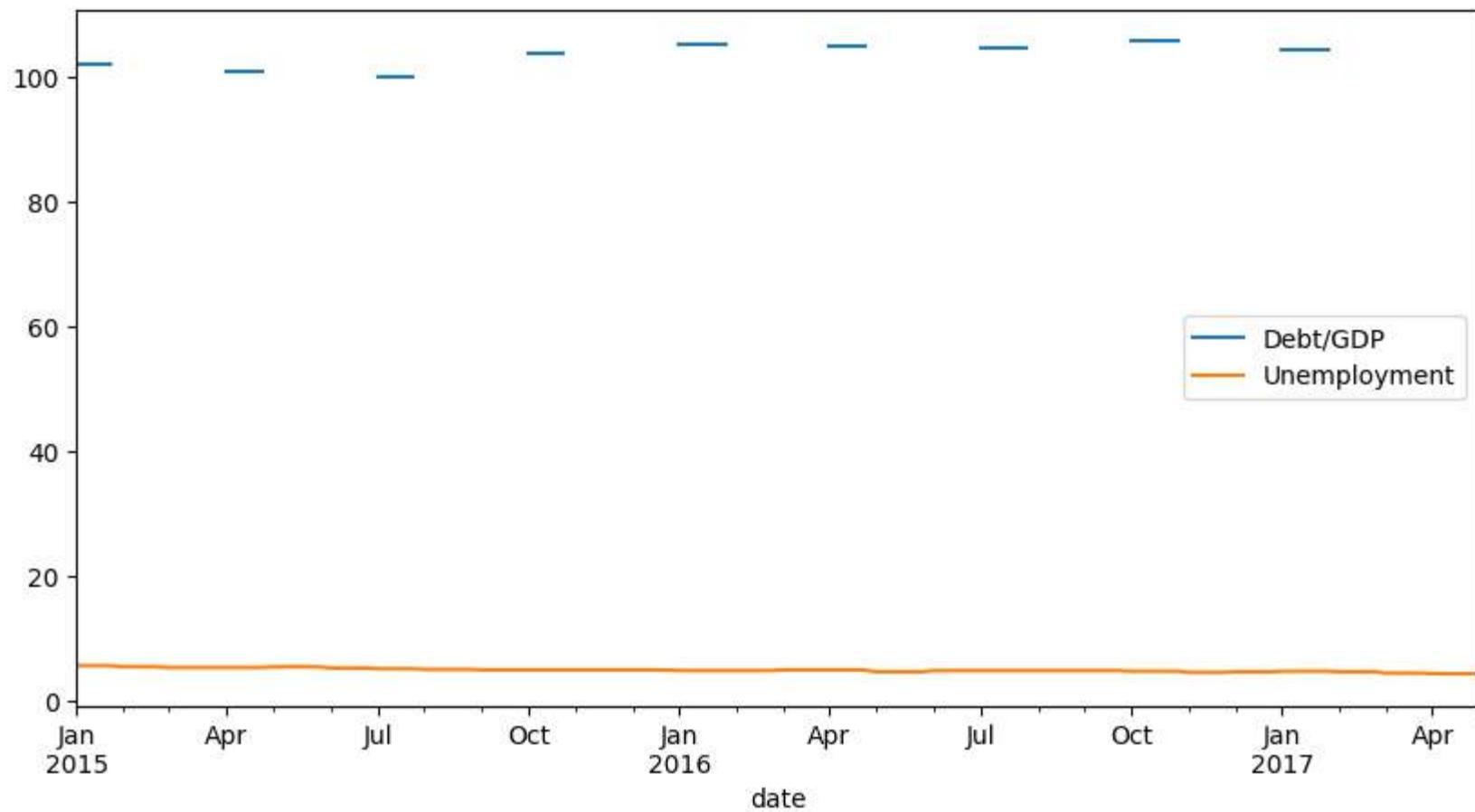
# Show first five rows of weekly series with bfill option
print(data.asfreq('W', method='bfill').head(5))

# Create weekly series with ffill option and show first five rows
weekly_ffill = data.asfreq('W', method='ffill')
print(weekly_ffill.head())

# Plot weekly_fill starting 2015 here
```

```
weekly_ffill['2015'].plot()  
plt.show();
```

```
Debt/GDP  Unemployment  
date  
2010-01-03      NaN        NaN  
2010-01-10      NaN        NaN  
2010-01-17      NaN        NaN  
2010-01-24      NaN        NaN  
2010-01-31      NaN        NaN  
Debt/GDP  Unemployment  
date  
2010-01-03      NaN    9.8  
2010-01-10      NaN    9.8  
2010-01-17      NaN    9.8  
2010-01-24      NaN    9.8  
2010-01-31      NaN    9.8  
Debt/GDP  Unemployment  
date  
2010-01-03  87.00386    9.8  
2010-01-10  87.00386    9.8  
2010-01-17  87.00386    9.8  
2010-01-24  87.00386    9.8  
2010-01-31  87.00386    9.8
```



Use interpolation to create weekly employment data

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

# Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data\\df.drop(['Debt/GDP'], axis=1, inplace=True)
df.head()
```

```
Out[ ]:
```

### Debt/GDP Unemployment

#### date

2010-01-01	87.00386	9.8
2010-02-01	NaN	9.8
2010-03-01	NaN	9.9
2010-04-01	88.67047	9.9
2010-05-01	NaN	9.6

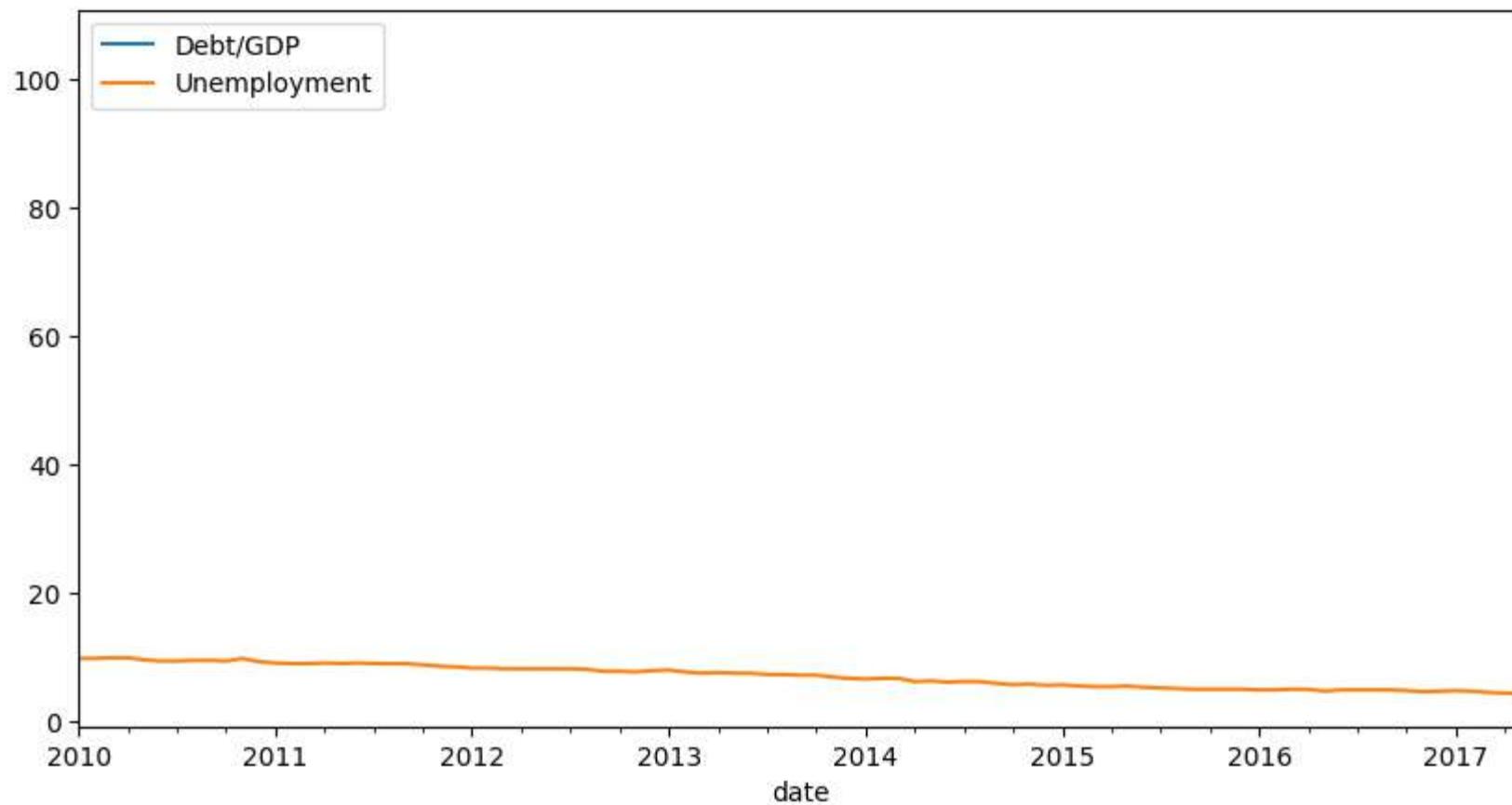
```
In [ ]:
```

```
monthly = df  
# Inspect data here  
print(monthly.info())
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 89 entries, 2010-01-01 to 2017-05-01  
Data columns (total 2 columns):  
 #   Column      Non-Null Count  Dtype     
---  --    
 0   Debt/GDP    29 non-null    float64  
 1   Unemployment 89 non-null    float64  
dtypes: float64(2)  
memory usage: 2.1 KB  
None
```

```
In [ ]:
```

```
monthly.plot()  
plt.show();
```



```
In [ ]: # Create weekly dates
# Create a pd.date_range() with weekly dates, using the .min() and .max()
# of the index of monthly as start and end, respectively,
# and assign the result to weekly_dates.
weekly_dates = pd.date_range(start=monthly.index.min(), end=monthly.index.max(), freq='W')
weekly_dates
```

```
Out[ ]: DatetimeIndex(['2010-01-03', '2010-01-10', '2010-01-17', '2010-01-24',
 '2010-01-31', '2010-02-07', '2010-02-14', '2010-02-21',
 '2010-02-28', '2010-03-07',
 ...,
 '2017-02-26', '2017-03-05', '2017-03-12', '2017-03-19',
 '2017-03-26', '2017-04-02', '2017-04-09', '2017-04-16',
 '2017-04-23', '2017-04-30'],
 dtype='datetime64[ns]', length=383, freq='W-SUN')
```

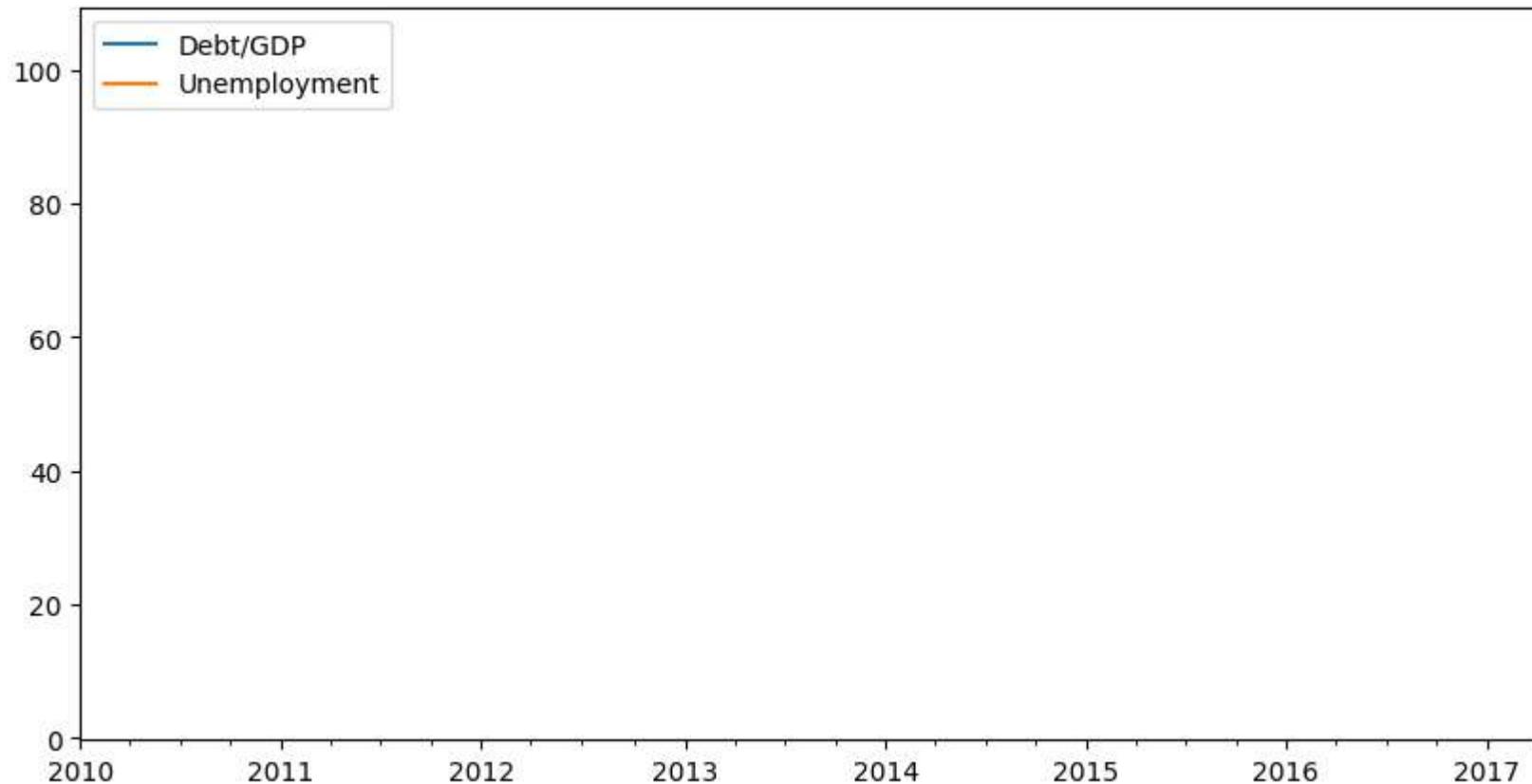
```
In [ ]: # Reindex monthly to weekly data
# Apply .reindex() using weekly_dates to monthly
# and assign the output to weekly.
weekly = monthly.reindex(weekly_dates)
weekly.tail()
```

Out[ ]:

	Debt/GDP	Unemployment
2017-04-02	NaN	NaN
2017-04-09	NaN	NaN
2017-04-16	NaN	NaN
2017-04-23	NaN	NaN
2017-04-30	NaN	NaN

In [ ]:

```
weekly.plot()  
plt.show();
```



In [ ]:

```
# Create ffill and interpolated columns  
weekly['ffill'] = weekly.UNRATE.ffill()  
weekly['interpolated'] = weekly.UNRATE.interpolate()
```

In [ ]:

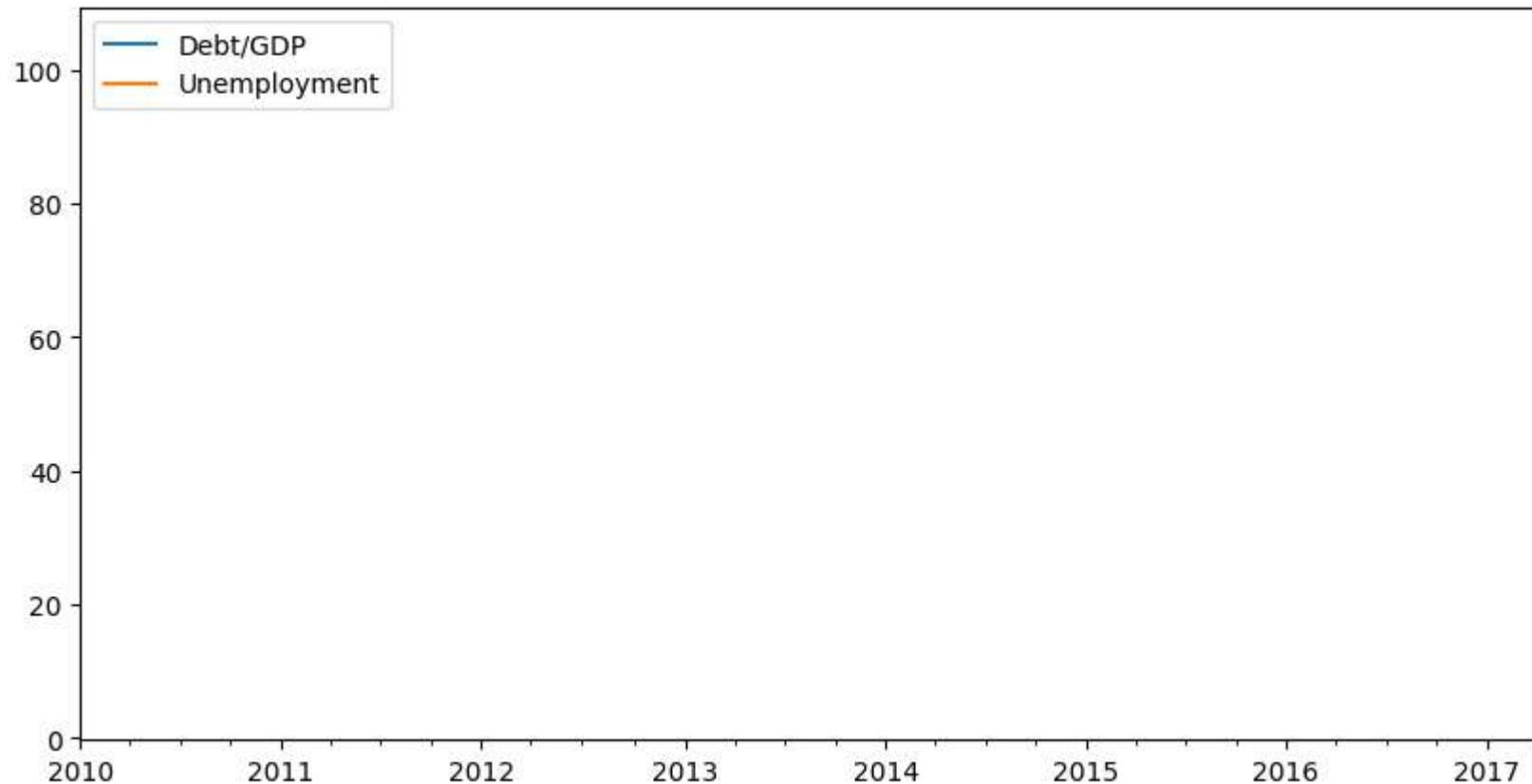
```
weekly.tail()
```

Out[ ]:

	Debt/GDP	Unemployment
2017-04-02	NaN	NaN
2017-04-09	NaN	NaN
2017-04-16	NaN	NaN
2017-04-23	NaN	NaN
2017-04-30	NaN	NaN

In [ ]:

```
# Plot weekly
weekly.plot()
plt.show();
```



### Interpolate debt/GDP and compare to unemployment

Since you have learned how to interpolate time series, you can now apply this new skill to the quarterly debt/GDP series, and compare the result to the monthly unemployment rate.

```
In [ ]: # Import & inspect data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data.csv')
print(data.info())
data.head()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 89 entries, 2010-01-01 to 2017-05-01
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   Debt/GDP    29 non-null     float64 
 1   Unemployment 89 non-null     float64 
dtypes: float64(2)
memory usage: 4.1 KB
None
```

```
Out[ ]:          Debt/GDP  Unemployment
```

	date	
2010-01-01	87.00386	9.8
2010-02-01	NaN	9.8
2010-03-01	NaN	9.9
2010-04-01	88.67047	9.9
2010-05-01	NaN	9.6

```
In [ ]: # Interpolate and inspect here
interpolated = data.interpolate()
print(interpolated.info())
interpolated.head()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 89 entries, 2010-01-01 to 2017-05-01
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   Debt/GDP    89 non-null     float64 
 1   Unemployment 89 non-null     float64 
dtypes: float64(2)
memory usage: 4.1 KB
None
```

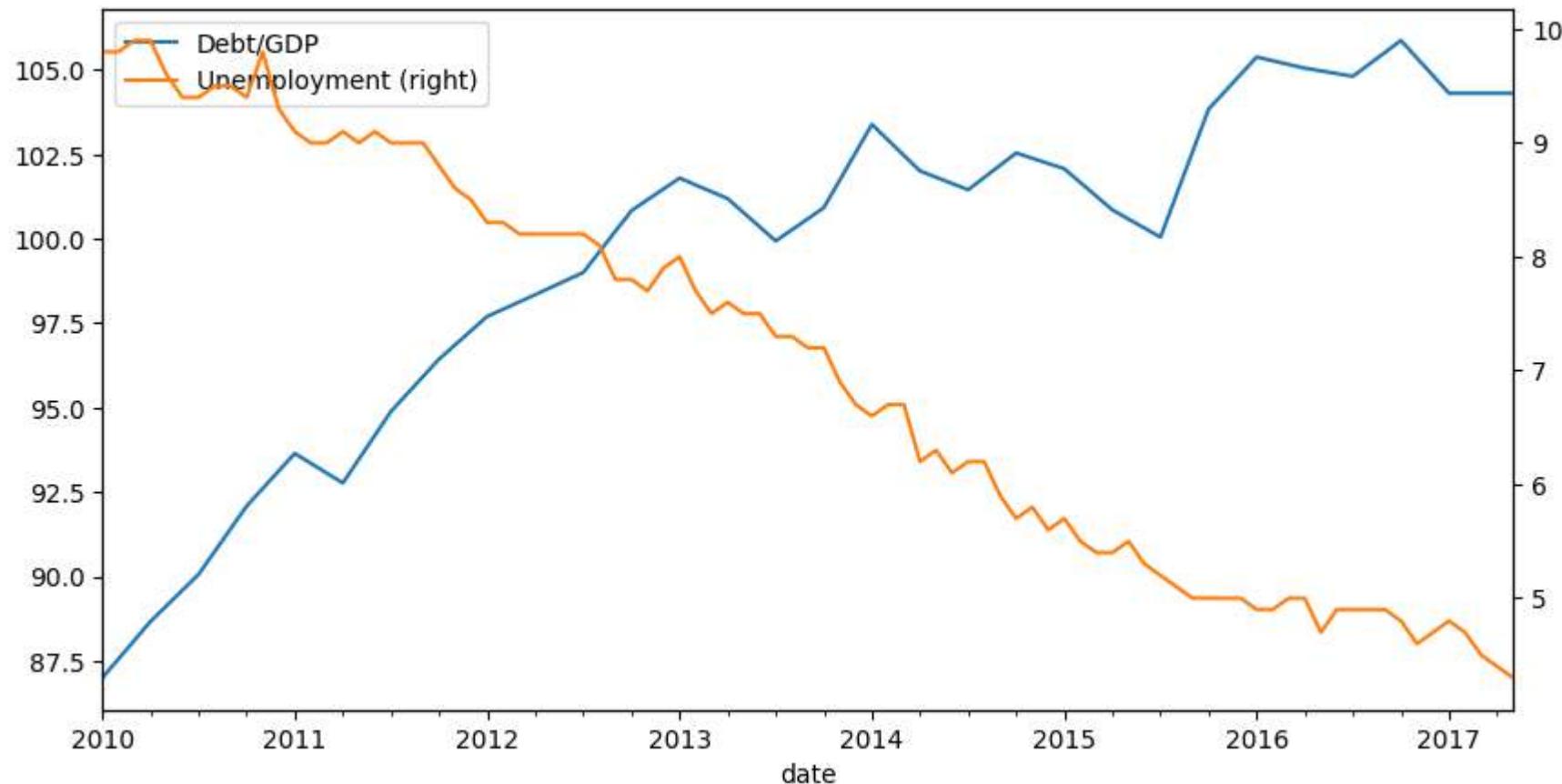
Out[ ]:

### Debt/GDP    Unemployment

date		
2010-01-01	87.003860	9.8
2010-02-01	87.559397	9.8
2010-03-01	88.114933	9.9
2010-04-01	88.670470	9.9
2010-05-01	89.135103	9.6

In [ ]:

```
# Plot interpolated data here
interpolated.plot(secondary_y='Unemployment')
plt.show();
```



## Downsampling & aggregation

Compare weekly, monthly and annual ozone trends for NYC-LA

Downsample and aggregate time series ozone data for both NYC and LA since 2000 to compare the air quality trend at weekly, monthly and annual frequencies and explore how different resampling periods impact the visualization.

In [ ]:

```
# Import and inspect data here
ozone = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
print(ozone.info())
ozone.head()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6291 entries, 2000-01-01 to 2017-03-31
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   Los Angeles  5488 non-null   float64 
 1   New York    6167 non-null   float64 
dtypes: float64(2)
memory usage: 147.4 KB
None
```

Out[ ]:

Los Angeles New York

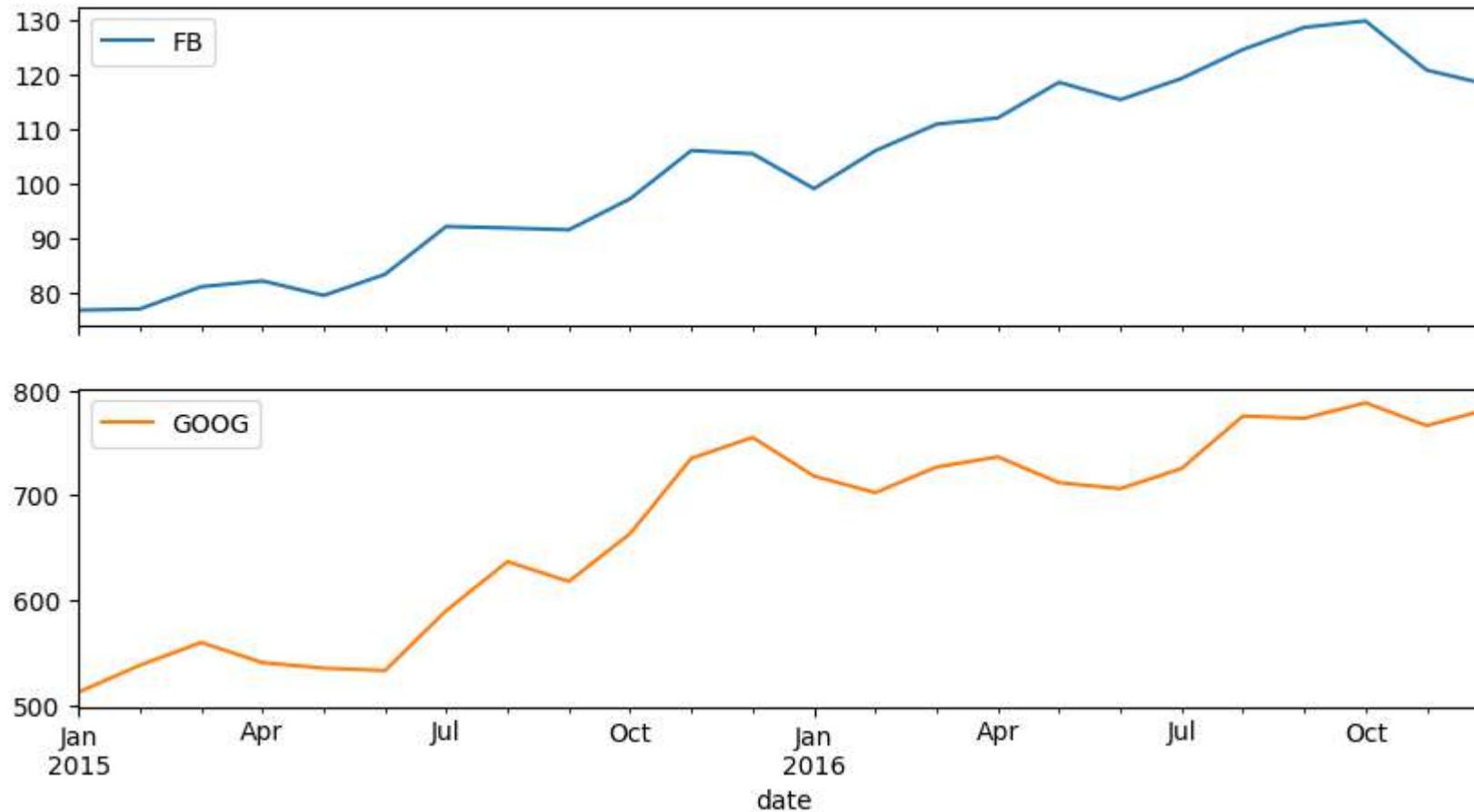
	date	
2000-01-01	0.008375	0.004032
2000-01-02	Nan	0.009486
2000-01-03	Nan	0.005580
2000-01-04	0.005500	0.008717
2000-01-05	0.005000	0.013754

In [ ]:

```
# Import and inspect data here
stocks = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
print(stocks.info())

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 504 entries, 2015-01-02 to 2016-12-30
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  --  
 0   FB      504 non-null   float64 
 1   GOOG    504 non-null   float64 
dtypes: float64(2)
memory usage: 11.8 KB
None
```

```
In [ ]: # Calculate and plot the monthly averages
monthly_average = stocks.resample('M').mean()
monthly_average.plot(subplots=True)
plt.show();
```



### Compare quarterly GDP growth rate and stock returns

With your new skill to downsample and aggregate time series, you can compare higher-frequency stock price series to lower-frequency economic time series.

As a first example, let's compare the quarterly GDP growth rate to the quarterly rate of return on the (resampled) Dow Jones Industrial index of 30 large US stocks.

GDP growth is reported at the beginning of each quarter for the previous quarter. To calculate matching stock returns, you'll resample the stock index to quarter start frequency using the alias 'QS', and aggregating using the .first() observations.

```
In [ ]: # Import and inspect gdp_growth here
gdp_growth = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data\\\\gdp.csv')
gdp_growth.info()

<class 'pandas.core.frame.DataFrame'\>
DatetimeIndex: 41 entries, 2007-01-01 to 2017-01-01
Data columns (total 1 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   gdp_growth  41 non-null    float64 
dtypes: float64(1)
memory usage: 656.0 bytes

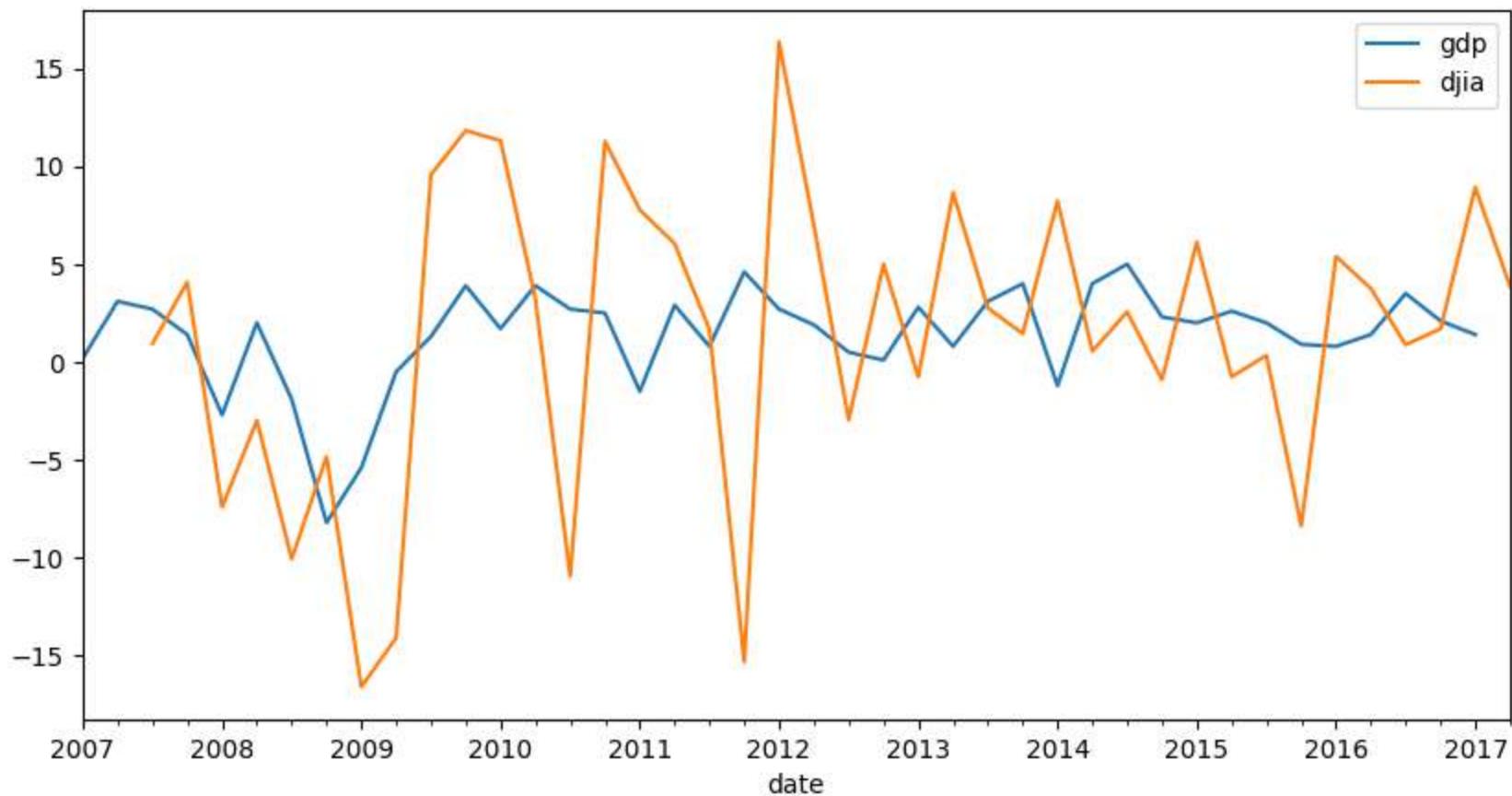
In [ ]: # Import and inspect djia here
djia = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data\\\\djia.csv')
djia.info()

<class 'pandas.core.frame.DataFrame'\>
DatetimeIndex: 2610 entries, 2007-06-29 to 2017-06-29
Data columns (total 1 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   djia        2519 non-null    float64 
dtypes: float64(1)
memory usage: 40.8 KB

In [ ]: # Calculate djia quarterly returns here
djia_quarterly = djia.resample('QS').first()
djia_quarterly_return = djia_quarterly.pct_change().mul(100)

In [ ]: # Concatenate, rename and plot djia_quarterly_return and gdp_growth here
# Use pd.concat() to concatenate gdp_growth and djia_quarterly_return
# along axis=1, and assign to data. Rename the columns using .columns
# and the new labels 'gdp' and 'djia', then .plot() the results.
data = pd.concat([gdp_growth, djia_quarterly_return], axis=1)
data.columns = ['gdp', 'djia']

In [ ]: data.plot()
plt.show();
```



### Visualize monthly mean, median and standard deviation of S&P500 returns

You have also learned how to calculate several aggregate statistics from upsampled data.

Let's use this to explore how the monthly mean, median and standard deviation of daily S&P500 returns have trended over the last 10 years.

```
In [ ]: # Import data here
sp500 = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
print(sp500.info())
sp500.head()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2395 entries, 2007-06-29 to 2016-12-30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   SP500    2395 non-null   float64
dtypes: float64(1)
memory usage: 37.4 KB
None
```

```
Out[ ]:
```

SP500

date

date	SP500
2007-06-29	1503.35
2007-07-02	1519.43
2007-07-03	1524.87
2007-07-05	1525.40
2007-07-06	1530.44

```
In [ ]:
```

```
# Calculate daily returns here
# Convert sp500 to a pd.Series() using .squeeze(),
# and apply .pct_change() to calculate daily_returns.
daily_returns = sp500.squeeze().pct_change()
daily_returns
```

```
Out[ ]:
```

```
date
2007-06-29      NaN
2007-07-02    0.010696
2007-07-03    0.003580
2007-07-05    0.000348
2007-07-06    0.003304
...
2016-12-23    0.001252
2016-12-27    0.002248
2016-12-28   -0.008357
2016-12-29   -0.000293
2016-12-30   -0.004637
Name: SP500, Length: 2395, dtype: float64
```

```
In [ ]:
```

```
# Resample and calculate statistics
# .resample() daily_returns to month-end frequency (alias: 'M'),
# and apply .agg() to calculate 'mean', 'median', and 'std'.
# Assign the result to stats.
stats = daily_returns.resample('M').agg(['mean', 'median', 'std'])
stats
```

Out[ ]:

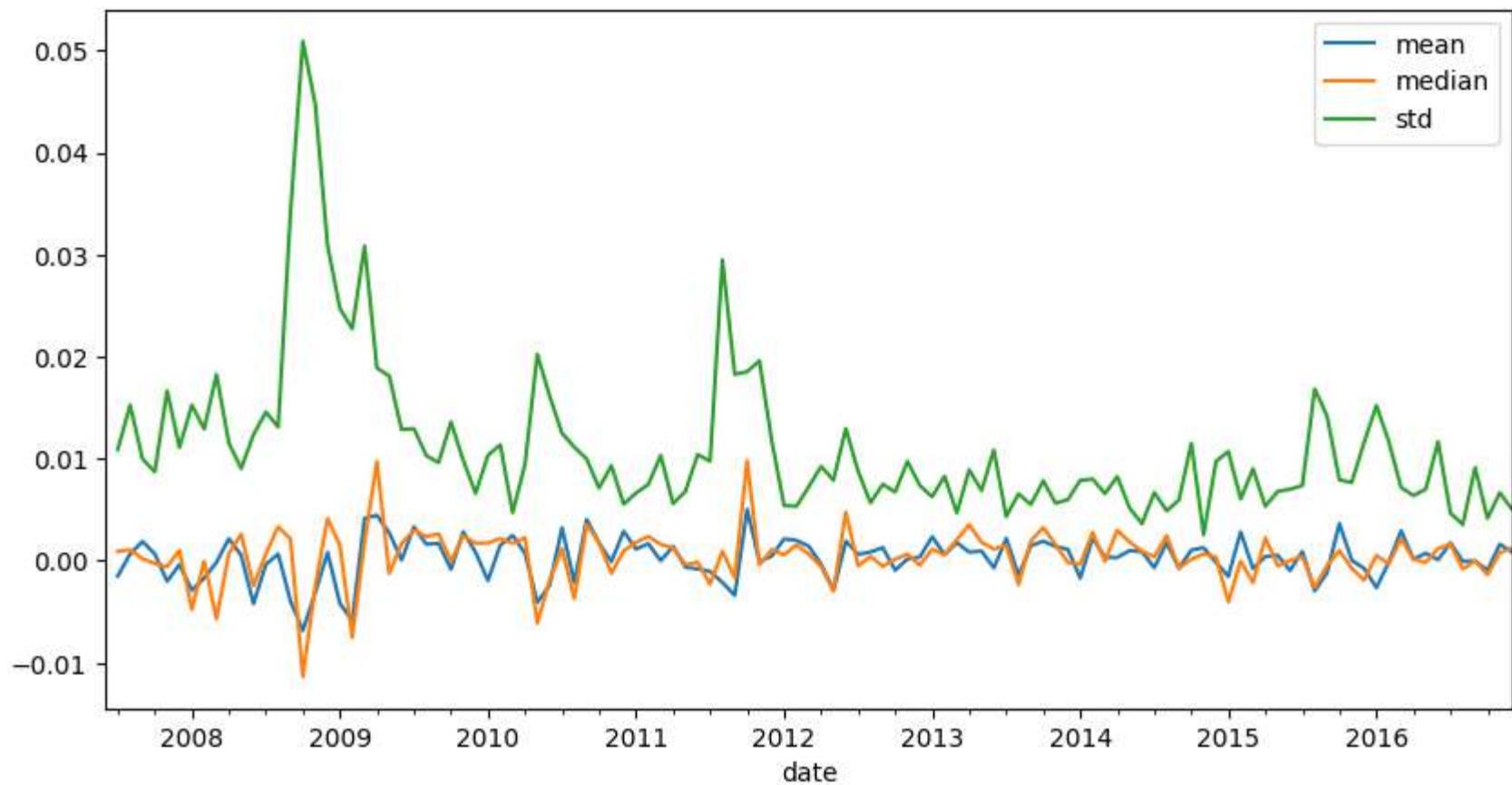
mean median std

date	mean	median	std
2007-06-30	NaN	NaN	NaN
2007-07-31	-0.001490	0.000921	0.010908
2007-08-31	0.000668	0.001086	0.015261
2007-09-30	0.001900	0.000202	0.010000
2007-10-31	0.000676	-0.000265	0.008719
...	...	...	...
2016-08-31	-0.000047	-0.000796	0.003562
2016-09-30	-0.000019	-0.000019	0.009146
2016-10-31	-0.000925	-0.001376	0.004160
2016-11-30	0.001623	0.000808	0.006675
2016-12-31	0.000871	0.001252	0.005040

115 rows × 3 columns

In [ ]:

```
# Plot stats here
stats.plot()
plt.show();
```



In [ ]:

## Chapter 3 - Window Functions: Rolling & Expanding Metrics

### **Rolling average air quality since 2010 for new york city**

To practice rolling window functions, you'll start with air quality trends for New York City since 2010. In particular, you'll be using the daily Ozone concentration levels provided by the Environmental Protection Agency to calculate & plot the 90 and 360 day rolling average.

In [ ]:

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

In [ ]:

```
# Import and inspect ozone data here
data = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course D
print(data.info())
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6291 entries, 2000-01-01 to 2017-03-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   Ozone    6167 non-null   float64 
dtypes: float64(1)
memory usage: 98.3 KB
None
```

### Ozone

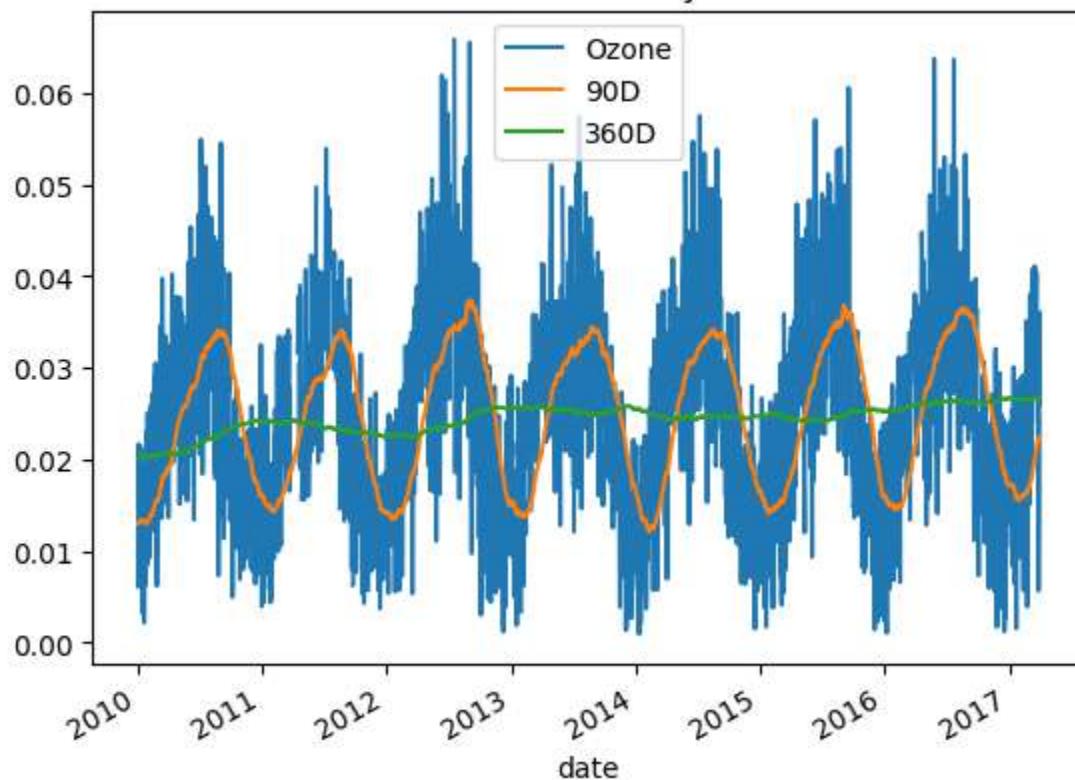
	date
2000-01-01	0.004032
2000-01-02	0.009486
2000-01-03	0.005580
2000-01-04	0.008717
2000-01-05	0.013754

```
In [ ]: # Calculate 90d and 360d rolling mean for the last price
data['90D'] = data.Ozone.rolling(window='90D').mean()
data['360D'] = data.Ozone.rolling(window='360D').mean()
data.head()
```

	Ozone	90D	360D
	date		
2000-01-01	0.004032	0.004032	0.004032
2000-01-02	0.009486	0.006759	0.006759
2000-01-03	0.005580	0.006366	0.006366
2000-01-04	0.008717	0.006954	0.006954
2000-01-05	0.013754	0.008314	0.008314

```
In [ ]: # Plot data
data['2010':].plot()
plt.title('New York City')
plt.show();
```

## New York City



### Rolling 360-day median & std. deviation for nyc ozone data since 2000

Calculate several rolling statistics using the `.agg()` method, similar to `.groupby()`.

Let's take a closer look at the air quality history of NYC using the Ozone data you have seen before. The daily data are very volatile, so using a longer term rolling average can help reveal a longer term trend.

You'll be using a 360 day rolling window, and `.agg()` to calculate the rolling median and standard deviation for the daily average ozone values since 2000.

```
In [ ]: # Import and inspect ozone data here
data = pd.read_csv('C:\\Users\\yeiso\\OneDrive - Douglas College\\0. DOUGLAS COLLEGE\\3. Fund Machine Learning\\0. Python Course D
data.head()
```

## Ozone

### date

<b>2000-01-01</b>	0.004032
<b>2000-01-02</b>	0.009486
<b>2000-01-03</b>	0.005580
<b>2000-01-04</b>	0.008717
<b>2000-01-05</b>	0.013754

In [ ]:

```
# Calculate the rolling mean and std here
rolling_stats = data.Ozone.rolling(360).agg(['mean', 'std'])
rolling_stats.tail()
```

### mean std

### date

<b>2017-03-27</b>	0.026629	0.011599
<b>2017-03-28</b>	0.026583	0.011617
<b>2017-03-29</b>	0.026584	0.011617
<b>2017-03-30</b>	0.026599	0.011613
<b>2017-03-31</b>	0.026607	0.011618

In [ ]:

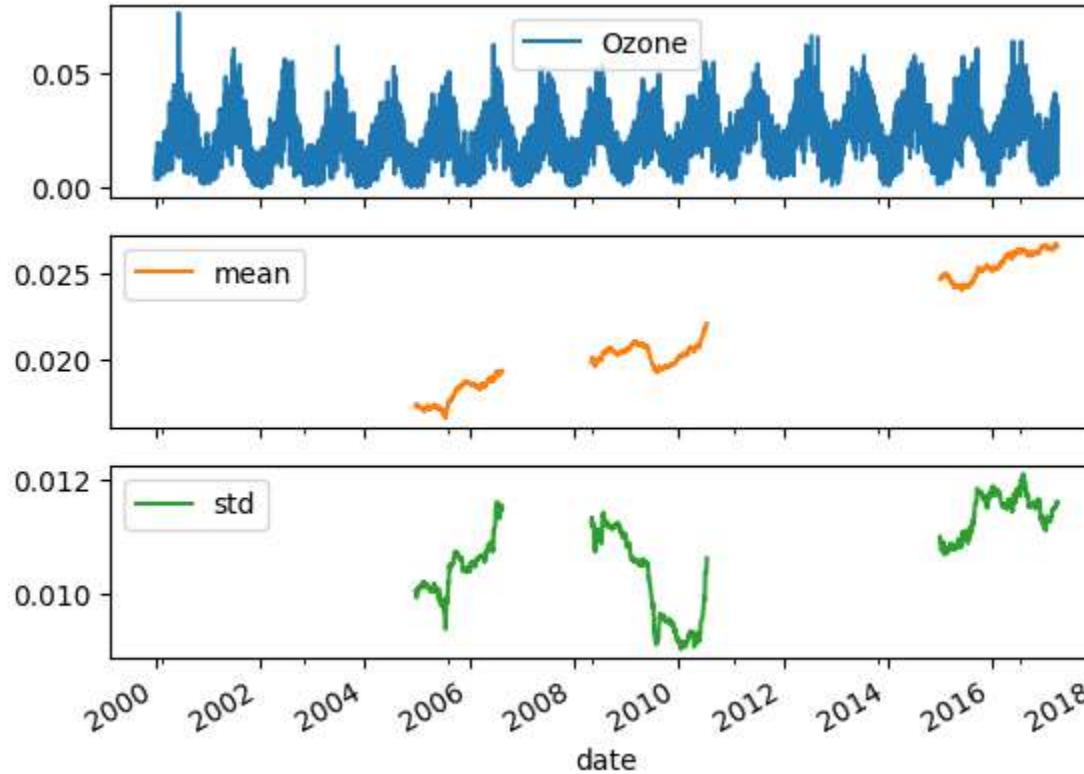
```
# Join rolling_stats with ozone data
stats = data.join(rolling_stats)
stats.head()
```

### Ozone mean std

### date

<b>2000-01-01</b>	0.004032	NaN	NaN
<b>2000-01-02</b>	0.009486	NaN	NaN
<b>2000-01-03</b>	0.005580	NaN	NaN
<b>2000-01-04</b>	0.008717	NaN	NaN
<b>2000-01-05</b>	0.013754	NaN	NaN

```
In [ ]: # Plot stats  
stats.plot(subplots=True)  
plt.show();
```



### Rolling quantiles for daily air quality in nyc

Calculate rolling quantiles to describe changes in the dispersion of a time series over time in a way that is less sensitive to outliers than using the mean and standard deviation.

Let's calculate rolling quantiles - at 10%, 50% (median) and 90% - of the distribution of daily average ozone concentration in NYC using a 360-day rolling window.

```
In [ ]: # Import and inspect ozone data here  
data = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course D  
data.head()
```

## Ozone

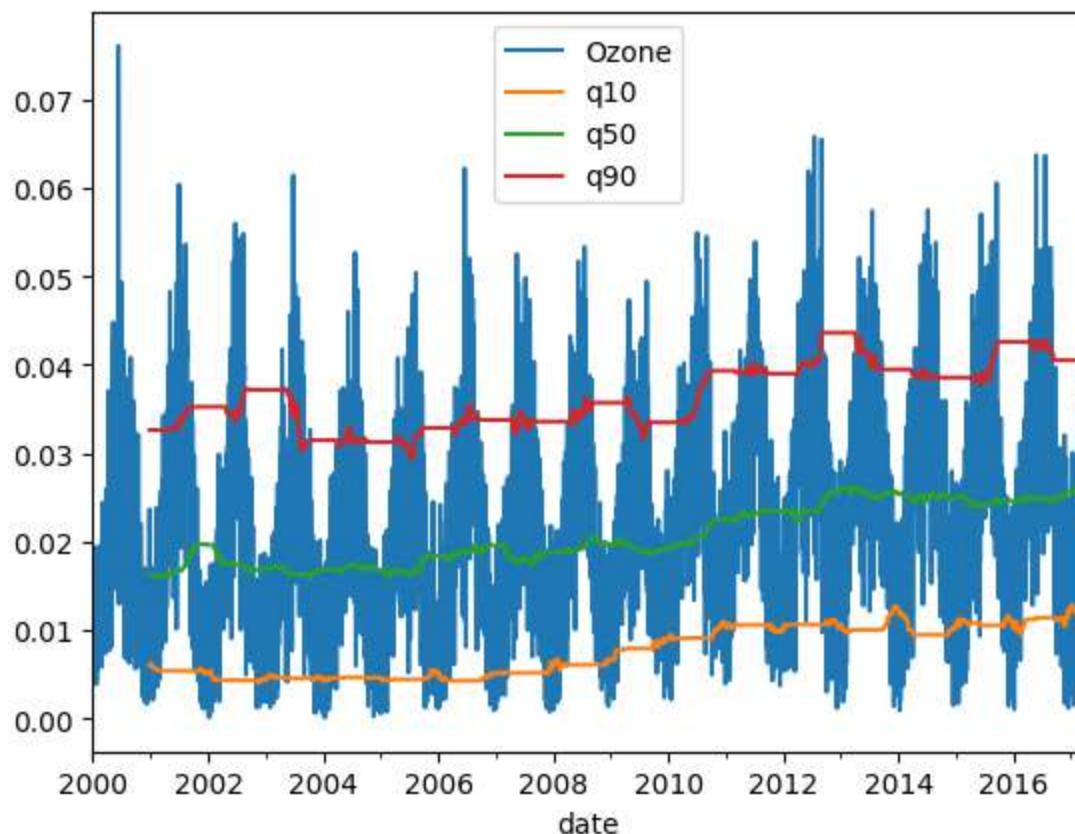
### date

date	Ozone
2000-01-01	0.004032
2000-01-02	0.009486
2000-01-03	0.005580
2000-01-04	0.008717
2000-01-05	0.013754

```
In [ ]: # Resample, interpolate and inspect ozone data here  
data = data.resample('D').interpolate()  
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 6300 entries, 2000-01-01 to 2017-03-31  
Freq: D  
Data columns (total 1 columns):  
 #   Column  Non-Null Count  Dtype    
---  --     --     --  
 0   Ozone   6300 non-null   float64  
dtypes: float64(1)  
memory usage: 98.4 KB  
None
```

```
In [ ]: # Create the rolling window  
rolling = data.Ozone.rolling(360)  
  
# Insert the rolling quantiles to the monthly returns  
data['q10'] = rolling.quantile(0.1)  
data['q50'] = rolling.quantile(0.5)  
data['q90'] = rolling.quantile(0.9)  
  
# Plot the data  
data.plot()  
plt.show();
```



## Expanding window functions with pandas

### Cumulative sum vs .diff()

Expanding windows allow you to run cumulative calculations.

The cumulative sum method has in fact the opposite effect of the `.diff()` method that you came across earlier. To illustrate this, let's use the Google stock price time series, create the differences between prices, and reconstruct the series using the cumulative sum.

In [ ]:

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

# Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data\\\\Google Stock Prices.csv')
df.head()
```

**Close**

**Date**

Date	Close
2014-01-02	556.00
2014-01-03	551.95
2014-01-06	558.10
2014-01-07	568.86
2014-01-08	570.04

In [ ]:

```
# Calculate differences
differences = df.diff().dropna()
differences.head()
```

**Close**

**Date**

Date	Close
2014-01-03	-4.05
2014-01-06	6.15
2014-01-07	10.76
2014-01-08	1.18
2014-01-09	-5.49

In [ ]:

```
# Select start price
start_price = df.first('D')
start_price
```

C:\Users\yeiso\AppData\Local\Temp\ipykernel\_38792\863676461.py:2: FutureWarning: first is deprecated and will be removed in a future version. Please create a mask and filter using `.loc` instead
start\_price = df.first('D')

**Close**

**Date**

Date	Close
2014-01-02	556.0

In [ ]:

```
#este era el codigo original pero no funciono
#cumulative_sum = start_price.append(differences).cumsum()
#cumulative_sum.head()
```

```
#en vez de eso... use este!
# Calculate cumulative sum
cumulative_sum = pd.concat([start_price, differences]).cumsum()
cumulative_sum.head()
```

**Close**

**Date**

<b>2014-01-02</b>	556.00
<b>2014-01-03</b>	551.95
<b>2014-01-06</b>	558.10
<b>2014-01-07</b>	568.86
<b>2014-01-08</b>	570.04

```
In [ ]: # Validate cumulative sum equals data
print(df.equals(cumulative_sum))
```

True

### Cumulative return on \$1,000 invested in google vs apple I

To put your new ability to do cumulative return calculations to practical use, let's compare how much \$1,000 would be worth if invested in Google ('GOOG') or Apple ('AAPL') in 2010.

```
In [ ]: # Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data
df.head()
```

**AAPL GOOG**

**Date**

<b>2010-01-04</b>	NaN	313.06
<b>2010-01-05</b>	NaN	311.68
<b>2010-01-06</b>	NaN	303.83
<b>2010-01-07</b>	NaN	296.75
<b>2010-01-08</b>	NaN	300.71

```
In [ ]: # Define your investment
investment = 1000
```

```
In [ ]: # Calculate the daily returns here  
returns = df.pct_change()  
returns
```

	AAPL	GOOG
Date		
2010-01-04	NaN	NaN
2010-01-05	NaN	-0.004408
2010-01-06	NaN	-0.025186
2010-01-07	NaN	-0.023303
2010-01-08	NaN	0.013345
...	...	...
2017-05-24	-0.002991	0.006471
2017-05-25	0.003456	0.015268
2017-05-26	-0.001690	0.001991
2017-05-30	0.000391	0.004540
2017-05-31	-0.005922	-0.011292

1864 rows × 2 columns

```
In [ ]: # Calculate the cumulative returns here  
returns_plus_one = returns + 1  
cumulative_return = returns_plus_one.cumprod()
```

```
In [ ]: # Calculate and plot the investment return here  
cumulative_return.mul(investment).plot()  
plt.show();
```



### Cumulative return on \$1,000 invested in google vs apple II

Apple outperformed Google over the entire period, but this may have been different over various 1-year sub periods, so that switching between the two stocks might have yielded an even better result.

To analyze this, calculate that cumulative return for rolling 1-year periods, and then plot the returns to see when each stock was superior.

```
In [ ]: # Import numpy  
import numpy as np
```

```
In [ ]: # Define a multi_period_return function  
def multi_period_return(period_returns):  
    return np.prod(period_returns + 1) - 1
```

```
In [ ]: # Import data here  
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat  
df.head()
```

**AAPL GOOG**

**Date**

<b>2010-01-04</b>	NaN	313.06
<b>2010-01-05</b>	NaN	311.68
<b>2010-01-06</b>	NaN	303.83
<b>2010-01-07</b>	NaN	296.75
<b>2010-01-08</b>	NaN	300.71

In [ ]:

```
# Calculate daily returns
daily_returns = df.pct_change()
daily_returns.head()
```

**AAPL GOOG**

**Date**

<b>2010-01-04</b>	NaN	NaN
<b>2010-01-05</b>	NaN	-0.004408
<b>2010-01-06</b>	NaN	-0.025186
<b>2010-01-07</b>	NaN	-0.023303
<b>2010-01-08</b>	NaN	0.013345

In [ ]:

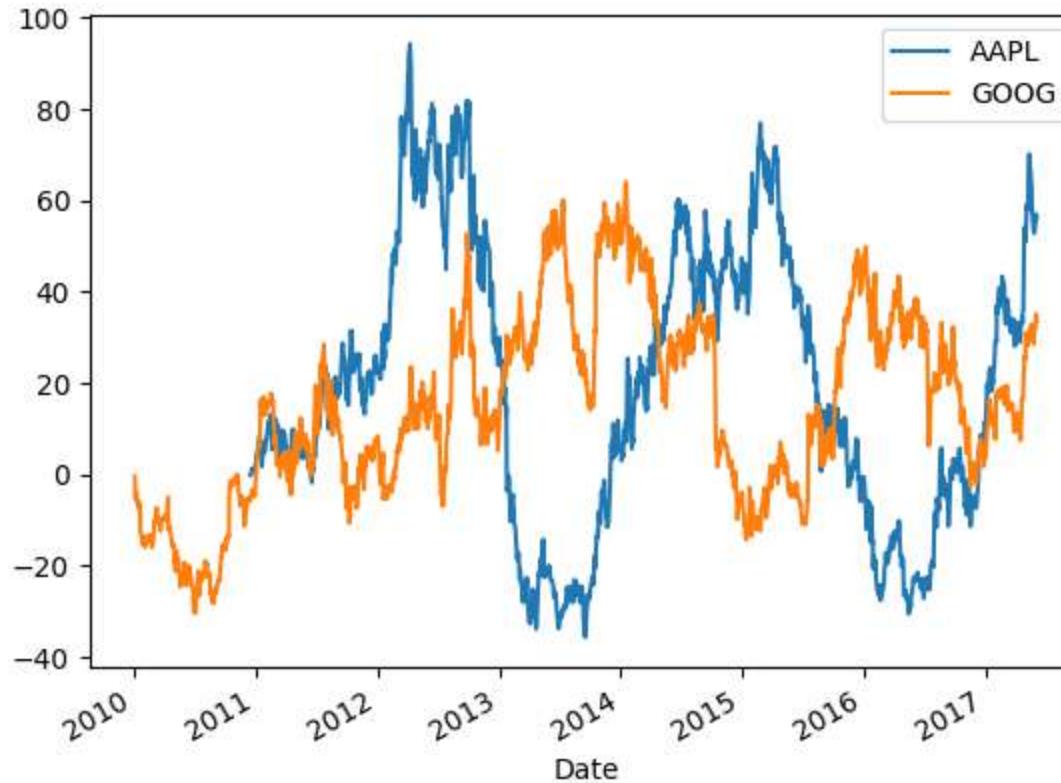
```
# Calculate rolling annual returns
rolling_annual_returns = daily_returns.rolling('360D').apply(multi_period_return)
rolling_annual_returns.tail()
```

**AAPL GOOG**

**Date**

<b>2017-05-24</b>	0.528052	0.303415
<b>2017-05-25</b>	0.533333	0.323315
<b>2017-05-26</b>	0.538254	0.320434
<b>2017-05-30</b>	0.569342	0.350998
<b>2017-05-31</b>	0.560049	0.335742

```
In [ ]: # Plot rolling annual returns  
rolling_annual_returns.mul(100).plot()  
plt.show();
```



### Case study: S&P500 price simulation

In this exercise, you'll build your own random walk by drawing random numbers from the normal distribution with the help of numpy.

**Random walk I** You'll build your own random walk by drawing random numbers from the normal distribution with the help of numpy.

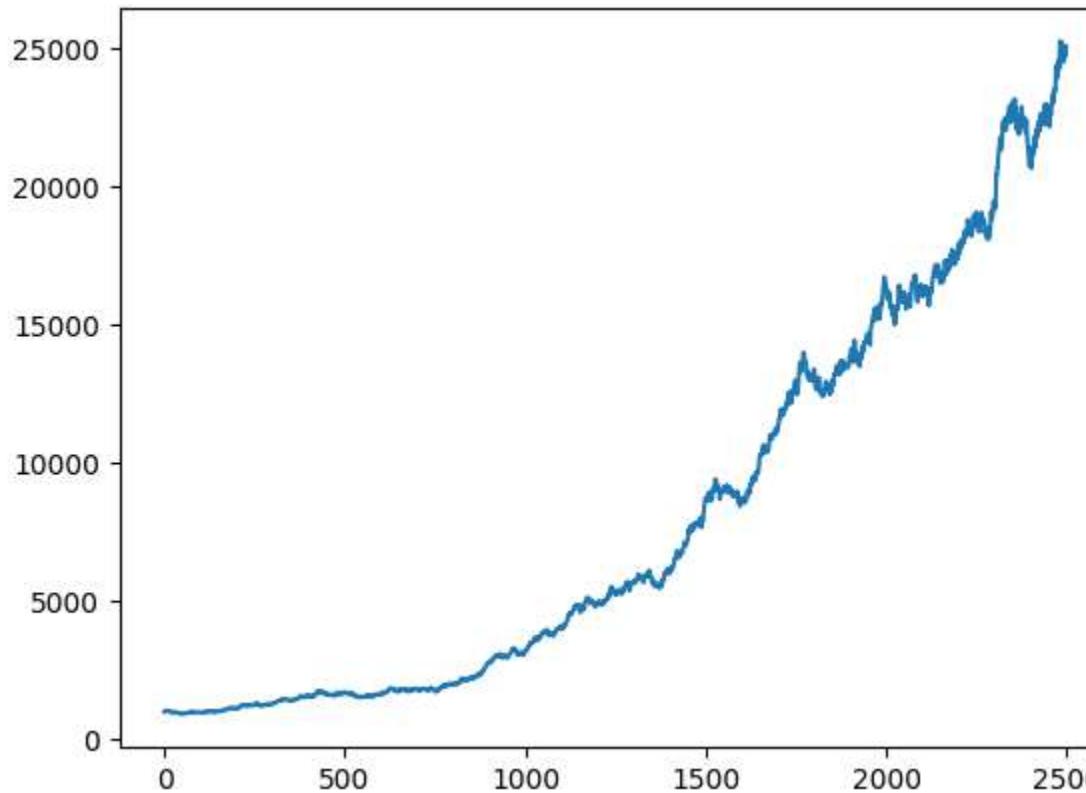
```
In [ ]: from numpy.random import normal, seed  
from scipy.stats import norm
```

```
In [ ]: # Set seed here  
seed = 42  
np.random.seed(seed)  
  
# Create random_walk  
# Use normal to generate 2,500 random returns with the parameters  
# loc=.001, scale=.01 and assign this to random_walk.  
random_walk = normal(loc=.001, scale=0.01, size=2500)
```

```
# Convert random_walk to pd.Series
random_walk = pd.Series(random_walk)

# Create random_prices
# Create random_prices by adding 1 to random_walk
# and calculating the cumulative product.
random_prices = random_walk.add(1).cumprod()

# Plot random_prices here
# Multiply random_prices by 1,000
# and plot the result for a price series starting at 1,000.
random_prices.mul(1000).plot()
plt.show();
```



## Random walk II

You'll build a random walk using historical returns from Facebook's stock price since IPO through the end of May 31, 2017. Then you'll simulate an alternative random price path in the next exercise.

In [ ]:

```
# Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data'
df.index.name = 'date'
```

```
df.columns = ['price']
df.head()
```

price

date	price
2012-05-17	38.00
2012-05-18	38.23
2012-05-21	34.03
2012-05-22	31.00
2012-05-23	32.00

```
In [ ]: fb = df['price']
print(type(fb))
fb
```

```
<class 'pandas.core.series.Series'>
date
2012-05-17    38.00
2012-05-18    38.23
2012-05-21    34.03
2012-05-22    31.00
2012-05-23    32.00
...
2017-05-24    150.04
2017-05-25    151.96
2017-05-26    152.13
2017-05-30    152.38
2017-05-31    151.46
Name: price, Length: 1267, dtype: float64
```

```
In [ ]: import pandas as pd
from numpy.random import choice, random
import matplotlib.pyplot as plt
import seaborn as sns

# Set seed here
#sometimes using >>> seed(42) <<< could works!
seed = 42
np.random.seed(seed)
```

```
In [ ]: # Calculate daily_returns here
daily_returns = fb.pct_change().dropna()
print(daily_returns)
```

```
date
2012-05-18    0.006053
2012-05-21   -0.109861
2012-05-22   -0.089039
2012-05-23    0.032258
2012-05-24    0.032188
...
2017-05-24    0.013305
2017-05-25    0.012797
2017-05-26    0.001119
2017-05-30    0.001643
2017-05-31   -0.006038
Name: price, Length: 1266, dtype: float64
```

```
In [ ]: # Get n_obs
n_obs = daily_returns.count()
print(n_obs)
```

```
1266
```

```
In [ ]: # Create random_walk
random_walk = choice(daily_returns, size=n_obs)
random_walk
```

```
array([-0.00637783, -0.00854701,  0.00833254, ..., -0.00832266,
       -0.00044709, -0.00940827])
```

```
In [ ]: # Convert random_walk to pd.Series
random_walk = pd.Series(random_walk)
```

```
In [ ]: # Plot random_walk distribution
sns.distplot(random_walk)
plt.show();
```

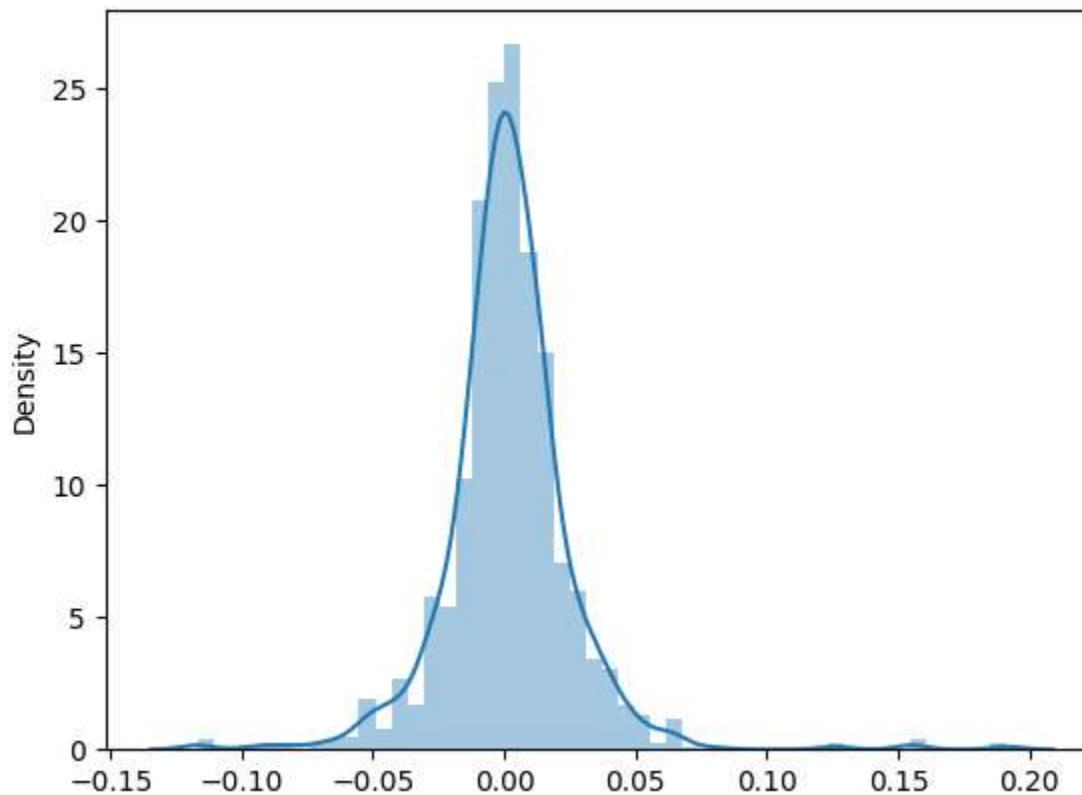
```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_38792\3014660000.py:2: UserWarning:
```

```
`distplot` is a deprecated function and will be removed in seaborn v0.14.0.
```

```
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```

```
For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751
```

```
sns.distplot(random_walk)
```



### Random walk III

In this exercise, you'll complete your random walk simulation using Facebook stock returns over the last five years. You'll start off with a random sample of returns like the one you've generated during the last exercise and use it to create a random stock price path.

```
In [ ]: # Select fb start price here
start = fb.price.first('D')

# Add 1 to random walk and append to start
random_walk = random_walk.add(1)
random_price = start.append(random_walk)

# Calculate cumulative product here
random_price = random_price.cumprod()

# Insert into fb and plot
fb['random'] = random_price

fb.plot()
plt.show()
```

You have seen in the video how to calculate correlations, and visualize the result.

In this exercise, we have provided you with the historical stock prices for Apple (AAPL), Amazon (AMZN), IBM (IBM), WalMart (WMT), and Exxon Mobile (XOM) for the last 4,000 trading days from July 2001 until the end of May 2017.

You'll calculate the year-end returns, the pairwise correlations among all stocks, and visualize the result as an annotated heatmap.

```
In [ ]: # Import data here
df = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
df.head()
```

	AAPL	AMZN	IBM	WMT	XOM
Date					
2001-07-05	1.66	15.27	NaN	NaN	NaN
2001-07-06	1.57	15.27	106.50	47.34	43.40
2001-07-09	1.62	15.81	104.72	48.25	43.36
2001-07-10	1.51	15.61	101.96	47.50	42.88
2001-07-11	1.61	15.34	103.85	48.85	42.48

```
In [ ]: data = df
# Inspect data here
print(data.info())

# Calculate year-end prices here
annual_prices = data.resample('A').last()

# Calculate annual returns here
annual_returns = annual_prices.pct_change()

# Calculate and print the correlation matrix here
correlations = annual_returns.corr()
print(correlations)

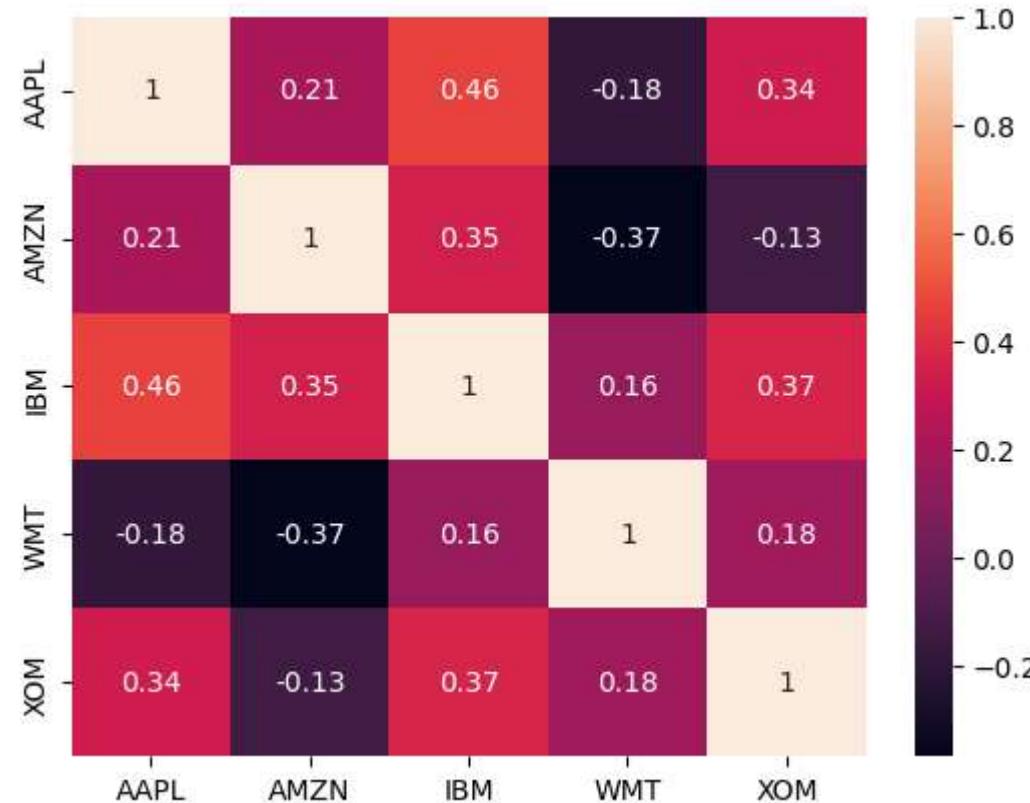
# Visualize the correlations as heatmap here
sns.heatmap(correlations, annot=True)
plt.show();
```

```

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4001 entries, 2001-07-05 to 2017-05-31
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   AAPL    4000 non-null   float64
 1   AMZN    4000 non-null   float64
 2   IBM     4000 non-null   float64
 3   WMT    4000 non-null   float64
 4   XOM    4000 non-null   float64
dtypes: float64(5)
memory usage: 187.5 KB
None

```

	AAPL	AMZN	IBM	WMT	XOM
AAPL	1.000000	0.208731	0.460568	-0.183553	0.336413
AMZN	0.208731	1.000000	0.346407	-0.367620	-0.133965
IBM	0.460568	0.346407	1.000000	0.155445	0.367253
WMT	-0.183553	-0.367620	0.155445	1.000000	0.178833
XOM	0.336413	-0.133965	0.367253	0.178833	1.000000



## Chapter 4 - Putting it all together

```
In [ ]: #import requiered libraries
```

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

```
In [ ]: listings_nyse = pd.read_excel('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python\\\\listings_nyse.xlsx')
listings_amex = pd.read_excel('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python\\\\listings_amex.xlsx')
listings_nasdaq = pd.read_excel('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python\\\\listings_nasdaq.xlsx')
```

```
c:\\\\Users\\\\yeiso\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python312\\\\Lib\\\\site-packages\\\\openpyxl\\\\worksheet\\\\_reader.py:329: UserWarning: Unknown
extension is not supported and will be removed
    warn(msg)
c:\\\\Users\\\\yeiso\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python312\\\\Lib\\\\site-packages\\\\openpyxl\\\\worksheet\\\\_reader.py:329: UserWarning: Unknown
extension is not supported and will be removed
    warn(msg)
c:\\\\Users\\\\yeiso\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python312\\\\Lib\\\\site-packages\\\\openpyxl\\\\worksheet\\\\_reader.py:329: UserWarning: Unknown
extension is not supported and will be removed
    warn(msg)
```

```
In [ ]: listings_nyse['Exchange'] = 'nyse'
listings_amex['Exchange'] = 'amex'
listings_nasdaq['Exchange'] = 'nasdaq'
```

```
In [ ]: listings = pd.concat([listings_amex, listings_nasdaq, listings_nyse], axis=0)
listings.reset_index(inplace=True)
listings.drop(['index'], axis=1, inplace=True)
listings['Market Capitalization'] /= 1e6
```

```
In [ ]: print(listings.info())

# Move 'stock symbol' into the index
listings.set_index('Stock Symbol', inplace=True)

# Drop rows with missing 'sector' data
listings.dropna(subset=['Sector'], inplace=True)

# Select companies with IPO Year before 2019
listings = listings[listings['IPO Year'] < 2019]

# Inspect the new listings data
print(listings.info())

# Show the number of companies per sector
print(listings.groupby('Sector').size().sort_values(ascending=False))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9441 entries, 0 to 9440
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Stock Symbol     9441 non-null   object  
 1   Company Name     9441 non-null   object  
 2   Last Sale        9237 non-null   float64 
 3   Market Capitalization 9441 non-null   float64 
 4   IPO Year         4083 non-null   float64 
 5   Sector            6531 non-null   object  
 6   Industry          6531 non-null   object  
 7   Exchange          9441 non-null   object  
dtypes: float64(3), object(5)
memory usage: 590.2+ KB
```

None

```
<class 'pandas.core.frame.DataFrame'>
Index: 2901 entries, WBAI to ZTO
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Company Name     2901 non-null   object  
 1   Last Sale        2901 non-null   float64 
 2   Market Capitalization 2901 non-null   float64 
 3   IPO Year         2901 non-null   float64 
 4   Sector            2901 non-null   object  
 5   Industry          2901 non-null   object  
 6   Exchange          2901 non-null   object  
dtypes: float64(3), object(4)
memory usage: 181.3+ KB
```

None

Sector

Consumer Services	720
Finance	396
Energy	339
Technology	324
Public Utilities	228
Basic Industries	198
Capital Goods	183
Consumer Non-Durables	141
Health Care	129
Transportation	111
Miscellaneous	75
Consumer Durables	57

```
dtype: int64
```

## Select and inspect index components

Now that you have imported and cleaned the listings data, you can proceed to select the index components as the largest company for each sector by market capitalization.

You'll also have the opportunity to take a closer look at the components, their last market value, and last price.

```
In [ ]: components = listings.groupby('Sector')['Market Capitalization'].nlargest(1)
```

```
# Print components, sorted by market cap
print(components.sort_values(ascending=False))
```

```
# Select stock symbols and print the result
tickers = components.index.get_level_values('Stock Symbol')
print(tickers)
```

```
# Print company name, market cap, and last price for each components
info_cols = ['Company Name', 'Market Capitalization', 'Last Sale']
print(listings.loc[tickers,info_cols].sort_values('Market Capitalization', ascending=False))
```

Sector	Stock Symbol	Market Capitalization
Miscellaneous	BABA	275525.000000
Technology	ORCL	181046.096000
Health Care	ABBV	102196.076208
Transportation	UPS	90180.886756
Finance	GS	88840.590477
Consumer Non-Durables	ABEV	88240.198455
Basic Industries	RIO	70431.476895
Public Utilities	TEF	54609.806092
Capital Goods	GM	50086.335099
Consumer Services	LVS	44384.295569
Energy	PAA	22223.001416
Consumer Durables	WRK	12354.903312

Name: Market Capitalization, dtype: float64

```
Index(['RIO', 'GM', 'WRK', 'ABEV', 'LVS', 'PAA', 'GS', 'ABBV', 'BABA', 'TEF',
       'ORCL', 'UPS'],
      dtype='object', name='Stock Symbol')
```

Company Name    Market Capitalization \

Stock Symbol

BABA	Alibaba Group Holding Limited	275525.000000
BABA	Alibaba Group Holding Limited	275525.000000
BABA	Alibaba Group Holding Limited	275525.000000
ORCL	Oracle Corporation	181046.096000
ORCL	Oracle Corporation	181046.096000
ORCL	Oracle Corporation	181046.096000
ABBV	AbbVie Inc.	102196.076208
ABBV	AbbVie Inc.	102196.076208
ABBV	AbbVie Inc.	102196.076208
UPS	United Parcel Service, Inc.	90180.886756
UPS	United Parcel Service, Inc.	90180.886756
UPS	United Parcel Service, Inc.	90180.886756
GS	Goldman Sachs Group, Inc. (The)	88840.590477
GS	Goldman Sachs Group, Inc. (The)	88840.590477
GS	Goldman Sachs Group, Inc. (The)	88840.590477
ABEV	Ambev S.A.	88240.198455
ABEV	Ambev S.A.	88240.198455
ABEV	Ambev S.A.	88240.198455
RIO	Rio Tinto Plc	70431.476895
RIO	Rio Tinto Plc	70431.476895
RIO	Rio Tinto Plc	70431.476895
TEF	Telefonica SA	54609.806092
TEF	Telefonica SA	54609.806092
TEF	Telefonica SA	54609.806092
GM	General Motors Company	50086.335099
GM	General Motors Company	50086.335099
GM	General Motors Company	50086.335099
LVS	Las Vegas Sands Corp.	44384.295569
LVS	Las Vegas Sands Corp.	44384.295569
LVS	Las Vegas Sands Corp.	44384.295569
PAA	Plains All American Pipeline, L.P.	22223.001416

PAA	Plains All American Pipeline, L.P.	22223.001416
PAA	Plains All American Pipeline, L.P.	22223.001416
WRK	Westrock Company	12354.903312
WRK	Westrock Company	12354.903312
WRK	Westrock Company	12354.903312

#### Last Sale

Stock Symbol	
BABA	110.21
BABA	110.21
BABA	110.21
ORCL	44.00
ORCL	44.00
ORCL	44.00
ABBV	64.13
ABBV	64.13
ABBV	64.13
UPS	103.74
UPS	103.74
UPS	103.74
GS	223.32
GS	223.32
GS	223.32
ABEV	5.62
ABEV	5.62
ABEV	5.62
RIO	38.94
RIO	38.94
RIO	38.94
TEF	10.84
TEF	10.84
TEF	10.84
GM	33.39
GM	33.39
GM	33.39
LVS	55.90
LVS	55.90
LVS	55.90
PAA	30.72
PAA	30.72
PAA	30.72
WRK	49.34
WRK	49.34
WRK	49.34

#### Import index component price information

Now you'll use the stock symbols for the companies you selected in the last exercise to calculate returns for each company.

```
In [ ]: tickers = tickers.tolist()

In [ ]: print(tickers)

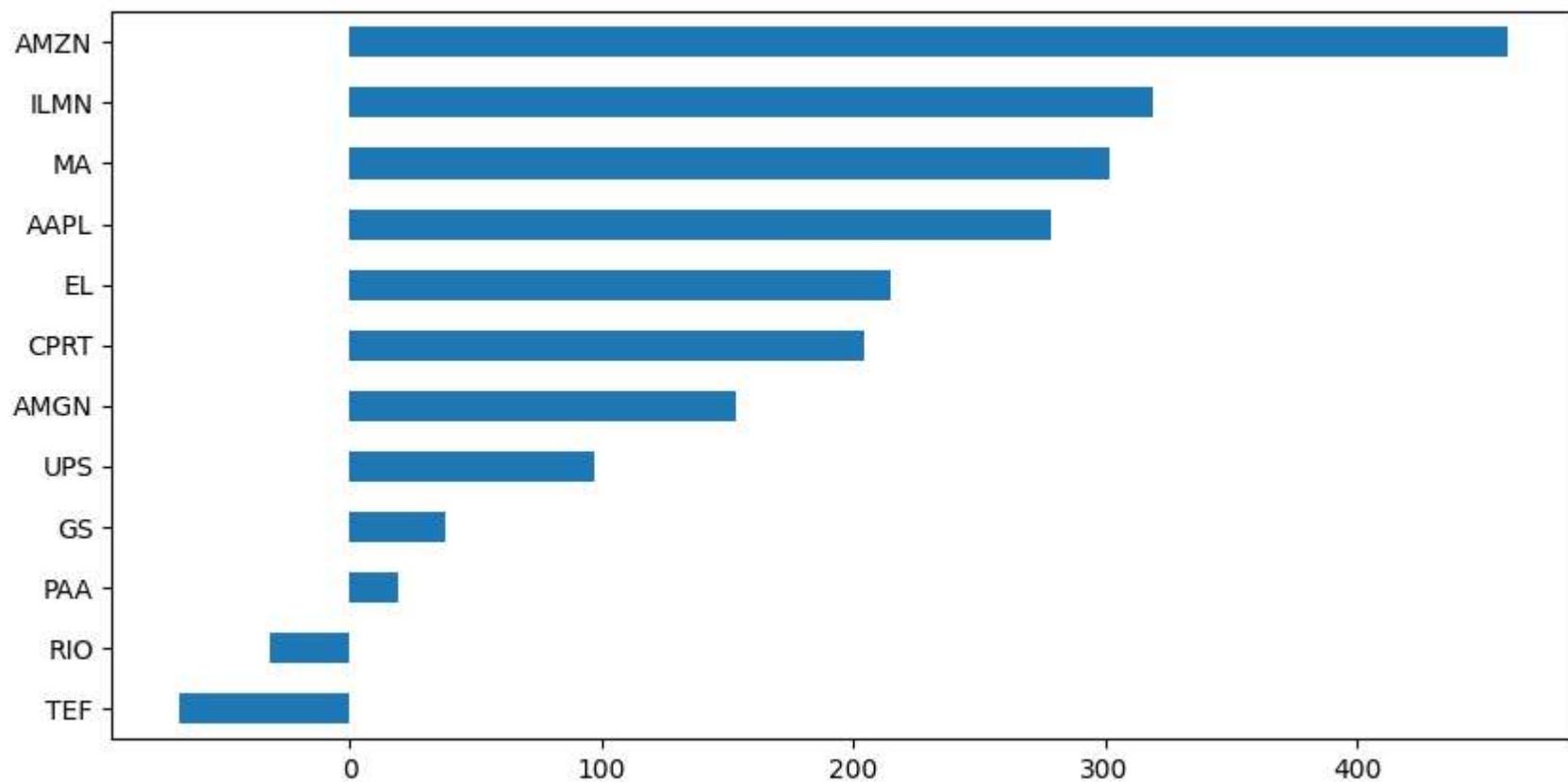
# Import prices and inspect result
stock_prices = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python
print(stock_prices.info())

# Calculate the returns
price_return = stock_prices.iloc[-1].div(stock_prices.iloc[0]).sub(1).mul(100)

# Plot horizontal bar chart of sorted price_return
price_return.sort_values().plot(kind='barh', title='Stock Price Returns');

['RIO', 'GM', 'WRK', 'ABEV', 'LVS', 'PAA', 'GS', 'ABBV', 'BABA', 'TEF', 'ORCL', 'UPS']
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   AAPL    1761 non-null   float64
 1   AMGN    1761 non-null   float64
 2   AMZN    1761 non-null   float64
 3   CPRT    1761 non-null   float64
 4   EL      1762 non-null   float64
 5   GS      1762 non-null   float64
 6   ILMN    1761 non-null   float64
 7   MA      1762 non-null   float64
 8   PAA     1762 non-null   float64
 9   RIO     1762 non-null   float64
 10  TEF     1762 non-null   float64
 11  UPS     1762 non-null   float64
dtypes: float64(12)
memory usage: 179.0 KB
None
```

## Stock Price Returns



### Build a market-cap weighted index

Calculate number of shares outstanding The next step towards building a value-weighted index is to calculate the number of shares for each index component.

The number of shares will allow you to calculate the total market capitalization for each component given the historical price series in the next exercise.

In [ ]:

```
print(listings.info())
print(tickers)

# Select components and relevant columns from Listings
components = listings[['Market Capitalization', 'Last Sale']].loc[tickers]

# Print the first rows of components
print(components.head(5))

# Calculate the number of shares here
no_shares = components['Market Capitalization'].div(components['Last Sale'])
```

```
# Print the sorted no_shares
print(no_shares.sort_values(ascending=False))
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2901 entries, WBAI to ZTO
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Company Name     2901 non-null    object  
 1   Last Sale        2901 non-null    float64 
 2   Market Capitalization 2901 non-null    float64 
 3   IPO Year         2901 non-null    float64 
 4   Sector            2901 non-null    object  
 5   Industry          2901 non-null    object  
 6   Exchange          2901 non-null    object  
dtypes: float64(3), object(4)
memory usage: 245.9+ KB
None
['RIO', 'GM', 'WRK', 'ABEV', 'LVS', 'PAA', 'GS', 'ABBV', 'BABA', 'TEF', 'ORCL', 'UPS']
Market Capitalization  Last Sale
Stock Symbol
RIO                  70431.476895  38.94
RIO                  70431.476895  38.94
RIO                  70431.476895  38.94
GM                   50086.335099  33.39
GM                   50086.335099  33.39
Stock Symbol
ABEV    15701.102928
ABEV    15701.102928
ABEV    15701.102928
TEF     5037.804990
TEF     5037.804990
TEF     5037.804990
ORCL   4114.684000
ORCL   4114.684000
ORCL   4114.684000
BABA   2500.000000
BABA   2500.000000
BABA   2500.000000
RIO    1808.717948
RIO    1808.717948
RIO    1808.717948
ABBV   1593.576738
ABBV   1593.576738
ABBV   1593.576738
GM     1500.039985
GM     1500.039985
GM     1500.039985
UPS    869.297154
UPS    869.297154
UPS    869.297154
LVS    793.994554
LVS    793.994554
```

```
LVS    793.994554  
PAA    723.404994  
PAA    723.404994  
PAA    723.404994  
GS     397.817439  
GS     397.817439  
GS     397.817439  
WRK   250.403391  
WRK   250.403391  
WRK   250.403391  
dtype: float64
```

### Create time series of market value

You can now use the number of shares to calculate the total market capitalization for each component and trading date from the historical price series.

The result will be the key input to construct the value-weighted stock index, which you will complete in the next exercise.

```
In [ ]: components['Number of Shares'] = no_shares  
print(no_shares.sort_values())
```

Stock Symbol

```
WRK      250.403391
WRK      250.403391
WRK      250.403391
GS       397.817439
GS       397.817439
GS       397.817439
PAA      723.404994
PAA      723.404994
PAA      723.404994
LVS      793.994554
LVS      793.994554
LVS      793.994554
UPS      869.297154
UPS      869.297154
UPS      869.297154
GM       1500.039985
GM       1500.039985
GM       1500.039985
ABBV     1593.576738
ABBV     1593.576738
ABBV     1593.576738
RIO      1808.717948
RIO      1808.717948
RIO      1808.717948
BABA     2500.000000
BABA     2500.000000
BABA     2500.000000
ORCL     4114.684000
ORCL     4114.684000
ORCL     4114.684000
TEF      5037.804990
TEF      5037.804990
TEF      5037.804990
ABEV     15701.102928
ABEV     15701.102928
ABEV     15701.102928
dtype: float64
```

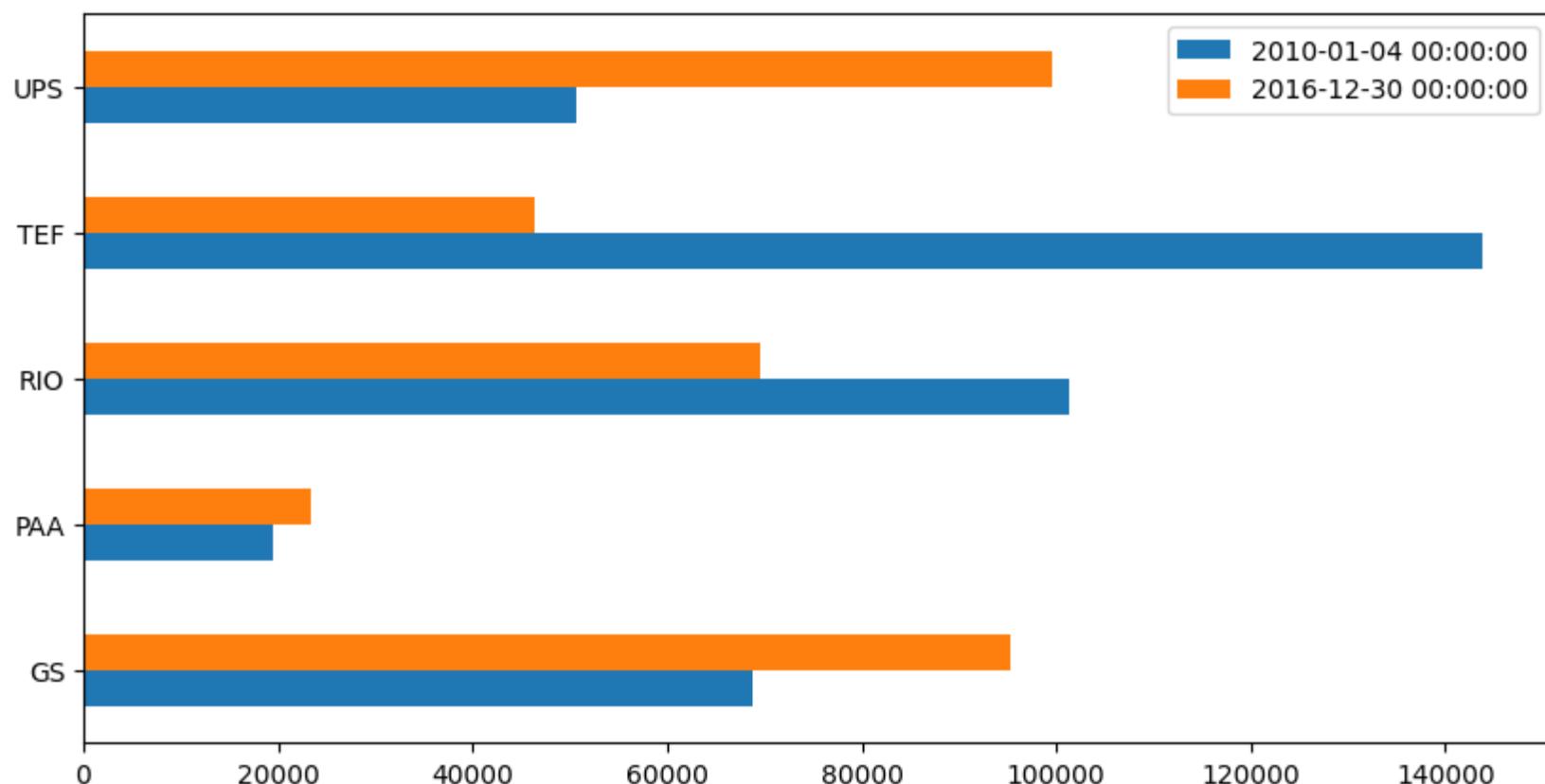
In [ ]: no\_shares\_no\_duplicates = no\_shares.drop\_duplicates()  
print(no\_shares\_no\_duplicates.sort\_values())

```
Stock Symbol
WRK      250.403391
GS       397.817439
PAA      723.404994
LVS      793.994554
UPS      869.297154
GM       1500.039985
ABBV     1593.576738
RIO      1808.717948
BABA     2500.000000
ORCL     4114.684000
TEF      5037.804990
ABEV     15701.102928
dtype: float64
```

```
In [ ]: # Create the series of market cap per ticker
market_cap = stock_prices.mul(no_shares_no_duplicates)

# Select first and last market cap here
first_value = market_cap.iloc[0]
last_value = market_cap.iloc[-1]

# Concatenate and plot first and last market cap here
pd.concat([first_value, last_value], axis=1).dropna().plot(kind='barh');
plt.savefig('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp')
plt.show()
```



### Calculate & plot the composite index

By now you have all ingredients that you need to calculate the aggregate stock performance for your group of companies.

Use the time series of market capitalization that you created in the last exercise to aggregate the market value for each period, and then normalize this series to convert it to an index.

```
In [ ]: market_cap_series = market_cap[pd.concat([first_value, last_value], axis=1).dropna().index.tolist()]
```

```
In [ ]: market_cap_series
```

Out[ ]:

GS PAA RIO TEF UPS

Date

Date	GS	PAA	RIO	TEF	UPS
2010-01-04	68854.242342	19531.934838	101342.466626	143829.332464	50575.708420
2010-01-05	70071.563705	19748.956336	102916.051241	143728.576365	50662.638135
2010-01-06	69323.666920	19741.722286	106063.220471	142217.234868	50288.840359
2010-01-07	70680.224387	19502.998638	106081.307650	139799.088472	49906.349611
2010-01-08	69343.557792	19568.105088	107256.974316	138892.283574	52305.609756
...	...	...	...	...	...
2016-12-23	95862.068276	24168.960850	68948.328178	46196.671758	100812.390949
2016-12-27	96096.780565	24226.833249	69364.333306	45995.159559	100951.478494
2016-12-28	95734.766695	23908.535052	70304.866639	45491.379060	100143.032141
2016-12-29	94752.157621	23488.960155	70359.128177	45995.159559	99951.786767
2016-12-30	95257.385769	23358.747256	69563.292280	46347.805908	99656.225735

1762 rows × 5 columns

In [ ]:

```
# Aggregate and print the market cap per trading day
raw_index = market_cap_series.sum(axis=1)
print(raw_index)

# Normalize the aggregate market cap here
index = raw_index.div(raw_index.iloc[0]).mul(100)
print(index)

# Plot the index here
index.plot(title='Market-Cap Weighted Index')
plt.show();
```

```
Date  
2010-01-04    384133.684691  
2010-01-05    387127.785783  
2010-01-06    387634.684904  
2010-01-07    385969.968759  
2010-01-08    387366.530527
```

```
...
```

```
2016-12-23    335988.420011  
2016-12-27    336634.585172  
2016-12-28    335582.579586  
2016-12-29    334547.192279  
2016-12-30    334183.456947
```

```
Length: 1762, dtype: float64
```

```
Date
```

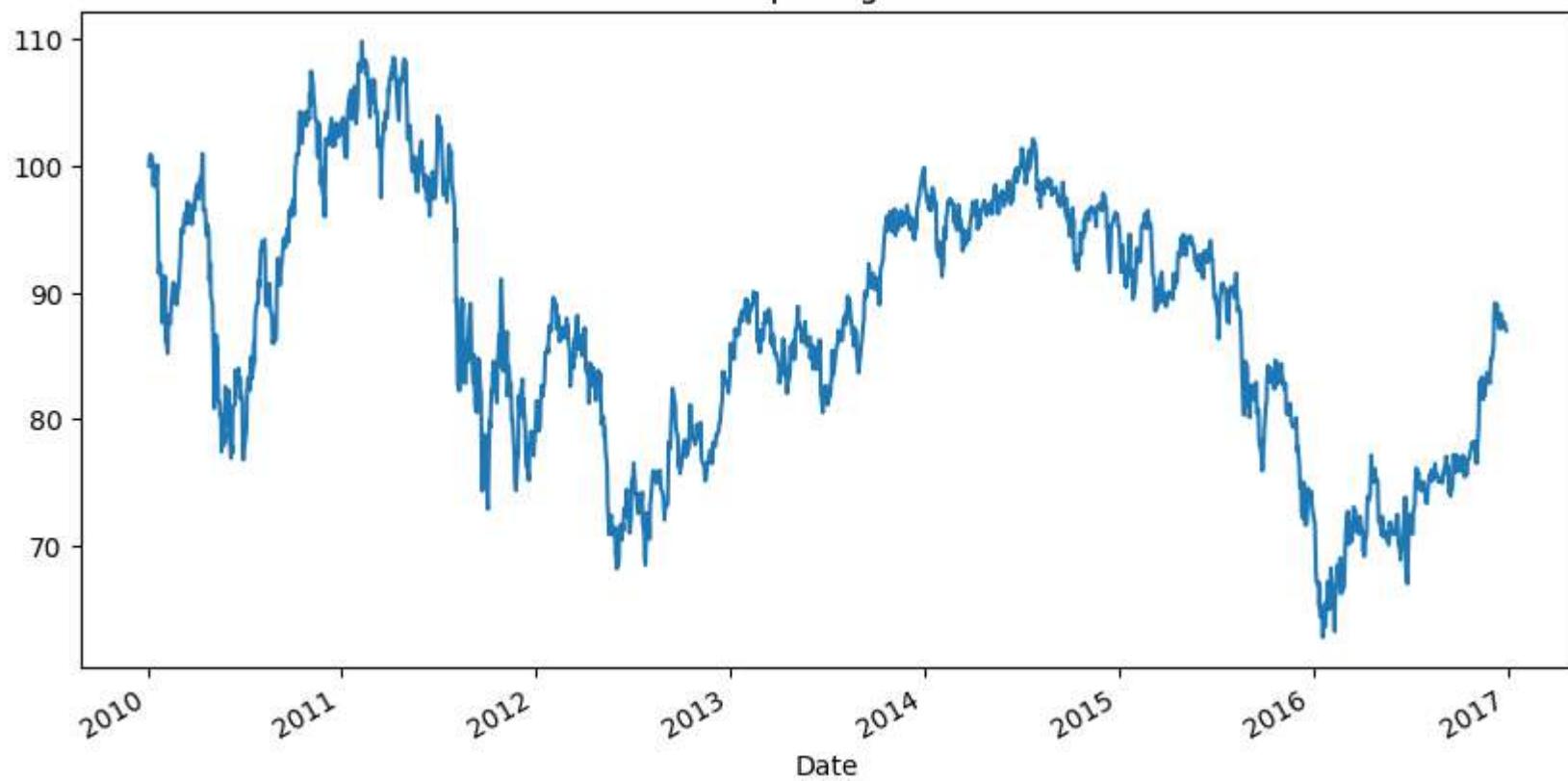
```
2010-01-04    100.000000  
2010-01-05    100.779442  
2010-01-06    100.911402  
2010-01-07    100.478033  
2010-01-08    100.841594
```

```
...
```

```
2016-12-23    87.466534  
2016-12-27    87.634748  
2016-12-28    87.360883  
2016-12-29    87.091345  
2016-12-30    86.996655
```

```
Length: 1762, dtype: float64
```

## Market-Cap Weighted Index



### Evaluate index performance

- Index return:
  - Total index return
  - Contribution by component
- Performance vs Benchmark -Total period return
  - Rolling returns for sub periods

### Calculate the contribution of each stock to the index

You have successfully built the value-weighted index. Let's now explore how it performed over the 2010-2016 period.

Let's also determine how much each stock has contributed to the index return.

```
In [ ]: # Calculate and print the index return here
index_return = (index.iloc[-1] / index.iloc[0] - 1) * 100
print(index_return)
```

```
# Select the market capitalization
market_cap = components['Market Capitalization']

# Calculate the total market cap
total_market_cap = market_cap.sum()

# Calculate the component weights , and print the result
weights = market_cap.div(total_market_cap)
print(weights.sort_values())

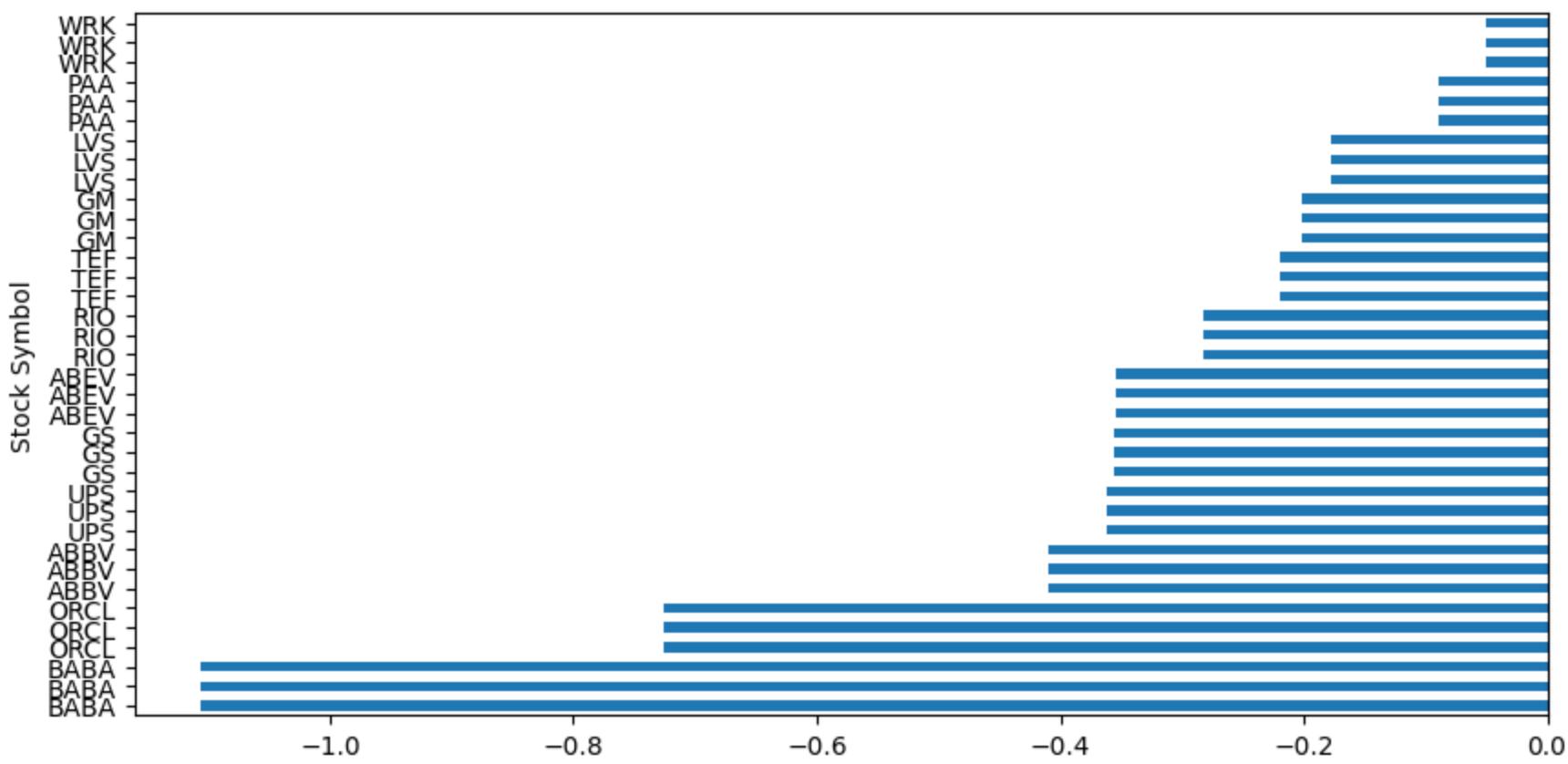
# Calculate and plot the distribution by component
weights.mul(index_return).sort_values().plot(kind='barh')
plt.show();
```

-13.003344859886202

Stock Symbol

WRK	0.003813
WRK	0.003813
WRK	0.003813
PAA	0.006858
PAA	0.006858
PAA	0.006858
LVS	0.013697
LVS	0.013697
LVS	0.013697
GM	0.015457
GM	0.015457
GM	0.015457
TEF	0.016853
TEF	0.016853
TEF	0.016853
RIO	0.021736
RIO	0.021736
RIO	0.021736
ABEV	0.027232
ABEV	0.027232
ABEV	0.027232
GS	0.027417
GS	0.027417
GS	0.027417
UPS	0.027831
UPS	0.027831
UPS	0.027831
ABBV	0.031539
ABBV	0.031539
ABBV	0.031539
ORCL	0.055872
ORCL	0.055872
ORCL	0.055872
BABA	0.085029
BABA	0.085029
BABA	0.085029

Name: Market Capitalization, dtype: float64



## Compare index performance against benchmark I

The next step in analyzing the performance of your index is to compare it against a benchmark.

In the video, we used the S&P 500 as benchmark. You can also use the Dow Jones Industrial Average, which contains the 30 largest stocks, and would also be a reasonable benchmark for the largest stocks from all sectors across the three exchanges.

```
In [ ]: djia = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course D
```

```
In [ ]: # Convert index series to dataframe here
data = index.to_frame('Index')

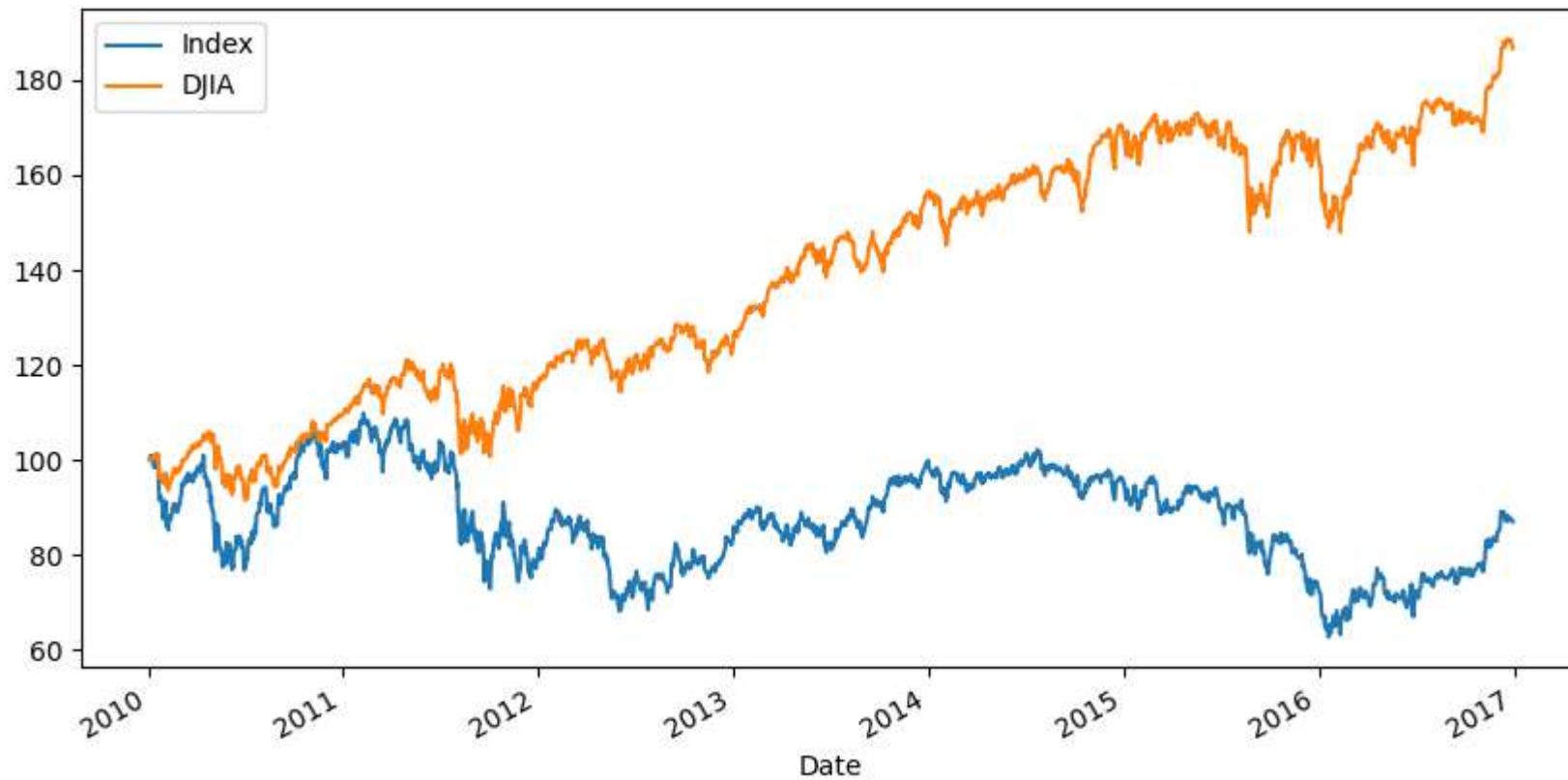
# Normalize djia series and add as new column to data
djia = djia.div(djia.iloc[0]).mul(100)
data['DJIA'] = djia

# Print total return for both index and djia
print((data.iloc[-1] / data.iloc[0] - 1) * 100)

# Plot both series
```

```
data.plot()  
plt.show();
```

```
Index   -13.003345  
DJIA    86.722172  
dtype: float64
```



## Compare index performance against benchmark II

The next step in analyzing the performance of your index is to compare it against a benchmark.

In the video, we have used the S&P 500 as benchmark. You can also use the Dow Jones Industrial Average, which contains the 30 largest stocks, and would also be a reasonable benchmark for the largest stocks from all sectors across the three exchanges.

In [ ]:

```
print(data.info())  
print(data.head(5))  
  
# Create multi_period_return function here  
def multi_period_return(r):  
    return (np.prod(r + 1) - 1) * 100  
  
# Calculate rolling_return_360  
rolling_return_360 = data.pct_change().rolling('360D').apply(multi_period_return)
```

```
# Plot rolling_return_360 here
rolling_return_360.plot(title='Rolling 360D Return');
plt.show()
```

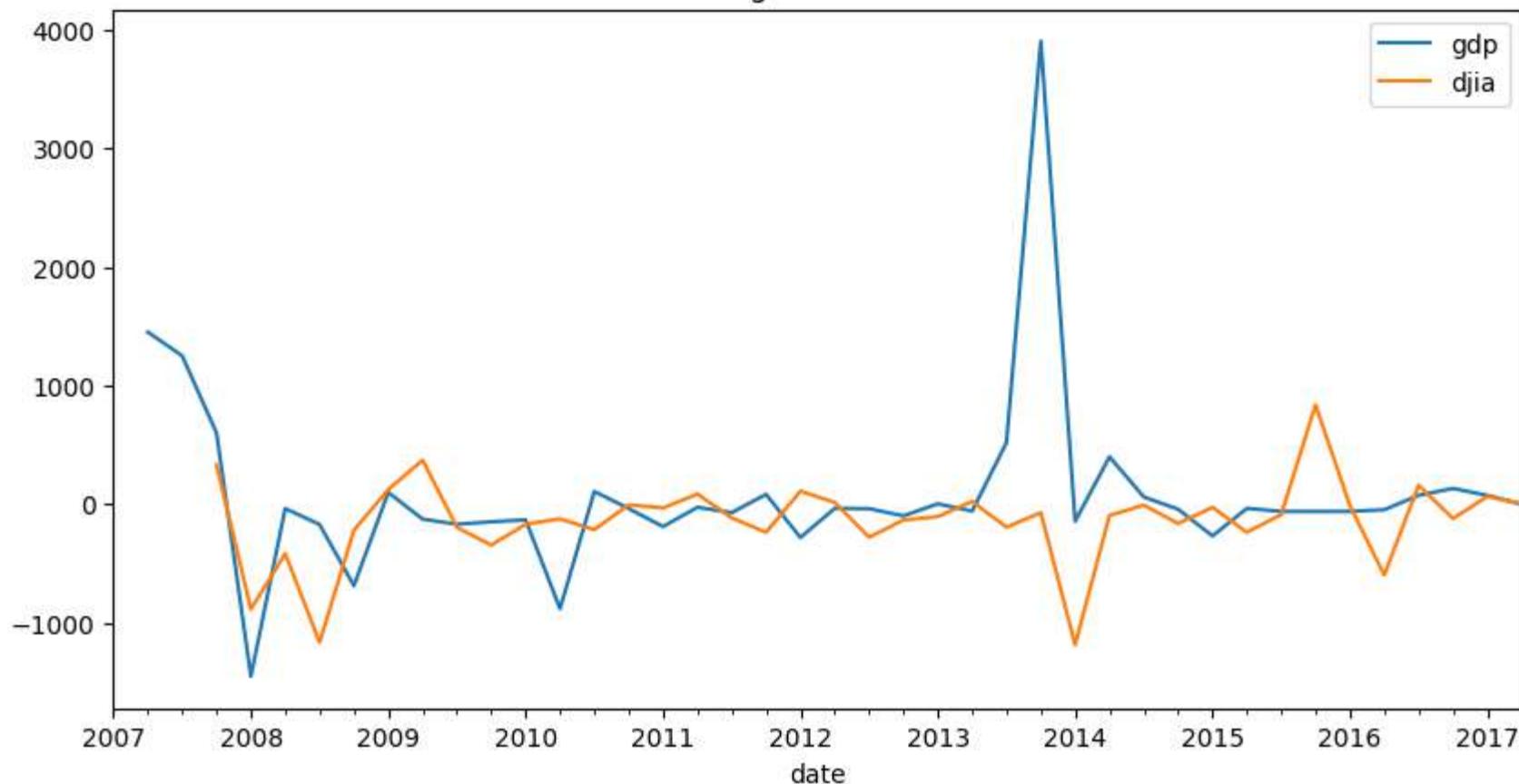
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 42 entries, 2007-01-01 to 2017-04-01
Freq: QS-OCT
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   gdp     41 non-null    float64 
 1   djia    40 non-null    float64 
dtypes: float64(2)
memory usage: 1008.0 bytes
None
```

	gdp	djia
date		
2007-01-01	0.2	NaN
2007-04-01	3.1	NaN
2007-07-01	2.7	0.945735
2007-10-01	1.4	4.079072
2008-01-01	-2.7	-7.407889

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_19956\4027336615.py:9: FutureWarning: The default fill_method='pad' in DataFrame.pct_
change is deprecated and will be removed in a future version. Call ffill before calling pct_change to retain current behavior and
silence this warning.
```

```
rolling_return_360 = data.pct_change().rolling('360D').apply(multi_period_return)
```

## Rolling 360D Return



### Index correlation & exporting to Excel

Visualize your index constituent correlations To better understand the characteristics of your index constituents, you can calculate the return correlations.

Use the daily stock prices or your index companies, and show a heatmap of the daily return correlations!

```
In [ ]: stock_prices = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python\nprint(stock_prices.head())')
```

	AAPL	AMGN	AMZN	CPRT	EL	GS	ILMN	MA	PAA	\
Date										
2010-01-04	30.57	57.72	133.90	4.55	24.27	173.08	30.55	25.68	27.00	
2010-01-05	30.63	57.22	134.69	4.55	24.18	176.14	30.35	25.61	27.30	
2010-01-06	30.14	56.79	132.25	4.53	24.25	174.26	32.22	25.56	27.29	
2010-01-07	30.08	56.27	130.00	4.50	24.56	177.67	32.77	25.39	26.96	
2010-01-08	30.28	56.77	133.52	4.52	24.66	174.31	33.15	25.40	27.05	

	RIO	TEF	UPS
Date			
2010-01-04	56.03	28.55	58.18
2010-01-05	56.90	28.53	58.28
2010-01-06	58.64	28.23	57.85
2010-01-07	58.65	27.75	57.41
2010-01-08	59.30	27.57	60.17

```
In [ ]: # Inspect stock_prices here
print(stock_prices.info())

# Calculate the daily returns
returns = stock_prices.pct_change()

# Calculate and print the pairwise correlations
correlations = returns.corr()
print(correlations)

# Plot a heatmap of daily return correlations
sns.heatmap(correlations, annot=True)
plt.title('Daily Return Correlations')
plt.show();
```

```

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   AAPL    1761 non-null   float64
 1   AMGN    1761 non-null   float64
 2   AMZN    1761 non-null   float64
 3   CPRT    1761 non-null   float64
 4   EL      1762 non-null   float64
 5   GS      1762 non-null   float64
 6   ILMN    1761 non-null   float64
 7   MA      1762 non-null   float64
 8   PAA     1762 non-null   float64
 9   RIO     1762 non-null   float64
 10  TEF     1762 non-null   float64
 11  UPS     1762 non-null   float64
dtypes: float64(12)
memory usage: 179.0 KB

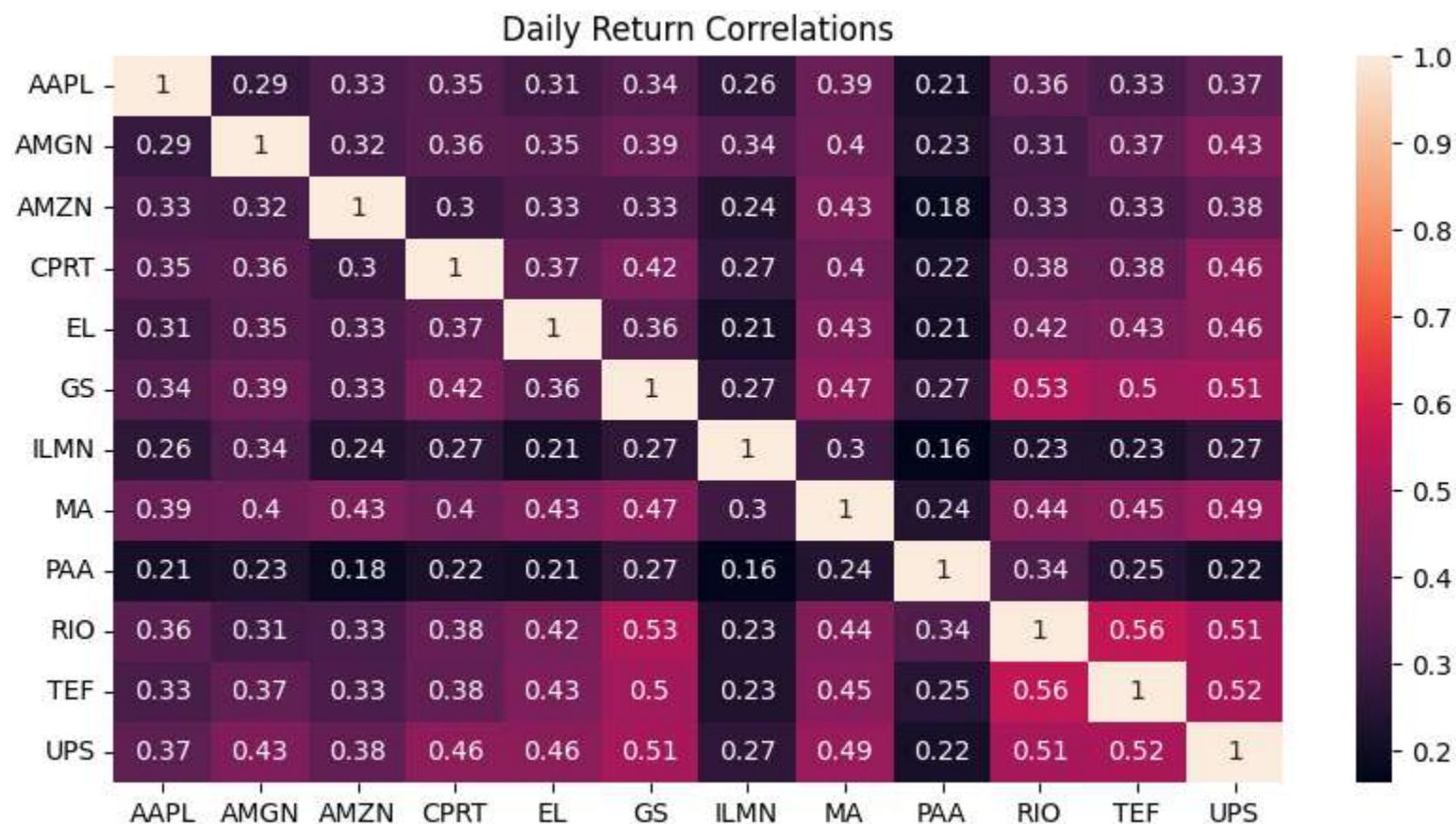
```

None

	AAPL	AMGN	AMZN	CPRT	EL	GS	ILMN	\
AAPL	1.000000	0.286898	0.327611	0.346616	0.306770	0.344981	0.264791	
AMGN	0.286898	1.000000	0.323408	0.355892	0.349893	0.390076	0.336927	
AMZN	0.327611	0.323408	1.000000	0.298929	0.334031	0.333402	0.242726	
CPRT	0.346616	0.355892	0.298929	1.000000	0.371763	0.423160	0.265665	
EL	0.306770	0.349893	0.334031	0.371763	1.000000	0.358318	0.214027	
GS	0.344981	0.390076	0.333402	0.423160	0.358318	1.000000	0.266063	
ILMN	0.264791	0.336927	0.242726	0.265665	0.214027	0.266063	1.000000	
MA	0.391421	0.400230	0.428330	0.401352	0.431556	0.466796	0.301392	
PAA	0.212960	0.229255	0.182438	0.221273	0.206056	0.271982	0.162796	
RIO	0.361684	0.313878	0.326229	0.384944	0.415416	0.527298	0.234445	
TEF	0.325309	0.374555	0.331867	0.376767	0.428925	0.498230	0.231173	
UPS	0.366039	0.432468	0.378399	0.462716	0.456952	0.506407	0.267801	

	MA	PAA	RIO	TEF	UPS
AAPL	0.391421	0.212960	0.361684	0.325309	0.366039
AMGN	0.400230	0.229255	0.313878	0.374555	0.432468
AMZN	0.428330	0.182438	0.326229	0.331867	0.378399
CPRT	0.401352	0.221273	0.384944	0.376767	0.462716
EL	0.431556	0.206056	0.415416	0.428925	0.456952
GS	0.466796	0.271982	0.527298	0.498230	0.506407
ILMN	0.301392	0.162796	0.234445	0.231173	0.267801
MA	1.000000	0.243761	0.437778	0.448438	0.486512
PAA	0.243761	1.000000	0.337448	0.253598	0.217523
RIO	0.437778	0.337448	1.000000	0.559264	0.509809
TEF	0.448438	0.253598	0.559264	1.000000	0.516242
UPS	0.486512	0.217523	0.509809	0.516242	1.000000

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_19956\2553049458.py:5: FutureWarning: The default fill_method='pad' in DataFrame.pct_change is deprecated and will be removed in a future version. Call ffill before calling pct_change to retain current behavior and silence this warning.  
    returns = stock_prices.pct_change()
```



### Save your analysis to multiple excel worksheets

Now that you have completed your analysis, you may want to save all results into a single Excel workbook.

Let's practice exporting various DataFrame to multiple Excel worksheets.

```
In [ ]: index = index.to_frame('Index')
```

```
In [ ]: print(index.info())  
print(stock_prices.info())  
  
# Join index to stock_prices, and inspect the result  
data = stock_prices.join(index)  
print(data.info())
```

```
# Create index & stock price returns
returns = data.pct_change()

#esta es la forma original.... pero NO funciono!
# Export data and data as returns to excel
#with pd.ExcelWriter('data.xls') as writer:
#    data.to_excel(writer, sheet_name='data')
#    returns.to_excel(writer, sheet_name='returns')

# Export data and data as returns to Excel with 'xls' format using 'openpyxl' engine
with pd.ExcelWriter('data.xls', engine='openpyxl') as writer:
    data.to_excel(writer, sheet_name='data', engine='openpyxl')
    returns.to_excel(writer, sheet_name='returns', engine='openpyxl')
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  --  
 0   Index    1762 non-null   float64 
dtypes: float64(1)
memory usage: 27.5 KB
None
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype  
---  --  
 0   AAPL    1761 non-null   float64 
 1   AMGN    1761 non-null   float64 
 2   AMZN    1761 non-null   float64 
 3   CPRT    1761 non-null   float64 
 4   EL      1762 non-null   float64 
 5   GS      1762 non-null   float64 
 6   ILMN    1761 non-null   float64 
 7   MA      1762 non-null   float64 
 8   PAA     1762 non-null   float64 
 9   RIO     1762 non-null   float64 
 10  TEF     1762 non-null   float64 
 11  UPS     1762 non-null   float64 
dtypes: float64(12)
memory usage: 179.0 KB
None
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1762 entries, 2010-01-04 to 2016-12-30
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype  
---  --  
 0   AAPL    1761 non-null   float64 
 1   AMGN    1761 non-null   float64 
 2   AMZN    1761 non-null   float64 
 3   CPRT    1761 non-null   float64 
 4   EL      1762 non-null   float64 
 5   GS      1762 non-null   float64 
 6   ILMN    1761 non-null   float64 
 7   MA      1762 non-null   float64 
 8   PAA     1762 non-null   float64 
 9   RIO     1762 non-null   float64 
 10  TEF     1762 non-null   float64 
 11  UPS     1762 non-null   float64 
 12  Index   1762 non-null   float64 
dtypes: float64(13)
memory usage: 257.3 KB
None
```

C:\Users\yeiso\AppData\Local\Temp\ipykernel\_19956\1964733718.py:9: FutureWarning: The default fill\_method='pad' in DataFrame.pct\_change is deprecated and will be removed in a future version. Call ffill before calling pct\_change to retain current behavior and silence this warning.

```
returns = data.pct_change()
```

# 10. Introduction to Data Visualization with Matplotlib

## Chapter 1 - Introduction to Matplotlib

This chapter introduces the Matplotlib visualization library and demonstrates how to use it with data.

### Using the `matplotlib.pyplot` interface

There are many ways to use Matplotlib. In this course, we will focus on the pyplot interface, which provides the most flexibility in creating and customizing data visualizations.

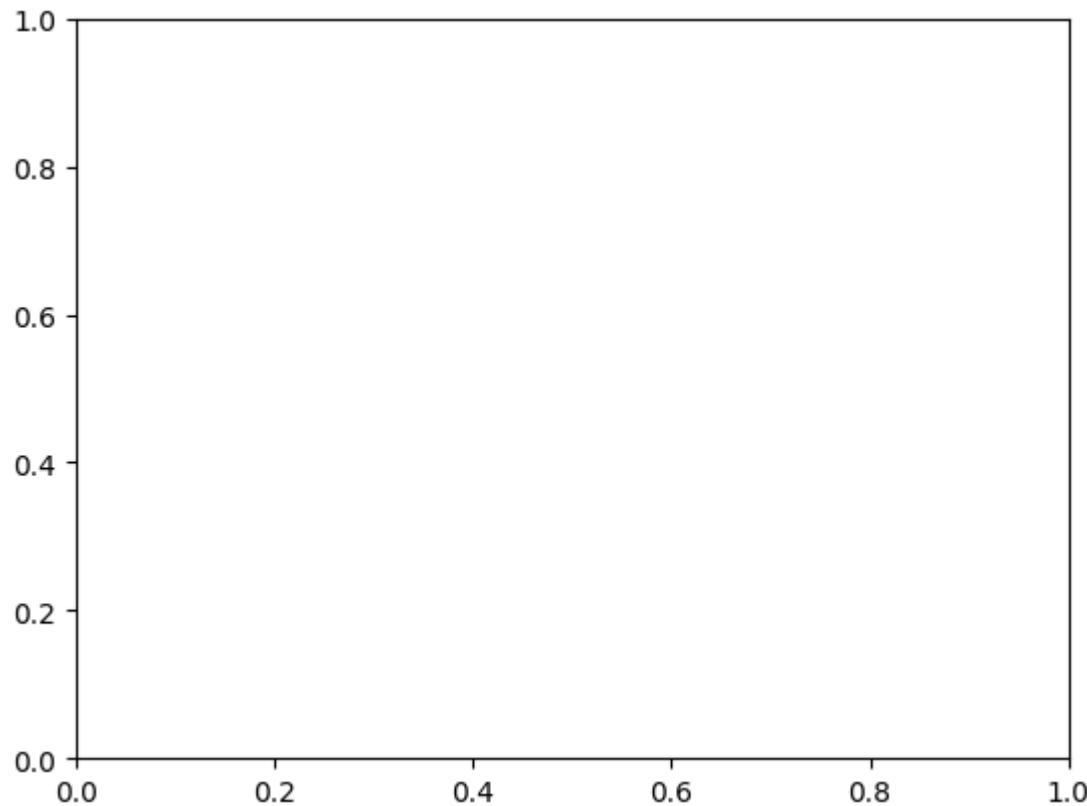
Initially, we will use the pyplot interface to create two kinds of objects: Figure objects and Axes objects.

This course introduces a lot of new concepts, so if you ever need a quick refresher, download the Matplotlib Cheat Sheet and keep it handy!

```
In [ ]: # Import the matplotlib.pyplot submodule and name it plt
import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots
fig, ax = plt.subplots()

# Call the show function to show the result
plt.show()
```



Adding data to an Axes object

```
In [ ]: import pandas as pd

# Specify the file paths using double backslashes
austin_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\weather\\\\austin_2010_to_2014.csv')

seattle_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\weather\\\\seattle_2010_to_2014.csv')

# Import the matplotlib.pyplot submodule and name it plt
import matplotlib.pyplot as plt

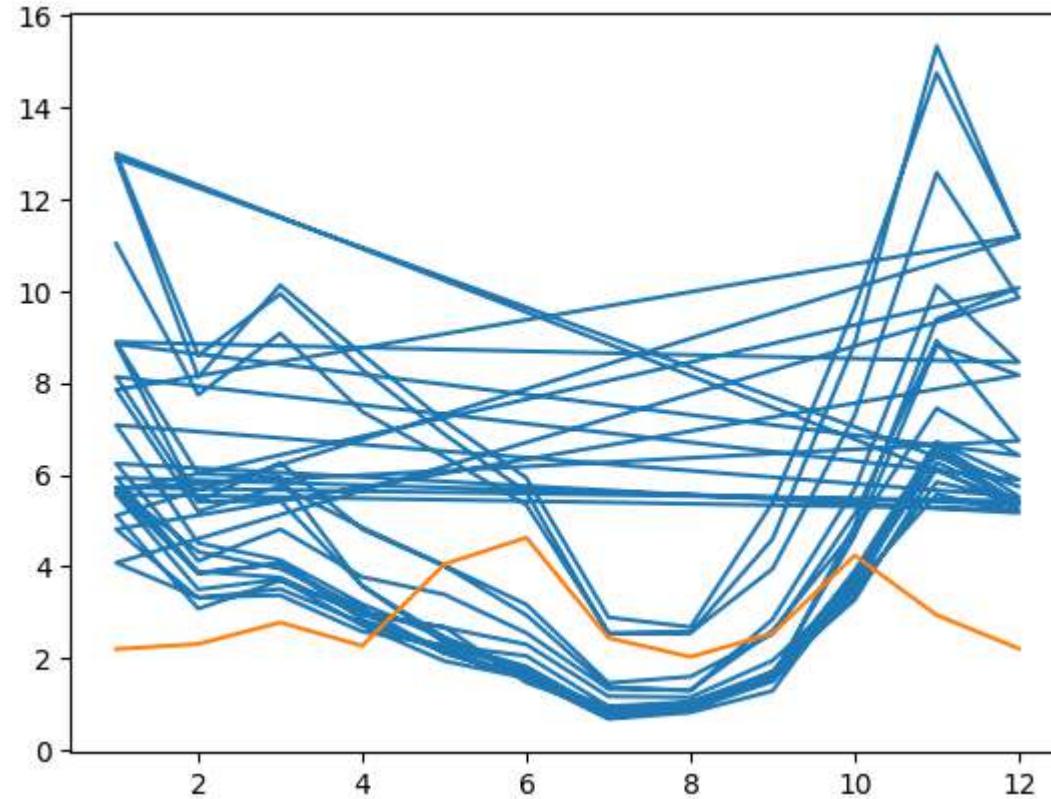
# Create a Figure and an Axes with plt.subplots
fig, ax = plt.subplots()

# Plot MLY-PRCP-NORMAL from seattle_weather against the MONTH
ax.plot(seattle_weather["DATE"], seattle_weather["MLY-PRCP-NORMAL"])

# Plot MLY-PRCP-NORMAL from austin_weather against MONTH
ax.plot(austin_weather["DATE"], austin_weather["MLY-PRCP-NORMAL"])
```

```
ax.plot(austin_weather["DATE"], austin_weather["MLY-PRCP-NORMAL"])

# Call the show function
plt.show()
```



Customizing data appearance

```
In [ ]: import pandas as pd

# Specify the file paths using double backslashes
austin_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\weather\\\\austin\\\\austin.csv')

seattle_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\weather\\\\seattle\\\\seattle.csv')

# Import the matplotlib.pyplot submodule and name it plt
import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots
fig, ax = plt.subplots()
```

```

#-----
#-----  

#second part of the code  

# Plot Seattle data, setting data appearance  

ax.plot(seattle_weather["DATE"], seattle_weather["MLY-PRCP-NORMAL"], color='b', marker='o', linestyle='--')  

# Plot Austin data, setting data appearance  

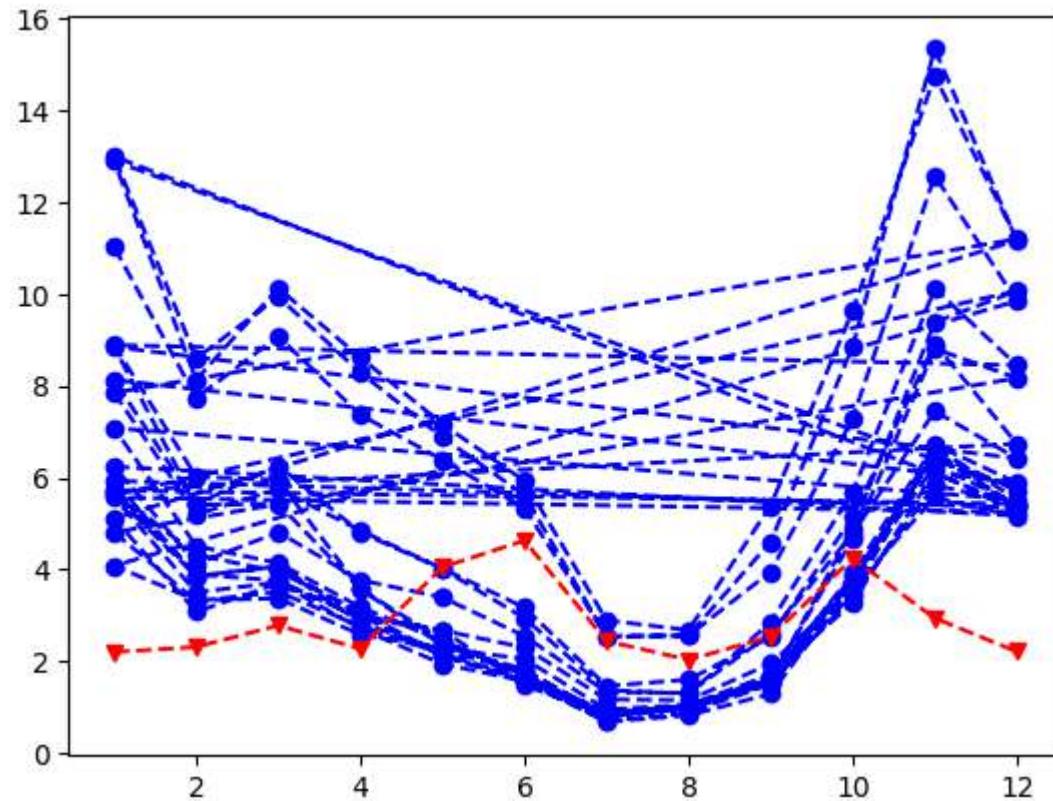
ax.plot(austin_weather["DATE"], austin_weather["MLY-PRCP-NORMAL"], color='r', marker='v', linestyle='--')  

# Call show to display the resulting plot  

plt.show();

```



Customizing axis labels and adding titles

In [ ]: `import pandas as pd`

```

# Specify the file paths using double backslashes
austin_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course

```

```
seattle_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course\\\\weather\\\\seattle.csv')

# Import the matplotlib.pyplot submodule and name it plt
import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots
fig, ax = plt.subplots()

#-----
#-----

#second part of the code
# Plot Seattle data, setting data appearance
ax.plot(seattle_weather["DATE"], seattle_weather["MLY-PRCP-NORMAL"], color='b', marker='o', linestyle='--')

# Plot Austin data, setting data appearance
ax.plot(austin_weather["DATE"], austin_weather["MLY-PRCP-NORMAL"], color='r', marker='v', linestyle='--')

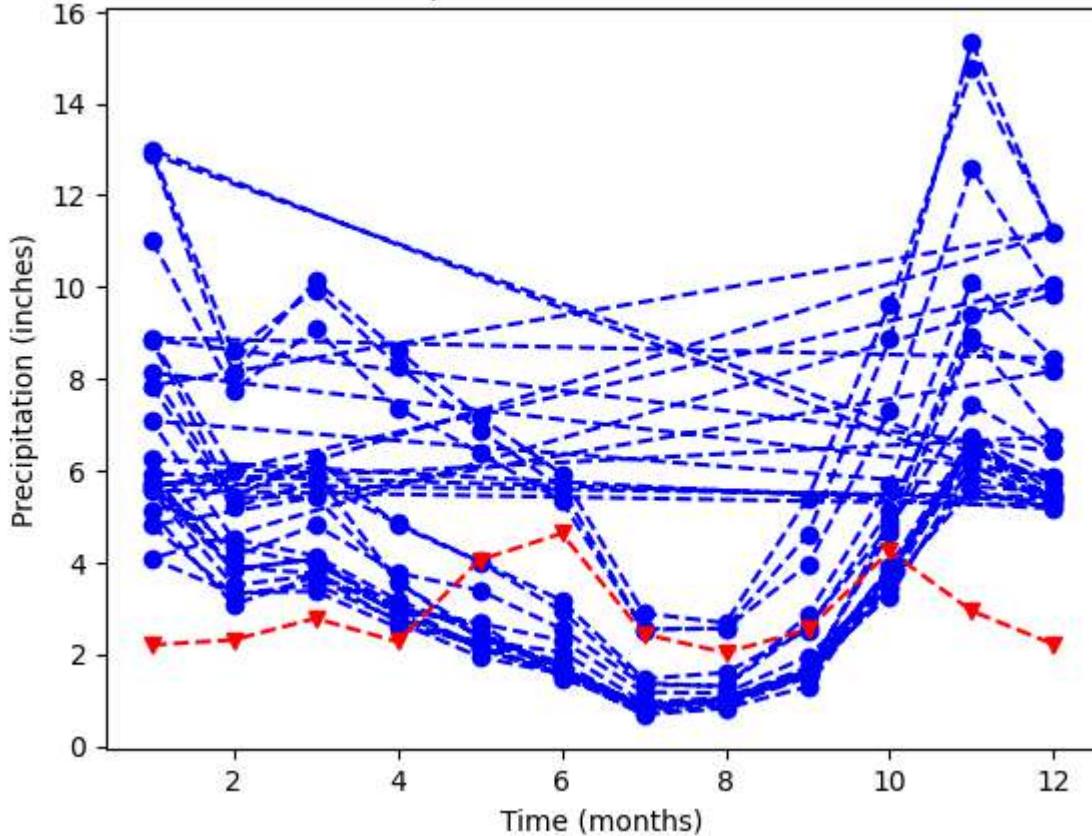
# Customize the x-axis label
ax.set_xlabel('Time (months)')

# Customize the y-axis label
ax.set_ylabel('Precipitation (inches)')

# Add the title
ax.set_title('Weather patterns in Austin and Seattle')

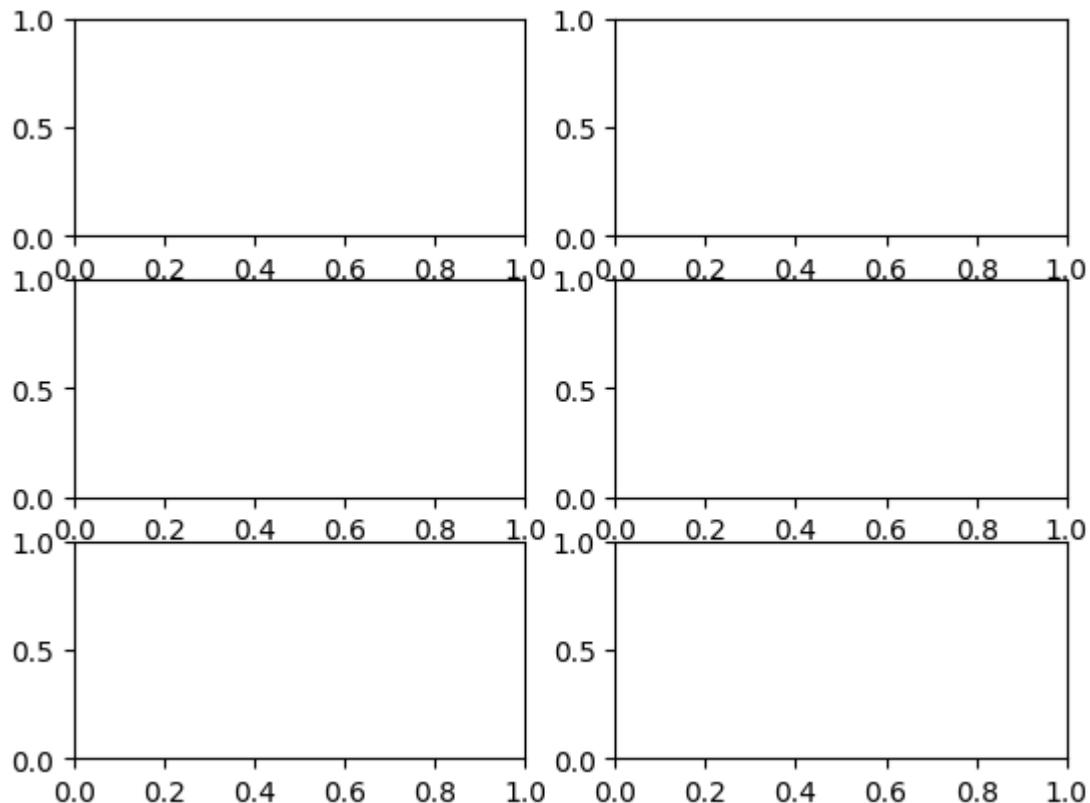
# Display the figure
plt.show()
```

## Weather patterns in Austin and Seattle



Creating a grid of subplots

```
In [ ]: fig, ax = plt.subplots(3, 2)
```



Creating small multiples with plt.subplots

```
In [ ]: # Create a Figure and an array of subplots with 2 rows and 2 columns
fig, ax = plt.subplots(2, 2)

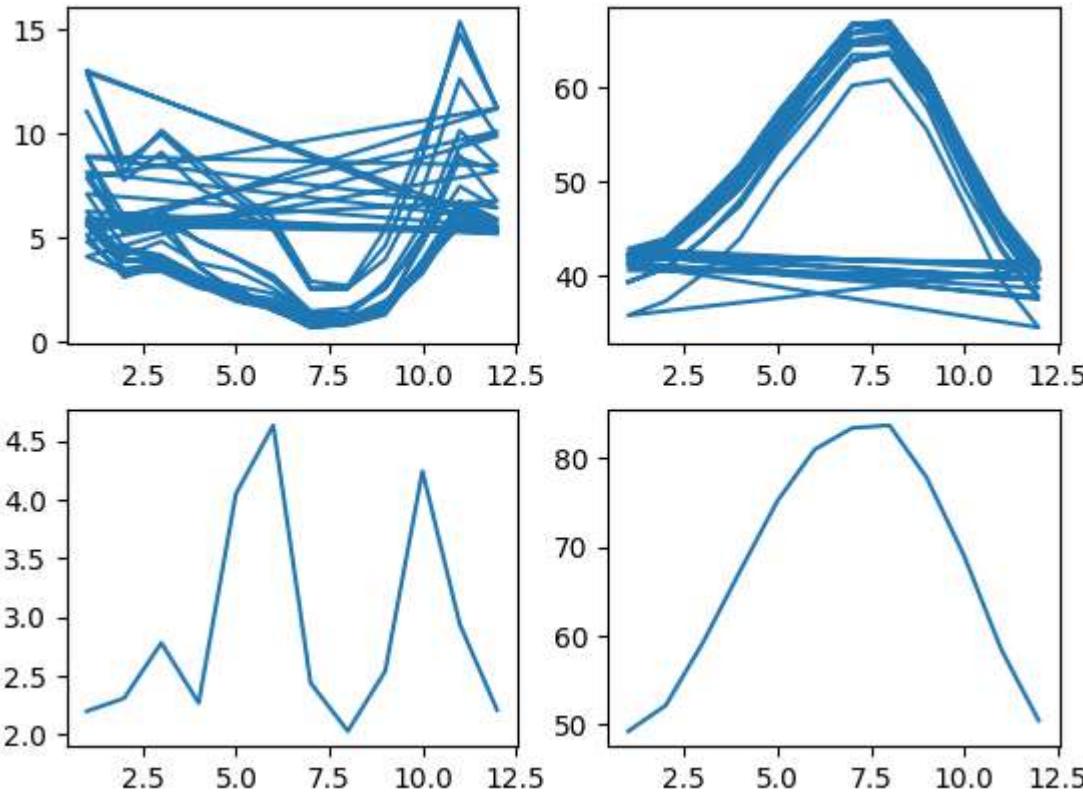
# Addressing the top left Axes as index 0, 0, plot month and Seattle precipitation
ax[0, 0].plot(seattle_weather['DATE'], seattle_weather['MLY-PRCP-NORMAL'])

# In the top right (index 0,1), plot month and Seattle temperatures
ax[0, 1].plot(seattle_weather['DATE'], seattle_weather['MLY-TAVG-NORMAL'])

# In the bottom left (1, 0) plot month and Austin precipitations
ax[1, 0].plot(austin_weather['DATE'], austin_weather['MLY-PRCP-NORMAL'])

# In the bottom right (1, 1) plot month and Austin temperatures
ax[1, 1].plot(austin_weather['DATE'], austin_weather['MLY-TAVG-NORMAL'])

plt.show()
```



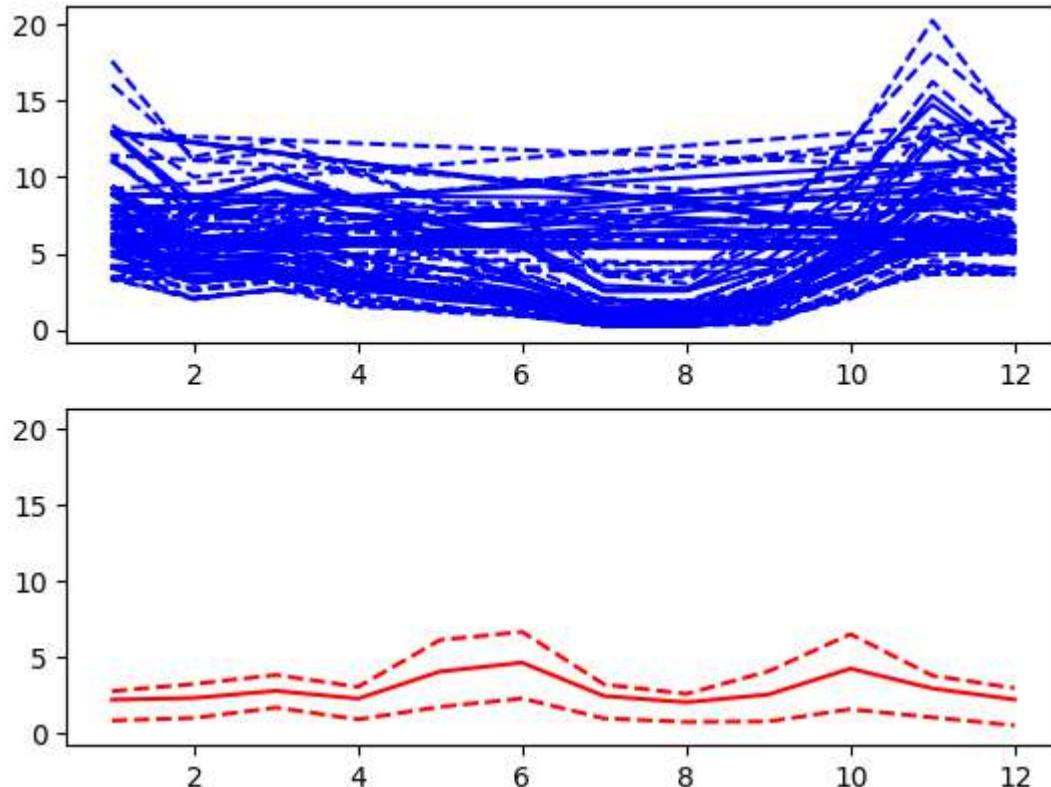
Small multiples with shared y axis

```
In [ ]: # Create a figure and an array of axes: 2 rows, 1 column with shared y axis
fig, ax = plt.subplots(2, 1, sharey=True)

# Plot Seattle precipitation data in the top axes
ax[0].plot(seattle_weather['DATE'], seattle_weather['MLY-PRCP-NORMAL'], color = 'b')
ax[0].plot(seattle_weather['DATE'], seattle_weather['MLY-PRCP-25PCTL'], color = 'b', linestyle = '--')
ax[0].plot(seattle_weather['DATE'], seattle_weather['MLY-PRCP-75PCTL'], color = 'b', linestyle = '--')

# Plot Austin precipitation data in the bottom axes
ax[1].plot(austin_weather['DATE'], austin_weather['MLY-PRCP-NORMAL'], color = 'r')
ax[1].plot(austin_weather['DATE'], austin_weather['MLY-PRCP-25PCTL'], color = 'r', linestyle = '--')
ax[1].plot(austin_weather['DATE'], austin_weather['MLY-PRCP-75PCTL'], color = 'r', linestyle = '--')

plt.show()
```



## Chapter 2 -Plotting time-series

Time series data is data that is recorded. Visualizing this type of data helps clarify trends and illuminates relationships between data.

Read data with a time index

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

# Read the data from file using read_csv
climate_change = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
climate_change.head()
```

Out[ ]:

co2 relative\_temp

date		
1958-03-06	315.71	0.10
1958-04-06	317.45	0.01
1958-05-06	317.50	0.08
1958-06-06	NaN	-0.05
1958-07-06	315.86	0.06

Plot time-series data

In [ ]:

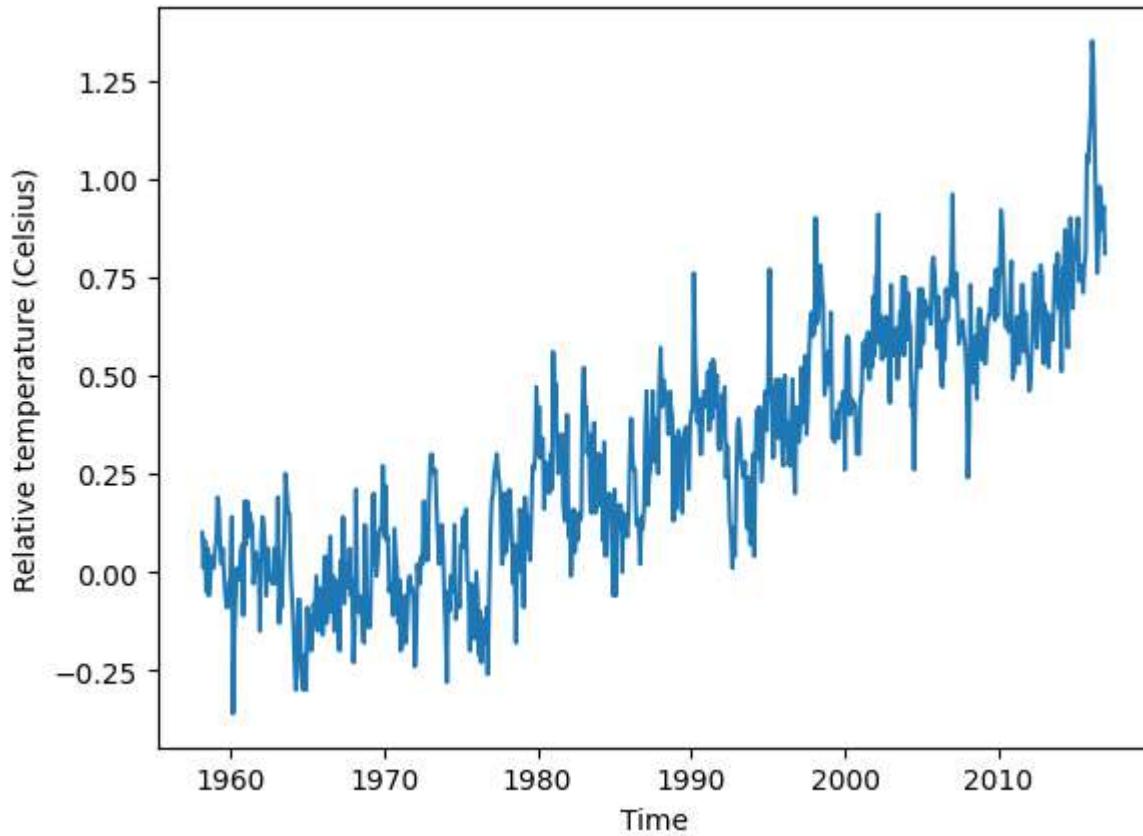
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Add the time-series for "relative_temp" to the plot
ax.plot(climate_change.index, climate_change['relative_temp'])

# Set the x-axis label
ax.set_xlabel('Time')

# Set the y-axis label
ax.set_ylabel('Relative temperature (Celsius)')

# Show the figure
plt.show()
```



Using a time index to zoom in

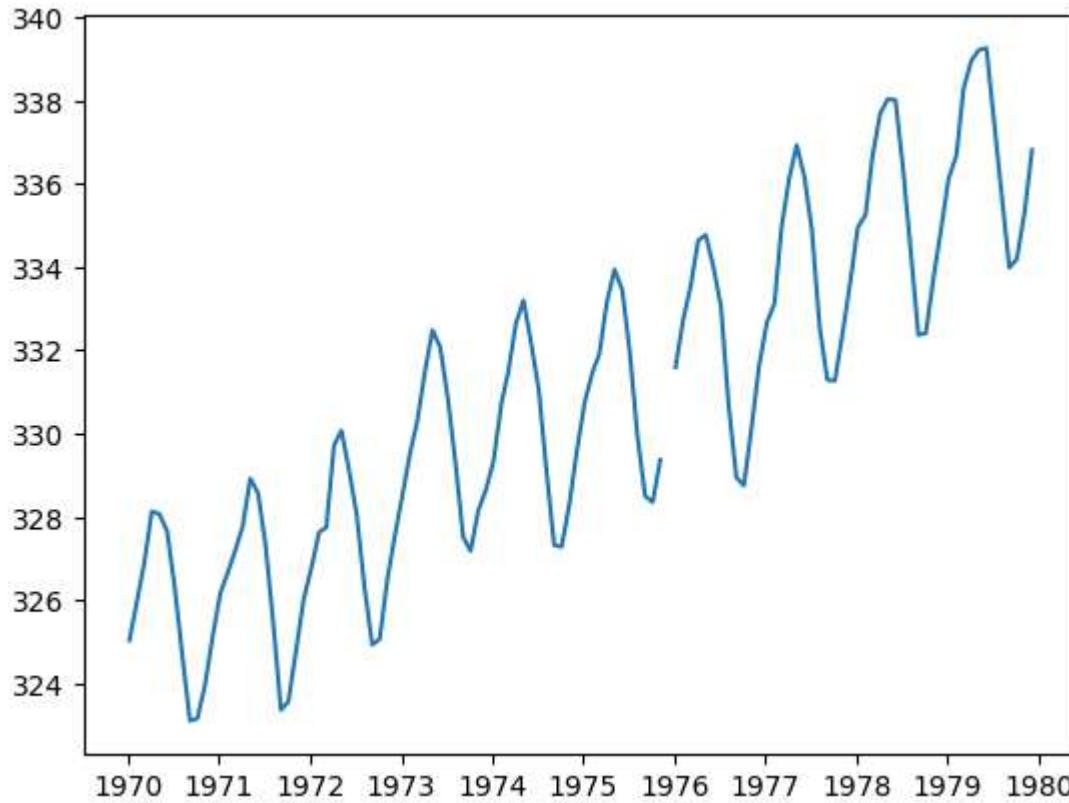
```
In [ ]: import matplotlib.pyplot as plt

# Use plt.subplots to create fig and ax
fig, ax = plt.subplots()

# Create variable seventies with data from "1970-01-01" to "1979-12-31"
seventies = climate_change["1970-01-01":"1979-12-31"]

# Add the time-series for "co2" data from seventies to the plot
ax.plot(seventies.index, seventies["co2"])

# Show the figure
plt.show()
```



Plotting two variables

```
In [ ]: import matplotlib.pyplot as plt

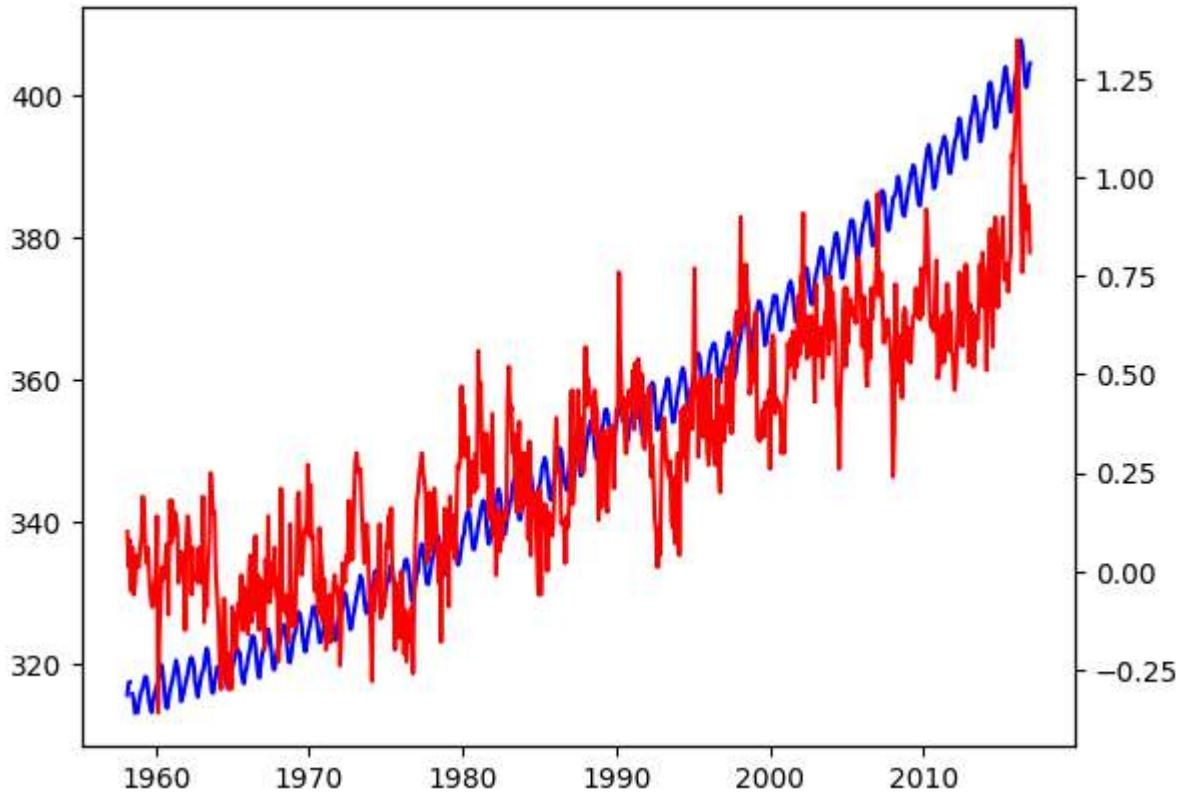
# Initialize a Figure and Axes
fig, ax = plt.subplots()

# Plot the CO2 variable in blue
ax.plot(climate_change.index, climate_change['co2'], color='b')

# Create a twin Axes that shares the x-axis
ax2 = ax.twinx()

# Plot the relative temperature in red
ax2.plot(climate_change.index, climate_change['relative_temp'], color='r')

plt.show()
```



Defining a function that plots time-series data

```
In [ ]: # Define a function called plot_timeseries
def plot_timeseries(axes, x, y, color, xlabel, ylabel):

    # Plot the inputs x,y in the provided color
    axes.plot(x, y, color=color)

    # Set the x-axis label
    axes.set_xlabel(xlabel)

    # Set the y-axis label
    axes.set_ylabel(ylabel, color=color)

    # Set the colors tick params for y-axis
    axes.tick_params('y', colors=color)
```

Using a plotting function

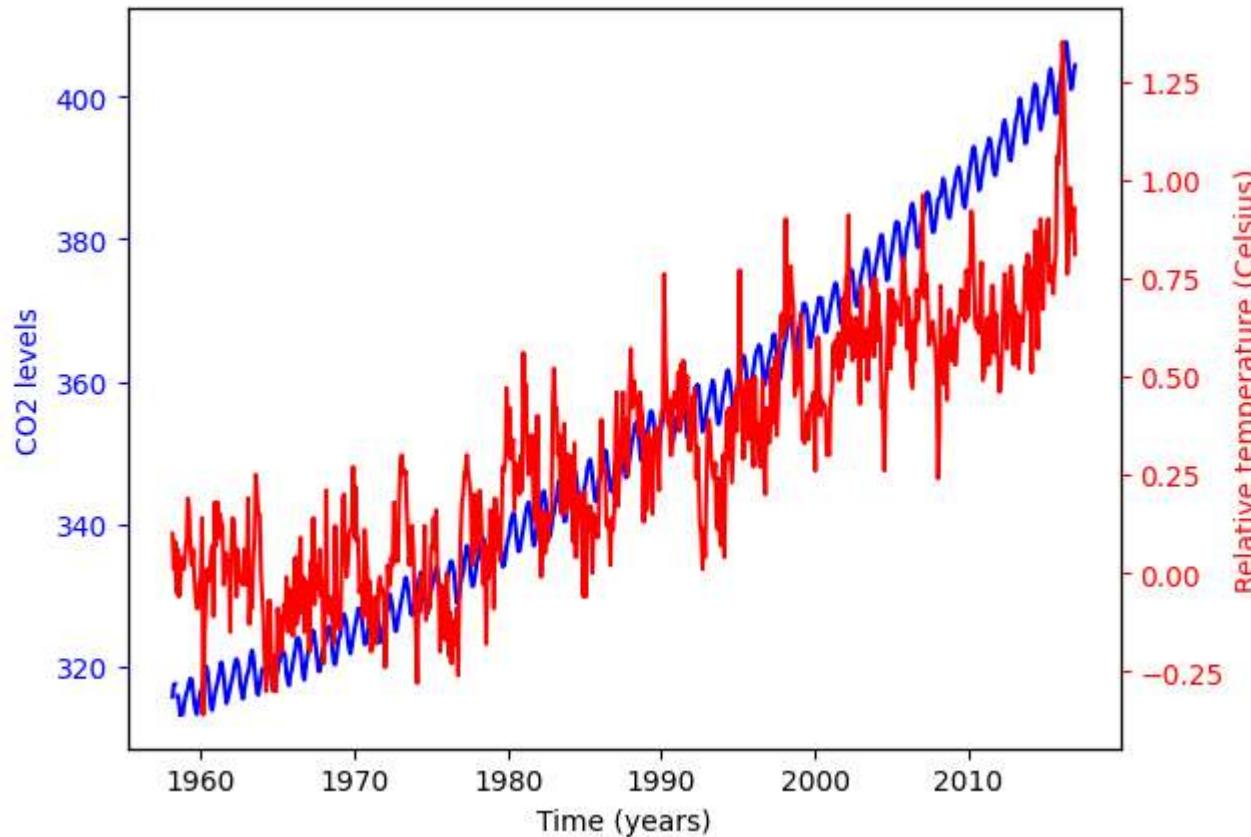
```
In [ ]: fig, ax = plt.subplots()

# Plot the CO2 levels time-series in blue
plot_timeseries(ax, climate_change.index, climate_change['co2'], "blue", "Time (years)", "CO2 levels")

# Create a twin Axes object that shares the x-axis
ax2 = ax.twinx()

# Plot the relative temperature data in red
plot_timeseries(ax2, climate_change.index, climate_change['relative_temp'], "red", "Time (years)", "Relative temperature (Celsius)")

plt.show()
```



Annotating a plot of time-series data

```
In [ ]: fig, ax = plt.subplots()
```

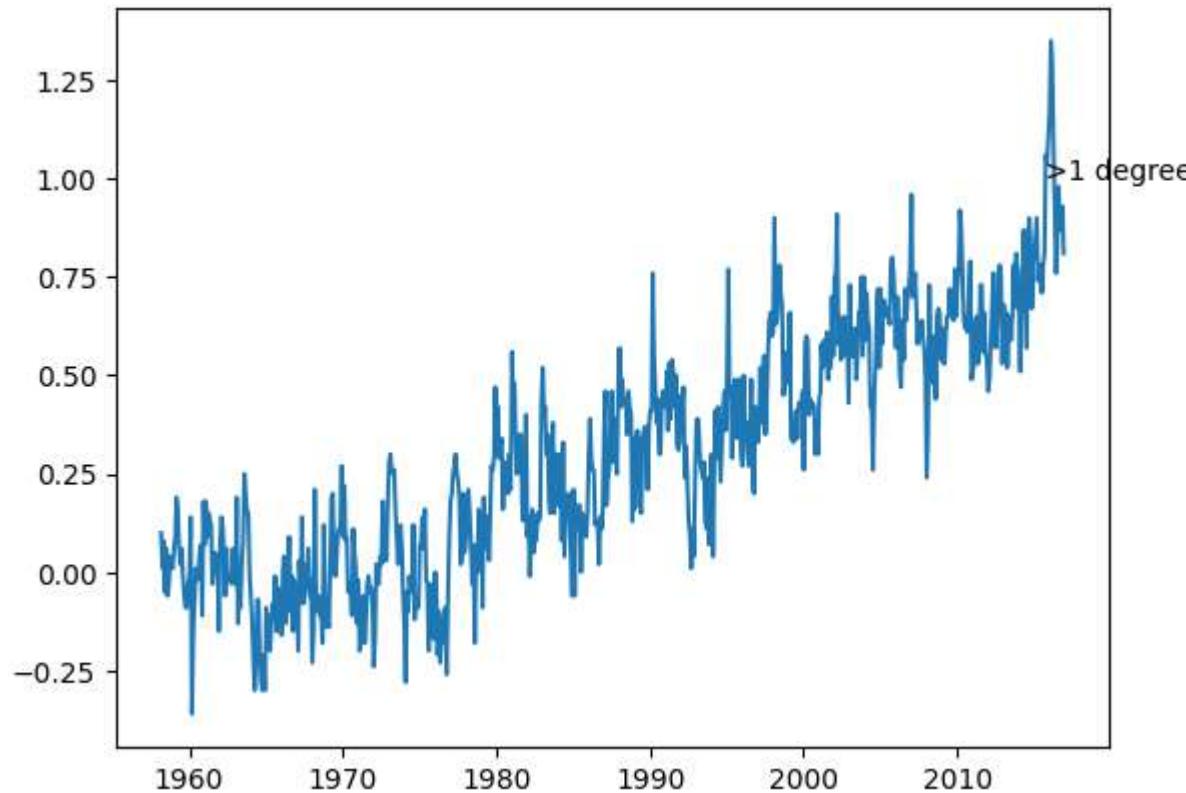
```

# Plot the relative temperature data
ax.plot(climate_change.index, climate_change.relative_temp)

# Annotate the date at which temperatures exceeded 1 degree
ax.annotate('>1 degree', xy=(pd.Timestamp('2015-10-06'), 1))

plt.show()

```



Plotting time-series: putting it all together

```

In [ ]: fig, ax = plt.subplots()

# Plot the CO2 levels time-series in blue
plot_timeseries(ax, climate_change.index, climate_change.co2, 'blue', 'Time (years)', 'CO2 levels')

# Create an Axes object that shares the x-axis
ax2 = ax.twinx()

# Plot the relative temperature data in red

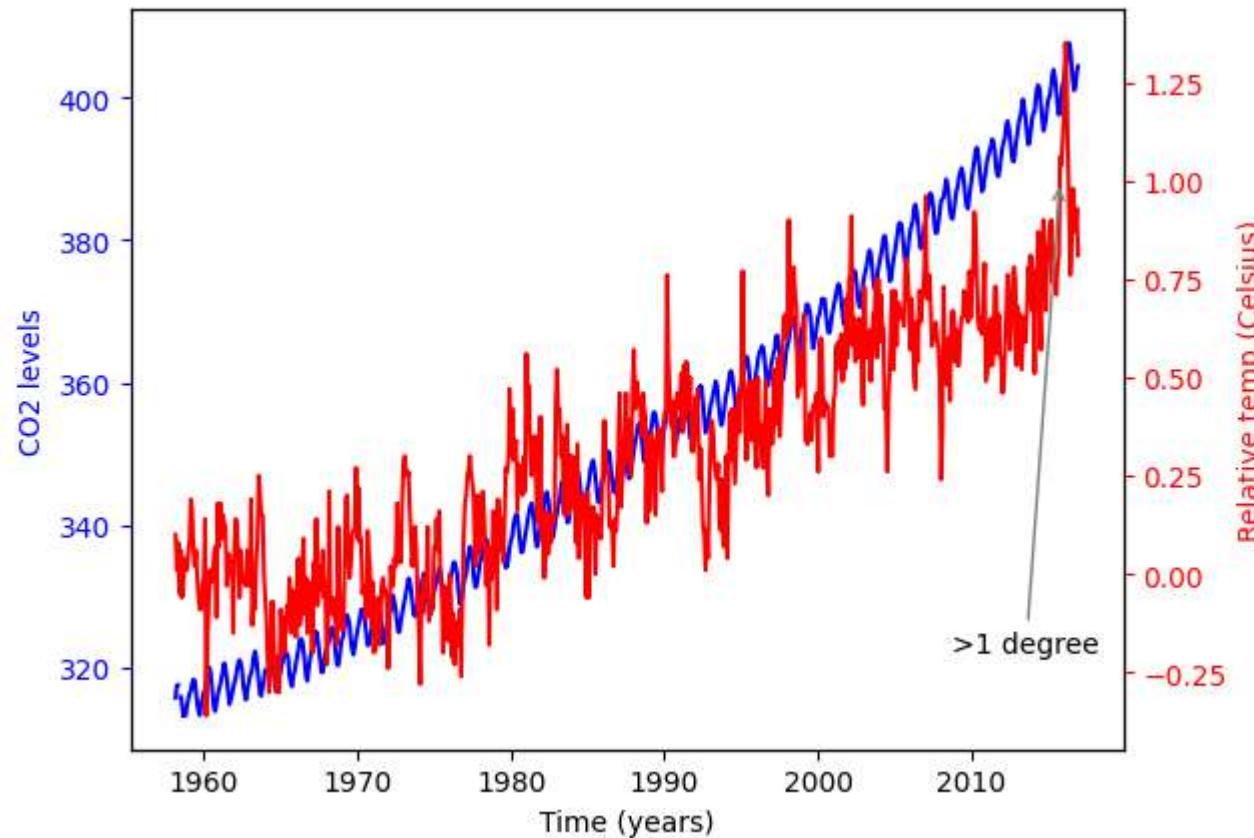
```

```

plot_timeseries(ax2, climate_change.index, climate_change.relative_temp, 'red', 'Time (years)', 'Relative temp (Celsius)')

# Annotate point with relative temperature >1 degree
ax2.annotate(">1 degree", xy=(pd.Timestamp('2015-10-06'), 1), xytext=(pd.Timestamp('2008-10-06'), -0.2), arrowprops={'arrowstyle': '->', 'color': 'black', 'lw': 1}, va='top', ha='left')
plt.show()

```



## Chapter 3 - Quantitative comparisons and statistical visualizations

Visualizations can be used to compare data in a quantitative manner. This chapter explains several methods for quantitative visualizations.

In [ ]:

```

import pandas as pd
import matplotlib.pyplot as plt

```

```
# Specify the file path using double backslashes
```

```
medals = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\medals.csv')

# Rename the 'Unnamed: 0' column to 'Country'
medals.rename(columns={'Unnamed: 0': 'Country'}, inplace=True)

# Display the first few rows of the DataFrame
medals.head()
```

Out[ ]:

	Country	Bronze	Gold	Silver
0	United States	67	137	52
1	Germany	67	47	43
2	Great Britain	26	64	55
3	Russia	35	50	28
4	China	35	44	30

Bar chart

In [ ]:

```
fig, ax = plt.subplots()

# Set 'Country' as the index
medals.set_index('Country', inplace=True)

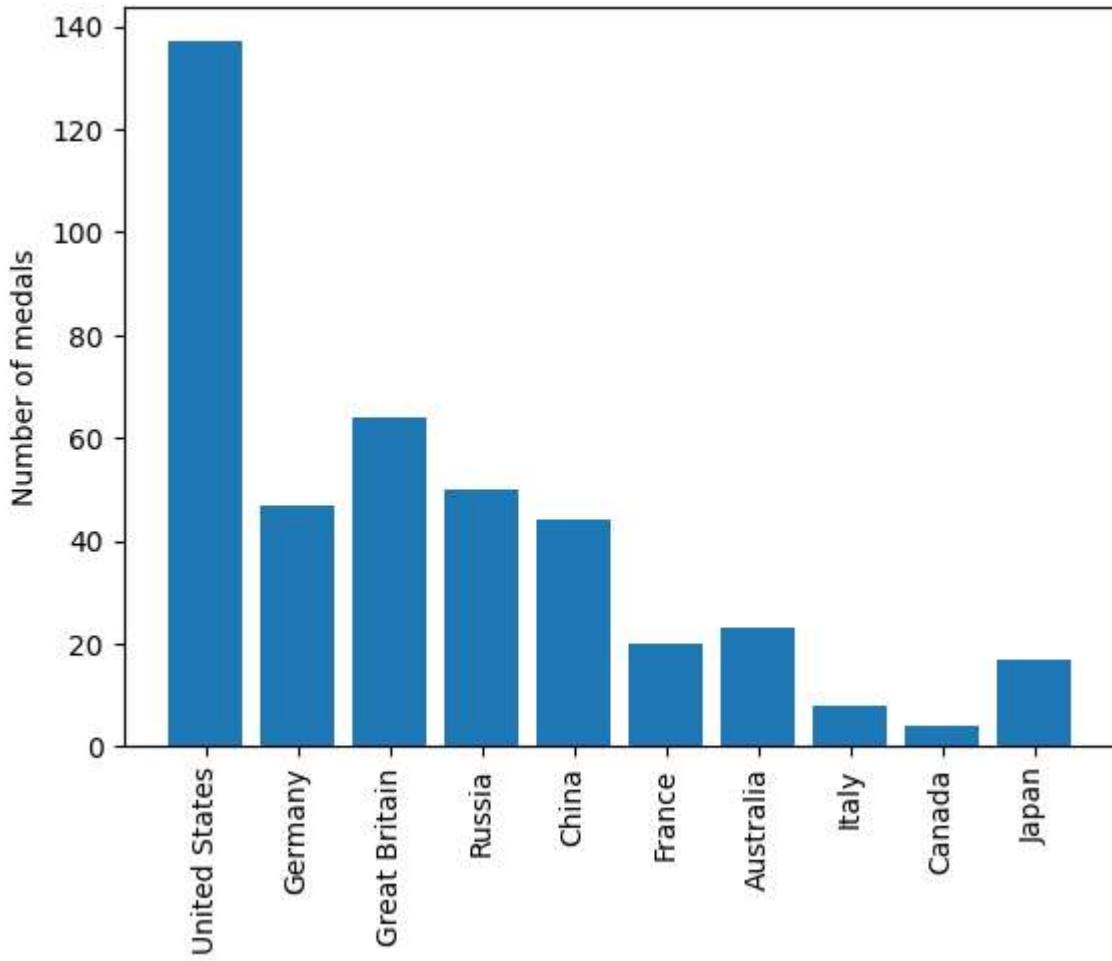
# Plot a bar-chart of gold medals as a function of country
ax.bar(medals.index, medals.Gold)

# Set the x-axis tick labels to the country names
ax.set_xticklabels(medals.index, rotation=90)

# Set the y-axis label
ax.set_ylabel('Number of medals')

plt.show()
```

C:\Users\yeiso\AppData\Local\Temp\ipykernel\_18080\130371325.py:12: UserWarning: set\_xticklabels() should only be used with a fixed number of ticks, i.e. after set\_ticks() or using a FixedLocator.  
ax.set\_xticklabels(medals.index, rotation=90)



Stacked bar chart

```
In [ ]: fig, ax = plt.subplots()

# Plot a bar-chart of gold medals as a function of country
ax.bar(medals.index, medals.Gold)

# Set the x-axis tick labels to the country names
ax.set_xticklabels(medals.index, rotation=90)

# Set the y-axis label
ax.set_ylabel('Number of medals')
```

```
#second part of the code
# Add bars for "Gold" with the label "Gold"
ax.bar(medals.index, medals.Gold, label='Gold')

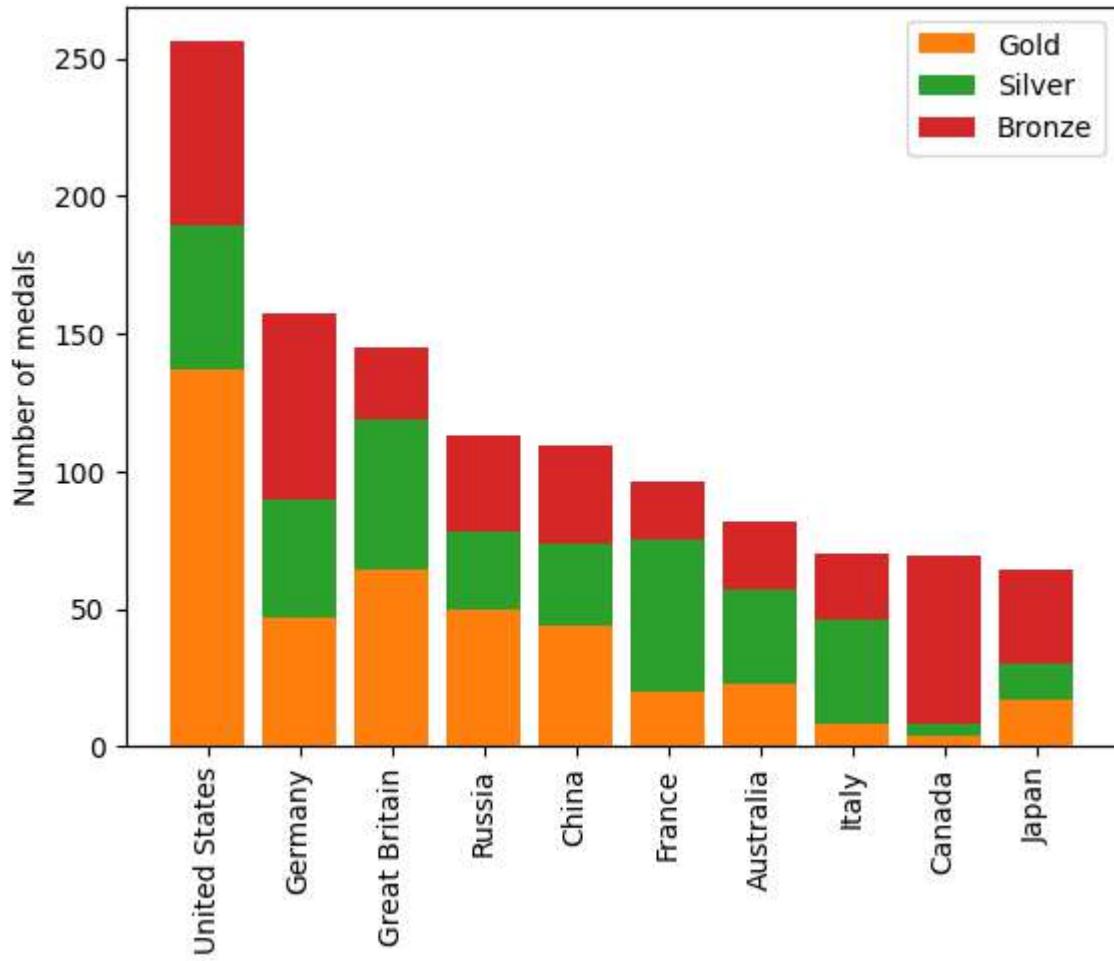
# Stack bars for "Silver" on top with label "Silver"
ax.bar(medals.index, medals.Silver, bottom=medals.Gold, label='Silver')

# Stack bars for "Bronze" on top of that with label "Bronze"
ax.bar(medals.index, medals.Bronze, bottom=medals.Gold + medals.Silver, label='Bronze')

# Display the legend
ax.legend()

plt.show()
```

C:\Users\yeiso\AppData\Local\Temp\ipykernel\_18080\1355922929.py:7: UserWarning: set\_ticklabels() should only be used with a fixed number of ticks, i.e. after set\_ticks() or using a FixedLocator.  
  ax.set\_xticklabels(medals.index, rotation=90)



Creating histograms

In [ ]:

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

# Specify the file path using double backslashes
summer2016 = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data

# Display the first few rows of the DataFrame
summer2016.head()
```

Out[ ]:

	Unnamed: 0	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
0	158	62	Giovanni Abagnale	M	21.0	198.0	90.0	Italy	ITA	2016 Summer	2016	Summer	Rio de Janeiro	Rowing	Rowing Men's Coxless Pairs	Bronze
1	161	65	Patimat Abakarova	F	21.0	165.0	49.0	Azerbaijan	AZE	2016 Summer	2016	Summer	Rio de Janeiro	Taekwondo	Taekwondo Women's Flyweight	Bronze
2	175	73	Luc Abalo	M	31.0	182.0	86.0	France	FRA	2016 Summer	2016	Summer	Rio de Janeiro	Handball	Handball Men's Handball	Silver
3	450	250	Saeid Morad Abdevali	M	26.0	170.0	80.0	Iran	IRI	2016 Summer	2016	Summer	Rio de Janeiro	Wrestling	Wrestling Men's Middleweight, Greco-Roman	Bronze
4	794	455	Denis Mikhaylovich Ablyazin	M	24.0	161.0	62.0	Russia	RUS	2016 Summer	2016	Summer	Rio de Janeiro	Gymnastics	Gymnastics Men's Team All-Around	Silver

In [ ]:

```
# Select data for mens_rowing
mens_rowing = summer2016[summer2016['Sport'] == "Rowing"]

# Select data for mens_gymnastics
mens_gymnastics = summer2016[summer2016['Sport'] == "Gymnastics"]

#second part of the code

fig, ax = plt.subplots()
# Plot a histogram of "Weight" for mens_rowing

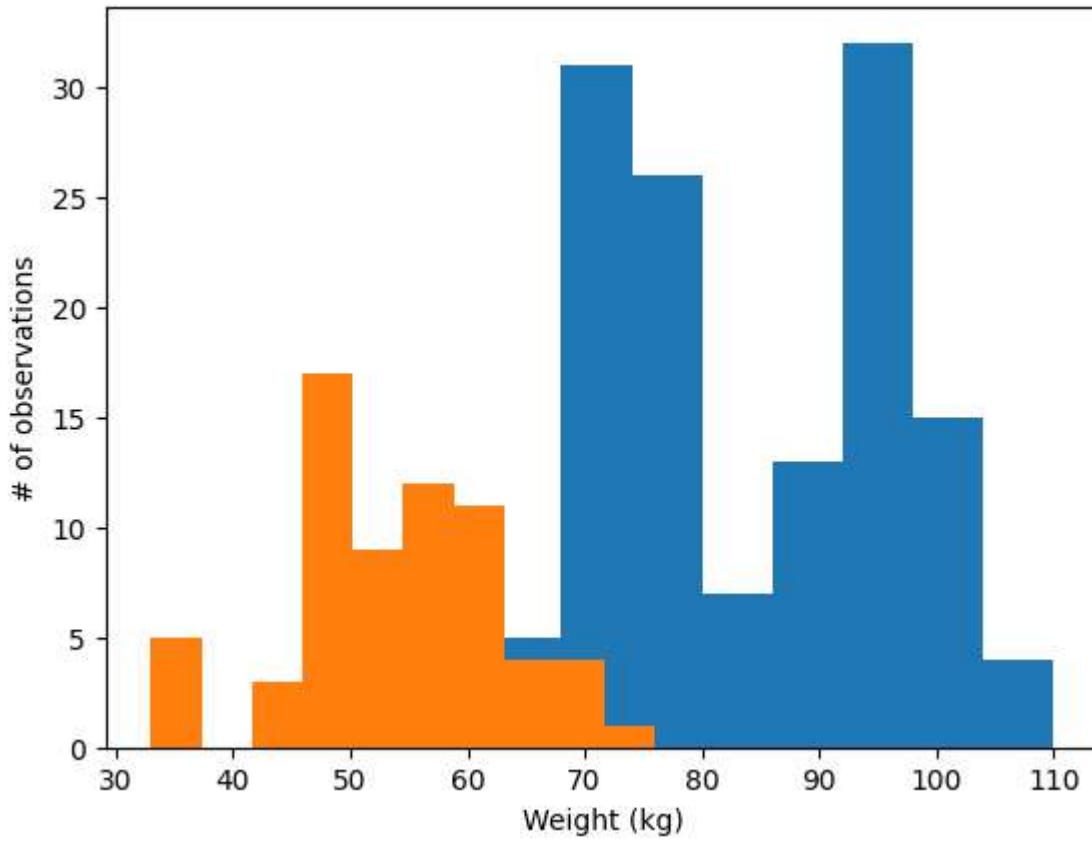
ax.hist(mens_rowing.Weight)

# Compare to histogram of "Weight" for mens_gymnastics
ax.hist(mens_gymnastics.Weight)

# Set the x-axis Label to "Weight (kg)"
ax.set_xlabel('Weight (kg)')

# Set the y-axis Label to "# of observations"
ax.set_ylabel('# of observations')

plt.show()
```



"Step" histogram

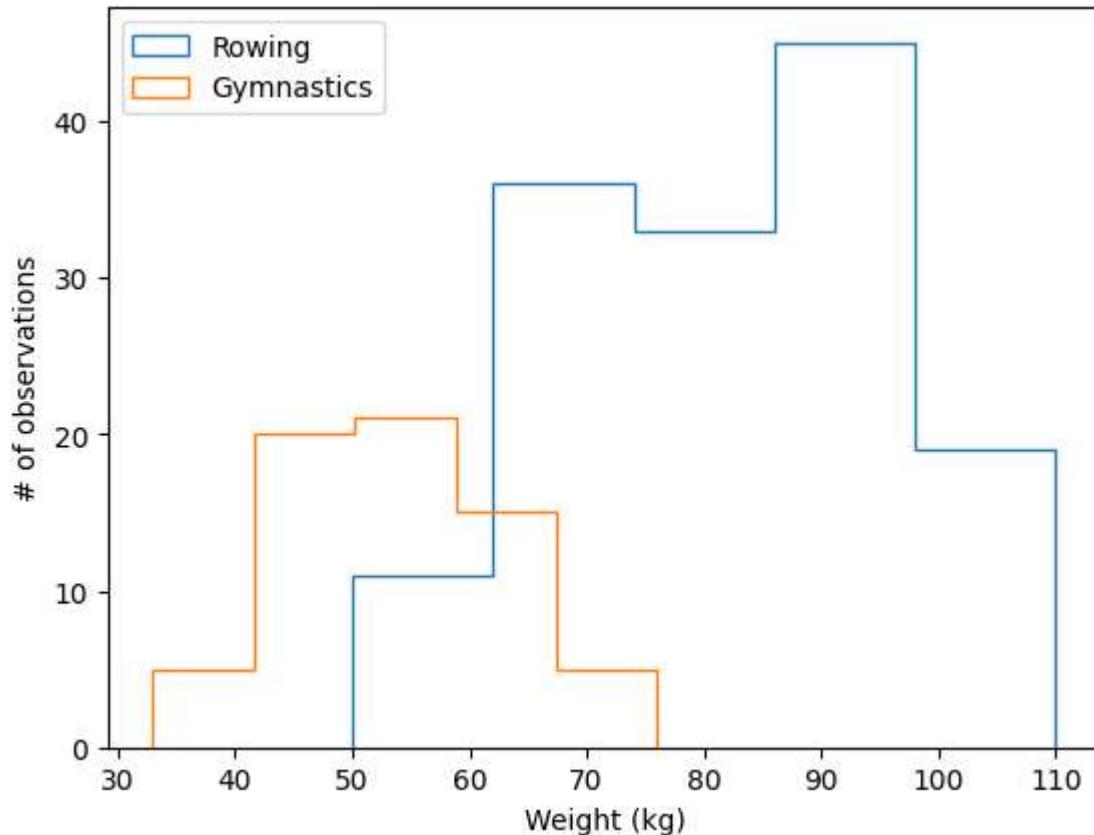
```
In [ ]: fig, ax = plt.subplots()

# Plot a histogram of "Weight" for mens_rowing
ax.hist(mens_rowing.Weight, label='Rowing', histtype='step', bins=5)

# Compare to histogram of "Weight" for mens_gymnastics
ax.hist(mens_gymnastics.Weight, label='Gymnastics', histtype='step', bins=5)

ax.set_xlabel("Weight (kg)")
ax.set_ylabel("# of observations")

# Add the legend and show the Figure
ax.legend()
plt.show()
```



Adding error-bars to a bar chart

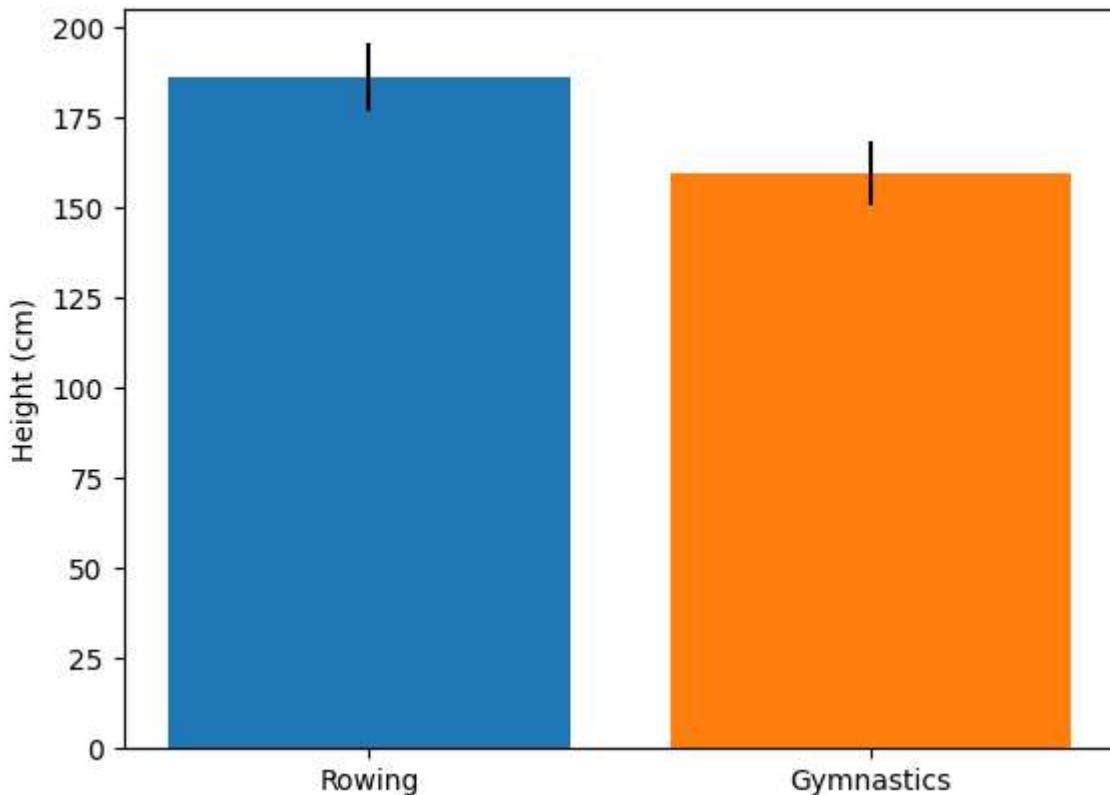
```
In [ ]: fig, ax = plt.subplots()

# Add a bar for the rowing "Height" column mean/std
ax.bar("Rowing", mens_rowing.Height.mean(), yerr=mens_rowing.Height.std())

# Add a bar for the gymnastics "Height" column mean/std
ax.bar("Gymnastics", mens_gymnastics.Height.mean(), yerr=mens_gymnastics.Height.std())

# Label the y-axis
ax.set_ylabel("Height (cm)")

plt.show()
```



Adding error-bars to a plot

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

# Read the data from file using read_csv
austin_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
seattle_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course

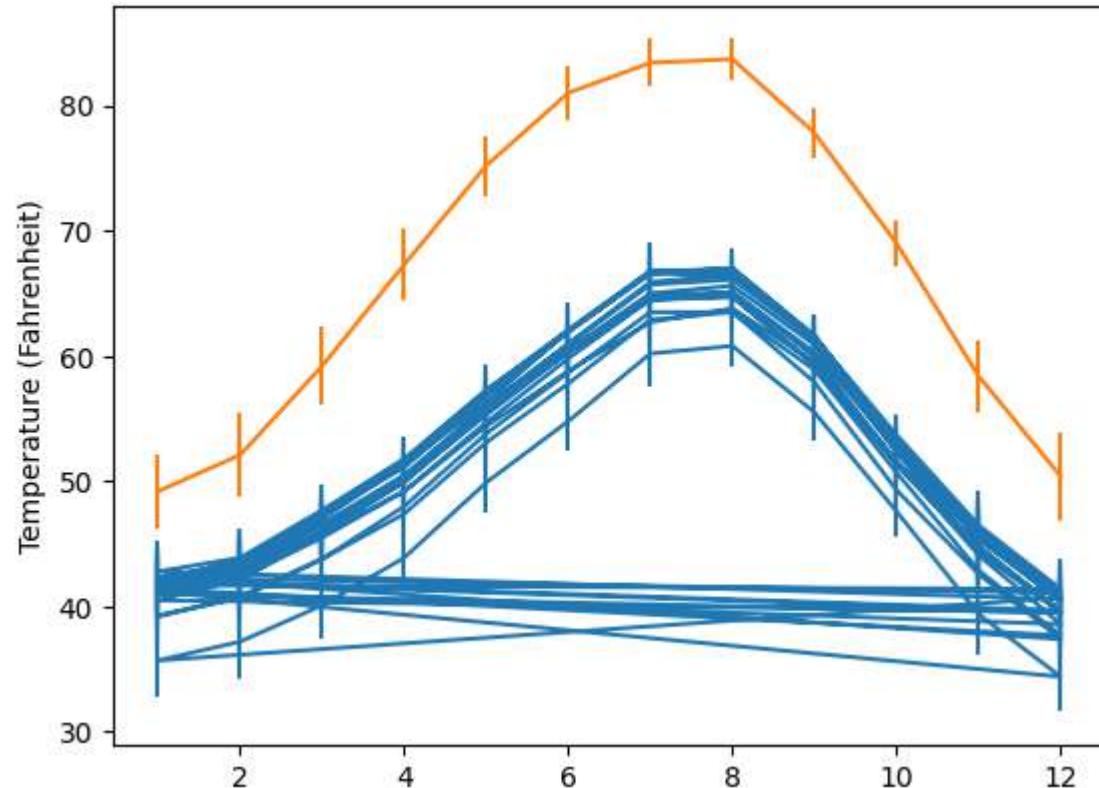
fig, ax = plt.subplots()

# Add Seattle temperature data in each month with error bars
ax.errorbar(seattle_weather.DATE, seattle_weather['MLY-TAVG-NORMAL'], yerr=seattle_weather['MLY-TAVG-STDDEV'])

# Add Austin temperature data in each month with error bars
ax.errorbar(austin_weather.DATE, austin_weather['MLY-TAVG-NORMAL'], yerr=austin_weather['MLY-TAVG-STDDEV'])

# Set the y-axis label
ax.set_ylabel('Temperature (Fahrenheit)')
```

```
plt.show()
```



Creating boxplots

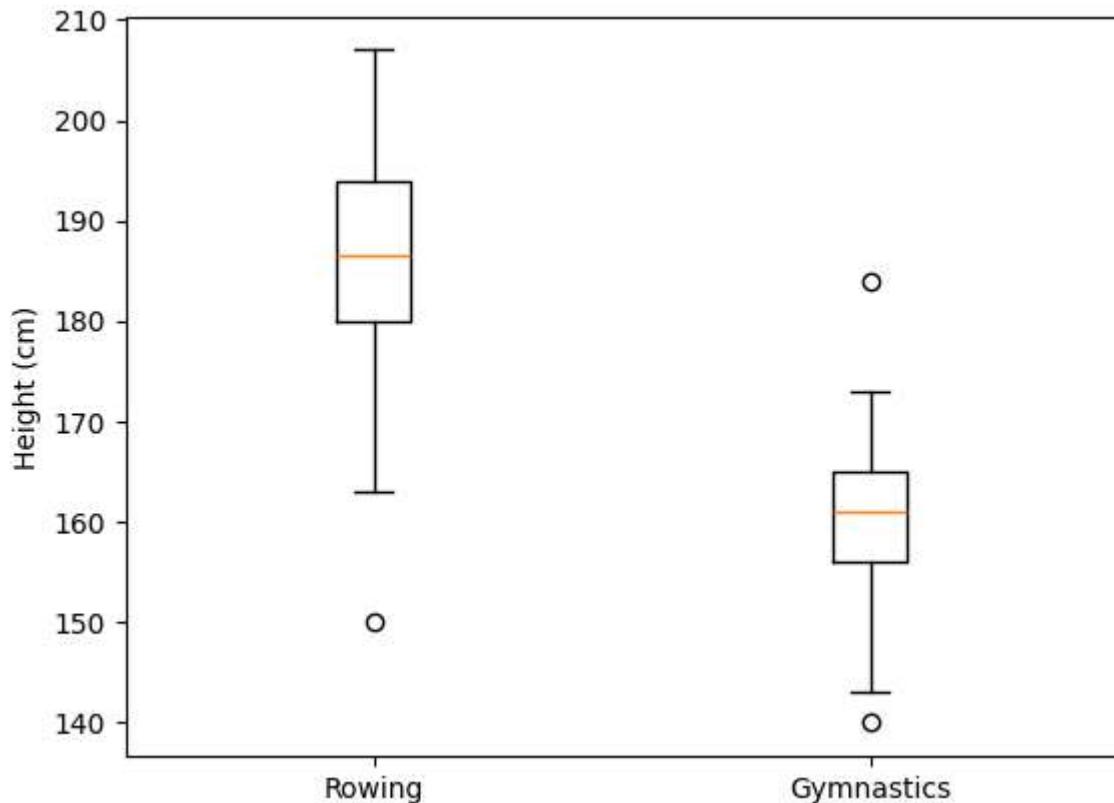
```
In [ ]: fig, ax = plt.subplots()

# Add a boxplot for the "Height" column in the DataFrames
ax.boxplot([mens_rowing.Height, mens_gymnastics.Height])

# Add x-axis tick labels:
ax.set_xticklabels(['Rowing', 'Gymnastics'])

# Add a y-axis label
ax.set_ylabel('Height (cm)')

plt.show()
```



Simple scatter plot

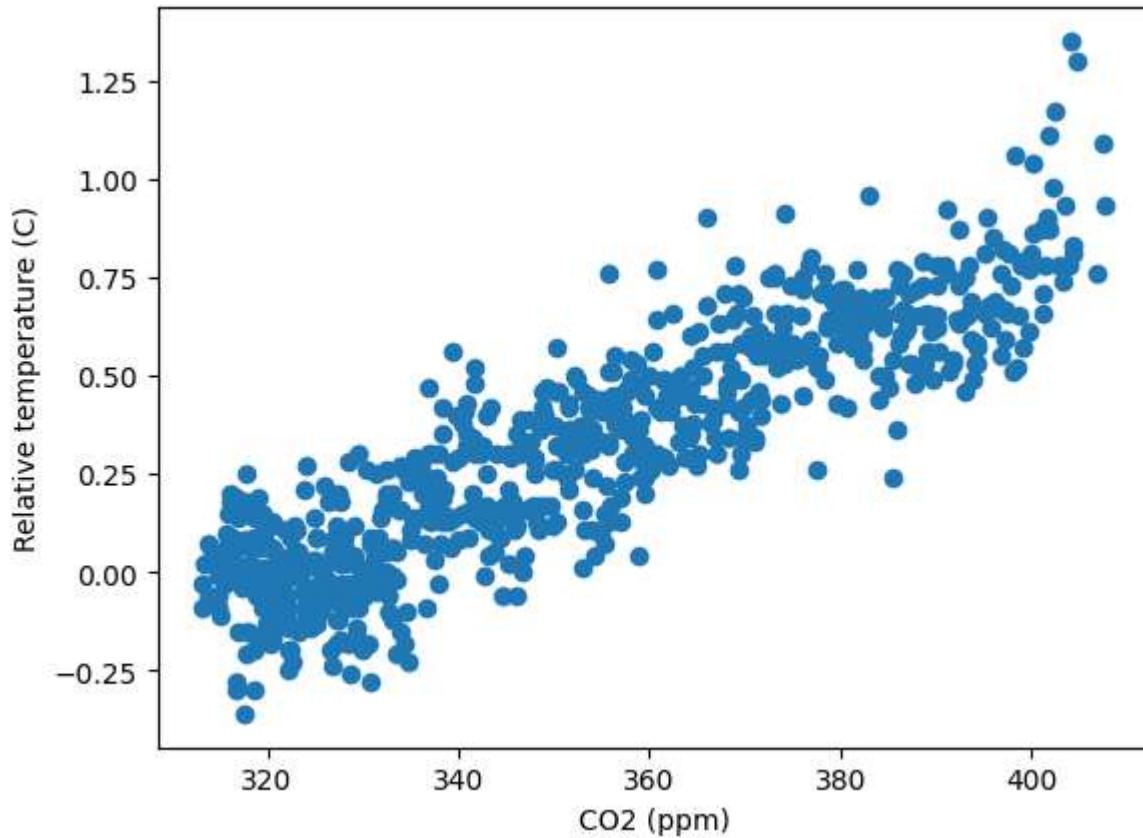
```
In [ ]: fig, ax = plt.subplots()

# Add data: "co2" on x-axis, "relative_temp" on y-axis
ax.scatter(climate_change.co2, climate_change.relative_temp)

# Set the x-axis label to "CO2 (ppm)"
ax.set_xlabel('CO2 (ppm)')

# Set the y-axis label to "Relative temperature (C)"
ax.set_ylabel('Relative temperature (C)')

plt.show()
```



Encoding time by color

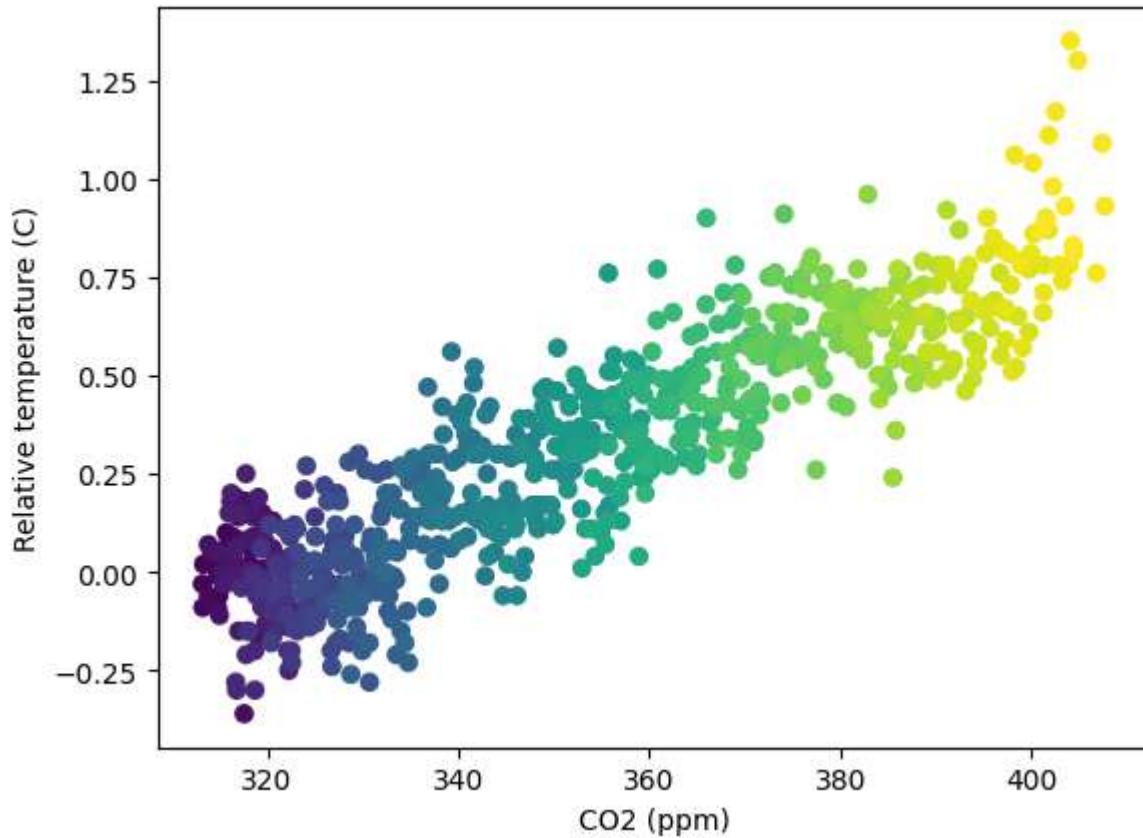
```
In [ ]: fig, ax = plt.subplots()

# Add data: "co2", "relative_temp" as x-y, index as color
ax.scatter(climate_change.co2, climate_change.relative_temp, c=climate_change.index)

# Set the x-axis label to "CO2 (ppm)"
ax.set_xlabel('CO2 (ppm)')

# Set the y-axis label to "Relative temperature (C)"
ax.set_ylabel('Relative temperature (C)')

plt.show()
```



## Chapter 4 - Sharing visualizations with others

This chapter shows you how to share your visualizations with others: how to save your figures as files, how to adjust their look and feel, and how to automate their creation based on input data.

### Selecting a style for printing

This chapter shows you how to share your visualizations with others: how to save your figures as files, how to adjust their look and feel, and how to automate their creation based on input data.

Selecting a style for printing

In [ ]:

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

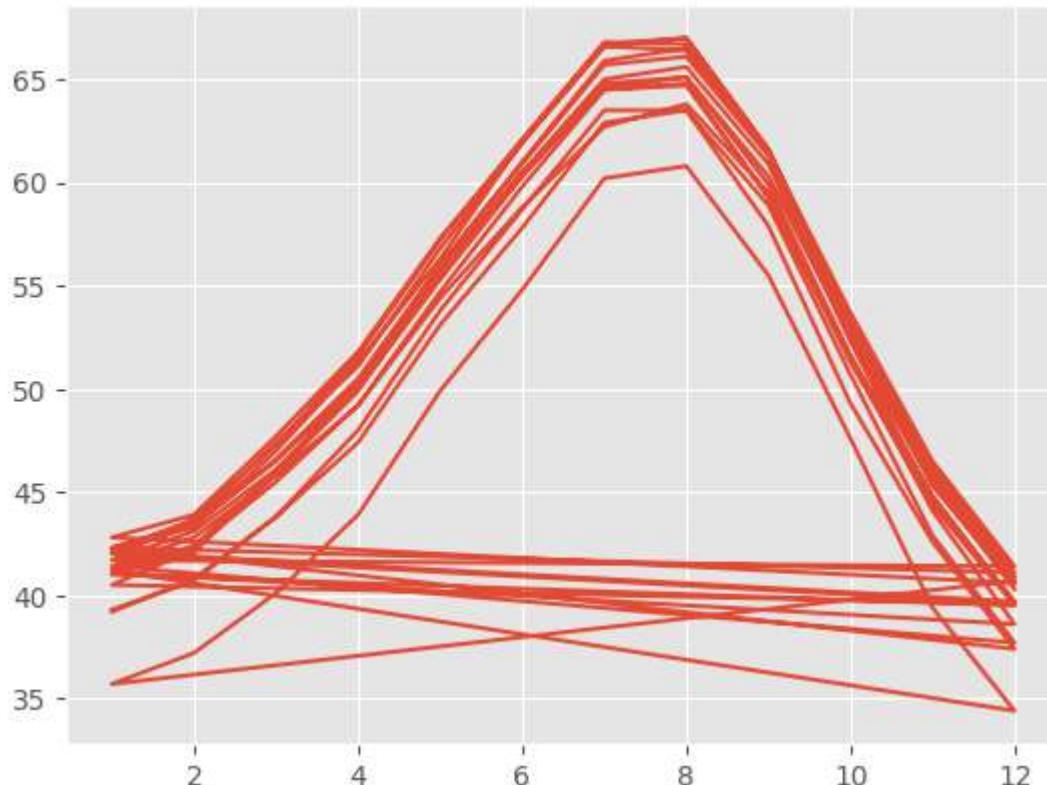
```
In [ ]: # Read the data from file using read_csv
austin_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
seattle_weather = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course
```

```
In [ ]: 'grayscale'
```

```
Out[ ]: 'grayscale'
```

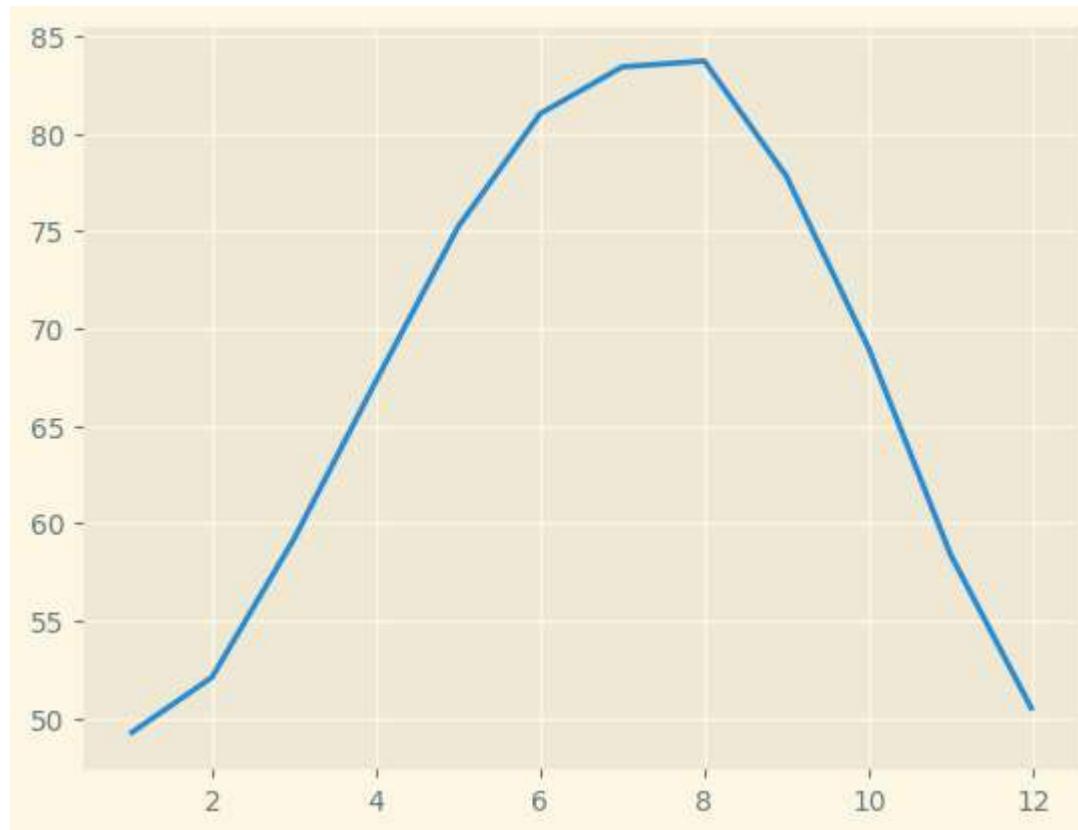
Switching between styles

```
In [ ]: # Use the "ggplot" style and create new Figure/Axes
plt.style.use('ggplot')
fig, ax = plt.subplots()
ax.plot(seattle_weather["DATE"], seattle_weather["MLY-TAVG-NORMAL"])
plt.show()
```



```
In [ ]: # Use the "Solarize_Light2" style and create new Figure/Axes
plt.style.use('Solarize_Light2')
```

```
fig, ax = plt.subplots()
ax.plot(austin_weather["DATE"], austin_weather["MLY-TAVG-NORMAL"])
plt.show()
```



Saving a file several times

```
In [ ]: # Show the figure
plt.show()

# Save as a PNG file
fig.savefig('my_figure.png')
# Save as a PNG file with 300 dpi

fig.savefig('my_figure_300dpi.png', dpi=300)
```

Save a figure with different sizes

```
In [ ]: # Set figure dimensions and save as a PNG
fig.set_size_inches([3,5])
fig.savefig('figure_3_5.png')

# Set figure dimensions and save as a PNG
fig.set_size_inches([5,3])
fig.savefig('figure_5_3.png')
```

Unique values of a column

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

# Specify the file path using double backslashes
summer_2016_medals = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Cou

# Display the first few rows of the DataFrame
summer_2016_medals.head()
```

	Unnamed: 0		ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
0	158	62	Giovanni Abagnale	M	21.0	198.0	90.0		Italy	ITA	2016 Summer	2016	Summer	Rio de Janeiro	Rowing	Rowing Men's Coxless Pairs	Bronze
1	161	65	Patimat Abakarova	F	21.0	165.0	49.0	Azerbaijan	AZE	2016 Summer	2016	Summer	Rio de Janeiro	Taekwondo	Taekwondo Women's Flyweight	Bronze	
2	175	73	Luc Abalo	M	31.0	182.0	86.0	France	FRA	2016 Summer	2016	Summer	Rio de Janeiro	Handball	Handball Men's Handball	Silver	
3	450	250	Saeid Morad Abdevali	M	26.0	170.0	80.0	Iran	IRI	2016 Summer	2016	Summer	Rio de Janeiro	Wrestling	Wrestling Men's Middleweight, Greco-Roman	Bronze	
4	794	455	Denis Mikhaylovich Ablyazin	M	24.0	161.0	62.0	Russia	RUS	2016 Summer	2016	Summer	Rio de Janeiro	Gymnastics	Gymnastics Men's Team All-Around	Silver	

```
In [ ]: # Extract the "Sport" column
sports_column = summer_2016_medals['Sport']

# Find the unique values of the "Sport" column
sports = sports_column.unique()
```

```
# Print out the unique sports values
print(sports)
```

```
['Rowing' 'Taekwondo' 'Handball' 'Wrestling' 'Gymnastics' 'Swimming'
 'Basketball' 'Boxing' 'Volleyball' 'Athletics' 'Rugby Sevens' 'Judo'
 'Rhythmic Gymnastics' 'Weightlifting' 'Equestrianism' 'Badminton'
 'Water Polo' 'Football' 'Fencing' 'Shooting' 'Sailing' 'Beach Volleyball'
 'Canoeing' 'Hockey' 'Cycling' 'Tennis' 'Diving' 'Table Tennis'
 'Triathlon' 'Archery' 'Synchronized Swimming' 'Modern Pentathlon'
 'Trampolining' 'Golf']
```

Automate your visualization

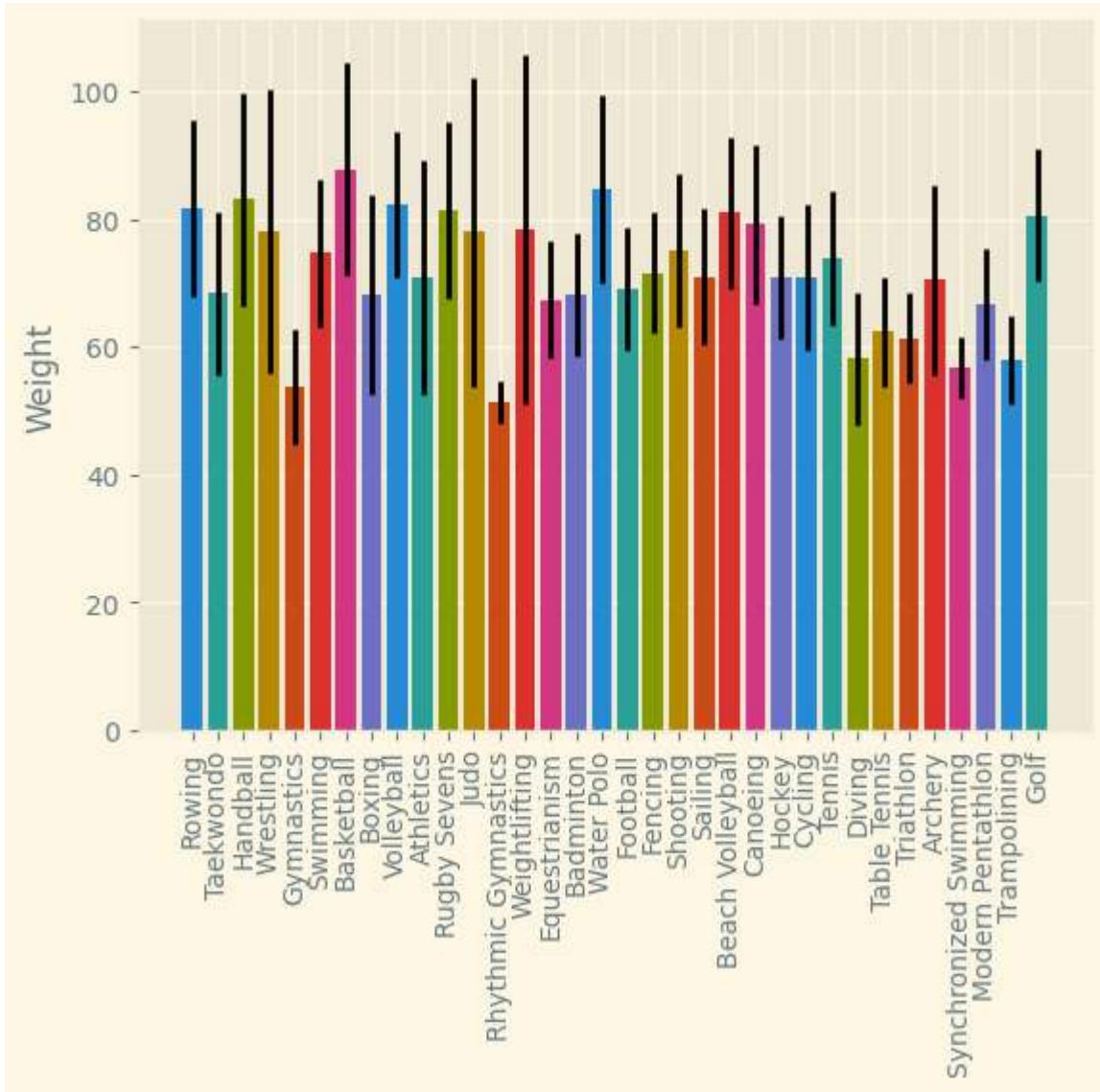
```
In [ ]: fig, ax = plt.subplots()
```

```
# Loop over the different sports branches
for sport in sports:
    # Extract the rows only for this sport
    sport_df = summer_2016_medals[summer_2016_medals['Sport'] == sport]
    # Add a bar for the "Weight" mean with std y error bar
    ax.bar(sport, sport_df["Weight"].mean(), yerr=sport_df["Weight"].std())

ax.set_ylabel("Weight")
ax.set_xticklabels(sports, rotation=90)

# Save the figure to file
fig.savefig('sports_weights.png')
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_18080\1401696868.py:11: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(sports, rotation=90)
```



# 11. Visualizing Time Series Data in Python

## Chapter 1 - Line Plots

You will learn how to leverage basic plottings tools in Python, and how to annotate and personalize your time series plots. By the end of this chapter, you will be able to take any static dataset and produce compelling plots of your data.

### Load your time series data

The most common way to import time series data in Python is by using the pandas library. You can use the `read_csv()` from pandas to read the contents of a file into a DataFrame. This can be achieved using the following command:

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

In [ ]: discoveries = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Dat
discoveries.head(5)

Out[ ]:      date   Y
 0  1860-01-1  5
 1  1861-01-1  3
 2  1862-01-1  0
 3  1863-01-1  2
 4  1864-01-1  0
```

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

When working with time series data in pandas, any date information should be formatted as a `datetime64` type. Therefore, it is important to check that the columns containing the date information are of the correct type. You can check the type of each column in a DataFrame by using the `.dtypes` attribute. Fortunately, if your date columns come as strings, epochs, etc... you can use the `to_datetime()` function to convert them to the appropriate `datetime64` type:

```
In [ ]: print(discoveries.dtypes)
```

```
date    object
Y      int64
dtype: object
```

```
In [ ]: # Convert the date column to a datetime type
discoveries['date'] = pd.to_datetime(discoveries['date'])
```

```
In [ ]: print(discoveries.dtypes)
```

```
date    datetime64[ns]
Y      int64
dtype: object
```

### Your first plot!

Let's take everything you have learned so far and plot your first time series plot. You will set the groundwork by producing a time series plot of your data and labeling the axes of your plot, as this makes the plot more readable and interpretable for the intended audience.

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 [ ]: import matplotlib.pyplot as plt

print(discoveries.head(2))
# Set the date column as the index of your DataFrame discoveries
discoveries = discoveries.set_index('date')

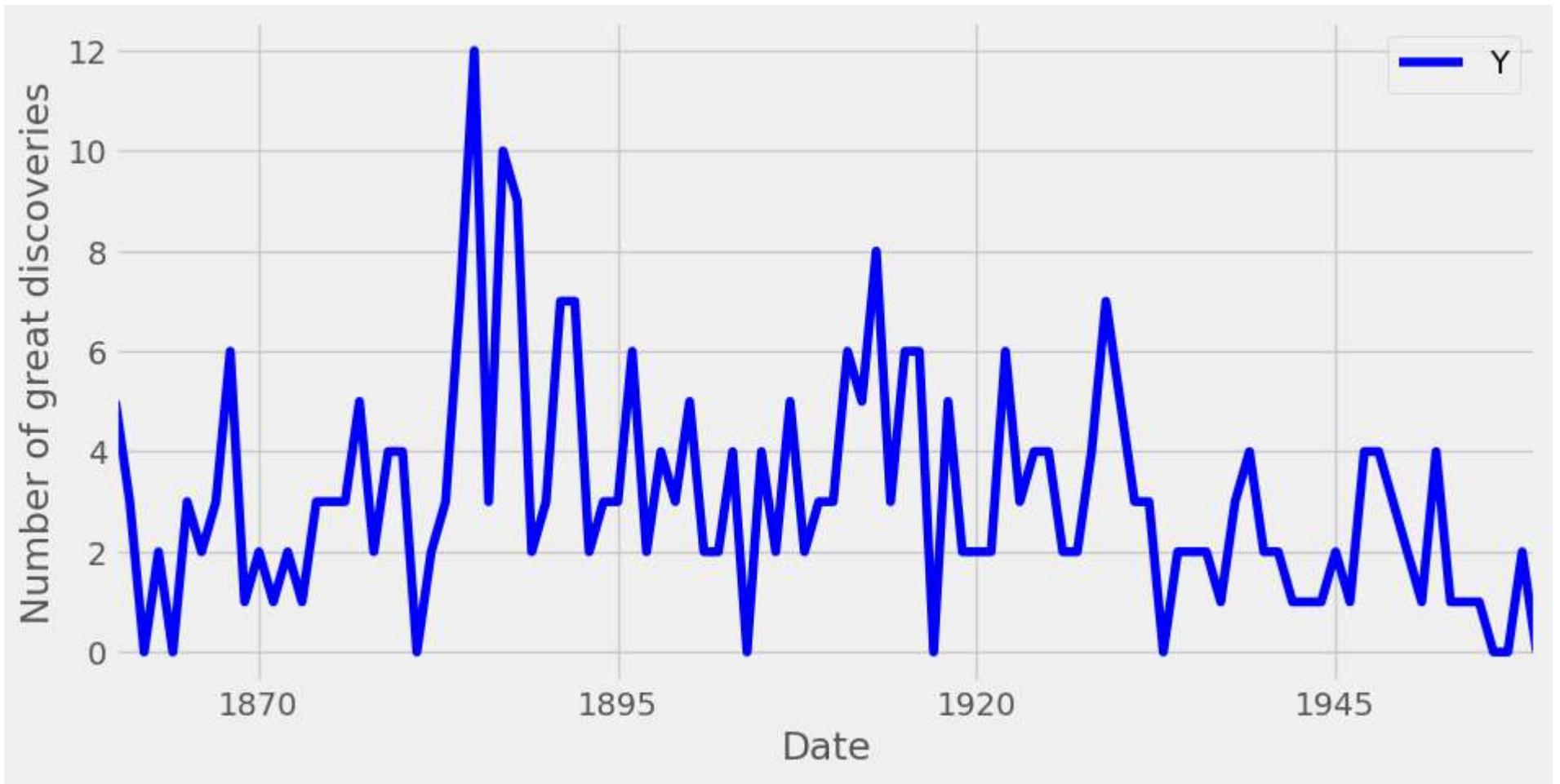
# Plot the time series in your DataFrame
ax = discoveries.plot(color='blue')

# Specify the x-axis label in your plot
ax.set_xlabel('Date')

# Specify the y-axis label in your plot
ax.set_ylabel('Number of great discoveries')

# Show plot
plt.show();
```

	date	Y
0	1860-01-01	5
1	1861-01-01	3



### Specify plot styles

The matplotlib library also comes with a number of built-in stylesheets that allow you to customize the appearance of your plots. To use a particular style sheet for your plots, you can use the command `plt.style.use(your_stylesheet)` where `your_stylesheet` is the name of the style sheet.

In order to see the list of available style sheets that can be used, you can use the command `print(plt.style.available)`. For the rest of this course, we will use the awesome `fivethirtyeight` style sheet.

Import `matplotlib.pyplot` using its usual alias `plt`.

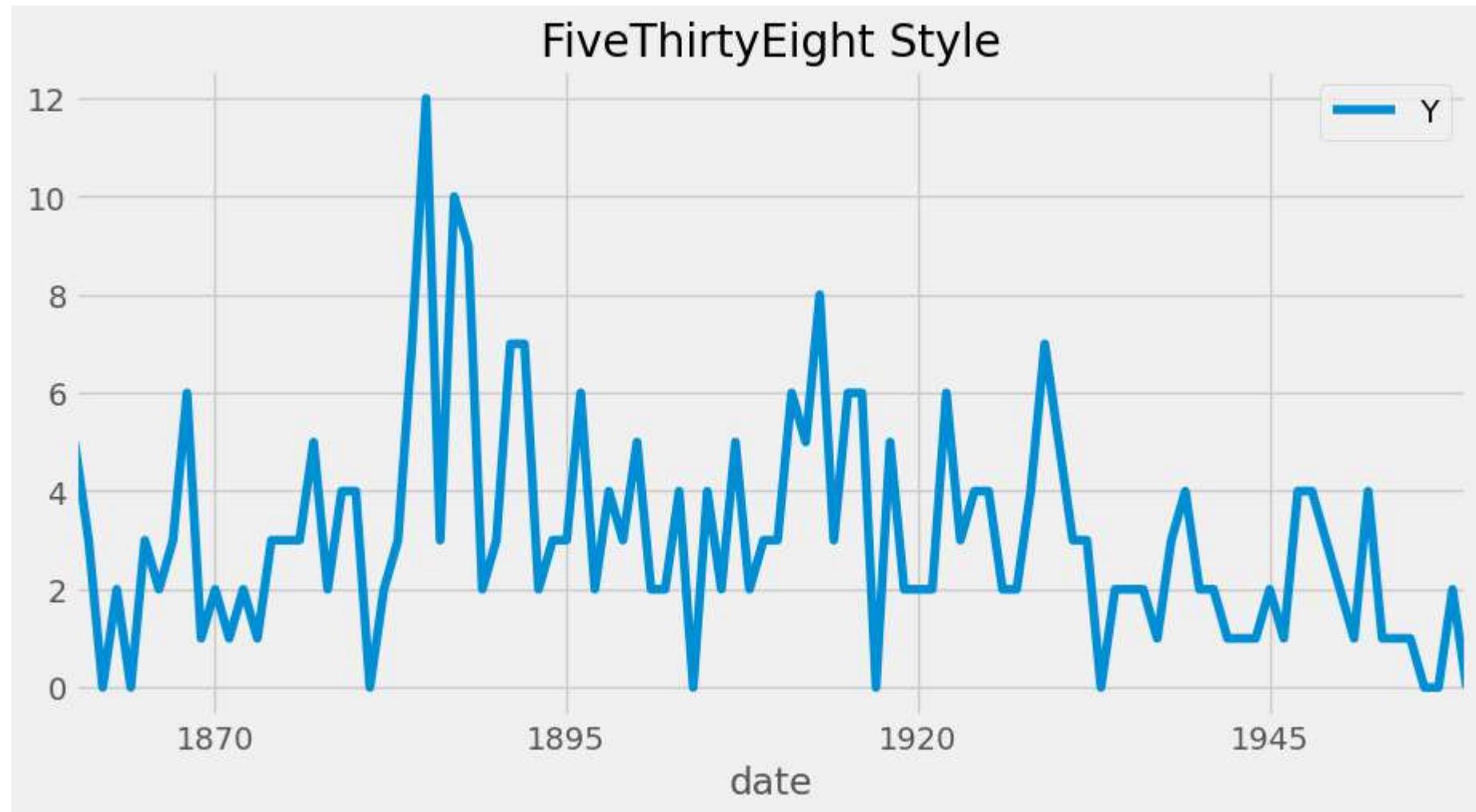
Use the `fivethirtyeight` style sheet to plot a line plot of the discoveries data.

In [ ]:

```
# Import the matplotlib.pyplot sub-module
import matplotlib.pyplot as plt

# Use the fivethirtyeight style
plt.style.use('fivethirtyeight')

# Plot the time series
ax1 = discoveries.plot()
ax1.set_title('FiveThirtyEight Style')
plt.show()
```



Use the `ggplot` style sheet to plot a line plot of the discoveries data.

Set the title of your second plot as 'ggplot Style'.

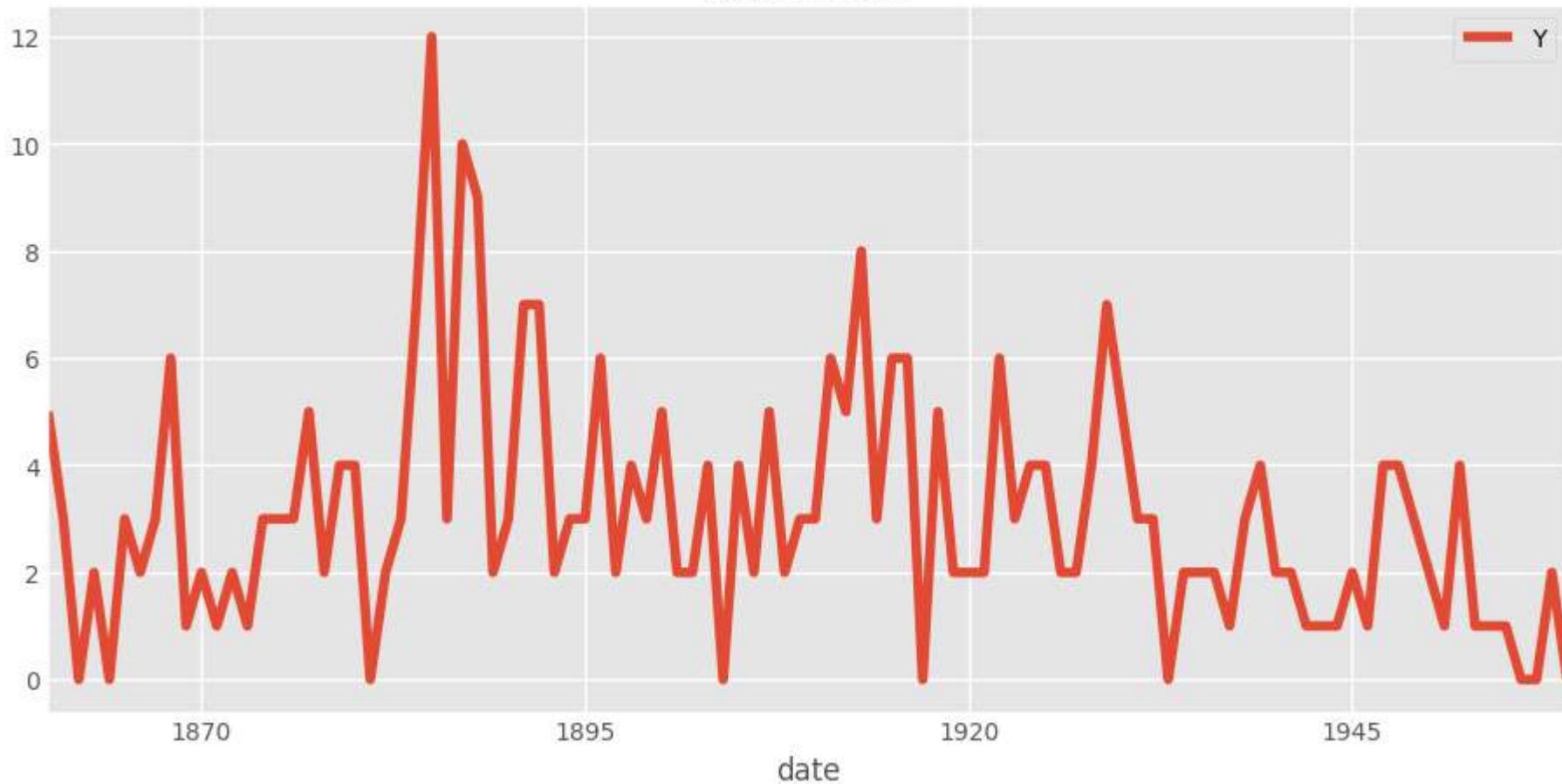
In [ ]:

```
# Import the matplotlib.pyplot sub-module
import matplotlib.pyplot as plt

# Use the ggplot style
plt.style.use('ggplot')
ax2 = discoveries.plot()

# Set the title
ax2.set_title('ggplot Style')
plt.show()
```

## ggplot Style



```
In [ ]: # Plot a Line chart of the discoveries DataFrame using the specified arguments
```

```
ax = discoveries.plot(color='blue', figsize=(8, 3), linewidth=2, fontsize=6)
```

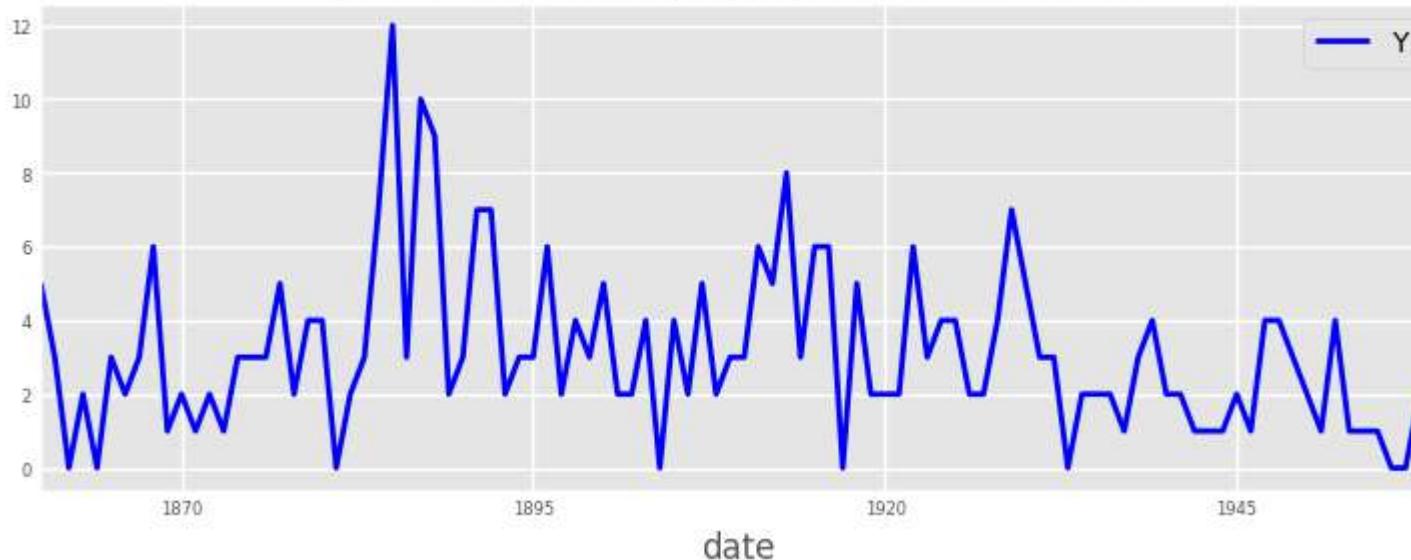
```
# Specify the title in your plot
```

```
ax.set_title('Number of great inventions and scientific discoveries from 1860 to 1959', fontsize=8)
```

```
# Show plot
```

```
plt.show()
```

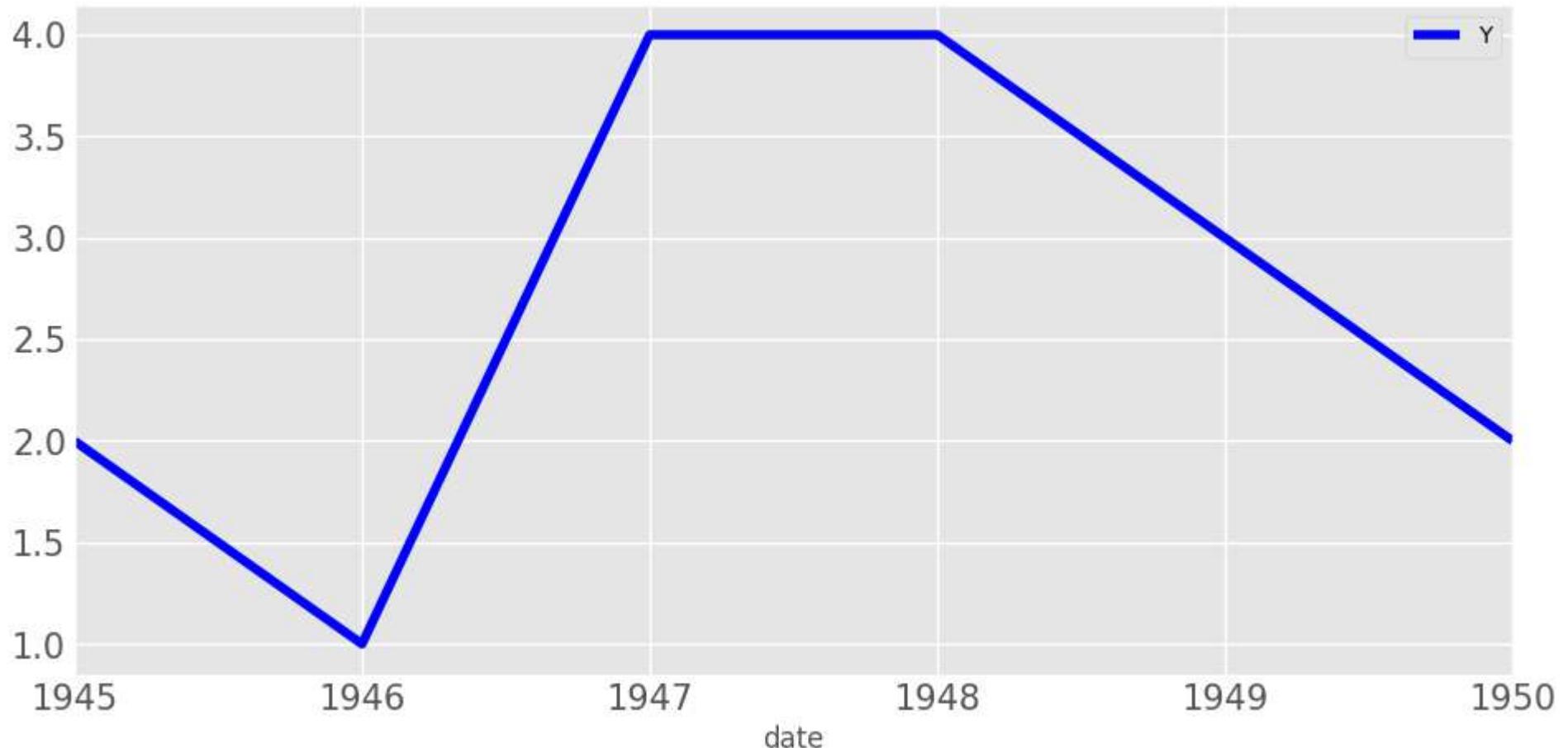
Number of great inventions and scientific discoveries from 1860 to 1959



```
In [ ]: # Select the subset of data between 1945 and 1950
discoveries_subset_1 = discoveries['1945-01-01':'1950-01-01']

# Plot the time series in your DataFrame as a blue area chart
ax = discoveries_subset_1.plot(color='blue', fontsize=15)

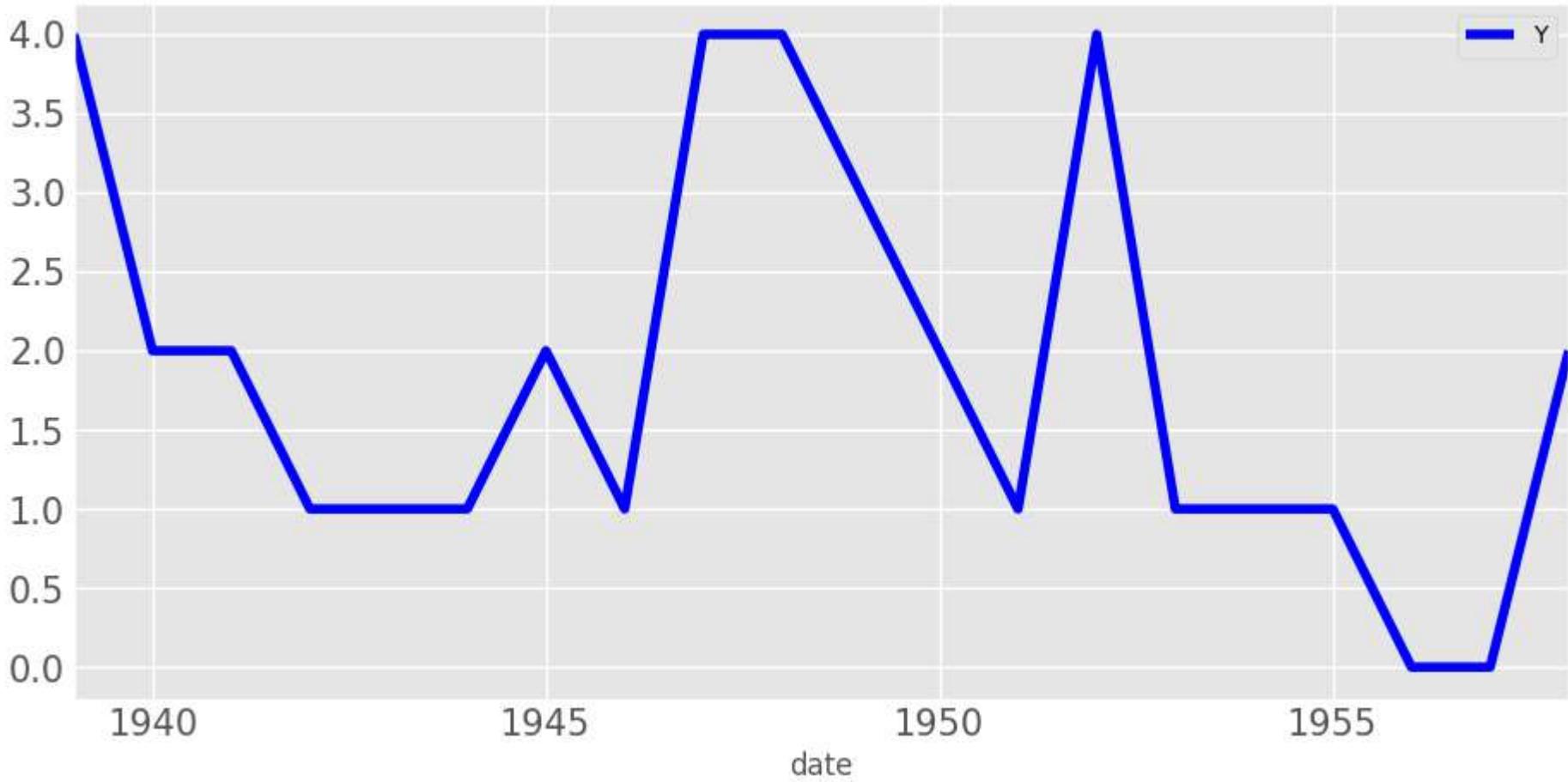
# Show plot
plt.show();
```



```
In [ ]: # Select the subset of data between 1939 and 1958
discoveries_subset_2 = discoveries['1939-01-01':'1958-01-01']

# Plot the time series in your DataFrame as a blue area chart
ax = discoveries_subset_2.plot(color='blue', fontsize=15)

# Show plot
plt.show();
```



### Add vertical and horizontal markers

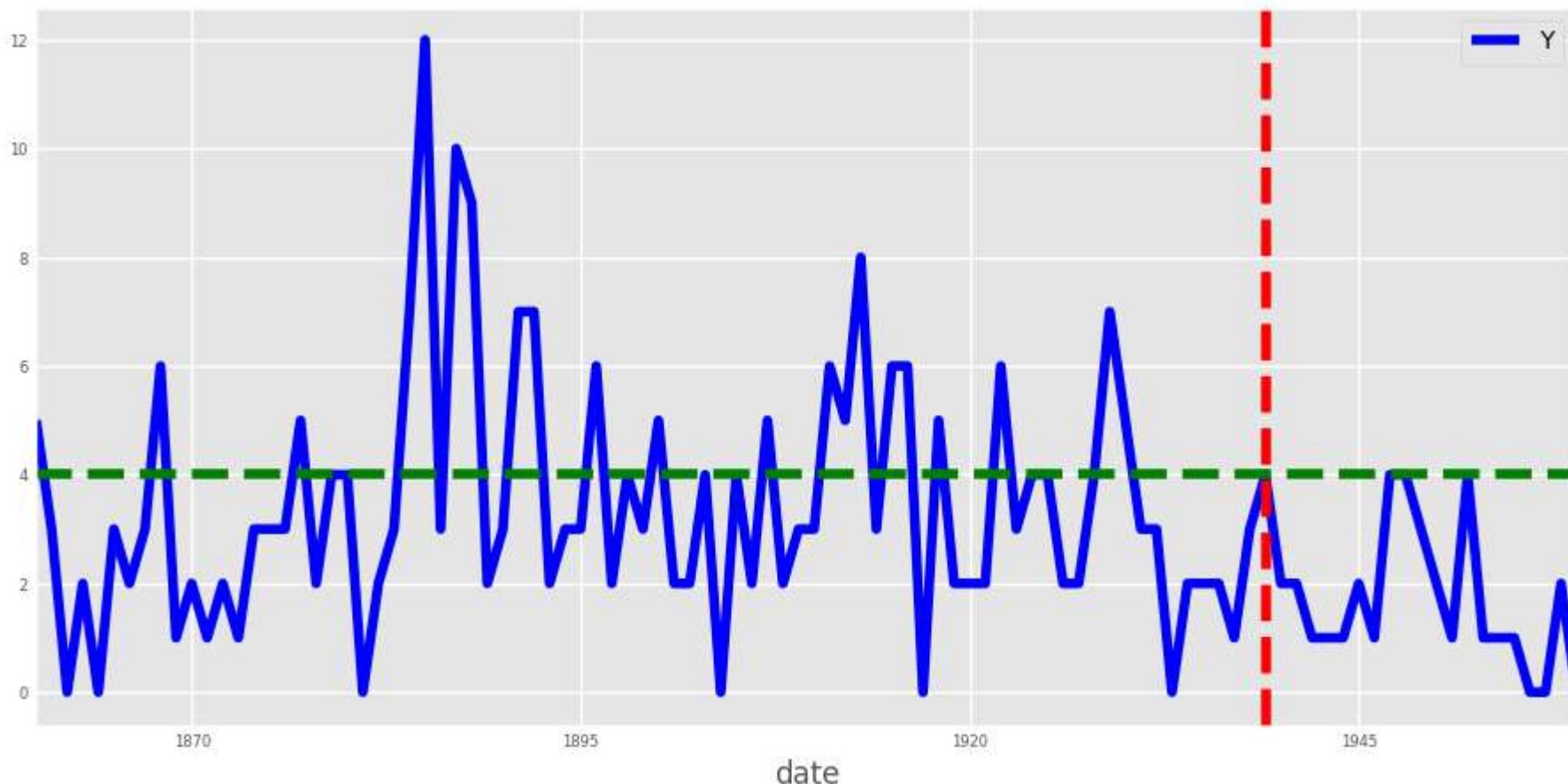
Additional annotations can help further emphasize specific observations or events. Here, you will learn how to highlight significant events by adding markers at specific timestamps of your time series plot. The matplotlib library makes it possible to draw vertical and horizontal lines to identify particular dates.

Recall that the index of the discoveries DataFrame are of the datetime type, so the x-axis values of a plot will also contain dates, and it is possible to directly input a date when annotating your plots with vertical lines. For example, a vertical line at January 1, 1945 can be added to your plot by using the command: `ax.axvline('1945-01-01', linestyle='--')`.

```
In [ ]: # Plot your the discoveries time series
ax = discoveries.plot(color='blue', fontsize=6)

# Add a red vertical line
ax.axvline('1939-01-01', color='red', linestyle='--')
```

```
# Add a green horizontal line  
ax.axhline(4, color='green', linestyle='--')  
  
plt.show();
```



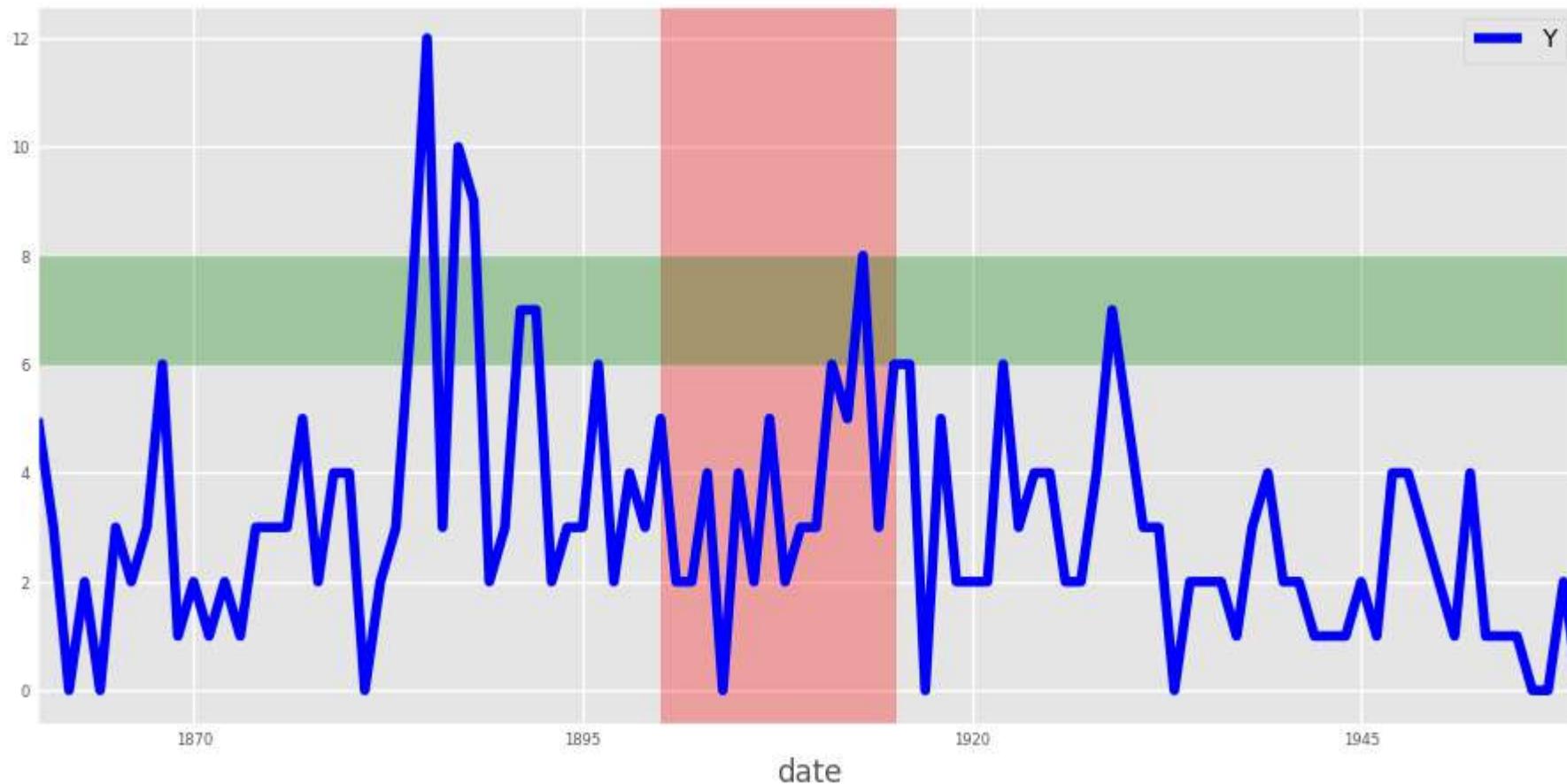
### Add shaded regions to your plot

When plotting time series data in Python, it is also possible to highlight complete regions of your time series plot. In order to add a shaded region between January 1, 1936 and January 1, 1950, you can use the command: `ax.axvspan('1936-01-01', '1950-01-01', color='red', alpha=0.5)`.

Here we specified the overall transparency of the region by using the `alpha` argument (where 0 is completely transparent and 1 is full color).

```
In [ ]: # Plot your the discoveries time series  
ax = discoveries.plot(color='blue', fontsize=6)
```

```
# Add a vertical red shaded region  
ax.axvspan('1900-01-01', '1915-01-01', color='red', alpha=0.3)  
  
# Add a horizontal green shaded region  
ax.axhspan(6, 8, color='green', alpha=0.3)  
  
plt.show();
```



## Chapter 2 - Summary Statistics and Diagnostics

In [ ]:

```
# Import pandas  
import pandas as pd  
import matplotlib.pyplot as plt
```

## Find missing values

In the field of Data Science, it is common to encounter datasets with missing values. This is especially true in the case of time series data, where missing values can occur if a measurement fails to record the value at a specific timestamp. To count the number of missing values in a DataFrame called df that contains time series data, you can use the command:

```
missing_values = df.isnull().sum().
```

```
In [ ]: # Read in the file content in a DataFrame called co2_levels
co2_levels = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course Data

# Display the first seven lines of the DataFrame
print(co2_levels.head(7))
```

```
datestamp      co2
0 1958-03-29  316.1
1 1958-04-05  317.3
2 1958-04-12  317.6
3 1958-04-19  317.5
4 1958-04-26  316.4
5 1958-05-03  316.9
6 1958-05-10    NaN
```

```
In [ ]: # Set datestamp column as index
co2_levels = co2_levels.set_index('datestamp')

# Print out the number of missing values
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.

Recall from the previous exercise that co2\_levels has 59 missing values.

```
In [ ]: # Impute missing values with the next valid observation  
co2_levels = co2_levels.fillna(method='bfill')  
  
# Print out the number of missing values  
print(co2_levels.isnull().sum())
```

```
co2      0  
dtype: int64
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_39800\2624947007.py:2: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
```

```
co2_levels = co2_levels.fillna(method='bfill')
```

### Display rolling averages (aggregating data)

It is also possible to visualize rolling averages of the values in your time series. This is equivalent to "smoothing" your data, and can be particularly useful when your time series contains a lot of noise or outliers. For a given DataFrame df, you can obtain the rolling average of the time series by using the command:

```
df_mean = df.rolling(window=12).mean()
```

The window parameter should be set according to the granularity of your time series. For example, if your time series contains daily data and you are looking for rolling values over a whole year, you should specify the parameter to window=365. In addition, it is easy to get rolling values for other other metrics, such as the standard deviation (.std()) or variance (.var()).

```
In [ ]: # Compute the 52 weeks rolling mean of the co2_levels DataFrame  
ma = co2_levels.rolling(window=52).mean()  
ma
```

Out[ ]:

co2

datestamp	
1958-03-29	NaN
1958-04-05	NaN
1958-04-12	NaN
1958-04-19	NaN
1958-04-26	NaN
...	...
2001-12-01	370.738462
2001-12-08	370.761538
2001-12-15	370.798077
2001-12-22	370.832692
2001-12-29	370.865385

2284 rows × 1 columns

In [ ]:

```
# Compute the 52 weeks rolling standard deviation of the co2_levels DataFrame
mstd = co2_levels.rolling(window=52).std()
mstd
```

Out[ ]:

co2

**datestamp**

1958-03-29	NaN
1958-04-05	NaN
1958-04-12	NaN
1958-04-19	NaN
1958-04-26	NaN
...	...
2001-12-01	1.932199
2001-12-08	1.925490
2001-12-15	1.915212
2001-12-22	1.907541
2001-12-29	1.904060

2284 rows × 1 columns

In [ ]:

```
# Add the upper bound column to the ma DataFrame  
ma['upper'] = ma['co2'] + (mstd['co2'] * 2)
```

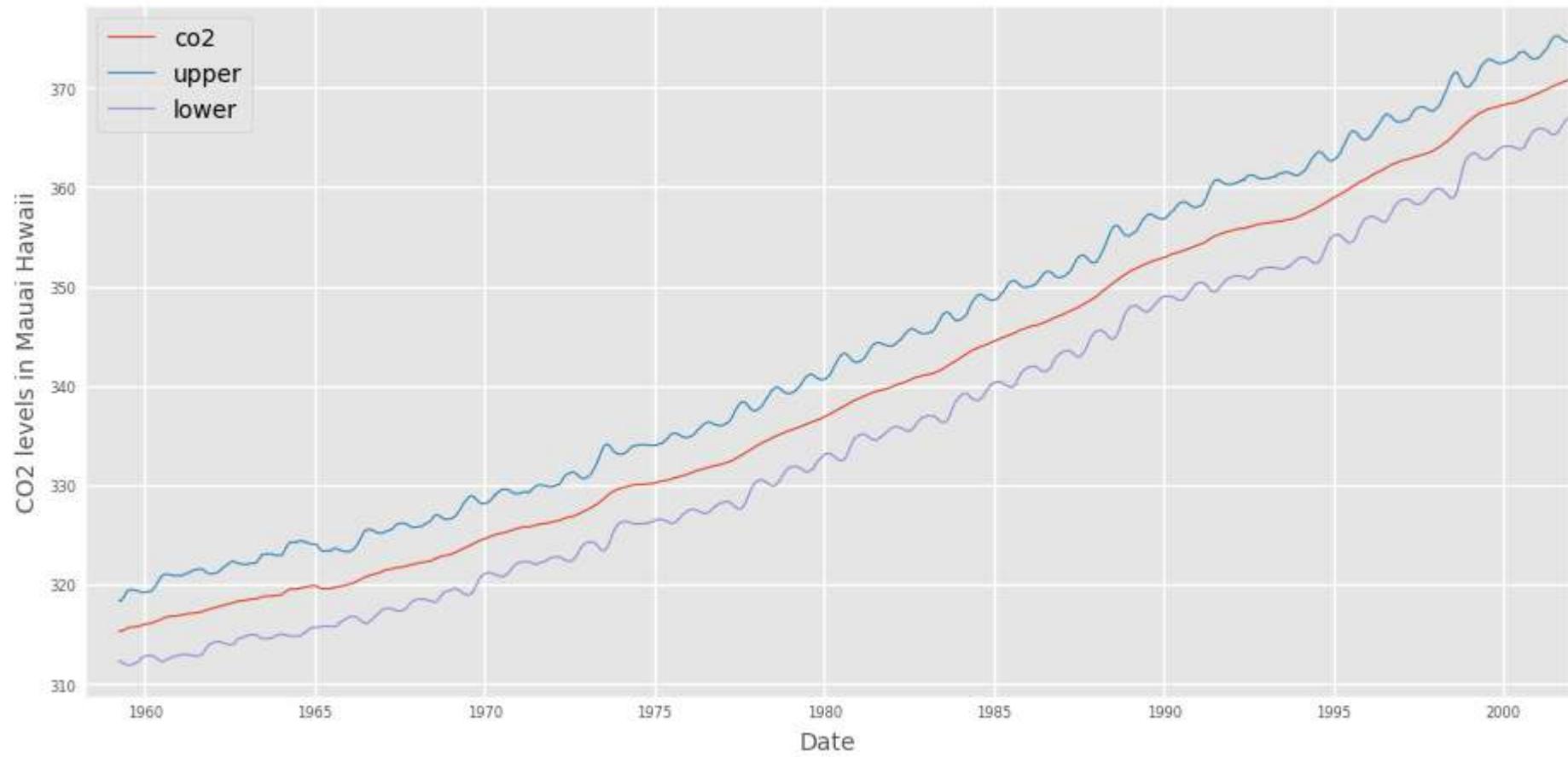
In [ ]:

```
# Add the lower bound column to the ma DataFrame  
ma['lower'] = ma['co2'] - (mstd['co2'] * 2)
```

In [ ]:

```
# Plot the content of the ma DataFrame  
ax = ma.plot(linewidth=0.8, fontsize=6)  
  
# Specify labels, legend, and show the plot  
ax.set_xlabel('Date', fontsize=10)  
ax.set_ylabel('CO2 levels in Mauai Hawaii', fontsize=10)  
ax.set_title('Rolling mean and variance of CO2 levels\nin Mauai Hawaii from 1958 to 2001', fontsize=10)  
plt.show();
```

Rolling mean and variance of CO2 levels  
in Mauai Hawaii from 1958 to 2001



### Computing aggregate values of your time series

```
In [ ]: co2_levels.index
```

```
Out[ ]: DatetimeIndex(['1958-03-29', '1958-04-05', '1958-04-12', '1958-04-19',
       '1958-04-26', '1958-05-03', '1958-05-10', '1958-05-17',
       '1958-05-24', '1958-05-31',
       ...
       '2001-10-27', '2001-11-03', '2001-11-10', '2001-11-17',
       '2001-11-24', '2001-12-01', '2001-12-08', '2001-12-15',
       '2001-12-22', '2001-12-29'],
      dtype='datetime64[ns]', name='datestamp', length=2284, freq=None)
```

```
In [ ]: co2_levels.index.month
```

```
Out[ ]: Index([ 3,  4,  4,  4,  4,  5,  5,  5,  5,  5,
   ...
   10, 11, 11, 11, 11, 12, 12, 12, 12, 12],
  dtype='int32', name='datestamp', length=2284)
```

```
In [ ]: co2_levels.index.year
```

```
Out[ ]: Index([1958, 1958, 1958, 1958, 1958, 1958, 1958, 1958, 1958,
   ...
   2001, 2001, 2001, 2001, 2001, 2001, 2001, 2001, 2001],
  dtype='int32', name='datestamp', length=2284)
```

## 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: 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().

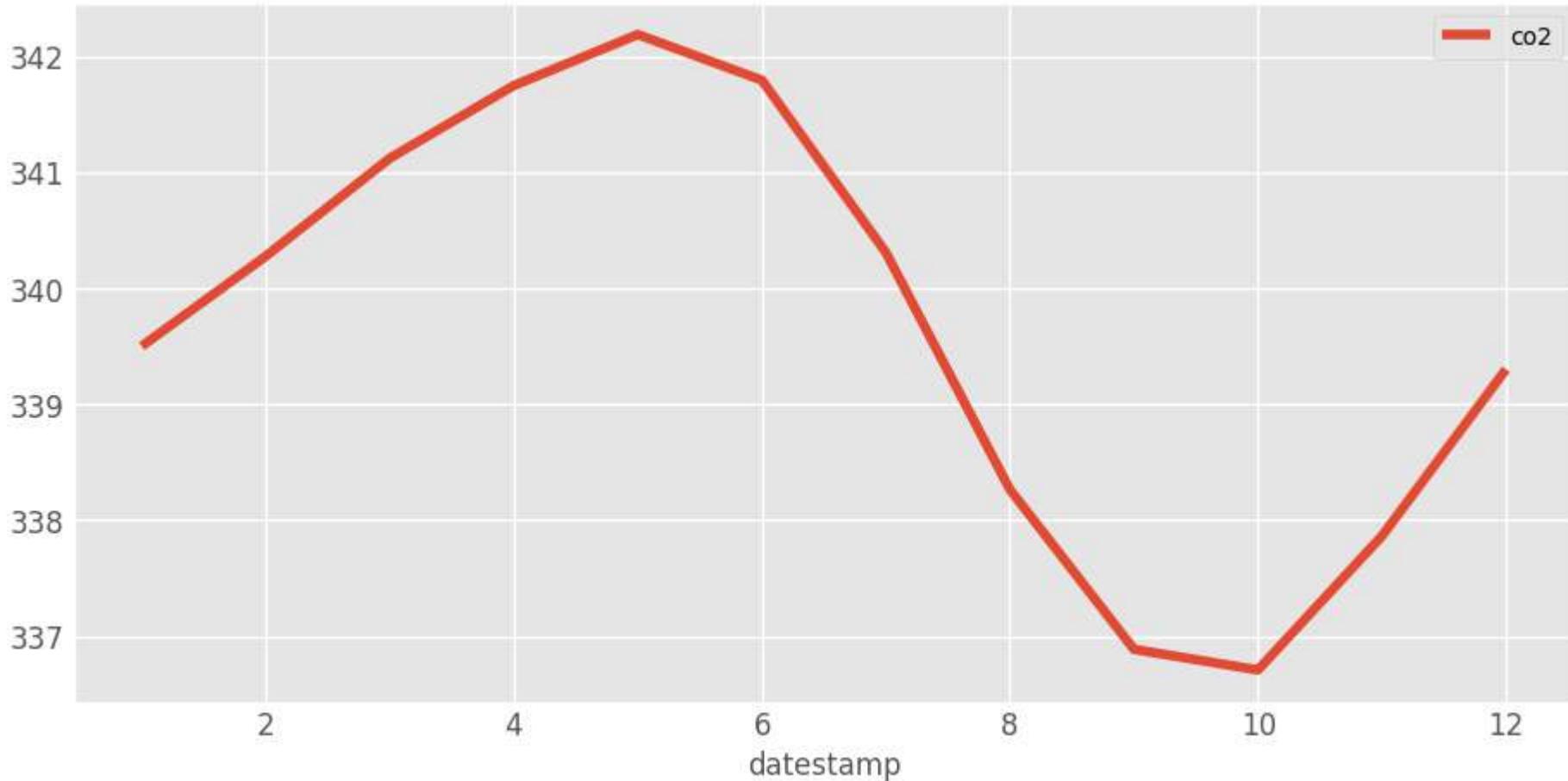
```
In [ ]: # Get month for each dates in the index of co2_levels
index_month = co2_levels.index.month

# Compute the mean CO2 Levels for each month of the year
# Using the groupby and mean functions from the pandas library,
# compute the monthly mean CO2 levels in the co2_levels DataFrame
# and assign that to a new DataFrame called mean_co2_levels_by_month.
mean_co2_levels_by_month = co2_levels.groupby(index_month).mean()

# Plot the mean CO2 Levels for each month of the year
mean_co2_levels_by_month.plot(fontsize=12)

# Specify the fontsize on the legend
plt.legend(fontsize=10)

# Show plot
plt.show();
```



By plotting the mean CO2 levels data for each month, you can see how CO2 levels are high during the summer months, and lower during the winter months. This is because of the increased sunlight and CO2 production by plants!

### Compute numerical summaries

You have learnt how to display and annotate time series data in multiple ways, but it is also informative to collect summary statistics of your data. Being able to achieve this task will allow you to share and discuss statistical properties of your data that can further support the plots you generate. In pandas, it is possible to quickly obtain summaries of columns in your DataFrame by using the command: `print(df.describe())`.

This will print statistics including the mean, the standard deviation, the minima and maxima and the number of observations for all numeric columns in your pandas DataFrame.

```
In [ ]: # Print out summary statistics of the co2_Levels DataFrame  
print(co2_levels.describe())  
  
# Print out the minima of the co2 column in the co2_Levels DataFrame  
print(co2_levels.co2.min())  
  
# Print out the maxima of the co2 column in the co2_Levels DataFrame  
print(co2_levels.co2.max())
```

```
      co2  
count  2284.000000  
mean   339.657750  
std    17.100899  
min    313.000000  
25%   323.975000  
50%   337.700000  
75%   354.500000  
max    373.900000  
313.0  
373.9
```

It looks like the CO2 levels data has an average value of 339.657750.

## Boxplots and Histograms

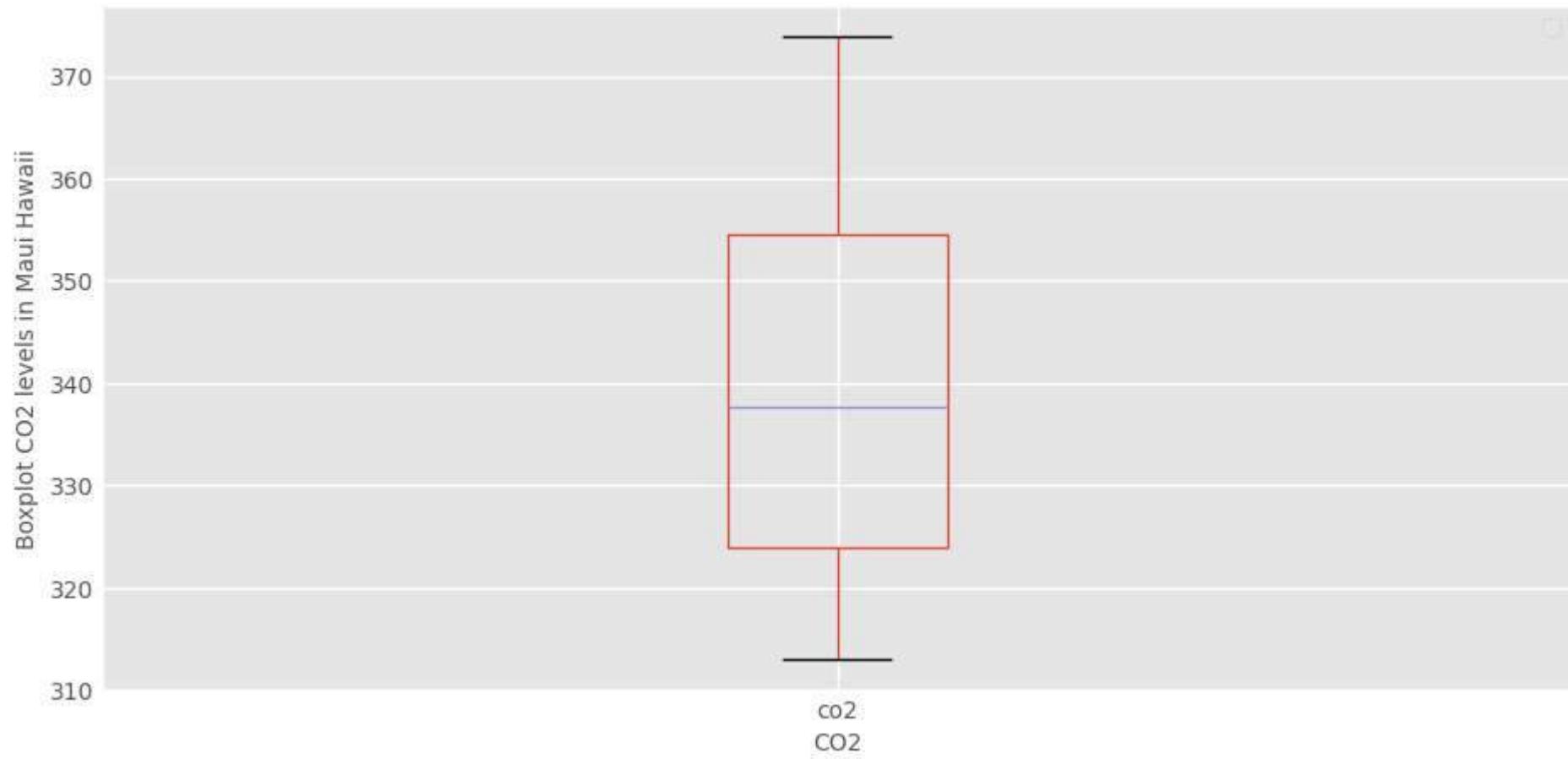
Boxplots represent a graphical rendition of the minimum, median, quartiles, and maximum of your data. You can generate a boxplot by calling the .boxplot() method on a DataFrame.

Another method to produce visual summaries is by leveraging histograms, which allow you to inspect the data and uncover its underlying distribution, as well as the presence of outliers and overall spread. An example of how to generate a histogram is shown below: ax = co2\_levels.plot(kind='hist', bins=100).

Here, we used the standard .plot() method but specified the kind argument to be 'hist'. In addition, we also added the bins=100 parameter, which specifies how many intervals (i.e bins) we should cut our data into.

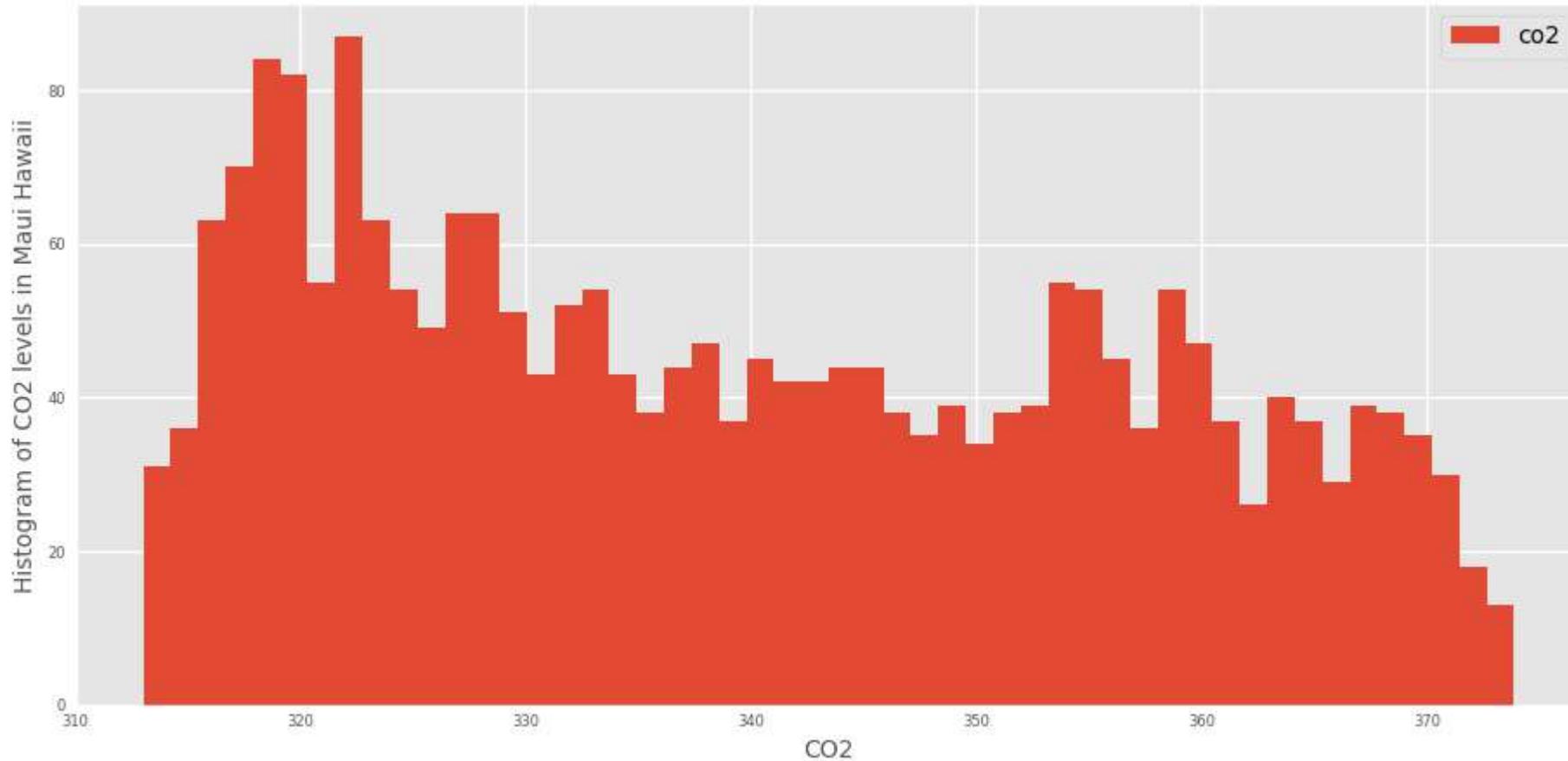
```
In [ ]: # Generate a boxplot  
ax = co2_levels.boxplot()  
  
# Set the labels and display the plot  
ax.set_xlabel('CO2', fontsize=10)  
ax.set_ylabel('Boxplot CO2 levels in Maui Hawaii', fontsize=10)  
plt.legend(fontsize=10)  
plt.show();
```

No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



```
In [ ]: # Generate a histogram
ax = co2_levels.plot(kind='hist', bins=50, fontsize=6)

# Set the labels and display the plot
ax.set_xlabel('CO2', fontsize=10)
ax.set_ylabel('Histogram of CO2 levels in Maui Hawaii', fontsize=10)
plt.legend(fontsize=10)
plt.show();
```



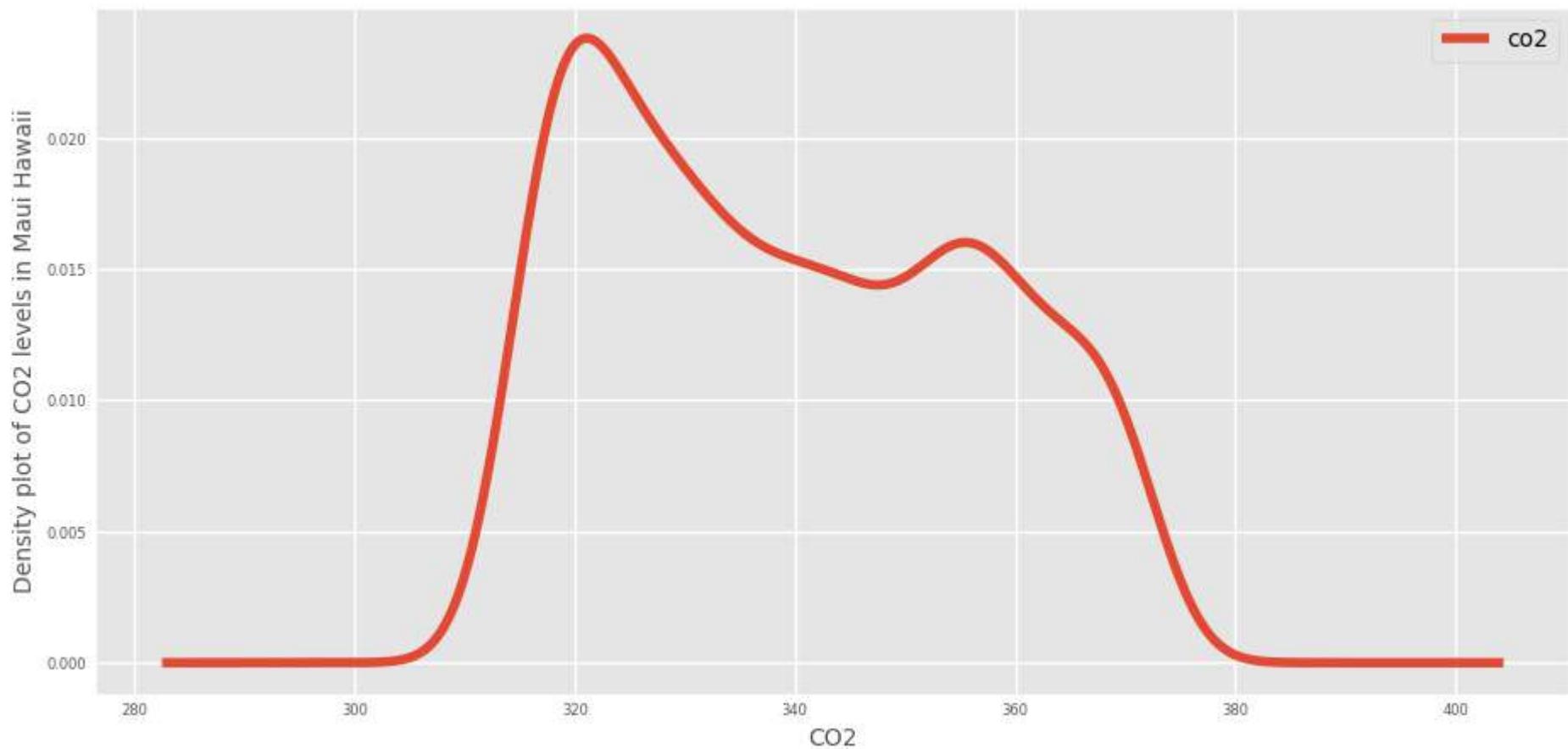
## Density plots

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: `ax = df.plot(kind='density', linewidth=2)`.

The standard `.plot()` method is specified with the `kind` argument set to `'density'`. We also specified an additional parameter `linewidth`, which controls the width of the line to be plotted.

```
In [ ]: # Display density plot of CO2 Levels values  
ax = co2_levels.plot(kind='density', linewidth=4, fontsize=6)  
  
# Annotate x-axis Labels  
ax.set_xlabel('CO2', fontsize=10)
```

```
# Annotate y-axis labels  
ax.set_ylabel('Density plot of CO2 levels in Maui Hawaii', fontsize=10)  
  
plt.show();
```



## Chapter 3 - Seasonality, Trend and Noise

Autocorrelation is measured as the correlation between a time series and a delayed copy of itself. It is used to find repeating patterns or periodic signals in time series data. Sometimes called autocovariance.

Seasonality: does the data display a clear periodic pattern?

Trend: does the data follow a consistent upwards or downwards slope?

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

```
In [ ]: # Import pandas
import pandas as pd
import matplotlib.pyplot as plt
```

## Autocorrelation in time series data

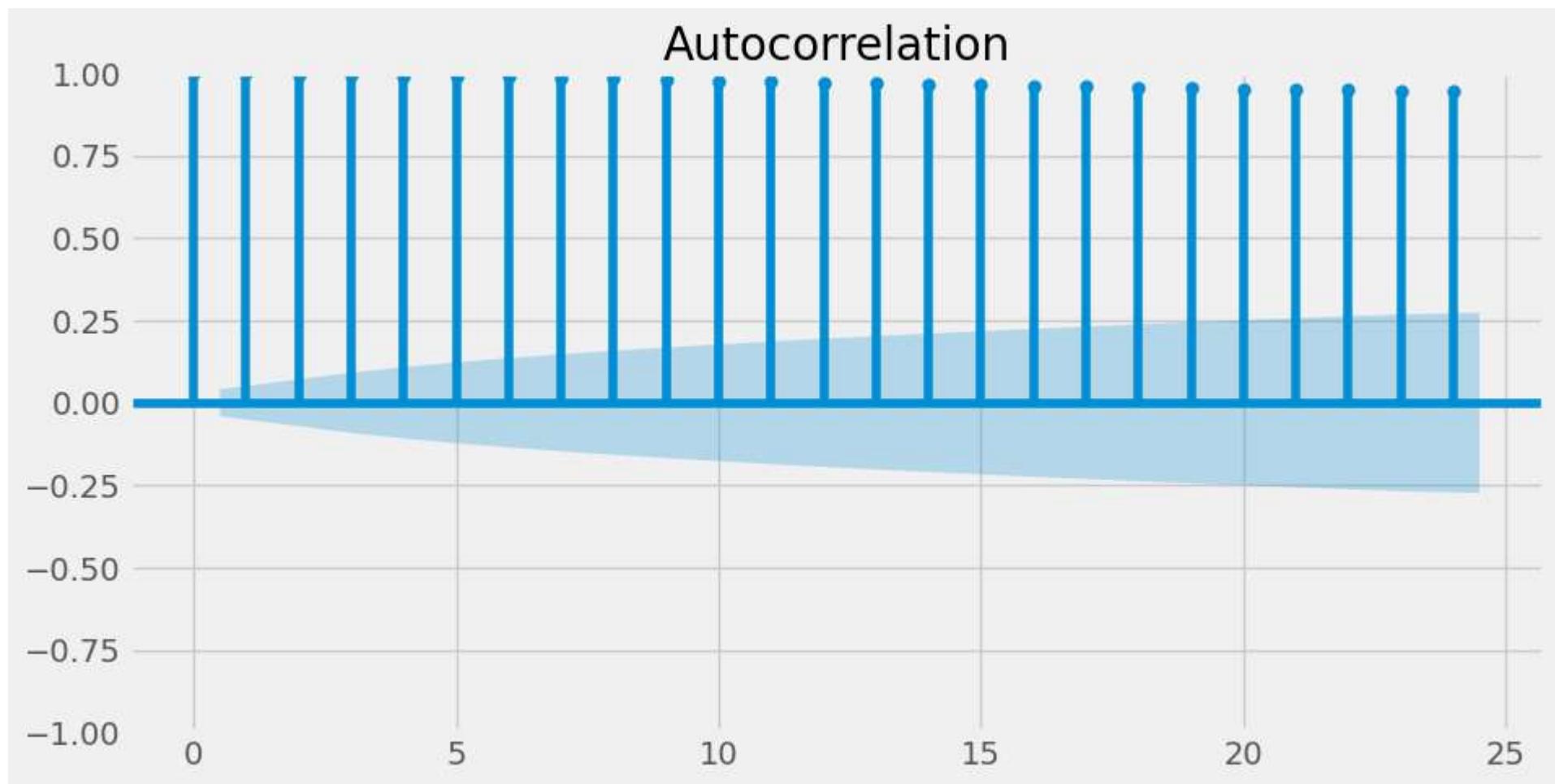
In the field of time series analysis, autocorrelation refers to the correlation of a time series with a lagged version of itself. For example, an autocorrelation of order 3 returns the correlation between a time series and its own values lagged by 3 time points.

It is common to use the autocorrelation (ACF) plot, also known as self-autocorrelation, to visualize the autocorrelation of a time-series. The `plot_acf()` function in the `statsmodels` library can be used to measure and plot the autocorrelation of a time series.

```
In [ ]: # Import required libraries
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from statsmodels.graphics import tsaplots

# Display the autocorrelation plot of your time series
fig = tsaplots.plot_acf(co2_levels['co2'], lags=24)

# Show plot
plt.show();
```



They are highly correlated and statistically significant.

#### Interpret autocorrelation plots

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 above, at which lag values do we have statistically significant partial autocorrelations?

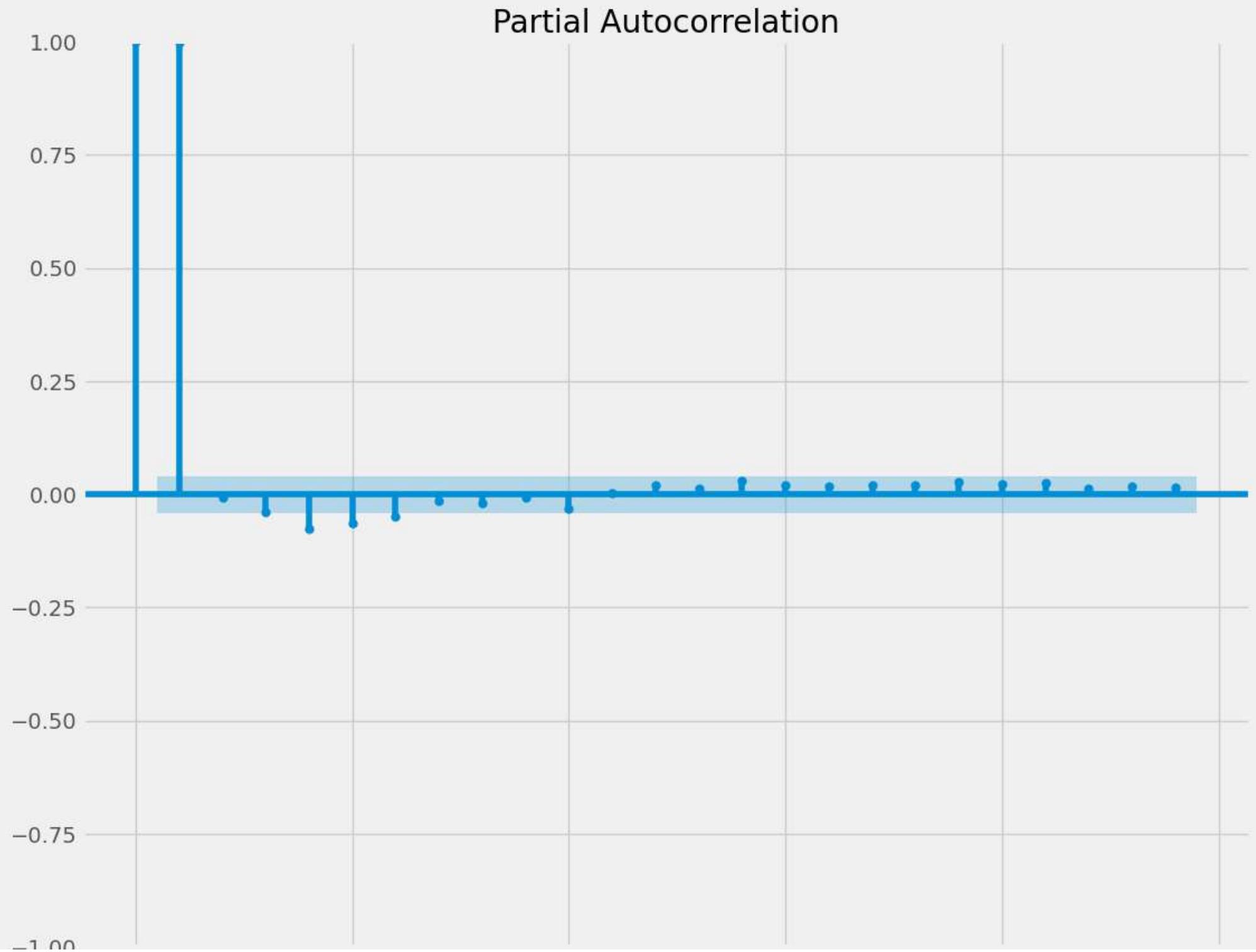
Answer: 0, 1, 3, 4, 5 and 6.

```
In [ ]: # Import required libraries
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from statsmodels.graphics import tsaplots

# Display the partial autocorrelation plot of your time series
fig = tsaplots.plot_pacf(co2_levels['co2'], lags=24)
fig.set_figheight(10)
fig.set_figwidth(12)

# Show plot
plt.show();
```

## Partial Autocorrelation



## 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: decomposition = sm.tsa.seasonal\_decompose(time\_series).

You can extract a specific component, for example seasonality, by accessing the seasonal attribute of the decomposition object.

```
In [ ]: # Import statsmodels.api as sm
import statsmodels.api as sm

# Perform time series decompositon
decomposition = sm.tsa.seasonal_decompose(co2_levels)

# Print the seasonality component
print(decomposition.seasonal)

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
...
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
Name: seasonal, Length: 2284, dtype: float64
```

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.

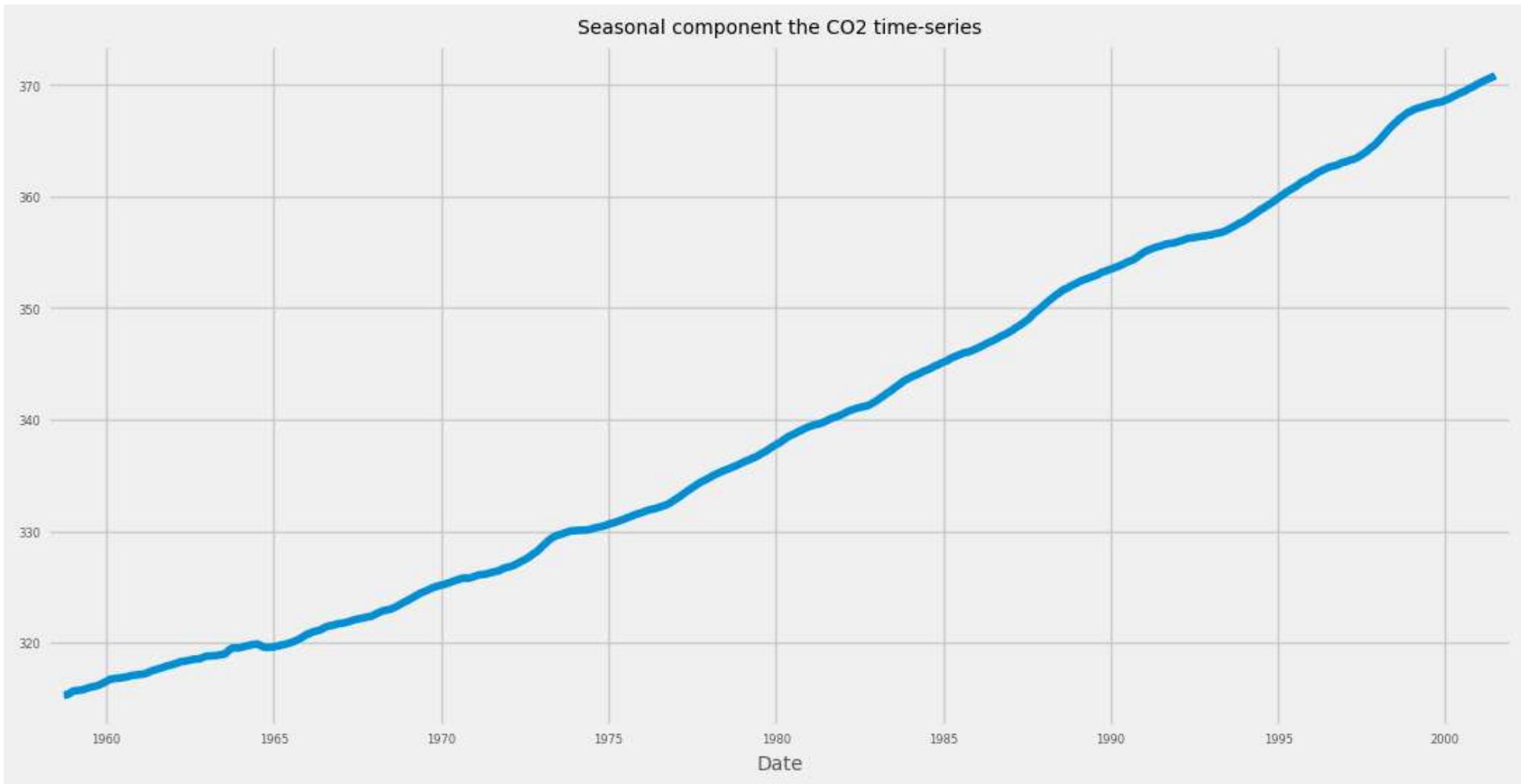
```
observed = decomposition.observed
trend = decomposition.trend
residuals = decomposition.resid
```

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

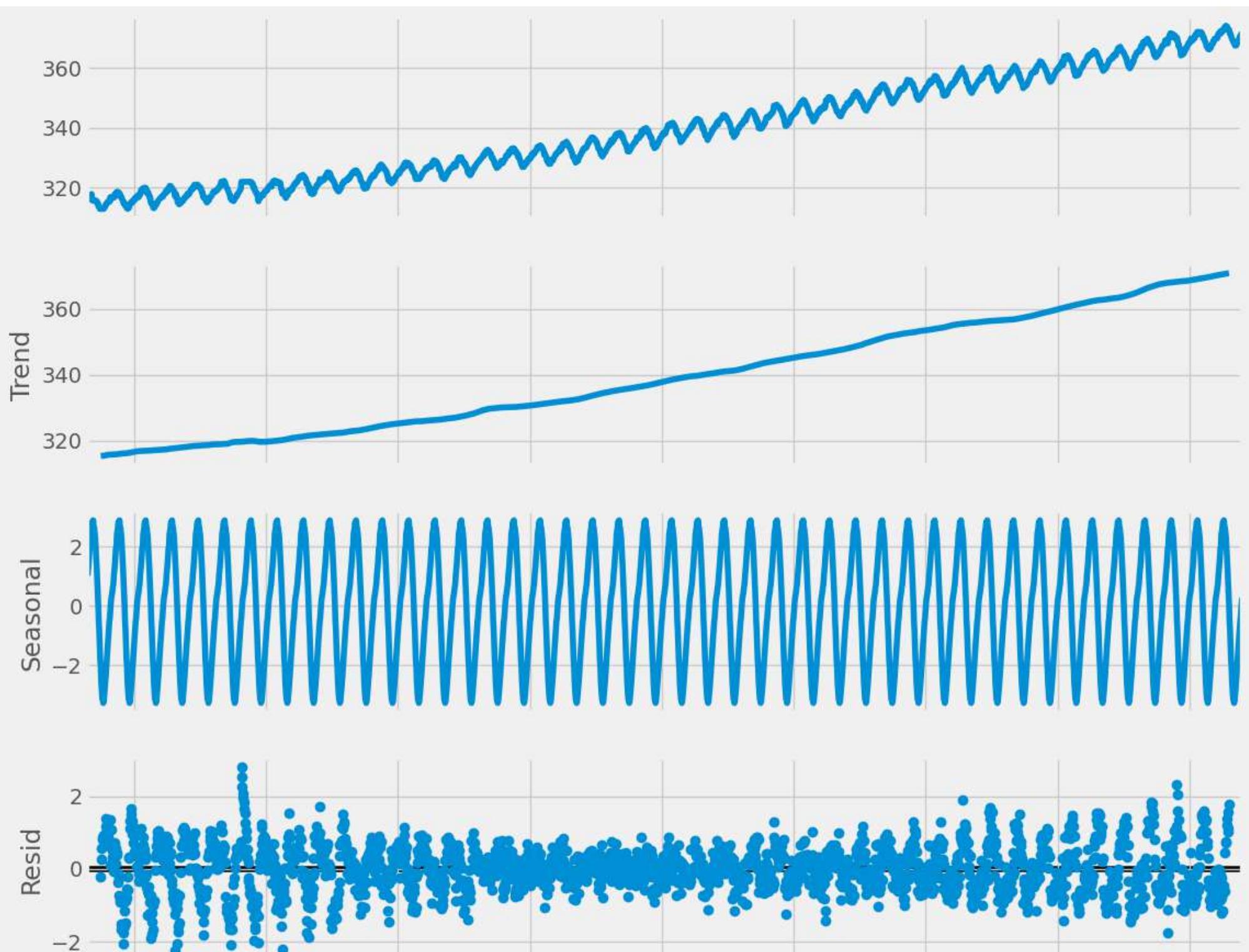
```
In [ ]: # Extract the trend component
trend = decomposition.trend

# Plot the values of the trend
ax = trend.plot(figsize=(12, 6), fontsize=6)

# Specify axis Labels
ax.set_xlabel('Date', fontsize=10)
ax.set_title('Seasonal component the CO2 time-series', fontsize=10)
plt.show();
```



```
In [ ]: fig = decomposition.plot()  
fig.set_figheight(10)  
fig.set_figwidth(12)  
plt.show();
```



1960

1965

1970

1975

1980

1985

1990

1995

2000

## Visualize the airline dataset

You will have the opportunity to work with a new dataset that contains the monthly number of passengers who took a commercial flight between January 1949 and December 1960.

```
In [ ]: import pandas as pd

# Read in the file content in a DataFrame called airline
airline = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\airline-passenger.csv')

# Display the first five lines of the DataFrame
print(airline.head())
```

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

```
In [ ]: print(airline.tail())
```

```
AirPassengers
Month
1960-08-01      606
1960-09-01      508
1960-10-01      461
1960-11-01      390
1960-12-01      432
```

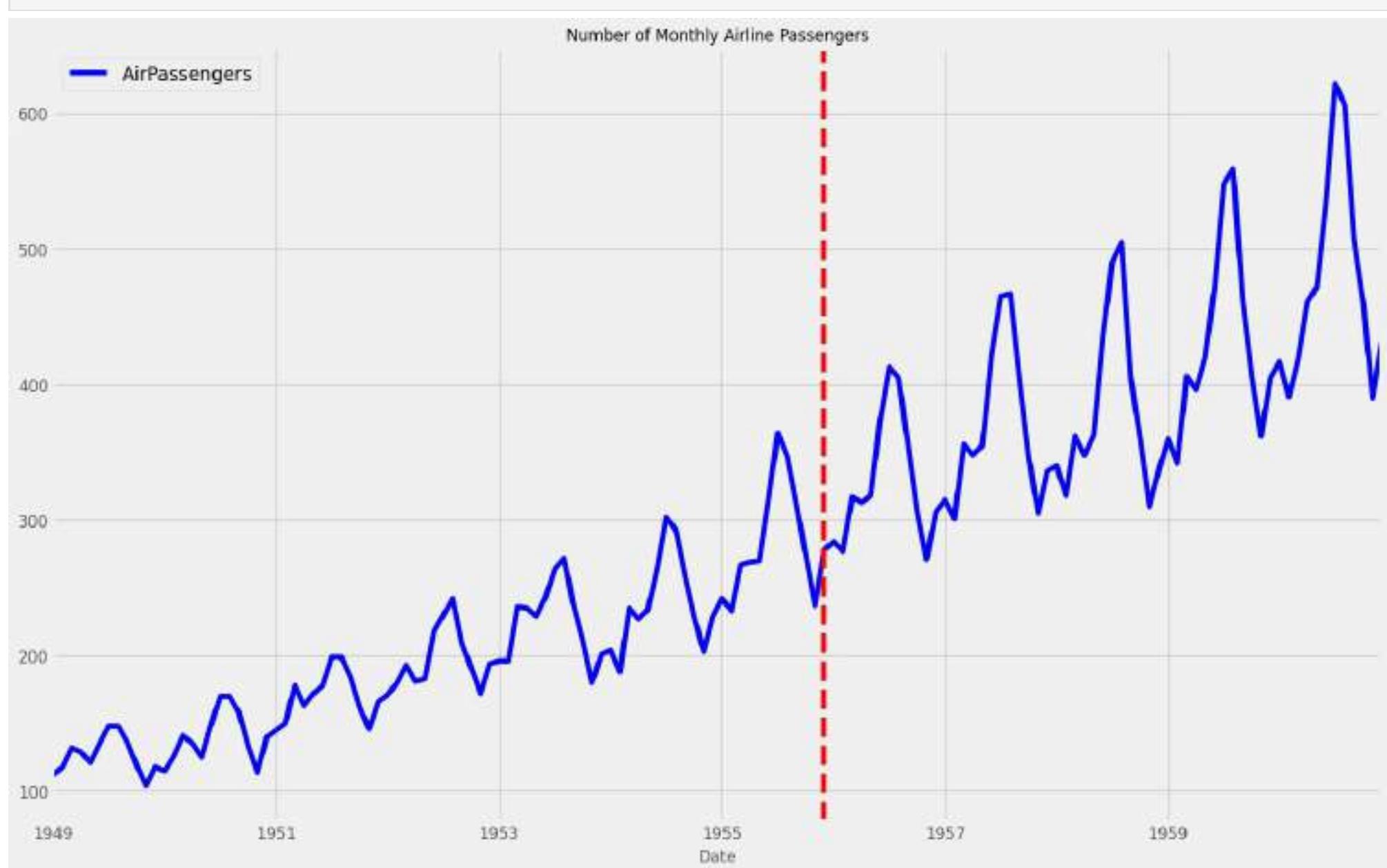
```
In [ ]: plt.rcParams["figure.figsize"] = (16,10)

# Plot the time series in your dataframe
ax = airline.plot(color="blue", fontsize=12)

# Add a red vertical line at the date 1955-12-01
ax.axvline('1955-12-01', color='red', linestyle='--')

# Specify the labels in your plot
ax.set_xlabel('Date', fontsize=12)
```

```
ax.set_title('Number of Monthly Airline Passengers', fontsize=12)
plt.show();
```



*The number of airline passengers has risen a lot over time. Can you find any interesting patterns in this time series?*

## Analyze the airline dataset

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

```
In [ ]: # Print out the number of missing values
```

```
print(airline.isnull().sum())
```

```
# Print out summary statistics of the airline DataFrame
```

```
print(airline.describe())
```

```
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
```

```
In [ ]: ax = airline.plot(kind='hist')
```

```
# Mean
```

```
ax.axvline(280, color='red', linestyle='--', label="mean")
```

```
# Median
```

```
ax.axvline(265, color='green', linestyle='--', label="median")
```

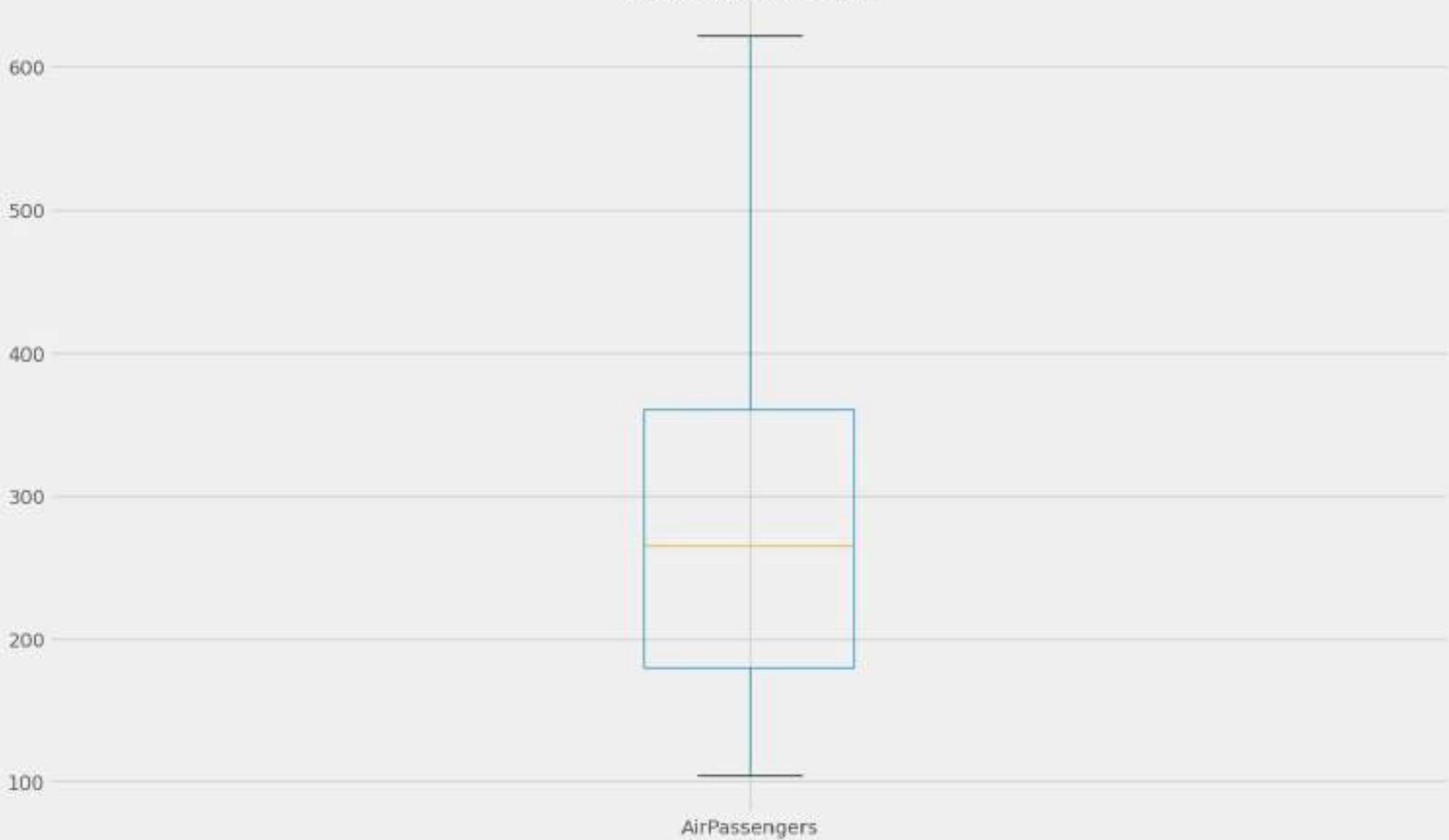
```
plt.show();
```



```
In [ ]: # Display boxplot of airline values
ax = airline.boxplot()

# Specify the title of your plot
ax.set_title('Boxplot of Monthly Airline\nPassengers Count', fontsize=20)
plt.show();
```

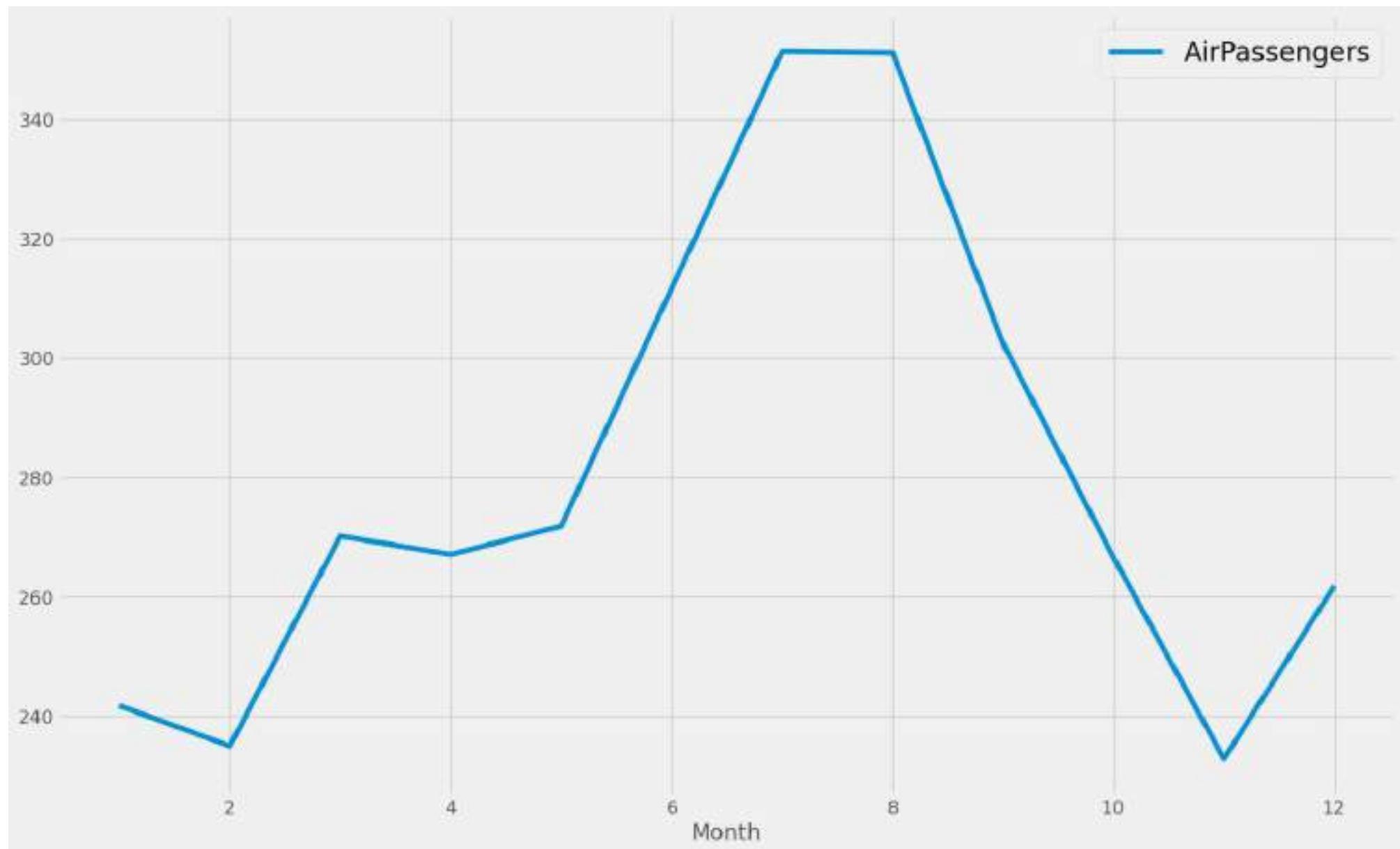
### Boxplot of Monthly Airline Passengers Count



```
In [ ]: # Get month for each dates from the index of airline
index_month = airline.index.month

# Compute the mean number of passengers for each month of the year
mean_airline_by_month = airline.groupby(index_month).mean()
```

```
# Plot the mean number of passengers for each month of the year  
mean_airline_by_month.plot()  
plt.legend(fontsize=20)  
plt.show();
```



*Looks like July and August are the busiest months!*

## Time series decomposition of the airline dataset

In this exercise, you will apply time series decomposition to the airline dataset, and visualize the trend and seasonal components.

```
In [ ]: # Import statsmodels.api as sm  
import statsmodels.api as sm
```

```
# Perform time series decomposition  
decomposition = sm.tsa.seasonal_decompose(airline)  
  
# Extract the trend and seasonal components  
trend = decomposition.trend  
seasonal = decomposition.seasonal  
  
noise = decomposition.resid
```

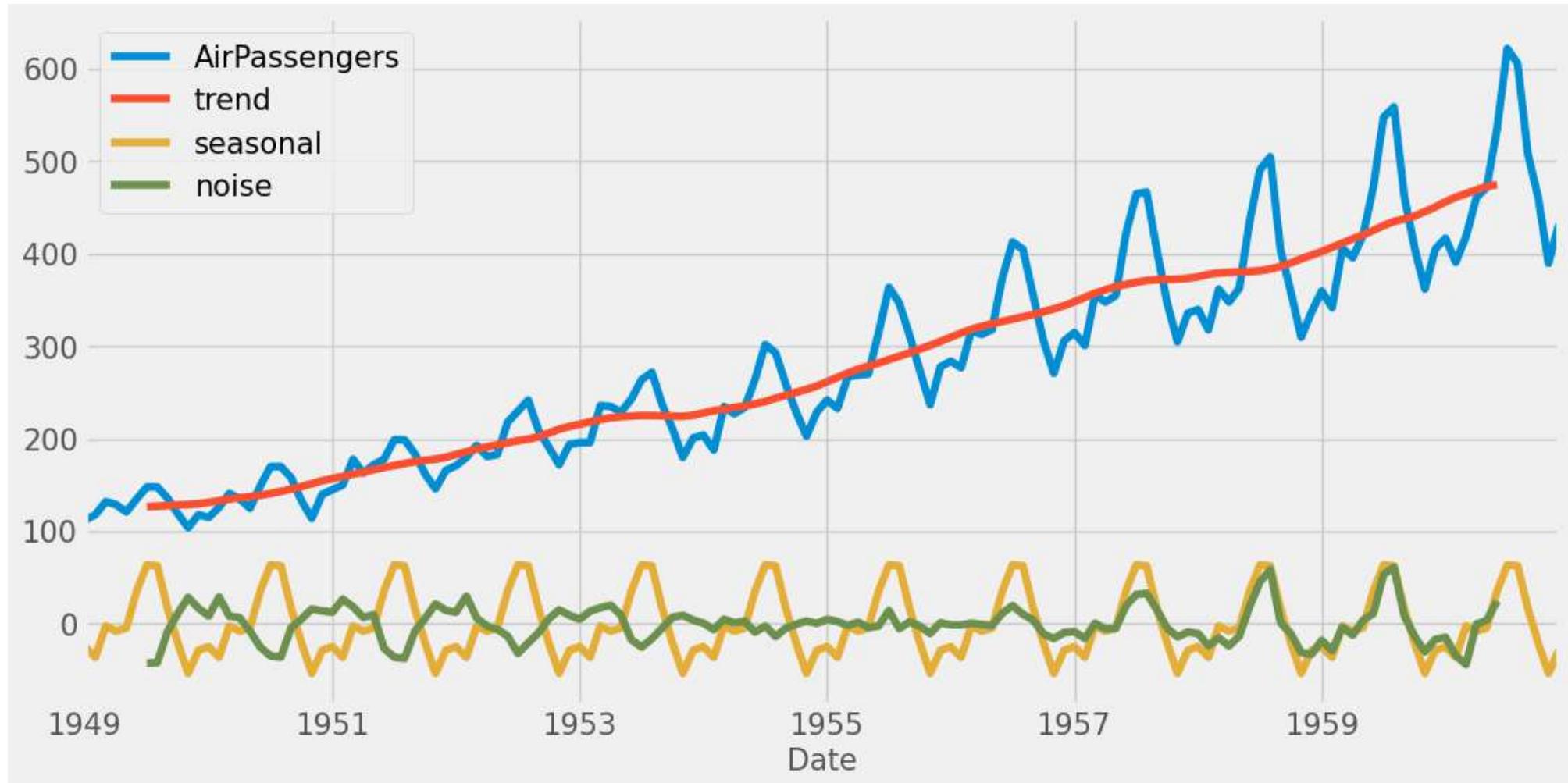
```
In [ ]: airline_decomposed = airline.copy()  
airline_decomposed['trend'] = trend  
airline_decomposed['seasonal'] = seasonal  
airline_decomposed['noise'] = noise  
airline_decomposed.head()
```

```
Out[ ]:      AirPassengers  trend  seasonal  noise  
Month  
1949-01-01        112    NaN   -24.748737    NaN  
1949-02-01        118    NaN   -36.188131    NaN  
1949-03-01        132    NaN   -2.241162    NaN  
1949-04-01        129    NaN   -8.036616    NaN  
1949-05-01        121    NaN   -4.506313    NaN
```

```
In [ ]: # Print the first 5 rows of airline_decomposed  
print(airline_decomposed.head())  
  
# Plot the values of the airline_decomposed DataFrame  
ax = airline_decomposed.plot(figsize=(12, 6), fontsize=15)  
  
# Specify axis labels  
ax.set_xlabel('Date', fontsize=15)
```

```
plt.legend(fontsize=15)  
plt.show();
```

	AirPassengers	trend	seasonal	noise
Month				
1949-01-01	112	NaN	-24.748737	NaN
1949-02-01	118	NaN	-36.188131	NaN
1949-03-01	132	NaN	-2.241162	NaN
1949-04-01	129	NaN	-8.036616	NaN
1949-05-01	121	NaN	-4.506313	NaN



## Chapter 4- Work with Multiple Time Series

In the field of Data Science, it is common to be involved in projects where multiple time series need to be studied simultaneously. In this chapter, we will show you how to plot multiple time series at once, and how to discover and describe relationships between multiple time series.

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

```
In [ ]: # Read in the file content in a DataFrame called meat
meat = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\meat.csv')
print(meat.head());
```

	date	beef	veal	pork	lamb_and_mutton	broilers	other_chicken	\
0	1944-01-01	751.0	85.0	1280.0	89.0	NaN	NaN	
1	1944-02-01	713.0	77.0	1169.0	72.0	NaN	NaN	
2	1944-03-01	741.0	90.0	1128.0	75.0	NaN	NaN	
3	1944-04-01	650.0	89.0	978.0	66.0	NaN	NaN	
4	1944-05-01	681.0	106.0	1029.0	78.0	NaN	NaN	

	turkey
0	NaN
1	NaN
2	NaN
3	NaN
4	NaN

## Load multiple time series

Whether it is during personal projects or your day-to-day work as a Data Scientist, it is likely that you will encounter situations that require the analysis and visualization of multiple time series at the same time.

Provided that the data for each time series is stored in distinct columns of a file, the pandas library makes it easy to work with multiple time series. In the following exercises, you will work with a new time series dataset that contains the amount of different types of meat produced in the USA between 1944 and 2012.

```
In [ ]: # Review the first five lines of the meat DataFrame
print(meat.head(5))

# Convert the date column to a datetime type
meat['date'] = pd.to_datetime(meat['date'])

# Set the date column as the index of your DataFrame meat
meat = meat.set_index('date')

# Print the summary statistics of the DataFrame
print(meat.describe())
```

```
      date  beef   veal    pork lamb_and_mutton  broilers other_chicken \
0  1944-01-01  751.0  85.0  1280.0           89.0       NaN        NaN
1  1944-02-01  713.0  77.0  1169.0           72.0       NaN        NaN
2  1944-03-01  741.0  90.0  1128.0           75.0       NaN        NaN
3  1944-04-01  650.0  89.0  978.0            66.0       NaN        NaN
4  1944-05-01  681.0  106.0 1029.0            78.0       NaN        NaN
```

### turkey

```
0      NaN
1      NaN
2      NaN
3      NaN
4      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
```

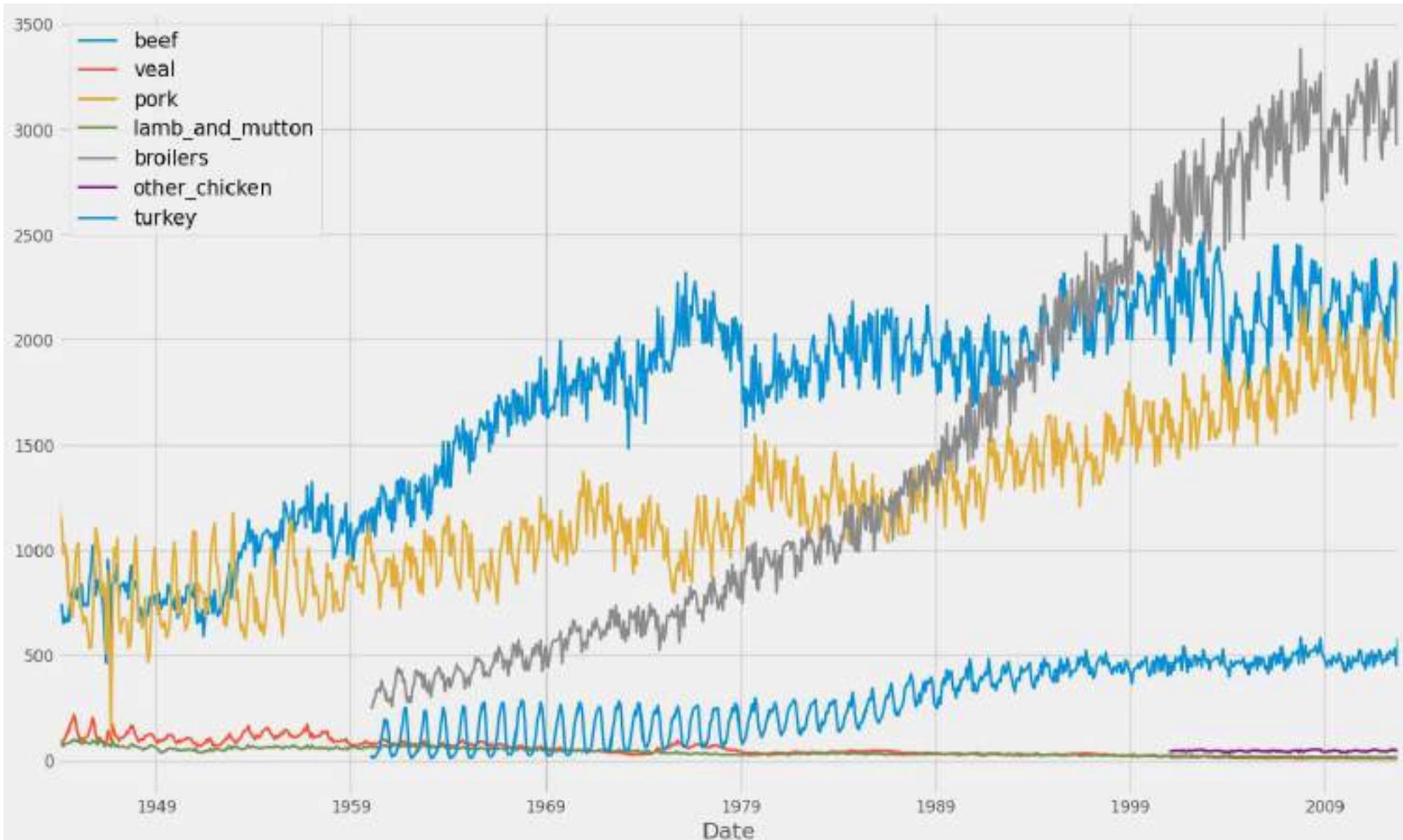
## Visualize multiple time series

If there are multiple time series in a single DataFrame, you can still use the .plot() method to plot a line chart of all the time series. Another interesting way to plot these is to use area charts. Area charts are commonly used when dealing with multiple time series, and can be used to display cumulated totals.

With the pandas library, you can simply leverage the .plot.area() method to produce area charts of the time series data in your DataFrame.

```
In [ ]: import matplotlib.pyplot as plt
ax = meat.plot(linewidth=2, fontsize=12)
```

```
# Additional customizations  
ax.set_xlabel('Date')  
ax.legend(fontsize=15)  
  
# Show plot  
plt.show();
```



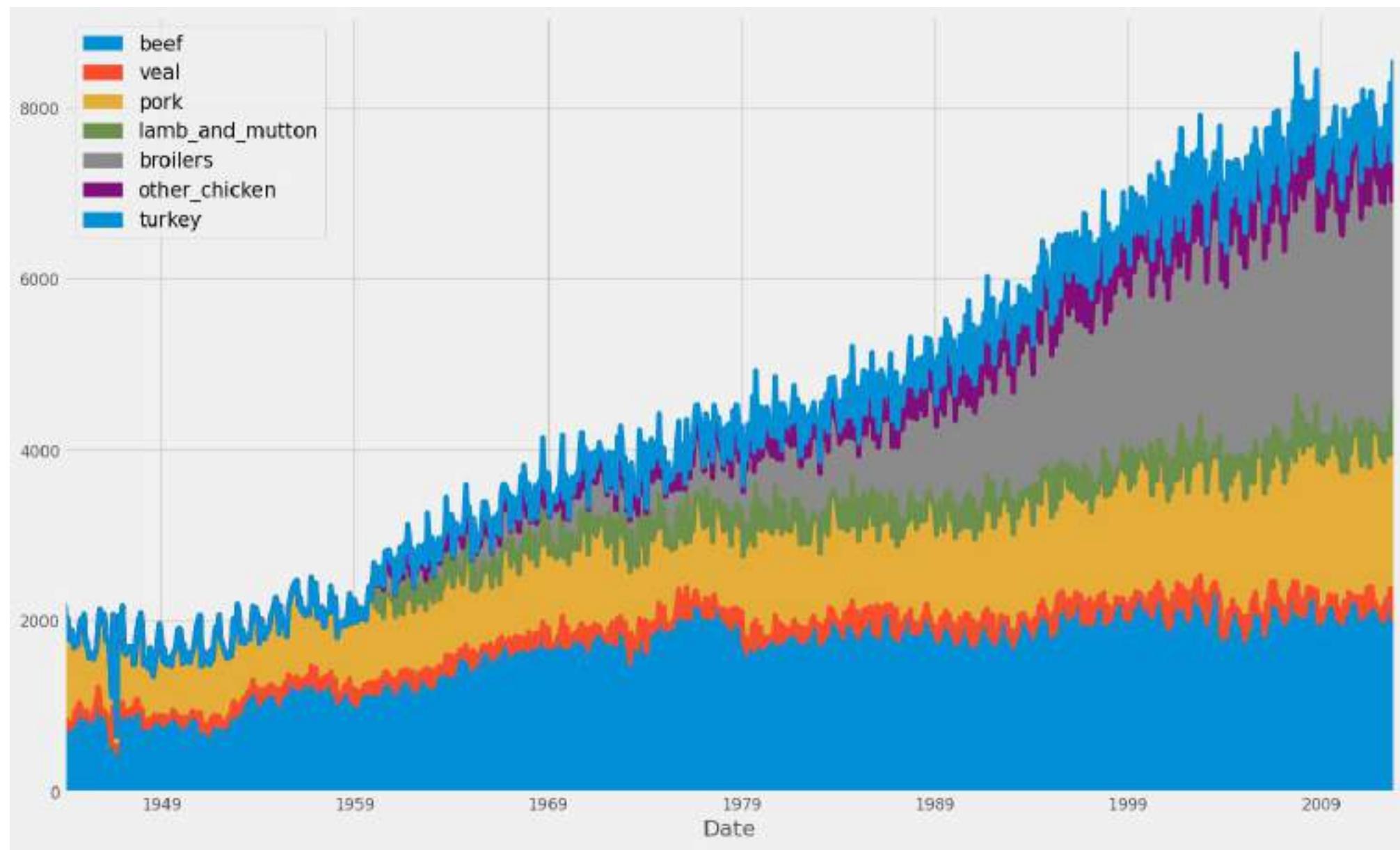
Area chart

It looks the time series in the meat DataFrame have very different growth rates!

```
In [ ]: # Plot an area chart
ax = meat.plot.area(fontsize=12)

# Additional customizations
ax.set_xlabel('Date')
ax.legend(fontsize=15)

# Show plot
plt.show();
```



### Statistical summaries of multiple time series

As seen in the last exercise, the time series in the meat DataFrame display very different behavior over time.

Using the summary statistics presented below, can you identify the time series with the highest mean and maximum value, respectively? Answer: beef has the highest mean and broilers has the maximum value.

```
In [ ]: meat.describe()
```

	beef	veal	pork	lamb_and_mutton	broilers	other_chicken	turkey
<b>count</b>	827.000000	827.000000	827.000000	827.000000	635.000000	143.000000	635.000000
<b>mean</b>	1683.463362	54.198549	1211.683797	38.360701	1516.582520	43.033566	292.814646
<b>std</b>	501.698480	39.062804	371.311802	19.624340	963.012101	3.867141	162.482638
<b>min</b>	366.000000	8.800000	124.000000	10.900000	250.900000	32.300000	12.400000
<b>25%</b>	1231.500000	24.000000	934.500000	23.000000	636.350000	40.200000	154.150000
<b>50%</b>	1853.000000	40.000000	1156.000000	31.000000	1211.300000	43.400000	278.300000
<b>75%</b>	2070.000000	79.000000	1466.000000	55.000000	2426.650000	45.650000	449.150000
<b>max</b>	2512.000000	215.000000	2210.400000	109.000000	3383.800000	51.100000	585.100000

## 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.

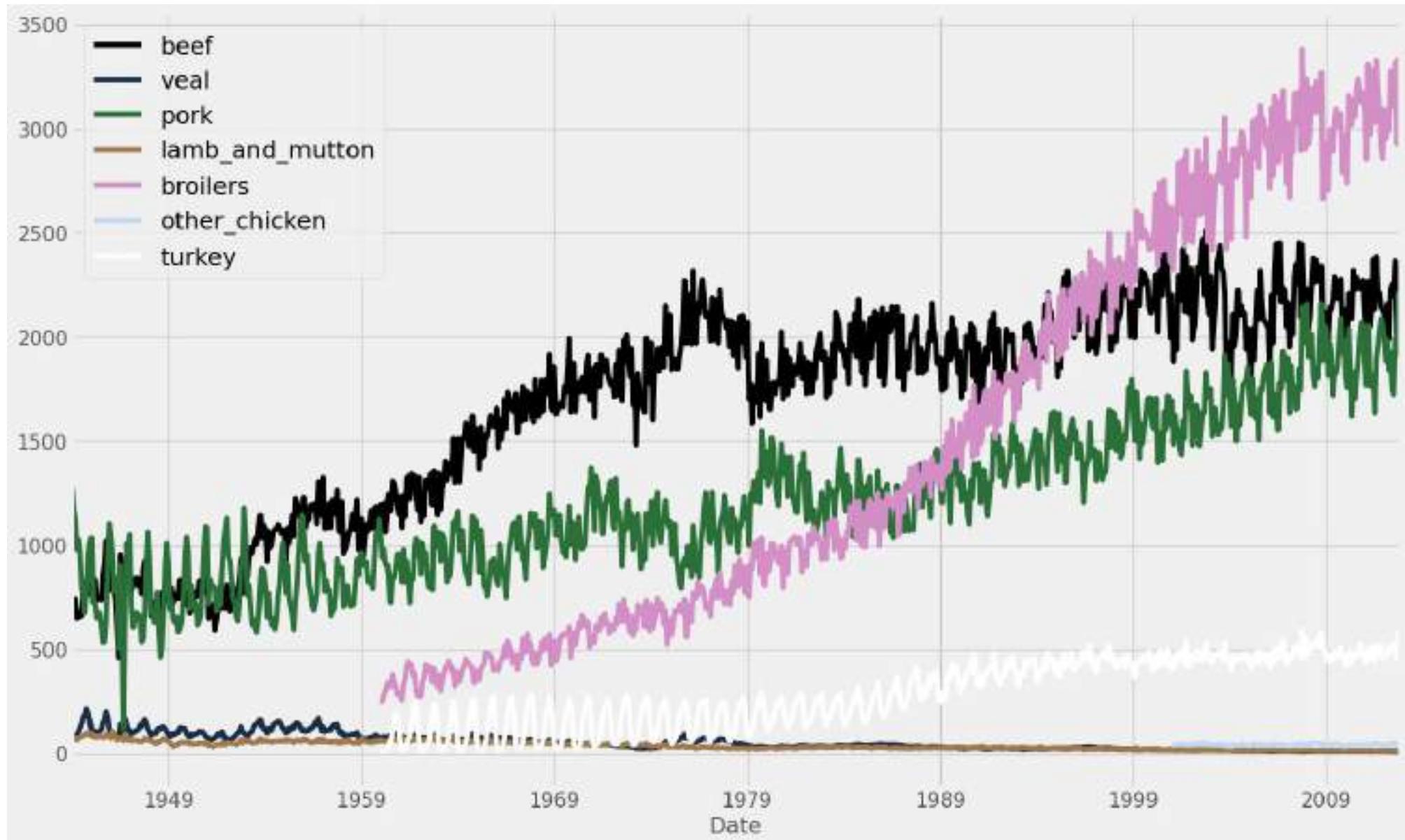
To remedy this, you can define each color manually, but this may be time-consuming. Fortunately, it is possible to leverage the colormap argument to .plot() to automatically assign specific color palettes with varying contrasts. You can either provide a matplotlib colormap as an input to this parameter,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

For example, you can specify the 'viridis' colormap using the following command: df.plot(colormap='viridis').

```
In [ ]: # Plot time series dataset using the cubehelix color palette
ax = meat.plot(colormap='cubehelix', fontsize=15)

# Additional customizations
ax.set_xlabel('Date')
ax.legend(fontsize=18)

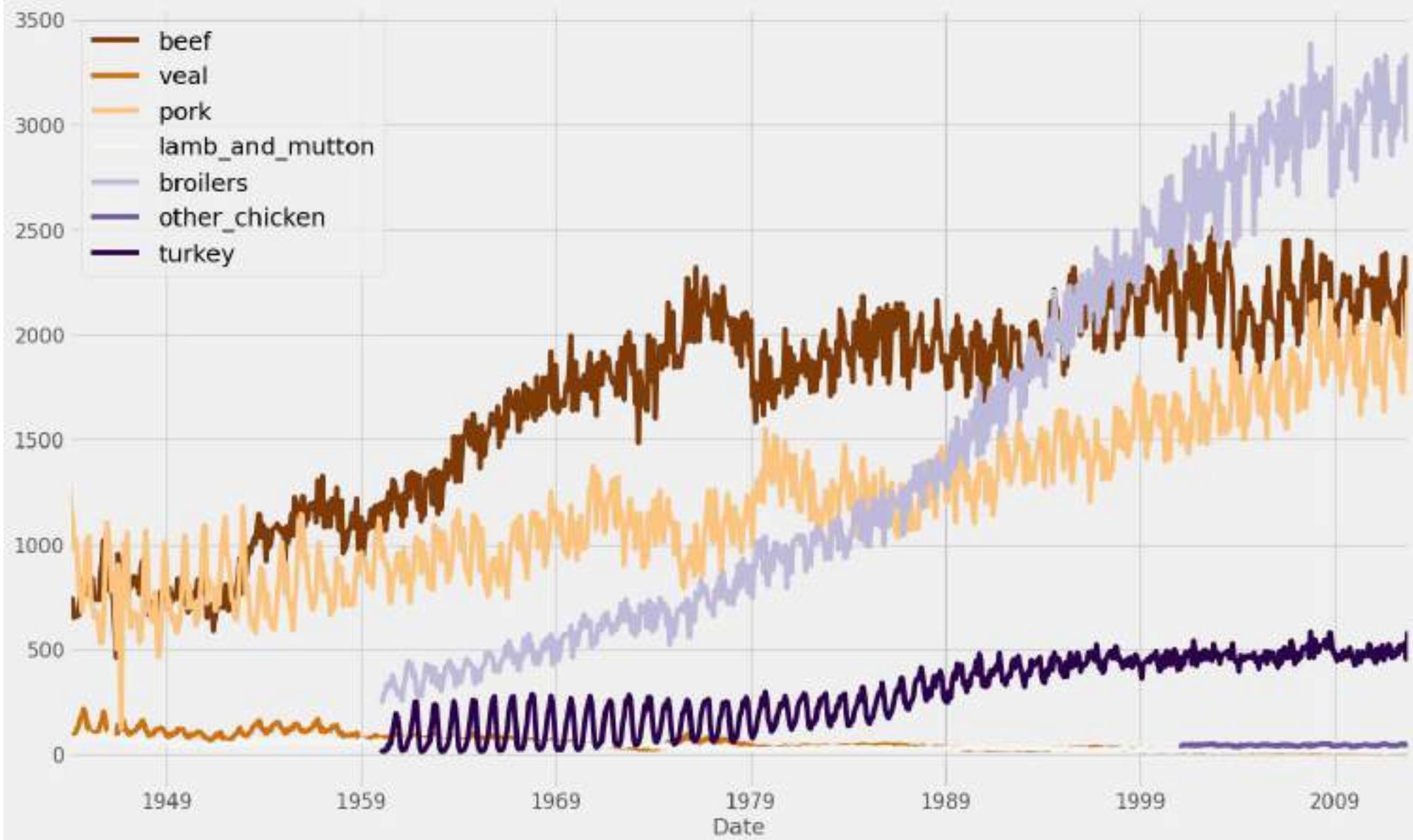
# Show plot
plt.show();
```



```
In [ ]: # Plot time series dataset using the cubehelix color palette
ax = meat.plot(colormap='PuOr', fontsize=15)

# Additional customizations
ax.set_xlabel('Date')
ax.legend(fontsize=18)
```

```
# Show plot  
plt.show();
```



### Add summary statistics to your time series plot

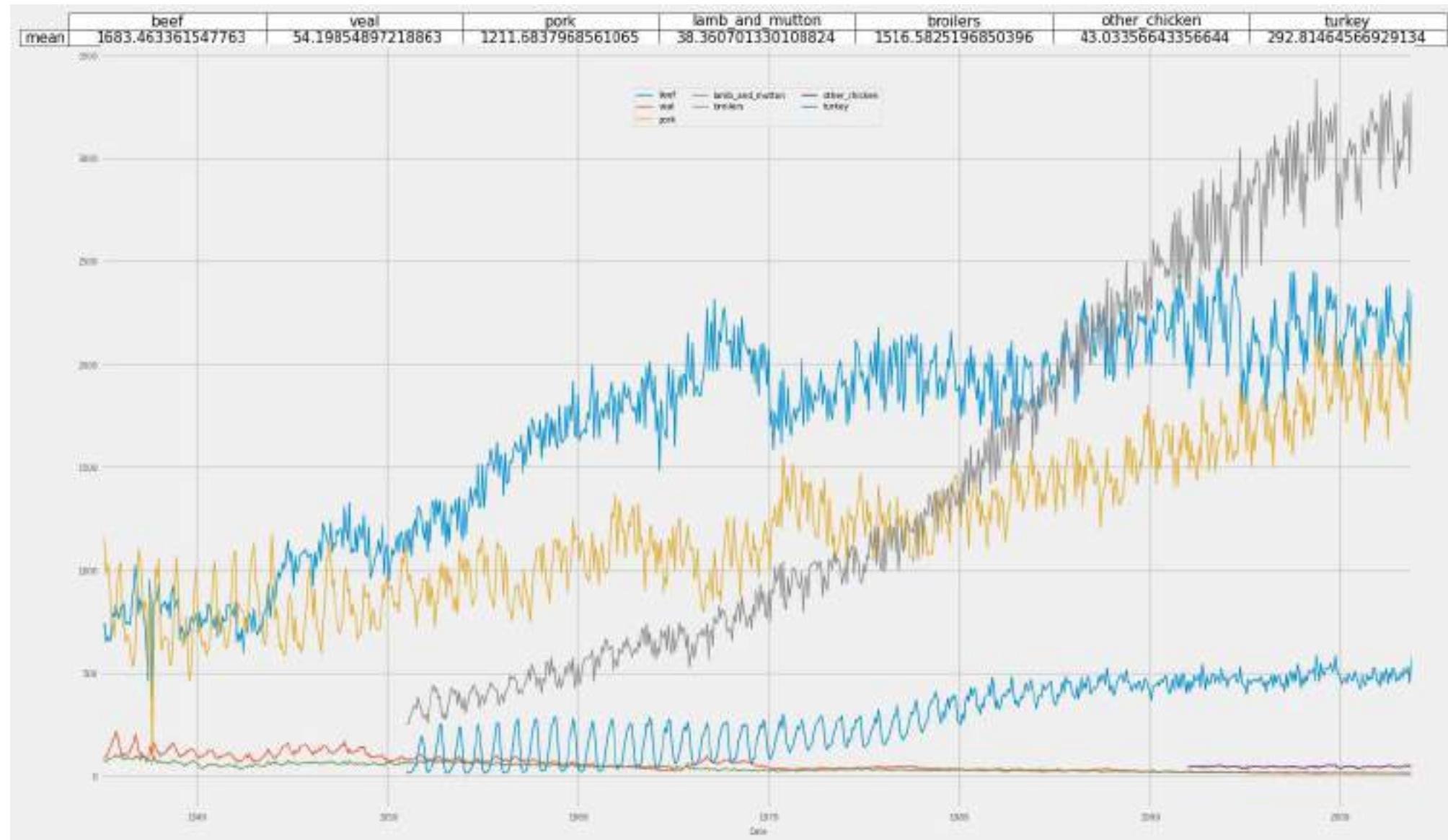
It is possible to visualize time series plots and numerical summaries on one single graph by using the pandas API to matplotlib along with the **table** method:

```
# Plot the time series data in the DataFrame  
ax = df.plot()  
  
# Compute summary statistics of the df DataFrame  
df_summary = df.describe()  
  
# Add summary table information to the plot  
ax.table(cellText=df_summary.values,  
         colWidths=[0.3]*len(df.columns),  
         rowLabels=df_summary.index,  
         colLabels=df_summary.columns,  
         loc='top')
```

```
In [ ]: meat_mean = meat.agg(['mean'])  
meat_mean
```

```
Out[ ]:      beef    veal    pork lamb_and_mutton    broilers other_chicken    turkey  
mean  1683.463362  54.198549 1211.683797          38.360701  1516.58252     43.033566  292.814646
```

```
# Plot the meat data  
ax = meat.plot(fontsize=6, linewidth=1)  
  
# Add x-axis labels  
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')  
  
# Specify the fontsize and location of your legend  
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 0.95), ncol=3, fontsize=6)  
  
# Show plot  
plt.show();
```



### Plot your time series on individual plots

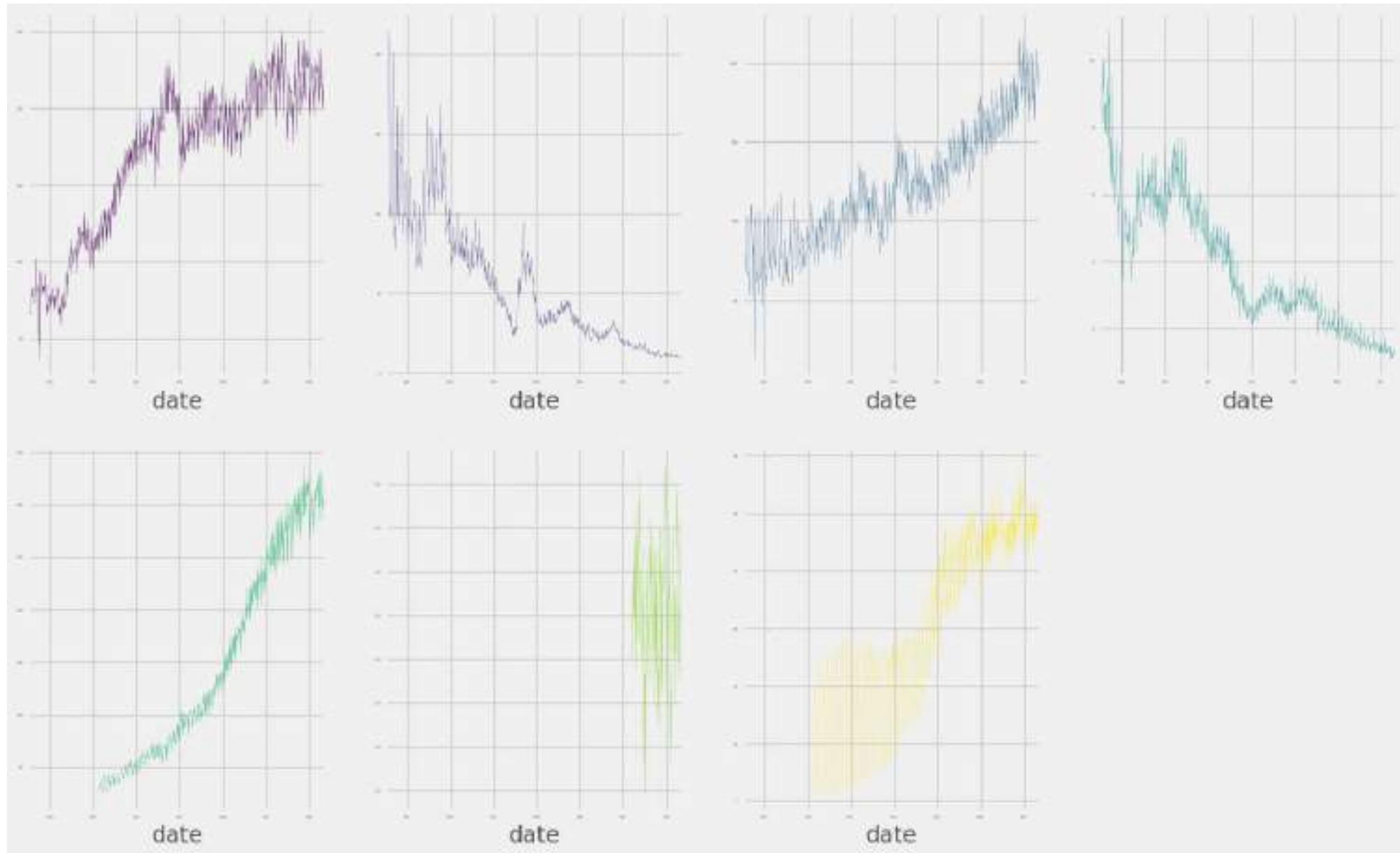
It can be beneficial to plot individual time series on separate graphs as this may improve clarity and provide more context around each time series in your DataFrame.

It is possible to create a "grid" of individual graphs by "faceting" each time series by setting the subplots argument to True. In addition, the arguments that can be added are:

- layout: specifies the number of rows x columns to use.
- sharex and sharey: specifies whether the x-axis and y-axis values should be shared between your plots.

```
In [ ]: # 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

In the field of Statistics, the correlation coefficient is a measure used to determine the strength or lack relationship between two variables:

Pearson's coefficient can be used to compute the correlation coefficient between variables for which the relationship is thought to be linear.

Jendall Tau or Spearman rank can be used to compute the correlation coefficient between variables for which the relationship is thought to be non-linear.

The correlation coefficient can be used to determine how multiple variables (or a group of time series) are associated with one another. The result is a correlation matrix that describes the correlation between time series. Note that the diagonal values in a correlation matrix will always be 1, since a time series will always be perfectly correlated with itself.

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 [ ]: from scipy.stats.stats import pearsonr
from scipy.stats.stats import spearmanr
from scipy.stats.stats import kendalltau
x = [1, 2, 4, 7]
y = [1, 3, 4, 8]
pearsonr(x, y)
```

```
C:\Users\yeiso\AppData\Local\Temp\ipykernel_39800\2107548140.py:1: DeprecationWarning: Please use `pearsonr` from the `scipy.stats` namespace, the `scipy.stats.stats` namespace is deprecated.
  from scipy.stats.stats import pearsonr
C:\Users\yeiso\AppData\Local\Temp\ipykernel_39800\2107548140.py:2: DeprecationWarning: Please use `spearmanr` from the `scipy.stats` namespace, the `scipy.stats.stats` namespace is deprecated.
  from scipy.stats.stats import spearmanr
C:\Users\yeiso\AppData\Local\Temp\ipykernel_39800\2107548140.py:3: DeprecationWarning: Please use `kendalltau` from the `scipy.stats` namespace, the `scipy.stats.stats` namespace is deprecated.
  from scipy.stats.stats import kendalltau
```

```
Out[ ]: PearsonRResult(statistic=0.9843091327750998, pvalue=0.015690867224900096)
```

```
In [ ]: spearmanr(x, y)
```

```
Out[ ]: SignificanceResult(statistic=1.0, pvalue=0.0)
```

```
In [ ]: kendalltau(x, y)
```

```
Out[ ]: SignificanceResult(statistic=1.0, pvalue=0.08333333333333333)
```

```
In [ ]: # Compute the correlation between the beef and pork columns using the spearman method
print(meat[['beef', 'pork']].corr(method='spearman'))
```

```
corr_s = meat[['beef', 'pork']].corr(method='spearman')
```

```
      beef      pork  
beef  1.000000  0.827587  
pork  0.827587  1.000000
```

```
In [ ]: type(corr_s)
```

```
Out[ ]: pandas.core.frame.DataFrame
```

```
In [ ]: # Print the correlation matrix between the beef and pork columns using the spearman method  
print(meat[['beef', 'pork']].corr(method='spearman'))
```

```
# Print the correlation between beef and pork columns  
print(0.828)
```

```
      beef      pork  
beef  1.000000  0.827587  
pork  0.827587  1.000000  
0.828
```

```
In [ ]: # Compute the correlation between the pork, veal and turkey columns using the pearson method
```

```
print(meat[['pork', 'veal', 'turkey']].corr('pearson'))
```

```
# Print the correlation between veal and pork columns  
print(-0.809)
```

```
# Print the correlation between veal and turkey columns  
print(-0.768)
```

```
# Print the correlation between pork and turkey columns  
print(0.835 )
```

```
      pork      veal      turkey  
pork  1.000000 -0.808834  0.835215  
veal -0.808834  1.000000 -0.768366  
turkey  0.835215 -0.768366  1.000000  
-0.809  
-0.768  
0.835
```

## 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.

```
df_corr = df.corr()

sns.heatmap(df_corr)
plt.xticks(rotation=90)
plt.yticks(rotation=0)
You can use the .xticks() and .yticks() methods to rotate the axis labels so they don't overlap.
```

To learn about the arguments to the heatmap() function, refer to this page.

```
In [ ]: # Import seaborn library
import seaborn as sns
```

```
# Get correlation matrix of the meat DataFrame: corr_meat
corr_meat = meat.corr(method='spearman')

# Customize the heatmap of the corr_meat correlation matrix
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:

```
df_corr = df.corr()

fig = sns.clustermap(df_corr)

plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90)

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

To prevent overlapping of axis labels, you can reference the Axes from the underlying fig object and specify the rotation. You can learn about the arguments to the clustermap() function here (<https://seaborn.pydata.org/generated/seaborn.clustermap.html>)

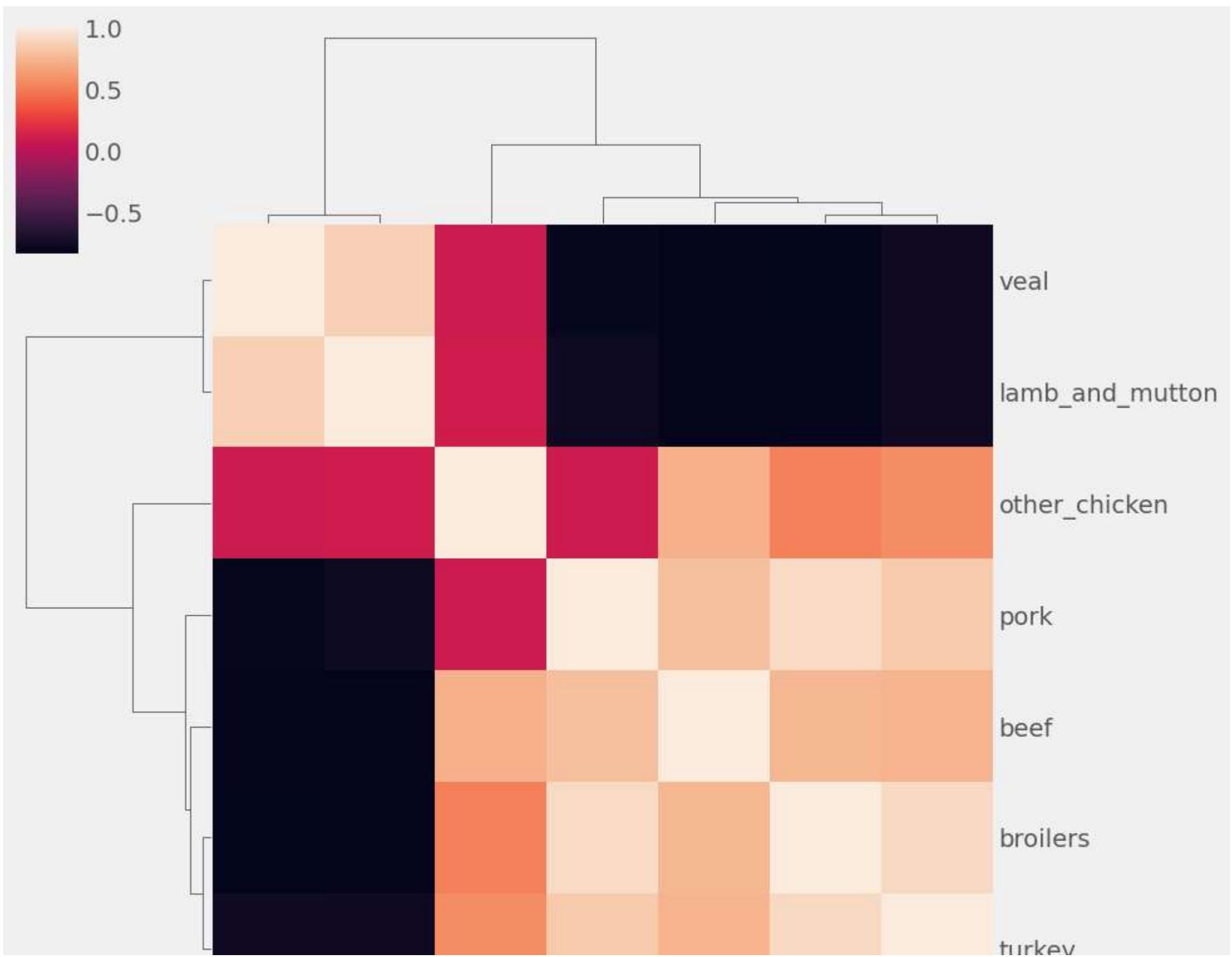
In [ ]:

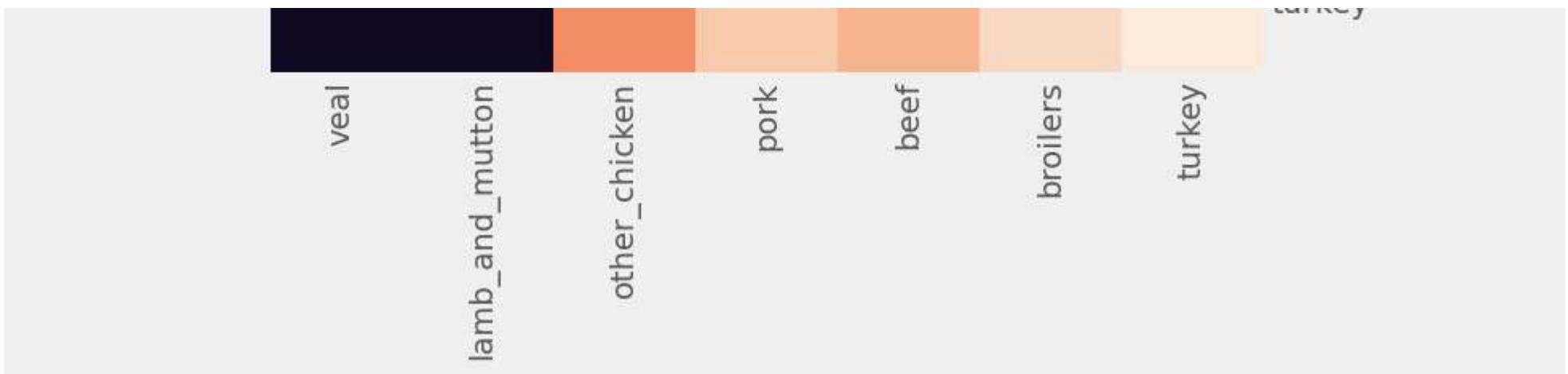
```
# Import seaborn library
import seaborn as sns

# Get correlation matrix of the meat DataFrame
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();
```





## Explore the Jobs dataset

In this exercise, you will explore the new jobs DataFrame, which contains the unemployment rate of different industries in the USA during the years of 2000-2010. As you will see, the dataset contains time series for 16 industries and across 122 timepoints (one per month for 10 years). In general, the typical workflow of a Data Science project will involve data cleaning and exploration, so we will begin by reading in the data and checking for missing values.

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

plt.rcParams['figure.figsize'] = (10, 5)
plt.style.use('fivethirtyeight')
```

```
In [ ]: jobs = pd.read_csv('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\

# Print first five lines of your DataFrame
print(jobs.head(5))

# Check the type of each column in your DataFrame
print(jobs.dtypes)

# Convert datestamp column to a datetime object
jobs['datestamp'] = pd.to_datetime(jobs['datestamp'])

# Set the datestamp columns as the index of your DataFrame
jobs = jobs.set_index('datestamp')
```

```
# Check the number of missing values in each columns  
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	

	object
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

You should always explore the distribution of the variables, and because you are working with time series, you will explore their properties using boxplots and numerical summaries. As a reminder, you can plot data in a DataFrame as boxplots with the command:

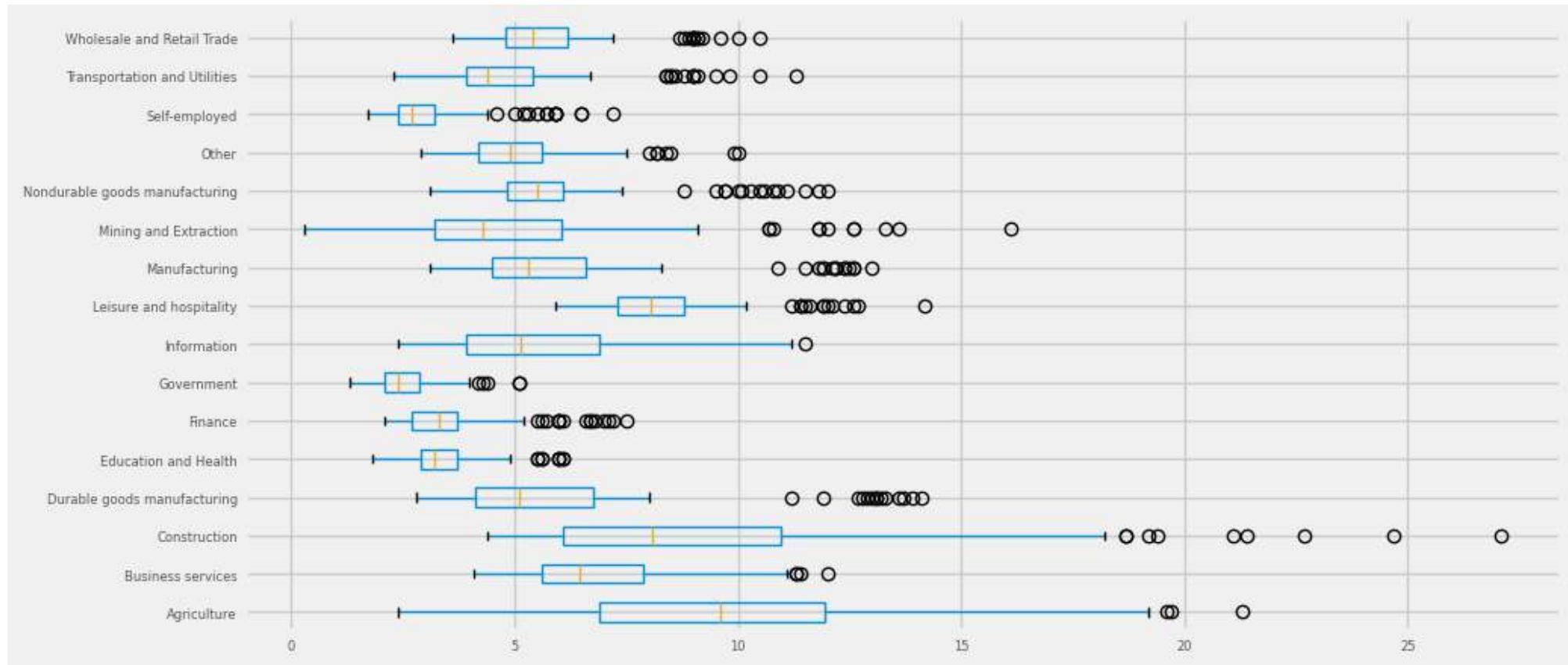
In [ ]:

```
# Generate a boxplot
jobs.boxplot(fontsize=6, vert=False)
plt.show()

# Generate numerical summaries
print(jobs.describe())

# Print the name of the time series with the highest mean
print(jobs.mean().idxmax())

# Print the name of the time series with the highest variability
print(jobs.std().idxmax())
```



	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
```

```
In [ ]: #you can also use 'describe'
print(jobs.describe)
```

		Agriculture	Business services	Construction	\
datestamp					
2000-01-01	10.3	5.7	9.7		
2000-02-01	11.5	5.2	10.6		
2000-03-01	10.4	5.4	8.7		
2000-04-01	8.9	4.5	5.8		
2000-05-01	5.1	4.7	5.0		
...	...	...	...		
2009-10-01	11.8	10.3	18.7		
2009-11-01	12.6	10.6	19.4		
2009-12-01	19.7	10.3	22.7		
2010-01-01	21.3	11.1	24.7		
2010-02-01	18.8	12.0	27.1		

	Durable goods manufacturing	Education and Health	Finance	\
datestamp				
2000-01-01		3.2	2.3	2.7
2000-02-01		2.9	2.2	2.8
2000-03-01		2.8	2.5	2.6
2000-04-01		3.4	2.1	2.3
2000-05-01		3.4	2.7	2.2
...	...	...	...	
2009-10-01	12.9	6.0	7.0	
2009-11-01	12.7	5.5	6.7	
2009-12-01	13.3	5.6	7.2	
2010-01-01	14.1	5.5	6.6	
2010-02-01	13.6	5.6	7.5	

	Government	Information	Leisure and hospitality	Manufacturing	\
datestamp					
2000-01-01	2.1	3.4	7.5	3.6	
2000-02-01	2.0	2.9	7.5	3.4	
2000-03-01	1.5	3.6	7.4	3.6	
2000-04-01	1.3	2.4	6.1	3.7	
2000-05-01	1.9	3.5	6.2	3.4	
...	...	...	...	...	
2009-10-01	3.5	8.2	12.4	12.2	
2009-11-01	3.4	7.6	11.9	12.5	
2009-12-01	3.6	8.5	12.6	11.9	
2010-01-01	4.3	10.0	14.2	13.0	
2010-02-01	4.0	10.0	12.7	12.1	

	Mining and Extraction	Nondurable goods manufacturing	Other	\
datestamp				
2000-01-01		3.9	4.4	4.9
2000-02-01		5.5	4.2	4.1

2000-03-01	3.7	5.1	4.3
2000-04-01	4.1	4.0	4.2
2000-05-01	5.3	3.6	4.5
...	...	...	...
2009-10-01	10.8	10.9	8.5
2009-11-01	12.0	12.0	8.0
2009-12-01	11.8	9.5	8.2
2010-01-01	9.1	11.1	10.0
2010-02-01	10.7	9.7	9.9

#### Self-employed Transportation and Utilities \

datestamp		
2000-01-01	2.3	4.3
2000-02-01	2.5	4.0
2000-03-01	2.0	3.5
2000-04-01	2.0	3.4
2000-05-01	1.9	3.4
...	...	...
2009-10-01	5.9	8.6
2009-11-01	5.7	8.5
2009-12-01	5.9	9.0
2010-01-01	7.2	11.3
2010-02-01	6.5	10.5

#### Wholesale and Retail Trade

datestamp	
2000-01-01	5.0
2000-02-01	5.2
2000-03-01	5.1
2000-04-01	4.1
2000-05-01	4.3
...	...
2009-10-01	9.6
2009-11-01	9.2
2009-12-01	9.1
2010-01-01	10.5
2010-02-01	10.0

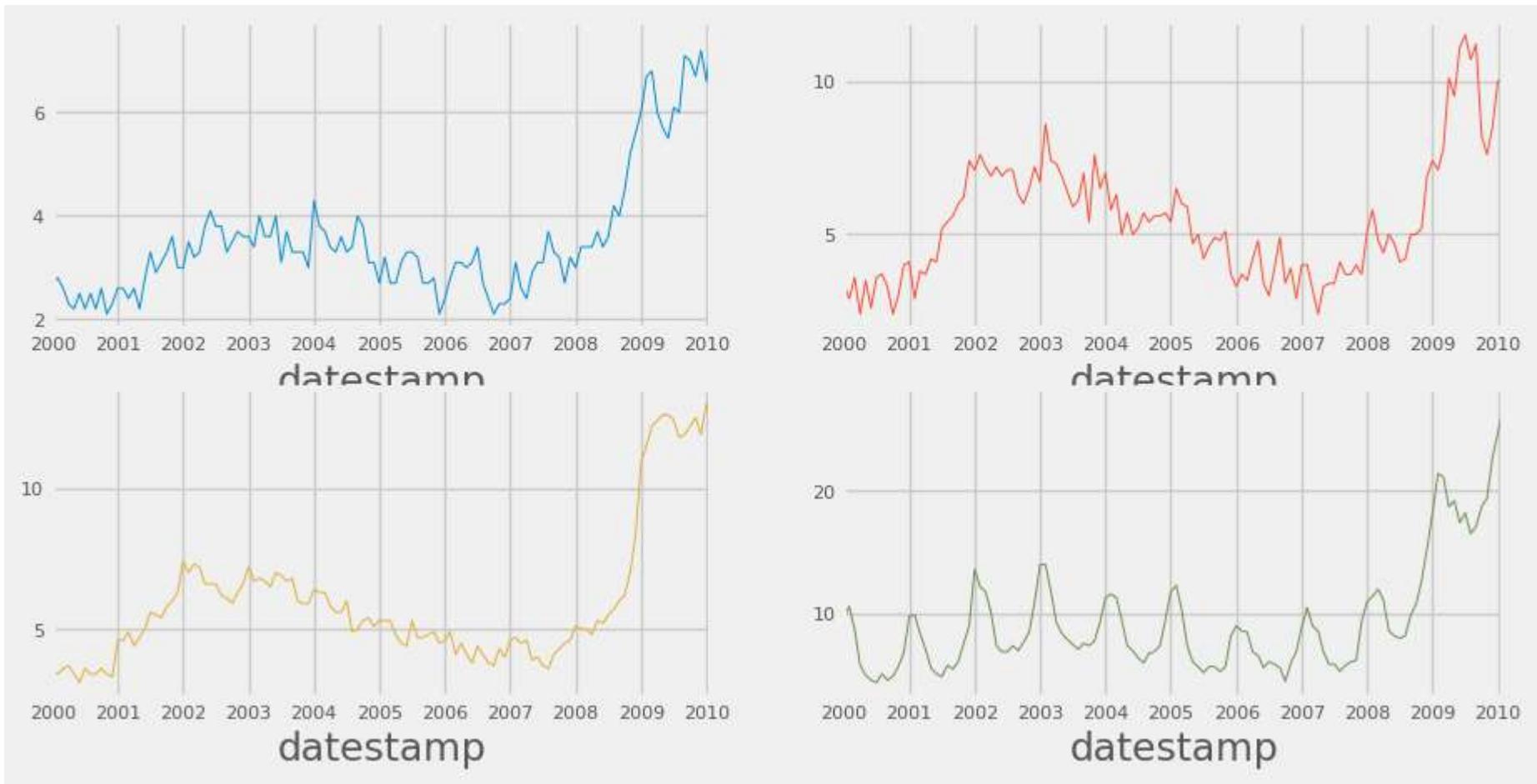
[122 rows x 16 columns]>

## Beyond summary statistics

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 [ ]: jobs_subset = jobs[['Finance', 'Information', 'Manufacturing', 'Construction']]  
  
# Print the first 5 rows of jobs_subset  
print(jobs_subset.head(5))  
  
# Create a faceted graph with 2 rows and 2 columns  
ax = jobs_subset.plot(subplots=True,  
                      layout=(2, 2),  
                      sharex=False,  
                      sharey=False,  
                      linewidth=0.7,  
                      fontsize=8,  
                      legend=False);
```

	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

When plotting the Finance, Information, Manufacturing and Construction time series of the jobs DataFrame, you observed a distinct increase in unemployment rates during 2001 and 2008. In general, time series plots can be made even more informative if you include additional annotations that emphasize specific observations or events. This allows you to quickly highlight parts of the graph to viewers, and can help infer what may have caused a specific event.

Recall that you have already set the datestamp column as the index of the jobs DataFrame, so you are prepared to directly annotate your plots with vertical or horizontal lines.

```
In [ ]: ax = jobs.plot(colormap='Spectral', fontsize=6, linewidth=0.8);

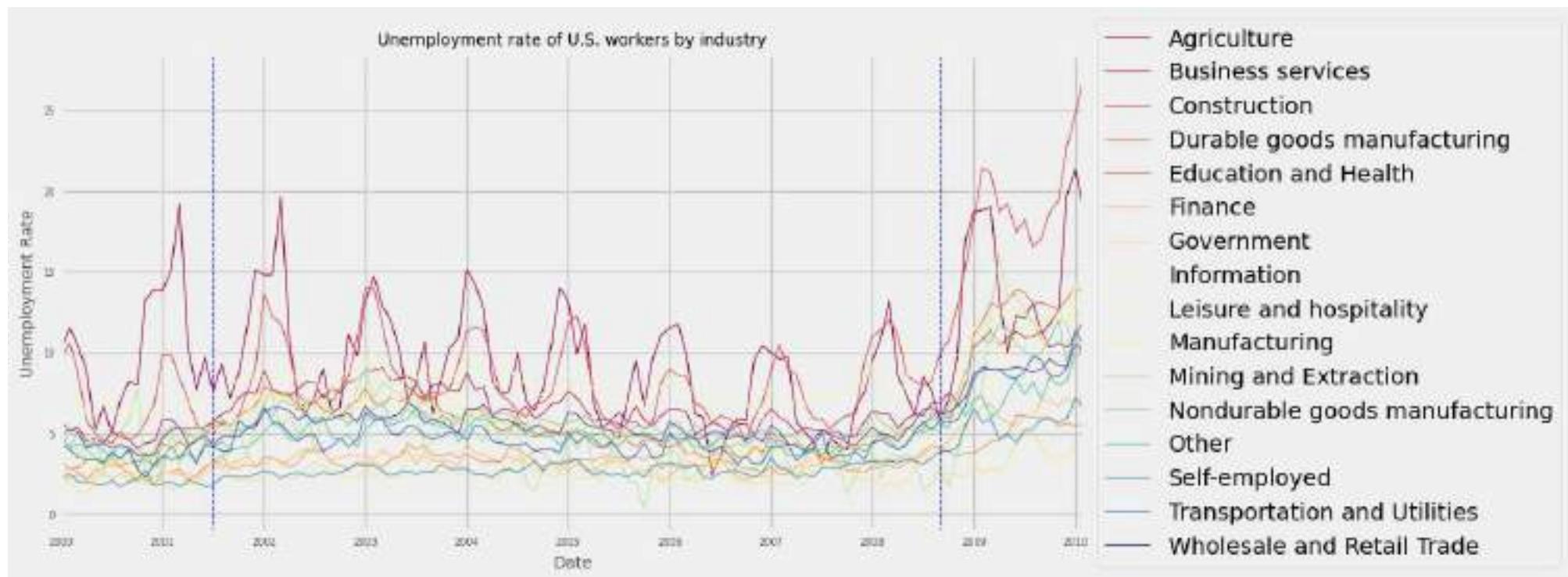
# Set Labels and Legend
ax.set_xlabel('Date', fontsize=10);
```

```

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))

# Annotate your plots with vertical lines
ax.axvline('2001-07-01', color='blue', linestyle='--', linewidth=0.8);
ax.axvline('2008-09-01', color='blue', linestyle='--', linewidth=0.8);

```



## Plot monthly and yearly trends

Like we saw in Chapter 2, when the index of a DataFrame is of the datetime type, it is possible to directly extract the day, month or year of each date in the index. As a reminder, you can extract the year of each date in the index using the `.index.year` attribute. You can then use the `.groupby()` and `.mean()` methods to compute the mean annual value of each time series in your DataFrame:

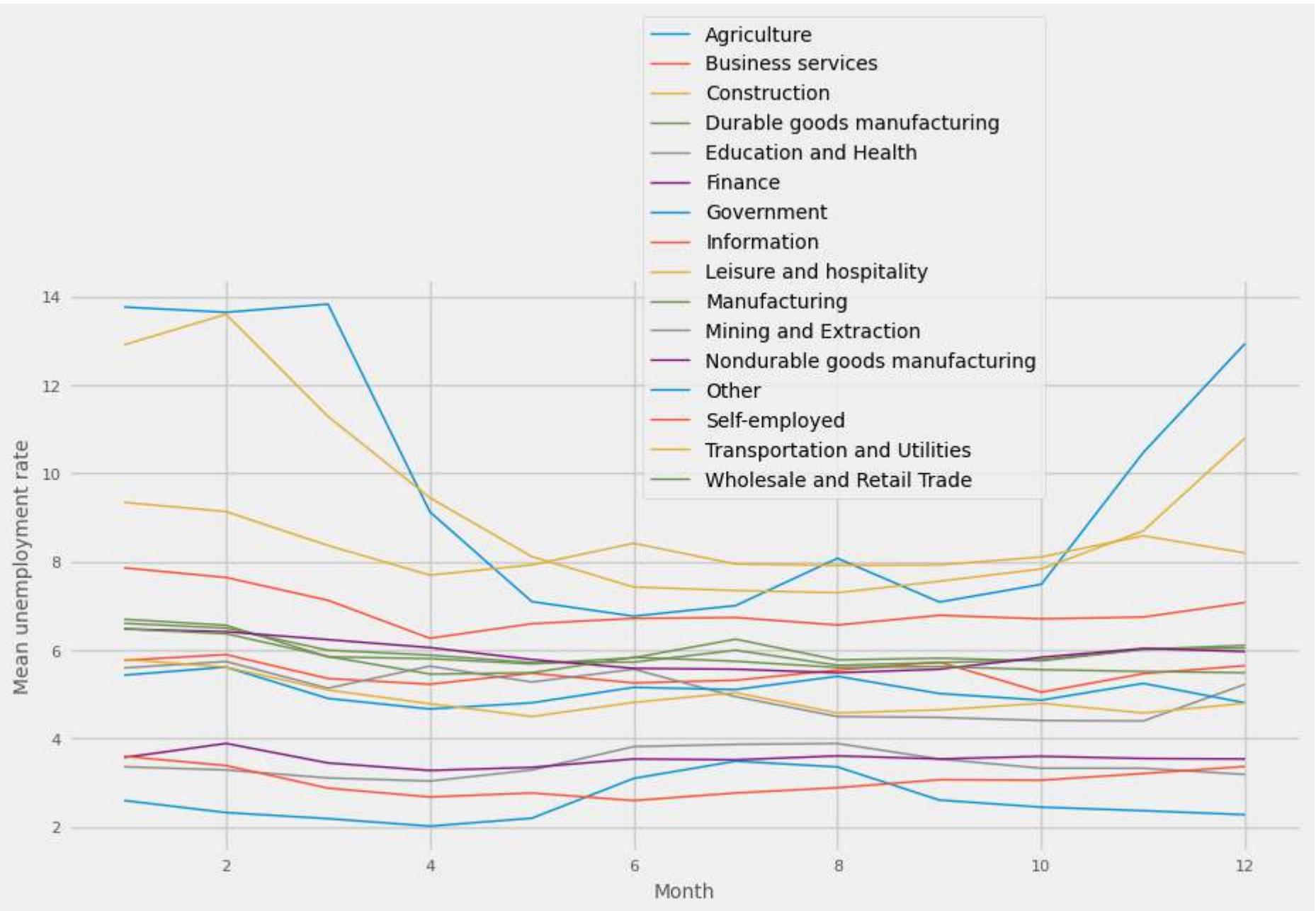
```
In [ ]: # Extract the month from the index of jobs
index_month = jobs.index.month
```

```
# Compute the mean unemployment rate for each month
jobs_by_month = jobs.groupby(index_month).mean()
```

```
# Plot the mean unemployment rate for each month
```

```
ax = jobs_by_month.plot(fontsize=8, linewidth=1);

# Set axis labels and legend
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()
```



```
In [ ]: index_year = jobs.index.year
```

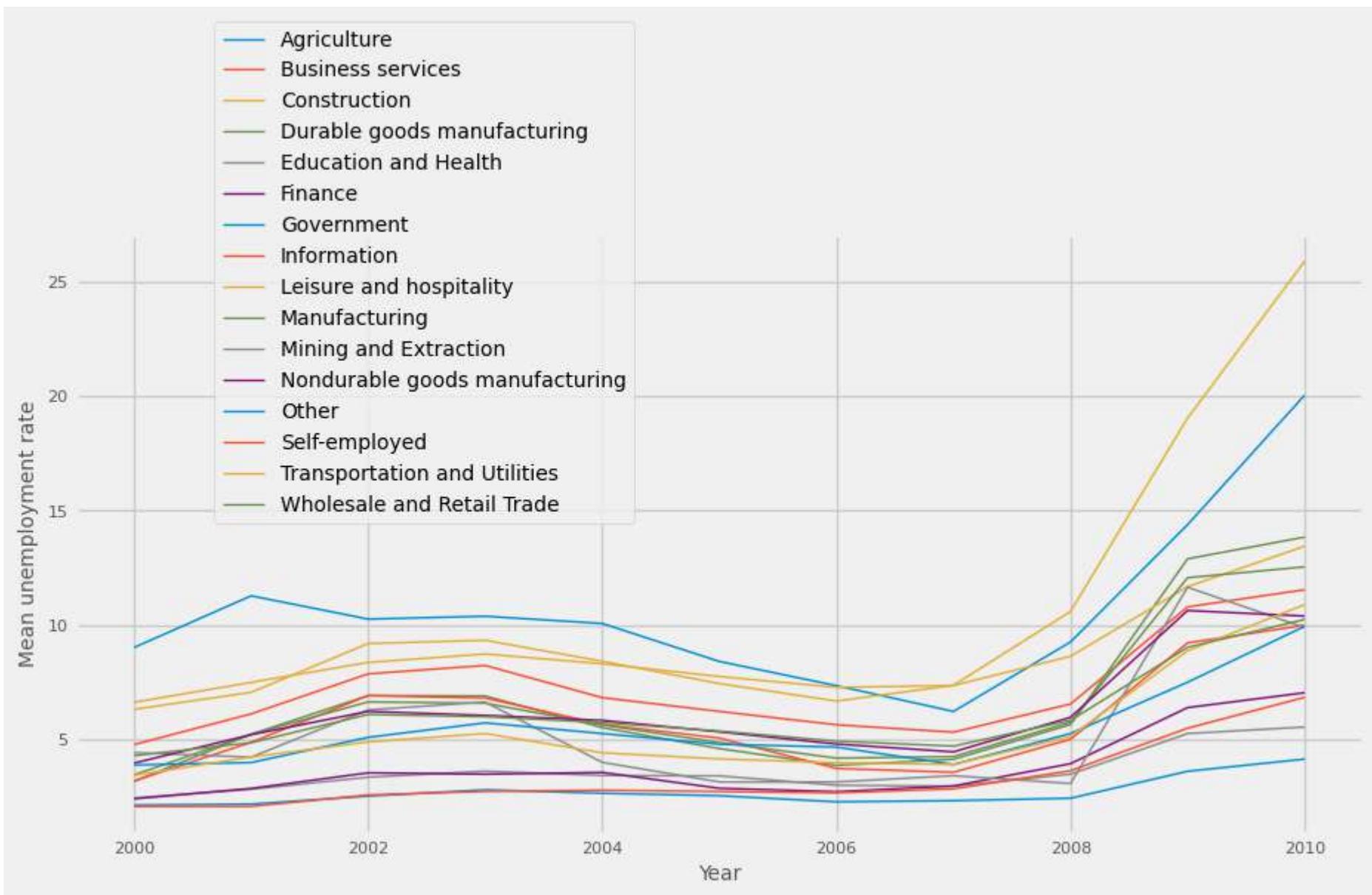
```
# Compute the mean unemployment rate for each year
jobs_by_year = jobs.groupby(index_year).mean()
```

```

# Plot the mean unemployment rate for each year
ax = jobs_by_year.plot(fontsize=8, linewidth=1);

# Set axis labels and legend
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);

```



## Decompose time series data

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 [ ]: import statsmodels.api as sm

# Initialize dictionary
jobs_decomp = {}

# Get the names of each time series in the DataFrame
jobs_names = jobs.columns

# run time series decomposition on each time series of the DataFrame
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.

```
In [ ]: jobs_seasonal = {}

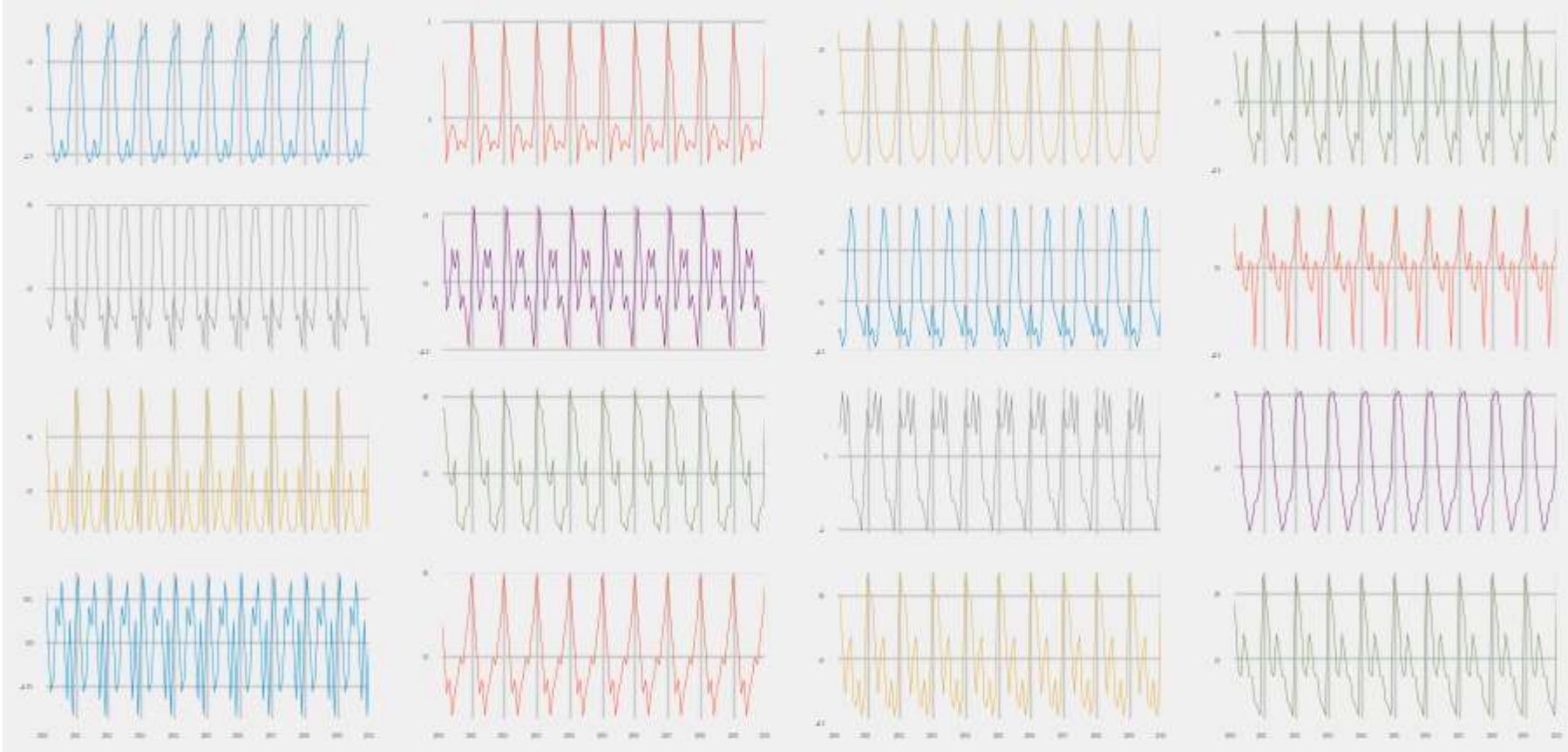
# Extract the seasonal values for the decomposition of each time series
for ts in jobs_names:
    jobs_seasonal[ts] = jobs_decomp[ts].seasonal

# 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);
```



### Compute correlations between time series

Correlations between multiple time series In the previous exercise, you extracted the seasonal component of each time series in the jobs DataFrame and stored those results in new DataFrame called seasonality\_df. In the context of jobs data, it can be interesting to compare seasonality behavior, as this may help uncover which job industries are the most similar or the most different.

This can be achieved by using the seasonality\_df DataFrame and computing the correlation between each time series in the dataset. In this exercise, you will leverage what you have learned in Chapter 4 to compute and create a clustermap visualization of the correlations between time series in the seasonality\_df DataFrame.

In [ ]:

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

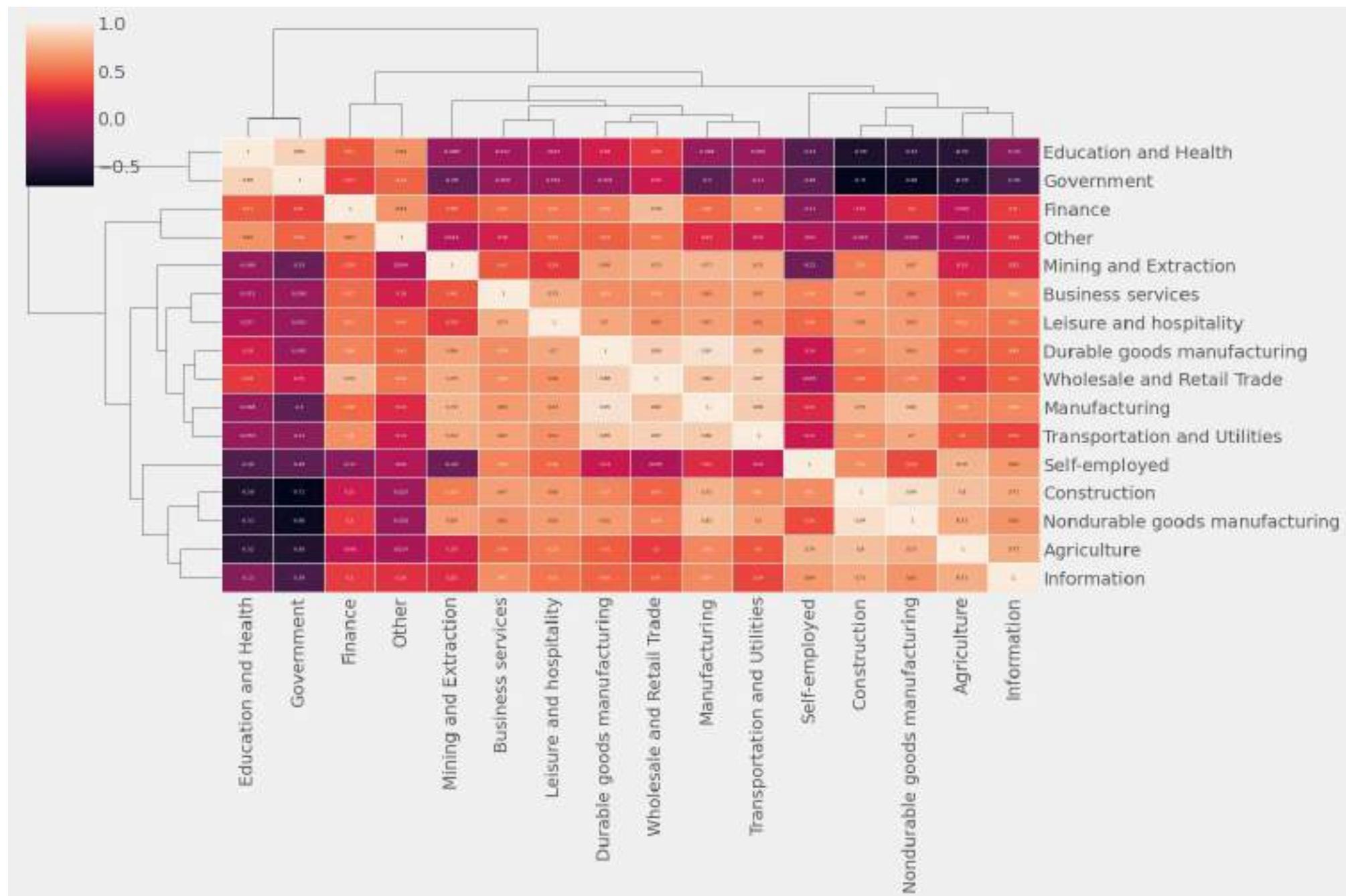
# Customize the clustermap of the seasonality_corr correlation matrix
fig = sns.clustermap(seasonality_corr,
                      annot=True,
                      annot_kws={"size": 4},
                      linewidths=.4,
                      figsize=(15, 10));
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(),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

#save the image of the clustermap of the seasonality_corr correlation matrix
plt.savefig('C:\\\\Users\\\\yeiso\\\\OneDrive - Douglas College\\\\0. DOUGLAS COLLEGE\\\\3. Fund Machine Learning\\\\0. Python Course DataCamp\\\\Course-'

plt.show()

# Print the correlation between the seasonalities of the Government and Education & Health industries
print("correlation calculated = ", seasonality_corr.loc['Government', 'Education and Health'])

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



correlation calculated = 0.8887968296115875

correlation typed = 0.89