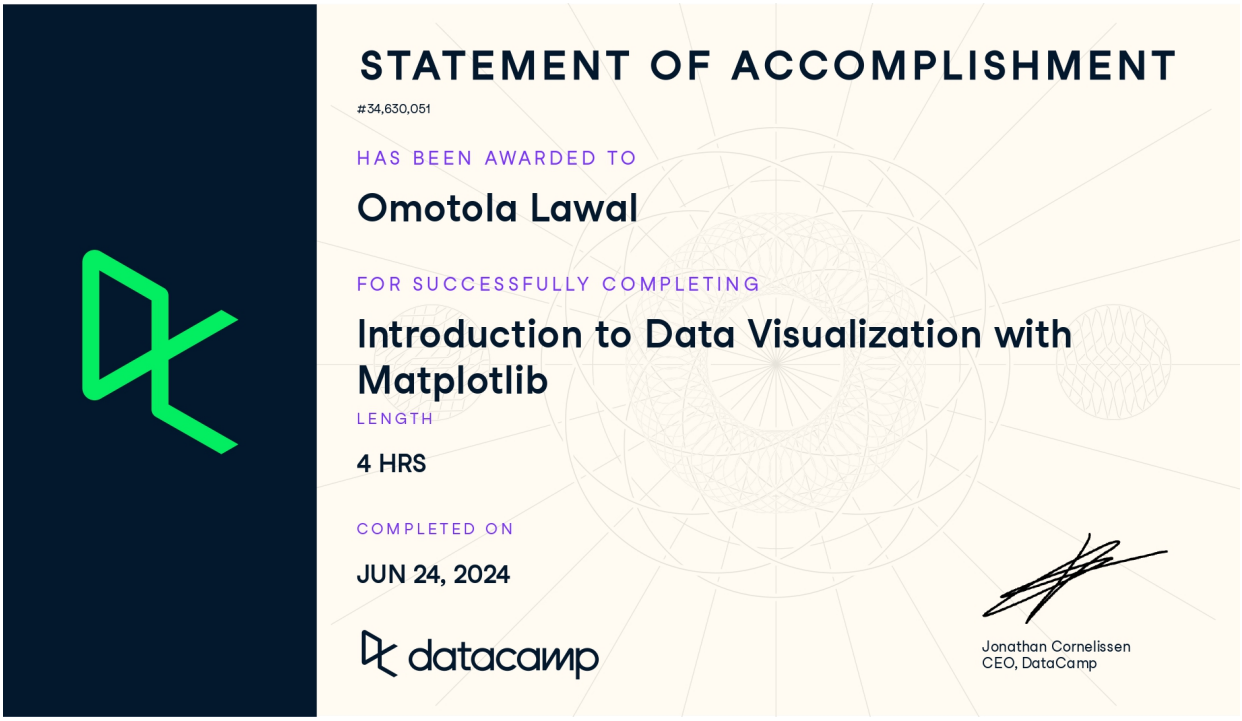
Introduction to Data Visualization with Matplotlib

Lawal's Note

2025-10-17

Table of contents



Chapter 1

Chapter 1.1: Introduction to Data Visualization with Matplotlib

---

Hello and welcome to this course on data visualization with Matplotlib! A picture is worth a thousand words. Data visualizations let you derive insights from data and let you communicate about the data with others.

Data visualization

For example, this visualization shows an animated history of an outbreak of Ebola in West Africa. The amount of information in this complex visualization is simply staggering! This visualization was created using Matplotlib, a Python library that is widely used to visualize data. There are many software libraries that visualize data. One of the main advantages of Matplotlib is that it gives you complete control over the properties of your plot. This allows you to customize and control the precise properties of your visualizations. At the end of this course, you will know not

only how to control your visualizations, but also how to create programs that automatically create visualizations based on your data.

Introducing the pyplot interface

There are many different ways to use Matplotlib. In this course, we will use the main object-oriented interface. This interface is provided through the `pyplot` submodule. Here, we import this submodule and name it `plt`. While using the name `plt` is not necessary for the program to work, this is a very strongly-followed convention, and we will follow it here as well. The `plt.subplots` command, when called without any inputs, creates two different objects: a Figure object and an Axes object. The Figure object is a container that holds everything that you see on the page. Meanwhile, the Axes is the part of the page that holds the data. It is the canvas on which we will draw with our data, to visualize it. It will show a Figure with empty Axes. No data has been added yet.

Adding data to axes

Let's add some data to our figure. Here is some data. This is a DataFrame that contains information about the weather in the city of Seattle in the different months of the year. The `"MONTH"` column contains the three-letter names of the months of the year. The `"monthly average normal temperature"` column contains the temperatures in these months, in Fahrenheit degrees, averaged over a ten-year period.

To add the data to the Axes, we call a plotting command. The plotting commands are methods of the Axes object. For example, here we call the method called `plot` with the `month` column as the first argument and the `temperature` column as the second argument. Finally, we call the `plt.show` function to show the effect of the plotting command. This adds a line to the plot. The horizontal dimension of the plot represents the months according to their order and the height of the line at each month represents the average temperature. The trends in the data are now much clearer than they were just by reading off the temperatures from the table.

Adding more data

If you want, you can add more data to the plot. For example, we also have a table that stores data about the average temperatures in the city of Austin, Texas. We add these data to the `axes` by calling the plot method again.

Putting it all together

Here is what all of the code to create this figure would then look like. First, we create the Figure and the Axes objects. We call the Axes method plot to add first the Seattle temperatures, and then the Austin temperatures to the Axes. Finally, we ask Matplotlib to show us the figure.

Exercise 1.1

Explore Datasets

- Using `austin_weather` and `seattle_weather`, create a Figure with an array of two Axes objects that share a y-axis range (`MONTHS` in this case). Plot Seattle's and Austin's `MLY-TAVG-NORMAL` (for average temperature) in the top Axes and plot their `MLY-PRCP-NORMAL` (for average precipitation) in the bottom axes. The cities should have different colors and the line style should be different between precipitation and temperature. Make sure to label your viz!
- Using `climate_change`, create a twin Axes object with the shared x-axis as time. There should be two lines of different colors not sharing a y-axis: `co2` and `relative_temp`. Only include dates from the 2000s and annotate the first date at which `co2` exceeded 400.
- Create a scatter plot from `medals` comparing the number of Gold medals vs the number of Silver medals with each point labeled with the country name.
- Explore if the distribution of `Age` varies in different sports by creating histograms from `summer_2016`.
- Try out the different Matplotlib styles available and save your visualizations as a PNG file.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)
summer_2016 = pd.read_csv('datasets/summer2016.csv')
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")
```

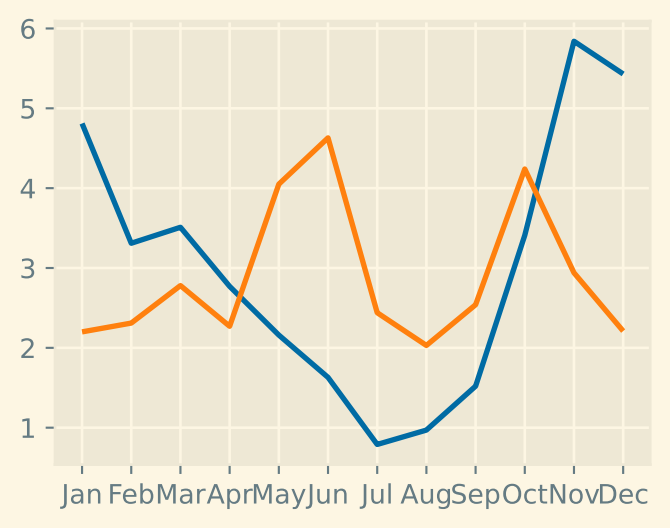
```
# Show first 5 dataset in weather
print(weather.head(5))
print(austin_weather.head(5))

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Create a Figure and an Axes with plt.subplots
fig, ax = plt.subplots()

# Add data from the seattle_weather DataFrame by calling the Axes plot method.
ax.plot(seattle_weather["MONTH"], seattle_weather['MLY-PRCP-NORMAL'])

# Add data from the austin_weather DataFrame in a similar manner and call plt.show to show the re
ax.plot(austin_weather['MONTH'], austin_weather['MLY-PRCP-NORMAL'])

# Call the show function
plt.show()
```

```
          STATION                 NAME  MLY-CLDD-BASE45  MLY-CLDD-BASE50  \
DATE
1     USC00456295  PALMER 3 ESE, WA US             13.0              1.0
2     USC00456295  PALMER 3 ESE, WA US             23.0              3.0
3     USC00456295  PALMER 3 ESE, WA US             50.0             11.0
4     USC00456295  PALMER 3 ESE, WA US            108.0             40.0
5     USC00456295  PALMER 3 ESE, WA US            255.0            130.0

      MLY-CLDD-BASE55  MLY-CLDD-BASE57  MLY-CLDD-BASE60  MLY-CLDD-BASE70  \
DATE
1             -7777.0              0.0              0.0              0.0
2             -7777.0          -7777.0              0.0              0.0
3                 1.0          -7777.0          -7777.0              0.0
4                12.0              7.0              3.0          -7777.0
5                56.0             38.0             19.0              1.0

      MLY-CLDD-BASE72  MLY-CLDD-NORMAL  ...  MLY-TMIN-AVGNDS-LSTH060  \
DATE                                    ...
1                 0.0              0.0  ...                    310.0
2                 0.0              0.0  ...                    280.0
3                 0.0              0.0  ...                    310.0
4                 0.0          -7777.0  ...                    300.0
```

```
5            -7777.0             5.0  ...                    310.0

      MLY-TMIN-AVGNDS-LSTH070  MLY-TMIN-NORMAL  MLY-TMIN-PRBOCC-LSTH016  \
DATE
1                       310.0             34.0                    168.0
2                       280.0             33.9                    132.0
3                       310.0             35.9                      5.0
4                       300.0             38.3                      0.0
5                       310.0             43.5                      0.0

      MLY-TMIN-PRBOCC-LSTH020  MLY-TMIN-PRBOCC-LSTH024  \
DATE
1                       327.0                    615.0
2                       275.0                    515.0
3                        47.0                    111.0
4                         0.0                      5.0
5                         0.0                      0.0

      MLY-TMIN-PRBOCC-LSTH028  MLY-TMIN-PRBOCC-LSTH032  \
DATE
1                       877.0                    987.0
2                       800.0                    963.0
3                       393.0                    848.0
4                       123.0                    571.0
5                         0.0                     83.0

      MLY-TMIN-PRBOCC-LSTH036  MLY-TMIN-STDDEV
DATE
1                      1000.0              2.8
2                      1000.0              3.1
3                      1000.0              2.1
4                       978.0              2.0
5                       610.0              2.0

[5 rows x 79 columns]
          STATION                                          NAME  \
DATE
1     USW00013904  AUSTIN BERGSTROM INTERNATIONAL AIRPORT, TX US
2     USW00013904  AUSTIN BERGSTROM INTERNATIONAL AIRPORT, TX US
3     USW00013904  AUSTIN BERGSTROM INTERNATIONAL AIRPORT, TX US
4     USW00013904  AUSTIN BERGSTROM INTERNATIONAL AIRPORT, TX US
5     USW00013904  AUSTIN BERGSTROM INTERNATIONAL AIRPORT, TX US

      MLY-CLDD-BASE45  MLY-CLDD-BASE50  MLY-CLDD-BASE55  MLY-CLDD-BASE57  \
DATE
```

```
1                 190              103               50               35
2                 228              132               68               49
3                 446              306              185              146
4                 668              519              373              318
5                 936              781              626              564

      MLY-CLDD-BASE60  MLY-CLDD-BASE70  MLY-CLDD-BASE72  MLY-CLDD-NORMAL  ...  \
DATE                                                                        ...
1                  18                1            -7777                5  ...
2                  29                3                1               11  ...
3                  98               13                6               42  ...
4                 240               53               32              130  ...
5                 471              181              134              319  ...

      MLY-TMIN-AVGNDS-LSTH060  MLY-TMIN-AVGNDS-LSTH070  MLY-TMIN-NORMAL  \
DATE
1                         302                      310             36.3
2                         264                      280             39.4
3                         270                      308             46.6
4                         193                      287             54.7
5                          89                      250             63.7

      MLY-TMIN-PRBOCC-LSTH016  MLY-TMIN-PRBOCC-LSTH020  \
DATE
1                         298                      570
2                         103                      327
3                          10                       73
4                           0                        0
5                           0                        0

      MLY-TMIN-PRBOCC-LSTH024  MLY-TMIN-PRBOCC-LSTH028  \
DATE
1                         839                      967
2                         614                      867
3                         242                      494
4                           0                       48
5                           0                        0

      MLY-TMIN-PRBOCC-LSTH032  MLY-TMIN-PRBOCC-LSTH036  MLY-TMIN-STDDEV
DATE
1                         997                     1000              2.9
2                         973                      999              3.2
3                         761                      928              3.6
4                         189                      453              4.1
5                           0                        0              2.5
```

```
[5 rows x 66 columns]
```



Chapter 1.2: Customizing your plots

Now that you know how to add data to a plot, let's start customizing your plots.

Customizing data appearance

First let's customize the appearance of the data in the plot. Here is the code that you previously used to plot the data about the weather in Seattle. One of the things that you might want to improve about this plot is that the data appears to be continuous, but it was actually only measured in monthly intervals. A way to indicate this would be to add markers to the plot that show us where the data exists and which parts are just lines that connect between the data points.

Adding markers

The plot method takes an optional keyword argument, marker, which lets you indicate that you are interested in adding markers to the plot and also what kind of markers you'd like. For example, passing the lower-case letter "o" indicates that you would like to use circles as markers.

Choosing markers

If you were to pass a lower case letter "v" instead, you would get markers shaped like triangles pointing downwards. To see all the possible marker styles, you can visit this page in the Matplotlib online documentation. In these versions of the plot, the measured data appears as markers of some shape, and it becomes more apparent that the lines are just connectors between them.

Setting the linestyle

But you can go even further to emphasize this by changing the appearance of these connecting lines. This is done by adding the linestyle keyword argument. Here two dashes are used to indicate that the line should be dashed. Like marker shapes, there are a few linestyles you can choose from, listed in this documentation page.

Eliminating lines with linestyle

You can even go so far as to eliminate the lines altogether, by passing the string `"None"` as input to this keyword argument.

Choosing color

Finally, you can choose the color that you would like to use for the data. For example, here we've chosen to show this data in red, indicated by the letter `"r"`.

Customizing the axes labels

Another important thing to customize are the axis labels. If you want your visualizations to communicate properly you need to always label the axes. This is really important but is something that is often neglected. In addition to the plot method, the Axes object has several methods that start with the word set. These are methods that you can use to change certain properties of the object, before calling show to display it. For example, there is a `set_xlabel` method that you can use to set the value of the label of the x-axis. Note that we capitalize axis labels as we would capitalize a sentence, where only the first word is always capitalized and subsequent words are capitalized only if they are proper nouns. If you then call `plt.show` you will see that the axis now has a label that indicates that the values on the x-axis denote time in months.

Setting the y axis label

Similarly, a `set_ylabel` method customizes the label that is associated with the y-axis. Here, we set the label to indicate that the height of the line in each month indicates the average temperature in that month.

Adding a title

Finally, you can also add a title to your Axes using the `set_title` method. This adds another source of information about the data to provide context for your visualization.

Exercise 1.2

- Call `ax.plot` to plot `"MLY-PRCP-NORMAL"` against `"MONTHS"` in both DataFrames.
- Pass the color key-word arguments to these commands to set the color of the Seattle data to blue ('b') and the Austin data to red ('r').
- Pass the marker key-word arguments to these commands to set the Seattle data to circle markers ('o') and the Austin markers to triangles pointing downwards ('v').

```
fig, ax =  plt.subplots()
ax.plot(seattle_weather["MONTH"], seattle_weather["MLY-PRCP-NORMAL"], color = "b", marker = "o",
ax.plot(austin_weather["MONTH"], austin_weather["MLY-PRCP-NORMAL"], color = "r", marker = "v", li

# Use the set_xlabel method to add the label: "Time (months)".
ax.set_xlabel("Time (months)")

# Use the set_ylabel method to add the label: "Precipitation (inches)".
ax.set_ylabel("Precipitation (inches)")

# Use the set_title method to add the title: "Weather patterns in Austin and Seattle".
ax.set_title("Weather patterns in Austin and Seattle")
plt.show()
```

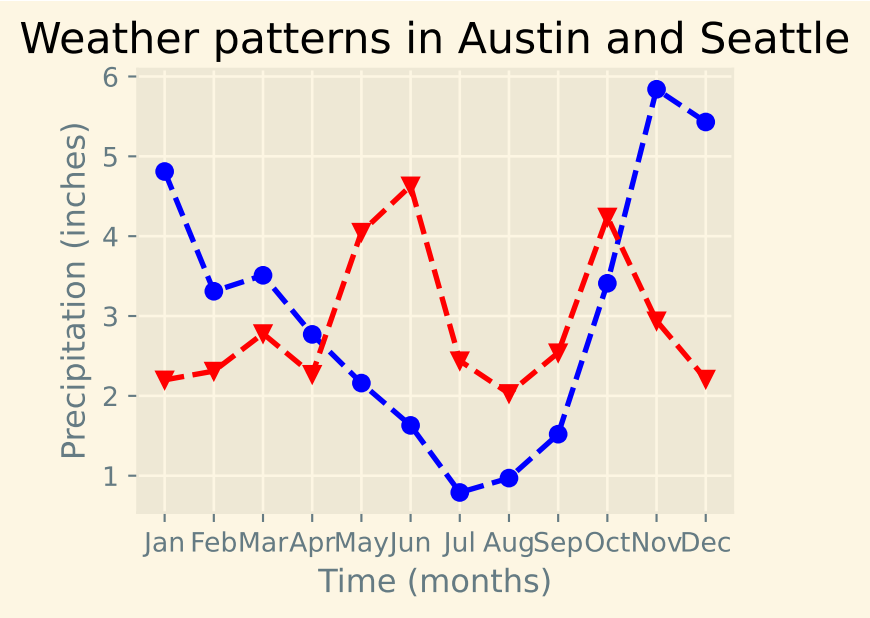


Figure 1: Average temperature in Seattle anD Austin.

Chapter 1.3: Small multiples

In some cases, adding more data to a plot can make the plot too busy, obscuring patterns rather than revealing them.

Adding data

For example, let's explore the data we have about weather in Seattle. Here we plot average precipitation in Seattle during the course of the year. But let's say that we are also interested in the range of values. We add the 25th percentile and the 75th percentile of the precipitation in dashed lines above and below the average. What would happen if we compared this to Austin?

And more data

This code adds the data from Austin to the plot. When we display the plot, it's a bit of a mess. There's too much data in this plot. One way to overcome this kind of mess is to use what are called small multiples. These are multiple small plots that show similar data across different conditions. For example, precipitation data across different cities.

Small multiples with plt.subplots

In Matplotlib, small multiples are called `sub-plots`. That is also the reason that the function that creates these is called `subplots`. Previously, we called this function with no inputs. This creates one subplot. Now, we'll give it some inputs. Small multiples are typically arranged on the page as a grid with rows and columns. Here, we are creating a Figure object with three rows of subplots, and two columns. This is what this would look like before we add any data to it. In this case, the variable `ax` is no longer only one Axes object.

Adding data to subplots

Instead, it is an array of Axes objects with a shape of 3 by 2. To add data, we would now have to index into this object and call the plot method on an element of the array.

Subplots with data

There is a special case for situations where you have only one row or only one column of plots. In this case, the resulting array will be one-dimensional and you will only have to provide one index to access the elements of this array. For example, consider what we might do with the rainfall data that we were plotting before. We create a figure and an array of Axes objects with two rows and one column. We address the first element in this array, which is the top sub-plot, and add

the data for Seattle to this plot. Then, we address the second element in the array, which is the bottom plot, and add the data from Austin to it. We can add a y-axis label to each one of these. Because they are one on top of the other, we only add an x-axis label to the bottom plot, by addressing only the second element in the array of Axes objects. When we show this, we see that the data are now cleanly presented in a way that facilitates the direct comparison between the two cities. One thing we still need to take care of is the range of the y-axis in the two plots, which is not exactly the same. This is because the highest and lowest values in the two datasets are not identical.

Sharing the y-axis range

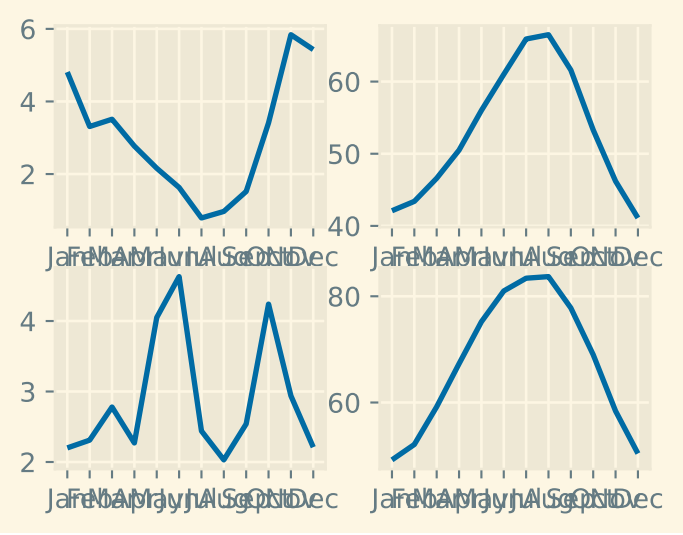
To make sure that all the subplots have the same range of y-axis values, we initialize the figure and its subplots with the key-word argument `sharey` set to `True`. This means that both subplots will have the same range of y-axis values, based on the data from both datasets. Now the comparison across datasets is more straightforward.

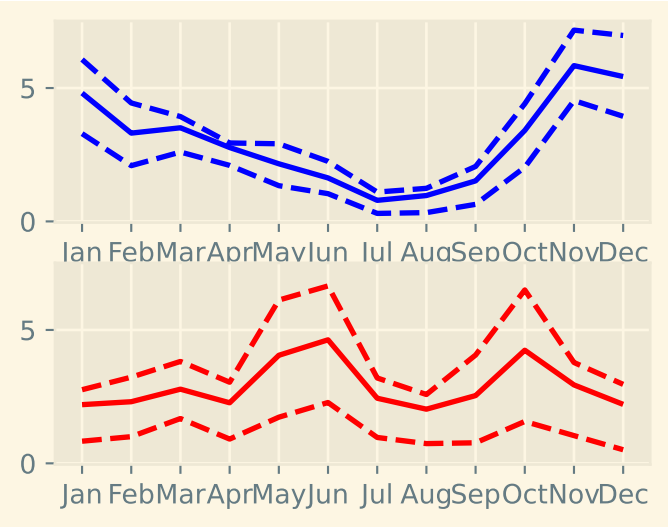
Exercise 1.3

1. Create a Figure and an array of subplots with 2 rows and 2 columns.
2. Addressing the top left Axes as index 0, 0, plot the Seattle precipitation.
3. In the top right (index 0,1), plot Seattle temperatures.
4. In the bottom left (1, 0) and bottom right (1, 1) plot Austin precipitations and temperatures.
5. Create a Figure with an array of two Axes objects that share their y-axis range.
6. Plot Seattle's `"MLY-PRCP-NORMAL"` in a solid blue line in the top Axes.
7. Add Seattle's `"MLY-PRCP-25PCTL"` and `"MLY-PRCP-75PCTL"` in dashed blue lines to the top Axes.
8. Plot Austin's `"MLY-PRCP-NORMAL"` in a solid red line in the bottom Axes and the `"MLY-PRCP-25PCTL"` and `"MLY-PRCP-75PCTL"` in dashed red lines.

```
# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# 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 the Seattle precipitation.
ax[0, 0].plot(seattle_weather['MONTH'], seattle_weather['MLY-PRCP-NORMAL'])

# In the top right (index 0,1), plot Seattle temperatures.
```

```
ax[0, 1].plot(seattle_weather['MONTH'], seattle_weather['MLY-TAVG-NORMAL'])

# In the bottom left (1, 0) and bottom right (1, 1) plot Austin precipitations and temperatures.
ax[1, 0].plot(austin_weather['MONTH'], austin_weather['MLY-PRCP-NORMAL'])

# In the bottom right (1, 1) plot month and Austin temperatures
ax[1, 1].plot(austin_weather['MONTH'], austin_weather['MLY-TAVG-NORMAL'])


# Small multiples with shared y axis
# Create a Figure with an array of two Axes objects that share their y-axis range.
fig, ax = plt.subplots(2, 1, sharey=True)

# Plot Seattle's "MLY-PRCP-NORMAL" in a solid blue line in the top Axes.
# Add Seattle's "MLY-PRCP-25PCTL" and "MLY-PRCP-75PCTL" in dashed blue lines to the top Axes.
ax[0].plot(seattle_weather['MONTH'], seattle_weather['MLY-PRCP-NORMAL'] , color = 'b')
ax[0].plot(seattle_weather['MONTH'], seattle_weather['MLY-PRCP-25PCTL'], color = 'b', linestyle =
ax[0].plot(seattle_weather['MONTH'], seattle_weather['MLY-PRCP-75PCTL'], color = 'b', linestyle =

# Plot Austin's "MLY-PRCP-NORMAL" in a solid red line in the bottom Axes and the "MLY-PRCP-25PCTL
ax[1].plot(austin_weather['MONTH'], austin_weather['MLY-PRCP-NORMAL'], color = 'r')
ax[1].plot(austin_weather['MONTH'], austin_weather['MLY-PRCP-25PCTL'], color = 'r',
linestyle = '--')
ax[1].plot(austin_weather['MONTH'], austin_weather['MLY-PRCP-75PCTL'], color = 'r',
linestyle = '--')

plt.show()
```





Chapter 2

Chapter 2.1: Plotting time-series data

Many kinds of data are organized as time-series, and visualizations of time-series are an excellent tool to detect patterns in the data.

Time-series data

For example, the weather dataset that we used in the previous chapter is a relatively simple example of time-series data. Continuous variables, such as precipitation or temperatures are organized in our data table according to a time-variable, the months of the year. In this chapter, we'll dive deeper into using Matplotlib to visualize time-series data.

Climate change time-series

Let's look at a more complex dataset, that contains records of the change in climate in the last half a century or so. The data is in a CSV file with three columns. The `"date"` column indicates when the recording was made and is stored in the year-month-date format. A measurement was taken on the 6th day of every month from 1958 until 2016. The column `"co2"` contains measurements of the carbon dioxide in the atmosphere. The number shown in each row is parts-per-million of carbon dioxide. The column `"relative_temp"` denotes the temperature measured at this date, relative to a baseline which is the average temperature in the first ten years of measurements. If we want pandas to recognize that this is a time-series, we'll need to tell it to parse the `"date"` column as a date. To use the full power of pandas indexing facilities, we'll also designate the date column as our index by using the `index_col` key-word argument.

DateTimeIndex

This is the index of our DataFrame. It's a DateTimeIndex object with 706 entries, one for each measurement. It has a DateTime datatype and Matplotlib will recognize that this is a variable that represents time. This will be important in a little bit.

Time-series data

The other two columns in the data are stored as regular columns of the DataFrame with a floating point data-type, which will allow us to calculate on them as continuous variables. There are a few points in the CO2 data that are stored as `NaNs` or `Not-a-Number`. These are missing values where measurements were not taken.

Plotting time-series data

To start plotting the data, we import Matplotlib and create a Figure and Axes. Next, we add the data to the plot. We add the index of our DataFrame for the x-axis and the `"co2"` column for the y-axis. We also label the x- and y-axes. Matplotlib automatically chooses to show the time on the x-axis as years, with intervals of 10 years. The data visualization tells a clear story: there are some small seasonal fluctuations in the amount of CO2 measured, and an overall increase in the amount of CO2 in the atmosphere from about 320 parts per million to about 400 parts per million.

Zooming in on a decade

We can select a decade of the data by slicing into the DataFrame with two strings that delimit the start date and end date of the period that we are interested in. When we do that, we get the plot of a part of the time-series encompassing only ten years worth of data. Matplotlib also now knows to label the x-axis ticks with years, with an interval of one year between ticks. Looking at this data, you'll also notice that the missing values in this time series are represented as breaks in the line plotted by Matplotlib.

Zooming in on one year

Zooming in even more, we can select the data from one year. Now the x-axis automatically denotes the months within that year.

Exercise 2.1

---

1. Read in the data from a CSV file called `'climate_change.csv'` using `pd.read_csv`.

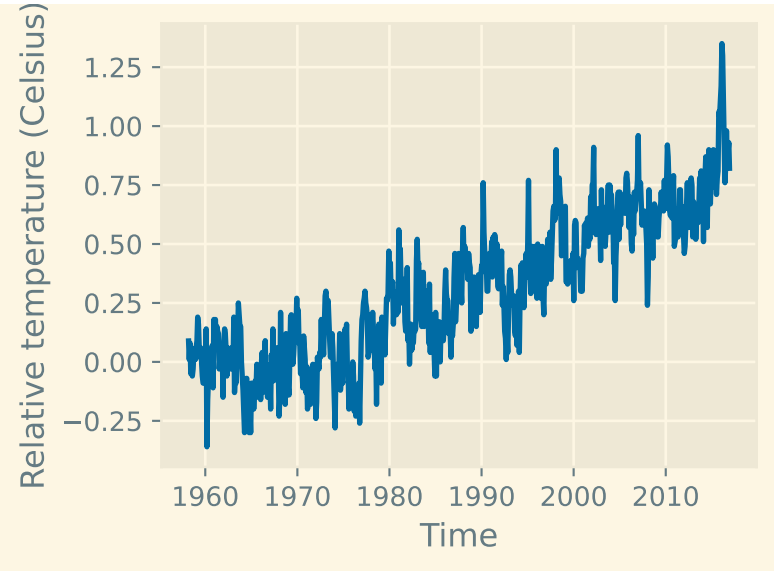
- Use the `parse_dates` key-word argument to parse the `"date"` column as dates.
- Use the `index_col` key-word argument to set the `"date"` column as the `index`.

2. Plot time-series data. Add the data from `climate_change` to the plot: use the DataFrame index for the `x` value and the `"relative_temp"` column for the `y` values.
3. Set the x-axis label to `'Time'`.
4. Set the y-axis label to `'Relative temperature (Celsius)'`.
5. Show the figure

```
# Read in the data from a CSV file called 'climate_change.csv' using pd.read_csv.
# Use the parse_dates key-word argument to parse the "date" column as dates.
# Use the index_col key-word argument to set the "date" column as the index.
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date

# Read the top 5 rows
print(climate_change.head(5))

# Plot time-series data
# Add the data from climate_change to the plot: use the DataFrame index for the x value and the "
fig, ax = plt.subplots()
```

```
# Add the data from climate_change to the plot: use the DataFrame index for the x value and the "
ax.plot(climate_change.index, climate_change['relative_temp'])

# Set the x-axis label to 'Time'.
ax.set_xlabel('Time')

# Set the y-axis label to 'Relative temperature (Celsius)'.
ax.set_ylabel('Relative temperature (Celsius)')

# Show the figure
plt.show()
```

```
               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
```



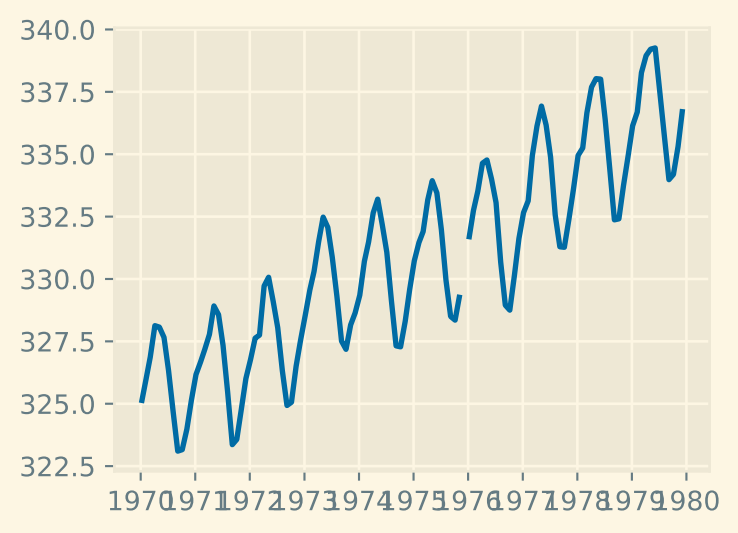
Exercise 2.1.1

---

1. Use `plt.subplots` to create a Figure with one Axes called `fig` and `ax`, respectively.

2. Create a variable called seventies that includes all the data between `"1970-01-01"` and `"1979-12-31"`.
3. Add the data from seventies to the plot: use the DataFrame index for the x value and the `"co2"` column for the y values.
4. Show the figure.

```
# Use plt.subplots to create a Figure with one Axes called fig and ax, respectively.
fig, ax = plt.subplots()

# Create a variable called seventies that includes all the data between "1970-01-01" and "1979-12
seventies = climate_change["1970-01-01":"1979-12-31"]

# Add the data from seventies to the plot: use the DataFrame index for the x value and the "co2"
ax.plot(seventies.index, seventies["co2"])

# Show the figure
plt.show()
```



Chapter 2.2: Plotting time-series with different variables

To relate two time-series that coincide in terms of their times, but record the values of different variables, we might want to plot them on the same Axes.

Plotting two time-series together

For example, consider the `climate_change` DataFrame that we've seen previously. This DataFrame contains two variables measured every month from 1958 until 2016: levels of carbon dioxide and relative temperatures.

As before, we can create a Figure and Axes and add the data from one variable to the plot. And we can add the data from the other variable to the plot. We also add axis labels and show the plot. But this doesn't look right. The line for carbon dioxide has shifted upwards, and the line for relative temperatures looks completely flat. The problem is that the scales for these two measurements are different.

Using twin axes

You've already seen how you could plot these time-series in separate sub-plots. Here, we're going to plot them in the same sub-plot, using two different y-axis scales. Again, we start by adding the first variable to our Axes. Then, we use the `twinx` method to create a twin of this Axes. This means that the two Axes share the same x-axis, but the y-axes are separate. We add the other variable to this second Axes object and show the figure. There is one y-axis scale on the left, for the carbon dioxide variable, and another y-axis scale to the right for the temperature variable. Now it will show the fluctuations in temperature more clearly. But this is still not quite right. The two lines have the same color. Let's take care of that.

Separating variables by color

To separate the variables, we'll encode each one with a different color. We add color to the first variable, using the `color` key-word argument in the call to the plot function. We also set the color in our call to the `set_ylabel` function. We repeat this in our calls to plot and `set_ylabel` from the twin Axes object. In the resulting figure, each variable has its own `color` and the y-axis labels clearly tell us which scale belongs to which variable.

Coloring the ticks

We can make encoding by color even more distinct by setting not only the color of the y-axis labels but also the y-axis ticks and the y-axis tick labels. This is done by adding a call to the `tick_params` method. This method takes either y or x as its first argument, pointing to the fact that we are modifying the parameters of the y-axis ticks and tick labels. To change their color, we use the `colors` key-word argument, setting it to `blue`. Similarly, we call the `tick_params` method from the twin Axes object, setting the colors for these ticks to `red`.

Coloring both the axis label and ticks makes it clear which scale to use with which variable. This seems like a useful pattern. Before we move on, let's implement this as a function that we can reuse.

A function that plots time-series

We use the `def` key-word to indicate that we are defining a function called `plot_timeseries`. This function takes as arguments an Axes object, `x` and `y` variables to plot, a `color` to associate with this variable, as well as x-axis and y-axis labels. The function calls the methods of the Axes object that we have seen before: `plot`, `set_xlabel`, `set_ylabel`, and `tick_params`.

Using our function

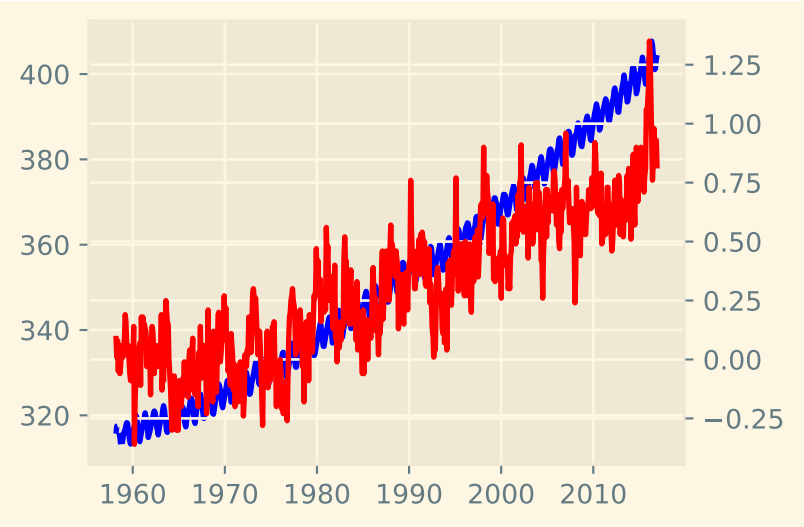
Using our function, we don't have to repeat these calls, and the code is simpler.

Exercise 2.2

---

1. Initalize a Figure and Axes.
2. Plot the `CO2` variable in blue.
3. Create a twin Axes that shares the x-axis.
4. Plot the relative temperature in red.
5. Show the figure.

```
# Initalize a Figure and Axes
fig, ax = plt.subplots()

# Plot the CO2 variable in blue
ax.plot(climate_change.index, climate_change['co2'], color='blue')

# 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='red')

plt.show()
```



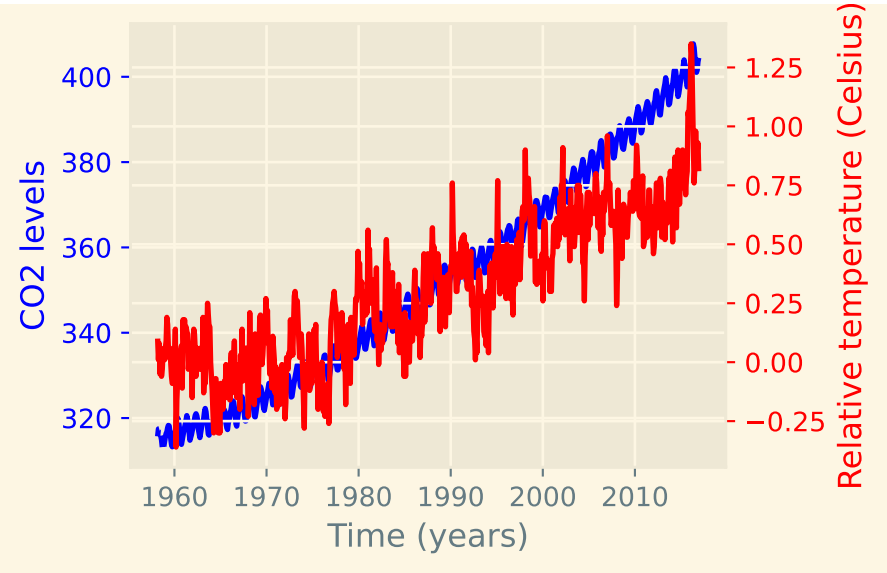
Exercise 2.2.1

Defining a function that plots time-series data. Once you realize that a particular section of code that you have written is useful, it is a good idea to define a function that saves that section of code for you, rather than copying it to other parts of your program where you would like to use this code.

1. Define a function called `plot_timeseries`, with axes, x, y, color, xlabel and ylabel as parameters.
2. In the provided ax object, use the function `plot_timeseries` to plot the `"co2"` column in `blue`, with the x-axis label `"Time (years)"` and y-axis label `"CO2 levels"`.
3. Use the `ax.twinx` method to add an Axes object to the figure that shares the x-axis with `ax`.
4. Use the function `plot_timeseries` to add the data in the `"relative_temp"` column in `red` to the twin Axes object, with the x-axis label `"Time (years)"` and y-axis label `"Relative temperature (Celsius)"`.

```
# Defining a function that plots time-series data
# Once you realize that a particular section of code that you have written is useful, it is a goo

# 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)

# In the provided ax object, use the function plot_timeseries to plot the "co2" column in blue, u
fig, ax = plt.subplots()
plot_timeseries(ax, climate_change.index, climate_change['co2'], "blue","Time (years)", "CO2 leve

# Use the ax.twinx method to add an Axes object to the figure that shares the x-axis with ax.
ax2 = ax.twinx()

# Use the function plot_timeseries to add the data in the "relative_temp" column in red to the tu
plot_timeseries(ax2, climate_change.index, climate_change['relative_temp'], "red", "Time (years)"

plt.show()
```



Chapter 2.3: Annotating time-series data

One important way to enhance a visualization is to add annotations. Annotations are usually small pieces of text that refer to a particular part of the visualization, focusing our attention on some feature of the data and explaining this feature.

Time-series data

For example, consider the data that we saw in previous videos in this chapter. This data shows the levels of measured carbon dioxide in the atmosphere over a period of more than 50 years in blue and the relative temperature over the same period of time in red. That's a lot of data, and, when presenting it, you might want to focus attention on a particular aspect of this data.

Annotation

One way to draw attention to part of a plot is by annotating it. This means drawing an arrow that points to part of the plot and being able to include text to explain it. For example, let's say that we noticed that the first date in which the relative temperature exceeded 1 degree Celsius was October 6th, 2015. We'd like to point this out in the plot. Here again is the code that generates the plot, using the function that we implemented previously. Next, we call a method of the Axes object called `annotate`. At the very least, this function takes the annotation text as input, in this case, the string ">1 degree", and the `xy` coordinate that we would like to annotate. Here, the value to annotate has the x position of the TimeStamp of that date. We use the pandas `time-stamp` object to define that. The y position of the data is 1, which is the 1 degree Celsius value at that date. But this doesn't look great. The text appears on top of the axis tick labels. Maybe we can move it somewhere else?

Positioning the text

The `annotate` method takes an optional `xy` text argument that selects the xy position of the text. After some experimentation, we've found that an x value of October 6th, 2008 and a y value of negative 0.2 degrees is a good place to put the text. The problem now is that there is no way to see which data point is the one that is being annotated. Let's add an arrow that connects the text to the data.

Adding arrows to annotation

To connect between the annotation text and the annotated data, we can add an arrow. The key-word argument to do this is called `arrowprops`, which stands for arrow properties. This key-word argument takes as input a dictionary that defines the properties of the arrow that we would like to use.

Customizing arrow properties

We can also customize the appearance of the arrow. For example, here we set the style of the arrow to be a thin line with a wide head. That's what the string with a dash and a smaller than sign means. We also set the color to gray. This is a bit more subtle.

Customizing annotations

There are many more options for customizing the arrow properties and other properties of the annotation, which you can read about in the Matplotlib documentation here.

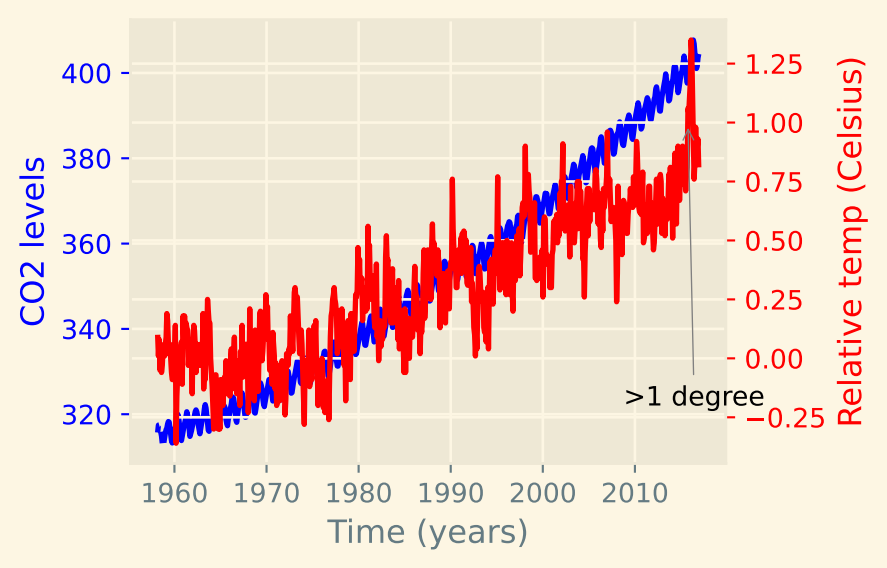
Exercise 2.3

---

1. Importing the dataset, `climate_change.csv`.
2. Annotating a plot of time-series data. Use the `plot_timeseries` function to plot `CO2` levels against `time`. Set `xlabel` to `"Time (years)"` ylabel to `"CO2 levels"` and `color` to `'blue'`.
3. Create `ax2`, as a twin of the first Axes.
4. In `ax2`, plot temperature against time, setting the `color` and `ylabel` to `"Relative temp (Celsius)"` and `color` to `'red'`.
5. Annotate the data using the `ax2.annotate` method. Place the text `">1 degree"` in `x = pd.Timestamp('2008-10-06'), y=-0.2` pointing with a `gray` thin arrow to `x = pd.Timestamp('2015-10-06'), y = 1`.

```
# Importing the course datasets
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date

# Annotating a plot of time-series data
fig, ax = plt.subplots()

# Use the plot_timeseries function to plot CO2 levels against time. Set xlabel to "Time (years)"
plot_timeseries(ax, climate_change.index, climate_change['co2'], 'blue', "Time (years)", "CO2 lev

# Create ax2, as a twin of the first Axes.
ax2 = ax.twinx()

# In ax2, plot temperature against time, setting the color ylabel to "Relative temp (Celsius)" an
plot_timeseries(ax2, climate_change.index, climate_change['relative_temp'], 'red', "Time (years)"
```

```
# Annotate the data using the ax2.annotate method. Place the text ">1 degree" in x=pd.Timestamp('
ax2.annotate(">1 degree", xy = (pd.Timestamp('2015-10-06'), 1), xytext = (pd.Timestamp('2008-10-0

plt.show()
```



Chapter 3.1: Quantitative comparisons: bar-charts

In the previous chapter, Section , you saw how you can turn data into visual descriptions. In this chapter, we will focus on quantitative comparisons between parts of the data.

Olympic medals

Let's look at a dataset that contains information about the number of medals won by a few countries in the 2016 Olympic Games. The data is not very large. Here is all of it. Although you can see all of it in front of you, it's not that easy to make comparisons between different countries and see which countries won which medals.

Olympic medals: visualizing the data

Let's start by reading the data in from a file. We tell pandas to create a DataFrame from a file that contains the data and to use the first column, which contains the country names, as the index for the DataFrame. Next, we can visualize the data about gold medals. We create a Figure and an Axes object and call the Axes bar method to create a bar chart. This chart shows a bar for every row in the `"Gold"` column of the DataFrame, where the height of the bar represents the number in that row. The labels of the x-axis ticks correspond to the index of the DataFrame, which contains the names of the different countries in the data table. Unfortunately, these names are rather long, so they overlap with each other. Let's fix that first.

Interlude: rotate the tick labels

To fix these labels, we can rotate them by 90 degrees. This is done by using the `set_xticklabels` method of the Axes. We also take the opportunity to add a label on the y-axis, telling us that the height corresponds to the number of medals. This looks good. Visualizing the data in this way shows us which countries got a high or low number of gold medals, but also allows us to see the differences between countries, based on the difference in heights between the bars.

Olympic medals: visualizing the other medals

Next, we would like to add the data about the other medals: Silver and Bronze. To add this information into the same plot, we'll create a stacked bar chart. This means that each new data will be stacked on top of the previous data. It starts the same way as before. Next, we add another call to the bar method to add the data from the `"Silver"` column of the DataFrame. We add the `bottom` key-word argument to tell Matplotlib that the bottom of this column's data should be at the height of the previous column's data. We add the x-axis tick labels, rotating them by 90 degrees, set the y-axis labels, and call `plt.show`.

Olympic medals: visualizing all three

Similarly, we can add in the number of Bronze medals, setting the bottom of this bar to be the sum of the number of gold medals and the number of silver medals.

Stacked bar chart

This is what the full stacked bar chart looks like.

Adding a legend

To make this figure easier to read and understand, we would also like to label which color corresponds to which medal. To do this we need to add two things.

The first is to add the `label` key-word argument to each call of the bar method with the label for the bars plotted in this call. The second is to add a call to the Axes `legend` method before calling `show`. This adds in a legend that tells us which color stands for which medal.

Exercise 3.1

Bar charts visualize data that is organized according to categories as a series of bars, where the height of each bar represents the values of the data in this category.

1. Plot a bar-chart of gold medals as a function of country.
2. Set the x-axis tick labels to the country names and rotate the x-axis tick labels by 90 degrees by using the rotation key-word argument.
3. Set the x-axis tick labels to the country names and rotate the x-axis tick labels by 90 degrees by using the rotation key-word argument.
4. Set the y-axis label and show plot.

```
# Importing the course datasets
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)
print(medals.head(5))

# Bar chart
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 and rotate the x-axis tick labels by 90 degrees
ax.set_xticklabels(medals.index, rotation = 90)
```

```
# Set the y-axis label
ax.set_ylabel("Number of medals")

plt.show()
```

```
               Bronze  Gold  Silver
United States      67   137      52
Germany            67    47      43
Great Britain      26    64      55
Russia             35    50      28
China              35    44      30
```

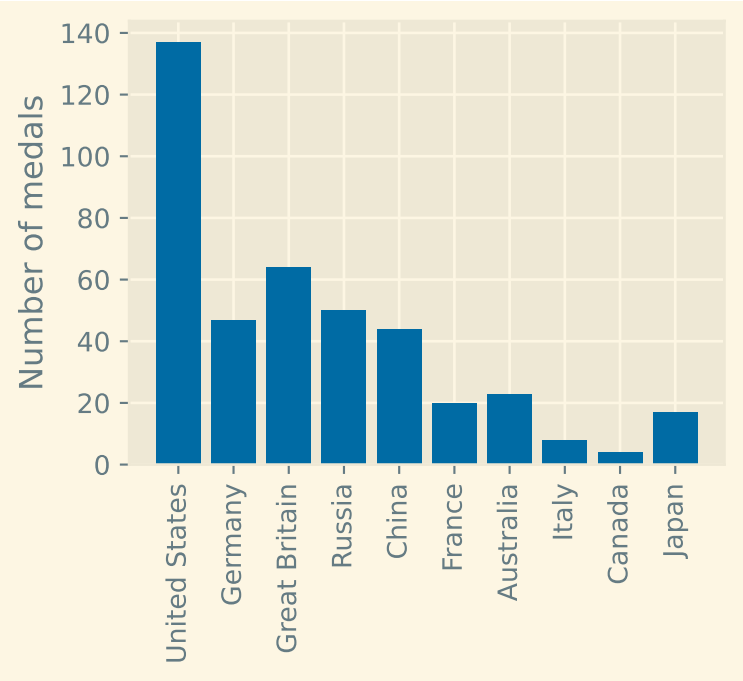


Figure 2: Number of Gold Medals Won Per Country in the Olympics.

Exercise 3.1

A stacked bar chart contains bars, where the height of each bar represents values. In addition, stacked on top of the first variable may be another variable. The additional height of this bar represents the value of this variable. And you can add more bars on top of that.

1. Call the `ax.bar` method to add the `"Gold"` medals. Call it with the label set to `"Gold"`.

2. Call the `ax.bar` method to stack `"Silver"` bars on top of that, using the `bottom` key-word argument so the bottom of the bars will be on top of the gold medal bars, and label to add the label `"Silver"`.
3. Use `ax.bar` to add `"Bronze"` bars on top of that, using the `bottom` key-word and label it as `"Bronze"`.

```
# Importing the course dataset
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)
print(medals.head(5))

# Bar chart
fig, ax = plt.subplots()

# 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")

# Set the x-axis tick labels to the country names and rotate the x-axis tick labels by 90 degrees
ax.set_xticklabels(medals.index, rotation = 90)

# Set the y-axis label
ax.set_ylabel("Number of medals")

# Display the legend
ax.legend()

plt.show()
```

```
               Bronze  Gold  Silver
United States      67   137      52
Germany            67    47      43
Great Britain      26    64      55
Russia             35    50      28
China              35    44      30
```

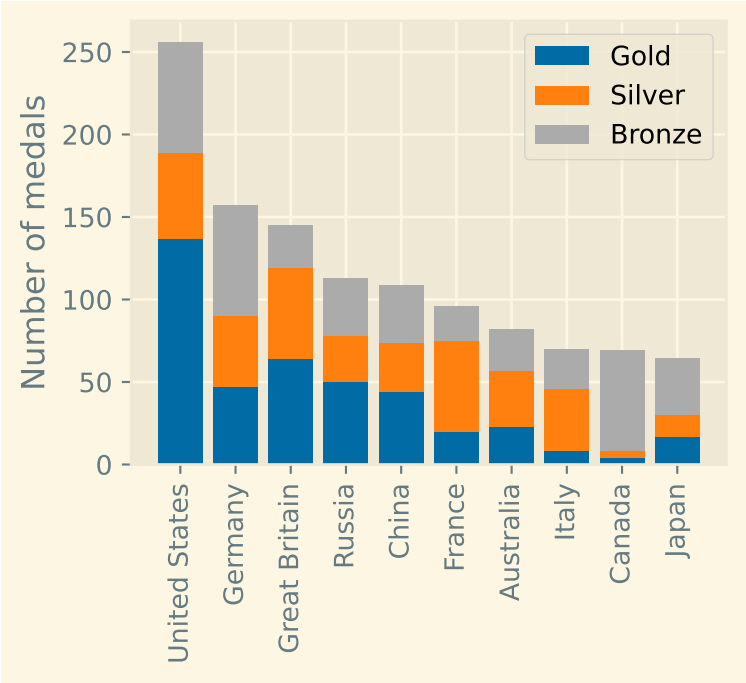


Figure 3: Number of Olympic Medals Per Country.

Chapter 3.2: Quantitative comparisons: histograms

Bar-charts show us the value of a variable in different conditions. Now, we're going to look at histograms. This visualization is useful because it shows us the entire distribution of values within a variable.

Histograms

Let's look at another example. In this case, we are looking at data about the athletes who participated in the 2016 Olympic Games. We've extracted two DataFrames from this data: all of the medal winners in men's gymnastics and all of the medal winners in men's rowing. Here are the five first rows in the men's rowing DataFrame. You can see that the data contains different kinds of information: what kinds of medals each competitor won, and also the competitor's height and weight.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course dataset
summer_2016 = pd.read_csv('datasets/summer2016.csv')
print(summer_2016.head(5))
```

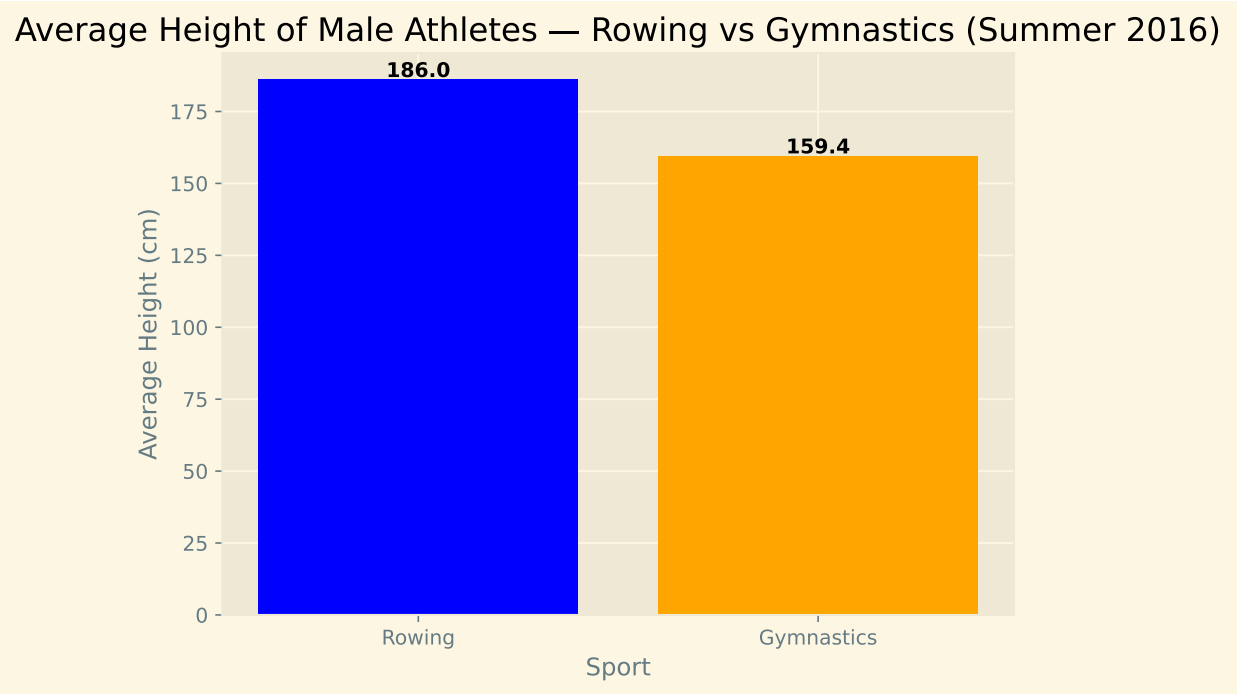
```
   Unnamed: 0   ID                          Name Sex   Age  Height  Weight  \
0         158   62             Giovanni Abagnale   M  21.0   198.0    90.0
1         161   65             Patimat Abakarova   F  21.0   165.0    49.0
2         175   73                     Luc Abalo   M  31.0   182.0    86.0
3         450  250          Saeid Morad Abdevali   M  26.0   170.0    80.0
4         794  455  Denis Mikhaylovich Ablyazin   M  24.0   161.0    62.0

         Team  NOC        Games  Year  Season            City       Sport  \
0       Italy  ITA  2016 Summer  2016  Summer  Rio de Janeiro      Rowing
1  Azerbaijan  AZE  2016 Summer  2016  Summer  Rio de Janeiro   Taekwondo
2      France  FRA  2016 Summer  2016  Summer  Rio de Janeiro    Handball
3        Iran  IRI  2016 Summer  2016  Summer  Rio de Janeiro   Wrestling
4      Russia  RUS  2016 Summer  2016  Summer  Rio de Janeiro  Gymnastics

                                    Event   Medal
0                Rowing Men's Coxless Pairs  Bronze
1               Taekwondo Women's Flyweight  Bronze
2                   Handball Men's Handball  Silver
3  Wrestling Men's Middleweight, Greco-Roman  Bronze
4           Gymnastics Men's Team All-Around  Silver
```

A bar chart again

Let's start by seeing what a comparison of heights would look like with a bar chart. After creating the Figure and Axes objects, we add to them a bar with the mean of the rowing "Height" column. Then, we add a bar with the mean of the gymnastics "Height" column. We set the y-axis label and show the figure, which gives us a sense for the difference between the groups.

```
import matplotlib.pyplot as plt

# Filter the data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]
```

```
# Compute the mean height for each sport
mean_rowing_height = rowing["Height"].mean()
mean_gymnastics_height = gymnastics["Height"].mean()

# Prepare data for the bar chart
sports = ["Rowing", "Gymnastics"]
mean_heights = [mean_rowing_height, mean_gymnastics_height]

# Create the figure and axis
fig, ax = plt.subplots(figsize=(7, 5))

# Plot the bar chart
ax.bar(sports, mean_heights, color=["blue", "orange"])

# Add labels and title
ax.set_xlabel("Sport")
ax.set_ylabel("Average Height (cm)")
ax.set_title("Average Height of Male Athletes - Rowing vs Gymnastics (Summer 2016)")

# Display the mean values on top of each bar
for i, v in enumerate(mean_heights):
    ax.text(i, v + 1, f"{v:.1f}", ha='center', fontweight='bold')

# Show the chart
plt.show()
```

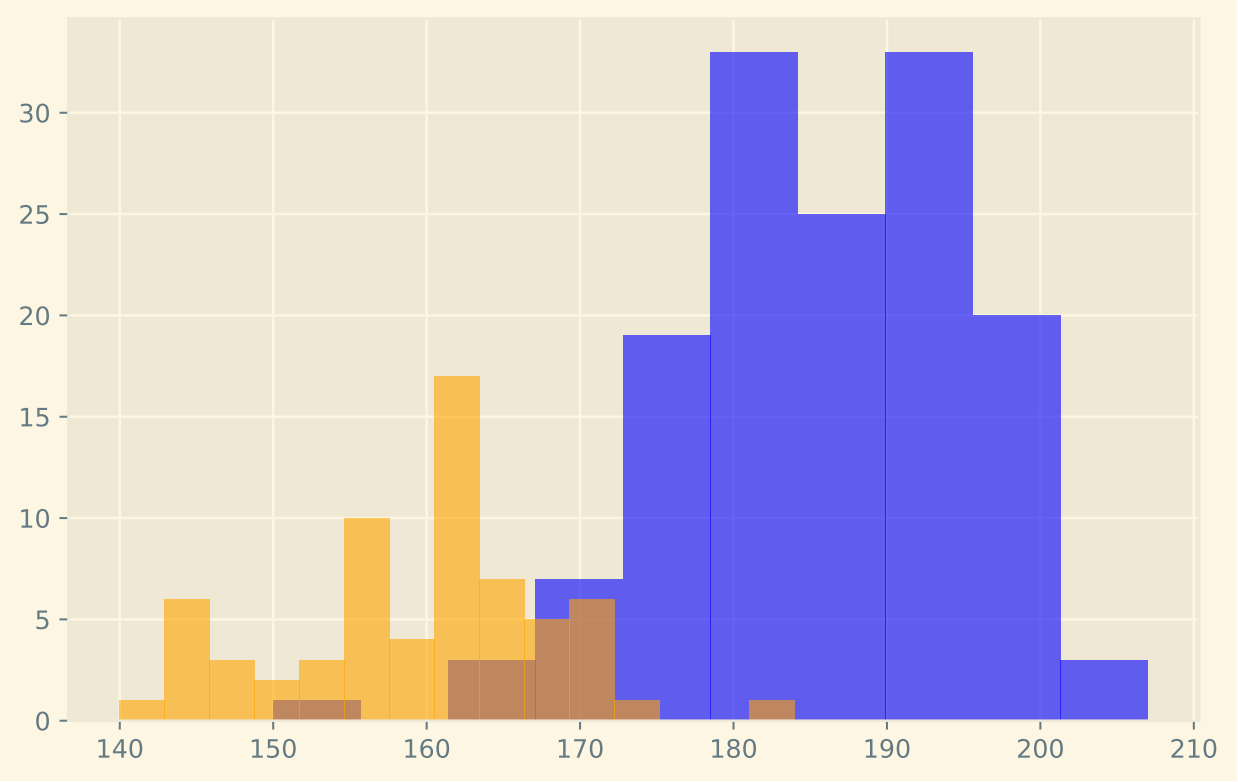


Introducing histograms

But a histogram would instead show the full distribution of values within each variable. Let's see that. We start again by initializing a Figure and Axes. We then call the Axes hist method with the entire `"Height"` column of the men's rowing DataFrame. We repeat this with the men's gymnastics DataFrame. In the histogram shown, the x-axis is the values within the variable and the height of the bars represents the number of observations within a particular bin of values. For example, there are 12 gymnasts with heights between 164 and 167 centimeters, so the highest bar in the orange histogram is 12 units high. Similarly, there are 20 rowers with heights between 188 and 192 centimeters, and the highest bar in the `blue` histogram is 20 units high.

```
# Filter data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]

# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 5))

# Plot histograms for comparison
ax.hist(rowing["Height"], alpha=0.6, label="Rowing", color="blue")
ax.hist(gymnastics["Height"], bins=15, alpha=0.6, label="Gymnastics", color="orange")
```

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

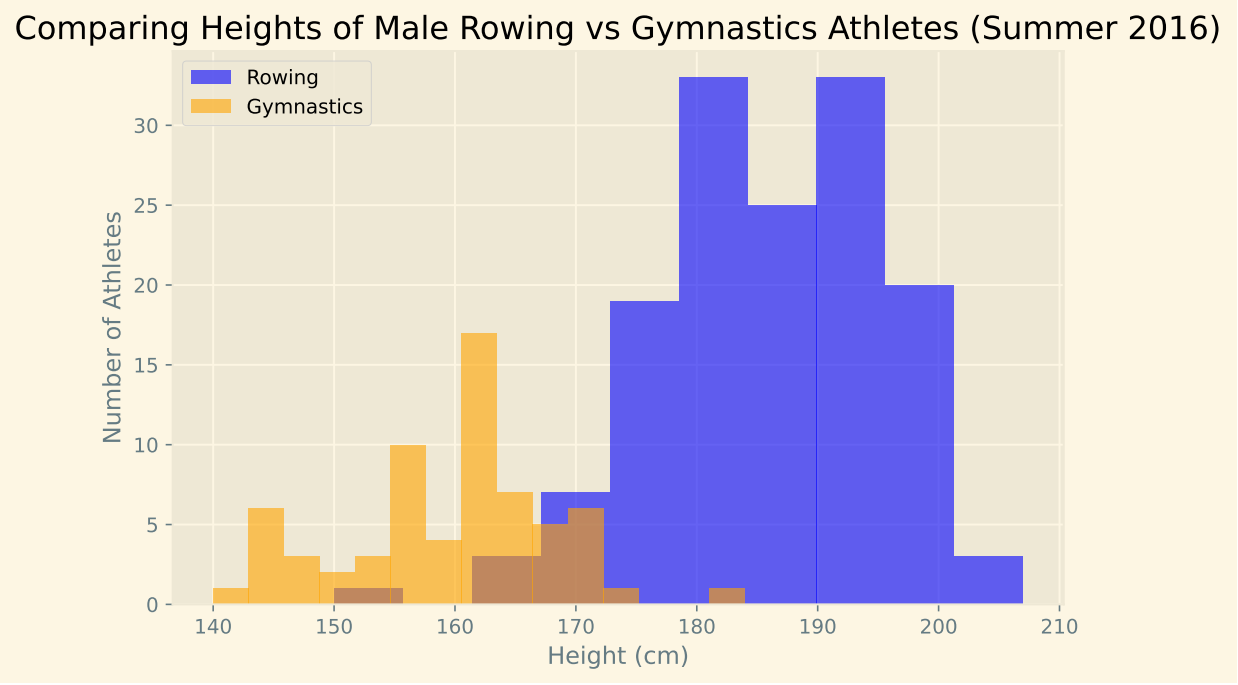


Labels are needed

Because the x-axis label no longer provides information about which color represents which variable, labels are really needed in histograms. As before, we can label a variable by calling the `hist` method with the `label` key-word argument and then calling the `legend` method before we call `plt.show`, so that a legend appears in the figure.

```
# Filter data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]

# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 5))

# Plot histograms for comparison
ax.hist(rowing["Height"], alpha=0.6, label="Rowing", color="blue")
ax.hist(gymnastics["Height"], bins=15, alpha=0.6, label="Gymnastics", color="orange")
```

```
# Add labels and title
ax.set_xlabel("Height (cm)")
ax.set_ylabel("Number of Athletes")
ax.set_title("Comparing Heights of Male Rowing vs Gymnastics Athletes (Summer 2016)")

# Add legend for clarity
ax.legend()

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

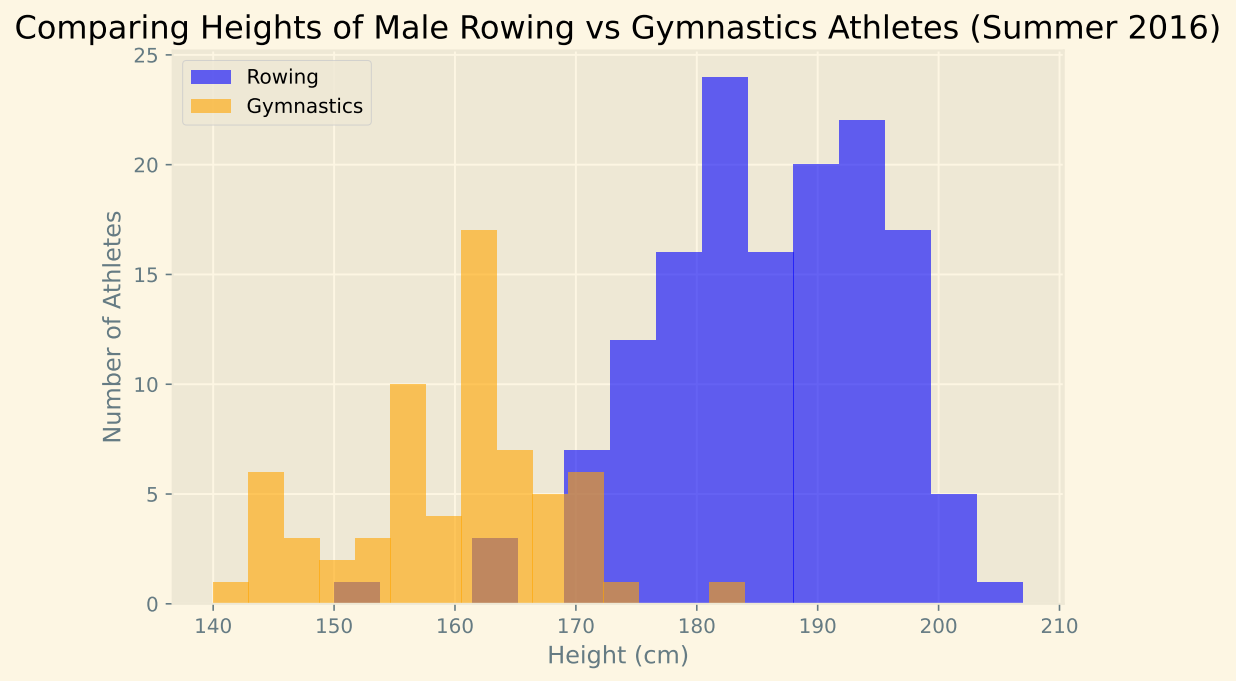


Customizing histograms: setting the number of bins

You might be wondering how `Matplotlib` decides how to divide the data up into the different bars. Per default, the number of bars or bins in a histogram is 10, but we can customize that. If we provide an integer number to the `bins` key-word argument, the histogram will have that number of bins.

```
# Filter data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]
```

```
# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 5))

# Plot histograms for comparison
ax.hist(rowing["Height"], bins = 15, alpha=0.6, label="Rowing", color="blue")
ax.hist(gymnastics["Height"], bins=15, alpha=0.6, label="Gymnastics", color="orange")

# Add labels and title
ax.set_xlabel("Height (cm)")
ax.set_ylabel("Number of Athletes")
ax.set_title("Comparing Heights of Male Rowing vs Gymnastics Athletes (Summer 2016)")

# Add legend for clarity
ax.legend()

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

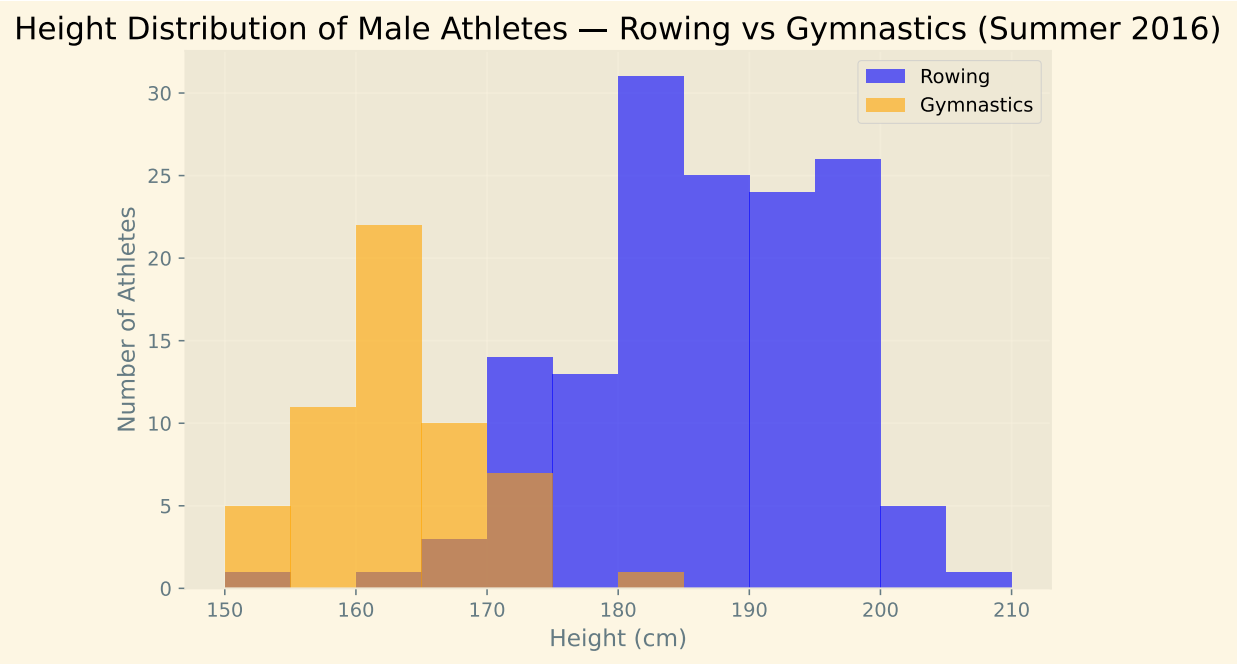


Customizing histograms: setting bin boundaries

If we instead provide a sequence of values, these numbers will be set to be the boundaries between the bins, as shown here. There is one last thing to customize. Looking at this figure, you might

wonder whether there are any rowing medalists with a height of less than 180 centimeters. This is hard to tell because the bars for the gymnastics histogram are occluding this information.

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

# Filter data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]

# Define bin boundaries (e.g., from 150 cm to 210 cm in steps of 5)
bins = np.arange(150, 215, 5)

# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 5))

# Plot histograms for comparison
ax.hist(rowing["Height"], bins=bins, alpha=0.6, label="Rowing", color="blue")
ax.hist(gymnastics["Height"], bins=bins, alpha=0.6, label="Gymnastics", color="orange")

# Add labels and title
ax.set_xlabel("Height (cm)")
ax.set_ylabel("Number of Athletes")
ax.set_title("Height Distribution of Male Athletes - Rowing vs Gymnastics (Summer 2016)")

# Add gridlines and legend
ax.grid(alpha=0.3)
ax.legend()

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

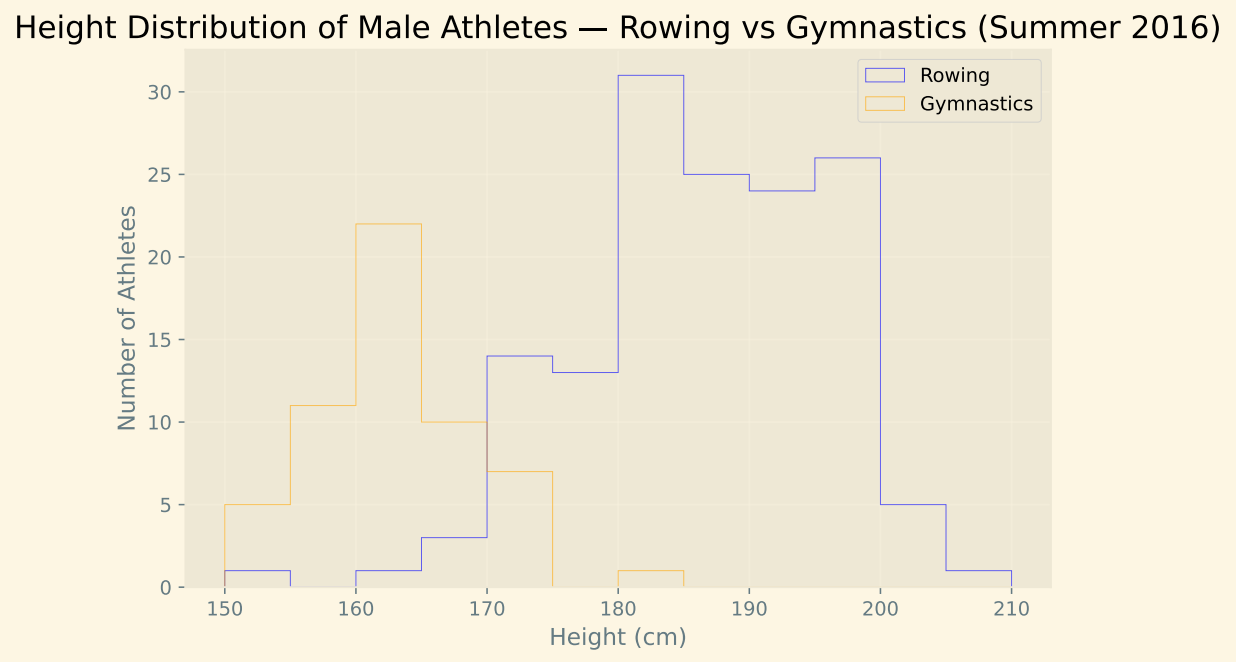


Customizing histograms: transparency

The occlusion can be eliminated by changing the type of histogram that is used. Instead of the "bar" type that is used per default, you can specify a histtype of "step", which displays the histogram as thin lines, instead of solid bars, exposing that yes: there are rowers with a height of less than 180 centimeters.

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

# Filter data for each sport
rowing = summer_2016[summer_2016["Sport"] == "Rowing"]
gymnastics = summer_2016[summer_2016["Sport"] == "Gymnastics"]

# Define bin boundaries (e.g., from 150 cm to 210 cm in steps of 5)
bins = np.arange(150, 215, 5)

# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 5))

# Plot histograms for comparison
ax.hist(rowing["Height"], bins=bins, alpha=0.6, label="Rowing", histtype = 'step', color="blue")
ax.hist(gymnastics["Height"], bins=bins, alpha=0.6, label="Gymnastics", histtype = 'step', color=
```

```
# Add labels and title
ax.set_xlabel("Height (cm)")
ax.set_ylabel("Number of Athletes")
ax.set_title("Height Distribution of Male Athletes - Rowing vs Gymnastics (Summer 2016)")

# Add gridlines and legend
ax.grid(alpha=0.3)
ax.legend()

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



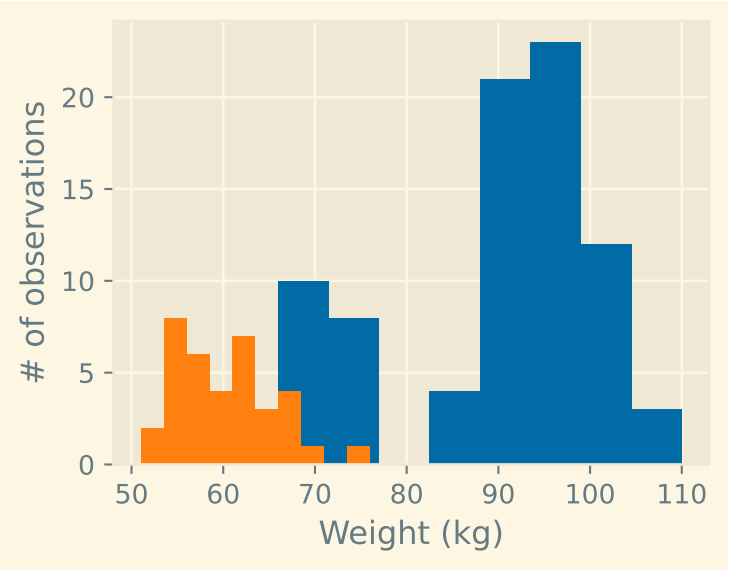
Exercise 3.2.1

Creating histograms

Histograms show the full distribution of a variable. In this exercise, we will display the distribution of weights of medalists in gymnastics and in rowing in the 2016 Olympic games for a comparison between them.

1. Use the `ax.hist` method to add a histogram of the `"Weight"` column from the `mens_rowing` DataFrame.
2. Use `ax.hist` to add a histogram of `"Weight"` for the `mens_gymnastics` DataFrame.
3. Set the `x-axis` label to `"Weight (kg)"` and the `y-axis` label to `"# of observations"`.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

rowing = summer_2016["Sport"] == "Rowing"
men = summer_2016["Sex"] == "M"
mens_rowing = summer_2016[rowing & men]

gymnastics = summer_2016["Sport"] == "Gymnastics"
men = summer_2016["Sex"] == "M"
mens_gymnastics = summer_2016[gymnastics & men]

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()
```



Exercise 3.2.2

"Step" histogram

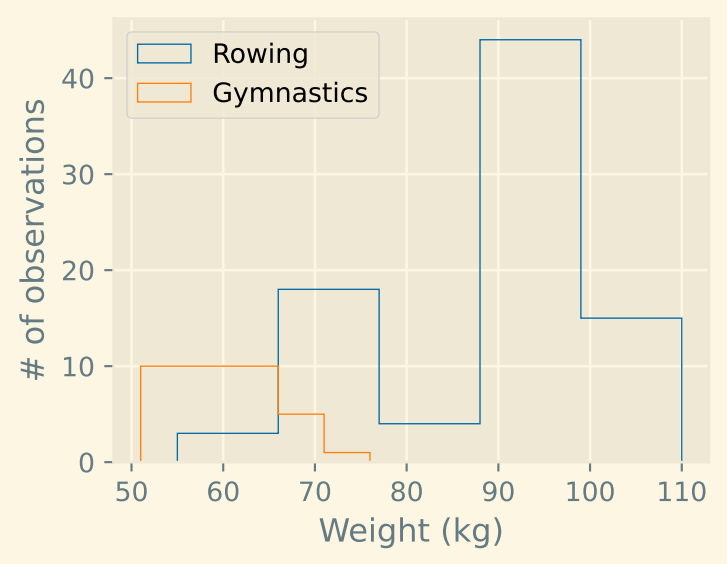
Histograms allow us to see the distributions of the data in different groups in our data. In this exercise, you will select groups from the Summer 2016 Olympic Games medalist dataset to compare the height of medalist athletes in two different sports.

In this exercise, you will visualize and label the histograms of two sports: `"Gymnastics"` and `"Rowing"` and see the marked difference between medalists in these two sports.

1. Use the `hist` method to display a histogram of the `"Weight"` column from the `mens_rowing` DataFrame, label this as `"Rowing"`.
2. Use `hist` to display a histogram of the `"Weight"` column from the `mens_gymnastics` DataFrame, and label this as `"Gymnastics"`.
3. For both histograms, use the `histtype` argument to visualize the data using the `'step'` type and set the number of `bins` to use to 5.
4. Add a `legend` to the figure, before it is displayed.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

rowing = summer_2016["Sport"] == "Rowing"
men = summer_2016["Sex"] == "M"
mens_rowing = summer_2016[rowing & men]

gymnastics = summer_2016["Sport"] == "Gymnastics"
men = summer_2016["Sex"] == "M"
mens_gymnastics = summer_2016[gymnastics & men]

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()
```



Chapter 4: Statistical plotting

In the previous lesson, Section , you saw how to create histograms that compare distributions of data. How can we make these comparisons more formal? Statistical plotting is a set of methods for using visualization to make comparisons. Here, we'll look at two of these techniques.

Adding error bars to bar charts

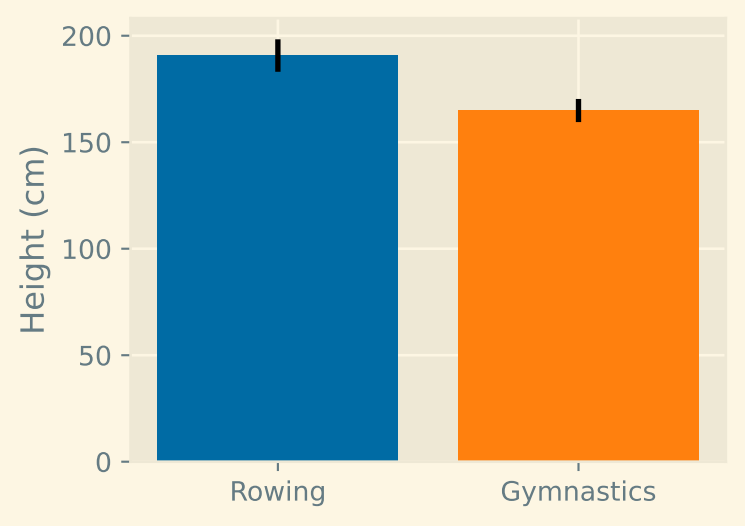
The first is the use of error bars in plots. These are additional markers on a plot or bar chart that tell us something about the distribution of the data. Histograms, that you have seen in the previous lesson, Section , show the entire distribution. Error bars instead summarize the distribution of the data in one number, such as the standard deviation of the values. To demonstrate this, we'll use the data about heights of medalists in the 2016 Olympic Games. There are at least two different ways to display error bars. Here, we add the error bar as an argument to a bar chart. Each call to the `ax.bar` method takes an `x` argument and a `y` argument. In this case, `y` is the mean of the `"Height"` column. The `yerr` key-word argument takes an additional number. In this case, the standard deviation of the `"Height"` column, and displays that as an additional vertical marker.

Error bars in a bar chart

Here is the plot. It is helpful because it summarizes the full distribution that you saw in the histograms in two numbers: the mean value, and the spread of values, quantified as the standard deviation.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course dataset
summer_2016 = pd.read_csv('datasets/summer2016.csv')

rowing = summer_2016["Sport"] == "Rowing"
men = summer_2016["Sex"] == "M"
mens_rowing = summer_2016[rowing & men]

gymnastics = summer_2016["Sport"] == "Gymnastics"
```

```
men = summer_2016["Sex"] == "M"
mens_gymnastics = summer_2016[gymnastics & men]

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()
```



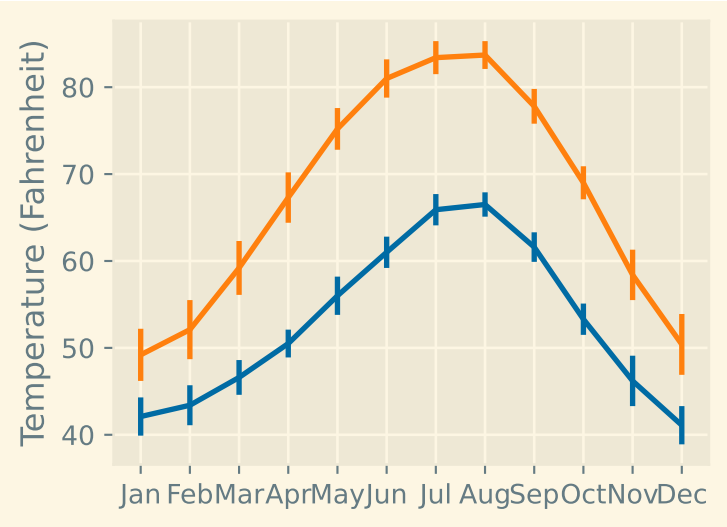
Adding error bars to plots

We can also add error bars to a line plot. For example, let's look at the weather data that we used in the first chapter of this course. To plot this data with error bars, we will use the Axes `errorbar` method. Like the plot method, this method takes a sequence of x values, in this case, the "MONTH" column, and a sequence of `y` values, in this case, the column with the normal average monthly temperatures. In addition, a `yerr` key-word argument can take the column in the data that contains the standard deviations of the average monthly temperatures.

Error bars in plots

Similar to before, this adds vertical markers to the plot, which look like this.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# SOLUTIONS
fig, ax = plt.subplots()

# Add Seattle temperature data in each month with error bars
ax.errorbar(seattle_weather['MONTH'], seattle_weather["MLY-TAVG-NORMAL"], yerr = seattle_weather[

# Add Austin temperature data in each month with error bars
ax.errorbar(austin_weather['MONTH'], austin_weather["MLY-TAVG-NORMAL"], yerr = austin_weather["ML

# Set the y-axis label
ax.set_ylabel("Temperature (Fahrenheit)")

plt.show()
```

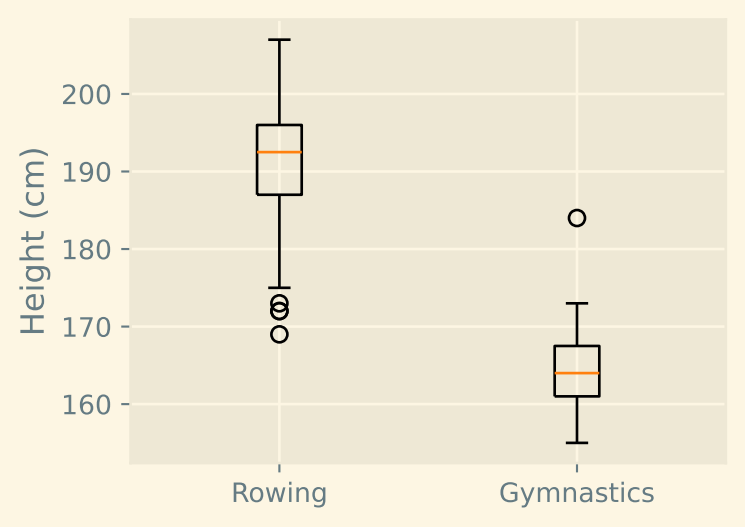


Adding boxplots

The second statistical visualization technique we will look at is the boxplot, a visualization technique invented by John Tukey, arguably the first data scientist. It is implemented as a method of the Axes object. We can call it with a sequence of sequences. In this case, we create a list with the men's rowing `"Height"` column and the men's gymnastics `"Height"` column and pass that list to the method. Because the box-plot doesn't know the labels on each of the variables, we add that separately, labeling the y-axis as well. Finally, we show the figure, which looks like this.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course dataset
summer_2016 = pd.read_csv('datasets/summer2016.csv')

rowing = summer_2016["Sport"] == "Rowing"
men = summer_2016["Sex"] == "M"
mens_rowing = summer_2016[rowing & men]

gymnastics = summer_2016["Sport"] == "Gymnastics"
men = summer_2016["Sex"] == "M"
mens_gymnastics = summer_2016[gymnastics & men]

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()
```



Interpreting boxplots

This kind of plot shows us several landmarks in each distribution. The red line indicates the median height. The edges of the box portion at the center indicate the inter-quartile range of the data, between the 25th and the 75th percentiles. The whiskers at the ends of the thin bars indicate one and a half times the size of the inter-quartile range beyond the 75th and 25th percentiles. This should encompass roughly 99 percent of the distribution if the data is Gaussian or normal. Points that appear beyond the whiskers are outliers. That means that they have values larger or smaller than what you would expect for 99 percent of the data in a Gaussian or normal distribution. For example, there are three unusually short rowers in this sample, and one unusually high gymnast.

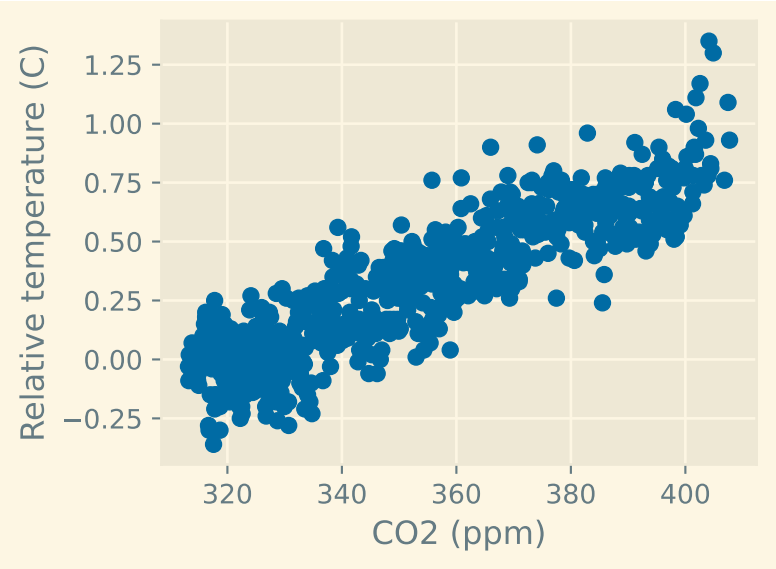
Chapter 5: Quantitative comparisons: scatter plots

Bar charts show us the values of one variable across different conditions, such as different countries. But what if you want to compare the values of different variables across observations? This is sometimes called a bi-variate comparison, because it involves the values of two different variables.

Introducing scatter plots

A standard visualization for bi-variate comparisons is a scatter plot. Let's look at an example. We'll use the climate change data that we have used previously. Recall that this dataset has a column with measurements of carbon dioxide and a column with concurrent measurements of the relative temperature. Because these measurements are paired up in this way, we can represent each measurement as a point, with the distance along the x-axis representing the measurement in one column and the height on the y-axis representing the measurement in the other column. To create this plot, we initialize a Figure and Axes objects and call the Axes scatter method. The first argument to this method will correspond to the distance along the x-axis and the second argument will correspond to the height along the y-axis. We also set the x-axis and y-axis labels, so that we can tell how to interpret the plot and call `plt.show` to display the figure.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date
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()
```



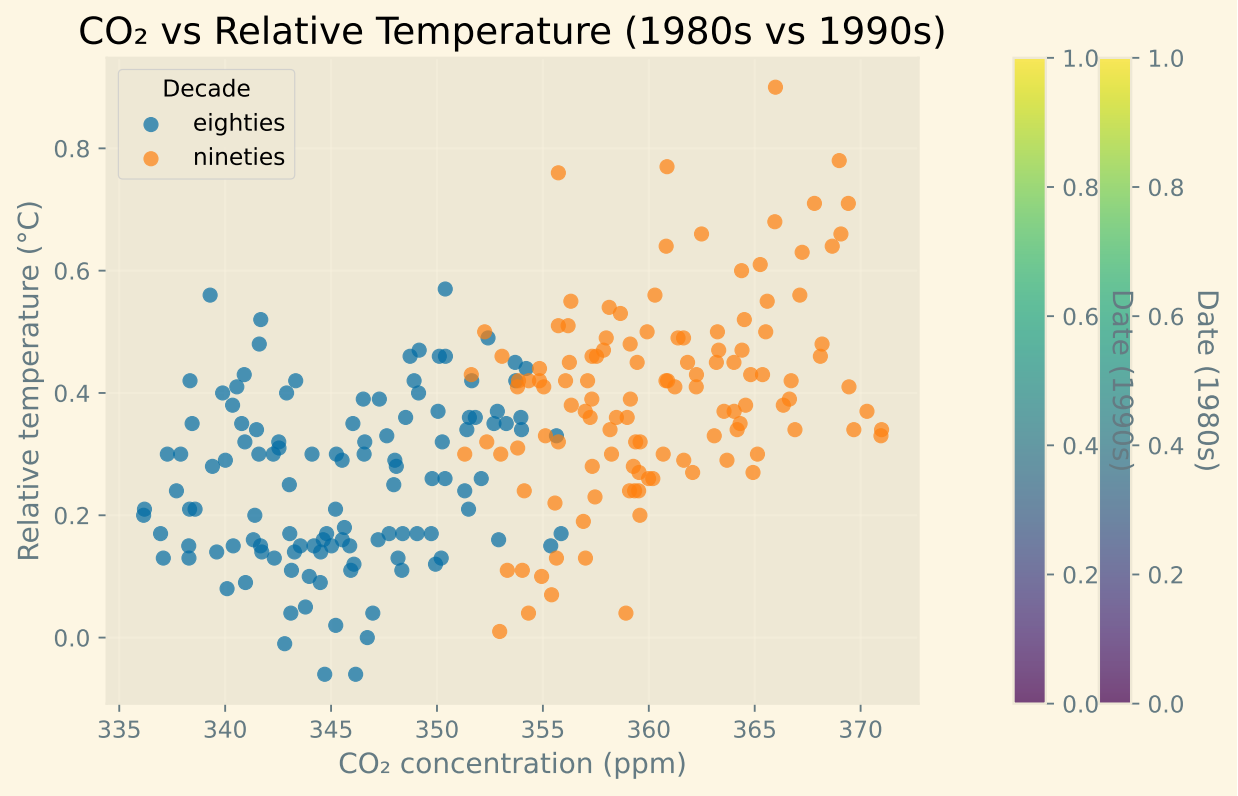
Customizing scatter plots

We can customize scatter plots in a manner that is similar to the customization that we introduced in other plots. For example, if we want to show two bivariate comparisons side-by-side, we want to make sure that they are visually distinct. Here, we are going to plot two scatter plots on the same axes. In one, we'll show the data from the nineteen-eighties and in the other, we'll show the data from the nineteen-nineties. We can select these parts of the data using the time-series indexing that you've seen before to create two DataFrames called eighties and nineties. Then, we add each one of these DataFrames into the Axes object. First, we add the data from the eighties. We add customization: we set the color of the points to be red and we label these data with the string `"eighties"`. Then, we add the data from the nineties. These points will be blue and we label them with the string `"nineties"`. We call the `legend` method to add a legend that will tell us which DataFrame is identified with which color, we add the axis labels and call `plt.show`.

```
#  Title: Comparing CO  Levels and Relative Temperature - 1980s vs 1990s

import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date

# Preview first few rows
print(climate_change.head())
```

```
# Create the figure and axes
fig, ax = plt.subplots(figsize=(8, 5))

# Subset data for the 1980s and 1990s
eighties = climate_change["1980-01-01":"1989-12-31"]
nineties = climate_change["1990-01-01":"1999-12-31"]

# Add scatter plot for the 1980s (red points)
sc1 = ax.scatter(eighties["co2"], eighties["relative_temp"], label="eighties", alpha=0.7)

# Add scatter plot for the 1990s (blue points)
sc2 = ax.scatter(nineties["co2"], nineties["relative_temp"], label="nineties", alpha=0.7)

# Customize the plot
ax.set_title("CO  vs Relative Temperature (1980s vs 1990s)")
ax.set_xlabel("CO  concentration (ppm)")
ax.set_ylabel("Relative temperature (°C)")
ax.legend(title="Decade")
ax.grid(alpha=0.3)

# Add colorbars to show the temporal encoding
cbar1 = plt.colorbar(sc1, ax=ax, orientation="vertical", fraction=0.046, pad=0.04)
cbar1.set_label("Date (1980s)", rotation=270, labelpad=15)
cbar2 = plt.colorbar(sc2, ax=ax, orientation="vertical", fraction=0.046, pad=0.10)
cbar2.set_label("Date (1990s)", rotation=270, labelpad=15)

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

```
              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
```



Encoding a comparison by color

This is what this figure looks like. You can see that the relationship between temperatures and carbon dioxide didn't change much during these years, but both levels of carbon dioxide and temperatures continued to rise in the nineties. Color can be used for a comparison, as we did here.

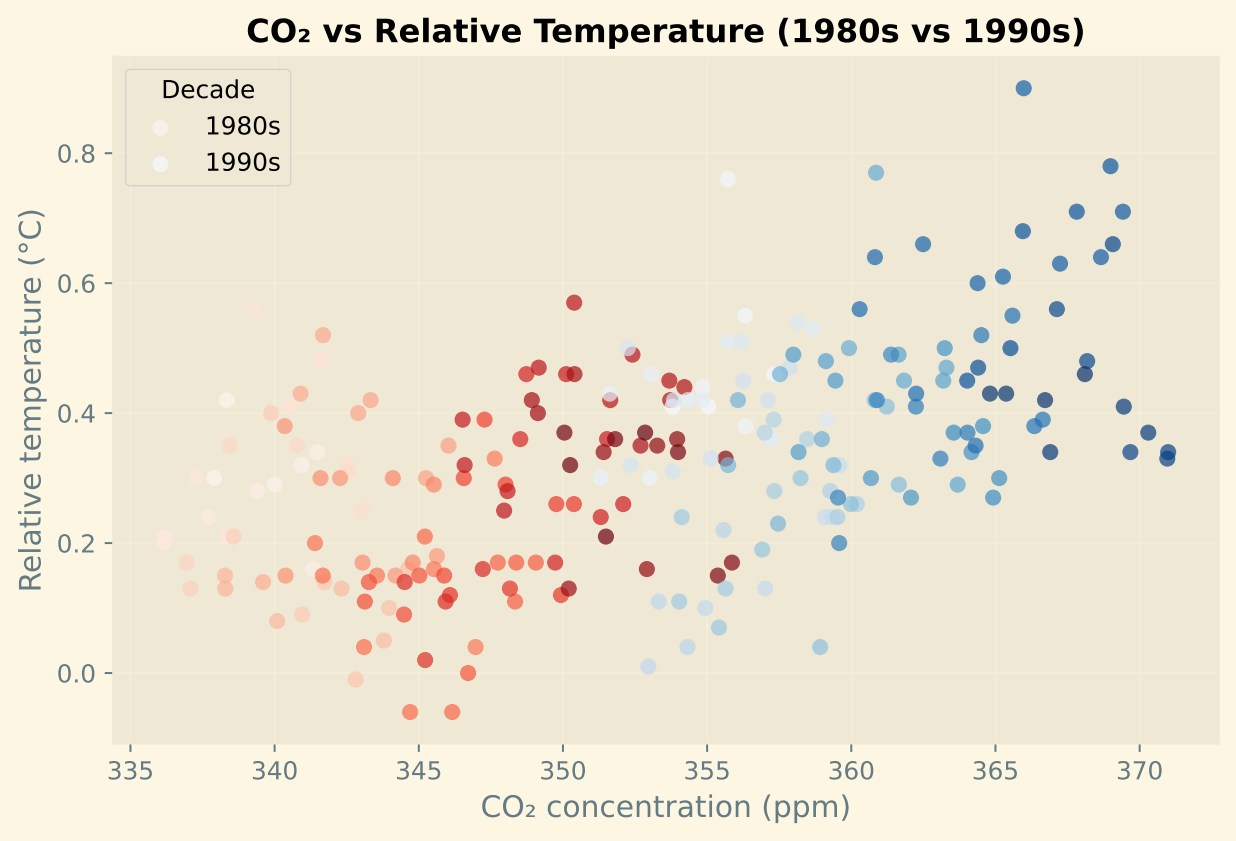
Encoding a third variable by color

But we can also use the color of the points to encode a third variable, providing additional information about the comparison. In the climate change data, we have a continuous variable denoting time stored in the DataFrame index. If we enter the index as input to the `c` key-word argument, this variable will get encoded as color. Note that this is not the color key-word argument that we used before, but is instead just the letter c. As before, we set the axis labels and call `plt.show`.

```
#   Title: Comparing CO  Levels and Relative Temperature - 1980s vs 1990s

import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
climate_change = pd.read_csv('datasets/climate_change.csv', parse_dates=["date"], index_col="date

# Preview first few rows
print(climate_change.head())

# Create the figure and axes
fig, ax = plt.subplots(figsize=(8, 5))

#   Convert DatetimeIndex to numeric (ordinal values)
eighties_numeric = eighties.index.map(pd.Timestamp.toordinal)
nineties_numeric = nineties.index.map(pd.Timestamp.toordinal)

# Add scatter plot for the 1980s, color encoded by time
ax.scatter(
    eighties["co2"],
    eighties["relative_temp"],
    c=eighties_numeric,
    cmap="Reds",
    label="1980s",
    alpha=0.7
)

# Add scatter plot for the 1990s, color encoded by time
ax.scatter(
    nineties["co2"],
    nineties["relative_temp"],
    c=nineties_numeric,
    cmap="Blues",
    label="1990s",
    alpha=0.7
)

# Customize the plot
ax.set_title("CO  vs Relative Temperature (1980s vs 1990s)", fontsize=13, weight="bold")
ax.set_xlabel("CO  concentration (ppm)")
ax.set_ylabel("Relative temperature (°C)")
ax.legend(title="Decade")
ax.grid(alpha=0.3)
```

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

```
                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
```



Encoding time in color

Now, time of the measurements is encoded in the brightness of the color applied to the points, with dark blue points early on and later points in bright yellow.

Chapter 6: Preparing your figures to share with others

This chapter will focus on creating visualizations that you can share with others and incorporate into automated data analysis pipelines. We'll start with customization of figure styles. Previously, you saw that you can change the appearance of individual elements of the figure, such as the line color, or marker shapes.

Changing plot style

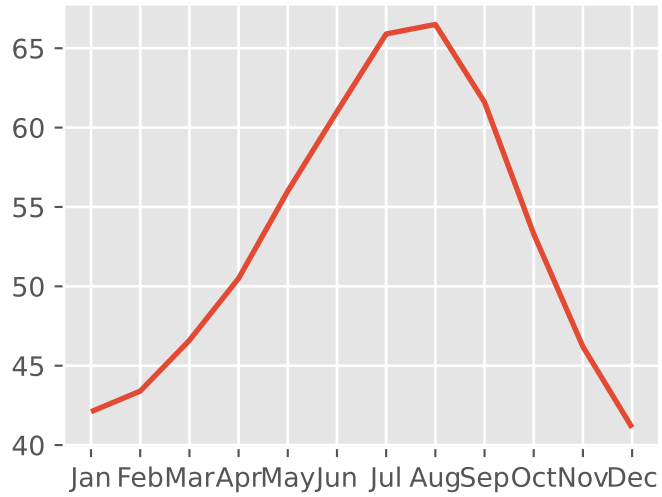
Here, we'll change the overall style of the figure. To see what that means, let's look at one of the figures we created in a previous lesson. This figure shows the average temperatures in Seattle and Austin as a function of the months of the year. This is what it looks like per default, Figure 1.

Choosing a style

If instead, we add this line of code before the plotting code, the figure style will look completely different. The style we chose here emulates the style of the R library ggplot. Maybe you know this library and this looks familiar to you, or you can learn about `ggplot` in a DataCamp course devoted to this library. Either way, you will notice that the setting of the style didn't change the appearance of just one element in the figure. Rather, it changed multiple elements: the colors are different, the fonts used in the text are different, and there is an added gray background that creates a faint white grid marking the x-axis and y-axis tick locations within the plot area. Furthermore, this style will now apply to all of the figures in this session, until you change it by choosing another style.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
```

```
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Use the "ggplot" style and create new Figure/Axes
plt.style.use("ggplot")
fig, ax = plt.subplots()
ax.plot(seattle_weather["MONTH"], seattle_weather["MLY-TAVG-NORMAL"])
plt.show()
```



Back to the default

For example, to go back to the default style, you would run `plt.style.use "default"`.

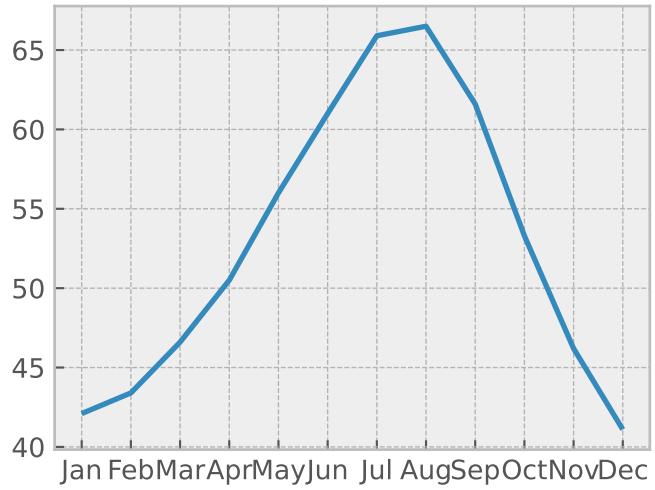
The available styles

Matplotlib contains implementations of several different styles and you can see the different styles available by going to this webpage, which contains a series of visualizations that have each been created using one of the available styles.

The "bmh" style

For example, this is what you get if you use "bmh" as the style.

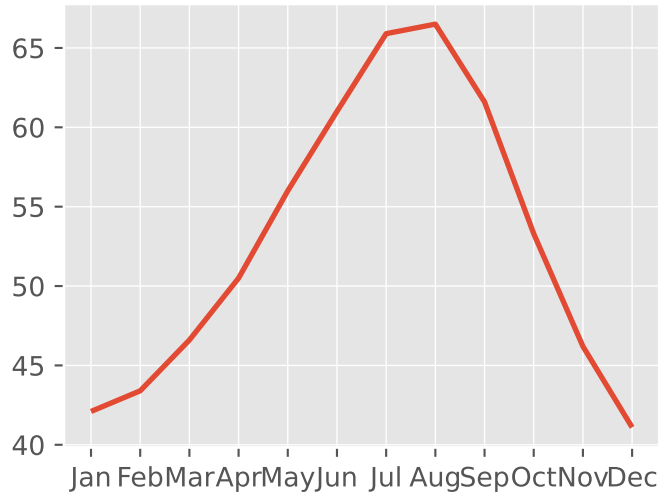
```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Use the "bmh" style and create new Figure/Axes
plt.style.use("bmh")
fig, ax = plt.subplots()
ax.plot(seattle_weather["MONTH"], seattle_weather["MLY-TAVG-NORMAL"])
plt.show()
```



Seaborn styles

This is what you get if you select `seaborn-colorblind`. In fact, if you visit the documentation web-page, you will see that there are several available styles that are named after the Seaborn software library. This is a software library for statistical visualization that is based on Matplotlib, and Matplotlib adopted back several of the styles developed there. You can learn more about Seaborn in other DataCamp courses.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn

# Importing the course datasets
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Use the "ggplot" style and create new Figure/Axes
plt.style.use("ggplot")
fig, ax = plt.subplots()
ax.plot(seattle_weather["MONTH"], seattle_weather["MLY-TAVG-NORMAL"])
plt.show()
```

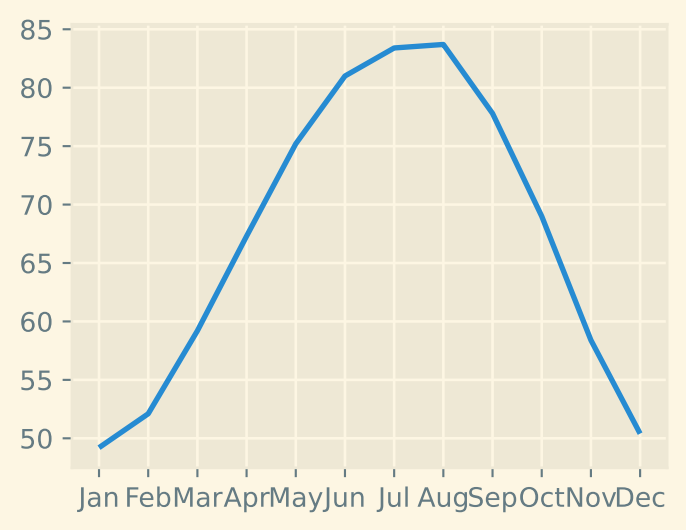



```
# Use the "seaborn-colorblind" style and create new Figure/Axes
plt.style.use("seaborn-colorblind")
fig, ax = plt.subplots()
ax.plot(austin_weather["MONTH"], austin_weather["MLY-TAVG-NORMAL"])
plt.show()
```

Using "Solarize_Light2" style

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")

# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Use the "Solarize_Light2" style and create new Figure/Axes
plt.style.use("Solarize_Light2")
fig, ax = plt.subplots()
```

```
ax.plot(austin_weather["MONTH"], austin_weather["MLY-TAVG-NORMAL"])
plt.show()
```

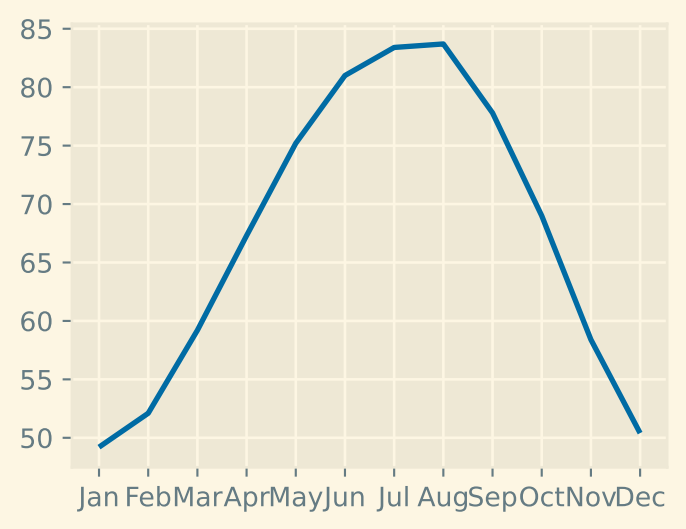


Guidelines for choosing plotting style

How would you choose which style to use? If your goal is primarily to communicate with others, think about how they might see it. Dark backgrounds are generally discouraged as they are less visible, so only use them if you have a good reason to do so. If colors are important, consider using a colorblind-friendly style, such as `seaborn-colorblind` or `tableau-colorblind10`. These are designed to retain color differences even when viewed by colorblind individuals. That might sound like a minor consideration, but approximately 1 out of 20 individuals is colorblind. Figures that are designed for use on websites have different considerations than figures in printed reports. For example, if someone is going to print out your figures, you might want to use less ink. That is, avoid colored backgrounds, like the background that appears in the "ggplot" style that we demonstrated before. If the printer used is likely to be black-and-white, consider using the `"grayscale"` style. This will retain the differences you see on your screen when printed out in a black-and-white printer.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
austin_weather = pd.read_csv("datasets/austin_weather.csv", index_col="DATE")
weather = pd.read_csv("datasets/seattle_weather.csv", index_col="DATE")
```

```
# Some pre-processing on the weather datasets, including adding a month column
seattle_weather = weather[weather["STATION"] == "USW00094290"]
month = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
seattle_weather["MONTH"] = month
austin_weather["MONTH"] = month

# Use the "tableau-colorblind10" style and create new Figure/Axes
plt.style.use("tableau-colorblind10")
fig, ax = plt.subplots()
ax.plot(austin_weather["MONTH"], austin_weather["MLY-TAVG-NORMAL"])
plt.show()
```



Chapter 7: Sharing your visualizations with others

After you have created your visualizations, you are ready to share them with your collaborators, colleagues, and with others. Here, we will show how you would go about doing final customizations to your figures, and saving them in an appropriate format.

A figure to share

Take for example this figure, Figure 2, that you previously created to display data about the number of gold medals that each of several countries won in the 2016 Olympic Games. When you previously ran this code, it displayed the figure on your screen when you called the `plt.show` method at the end of this code.

Saving the figure to file

Now, we replace the call to `plt.show` with a call to the Figure object's `savefig` method. We provide a file-name as input to the function. If we do this, the figure will no longer appear on our screen, but instead appear as a file on our file-system called `"gold-underscore-medals.png"`. In the interactive Python shell that we are using here, we can call the unix `ls` function, which gives us a listing of the files in the present working directory. In this case, only the file that we created is present. We can then share this file that now contains the visualization with others.

```
# Importing the course datasets
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)
print(medals.head(5))

# Bar chart
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 and rotate the x-axis tick labels by 90 degrees
ax.set_xticklabels(medals.index, rotation = 90)

# Set the y-axis label
ax.set_ylabel("Number of medals")

fig.savefig("gold-underscore-medals.png")
```

Different file formats

In the previous slide, we saved the figure as a PNG file. This file format provides lossless compression of your image. That means that the image will retain high quality, but will also take up relatively large amounts of diskspace or bandwidth. You can choose other file formats, depending on your need. For example, if the image is going to be part of a website, you might want to choose the `jpg` format used here, instead. This format uses lossy compression, and can be used to create figures that take up less diskspace and less bandwidth. You can control how small the resulting file will be, and the degree of loss of quality, by setting the `quality` key-word argument. This will be a number between 1 and 100, but you should avoid values above 95, because at that point the compression is no longer effective. Choosing the `svg` file-format will produce a vector graphics file where different elements can be edited in detail by advanced graphics software, such as Gimp or Adobe Illustrator. If you need to edit the figure after producing it, this might be a good choice.

```python
#| label: fig-gold-medals
#| fig-cap: "Number of Gold Medals by Country - Summer Olympics 2016"

import pandas as pd
import matplotlib.pyplot as plt

# Import the dataset
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)

# Display the first few rows
print(medals.head(5))

# Create a figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Plot a high-quality bar chart of gold medals by country
bars = ax.bar(
    medals.index,
    medals["Gold"],
    color="gold",
    edgecolor="black",
    linewidth=0.8
)

# Add chart enhancements
ax.set_title("  Gold Medals by Country - Summer Olympics 2016", fontsize=14, weight="bold")
ax.set_xlabel("Country", fontsize=12)
ax.set_ylabel("Number of Gold Medals", fontsize=12)
ax.set_xticks(range(len(medals.index)))
ax.set_xticklabels(medals.index, rotation=75, ha='right', fontsize=10)

# Add value labels on top of each bar for readability
for bar in bars:
    height = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width()/2,
        height + 0.3,
        f'{int(height)}',
        ha='center',
        va='bottom',
        fontsize=9,
        color='black'
    )

# Use a cleaner grid and layout
```

```
ax.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()

# Save the figure in high resolution
fig.savefig("gold_medals_by_country.png", dpi=300, bbox_inches="tight")
```

Resolution

Another key-word that you can use to control the quality of the images that you produce is the `dpi` key-word argument. This stands for dots per inch. The higher this number, the more densely the image will be rendered. If you set this number to 300, for example, this will render a fairly high-quality resolution of your image to file. Of course, the higher the resolution that you ask for, the larger the file-size will be.

```
#| label: fig-gold-medals
#| fig-cap: "Number of Gold Medals by Country - Summer Olympics 2016"

import pandas as pd
import matplotlib.pyplot as plt

# Import the dataset
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)

# Display the first few rows
print(medals.head(5))

# Create a figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Plot a high-quality bar chart of gold medals by country
bars = ax.bar(
    medals.index,
    medals["Gold"],
    color="gold",
    edgecolor="black",
    linewidth=0.8
)

# Add chart enhancements
ax.set_title("  Gold Medals by Country - Summer Olympics 2016", fontsize=14, weight="bold")
ax.set_xlabel("Country", fontsize=12)
ax.set_ylabel("Number of Gold Medals", fontsize=12)
```

```
ax.set_xticks(range(len(medals.index)))
ax.set_xticklabels(medals.index, rotation=75, ha='right', fontsize=10)

# Add value labels on top of each bar for readability
for bar in bars:
    height = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width()/2,
        height + 0.3,
        f'{int(height)}',
        ha='center',
        va='bottom',
        fontsize=9,
        color='black'
    )

# Use a cleaner grid and layout
ax.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()

# Save the figure in high resolution
fig.savefig("gold_medals_by_country.png", dpi=300, bbox_inches="tight") # PNG
fig.savefig("gold_medals_by_country.jpg", dpi=300, bbox_inches="tight")  # JPG
fig.savefig("gold_medals_by_country.svg", bbox_inches="tight")  # SVG (vector)
```

Size

Finally, another thing that you might want to control is the size of the figure. To control this, the Figure object also has a function called `set_size_inches`. This function takes a sequence of numbers. The first number sets the width of the figure on the page and the second number sets the height of the figure. So setting the size would also determine the aspect ratio of the figure. For example, you can set your figure to be wide and short or long and narrow, like here.

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

# Import dataset
medals = pd.read_csv('datasets/medals_by_country_2016.csv', index_col=0)

# Create the figure and axis
fig, ax = plt.subplots()
```

```python
# --- Control figure size directly ---
fig.set_size_inches(10, 6)  # width=10 inches, height=6 inches

# Plot a bar chart of gold medals
bars = ax.bar(
    medals.index,
    medals["Gold"],
    color="gold",
    edgecolor="black",
    linewidth=0.8
)

# Labeling and formatting
ax.set_title("  Gold Medals by Country - Summer Olympics 2016", fontsize=14, weight="bold")
ax.set_xlabel("Country", fontsize=12)
ax.set_ylabel("Number of Gold Medals", fontsize=12)
ax.set_xticks(range(len(medals.index)))
ax.set_xticklabels(medals.index, rotation=75, ha='right', fontsize=10)

# Add value labels above bars
for bar in bars:
    height = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        height + 0.3,
        f'{int(height)}',
        ha='center',
        va='bottom',
        fontsize=9,
        color='black'
    )

# Add grid and adjust layout
ax.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()

# --- Save in multiple high-quality formats ---
fig.savefig("gold_medals_by_country.png", dpi=300, bbox_inches="tight")
fig.savefig("gold_medals_by_country.jpg", dpi=300, bbox_inches="tight")
fig.savefig("gold_medals_by_country.svg", bbox_inches="tight")

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

Key Takeaways

- Use `savefig()` to export figures directly to disk.
- Choose formats wisely: **PNG** for quality, **JPG** for web, **SVG** for editing.
- Control quality with `dpi`, and adjust layout with `bbox_inches='tight'`.
- Manage aspect ratios and size using `set_size_inches()`.
- Always preview saved images before publication to ensure clarity and readability.

Chapter 8: Automating figures from data

One of the strengths of Matplotlib is that, when programmed correctly, it can flexibly adapt to the inputs that are provided.

Why automate?

This means that you can write functions and programs that automatically adjust what they are doing based on the input data. Why would you want to automate figure creation based on the data? Automation makes it easier to do more. It also allows you to be faster. This is one of the major benefits of using a programming language like Python and software libraries such as Matplotlib, over tools that require you to interact with a graphical user interface every time you want to create a new figure. Inspecting the incoming data and changing the behavior of the program based on the data provides flexibility, as well as robustness. Finally, an automatic program that adjusts to the data provides reproducible behavior across different runs.

How many different kinds of data?

Let's see what that means for Matplotlib. Consider the data about Olympic medal winners that we've looked at before. Until now, we always looked at two different branches of sports and compared them to each other, but what if we get a new data file, and we don't know how many different sports branches are included in the data? For example, what if we had a data-frame with hundreds of rows and a `"Sport"` column that indicates which branch of sport each row belongs to.

Getting unique values of a column

A column in a pandas DataFrame is a pandas Series object, so we can get the list of different sports present in the data by calling the `unique` method of that column. This tells us that there are 34 different branches of sport here.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

# Extract the "Sport" column
sports_column = summer_2016['Sport']

# Find the unique values of the "Sport" column
sports = sports_column.unique()
N_sport = sports_column.nunique()

# Print out the unique sports values
print("The unique values of the Sport column:\n")
print(sports, "\n")
print("There are", N_sport , "different branches of sport here.")
```

```
The unique values of the Sport column:

['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']

There are 34 different branches of sport here.
```

Bar-chart of heights for all sports

Let's say that we would like to visualize the height of athletes in each one of the sports, with a standard deviation error bar. Given that we don't know in advance how many sports there are in the DataFrame, once we've extracted the unique values, we can loop over them. In each iteration through, we set a loop variable called sport to be equal to one of these unique values. We then create a smaller DataFrame, that we call `sport_df`, by selecting the rows in which the `"Sport"` column is equal to the `sport` selected in this iteration. We can call the `bar` method of the Axes we created for this plot. As before, it is called with the string that holds the name of the sport as the first argument, the `mean` method of the `"Height"` column is set to be the height of the bar and an `error bar` is set to be equal to the standard deviation of the values in the column. After iterating over all of the sports, we exit the loop. We can then set the y-label to indicate the meaning of the height of each bar and we can set the x-axis tick labels to be equal to the names of the sports. As we did with the country names in the stacked bar chart that you saw in a previous lesson, we rotate these labels 90 degrees, so that they don't run over each other.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

# Extract the "Sport" column
sports_column = summer_2016['Sport']

# Find the unique values of the "Sport" column
sports = sports_column.unique()

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[summer_2016["Sport"] == sport]
  # Add a bar for the "Height" mean with std y error bar
  ax.bar(sport, sport_df['Height'].mean(), yerr = sport_df["Height"].std())
```

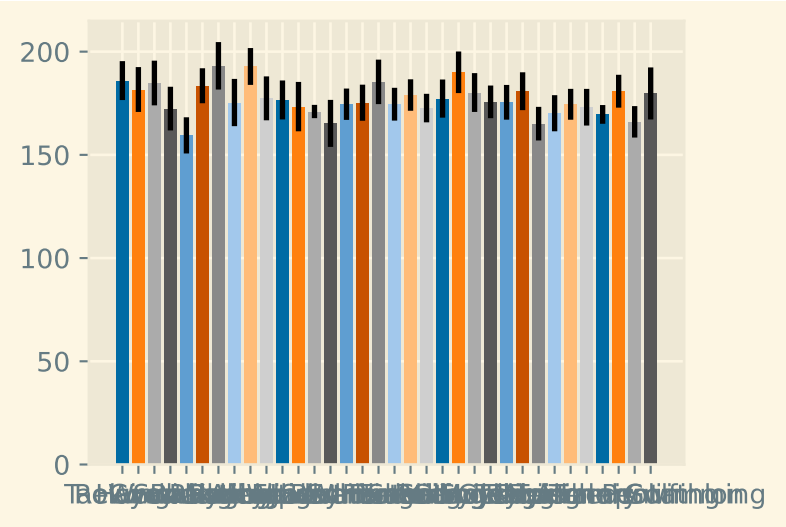
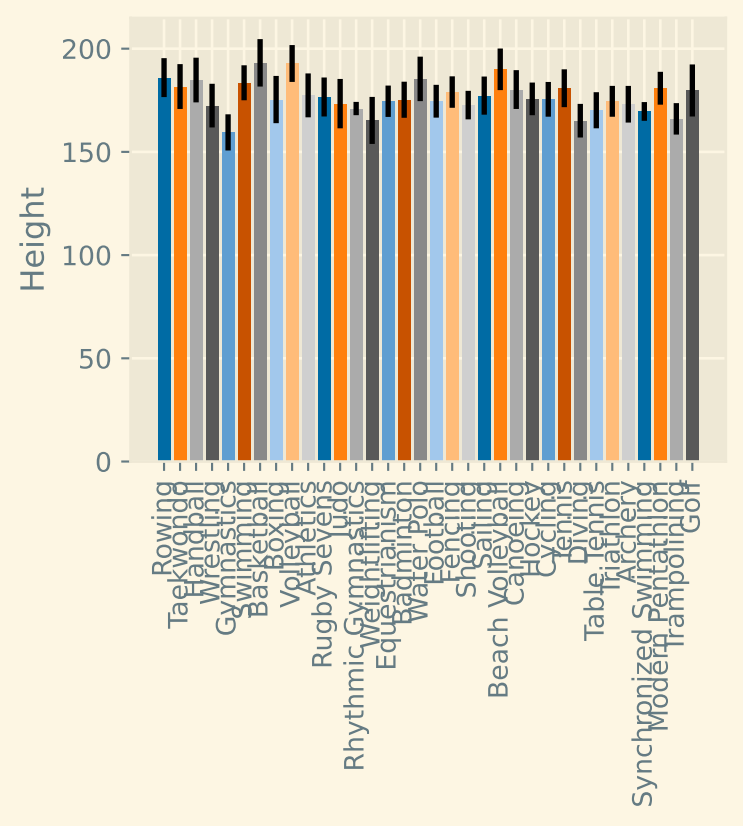


Figure derived automatically from the data

This is what this figure would look like. Importantly, at no point during the creation of this figure did we need to know how many different sports are recorded in the DataFrame. Our code would automatically add bars or reduce the number of bars, depending on the input data.

```
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

# Extract the "Sport" column
sports_column = summer_2016['Sport']

# Find the unique values of the "Sport" column
sports = sports_column.unique()

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[summer_2016["Sport"] == sport]
  # Add a bar for the "Height" mean with std y error bar
```

```
  ax.bar(sport, sport_df['Height'].mean(), yerr = sport_df["Height"].std())

ax.set_ylabel("Height")
ax.set_xticklabels(sports, rotation=90)

plt.show()
```



Exercise 8

INSTRUCTIONS

---

1. Create a variable called `sports_column` that holds the data from the `"Sport"` column of the DataFrame object.
2. Use the `unique` method of this variable to find all the unique different sports that are present in this data, and assign these values into a new variable called `sports`.

3. Print the `sports` variable to the console.

```
# SOLUTIONS
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

# Extract the "Sport" column
sports_column = summer_2016['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

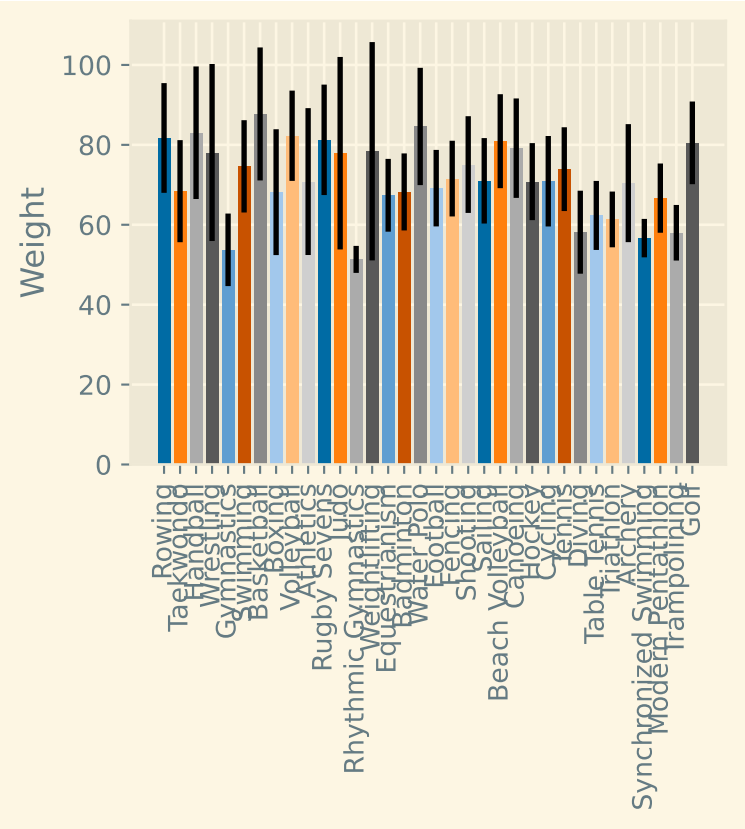
One of the main strengths of Matplotlib is that it can be automated to adapt to the data that it receives as input. For example, if you receive data that has an unknown number of categories, you can still create a bar plot that has bars for each category.

Instructions

1. Iterate over the values of `sports` setting `sport` as your loop variable.
2. In each iteration, extract the rows where the `"Sport"` column is equal to sport.
3. Add a bar to the provided ax object, labeled with the sport name, with the mean of the "Weight" column as its height, and the standard deviation as a y-axis error bar.

4. Show the plot.

```
# SOLUTIONS
# Importing the course packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course datasets
summer_2016 = pd.read_csv('datasets/summer2016.csv')

# Extract the "Sport" column
sports_column = summer_2016['Sport']

# Find the unique values of the "Sport" column
sports = sports_column.unique()

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[summer_2016["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)

# Show plot
plt.show()
```



References

- Introduction to Data Visualization with Matplotlib Course in Intermediate Python Course for Associate Data Scientist in Python Carrer Track in DataCamp by Ariel Rokem
- Python For Data Analysis 3E (Online) by Wes Mckinney Click here